Learn Go with Pocket-Sized Projects

Aliénor Latour
Donia Chaiehloudj
Pascal Bertrand

MANNING

Learn Go with Pocket-Sized Projects

Aliénor Latour
Donia Chaiehloudj
Pascal Bertrand

MEAP

MANNING

Learn Go with Pocket-Sized Projects

welcome

Thank you for purchasing the *Learn Go with Pocket-Sized Projects*! We hope you will have fun and make immediate use of your learnings.

This book is for developers who want to learn the language in a fun and interactive way, and be comfortable enough to use it professionally. Each chapter is an independent pocket-sized project. The book covers the specificities of the language, such as implicit interfaces and how they help in test design. Testing the code is included throughout the book. We want to help the reader become a good modern software developer while using the Go language.

This book also contains tutorials for command-line interfaces, and for both REST and gRPC microservices, showing how the language is great for cloud computing. It finishes with a project that uses TinyGo, the compiler for embedded systems.

Each pocket-sized project is written in a reasonable number of lines. Our goal is to provide various exercises so any developer who wants to begin with Go or to explore the specificities of the language can follow the steps described in each chapter. This is not a book to learn development from scratch and the chapters are graded.

We encourage you to ask your questions and post the feedback you have about the content in the liveBook Discussion forum. We want you to get the most out of your readings to increase your understanding of the projects.

— Aliénor Latour, Donia Chaiehloudj and Pascal Bertrand

In this book

1 Meet Go

This chapter covers

- Introducing the Go language and why you would want to learn it
- Presenting this book and how to use it
- Detailing why wanted to write this book
- Writing clean and tested code

This book provides you with a set of fun projects to progressively explore the features of the Go language. Each pocket-sized project is written in a reasonable number of lines. Our goal is to provide various exercises so any developer who wants to begin with Go or to explore the language can follow the steps described in each chapter.

We want to help the reader become a good modern software developer by using the Go language. We will use our experience as software engineers to provide meaningful advice for newcomers and seasoned developers.

This book also contains tutorials for implementing APIs with microservices, demonstrating how the language is great for cloud computing. It finishes with a project that uses TinyGo, the compiler for embedded systems.

If you are a beginner at programming, we wholeheartedly suggest starting your Go experience with https://www.manning.com/books/get-programming-with-go.

1.1 What is Go?

Go is a programming language that was originally designed to solve various problems in large-scale software development in the real world, initially within Google and then for the rest of the business world. It addresses slow program construction, out-of-control dependency management, complexity of code, and difficult cross-language construction.

Each language tries to address them in a different way, either by restricting the user or by making it as flexible as possible. Go’s team chose to tackle them by targeting modern engineering. That’s why it comes with a rich tool chain.

The tool chain covers compilation and construction, code formatting, package dependency management, static code inspection, testing, document generation and viewing, performance analysis, language server, run-time program tracking and much more.

Being built for concurrency and networked servers explains the fast adoption of the language in software companies of all sizes in the last few years.

Additionally, Go is used by a large community of developers, who have been sharing their source code on public platforms, for others to use or be inspired. As developers, we love to share and reuse what other clever people have written.

1.1.1 History and philosophy

Go started in September 2007 when Robert Griesemer, Ken Thompson, and I began discussing a new language to address the engineering challenges we and our colleagues at Google were facing in our daily work.

-- Rob Pike

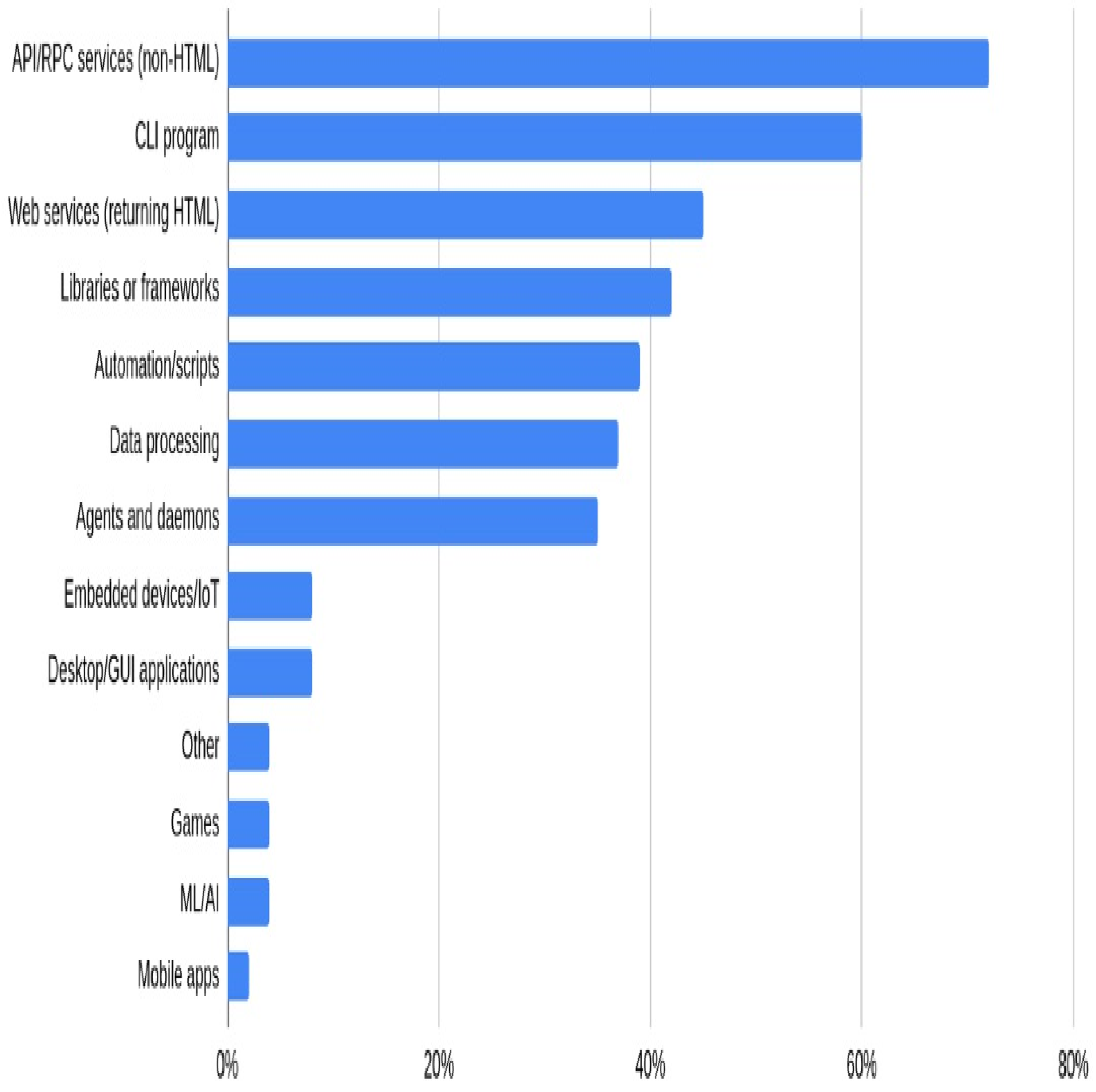
Go was designed to improve productivity in a time when multicore networked machines and large codebases were becoming the norm.

The design choices are driven primarily by *simplicity*, which makes learning the language a quick task. There are only 25 reserved keywords in the entire language (pre-version 1.18 - a version which you will read about a lot). The rest is simply the sense you want to give to it. And poetry.

Even more importantly to us, it provides for the needs of the modern software industry. Dependency management, tools for unit testing, benchmarking and fuzzing, formatters, all of the usual tools of a developer are built-in and standardised.

1.1.2 Usage today

According to the 2021 Go survey[1], the language is vastly used for API/RPC services. The close second usage is runnable programs with a command-line interface (CLI), then web services that return HTML, libraries and frameworks, automation, data processing, agents and daemons. 8% of Go developers use it on embedded systems, 4% for games and 4% for machine learning or artificial intelligence.



Although it cannot yet call on decades of libraries like Java-based languages, it benefits from a vast and welcoming community of bloggers, teachers, open source contributors, who create learning material in every form, this very book included.

There is currently a high demand for Go engineers. Learning the language can therefore allow a big career jump forward. From the authors' personal experience, the recruiting fields include areas like fintech, medtech, foodtech, gaming, music, all sorts of e-commerce platforms, aerospace research or satellite imaging processing.

1.2 Why you should learn Go

This book aims to get the reader up and running using the Go language on the job, in the context of modern engineering, be it for personal curiosity, as part of a research exercise, or in the context of an industrial project.

We will not cover everything there is to know about the language, but focus instead on the main things we need, as developers, in order to be productive and efficient.

Let’s see why Go is a great investment of your learning time as a developer.

1.2.1 How and where can Go help you?

Go is a versatile language designed for maintainability and readability. It is optimal for backend software development and has great integration with modern cloud technologies.

Considering the average turnover in tech companies is getting lower every year, a little over a year today, it is important that code written by one person can be read by another after they leave the company. It is therefore crucial that the language chosen by the company aims for readability.

Go’s key features make it a reliable and secure language, with a fast build time.

Some applications leverage goroutines which are a safer and less costly way of dealing with parallel computing than threads. Threads rely on the OS which has a limit related to the size and power of the CPU, whereas goroutines happen at the application’s level. To make the stacks small, Go uses resizable, bounded stacks. A newly minted goroutine is given a few kilobytes, which is almost always enough, and can grow and shrink, allowing many goroutines to live in a modest amount of memory. It is practical to create hundreds of thousands of goroutines in the same address space.

Even though it is not an object-oriented language and has no inheritance system, Go supports most of the features via composition and implicit interfaces. The old argument of Go missing generics, making it verbose and requiring a lot of boilerplate copy paste, has come a long way towards becoming invalid since the famous 1.18 version. As we write, discussions are still in progress to make it as versatile as it is simple.

Go is a compiled language, meaning that all syntax errors will be found during compilation rather than at runtime. We all prefer to know about mistakes in the safety of our own computer rather than discovering them, say, in production.

It is easier to run applications at scale with Go than with many other languages. Go was built by Google to solve problems at a Google-size scale. It is ideal on large concurrent applications.

Cloud platforms love Go. They provide support for Go as a major language. For example, cloud functions and lambdas support Go in all the most-used providers. Major cloud tools, such as Kubernetes or Docker, are written in Go.

1.2.2 Where can Go NOT help you?

Despite its high versatility, there are few use cases that Go is not made to cover.

Go relies on a garbage collector to release the memory it uses. If your application requires full control over memory, pick a lower-level language like the C family can provide. Go can wrap libraries written in C/C++ with CGo, a translation layer created to ease the transition between the 2 languages. With this CGo twist, you can wrap dynamically-linkable libraries.

The Go toolchain mostly produces executables - generating a Go-compiled library is painfully achievable. We won’t cover it in this book. In most common cases, updating the version of a Go dependency implies rebuilding the binary with that new version. This also means that, in order to use a Go library, you need to have access to its source code.

The Go compiler supports an interesting list of different platforms and operating systems, but we wouldn't recommend writing an operating system with Go, although many brave souls have done it. The main reason for this is the handling of the memory as it is done in Go: the garbage collector regularly discards bits that are no longer used. As with all garbage collectors, it is adjustable, but it won’t release memory exactly how you want or when you want.

Go binary files are known to be bigger than average. It is usually not a problem in a Cloud environment, but if you require light binaries, consider using the TinyGo compiler. See the last chapter of this book for an introduction.

Last, the difficulty to google up answers - seriously, who names their language with such a common word? Apparently, Google themselves. Here’s a pro tip: when trying to find an answer, use “golang”, which is not the real name but is what search engines will recognise. Sometimes it’s like trying to find documentation in C on strings – you don’t get what you were expecting.

We can also mention the difficulty in hiring Go developers, which is, actually, good for us developers.

1.2.3 Comparison with commonly used languages

The main reason to use a language other than Go, up to 2021, was the absence of some features and the lack of maturity in the ecosystem. That was before 1.18, a version that changed the game.

We compiled some of the features that developers consider in their choice of a language to address their project’s needs. In our experience, verbosity and garbage collection are important criteria today to make us more productive.

Table 1.1 Comparison of four programming languages

	C++	Python	Java	Go

design philosophy	high-level object oriented, procedural, multiparadigm	high-level object oriented	high-level object oriented	procedural and data-oriented programming, supports most OOP features
error management	via exceptions	via exceptions	via exceptions	errors are values
types and compilation	statically typed language compiled	dynamically type compiled at runtime	compiled runs in a virtual machine	statically typed language compiled
concurrency	either memory-level or os-level threads	os-level threads	os-level threads and libraries	goroutines and channels
interfaces	explicit	implicit and explicit	explicit	implicit
memory release	full control	garbage collector	garbage collector	garbage collector
main use cases	high performance low overhead	excellent for data analysis	well suited for web applications	adapted for web APIs and cloud computing

testing tools	external frameworks	built-in native tools for testing	external framework JUnit	built-in native tools (test, bench, fuzz)

1.3 Why Pocket Projects?

At the end of the XIXth century, scientists started to theorise learning. Among them, John Dewey wrote in 1897 a long list of good reasons why doing is the best way to learn. Since then, experience has proven his claims in many education systems and learning situations.

The projects that we propose here are timed for busy people. We made sure to keep them as small as possible while still making them rewarding. We admit that some of them are not particularly fun, but these are the most useful in a real-world project.

1.3.1 Whom this book is for

First and foremost, we think about developers who know and use another language and would like to extend their professional skills. We want to get you up and running by sharing practical usage of the language.

We focus on industry standards, including long-term considerations and not just a tiny project’s throwaway code that you don’t bother to test.

We are also thinking about teams who are evaluating Go for their next project. Diving into Go’s main features will help you decide that it’s the best language you can possibly invest in.

1.3.2 What you will know after reading the book (and writing the code)

First, we want to make sure that you understand what the book explains. For this matter, we will guide you through our journey of chapters describing the implementation of the current code iteratively, on a commit-by-commit basis, as we consider it important to understand what's happening bit by bit.

Second, we aim at providing good and clear examples for writing industry-level Go code - recommendations that apply outside our examples, and that will help you venture into the real world of development. All of our examples contain functions that are reusable in a company.

Last, our goal is to make you realise that you can write excellent Go code by yourself once you've understood the basics.

We start at the Hello-World level, discovering the syntax of the language, and progress all the way to a service ready to be deployed in the Cloud, walking you through architectural decisions

Grammar and syntax

The first chapters focus on the grammar specific to Go. For example, how all loops start with the same keyword, breaks are implicit in switches, but also how to expose and not some of your constants and methods (what Java calls public or private).

What Go chose in its code design was to make its interfaces *implicit*. In most of the other big languages, in order to have one entity (or class) be considered as implementing an interface, it needs to explicitly state it in its definition. In Go, implementing the methods is enough. You can therefore unknowingly implement an interface that you don't know yet. This opens worlds of new possibilities in how we envision mocking and stubbing, dependency injection, and interoperability.

Even though goroutines are a great feature of Go, we won't dwell on them. In our experience, you can program efficiently in Go without them. Only one project uses them.

Finally, as you will learn throughout the book, Go does not use exceptions. It prefers to consider errors as values. This changes the way we deal with flows that don't follow the happy path, where nothing ever fails. Every program has to deal with errors at some point, and we'll cover this throughout the projects.

Testing your code

Each and every chapter includes unit testing.

All of them.

No developer today would dream about delivering code in production that is not covered by at least some tests, whatever their level. They are indispensable for any evolution the software grows through. That is why we include unit tests everywhere.

Go is also great at benchmarking different algorithms with a built-in bench command. It allows developers to compare versions too, which means you can use it to check on every commit that your code-level performance does not decrease. You will see a few examples throughout the book.

One last recent feature of the Go tooling test chain is fuzzing. Fuzzing is a way to test a system by throwing random values at it and seeing how it behaves. It is a great help in checking for vulnerabilities.

Clean code best practices

Any **code** of your own that you haven’t looked at for six or more months[2] might as well have been written by someone else.

-- Eagleson's Law

While the first few projects fit in one file, we’ll quickly need to organise the code in a way that makes it easy to maintain. By maintain, we mean that it should allow a newcomer to find their way through your code in order to fix a bug or add a feature. This fictional newcomer could be you in a very short time.

We suggest and explain some code organisation practices. We believe that Go is great for domain-driven design and organise our code accordingly. There is of course no single folder organisation for a Go project, but we aim for what makes more sense.

What to expose and what to keep for yourself has shook humanity for millenia, and software developers for decades. This question is covered as soon as we create something that goes beyond one package.

Architectural decisions

As Go is mostly used for writing services deployed in cloud environments, we added two projects to help you pick your favourite protocol: one serves HTML over HTTP, the other uses protobuf over gRPC. You will write fully-functioning services that you can easily deploy to play around and see what you prefer and what best fits your needs.

Once they are running, you need to monitor what happens in your program. One of the early and easy projects is a logger that goes beyond what the default standard library does. Another one reads an API and acts as an anti-corruption layer to insert the data from that API into your domain. A third one is a simple load balancer for your system’s traffic, that you can make more complex according to your needs.

IoT is fun

The last project is designed to run on an Arduino microcontroller, using a different compiler. It is not enough to make you an embedded systems expert. It will just play as an intro to some of the lesser-known features of Go. We hope to tickle your creativity and hope you will enjoy it all.

1.4 Summary

- Go is a modern, industry-oriented, simple and versatile language, best for backend development, widely used for cloud-oriented tools, great for CLI and even adapted to embedded systems.
- Easy to learn, teams are quickly efficient with it, writing complicated frustrating code is possible but not easy.
- This book: learn by doing pocket projects that only take a few hours, gets you up and running.

[1] https://go.dev/blog/survey2021-results

[2] Honestly, 6 months is generous.

2 Hello, Earth! Extend your hello world

This chapter covers

- Writing to the standard output
- Testing writing to the standard output
- Writing table-driven tests
- Using a hash table to hold key-value pairs
- Using flags to read command-line parameters

As developers, our main task is to write valid programs. These programs are executed on a computer, and they'll accept some inputs (e.g: keys pressed on a keyboard, a signal received from a microphone), and will produce outputs (e.g: emit a beep, send data over the network). The simplest program of all does nothing, and simply exits. That wouldn't be a very gratifying introduction to coding, would it? Instead, let's have a hearty welcoming message!

Since 1972, learning programmers discover their new language through variations of the same sentence: `Hello world`. A programmer's first autonomous step is, thus, usually to change this standard message, and see what happens when the greeting message slightly changes. Type, compile, run, smile. This is what developing a `Hello world` is about.

Programmatic greeting history

This programmatic greeting was made popular by Brian Kernighan and Dennis Ritchie's "The C Programming Language" book, published in 1978. The sentence originally came from another publication, also by Brian Kernighan, "A Tutorial Introduction To The Language B", published in 1972. This was, in all honesty, the second example printing characters in this publication - the first one having the program print hi!. The reason was that B had a limitation over the number of ASCII characters it could have in a single variable - a variable couldn't hold more than 4 characters. Hello, world!, as a result, was achieved with several calls to the printing function. This message was inspired by a bird hatching out of its egg, in a comic strip.

The goal of this chapter is to go a bit beyond these simple steps. We consider good code should be both documented and tested. For this reason, we'll have to understand how to test a function whose purpose is to write to the standard output. On top of that, thanks to Go’s native support of Unicode characters, this first chapter will be our opportunity to greet people using languages other than English and writing systems other than the Latin alphabet.

If you don’t have the go compiler on your machine yet, install it by following the steps in Appendix A. We’ll assume that, from hereon, the setup of your development environment has been completed.

Requirements

- Write a program that takes the language of your choice and print the associated greeting
- This program must be covered by unit tests

2.1 Any travel begins at home

Our journey as developers starts where everyone’s started: on a chair in front of a keyboard and a screen. To initiate this wonderful adventure, let’s write a small program that will greet us every time we run it - the well-honed hello world. As good programmers, we’ll also want to ensure the code works as expected, so we’ll test it properly.

As mentioned in Appendix A, Go code runs inside modules. Start fresh in a new directory by initialising your module using `go mod init` followed by the name you choose for your package. This name is usually the path to your code repository.

```
go mod init example.com/your-repository
```

or

```
go mod init learngo-pockets/hello
```

2.1.1 Our first program: main.go

How can we achieve getting a program to print a message to our screen? Let’s get into it! We need to write the following code in a file named `main.go`.

Listing 2.1 main.go

```
package main

import "fmt"

func main() {
   fmt.Println("Hello world")
}
```

Phew! That was a lot for a first mission. Before we take a step into these lines, you might fancy some satisfaction and run this first program. The go command for this is as follows (run it in the same directory as the `main.go` file). Hopefully, you’ll be delighted to see the expected message appear on your screen!

```
> go run main.go
Hello world
```

Yay!

What’s in a name?

As programmers, when it comes to writing code, the biggest challenge we face, on a daily basis, is giving names to variables, constants, types, package aliases, functions, or files. Or git repositories, or microservices, endpoints, namespaces, and so forth. The list is endless. Here are some tips that will help you name variables in future projects:

· If the scope of the variable is limited, say two or three lines, a one- or a two-letter placeholder is perfectly valid. However, don’t pick random letters. Use something that immediately reminds you of the purpose of this variable.

We’ll be using l for language and tc for testCase later in this chapter.

· Stay consistent between different functions: if the variable represents the same entity, use the same name.

· Otherwise, use a name that explicitly refers to the current entity. There is no need for abbreviations, unless they’re used in some other place in the code - url, id, would be clear enough and understandable by everyone. Think row, column, book, address, and order, …

· Go’s convention, when it comes to naming, is to use camelCase for unexposed functions, types, variables and constants. For packages, try as much as you can to use a single word.

· Go’s variables do not need to describe their type. The Hungarian notation is not in use in Go. Your IDE will be kind enough to let you know if a variable is a pointer or a value.

· Finally, variable names cannot start with a number, nor can functions, types or constants.

Our next step is to understand what we just wrote. Indeed, our tasks as Go developers will scarcely be achievable with mere copies of what we can find in remote resources. There is a part in coding relating to creativity that shouldn’t be ignored. As for every craft, practice makes perfection, and pretty soon, we should have acquired enough knowledge to dare alter this first program to meet our inspirations. This book will guide us through the different steps that will eventually guarantee self-confidence through understanding.

For starters, we will focus on the first line of the program.

```
package main
```

Every Go file begins with the name of its package, in this case `main`. Packages are Go’s way of organising code, similar to modules or libraries in other languages. For now, everything fits in the `main.go` file, which must reside in the `main` package. We will see more on how to make and use packages in Chapter 3.

The `main` package is a bit particular, for two reasons. First, it doesn’t respect Go’s convention of naming the package after its directory (or the other way around). Second, this is how the compiler knows the special function called `main()` will be found here. The `main()` function is what will be executed when the program is run.

After the package’s name follows the list of required imports this file will use. Imported packages are composed of standard library packages and third-party libraries.

```
import "fmt"
```

Most Go programs rely on external dependencies. A single Go file, without the help of imported packages, can only handle a limited set of tools. For the sake of the language’s unapologetic simplicity, these tools do not provide much, and writing to external devices is not in that limited set.

In order to use features in such external dependencies, we need to import the package where they reside. This is precisely what the `import` keyword performs - give visibility over the functions and variables provided in a specific package, somewhere else. External libraries are identified by the URL to their repository; more on this later. For the moment, the important information to remember is that any import that does not look like a URL is from the standard library, meaning it comes with the compiler.

In our case, we use the `fmt` package, the standard-library package for formatting and outputting any kind of data thanks to the `Println` function, standing for print with new line. A very useful function for cheap debugging!

Finally, we have the `main()` function itself. It does not take any argument and does not return anything. Simple. Go is a simple language.

```
func main() {
    fmt.Println("Hello world")
}
```

From the `fmt` package, the `Println` function writes to the standard output. If you give it an integer or a boolean variable, it will display the human-readable version of that entity. `Println` is a sibling of a vast family of functions in charge of formatting messages.

Note that indentation in Go is made with tabs. No need to start a debate, it’s written in the documentation and everyone does it that way.

A capital question

You might wonder why `Println` starts with a capital letter. The long chapter about scope and visibility is:

· Any symbol starting with a capital is exposed to external users of the package;

· Anything else is not accessible from outside the package. Common examples of unexposed names include those starting with a lowercase letter, and those starting with an underscore.

This applies to variables, constants, functions and types.

And that's it. Really.

That's why the name of the function `Println` starts with a capital.

2.1.2 Let's test with Example

Good job, now that we wrote the program, we can test it! As we’ll see, this isn’t the only way of developing - sometimes, we can start with writing the tests, and then the code. What’s important is that code and tests go hand-in-hand. Writing code with no tests is as dangerous as thinking your brand-new toaster will return perfectly crispy and still tender bread on its first use without checking any of its settings.

But what’s in a test? By “test”, we mean automated tests (or at least automatable), not relying on human evaluation. The test could be written in shell, in Fortran, in Go, or in any language of your choice. It has to be able to tell the human user that everything went fine, or that something didn’t - in which case, some verbosity is always welcome. For this first project, we consider that running the code and “seeing” that the output is `Hello world` is not enough, at least not as the sole test of our code. What if that space character between the words were a non-breakable space character, which we humans can’t differentiate from a regular space character? The output string wouldn’t be the same, but we wouldn't be able to tell.

And why should we test? After all, the code did execute as we wanted when we ran it, right? Although this is true, it’s only been true once. And, in a larger project, where a piece of code isn’t executed only once, and is regularly tinkered with, tests are a great method of ensuring we didn’t break previous behaviour. Tests are an important block of any continuous integration pipeline - if not the most important.

Example vs. Test

A little technical foreword is needed here. While Go functions usually return values, very few write specifically to the standard output. The test strategy that we’ll implement here is only necessary when checking the standard output, which means it won’t be the default approach for the rest of the code. However, since this is our first function, and as we want to test it, this is the easy way. We will see more about test functions very quickly.

Examples are not only used for testing the standard output but also, as their name suggests, for giving the users and maintainers of your code a good starting point. They will appear in the documentation generated by `go doc`.

Go offers lots of tools to test the code, let’s use them! Here, our goal will be to test the `main` function - a task that is quite uncommon. The vast majority of Go code lies in other functions - if not other packages - and those are the functions that we heavily test. Most of the time, the `main` function will call these tested functions and will simply be in charge of printing a string or returning a status code. Apart from this occasion, the tests in this book will not be on the `main` function, but rather on the functions it calls.

First, we need a test file, which we’ll name `main_internal_test.go`, for the following reasons:

- `main` because the file we test is named `main.go`
- `internal` because we want to access unexposed methods, a convention that we choose to follow in this book
- `test` because this is a test file. When it comes to building or executing the program, `*_test.go` files are ignored by the compiler and only there can tests be run.

Internal and external testing

There are two approaches to testing. In one case we test from the user's point of view, so we can only test what is exposed; we name this external testing. The test files should be in the {packagename}_test package.

In the second case, we know everything that goes on inside and we want to test the unexposed functions. The test files should be in the same package as the source file.

These two approaches are not exclusive, and should be seen as complementary.

Raise the standard

How do we test it? How can we make sure that something is sent to the standard output from within a function? Go provides a specific tool based on a test function's name, which can be used to test the standard output of that function. If a function's name, in a test file, matches the `Example<Function>` pattern - in our case, `ExampleMain` - Go will identify it as eligible for standard output verifications. Even though `main` is not exposed, the function is in PascalCase, requiring a capital M here.

Listing 2.2 main_internal_test.go: Test the printed output

```
package main

func ExampleMain() {
    main() #A
    // Output: #B
    // Hello world
}
```

The testing function wraps a call to `main`, the tested function, at line 04.

To assert that the expected output message `Hello world` has been sent to the standard output, we use Go’s `Example` syntax, which allows us to write a commented line containing `Output:`. Any commented lines right after this one will be the expected value that Go’s test utility uses to check the output generated by the body of this `Example` function.

An `Example` function without this Output will be compiled but not executed during tests. It will appear in the documentation and can be very useful for users of your code.

Let's run the test

To run a test, call Go’s `test` command in that directory.

```
> go test
PASS
ok      learngo-pockets/hello    0.048s
```

The output lists the test files Go went through. Each line will be the name of your module followed by the path to the package inside it.

Writing tests comes with several benefits.

Note

It’s important to keep in mind this quote by Edsger Dijkstra: “Testing can prove the presence of bugs, but not their absence!”. A single test will not demonstrate a piece of code is error-proof at all. The more tests we have, the more trustworthy the code is.

First, we have an automatable process that will check that the code we have produces a deterministic output. Second, with this test, we can start altering the code - and every change, every tiniest bit of line we modify, can be validated with a run of the previous test. Finally, and this will be covered in more detail in a later chapter, writing tests with `Example` plays an important part in Go’s documentation.

2.1.3 Calling the greet function

The goal of this chapter isn't only to print a nice message to the user. We want some variations, some modularity. Taking a step back, the `main` function does two distinct things: first, define a specific message, and second, print it. We've gobbled everything on a single line in the previous code, but that doesn't leave any space for adaptations.

Since we aim to enrich the message, we need some flexibility here. We'll begin by extracting the message generation into a dedicated `greet` function. This function returns a string that we can keep in a variable we call `greeting`.

Below is the full code refactored with the extraction.

Listing 2.3 main.go: Moving the Println call

```
package main

import "fmt"

func main() {
    greeting := greet() #A
    fmt.Println(greeting)
}

// greet returns a greeting to the world.
func greet() string { #B
    // return a simple greeting message
    return "Hello world"
}
```

Let's look closer.

The new function is called `greet` since it will return the greeting message. For now, it takes no parameters and simply returns the message in the form of a string.

```
// greet returns a greeting to the world.
func greet() string {
    return "Hello world"
}
```

In the main function, we call the new `greet` function and store its output in the `greeting` string variable, which we print.

```
func main() {
    greeting := greet()
    fmt.Println(greeting)
}
```

We refactored. Does the test still run? It should, but it is not as unitary as it could be. We can write a test around `greet` with a lot more flexibility.

2.1.4 Testing a specific function with testing

Refactoring, as we just did, shouldn't change the code's behaviour. We can still run our previous test, and it should still pass. But since we'll want to enrich the `greet` function, that's the one we should be covering with dedicated tests, as meagre as it is.

Go, as part of its standard library packages, offers the possibility to use the package `testing`. We'll be using it a lot throughout this book, trying to benefit from every aspect that the Go designers put into the language so we don't have to write our own tools or spend time benchmarking independent testing libraries. As its name nicely suggests, the `testing` package is written for writing tests.

We've already seen the `Example<Function>()` syntax, which is used for documentation and for testing standard output. Let's venture into a new set of test functions: those with the `Test<Function>(t *testing.T)` signature. There is an important difference here with the previous category: these functions accept a parameter - a pointer to a `testing.T` structure. The reasons for using a pointer here are beyond the scope of this chapter, we'll cover them later.

A `TestXxx` function runs one or more tests on a function, as defined by the developer. We will start with one, and then grow. A test consists in calling the function and checking its returned value, or the state of some variable, against a wanted value or state. Should they match, the test is considered as passing. Otherwise, it is considered failing.

Every test has four main steps:

- **The preparation phase**, where we set up everything we need to run the test - input values, expected outputs, environment variables, global variables, network connections, etc.;
- **The execution phase**, where we call the tested function - this step is usually a single line;
- **The decision phase**, where we check the output to the expected output - this might include several comparisons, evaluations, and sometimes some processing - and have the test either fail or pass;
- **The teardown phase**, where we kindly clean back to whatever the state was prior to the test's execution - this step is made extremely simple thanks to Go's `defer` keyword: anything that was altered or created during preparation should be fixed or destroyed here.

Our `TestGreet` function will be written in the same `main_internal_test.go` file as earlier, mostly because the tested function, `greet`, is also in the same `main.go` file. Let's have a look at the additions we brought to the file.

In Go, we like to use `want` to the expected value and `got` for the actual one.

Listing 2.4 main_internal_test.go: Testing `greet`

```
package main

import "testing"

func TestGreet(t *testing.T) {
    want := "Hello world"

    got := greet()

    if got != want {
        // mark this test as failed
        t.Errorf("expected: %q, got: %q", want, got)
    }
}
```

The first difference with the previous version of this file is at line 03: we now need to import the `testing` package, because we use a parameter of type `*testing.T` in our `TestGreet` function.

This is a line that will appear in every single test file we'll see as Go developers. Its absence should be a red flag when reviewing industry code.

```
import "testing"
```

The second important change in this file is, of course, the new `TestGreet` function.

```
func TestGreet(t *testing.T)
```

We've added comments in the body of this function so that it follows the previous list of steps.

The preparation step, in our case, consists in defining the expected output of the `greet` function call. Since this doesn't alter the environment, there is nothing to rewind after the execution of the test, and we don't need to defer any closure steps.

The execution phase simply consists in calling the tested `greet` function, and, of course, in capturing its output into a variable.

Listing 2.5 main_internal_test.go: Body of the test

```
    want := "Hello world" #A

    got := greet() #B

    if got != want { #C
        // mark this test as failed
        t.Errorf("expected: %q, got: %q", want, greeting)
}
```

The decision phase here isn't too tricky. We need to compare two strings, and we will accept no alteration, therefore the `!=` comparison operator works fine for us here. We'll soon face cases where comparing two strings isn't enough, but let's not skip steps, as we still have a final line here that needs more explanation.

```
t.Errorf("expected: %q, got: %q", want, got)
```

So far, the need for the `t` parameter wasn't obvious. As mentioned earlier, a test needs to be either `PASS`-ing or `FAIL`-ing. Calling `t.Errorf` is one way of letting the `go test` tool know that this very test was unsuccessful. `Errorf` has a similar signature as `Printf`; see Appendix B for more about formatting strings.

Once again, you can run the tests with the same command as earlier, `go test`.

Before we move on to the next section, now's time for playing a bit. Change the contents of the `want`, and re-run the tests.

The reason for this early refactoring might not appear right now. By the end of this chapter, however, as we implement new functionalities in our code, the file will grow in size. It is good practice, in Go as in many other development languages, to keep the scope of a function narrow. This serves several purposes, such as

- making the code testable;
- making debugging the code easier;
- making the mission of a function explicit.

Overall, the cognitive charge of a function should be minimal. No one wants to face a wall of text featuring multiple layers of indentations.

We've now written a program that greets the user with a lovely message. We know it works fine because we've written tests to cover the code. But there's a small catch. It will only write English greetings. Our program can be improved to be used by people using languages other than English. Imagine you're applying at a Canadian company, where employees speak both French and English. How nice would it be if they could use it too, and be greeted with a language of their choosing?

2.2 Are you a polyglot?

Our program is very static. It will always run and print the same message, regardless of the user. Let's adapt our code to support several languages, and have the user decide which one they want. In this section, we will:

- Add support for a new language in the `greet` method
- Handle the user’s language request
- Adapt the tests and ensure we didn’t break the previous behaviour

In order to display `Hello world` in a different language, we need to be able to tell the program which language we want to use. This will be performed in two iterations, the first one being supporting new languages, and the second one being opening our program to the user’s choice of language.

2.2.1 Parlez-vous français? The switch angle

Our current `greet` function only returns a hard-coded message. Since it can only return one message, we want some logic in there to determine which greeting to output. There are several options for this in Go. The first one that comes to mind, the `if` approach, would only work for one or two different languages. More, and the code becomes an unnecessarily long list of checks. Here, we’ll explore the other two. Since we need to support another language, let’s pick French. Here’s what the full code now looks like:

Listing 2.6 main.go: Adding a new language

```
package main

import "fmt"

func main() {
    greeting := greet("en")
    fmt.Println(greeting)
}

// language represents the language’s code
type language string

// greet says hello to the world in the specified language
func greet(l language) string {
    switch l {
    case "en":
        return "Hello world"
    case "fr":
        return "Bonjour le monde"
    default:
        return ""
    }
}
```

Clarity through typing

Using the proper type for variables is important. We need to know what we are talking about. And what we’re talking about is having a language parameter that will be used to specify which greeting message should be returned by the `greet` function. This language parameter could be a string containing the language description. It could be an integer referring to an index of existing languages. It could be the url to a dictionary. It could be many, many things. For now, we’ll keep it simple and use a string.

```
type language string
```

The input language will be a `string` that represents a `language`. This type definition helps us and the users of our libraries understand what values are expected and makes mixing up parameters really difficult.

Selecting the right language

Now that we have an explicit type, we can pass it as a parameter to the `greet` function.

The new signature becomes:

```
func greet(l language) string
```

To call it, we changed the first line of our main function:

```
greeting := greet("en")
```

How does the compiler know if this is a `string` or a `language`? By looking at the signature of the function. `greet` requires a `language`, so it will type the constant as such.

For the first iteration, we can add a `switch` on the `language` and return the corresponding greeting. The default value for the moment is just an empty string. We consider that `switch` is clearer when dealing with most types - the exceptions being `error`, pointers, and `bool`.

Listing 2.7 main.go: Switch on language

```
switch l {
    case "en":
        return "Hello world"
    case "fr":
        return "Bonjour le monde"
    default:
        return ""
}
```

Note that between each case, contrary to many other languages, we don’t break. Breaking is implicit in Go because it is such a potential source of errors. Of course, as we return here in each case, the point is moot, but now you know.

In the `main` function, we need to pass the desired `language` to our upgraded `greet` function and print the output. For example, `"en"` for English.

2.2.2 Adapt the test with Test<Function> functions

Previously, the `greet` function accepted no parameter. It now takes one, which means we broke the contract we had with the users of our code. Well, for now, the only user is a test, but that still counts. We now want to test the `greet` function with various inputs.

We will make a call to the `greet` function by passing the desired input language and storing the output in a variable, so we can verify it. The preparation phase now contains two variables: the desired language, and the expected greeting message.

Let’s use a new convention of the `testing` package: when testing a function with two (or more) different scenarios, we can write several functions, `Test<FunctionName>_<ScenarioName>`. The full test file now looks like this:

Listing 2.8 main_internal_test.go: Separated tests cases

```
package main
```

```
import "testing"

func ExampleMain() {
    ...
}

func TestGreet_English(t *testing.T) {
    lang := language("en")              #A
    want := "Hello world"

    got := greet(lang)  #B

    if got != want {    #C
        // mark this test as failed
        t.Errorf("expected: %q, got: %q", want, got)
    }
}

func TestGreet_French(t *testing.T) {
    lang := language("fr")                         #A
    want := "Bonjour le monde"

    got := greet(lang)       #B

    if greeting != want {        #C
        // mark this test as failed
        t.Errorf("expected: %q, got: %q", want, got)
    }
}

func TestGreet_Akkadian(t *testing.T) {
    // Akkadian is not implemented yet!
    lang := language("akk")                        #A
    want := ""

    got := greet(lang)               #B

    if got != want {               #C
        // mark this test as failed
        t.Errorf("expected: %q, got: %q", want, got)
    }
}
```

As you can see, the `TestGreet_English` function is in charge of testing the English greeting, while the `TestGreet_French` function tests the French message. While this approach is interesting and worth remembering, you will have noticed that in our case, there is no real change between the English and the French scenarios. Only the preparation step differs, and only slightly. The next section will improve on this.

To run the tests, simply run your new favourite `go test` command.

As you've noticed, we've added another function to test a language unknown to the program. Testing isn't always about making sure the "good" inputs provide "good" results. Making sure the safety nets are in place is almost as valuable as making sure the code works as intended.

2.3 Supporting more languages with a phrasebook

Adding entries to a `switch` clause reduces the readability of the code: it increases the size of the function, sometimes beyond screen dimensions, when the only answer we need is "if this language is supported, give me its greeting". To trim down our function without losing any functionality, in our next iteration of coding, we decide to use a `map`, a very common and useful data structure in Go. A `map` is a hash table, a set of pairs of distinct keys and their associated values. In this section, we will:

- Scale the number of supported languages
- Introduce the use of `map`

2.3.1 Introducing the Go map

Let's have a look at the implementation of the code using a `map` to store these pairs of language and greeting message:

Listing 2.9 main.go: Using a map

```
package main

import (
    "fmt"
)

func main() {
    greeting := greet("en")
    fmt.Println(greeting)
}

// language represents the language’s code
type language string

// phrasebook holds greeting for each supported language
var phrasebook = map[language]string{
    "el": "Χαίρετε Κόσμε",     // Greek
    "en": "Hello world",       // English
    "fr": "Bonjour le monde",  // French
    "he": "שלום עולם",         // Hebrew
"ur": "بیلو ",         // Urdu
    "vi": "Xin chào Thế Giới", // Vietnamese
}

// greet says hello to the world in various languages
func greet(l language) string {
    greeting, ok := phrasebook[l]
    if !ok {
        return fmt.Sprintf("unsupported language: %q", l)
    }

    return greeting
}
```

Our `map` defined below associates a greeting message to every language as a pair of `{language, greeting}`. For this chapter, we use a global variable that holds the greetings.

Listing 2.10 main.go: `map` definition

```
// phrasebook holds greeting for each supported language
var phrasebook = map[language]string{
    "el": "Χαίρετε Κόσμε",     // Greek
    "en": "Hello world",       // English
    "fr": "Bonjour le monde",  // French
    "he": "שלום עולם",         // Hebrew
"ur": "بیلو دنیا",         // Urdu
    "vi": "Xin chào Thế Giới", // Vietnamese
}
```

Our next step is to use this phrasebook instead of the `switch` in the `greet` function.

Listing 2.11 main.go: `greet` method

```
// greet says hello to the world in various languages
func greet(l language) string {
    greeting, ok := phrasebook[l]
    if !ok {
        return fmt.Sprintf("unsupported language: %q", l)
    }

    return greeting
}
```

Accessing an item in a Go `map` returns two pieces of valuable information: a value - in our case, the message associated with the key language `l` - and a boolean (`ok` as per convention) that informs us of whether the key was found. The syntax of affecting both returned values to two different variables on a single line might be new to some programmers - it doesn’t exist in Java or C. This is something we do a lot in Go.

```
greeting, ok := phrasebook[l]
    if !ok {
        return fmt.Sprintf("unsupported language: %q", l)
    }
```

It is necessary to check the second return value of the access to the map - if the language were unsupported, we’d receive the zero-value of a string, which is the empty string, with no knowledge of whether the map had an entry for our language.

Note that in production-ready code, we would be returning an error because an empty string does not carry any meaning. We chose to keep it simple for now. Errors will be covered in later chapters.

Multiple return values: We’ll see many occurrences of multiple value assignment, mostly in four common cases:

- Whenever we want to know whether a key is present in a map, as we do here, where we retrieve the value, and the information of the presence of the key in the map (as we did in this piece of code);
- Whenever we use the `range` keyword, which allows us to iterate through all the key-value pairs in a map, or all the index-value elements of a

slice or array (an example in the next version of the test file, more in the next chapter);

- Whenever we read from a channel with the `<-` operator, which returns a value and whether the channel is closed (examples can be found in later chapters);
- Finally, the most frequent case is when we retrieve multiple values returned by a single function. This book will contain numerous occurrences of this case, mostly due to Go’s handling of errors.

2.3.2 Writing a table-driven Test

Our previous tests were linear - they tested every language in a sequential way. Taking a step back, we realise each test runs the same sequence: take an input language, call the `greet` function, and check the greeting for that language is the expected one. This can be summed up in the following snippet of code that was executed for languages `"en"`, `"fr"` , or `"akk"` in our previous example:

Listing 2.12 main_internal_test.go: Call greet and chec

```
    got := greet(language(lang))
    if got != want {
        t.Errorf("expected: %q, got: %q", want, got)
    }
```

There is no point in duplicating this piece of code every time we want to check we properly support a new language. Isn’t the test always going to be the same? Do we really need to add an extra ten lines to our test file if only two of these lines change? This is not sustainable. That was our motivation to use maps in the body of the `greet` function, and this is also our motivation to use maps in our tests! We can make use of table-driven tests to enhance the reusability and clarity of our test file, and have the nice side-effect of shrinking it a lot! Let’s have a look at the new test before we explain it.

Listing 2.13 main_internal_test.go: Table-driven Tests

```
func TestGreet(t *testing.T) {
    type testCase struct {
        lang language
        want string
    }

    var tests = map[string]testCase{ #A
        "English": {
            lang: "en",
            want: "Hello world",
        },
        "French": {
            lang: "fr",
            want: "Bonjour le monde",
        },
        "Akkadian, not supported": {
            lang: "akk",
            want: `unsupported language: "akk"`,
        },
        "Greek": {
            lang: "el",
            want: "Χαίρετε Κόσμε",
        },
        "Hebrew": {
            lang: "he",
            want: "שלום עולם",
        },
        "Urdu": {
            lang: "ur",
            want: "ہیلو دنیا",
        },
        "Vietnamese": {
            lang: "vi",
            want: "Xin chào Thế Giới",
        },
        "Empty": {
            lang: "",
            want: `unsupported language: ""`,
        },
    }

    // range over all the scenarios
    for name, tc := range tests {
        t.Run(name, func(t *testing.T) {
            got := greet(tc.lang) #B

            if got != tc.want { #C
                t.Errorf("expected: %q, got: %q", tc.want, got)
            }
        })
    }
}
```

As we’ve seen previously, every test we want to run needs two values: the language of the desired message, and the expected greeting message that will be returned by the `greet` function. For this, we introduce a new structure that contains the input language, and the expected greeting. Structures are Go’s way of aggregating data types together in a meaningful entity. In our case, since the structure represents a test case, we’ll name it `testCase`. Our structure needs only to be accessible in the `TestGreet` function (and nowhere else), so let’s define it there.

```
type testCase struct {
    lang language
    want string
}
```

This will make writing a test over a pair of language and greeting even simpler.

Now we can easily write one test case, let’s see how to write lots. In Go, the common way of writing a list of test cases is to use a `map` structure that will refer to each test case with a specific description key. The description should be explicit about what this case tests.

We now have everything we need to write a list of test cases.

Listing 2.14 main_internal_test.go: Test cases definition

```
var tests = map[string]testCase{
    "English": {
        lang: "en",
        want: "Hello world",
    },
    "French": {
        lang: "fr",
        want: "Bonjour le monde",
    },
}
```

In order to test these scenarios, we can iterate over the `tests` map. As we’ll see in more detail in the next chapter, this `for` + `range` syntax returns the key and the value of each element of the map. We then pass the name as the first parameter to `Run`, a method from the testing package that makes tests so much easier to use: if a test case fails, the tool will give you its name so that you can find it and fix it. Also, most code editors let you run one single test case if you use this syntax.

Remember, this map associates a description to a test case, hence the name of the variable, `tc`.

Listing 2.15 main_internal_test.go: Execution and assertion phases

```
for name, tc := range tests {
    t.Run(name, func(t *testing.T) {
        got := greet(tc.language) #A

        if got != tc.want { #B
            t.Errorf("expected: %q, got: %q", tc.want, got)
        }
    })
}
```

Since the call to the `greet` function is the same regardless of the input language, creating a new test case only has us adding an entry in the `tests` map:

Listing 2.16 main_internal_test.go: Test cases

```
   var tests = map[string]testCase{
                 "English": {...},
        "French": {
            lang: "fr",
            want: "Bonjour le monde",
        },
        "Akkadian, not supported": {
            lang: "akk",
            want: `unsupported language: "akk"`,
        },
        // add new described scenarios here!
    }
```

Quotes in Go

You’ve probably noticed we used a different set of quotes in the expected greeting for Akkadian (akk). There are three types of quotes that are used in Go, each in its adequate context:

· The double quote ": it is used to create literal strings. Example: s := "Hello world"

· The backtick `: it is also used to create raw literal strings. Example: s := `Hello world`

· The single quote ': it is used to create runes. Example: r := '學'. A rune is a single unicode code point.

You’ve probably noticed the first two options can be used to create literal strings. The difference between raw literal strings and non-raw literal strings is that, in a raw literal string, there is no escape sequence. Writing a \n in a raw literal string will result in a backslash character \ followed by the letter n, when the string is printed. Raw literal strings are a nice way of not having to deal with escaping double quotes, which is very handy when it comes to writing JSON payloads.

We now have a program that can return a greeting in whichever language the user wants, but the only way the user gets to change the language used is, so far, to change the code of the program - which isn’t optimal! We want to get the input from the user without changing the code every time the request is sent. Since the user is running the program from the command-line, by running `go run main.go`, or by executing the compiled executable, that’s most likely where they’ll want to inform us of their choice of language.

2.4 Using the flag package to read the user's language

How can we use the input to get the user’s desired language of greeting? Go provides support for parsing the command-line arguments in both the `os` and the `flag` packages. The former is very close to C/C++’s handling of arguments - you get to access them by their position on the line, but whether they are of the form `--key=value`, `-key value`, or `-option` is left up to the developer to implement, and it's a real pain if you have repeatable fields. Oh, and that's only for parsing them, then we have to convert them to their right type.

On the other hand, the `flag` package offers support of a variety of types - integers, float numbers, time durations, strings, and booleans. Let's roll with this one!

- Use the `flag` (https://pkg.go.dev/flag) standard package to read from command-line arguments
- Call `flag.Parse` to retrieve the values
- Play with the program argument flags and check the output

2.4.1 Add a flag

The first thing we need to do, when it comes to exposing a parameter on our command-line executable, is to give it a nice and short name. Here, we'll offer the user a choice of language, which makes `lang` a fairly obvious choice.

Let's have a look at the updated code of the `main.go` file:

Listing 2.17 main.go: Using flags

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    var lang string
    flag.StringVar(&lang, "lang", "en", "The required language, e
    flag.Parse()

    greeting := greet(language(lang))
    fmt.Println(greeting)
}

// language represents a language
```

```
type language string

// phrasebook holds greeting for each supported language
var phrasebook = map[language]string{
    "el": "Χαίρετε Κόσμε",     // Greek
    "en": "Hello world",       // English
    "fr": "Bonjour le monde",  // French
    "he": "שלום עולם",         // Hebrew
"ur": "ہیلو دنیا",      // Urdu
    "vi": "Xin chào Thế Giới", // Vietnamese
}

// greet says hello to the world
func greet(l language) string {
    greeting, ok := phrasebook[l]
    if !ok {
        return fmt.Sprintf("unsupported language: %q", l)
    }

    return greeting
}
```

The goal of this section is to read the flags from the command-line, which means we need to import the `flag` package.

```
import (
    "flag"
    "fmt"
)
```

Now that we have imported this package, let’s use it. We want to read, from the command-line, the name of the language in which the user expects their greeting. In our code, the type for that entity would be a `language`, for which the closest type is a `string`.

The `flag` package offers two very similar functions to read a string from the command-line. The first requires a pointer to a variable that it will fill up.

```
var lang string
flag.StringVar(&lang, "lang", "en", "The required language, e.g.
```

The second creates the pointer and returns it:

```
lang := flag.String("lang", "en", "The required language, e.g. en
```

For this example, we’ll use the first one, mostly because it will let us introduce the `&` operator.

On the first line, we declare a variable of the type `string`. That variable will hold the value provided by the user. Let’s have a look at the syntax and different parameters of the `StringVar` call:

First, we pass to the function the address of the string. Second, we pass the name of the option, as it will appear on the command-line. Third, we give the default value for this variable. The default value is used if the user doesn’t provide the flag when calling the program. Finally, we write a short description of what this flag represents and some example values. Pointers are covered deeply in the Appendix E and used in the following chapters.

During the execution of a program, variables are stored in memory at a specific address. We can retrieve the address of a variable with the address operator &, used on a pointer. Similarly, when we have a pointer and we want to access the value, we can retrieve it with the indirection operator * used on the pointer.

In Go, when we call a function, the arguments are passed by copy. This means that if we want to allow a function to alter one of our variables, the simplest way is to give that function a pointer to our variable.

Finally, Go offers no pointer arithmetics. If you have a pointer to the first element of an array, it cannot be used to access the second element.

There is one important thing to remember, whenever we use the `flags` package, is that all these `StringVar`, `IntVar`, `UintVar` do not scan the command line and extract the value of the parameter. What does this trick of parsing the command line is the function `flag.Parse`. It will scan the input parameters and fill every variable we’ve told it would be a receiver. If you need a mnemotechnical sentence to remember it, try “Sunset `BoolVar` begins at the `Parse`-ific Ocean”.

After the call to `Parse`, the variable `l` will have the value passed by the user, and the rest of the code is the same as previously. Note that this conversion to the language type is acceptable in this context but in production code, it would be the perfect place to add validation of the value against a list of supported values, or at least a validation of the format (in our case, a string of 2 ASCII characters).

2.4.2 Test the Command-line Interface (CLI)

We have now completed the code, and it’s time to run some end-user testing. For this, we’ll simulate calls from the command-line. We have several options to make sure this works as expected. The first on our list is to simply try it out! After all, we’ve spent a good deal of pages making sure this works as we want, so some peace of mind is deserved, some time to rest for our neurons. We can pass the parameter on the command line with `go run main.go -lang=en`.

Some examples Here is an of running the main in Greek:

```
> go run main.go -lang=el
Χαίρετε Κόσμε
```

2.5 Exercises

Here is a series of exercises you can do:

1. Launch the program with the Urdu language as flag argument
2. Launch the program with no language
3. Finally, remember to check all possible scenarios. The user could be asking for languages our program doesn’t know. Launch the program with an unsupported language, e.g.: Akkadian, akk
4. Add support for the language of your choice

This concludes the first project, we hope you enjoyed it and already learned some practical info about Go.

2.6 Summary

Now that you’ve completed this first chapter, let’s take a glance back at everything that was covered:

- `go run` can be used to run a program.
- Write tests as code grows, not after.
- `go test` runs the tests.
- Testing in Go has default naming conventions.
- Table-driven tests are the best approach when we want to tinker with inputs or settings.
- The `testing` package contains all the tools needed to run tests.
- `Example` can be used to check what a function writes to the standard output.
- There are 4 phases in a test: preparation, execution, decision and teardown phase.
- The `flag` package allows for parsing command line arguments.
- Key-values are stored in `maps`.
- Accessing a `map` returns the value (if found), and whether it was found.
- Defining self-describing types rather than using built-in types makes code more understandable.
- Use `if` when only two options are possible. Otherwise, use `switch` or a `map`.

3 A bookworm’s digest: playing with loops and maps

This chapter covers

- Ranging over slice and map
- Using a map to store unique values
- Learning how to open and read a file
- Decoding JSON file
- Sorting a slice with custom comparators

Since the invention of writing, people have been using the tool to carve their thoughts through the centuries. Books were knowledge and became a hobby. We have been reading and collecting them on shelves. With technology, we are now able to share information more than ever, and give our opinion on everything, including books. In this chapter, we’ll join a group of bookworms who have been reading books. Fadi and Peggy have started registering the books they keep on their bookshelves, and they wonder if we can help them find books they both have read, and, maybe, suggest future reads.

In this chapter, we will reinforce what we learned about command line interfaces in chapter 2 by creating a book digest from bookworms’ book collections. Step by step, from a list of books per reader, we will build a program returning and printing the books found on more than one shelf. As a bonus, we will practise map and slice ranging to create a tool recommending books. The input of our executable is a JSON file, and we can learn how to read a file in Go and how to parse a JSON using the standard libraries from Go. For the sake of simplicity here, we will assume that each book has only one author. How ironic it is, you will say.

Requirements

- Write a CLI tool which takes a list of bookworms and their book collections in the form of a JSON file

- Find the common books on their shelves
- Print the common books on bookworms' shelves to the standard output
- As a bonus, recommend books for each bookworm based on their matching books with other bookworms

Limitations

- We assume each book has only one author
- Input JSON file will not surpass a megabyte

3.1 Load the JSON data

A new chapter, a new project, a new folder. Let’s launch the command to initialise the module we will be working on and call it `bookworms`:

```
go mod init learngo-pockets/bookworms
```

As a good practice, we do recommend creating a new `main.go` file with a simple empty `main` function. It is a standard first step and we will fill it throughout the chapter.

```
package main

func main() {
  // will be completed along the way
}
```

In this section, we will create the input JSON file and load the data it contains.

3.1.1 Define a JSON example

Let’s have a look at some example input data. It is a list of people with their name and their books. Each book has one author and a title.

A few words about the JSON format

The JavaScript Object Notation, widely referred to as JSON, is a file format that stores data using "key":value pairs. JSON keys are always strings, enclosed with double quotes, and JSON values can be any of the following:

· Decimal numbers (no enclosing character): `4, 3.1415`, `1e12`

· Strings (enclosed in quotes): `"Hello"`, `"1789"` (different from the number 1789)

· Arrays, enclosed in square brackets: `[1,2.5,-10]` is an array of numbers

· Boolean values (no enclosing character): `true`, `false`

· Objects, enclosed in curly braces: `{"name":"Nergüi"}`

JSON objects’ fields aren’t specifically sorted: in the example below, we could have the author appear before or after the title, and the payload would be the same. Arrays are ordered - switching the first and the second element would change the payload.

We can now write a sample bookworm file.

Listing 3.1 testdata/bookworms.json: Example of input file

```
[
  {
    "name": "Fadi",
    "books": [
      {
        "author": "Margaret Atwood",
        "title": "The Handmaid's Tale"
      },
      {
        "author": "Sylvia Plath",
        "title": "The Bell Jar"
      }
    ]
  },
  {
    "name": "Peggy",
    "books": [
      {
        "author": "Margaret Atwood",
        "title": "Oryx and Crake"
      },
      {
        "author": "Margaret Atwood",
        "title": "The Handmaid's Tale"
      },
      {
        "author": "Charlotte Brontë",
        "title": "Jane Eyre"
      }
    ]
  }
]
```

Simple enough for now.

There is a convention in Go by which any folder named `testdata` should contain, you guessed it, data for testing. To quote the `go tool` documentation, the go tool will ignore a directory named "testdata", making it available to hold ancillary data needed by the tests. Linters and other static code analysis tools should also ignore it.

Create a file named `bookworms.json` within a `testdata` folder, with some data like ours and pick your favourite books. Or you can go to our repository and copy our version.

The first step for reading this data is to open the file and load its contents as a file. The second is to parse the JSON.

3.1.2 Open a file

Because we don’t like getting lost in overly long files, we chose to cut the logic of the project into 2 files: first, `main.go` knows that it runs in a terminal and can display text; second, `bookworm.go` has the business logic, and could be copied and reused in a different setup. Don’t overthink it yet.

At this point, your file tree should look like below:

```
> tree
.
├── bookworm.go
├── go.mod
├── main.go
```

```
└── testdata
    └── bookworms.json
```

Loading the data will be the purpose of a new function that we can call `loadBookworms`. It would take the file path as a parameter and return the slice of `Bookworms` represented by the JSON document. If something wrong happens (file not found, invalid JSON, etc), it can also return an error. Don't forget to give it a docstring.

Listing 3.2 bookworm.go: `loadBookworms` signature

```
// loadBookworms reads the file and returns the list of bookworms
func loadBookworms(filePath string) ([]Bookworm, error) {
    return nil, nil #A
}
```

We previously talked about zero values, you can refer to Appendix C. In our case, the zero value of the slice of bookworms is `nil`, as is the zero value of the error interface. That's why `loadBookworms` returns `nil` and `nil` for the moment.

Go offers the platform-independent `os` package to operate system functionality. Here is the quote from the documentation: *The design is Unix-like, although the error handling is Go-like; failing calls return values of type error rather than error numbers.*

Inside the `os` package, there is a `os.File` type providing ways to open a file for reading or writing, changing rights of a file, creating a new file and many other system operations you can perform on a file. The whole list can be retrieved with `go doc os.File`. The simplest function to open our file is `os.Open`. We'll give it the path to our file as the `filePath` parameter, and it will return a pointer to a `File`, which is a file descriptor, or an error. The documentation is kind enough to let us know that the descriptor is in read-only mode, and that the returned error is of type `*PathError`.

Differences between `os.Create`, `os.Open` and `os.OpenFile`

As we can see in the documentation of the `os` package, several functions return a file descriptor, and each one has its best usage. Let's have a look at when `os.OpenFile` should be used to open a file instead of `os.Create` or `os.Open`.

`Create` creates a file with both read and write rights (but not execute) for all users (0666). If the file already existed, `Create` truncates it, sending its contents to oblivion. When `Create` succeeds, the returned file descriptor can be used to write data to the file.

`Open` opens the named file for reading only.

`OpenFile` is a more generic approach that lets the user decide whether they want to open a file for writing or reading. Most of the time, you won't need it - a call to `Open` or `Create` should do the trick. However, there are two very specific cases in which it is useful. The first case is when we want to append data to a file, without discarding its contents. Using `Open` here wouldn't - the `*File` would be in read-only -, nor would `Create` - the file's contents would be erased. The second parameter of the `OpenFile` function is a flag controlling how we open the file. The full list can be found with `go doc os.O_APPEND`. These flags are constants and should be combined to your taste. When creating or appending, use `os.O_APPEND|os.O_CREATE|os.O_WRONLY`. The second case is when we want to create a file for which the rights aren't the default ones of `Create`. `OpenFile` is the only one offering the possibility of setting specific access rights to the file, via its last parameter.

If there is an error, for all three methods, it will be of type `*os.PathError`. Note that most of the time, we will use `Open` and `Create`.

The constants in the os package are in capitals because they are part of operating system standards. Otherwise, Go prefers constants to be defined in PascalCase like everything else.

Defer

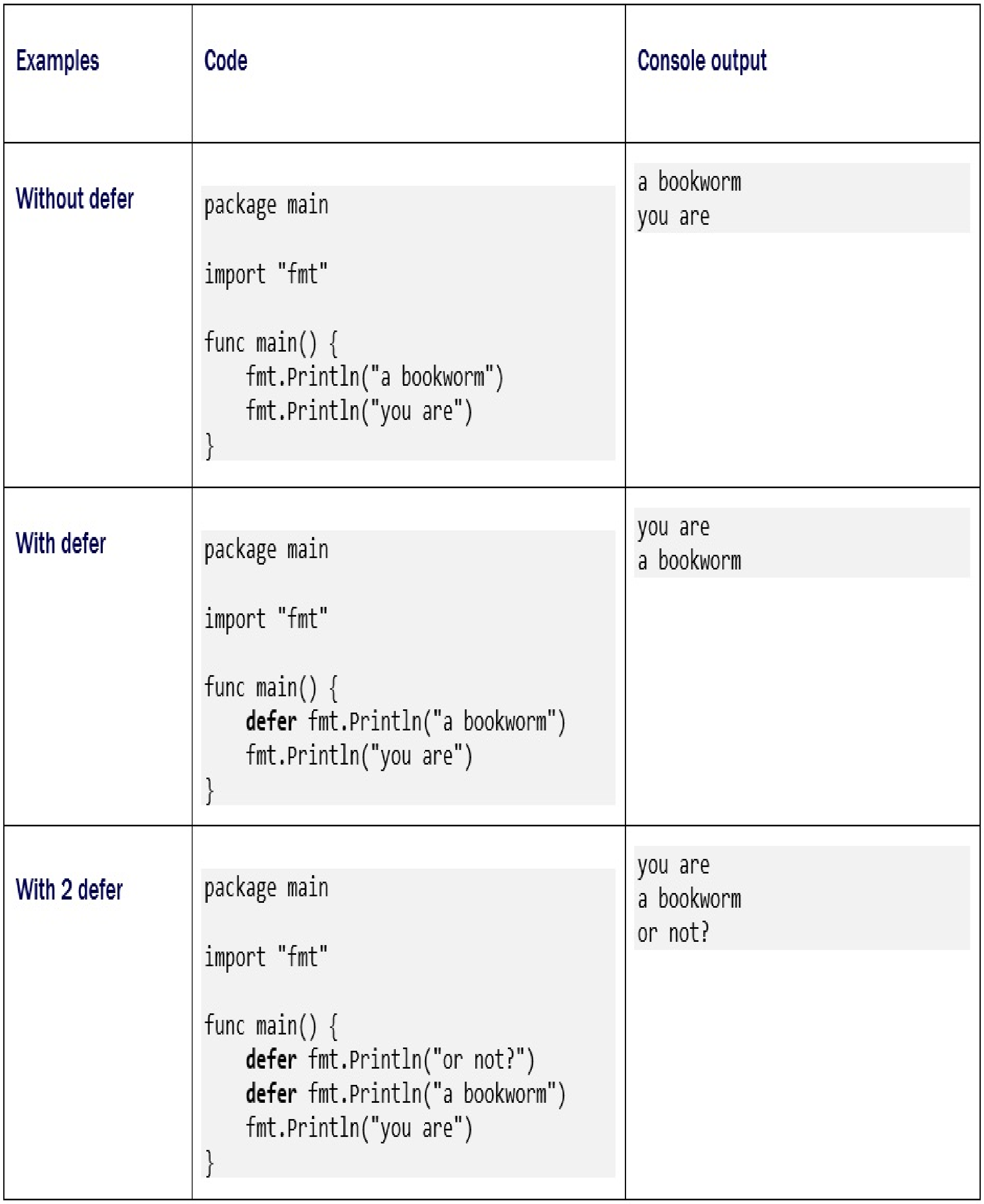
When you are done with I/O operations with a `*File`, you *must* close it by using the `Close` method on your file. This way, system resources used by the file are released and you don't create leaks with your program. If you don't close the descriptor, you may exhaust all available file handles of your system, and locking files has some complicated side effects on Windows, where you may end up blocking yourself from writing or deleting it. In theory, the garbage collector of Go should close the file at some point (no later than when your program exits), but it is always better to know when and how the file descriptor is closed. In other words, be polite and explicitly clean after yourself.

How do we know when we are done with the operations? Usually, it is by the end of the function, and that’s where we would place the call to `Close()`. But imagine a day when you have to refactor the function and you remove a lot of the code - what if you accidentally remove the call to `Close`, or if you return before it’s called? The best way to prevent it from getting lost in the rest of the function is to keep it next to `Open` or `Create`.

`defer` is a Go keyword that postpones a statement’s execution to the very end of the function, even if the `defer` appears at the beginning of it. The important point is that, whichever way a function returns, every `defer` the execution has been through will be executed. When a function returns, its deferred calls are executed in last-in-first-out order.

Let’s see simple examples:

Table 3.1 Program execution with and without defer



Examples	Code	Console output

Without defer	`package main`
	
	`import "fmt"`
	
	`func main() {`
	`    fmt.Println("a bookworm")`
	`    fmt.Println("you are")`
	`}`
		`a bookworm`
		`you are`

With defer	`package main`
	
	`import "fmt"`
	
	`func main() {`
	`    defer fmt.Println("a bookworm")`
	`    fmt.Println("you are")`
	`}`
		`you are`
		`a bookworm`

With 2 defer	`package main`
	
	`import "fmt"`
	
	`func main() {`
	`    defer fmt.Println("or not?")`
	`    defer fmt.Println("a bookworm")`
	`    fmt.Println("you are")`
	`}`
		`you are`
		`a bookworm`
		`or not?`

In the case of `os` operations, the `defer` statement is located right after checking the error returned by `Open`. If you see a call to `Open` in the code, you have to see the `Close` in the same code block, these two lines only make sense together.

The `defer` keyword is mostly used to close files, database connections, buffered readers, etc. Sometimes, you will find `defer` useful to compute the time spent inside a function. One of the go libraries for the Kafka event manager uses `Close()` because the server needs a graceful disconnect to stop it from continuing to attempt to send messages to the connected clients.

Let’s get back to our code. We want to open a file for reading. It may return an error that we have to deal with: in this case, we will simply return it to the caller. Once this is done, we know that we have a valid file description, so we need to close it.

Listing 3.3 bookworm.go: open a file

```
f, err := os.Open(filePath) #A
if err != nil { #B
    return nil, err
}
defer f.Close() #C
```

`io.Reader` and `io.Closer` are interfaces commonly used in Go packages and `os.File` implements both of them. `io.Reader` enables reading a stream of data into a slice of bytes.

3.1.3 Parse the JSON

In order to parse some JSON data, we will use the `encoding/json` package of Go. There is a good list of different encodings supported next to it, including XML, CSV, base64… We will see more details about parsing in Chapter 6 Money Converter, when we start decoding responses from HTTP calls.

The structure of Go’s standard library’s package doesn’t represent a tree of dependencies, but rather domains. Anything to do with the network will be in the `net` package, or in a package nested in `net`. Here, the `encoding` package is extremely lightweight - it only defines 4 interfaces - and we don’t have to include it to make use of the contents of the `encoding/json` package.

Define the structure associated to the JSON file

The general idea is that the Go structure that is used for decoding must match the JSON structure. Here we have a list of people whom we would call `Bookworms`, and each of them has a name and a list of `Books`. In order to tell Go which JSON field corresponds to a field in our Go structures, we use tags, which are enclosed in backticks. Tags will contain the name of the standard followed by the name of the field there.

The type of the list of books here is called a slice. More on that very soon, let's focus on the JSON first.

Listing 3.4 bookworm.go: Bookworm and Book structures

```
// A Bookworm contains the list of books on a bookworm's shelf.
type Bookworm struct {
    Name  string `json:"name"` #A
    Books []Book `json:"books"` #B
}

// Book describes a book on a bookworm's shelf.
type Book struct {
    Author  string `json:"author"`
    Title   string `json:"title"`
}
```

Look at the `json` tags. Each Go field is tagged with the name of the JSON field. Note that the name of the field does not have to match the name of the tag. It's more a convention and more readable. Here's another Go convention: fields that are slices should be named with a plural word.

Finally, and this is the most important part of the tag chapter: the decoder that we are about to code will need to write to the fields of our structure. For this, it needs to be able to "see" them, which means these fields must be exposed. Many an hour of debugging has been spent trying to understand why a field was always empty.

Decode the JSON file into a structure

Once the file is opened and fully loaded we can define a variable that will hold the information. This variable must be a slice of `Bookworms`, because this is what the JSON is giving us. We then pass a pointer to that variable to the decoder so that it can fill it up.

Remember, the alternative to passing a pointer is to pass a copy. The decoder would then fill up the copy and throw it away into the garbage collector, and we would be left empty-handed.

Listing 3.5 bookworm.go: JSON decoding in loadBookworms()

```
var bookworms []Bookworm #A

// Decode the file and store the content in the value bookworms.
err = json.NewDecoder(f). #B
Decode(&bookworms) #C
if err != nil { #D
    return nil, err
}
```

Notice how we both created and used a decoder on a single line. We were able to do this thanks to `NewDecoder` returning only a `Decoder` (and no error). Since we don’t use the decoder returned by `NewDecoder` anywhere else, it’s common practice to avoid declaring a variable for it, unless something else dictates it (e.g. line length, meaning). Instead, we simply use it by calling `Decode`.

There are a few more refined ways of decoding big JSON inputs or files, for example via an explicit streaming mechanism that avoids loading the entire file’s contents. You can look them up if you are curious in the section Improvements at the end of this chapter (3.5.2) but for the project, we trust that your test file will not exceed a few megabytes.

After that, we just need to return the bookworms that were decoded. The finalised function should look something like this:

Listing 3.6 bookworm.go: loadBookworms() opens and decodes Json file

```
// loadBookworms reads the file and returns the list of bookworms
func loadBookworms(filePath string) ([]Bookworm, error) {
    f, err := os.Open(filePath) #A
    if err != nil {
        return nil, err
    }
    defer f.Close() #B

    // Initialise the type in which the file will be decoded.
    var bookworms []Bookworm

    // Decode the file and store the content in the variable book
    err = json.NewDecoder(f).Decode(&bookworms) #C
    if err != nil {
        return nil, err
    }

    return bookworms, nil
}
```

In order to make this whole file compile, you need to import the `os` and `encoding/json` packages. If you are using a clever enough editor, it may have done it for you.

Just before we write a test, as an early reward, we can manually check our example with a simple print. Call the `loadBookworms` function in your `main` function, give it the JSON file's path as a parameter and print out the result.

Listing 3.7 main.go: Calling loadBookworms() in main()

```
package main

import “fmt”

func main() {
    bookworms, err := loadBookworms("testdata/bookworms.json") #A
    if err != nil {
        _, _ = fmt.Fprintf(os.Stderr, "failed to load bookworms: 
        os.Exit(1) #C
    }

    fmt.Println(bookworms) #D
}
```

In order to run it, you cannot use `go run main.go` anymore. Well, you can try, but you'll see Go is angry at you for calling an unknown function. This is because the `main` function makes a call to `loadBookworms`, a function that isn't declared in the `main.go` file nor in any of the packages that `main.go` imports. Indeed, `go run` will not look at files that weren't imported by the `main.go` file (why would it?). As a result, Go programs usually have a single file in the `main` package - the `main.go`. Alternatively, it is possible to `go run` a package or a directory, rather than a single file. In this case, all files in that directory or package will be used for execution, and, in our case, Go won't be angry at us anymore:

```
$ go run .
```

The output is gibberish but you can recognise the structure of the slice of bookworms:

```
[{Fadi [{Margaret Atwood The Handmaid's Tale} {Sylvia Plath The B
```

Not the best UI, but enough for debugging.

3.1.4 Test it

How do we make sure that this is going to work after future changes? Executing a command and checking the results, trying to see if the curly braces are at the correct position, isn't sustainable.

Let's write a test for this function. The `testdata` folder is the perfect place to hold various Json files with our different test cases. We are going to test an internal function that lies in the `bookworm.go` file, and for this reason, we will call our file `bookworms_internal_test.go` and write a test for `loadBookworms` named `TestLoadBookworms`.

The first step is to define the required parameters and returned values for our function. We will need the path of a file in `testdata`, the expected result, which is a slice of `Bookworm`, and because we also test the unhappy path, we will add whether we expect an error. For this chapter, we will not verify the exact type of the error but only the presence with a boolean value.

Each test case could be the purpose of a different function, but this strategy is rarely extendable. Instead, we use a map, where the key is the name of the test for humans to understand what we want to test, and the value is a structure with all the values specific to our test case. More on maps just after the test.

```
    type testCase struct {
        bookwormsFile string
        want          []Bookworm
        wantErr       bool
    }
    tests := map[string]testCase{
    }
```

To keep the writing of the `[]Bookworm` in test cases minimal, we can define books as global variables and reuse them over different tests. Global variables are usually not a good idea in code, but, in tests, we want the handy solution, especially because files names `*_test.go` are not accessible outside the package, even if you inadvertently name your variable with a capital.

```
var (
handmaidsTale = Book{Author: "Margaret Atwood", Title: "The Handm
oryxAndCrake  = Book{Author: "Margaret Atwood", Title: "Oryx and
)
```

Let’s now write the test for the successful use case. We’ll need a new JSON file, or we can reuse the existing one - as you prefer. Or you could nick those from our repo, we won’t file a complaint. We fill the expected result with the list of `Bookworms`, each bookworm having their name and the list of their books. Here’s an example:

```
"file exists": {
    bookwormsFile: "testdata/bookworms.json",
    want: []Bookworm{
        {Name: "Fadi", Books: []Book{handmaidsTale, theBellJar}},
        {Name: "Peggy", Books: []Book{oryxAndCrake, handmaidsTale
    },
    wantErr: false,
}
```

We can identify at least two error cases: first, if the pointed file does not exist and, second, if the formatting of the file is invalid.

Let’s invent a file path that does not exist. Can you fill in by yourself what will be the behaviour of `loadBookworms`?

```
"file doesn't exist": {
    bookwormsFile: "testdata/no_file_here.json",
    want:          nil,
    wantErr:       true,
}
```

As you can see, the expected result is nil as we expect an error since the opening of the file will fail early in the process, and we do want an error.

The second unhappy path that we can face is if the file is not valid, the format is not respecting JSON for example missing a bracket or a comma: in our test case, the file was truncated, and is therefore missing its closing `}`. Again, we want to verify the presence of the returned error. Create a file in `testdata` folder that has some invalid formatting and write the corresponding test case.

```
"invalid JSON": {
    bookwormsFile: "testdata/invalid.json",
    want:          nil,
    wantErr:       true,
}
```

Easy, isn’t it? We brushed over it in the previous chapter while writing table-driven tests, but in order to properly loop over the map, you will need to know more about loops.

Using loops in Go

All loop syntaxes in Go use the keyword `for`. All of them. Other languages might use `while`, `foreach`, `for`, etc. Let’s look at a few examples.

First, the classic `for`. Nothing quite out of the ordinary here. Count from a number to another number we know.

```
for i := 0; i < 5; i++ {
for i := 0; i < arrayLength; i++ {
for i := firstIndex; i < limit; i++ {
```

As a side note, Go differs from some languages with the postfix operators, `++` and `--`. In most languages, `i++` means “increment i by 1 and store that into i”.

Where Go is different from other languages such as C++ or Java in this syntax is that `i++` isn't a left value. It isn't comparable to anything. We can't write `i++ < 5` or `fmt.Println(i++)` in Go. This also implies that Go doesn't have a prefix operator - we can't write `++i` to mean "increase the value of i and return the increased value".

Next, the boolean expression, called *while* in some languages.

```
for iterator.Next() {
for line != lastLine {
for !gotResponse || response.invalid() {
```

Any boolean expression is valid at this place, just make sure you don't end up in an infinite loop.

If you do need an infinite loop on purpose, there is a way:

```
for {
```

Go on, forever. Usually, these contain either a return, a break, or something that will exit the program entirely.

Finally, when we need to iterate over the items in an array, a slice, a map, or a channel, `for` can be paired with the keyword `range`. In the case of a slice, for example, the list of bookworms we want to read from the file, `range` returns the index and a copy of the value at this index.

```
for i, bw := range bookworms {
```

At each iteration here, `i` will increase from 0 onwards to `len(bookworms)-1`, and `bw` is the same as `bookworms[i]`. The main difference is that `bw` is a copy, so if you make changes to it, there will be no change in the contents of the slice itself, which can be a good or a bad thing, depending on what you expect.

In the case of maps, like in our test, `range` simply returns a copy of the key and a copy of the value.

```
for name, testCase := range tests {
```

Before Go 1.20, the copies were made into the same variable, resulting in a few surprises when the variable was used in concurrent ways inside the loop.

If you don’t need one of the two parameters, you can ignore it in various ways. All of the following lines are valid, there is no difference for the machine between the 2nd and 3rd versions.

```
for _, bw := range bookworms
for i, _ := range bookworms
for i := range bookworms
```

Take the time to write the full test alone, then compare your solution to the one below.

Listing 3.8 bookworms_internal_test.go: Test LoadBookworms()

```
package main

import (
    "reflect"
    "testing"
)

var (
    handmaidsTale = Book{Author: "Margaret Atwood", Title: "The H
    oryxAndCrake  = Book{Author: "Margaret Atwood", Title: "Oryx
    theBellJar    = Book{Author: "Sylvia Plath", Title: "The Bell
    janeEyre      = Book{Author: "Charlotte Brontë", Title: "Jane
)

func TestLoadBookworms_Success(t *testing.T) {
    tests := map[string]struct {
        bookwormsFile string
        want          []Bookworm
        wantErr       bool
    }{
        "file exists": {
            bookwormsFile: "testdata/bookworms.json",
            want: []Bookworm{
                {Name: "Fadi", Books: []Book{handmaidsTale, theBe
                {Name: "Peggy", Books: []Book{oryxAndCrake, handm
            },
            wantErr: false,
        },
        "file doesn't exist": {...},
        "invalid JSON": {...},
    }
    for name, testCase := range tests {
        t.Run(name, func(t *testing.T) {
            got, err := loadBookworms(testCase.bookwormsFile)
            if err != nil && !testCase.wantErr { #A
                t.Fatalf("expected an error %s, got none", err.Er
            }

            if err == nil && testCase.wantErr { #B
                t.Fatalf("expected no error, got one %s", err.Err
            }

            if !equalBookworms(got, testCase.want) { #C
                t.Fatalf("different result: got %v, expected %v",
            }
        })
    }
}
```

What did you use to compare the expected bookworms and the returned ones?

The straightforward answer would be to write an `equal` function to compare the content of two lists of Bookworms. We will name the function `equalBookworms`, let’s see what it looks like in detail. First, the signature should be two lists of bookworms, we will name the one against which we want to check `target`. Because it is a utility function, we can specify to the compiler that it can be skipped and not print line information by adding `t.Helper()` at the beginning of our helper function. To do so, we need to pass `*testing.T` as parameter to equal. We will do the same for all the helpers in this chapter.

The content of the function consists of ranging over the bookworms and comparing each field, first the name which is pretty straightforward and then the books.

Listing 3.9 bookworms_internal_test.go: Helper to compare Bookworms

```
// equalBookworms is a helper to test the equality of two lists o
func equalBookworms(bookworms, target []Bookworm) bool {
    if len(bookworms) != len(target) {
        return false #A
    }

    for i := range bookworms {
        if bookworms[i].Name != target[i].Name { #B
            return false
        }

        if !equalBooks(bookworms[i].Books, target[i].Books) { #C
            return false
        }
    }

    return true #D
}
```

To compare the books, we can write a subfunction, `equalBooks`, encapsulating only the comparison of the books, which makes it easy to read and to reuse.

Regarding the implementation, do not forget that we can exit early by comparing the length of the two lists. Then we can range over the books and compare the two lists and return false if they are different.

Listing 3.10 bookworms_internal_test.go: Helper to compare Books

```
// equalBooks is a helper to test the equality of two lists of Bo
func equalBooks(books, target []Book) bool {
     if len(books) != len(target) {
        return false #A
    }

    for i := range books {
        if books[i] != target[i] { #B
            return false
        }
    }

    return true #C
}
```

Another way of doing it is by using the standard package `reflect` providing a simple but badly performing function to compare interfaces: `reflect.DeepEqual` which we will explore later in the book. It is not recommended for production code because it is not designed for performance but in our case, it will do the trick: less code to write is always a good thing.

```
if !reflect.DeepEqual(got, testCase.want) {
t.Fatalf("different result: got %v, expected %v", got, testCase.w
}
```

Now that we have read and parsed the input file into a Go structure, we are able to find which books are in more than one collection.

3.2 Find common books

Remember that the whole purpose of our tool is to find which books were read by both Fadi and Peggy, or other bookworms. In this section we will go through all the bookworms’ shelves, register books we find there, and then filter on those that appear more than once.

We will write a function for that: `findCommonBooks`. Let’s write its signature first. It takes the data we have, which is a list of bookworms and their collections, and returns the books in common, in the shape of a slice of `Book`.

Listing 3.11 bookworm.go: findCommonBooks() signature

```
// findCommonBooks returns books that are on more than one bookwo
func findCommonBooks(bookworms []Bookworm) []Book {
    return nil
}
```

How do we know that a book appears multiple times on shelves? Well, we need to count the number of occurrences of each book on all the bookworms’ shelves.

But is that enough, really? What should we do if a single person has the same book more than once on their shelf? We authors had a long conversation about that: does it ever happen? Who has multiple copies of the same book? It turns out that one of us has the same novel series in three different languages. Another has different editions of the same book. The third was surprised.

Anyway, what do we do? Let’s take for granted, for the time being, that each person’s list only has one instance of each book. It will slightly simplify the algorithm.

3.2.1 Count the books

How do we count all the books? We have access to a slice of bookworms, so we will start there: look at each bookworm’s collection, and “register” each book we find there. To “register” a book, for now, we can use a counter representing the number of times that book has been seen on shelves so far.

Maps in Go

Go offers a limited number of built-in types. Arrays, slices, maps, and, to a lesser extent, channels, are the core bricks that allow data collection. A map in Go is an unordered associative array that contains pairs of keys and values. Each key is associated with a single value (but two keys could have the same value). Maps are Go’s idiomatic way of creating collections of unique keys, as we’ll see in this chapter. A map’s key can be anything that is “comparable”. Think of it as “can we write key1 == key2?”. Even though, at first sight, it might be easy to think everything is comparable in Go, the hard truth is that not everything is. Slices, maps, and function types aren’t, and this means that any structure that contains a slice, a map, or a function type isn’t. We’ll face this soon enough.

Writing to a map

We’ll have to store data inside our map. In Go, associating the value `v` to the key `k` in the map is done with the following line:

```
myMap[k] = v
```

It’s as simple as that.

Reading from a map

In the same way that retrieving an item from a slice at index 3 is done with square brackets, retrieving an item from a map at key 3 looks exactly the same:

```
var slice []string
...
v := slice[3]

mapped := map[int]string{}
...
v := mapped[3]
```

The only difference is that a map will also return a boolean, telling you whether it found the key. In the case of the slice, you know that there is a value at index 3 as long as the length of the slice is at least 4 (yes, we still count from zero) - otherwise, you're facing an error leading to a `panic`. In the case of the map, if the key is not found, the returned value is simply the zero-value of its type - here, in the case of `map[int]string`, the empty string `""`, and the boolean is set to false.

```
v, ok := mapped[3]
if ok {...
```

Or in a more compact version, reducing the scope of the `v` variable to inside the `if`:

```
if v, ok := mapped[3]; ok {
    // do something with v
}
```

Now that we know how to access elements in a map, it's time to count the books.

Init counter

The counter would be saved in a map, where the key is the book and the value is a `uint`, an unsigned integer. Whereas it can seem strange to have multiple copies of the same book (apparently), having fewer than zero copies is downright impossible.

The built-in `make` function creates a map (or slice or channel) by allocating memory for it. If we knew in advance the number of distinct books, say 451, we could be explicit about the size of the map we need. This would slightly optimise the execution.

```
count := make(map[Book]uint, 451)
```

These 2 lines have identical behaviours, you can choose to be either explicit or concise:

```
count := make(map[Book]uint)
count := make(map[Book]uint, 0)
```

Let’s now fill up this counter.

We first need to iterate over our bookworms. We’ll use the `for` keyword that we described earlier. In this case, we will make use of the value of the iterator, and we don’t need the index. Let’s give our iterator a bit more verbose a name than `bw`, though.

```
for _, bookworm := range bookworms {
}
```

Inside this loop, we can loop with the exact same syntax over the books this person has read.

```
for _, book := range bookworm.Books {
}
```

The key of our map is a `Book` structure. Maps in Go can have some structures as their key. What matters here is that we need the structure to be hashable. As opposed to Java, Go’s compiler doesn’t need a `Hash` function - instead, it will know how to hash a structure in order to turn it into a valid key. Any type can be a map key as long as it is hashable. Types that aren’t comparable also aren’t hashable. A slice, for instance, isn’t hashable, and this means that any structure that contains a slice can’t be a map’s key - if you ever try to use a structure with a slice as a key, Go will let you know, with the nice compilation message “invalid map key type”.

Finally, we can increment the counter for this book.

```
count[book]++
```

But wait, we never set it to 0 in the first place. Should we not set the counter to `1` if it is absent and only use `++` if it already exists? Well, no, we don't have to. Here comes the beauty of the zero-value in Go.

See, `count[book]` returns the value in the map at the index `book`, or, if absent, the zero of the value type. Here the value type is `uint`, so it means `0`.

As in most C-like languages, `a++` is just syntactic sugar for `a = a + 1`. If you replace `a` with `count[book]`, we first retrieve the value from the map or get a zero if it doesn't exist, then add one and write back into the map in the same place.

This little counting logic is atomic enough that it would benefit from living in its own function. Let's call it `booksCount` and call it in `findCommonBooks()`. It should by now look like this:

Listing 3.12 bookworm.go: booksCount() function

```
// booksCount registers all the books and their occurrences from
func booksCount(bookworms []Bookworm) map[Book]uint {
    count := make(map[Book]uint) #A

    for _, bookworm := range bookworms { #B
        for _, book := range bookworm.Books {
            count[book]++ #C
        }
    }

    return count
}
```

Can we test it? It should be pretty straightforward.

Test it

Writing the test for this small function is not particularly tricky.

First, we can write a helper to compare the equality of two maps of books by verifying first that the keys in the `want` map are present in what we `got`. Let’s range over the `want` and call the key in `got`.

Listing 3.13 bookworms_internal_test.go: Helper to compare books count

```
// equalBooksCount is a helper to test the equality of two maps o
func equalBooksCount(got, want map[Book]uint) bool {
    if len(got) != len(want) { #A
        return false
    }

    for book, targetCount := range want { #B
        count, ok := got[book] #C
        if !ok || targetCount != count { #D
            return false #E
        }
    }

    return true #F
}
```

Note that in this version, nil and empty map are considered equal.

The helper is now written, let’s move to the longest part: think of all the test cases. The first test case we can think of is the nominal use case, and the second one is no bookworms at all. We can also have one bookworm without books, probably not a real bookworm or a very unhappy person.

Again, write the tests alone and compare your solution.

Listing 3.14 bookworms_internal_test.go: Test booksCount

```
func TestBooksCount(t *testing.T) {
    tt := map[string]struct {
        input []Bookworm
        want  map[Book]uint
    }{
        "nominal use case": {
            input: []Bookworm{
                {Name: "Fadi", Books: []Book{handmaidsTale, theBe
                {Name: "Peggy", Books: []Book{oryxAndCrake, handm
            },
            want: map[Book]uint{handmaidsTale: 2, theBellJar: 1,
        },
        "no bookworms": {
            input: []Bookworm{},
            want:  map[Book]uint{},#B
        },
        "bookworm without books": {...},
        "bookworm with twice the same book": {...},
    }

    for name, tc := range tt {
        t.Run(name, func(t *testing.T) {
            got := booksCount(tc.input)
            if !equalBooksCount(t, tc.want, got) { #C
                t.Fatalf("got a different list of books: %v, expe
            }
        })
    }
}
```

Launch the tests. Everything is passing? Good, this means we can use the `booksCount` function.

We can now add the call into the `findCommonBooks` function. Currently, you should have something like this:

Listing 3.15 bookworm.go: findCommonBooks() with booksCount() call

```
// findCommonBooks returns books that are on more than one bookwo
func findCommonBooks(bookworms []Bookworm) []Book {
    booksOnShelves := booksCount(bookworms) #A

    return nil
}
```

Note that it should not compile because we are not using the `booksOnShelves` variable for the moment. But it’s time to use it!

3.2.2 Keep higher occurrences

Now that we have counted the number of copies of each book on every bookshelf, the next step is to loop over all of them and keep those with more than 1 copy.

Let’s declare a slice that will contain all the books that were found multiple times in the collections of all bookworms.

```
var commonBooks []Book
```

```
return commonBooks
```

We could use the `make` built-in function again. How?

What is a slice, what is an array?

We keep using the word slice for a list of values of the same type. Many languages would simply call this an array, so what is the deal, is it just a fancy new word? You already know how to range over a slice, but there is a bit of theory required at this point.

The type `[n]T` is an *array* of `n` values of type `T`. For example `var a [5]string` declares a variable `a` as an array of five strings. An array's length is part of its type, so arrays cannot be resized. Very limiting. In real life, we practically never use arrays directly.

A slice, on the contrary, is a *dynamically-sized, flexible view into the elements of an array*, as described by the Go official website. The type `[]T` is a slice of elements of type `T`, built upon an array. As you can see, we don’t specify its size.

Slices have 3 fields that any developer needs to know about: the underlying array, stored as a pointer, the *length* of the slice, and its *capacity*. The length is the number of elements present in the slice, whereas the capacity is the number of elements that can be stored before a reallocation is necessary. You can get them via the `len` and `cap` built-in functions, and set them when you initialise the slice with `make`. Note that keeping the length as fields makes accessing this information an O(1) operation.

The last very useful thing you can do to a slice is appending an item by using the `append` built-in function.

Let’s look at some examples.

```
var books []Book
books = append(books, Book{...})
```

At first, the capacity and length are both 0, the slice is nil and the underlying array is not initialised. After we append, the number of items in the slice is one, so `len(books)` is 1, easy. The slice is not nil anymore. But trickier, the new array that we point to has a capacity of 1 element.

Note that `append` returns a slice. We cover what happens internally in the Appendix E, but, for now, the important message is that, when appending to a slice, it's always safe to override the extended slice with `append`'s output.

Another example of a slice initialisation, where we create a slice with a length of 5, a capacity of 5, and an underlying array of 5 zero-value books.

```
books := make([]Book, 5)
books[1].Author = "bell hooks"
```

All five books are all created and zeroed, which means we can access them directly and write into them.

Also, if we `append` a `Book` to this slice, it will appear in 6th position, after the 5 zero-value books already there.

Finally, if we know the final size of the slice but want to use `append`, we can specify both the initial length and the needed capacity.

```
books := make([]Book, 0, 5)
```

Now that we know everything about creating slices, let's look at the code we've written so far. Earlier, in section 3.1.3, we decoded the JSON message describing the bookshelves into a slice variable declared with the `var` syntax above. We didn't make a call to `make`, and therefore we didn't cause any allocation. The trick was that we passed the address of the slice to the `Decode` function, which could then fill it with values. We explore the mysteries of passing a slice by address or by copy in the Appendix E.

Range over a map

To fill the slice, we need to loop over the map `booksOnShelves` returned by `booksCount()` and check the value of the counter for each book. Books with a counter bigger than 1 were read by at least two bookworms - in our case both Fadi and Peggy.

```
for book, count := range booksOnShelves {
  if count > 1 {
     commonBooks = append(commonBooks, book)
  }
}
```

Below, the full code of the function `findCommonBooks`:

Listing 3.16 bookworm.go: findCommonBooks() implementation

```
// findCommonBooks returns books that are on more than one bookwo
func findCommonBooks(bookworms []Bookworm) []Book {
    booksOnShelves := booksCount(bookworms) #A

    var commonBooks []Book

    for book, count := range booksOnShelves { #B
        if count > 1 { #C
            commonBooks = append(commonBooks, book)
        }
    }

    return commonBooks
}
```

Test it

Testing this should be quite easy. We have some bookworms in and some books out. Some test cases are easy and you can write them without our help:

- Everyone has read the same books
- People have completely different lists
- More than 2 bookworms have a book in common
- One bookworm has no books (oh the sadness!)
- Nobody has any book (oh the agony!)

This is our version of the test.

Listing 3.17 bookworms_internal_test.go: Test findCommonBooks()

```
func TestFindCommonBooks(t *testing.T) {
    tt := map[string]struct {
        input []Bookworm
        want  []Book
    }{
        "no common book": {
            input: []Bookworm{
                {Name: "Fadi", Books: []Book{handmaidsTale, theBe
                {Name: "Peggy", Books: []Book{oryxAndCrake, janeE
            },
            want: nil,
        },
        "one common book": {...},
        "three bookworms have the same books on their shelves": {
    }

    for name, tc := range tt {
        t.Run(name, func(t *testing.T) {
            got := findCommonBooks(tc.input)
            if !equalBooks(t, tc.want, got) { #A
                t.Fatalf("got a different list of books: %v, expe
            }
        })
    }
}
```

Run your test. Now run it again a few times. Do you notice something strange?

3.2.3 Determinism

If you run your code a few times, you will see that the order of the output keeps changing. The output might not always be in the same order as we described the expected results in the slice `[]want`.

When we loop over a map, there is no guarantee from the Go language that the keys and values will be returned in a specific order. Depending on the situation, it can be better to be deterministic and always return the same result in the same order. It simplifies the testing, for one thing.

In this case, sorting the slice of books will make our lives easier. The `sort` package has a `Slice` function especially tailored for this situation: it takes a slice and a function. The function must return whether the item at the first of 2 indices must appear before the item at the second index. We can use an anonymous function defined at the level of this call, it is easier to evolve than naming it somewhere else. For now, we'll sort books by author first, and then by title. If at some later point, you want to sort by title first, and then author, here is the place.

As the sorting logic is not part of the algorithm to find common books, we prefer to put it in a different function, one that wraps the call to `sort.Slice`.

Listing 3.18 bookworm.go: Sort the books

```
// sortBooks sorts the books by Author and then Title.
func sortBooks(books []Book) []Book {
    sort.Slice(books, func(i, j int) bool { #A
        if books[i].Author != books[j].Author {
            return books[i].Author < books[j].Author #B
        }
        return books[i].Title < books[j].Title #B
    })

    return books
}
```

Note that the original slice is modified, the `sort.Slice` function does not create a sorted copy of the array. It can be a bad or a good thing, depending on the situation. This function's signature could be `sortBooks([]Book)`, with no return value. We'll cover the details of this in the Appendix E.

As we mentioned runes in the previous chapter, here is a disclaimer: using `<` to compare strings will look at the UTF encoding of the titles. Greek titles for example will therefore always appear after those written with Latin characters.

To use our sorting function, we just need to wrap the returning value of `findCommonBooks`:

```
return sortBooks(commonBooks)
```

Testing will now become way easier. You may need to fix the order of the expected results, but now you can automate your test and rely on it.

Our code is now tested and we can go back to `main.go` to print out the result.

3.3 Print

Back in the `main` function, we have loaded the data, and nothing more. Let’s call `findCommonBooks`. We now have a slice of books. How can we display that correctly? `fmt.Println` is nice but we need to loop over the collection of books.

Let’s write a function that prints out a list of books. You should be able to do it in five lines,

Listing 3.19 bookworms.go: displayBooks

```
// displayBooks prints out the titles and authors of a list of bo
func displayBooks(books []Book) {
    for _, book := range books {
        fmt.Println("-", book.Title, "by", book.Author)
    }
}
```

If you want to test this, you have to either write an `Example` or provide an `io.Writer`. We will see how to provide writers in the next chapter.

Listing 3.20 main.go: main function

```
func main() {
    bookworms, err := loadBookworms("testdata/bookworms.json")
    if err != nil {
        _, _ = fmt.Fprintf(os.Stderr, "failed to load bookworms: 
        os.Exit(1)
    }

    commonBooks := findCommonBooks(bookworms)

    fmt.Println("Here are the books in common:")
    displayBooks(commonBooks)
}
```

Before automating the test, let’s run the program manually.

```
go run .
```

Exercise 3.1: Use the flag from chapter 2 to pass the file path as a parameter to your program.

You already know how to write an `Example` test.

Listing 3.21 main_internal_test.go: testing main

```
package main

func Example_main() {
    main()
    // Output:
    // Here are the common books:
    // - The Handmaid's Tale by Margaret Atwood
}
```

Play around with different book lists!

3.4 Improvements

3.4.1 Exercise: reading recommendations

Now that you have gathered everybody’s data, you can do some cheap data analysis. Knowing what books your friend and you both read is a good conversation starter, but we can go deeper and write a program that will suggest a reading list from what other like-minded people read, and hopefully liked. Think of the section on the online bookshop: “other readers bought”, even though buying, reading and liking are three very different things.

In this section, we’ll go for a simplified approach: let’s consider that our bookworms only keep on their shelves books they appreciate. We don’t know what happens to other books - but we hope they are traded for even more books, or given to charity. We can assume, from now on, that books on a shelf are beloved and cherished, and, for this reason, we’ll take the shortcut of considering that, if Fadi has kept the same book on her shelf as Peggy did, she might be interested in what other books Peggy has kept on hers.

For one given target reader, we go through all other readers and compute a like-mindedness score, or similarity. Then, if the similarity is more than 0, we can add that score to each book that was not read by the target reader but was read by a similar person. Writing in code can sometimes be more clear.

Listing 3.22 recommendations.go: possible implementation

```
type Recommendation struct {
    Book  Book
    Score float64
}

func recommend(allReaders []Reader, target Reader, n int) []Recom
    read := newSet(target.Books...) #A

    recommendations := map[Book]float64{}
    for _, reader := range allReaders {
        if reader.Name == target.Name {
            continue
        }

        var similarity float64 #B
        for _, book := range reader.Books {
            if read.Contains(book) {
                similarity++
            }
            // you could also later extend to liked and dislike s
        }
        if similarity == 0 {
            continue
        }

        score := math.Log(similarity) + 1 #C
        for _, book := range readers.Books {
            if !read.Contains(book) {
                recommendations[book] += score
            }
        }
    }

    // TODO: sort by score
    // TODO: only output a certain amount of recommendations (n)
}
```

We are using math.Log, the function for natural logarithm, so that the score doesn't overpower all other suggestions when one person has too many similarities.

We need a type for the books read by our target. It must be able to quickly tell us whether it contains a book, and make sure that each book is only contained once.

```
type set map[Book]struct{}

func (s set) Contains(b Book) bool {
    _, ok := s[b]
    return ok
}
```

Storing in a map, as we saw, is the best (understand fastest) way to tell whether the list (of keys) contains a given value. We are using the empty struct rather than, say, a boolean, because a boolean takes up one bit of memory, and the empty struct takes zero. It means the map will not take up more memory than a slice of the same size.

Take your time to write the rest of the code, test it, and play around with it.

3.4.2 Implement sort.Interface

In order to sort the books in the output of the main function, we used the short sort.Slice, which takes a function as the sorting strategy. There is a second option that you may prefer.

Package `sort` provides `sort.Interface` that can be implemented to sort slices or user-defined collections. It becomes very handy when implementing custom sorting, in our case per author and title. The `sort.Interface` interface exposes 3 methods where elements are pointed by an integer index. Note that you need to fully implement the interface, meaning all three methods, even if you are not using all of them.

```
// Len is the number of elements in the collection.
Len() int

// Less reports whether the element with index i
// must sort before the element with index j.
Less(i, j int) bool

// Swap swaps the elements with indexes i and j.
Swap(i, j int)
```

As this only applies to collections, we add an intermediate custom type representing a collection of Books and implement the methods on it. This type is named after the way it sorts.

Listing 3.23 Implement sort.Interface on Books

```
// Books is a list of Books. Defining a custom type to implement
type byAuthor []Book

// Len implements sort.Interface by returning the length of the c
func (b byAuthor) Len() int { return len(b) } #A

// Swap implements sort.Interface and swaps two books.
func (b byAuthor) Swap(i, j int) {
    b[i], b[j] = b[j], b[i] #B
}

// Less implements sort.Interface and returns books sorted by Aut
func (b byAuthor) Less(i, j int) bool {
    if b[i].Author != b[j].Author { #C
        return b.LessByAuthor(i, j)
    }
    return b[i].Title < b[j].Title #D
}
```

The new function sortBooks can now call directly `sort.Sort` implementation. It’s important to notice that `sort.Sort` doesn’t return anything. Instead, it updates the slice’s contents, with the same elements, but in a sorted order.

```
// sortBooks sorts the books by Author and then Title in alphabet
func sortBooks(books []Book) []Book {
    sort.Sort(byAuthor(books))
    return books
}
```

You can find the full code in the repository.

3.4.3 Use bufio to open a file

The first step we achieved in this chapter was to read the contents of a file. Let’s have a closer look at how we did this.

We first opened the file, which returned a file descriptor, which implements the `io.Reader` interface.

```
f, err := os.Open(filePath)
if err != nil {...}
defer f.Close()
```

We then provided this reader to the `json.NewDecoder` function, and, from thereon, the magic happened in the `Decode` method of the `json` package.

```
var bookworms []Bookworm

err = json.NewDecoder(f).Decode(&bookworms)
if err != nil {...}
```

But do we really know how the reading actually happens?

Accessing files, either for reading or for writing, makes system calls. System calls are at the junction between our program and the operating system. System calls are expensive, and we usually want to reduce their number, which, fortunately, is controllable when reading or writing a file.

First, we need to understand the problem: how many system calls do we go through, when reading a file of size 10 MiB with our current implementation? The answer isn’t obvious, it’s hidden in the default buffer size of the file descriptor returned by `os.Open`, and we have no say in its value. Even worse than that - the `os.File` type is os-specific. So how can we improve this?

The answer lies in the `bufio` package. This package provides a `NewReaderSize` function that has the following description: *NewReaderSize returns a new Reader whose buffer has at least the specified size. If the argument io.Reader is already a Reader with large enough size, it returns the underlying Reader.* This means that, if we call `NewReaderSize` with any `io.Reader`, and give it a size of 1 MiB, we are guaranteed that the reading will happen by making system calls with chunks of 1 MiB. This way, we are able to tinker our program to have it behave exactly as we want. Note that finding the best size for the buffer of a reader isn't an easy task - giving it 1 GiB would obviously make system calls less frequent - but it would also consume 1 GiB of memory that - maybe - isn't fully used.

Let's make use of this nice feature, and decide that our buffer should be of "an average size" for a file, something in the megabyte range. Here's an example:

Listing 3.24 reading a file with a buffered reader

```
f, err := os.Open(...)
if err != nil { ... }
defer f.Close()

buffedReader := bufio.NewReaderSize(f, 1024*1024) #A
// bufio.Reader doesn't implement Closer
decoder := json.NewDecoder(buffedReader)
err = decoder.Decode(...)
```

Finally, the `bufio` package also offers an implementation of `io.Writer`. The latter is very useful when writing files, obviously, as it reduces the number of system calls drastically. Instead of writing line by line entries to a CSV file, we could be working with batches of 1 MiB.

But there is a very important point to keep in mind here: the `bufio.Write` method will only write data when its internal buffer is full. Most of the time, the last call to `bufio.Write` won't precisely fill the remainder of its buffer, and this last chunk could be lost! But don't panic, there is a way to flush the remaining bytes from the buffer to its destination, and it consists in making a simple call to `writer.Flush()`.

```
f, err := os.Create(...)
if err != nil { ... }
defer f.Close()

buffedWriter := bufio.NewWriter(f,1024*1024) #A
for _, data := contents {
  _, err = buffedWriter.Write(data)
  if err != nil { ... }
}

err = buffedWriter.Flush() #B
if err != nil { ... }
```

3.5 Summary

- The JSON format is commonly used when representing structured data. It can hold slices, maps, and objects.
- The `encoding/json` package exposes two ways of decoding a JSON message: the `json.Decoder` type, and the `json.Unmarshal` function. Most of the time, the `json.Decoder` should be used, as it can read from an `io.Reader` rather than from a complete slice of bytes. The encoding/json package also offers similar functions to encode a JSON message: `json.Encoder`, and `json.Marshal`.
- Fields of Go structures must be exported if something has to be decoded into them, so the decoder can access them.
- Test data should live in a folder named testdata, which will be automatically ignored by the `go` tool when compiling the code.
- All loops are `for` loop, the keyword can be followed by a variety of different syntaxes. The most common are `for i := min; i < limit; i++ {` and `for condition() {`.
- To loop over values stored in a slice or an array, use the `for index, value := range mySlice {` syntax. The `index` will start at `0` and its last value will be `len(values)-1`. If you don't really need the index, then you can iterate over the values with `for _, value := mySlice values {`. From time to time, you will see the following syntax: `for index := range mySlice {`. In this case, only the index is retrieved. This can sometimes be useful, as it doesn't cause each value of the slice to be copied.
- To loop over all pairs of keys and values stored in a map, we again use the for keyword: `for key, value := range myMap {`. If you are only interested in values, you can, just like we did with the slice, use `for _, value := range myMap {`. If, on the other hand, you are only interested in the keys of the map, then it's even shorter: `for key := range map {`.
- Slices are an abstraction of arrays. When in doubt, use slices.
- Sort slices with the `sort` package, either by giving a sorting function as

a parameter to `sort.Slice` or by implementing the package's `Interface`, which consists of a `Len` function, returning the length of the slice, a `Swap` function, which permutes two entries, and a `Less` function, that returns the comparison of two entries.

- A map's key can be an `int`, a `string`, or even a `struct`. When it's a structure, the structure can't contain a pointer, a slice, a map, a channel, a function, or a field of type `interface{}`. Indeed, these types cannot be compared with other values.
- If you want to discard repeated values from a slice - for instance, you want to have the list of bands that appear in your collection of CDs - you can use a `map[X]struct{}`. Iterate through your list and write into the map the elements you want to keep. In the music example, we'd write `myMap[cd.Band] = struct{}{}`. At the end of the loop, the keys of the map will be the unique values you had in your original list.
- When you want to know whether an element appears in a slice, and you know you'll do the operation so many times that you can't afford to iterate through the slice every time, you can use `map[X]bool`. Start by adding keys to this map by iterating over your list once, and set their value to `true`. After that, you can check whether an element is in the map by running `found := myMap[element]`. If it was initially in the slice, then the value returned is `true` - what we set it to -, and if the element wasn't in the slice, the value returned is the zero-value of a boolean - `false`.
- Remember to call `Close()` when you `Open` or `Create` a file. The `defer` keyword is your friend here.
- Use `defer` to keep together in the code lines that make sense together but need to be executed at different moments.
- Using a `bufio.Reader` or a `bufio.Writer` will reduce the amount of system calls your program executes. It will also make it simpler to count such system calls.
- When using a `bufio.Writer`, always keep in mind that the buffer needs to be `Flush`ed before the data can be considered fully written.

4 A log story: create a logging library

This chapter covers

- Understanding the need for a logger
- Implementing a 3-level logger
- Using an integer-based new type to create an `enum`
- Publishing a library with a stable exported API
- Implementing external and internal testing
- Understanding package-level exposition

The night is dark. Your colleague Susan and you have been working on trying to fix this bug for 2 hours straight. You don't understand what's happening with that `count` variable that should have the value `1`, but the result of the program seems to indicate that the value there is `2` instead. It’s late. You try to read the code, but the problem isn’t obvious. Is `count` really `2`? You decide to add a small line in the code, and relaunch everything, to get a better insight as to what's going on. The line you add will help you, at least, understand what the variable’s value is. You use:

```
fmt.Printf("counting entries, current value is %d\n", count)
```

We’ve all been there. Having the code say something we can understand at specific steps is our easiest way of following the program as it executes.

Then Susan notes - wouldn’t it have been nice to have this information from the initial run, without needing to redeploy an updated code? But then, would you also want to deploy this unconditional `fmt.Printf` (hint: the answer is an absolute no)? Were there other options that could have made your life simpler?

Debugging isn’t the only time when we want to know what’s happening in the entrails of our program. It is also valuable to inform the user that

"Everything is going extremely well". Or that something bad has happened, but the system recovered. Any trace of what’s happening might be useful - but that’s also a lot of messages, some aren’t as important as others.

Keeping track of the current state or events via readable messages is called logging. Every piece of tracked information is a log, and to log is the associated action.

What is a logger?

In computer science, a logger is in charge of noting down log messages (an activity called *logging). Historically, a log relates to a ship's logbook, a document in which were written records of the speed and progress of the ship. The logbook's name derives, itself, from a chip log, a piece of wood attached to a string that was tossed into the water to measure the speed of the vessel. The string had knots, at regular intervals, and measuring speed consisted in counting the number of knots that were unrolled during a given amount of time.*

Every application needs a logger, whose task is to write messages at specific moments in its execution so that they could be read and analysed later, if need be. Sometimes, we want these messages to be written to a file. Sometimes, to the standard output. Sometimes, to a printer, or streamed through the network to an aggregating tool, such as a database.

However, not all messages carry the same amount of information. *"Everything is going extremely well"* is very different from *"I just picked up a fault in the AE-35 unit"*. We might want to emphasise critical messages, or discard those of lesser matter. Acknowledging that there are different degrees of importance was already performed by scribes in Ancient Egypt, when they'd highlight specific sections of the text by writing them in red (this is where the word *rubric* originates).

This chapter will cover a specific need: write a piece of code that other projects can use. It is extremely common, as a programmer, to use existing code that we didn’t write. Think of it - it would be painfully tiresome to write over and over again simple functions, such as `cosine` or `ToUpper`, when they’ve already been written, thoroughly tested, and documented. Instead of copy-pasting code from other people, developers came up with the notion of "libraries" : code that one uses, but didn't write. In Go, libraries come in the shape of packages that we import. Go libraries are, of course, written in Go, and are made of (always) exported and (almost always) unexported types and functions. The exported part of the library (both functions and types) is called its application programming interface, which is always shortened down to API.

Now, let's write a library that anyone can use, and that you can reuse in any of your future projects. Susan will take care of the user code, and she will be interacting with our code via its API. First, we want to define the API - exported functions and types - and agree with any already identified user that it covers their needs. Then we will be able to publish the API, even before implementing the logic. Indeed, the sooner your library is out in the world, the sooner you can get feedback and improve it. Finally, we'll write the logging functions and test them.

Requirements

- A library that enables the user to log information of any type
- A library that makes available functions with signatures resembling that of `fmt.Printf`
- The user can set the threshold of importance for logging messages from their code
- The user can choose where logs are written

4.1 Define the API

Defining the way a caller interacts with a library is essential in making it stable and easy to use.

In order to make our users happy, the exported types and functions should be:

- easy to grasp - people don't want to spend hours trying to figure out how to use it. For this, making it small and simple is usually the better option: there should only be a single function to achieve each specific functionality

- stable - if you make evolutions, fix bugs or add functionalities, users should be able to take the latest version without changing their own code.

What we will export is an object that provides different methods to address criticality. For this, we will first define the different importance levels of logging. Then, our library will provide an object and a function to create it.

These tools will be implemented in a package.

Package summary

Go applications are organised in packages, i.e. collections of source files located in the same directory, each declaring the same package name. Go’s convention states that the name of the package is the name of the directory. As an example, we’ve used the `"fmt"` package in the previous chapter - it is located in Go’s sources, in a directory named `fmt`.

In Go, packages are the way we isolate the scope of functions and types. As previously mentioned, when a symbol’s name starts with a capital, it is visible outside the package. Exported symbols are available to users, the rest remains inside the package. If you know Java, the package has roughly the same level of importance as a Java class when it comes to what you can do with it, but it is close to a Java package in that it gathers together related types. What is public or exported should change as little as possible from version to version. This is especially important for packages intended to be consumed by other people. We want to preserve backward compatibility and avoid breaking our users’ code. If you want to improve performance or fix bugs, what is private or unexported can change.

Rules of a Go package

· A package is a collection of files located in the same folder that all share the same package name. Each Go file starts with the package declaration.

· It is customary to name the package after the name of the directory.

· Avoid overly long names if possible. Prefer a lowercase single word. If you compress the word, avoid abbreviations that make the package name more ambiguous.

· Any symbol (functions, types, variables, constants) starting with a capital letter will be exported outside the package and can be used by other packages, while those starting with a lowercase will not.

· camelCase and PascalCase are the conventions for functions, variables, constants and types. Package names stay lowercase.

Before you can start coding, don't forget to `go mod init learngo-pockets/logger` your module (see Appendix A.4 if you forgot how).

Go modules

A module is a collection of Go packages stored in a file tree with a go.mod file at its root. The `go.mod` file defines the module's module path, which is also the import path used for the root directory, and its dependency requirements, which are the other modules needed for a successful build. Each dependency requirement is written as a module path and a specific version.

Create a folder named `pocketlog`. The name of the package reflects its purpose. In there, create a file named `logger.go`. The name of the file should be explicit about its contents.

We add in there the `Logger` struct.

Listing 4.1 Logger empty struct

```
// Logger is used to log information.
type Logger struct {
}
```

Before we can start adding fields or methods to it, we need to think about the logging levels we want to support.

4.1.1 Exporting the supported levels

It is mandatory, when using a logger, to assign an importance level to a message. This is the task of the user, who has to think about the criticalness of the information that is about to be recorded. Loggers around the world have a wide variety of levels, which always follow the same pattern: they start with those of the lowest importance (usually "Trace" or "Debug"), and then they go up to (usually) "Error" or "Fatal". The number of different log levels varies from project to project, but these three are quite common:

- *Debug*: used by developers to help monitor any information - which way did a message go, how long did it take to process a request, what was the url of the request, etc. They’re usually used to print the contents of a single variable. In a production environment, we don’t print Debug messages;
- *Info*: used to track meaningful information, for instance “Payment of amount X from account Y received”;
- *Error*: used when something goes wrong, before we try to recover. Error logs are useful to help the maintainers investigate the source of problems, with messages such as “Database not responding”, “Processing request would cause a division by 0”.

We can declare these levels as an enumeration: they are a finite and defined list of possible values.

A matter of file size

Don’t be afraid to keep your files small. When a type is starting to support more and more methods, think about splitting them into multiple files: the scope for declaring methods on a type or accessing its unexported fields is the package in which this type is declared. You can split by usage, and business logic or keep exported methods together for example. Make sure reading your file does not get overwhelming. Incidentally, it reduces conflicts in your version control.

Create a file named `level.go`. Again, the first line of this file will be the package we’re developing: `package pocketlog`. Considering the targeted size of this new file, we could very well keep everything in the `logger.go` file, but we find it easier to open a file named `level` when looking for levels.

In this file, we declare a named type `Level`, of the underlying type `byte`. We could use `int32` as an underlying type as all we want is a number, but this would take 4 times more memory for no good reason. Other packages can use the type `Level`, as it is exported.

```
type Level byte
```

Wait! Experience and code reviews will tell you something is wrong. Any exported symbol requires a line of documentation - a commented sentence that starts with the name of the type, function or constant that you are documenting. Let's fix this.

```
// Level represents an available logging level.
type Level byte
```

Once we have a type for our levels, we can export them. Logging levels are constants that we declare as an **enumeration** - a finite list of entities of the same kind. This list of `Level`s belongs in the `level.go` file.

```
const (
   // LevelDebug represents the lowest level of log, mostly used
   LevelDebug Level = iota
   // LevelInfo represents a logging level that contains informat
   LevelInfo
   // LevelError represents the highest logging level, only to be
   LevelError
)
```

Enumerations

The syntax here is to use `= iota` to let the compiler know that we are starting an enumeration. `iota` allows us to create a sequence of numbers incremented on each line. We don't need to assign explicit values to these constants, the compiler does it automatically for us thanks to the iota syntax. iota can be used on any type that is based on an integer. The behaviour of iota is to increase on every line, which means we need to sort our levels by order of importance.

We now have 3 log levels; each one will have its own purpose. If we decide to add a level later, we will only need to add a line and not worry about renumbering everything. Feel free to add more such as Warn (between Info and Error) or Fatal (guess where).

4.1.2 Object-oriented Go ("GoOOP"?)

Object-oriented programming is a paradigm based on entities (the “objects”) that usually contain data (the “fields”, or “attributes”). OOP (Object-oriented programming) is a common paradigm amongst back-end languages - Java, C++, Python are amongst its most popular examples. But how about Go? The official documentation reads “*Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy*”. Go is indeed not natively an object-oriented language, but it has all of the necessary functionalities. Most of the principles that apply to object-oriented languages can apply to Go. Go has no inheritance, but don’t worry, it has other features that let you achieve similar goals, such as composition (more on that in a later chapter).

Let’s head back to the `logger.go` file. What we have created here for our logger is the definition of a *structure*. We want it to log lines of text at different levels. The user will then be able to pass this object as a dependency to any function that needs to log something.

Let’s take a look at two approaches that would fulfil the expectation of exposing methods on a variable `l` of type `Logger`, each with a signature similar to that of `fmt.Printf`:

- one method with a level parameter: `l.Log(pocketlog.Info, "message")` - in this case, the caller passes the level of log as the first parameter;
- as many methods as there are levels: `l.Info("message")` - in this case, the caller decides which function to call.

We picked the second option as we consider it clearer and simpler than the former, which requires a lot of dots and text before we reach the interesting part of the line. Remember that code needs to be easy to read.

To declare a method that can be called on an object, we use receiver methods: these functions are attached to an instance of a struct. A receiver method is a function that operates on the structure specified in parentheses before the function's name. Receiver methods can accept a copy or a reference - a pointer - to the structure. For the difference between these, check Appendix E. Since these methods operate on a `Logger` structure, the most intuitive place for them is in the `logger.go` file.

Listing 4.2 logger.go: Receiver methods

```
// Debugf formats and prints a message if the log level is debug
func (l *Logger) Debugf(format string, args ...any) {
    // implement me
}

// Infof formats and prints a message if the log level is info or
func (l *Logger) Infof(format string, args ...any) {
    // implement me
}
```

As required, we've used the same signature as the `Printf` method of the `fmt` package, a signature that developers are already accustomed to and that they know and understand. This is why we end the function name with the letter `f`.

Variadic functions

Sometimes, you want to pass a variable number of parameters to your function - none, exactly one, or more. The best approach in this case is offered by Go's variadic function syntax. The last argument of a function can be of the type `...{some type}`. When calling such a function, the user can provide any number of parameters - from 0 to too many. Inside the function, we can access the parameters as we would access elements from a slice, using the `[2]` notation.The most used variadic functions are the `fmt.Printf` ones, but we will see more examples before the end of this chapter!

Exercise 4.1: write the signature of the `Errorf` method on your logger. It should be a lot of copy pasting, but make sure you understand what you are writing.

Your `Logger` does nothing, but it can already be called from Susan's code. Before you publish it to her, who is jumping on her chair, impatient to use it in her service, there is one thing we want to add.

4.1.3 The New() function

At the moment, creating a new logger can be done in these two completely equivalent lines of code:

`var log pocketlog.Logger`	`log := pocketlog.Logger{}`

The former is explicitly defining a zero-value logger, the latter leaves room for initialisation, if we later want to add exported fields and give them a specific value. Picking one over the other is a question of how you think the code might need to change.

But as your logger evolves, there will be mandatory parameters, such as the threshold where it should start caring about messages. To gently convince users to stay up to date with evolutions, Go does not provide any constructor mechanism, but we can write a `New()` method that builds a new instance. People can still use the above syntax (you must make sure it is safe, as it will set every field of the structure to its zero value) but they should preferably not. We don’t need to specify `Logger` in the name of that function, because users will be calling the name of the `pocketlog` package first, making it clear that we are creating a new pocket logger. This way we avoid the stuttering `pocketlog.Logger` where `log` appears twice.

We add the `threshold` of the logger to the struct and define the `New()` method.

Listing 4.3 logger.go: Define a new object

```
// Logger is used to log information.
type Logger struct {
    threshold Level
}

// New returns you a logger, ready to log at the required thresho
func New(threshold Level) *Logger {
    return &Logger{
        threshold: threshold, #A
    }
}
```

You’ve probably noticed that the `threshold` field of the Logger is not exported. This is a decision we must make, whenever we declare a new field in a structure. When in doubt, don’t export: it is a lot safer than exposing everything. In this case, the user needs to define a logging threshold, and this is done via the `New` function. The internal structure of the logger is none of the business of our library’s callers.
We also made a second decision with this `New` function - we returned a pointer to a `Logger`, rather than a `Logger` itself. It is generally more useful that `New` should return a pointer. Think of it - the `new` built-in function in Go also returns a pointer, the bottom-line being that returning a pointer makes it easier for the caller to share the resource - the `Logger` - in their program.

Our logger still does nothing, but it can be used on Susan’s development branch and she won’t need to change anything while you make it work. You can commit.

Go’s zero-values

Every type in Go has a zero-value. This includes basic data types, struct types, functions, channels, interfaces, pointers, slices, and maps. Basically, any type for which you can declare a variable has a zero-value. The zero-value of a type is the value held by a non-initialised variable of that type. You can refer to Appendix C for details.

Exercise 4.2: What is the logging level of a logger defined with the `var log pocketlog.Logger` syntax?

4.1.4 And what about testing?

We just committed, but we have no test. This is subpar! Early is always the best moment to write a unit test.

How can we test this? We have a very clear definition of how the `Logger` should behave from the point of view of the user, but we don’t know much yet about how it will work internally. This is the perfect situation for closed-box testing, where we test a system from the outside. “Outside”, here, means “from another package”. We could test it from the same package, but we’d be able to access fields or functions that an external user won’t be able to access.

Here, we will start by creating a `logger_test.go` file: contrary to the previous chapter’s open-box tests, this one is not an internal test. As we want to test from the outside, the file will have to be part of another package, one different from the rest of the code, but still in the same directory, for consistency.

```
+ pocketlog/
|  |
|  + -- level.go #A
|  + -- logger.go #A
|  + -- logger_test.go #B
+ -- go.mod
+ -- main.go
```

Go will complain if we write two packages in the same directory, but there is an exception to this rule that allows for tests to be written close to the source code: we can have a `foo_test` package alongside a `foo` package. This is what we’ll use here:

```
package pocketlog_test
```

To access `pocketlog` functions, we need to import it:

```
import "learngo-prockets/logger/pocketlog"
```

From this `pocketlog_test` package, we only have access to what the package `pocketlog` exports - hidden functions, variables, constants, types, and fields of exported types aren’t accessible. As the logger is currently writing to the standard output, we can start with an `ExampleXxx` function to test it. We are testing the `Debug` method of the `Logger` struct, so the signature of the testing function is `ExampleLogger_Debugf`. We can optionally add details about the expected output or the test scenario after yet another underscore, i.e. `ExampleLogger_Debugf_runes` or `ExampleLogger_Debugf_quotes`.

Listing 4.4 logger_test.go: Test the standard output

```
func ExampleLogger_Debugf() {
    debugLogger := pocketlog.New(pocketlog.LevelDebug)
    debugLogger.Debugf("Hello, %s", "world")
    // Output: Hello, world
}
```

Run the test. It should be returning an error, because our `Logger` still does nothing. Fixing this error will be our next task. Then we can add test cases, because this one is not covering enough of the use cases.

4.1.5 Documenting code

An important part of exposing a library is to document it so that other people understand how to use it. Comments on exported functions, methods, structs or interfaces are extremely important - some IDEs will automatically show them as we hover over them. Tests are the second place where other users might look for advice as to how to use your library - sometimes, it’s even comments in tests that resolve the biggest mysteries.

We’ve already seen previously that a comment on an exported type or function should be a line starting with the type or function name as the first word of the line. It is also good practice to end the sentence with a period.

```
// New returns you a logger, ready to log at the required thresho
func New(level Level) *Logger {
```

`doc.go`: a special file

There is an unofficial convention to write a special file, in each Go package, that will describe the purpose of this package. Almost like a README, but intended for developers only. This file is named `doc.go`, and is called a package header. You’ll find one in most packages you use, if you venture in there.

The `doc.go` file contains no Go code, only one uncommented line: the package. And before that line, a verbose description of what the package is about. This is where we can tell how to properly use the package, in which order to call functions, and what we shouldn’t forget to `defer`, if need be.

The comments prior to the package name should be a multiline comment, where the first line starts with `Package pocketlog`, in our example. The capital matters, for code linters. Write this file, with every piece of information you deem important for the callers of our library. Here is our `doc.go` file:

Listing 4.5 doc.go: Documenting a package

```
/*
Package pocketlog exposes an API to log your work.

First, instantiate a logger with pocketlog.New, and giving it a t
Messages of lesser criticality won't be logged.

Sharing the logger is the responsibility of the caller.

The logger can be called to log messages on three levels:
  - Debug: mostly used to debug code, follow step-by-step process
  - Info: valuable messages providing  insights to the milestones
  - Error: error messages to understand what went wrong
*/
package pocketlog
```

The `go doc` command

One of the tools Go is shipped with is the `go doc` command. We’ve already mentioned it earlier to inspect the contents of standard packages. This command will give you the documentation of a package or symbol that the `go` command can find in the subdirectories. There is a minor limitation: `go doc` won’t go looking on the internet - it’s a local tool. This means that, in order to use it, you need to be working inside a project (with a `go.mod` file) for which the dependencies will have been downloaded - something that is achieved silently by some IDEs, but that can always be done manually with a `go mod download` command. In our case, we retrieve the documentation of the `pocketlog` package, and of the `New` function in the `pocketlog` package by running the following commands:

```
> go doc path/to/repo/pocketlog
> go doc path/to/repo/pocketlog.Logger
> go doc path/to/repo/pocketlog.New
> go doc path/to/repo/pocketlog.Logger.Debugf
```

In any case, documentation should always be part of what you deliver. It can take the form of comments, examples or of package headers. See more about it here: https://go.dev/doc/comment.

Now that we've explained how to use our library, it's high time to make it usable!

4.2 Implement the exported methods

In the previous section, we've published the API to Susan, and we've written some failing tests. The next step, to comply with the expectations, is to decide where the logger is going to do its deed and finally log. Should it always write to the standard output, and never the error one? How will it send a message to a ticket printer? Write in a file? Send via the network to a different aggregating mechanism? Should we implement this functionality for every possible use case?

As we want this package to be usable in any situation, we will leave the implementation of having a bespoke writer to the user's discretion. But Susan wants a default implementation that just spits out on the console (the standard output), so that she can focus on her business logic before improving her logging and monitoring (we may disagree on the priorities here, but the Products team insisted). So, `stdout` it is. We will improve it shortly after.

4.2.1 Default implementation

The first implementation is really the easy part. Think about how you would write the `Debugf()` method before spoiling your pleasure with the following solution. Remember that `Debugf()` should only log if the threshold level is Debug or lower.

Listing 4.6 logger.go: Debugf's default implementation for the console

```
// Debug formats and prints a message if the log level is debug o
func (l *Logger) Debugf(format string, args ...any) {
    if l.threshold > LevelDebug { #A
        return
    }
    _, _ = fmt.Printf(format+"\n", args...) #B
}
```

When calling the `Debugf` function, the user expects the message to be printed if the level of the logger allows for it. This means the first thing to do in this function is to make sure that we should be logging a message. The enum we declared for the levels allows us to compare two levels together, since the underlying type of the `Level` type is an integer.

This method could be just 3 lines if we chose to log inside the if and invert the condition, but always prefer to align the happy path unindented. Deal with errors and early exits inside your if blocks and keep real business logic as left as possible. This helps a lot when reading the code and makes extending it way easier.

Once we're sure we need to handle this message, let's log it. For now, we'll use the `fmt.Printf` function. This whole library might look like a verbose wrapping of this `Printf` call, but rest assured, there's more to it than meets the eye.

Exercise 4.3: Implement `Info`, `Infof`, `Error`, and `Errorf` levels methods. Also, implement all the other levels that you chose to add.

Now your previous test should be green. Let's discreetly postpone the testing of the other methods: we want to write `TestXxx` methods, which give more flexibility, so we need to write to non-standard outputs.

4.2.2 Interfacing

Writing bytes in various places is an extremely common use case in all computer programs. Writing json to an HTTP output, ones and zeros to a network router, bits into a digital port to turn a light on, encoded pixels to a printer or letters on a console, everything we do has to output its result somewhere in order to be useful at all.

Go has a set of standard interfaces for the most common uses, so that everyone who produces code that writes can match the same format and leverage intercompatibility.

io.Writer

Among the most commonly cited interfaces in the standard library, the `io` package holds the two famous `io.Writer` to write to any destination and `io.Reader` to read from any source (e.g. an array of bytes, a file, a json stream).

Here are the declarations of these interfaces :

Listing 4.7 `io` interfaces

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

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

We want the user of our logger to define the destination. We can ask for an `io.Writer` and simply write into it. They will be responsible for providing an implementation of their choice.

Implicit interfaces

One major difference between Go and the most-known Object languages (such as Java and C++) is that, in Go, interfaces are implicit. In order to implement an interface, just add the methods that your interface defines to an object and the compiler will recognise it.

We can already add the output to the structure, and the standard `Writer` to our `New()` builder.

Listing 4.8 logger.go: Add output field to the struct

```
// Logger is used to log information.
type Logger struct {
    threshold Level
    output    io.Writer
}
```

```
// New returns you a logger, ready to log at the required thresho
// The default output is Stdout.
func New(threshold Level, output io.Writer) *Logger {
  return &Logger{
    threshold: threshold,
    output:    output,
  }
}
```

Note the slightly enhanced documentation. Alternatively, it could default to os.Stderr, which represents the default error output.

Last but not least, each of the methods will need to use this new field. And let’s make sure that users who don’t follow our recommendation of using `New` to create a `Logger` don’t get nil pointer exceptions!

Listing 4.9 logger.go: Add output field to the struct

```
// Debug formats and prints a message if the log level is debug o
func (l Logger) Debugf(format string, args ...any) {
  // making sure we can safely write to the output
  if l.output == nil {
    l.output = os.Stdout
  }
  if l.threshold <= LevelDebug {
    _, _ = fmt.Fprintf(l.output, format, args...)
  }
}
```

In Go, the underscore symbol represents the Void. In other words, assigning a value to it will just discard the result.

What is the point? Go likes to be explicit. Here we explicitly say to the next developer, including future-us: I know there are values returned by this function, but I do not need them.

Here, the function returns an `int`, the number of written characters, and sometimes an error. There is nothing we want to do about that error at the moment, so we explicitly ignore it.

4.2.3 Refactoring

You might have noticed when implementing the `Info` and `Error` methods, that we're calling the same function `fmt.Fprintf` as our writing function. You might also have noticed that, as opposed to its sibling `fmt.Fprintf`, `fmt.Fprintln` appends an end-of-line character at the end of the string. In a way, the `printf` function allows you to do very fine craftsmanship, while the `println` function won't let you format exactly everything as you'd like it: no left-padding before numbers or strings, no hexadecimal representation of numbers, etc.

A new line is the guarantee that your log messages will be easily distinguishable in a console. As we want to export the `Printf` toolbox, we must add an explicit `\n` when we write the messages.

In our case, we need to add that new line three times, once in each of the functions. And whenever we want to make a change to the log message - for instance, add the logging level - we need to write the same lines three times. Should `Warn` or `Fatal` also be implemented, the count goes even higher. This (loudly) calls for a minor refactoring - we don't want to maintain the same code more than twice. Let's group all of them together and alter a single line every time we want to adapt the logger.

Create a `log()` method on the `Logger`. For now, it will have the same arguments as `Debug`, `Info` and `Error`. These three will call it: `log` is now the one method responsible for formatting and printing. The other three, the exported ones, are responsible for their log level and nothing more. There is zero good reason to export this one.

Listing 4.10 logger.go: Refactoring the logging methods

```
// Debugf formats and prints a message if the log level is debug
func (l *Logger) Debugf(format string, args ...any) {
    if l.threshold > LevelDebug {
        return
    }

    l.logf(format, args...)
}

// logf prints the message to the output.
// Add decorations here, if any. #A
```

```
func (l *Logger) logf(format string, args ...any) {
    _, _ = fmt.Fprintf(l.output, format+"\n", args...)
}
```

Run the tests. They should fail by now, as we haven’t updated them with the `os.Stdout` parameter for the `New` function. Once this is done, you can commit, and inform your colleague that the code is ready to be used. However, Susan tells you that her logs should be written to a specific file rather than on the standard output, because she already makes use of the standard output. Can the `Logger` achieve that by itself ?

4.3 The functional options pattern

Default values are a subject of high debate among the development theorists and theologists. Some people love them because it makes everything discoverable and therefore easier to bootstrap. Some people hate them because they don’t force you to know what you are doing. We believe default values are a good thing if used sparingly.

When we start developing, say, a new web service, we want to focus on the business logic. We want to make sure our logic works locally before making your service production-ready. In order to decrease the cognitive load, we start with the default version of the logger - architecturing a better logger can come later. Before writing the deployment code, we don’t need anything but the standard output.

But very quickly, we deploy to a cheap cloud provider so that we can pitch our prototype and show it to the world. Reading the standard output is not so trivial anymore. We pick a tool, like an aggregating database, that happens to publish a Go driver. The developers of this driver were smart enough to have a structure in their library that implements the `io.Writer` interface.

We still want to keep this default implementation, writing to the standard output, but we want to provide the option of writing somewhere else. One common way of doing this is by using the functional options pattern.

4.3.1 Create configurations

Create a new file, `options.go`, where our configuration functions will be written. Define a type of functions that can be passed to the `New()` function and that will be applied one after the other.

```
// Option defines a functional option to our logger.
type Option func(*Logger)
```

This function takes a pointer on our logger so that it can change it directly: in our case, change the default output to whatever the user gave us.

Listing 4.11 options.go: Optional function to change the output

```
// WithOutput returns a configuration function that sets the outp
func WithOutput(output io.Writer) Option {
    return func(lgr *Logger) {
        lgr.output = output
    }
}
```

This type of function can be passed to the `New()` function as variadic parameters: a list of zero or more arguments of the same type.

Listing 4.12 logger.go: Apply options

```
// New returns you a logger, ready to log at the required thresho
// Give it a list of configuration functions to tune it at your w
// The default output is Stdout.
func New(threshold Level, opts ...Option) *Logger {
    lgr := &Logger{threshold: threshold, output: os.Stdout}

    for _, configFunc := range opts {
        configFunc(lgr)
    }

    return lgr
}
```

Next time you want to add an option to your logger (e.g. a date formatter), just create a new `Option` and you’re set. There is an important point to notice here: adding configuration functions is quite easy, and lets the user set specific behaviours without altering the API of our library. Our `New` function accepts as many configuration functions as the user needs, from the list we implement in this package.

Usage example

Susan wants to know how to use your library. There is a documentation file, but human interaction is always so much more efficient. You write a small example and send it to her.

Outside of the library, init a new module and create a main.go file. Define a func `main()`, as you did in the previous chapter. In this function, instantiate a new logger and call a few methods to showcase your work.

Listing 4.13 main.go: Usage example.

```
package main

import (
    "os"
    "time"

    "learngo-pockets/logger/pocketlog"
)

func main() {
    lgr := pocketlog.New(pocketlog.LevelInfo, pocketlog.WithOutpu

    lgr.Infof("A little copying is better than a little dependenc
    lgr.Errorf("Errors are values. Documentation is for %s.", "us
    lgr.Debugf("Make the zero (%d) value useful.", 0)

    lgr.Infof("Hallo, %d %v", 2022, time.Now())
}
```

4.3.2 How to test that thing

We are already using the logger, but it is not fully tested! How unprofessional! Susan can use our library, but we don’t want her to come back with possible bugs.

The magic of interfaces means we can write a test helper that implements `io.Writer`, and give it to our `Logger` under test.

Test helper implementation

At the end of `logger_test.go`, write a new `testWriter` struct. Make it implement `io.Writer`, but instead of writing to a destination, it validates the output string. For example, you can keep a field in the struct where you concatenate the output, and you can validate that against the expected result.

Listing 4.14 logger_test.go: test helper implementation

```
// testWriter is a struct that implements io.Writer.
// We use it to validate that we can write to a specific output.
type testWriter struct {
    contents string
}

// Write implements the io.Writer interface.
func (tw *testWriter) Write(p []byte) (n int, err error) { #A
    tw.contents = tw.contents + string(p) #B
    return len(p), nil
}
```

This structure can be passed to the functional option higher in our test. At the end of the test, we can then check that the writer's contents are what we expect.

In practice, `strings.Builder` or `bytes.Buffer` can be used instead of the testWriter. Now you know how to do a mock in case the interface you need is not standard.

Update the test

Now that we are not forced to check the standard output anymore, we can write a `TestXxx` function, one that will test all of the logging methods together, sequentially. We can have one test case per required logging level and check that the outputs are different and the `Debugf()` call is mostly ignored.

Listing 4.15 logger_test.fo: Test function

```
const ( #A
    debugMessage = "Why write I still all one, ever the same,"
    infoMessage  = "And keep invention in a noted weed,"
    errorMessage = "That every word doth almost tell my name,"
)

func TestLogger_DebugfInfofErrorf(t *testing.T) {
    type testCase struct {
        level    pocketlog.Level
        expected string
    }

    tt := map[string]testCase{
        "debug": {
            level:    pocketlog.LevelDebug,
            expected: debugMessage + "\n" + infoMessage + "\n" +
        },
        "info": {...}, #B
        "error": {...,
    }

    for name, tc := range tt {
        t.Run(name, func(t *testing.T) {
            tw := &testWriter{}

            testedLogger := pocketlog.New(tc.level, pocketlog.Wit

            testedLogger.Debugf(debugMessage)
            testedLogger.Infof(infoMessage)
            testedLogger.Errorf(errorMessage)

            if tw.contents != tc.expected {
                t.Errorf("invalid contents, expected %q, got %q",
            }
        })
    }
}
```

Exercise 4.4: The test, as we have written it here, only tests the calls to functions in one order: `Debugf`, then `Infof`, then `Errorf`. What if we decide to add a buffer, and only think about writing everything in the `Errorf()` method? We will not see it in this situation, and Debug and Info messages might stay stuck.

Your logger is ready, fully functional, documented and tested. The rest of the company starts using it. Yet you keep dreaming up new functionalities for it.

Let’s explore a few and see where they lead.

4.4 Further functionalities

This tool offers endless possibilities for optimisations. The only limit, as always, will be the amount of time we’re ready to spend on it. For example, this library is not thread-safe: when multiple goroutines use the same `Writer` without any protections, the outcome can be unexpected - we will explore solutions to that in later chapters. In the meantime, let’s have a look at a few interesting enhancements we can add to our `Logger`.

4.4.1 Log the log level

Your service runs locally, with the lowest possible level of logs. You know everything that happens just by looking at your console. But now you would like to see the errors in red. You add an old awk command to your log tailing, but how do you know what to colour?

How can we know while reading the logs which message has which level? Well, let’s add that as an exercise.

Exercise 4.5: Add the log level to your output. Hint: change the contents of the `format` variable before printing, as we did when we added the current time.

4.4.2 Exposing the generic logging function

We chose from the start to export as many functions as there are logging levels, for it makes the user’s code easier to read.

Now imagine yourself in a situation where you only know what level to pick at runtime. Imagine you are logging the email address of your app’s user, but on one platform all the users are internal and the admins need to know who did what, and on another platform email addresses are covered by data protection laws and should not appear in logs, even in case of errors. You choose to have this information in your app’s configuration and would like to pass it directly to the logger. But you can’t change the logging level for the whole application, as this might discard some of your unrelated and important messages.

For this, we can add an exported `Logf()` function that takes a logging level as its first parameter.

Listing 4.16 logger.go: Exported Logf function

```
// Logf formats and prints a message if the log level is high eno
func (l *Logger) Logf(lvl Level, format string, args ...any) {
    if l.threshold > lvl {
        return
    }

    l.logf(lvl, format, args...)
}
```

From there, why not refactor so that all the other exported methods simply call this one? Of course, having both options will make your APIs more cluttered and harder to understand and maintain. And the user can always write her own function with a simple switch, using a variable defined in her own domain.

Exercise 4.6: Add a test for this method.

4.5 Logging: good practices

Now we've completed this library that you'll hopefully use and share, some recommendations might be necessary. What kind of information do we want to log? At what frequency do we want to log? There are two factors to take into account here: this is a story of compromise.

Logs are the trace of the past execution of a program, and as soon as the need for logs arise (usually "what was the value of `count`, at this moment?", "how long did this request take to process?"), the need to safekeep these logs also appears on the stage. Logs, when stored in a file, a database, a bucket in the cloud, or anywhere persistent, bluntly, cost money. The more logs you have, the easier it will be to understand what went wrong and quickly fix it - but all the more expensive it will be.

Every company will have its policy regarding what should - and shouldn’t - be logged, and the level at which they should be. However, here are a few recommendations we can share.

Write clear messages

Although it might be very tempting to log “Step 1”, “Step 2”, etc. inside a function, these messages won’t help you on the next day. Think what happened at step 1 - was the document inserted in the database? Was the email sent? Help your future self with clarity in messages. When a function has only one possible execution, the only value of a log message is the comforting reassurance that we’ve finished this or that piece of logic. Some valuable information here would be to know how long it took to process it, or something similar.

Avoid long messages

The amount of data written to the logs is directly related to the amount of money that will be spent to keep these messages. If your variable is a map with potentially thousands of keys, printing the map will be costly. Instead, wouldn’t having its size, or whether a specific key is present, be as valuable? If your data is a piece of an image or a song recording, the logger is not the place to keep a copy of the bytes that are being processed. Instead, write a function to save the image or the song.

Exercise 4.7: Ensure the logged message doesn’t exceed 1000 characters (or 1000 bytes, up to you), or, better, a value that would be optionally set. If it would and the limitation is activated, trim the end off to make sure all logged messages have a reasonable size.

Log at milestones, give heed to loops and recursion

Functions can be complex and span over hundreds of lines. When this happens, the most important question is to identify which sections really deserve a log and which simply don’t. If we return early because we found no item to process, should we say so? Maybe not, but we could always log the number of items found instead.

When retrieving items from a database, and processing everyone in a loop, do we want to know that we’re at item 5.567 out of 81.543? And 5.568? If the normal flow doesn’t require this level of detail, maybe we can simply log a message every 10.000 items, to get a rough idea of how much of this big list of data has already been processed.

Get to trust the code

Logging shouldn’t be a tool to debug the code. Most of the time, when you scratch your head wondering what’s going on when the input of this or that function has this or that value, it’s because the code is unclear. There are three ways of addressing this:

- Writing clearer code - splitting logical blocks in more smaller functions;
- Writing better documentation - comments, variable names, etc;
- Writing more tests - errors don’t happen on the happy path, they happen when something is wrong. Make sure you cover as many edge-case scenarios as possible.

Structured messages

In the recent world, most logs are, in fact, not processed by humans. They are mostly read by programs that use the logs to generate information displayed in dashboards - for instance, representing the number of errors that happen per minute over the course of time, or the time it took to process a request.

For this, we need to tell these computers how to parse the logs - which piece of information is valuable, which is not to be taken into account, etc. And the simplest solution, here, is to format the log messages into structured entities. A common structured log message format is JSON (displayed here on several lines for readability by humans):

```
{
   "time": "2022-31-10 23:06:30.148845Z",
   "level": "warning",
   "message": "platform not scaled up for request"
}
```

Some existing logging libraries offer several additional features, such as letting the user include their own set of keys and values to the logged message.

Exercise 4.8: (The concepts here are explained in Chapter 5.) Update the log function to print logs of the format above (you can ignore the `"time"` part now, as we need a bit more than what we've covered to validate it in the tests). This will require making use of the `encoding/json` package and the `Marshal` function it contains. This function will have to be called on a new type, which we'll define as a structure that contains a `Level` and a `Message`. One of the most common traps, when using the `Marshal` (or `Unmarshal`) function, is to forget that the json package needs to access the fields of the structure. It is tempting, for newcomers, to keep these fields unexported, but this makes them precisely unexported to the functions in charge of reading / writing them. We'll have more opportunities to cover this when we start implementing services.

4.6 Summary

- A library is a list of exported types and functions, the API, in a package, which a client can use out of the box.
- A library must only export what the user needs.
- Using explicit names, and reproducing existing signatures to help the caller of your library.
- Define domain types over primary ones for readability and maintainability of the code.
- Enumerations using `iota` are perfect for a type that has a small and finite number of possible values.
- Creating a `New()` method enables you to force the necessary parameters at the object initiation and guarantee the client of your library will use it properly, as it forces a clean initialisation of your object.
- Use the functional options pattern to set fields that have a default value.
- Implement and test the library using closed-box testing.
- Smaller files make your code easier to read.
- Be mindful of what you want to log and what not.

5 Gordle: play a word game in your terminal

This chapter covers

- Building a game that runs in a terminal
- Retrieving runes from the standard input
- Getting a random number in a slice
- Propagating errors
- Reading the contents of a text file

CLAUDIO. One word, good friend. Lucio, a word with you.

LUCIO. A hundred, if they'll do you any good.

This chapter is about a love story. During the 2020 pandemic, Mr Wardle, a passionate software developer, created a new game named Wordle for his partner Ms Shah, a word-game addict. After introducing the game to his relatives and seeing how well it was welcome, he decided to publish it. This is how this famous game began its journey before going public and rising like a rocket. There is now a daily release of a new word to find, mostly referenced throughout the world as “today’s wordle”. Since then, there have been lots of variations, based on geography, maths, terminology from Shakespeare, Tolkien, or Taylor Swift, and even more adaptations in different languages throughout the world (beyond time and space - a list offers ancient Greek, Quenya, and Klingon).

The game is pretty basic: you must guess a word of 5 characters in 6 attempts. After each attempt, the game tells you, for every character, whether it belongs to the solution, and whether it has the correct position.

The goal of this chapter is to create our own game named Gordle (did you get the pun?). It will be a configurable version of Wordle - the official version has 5 characters per word, but we can imagine passing longer or shorter words, and changing the number of attempts before a player’s game is over. Lucio will be our developer, while Claudio, the player, will execute a command that will start the game. In our code, we will progress step by step, starting with a simple function reading from the input and printing Claudio’s attempt. Then, we will iterate and have it evolve to give feedback to the player. Adding a corpus (a list of words) from where to pick a random solution will make the game more replayable. Finally, we will have the opportunity to support more languages and tweak the parameters as we want.

For the sake of simplicity, this version will only support writing systems where one character never needs more than one code point in Unicode. Supporting other writing systems is out of the scope of this chapter, but you can find extending ideas in the extras at the end.

Requirements

- Write a program that picks a random word in a list
- Read the guesses of the player from the standard input
- Give feedback on whether the characters are correctly placed or not
- The player wins if they find, or loses after the maximum number of unsuccessful attempts

5.1 Basic main version

The default approach to any coding exercise is always to simplify the problem to its absolute simplest version. We’ll have time to improve it later. First, let’s start with a basic version of the main function that will have a hardcoded solution. We’ll allow only one guess, and the program will answer ok or not ok.

As we did in previous chapters, we first initialise our module.

```
$ go mod init learngo-pockets/gordle
```

Then we create a new package named after the game `gordle` next to the `main.go` file. This package is where we’ll implement the game. Our project will have the following structure:

```
.
├── go.mod
├── gordle
│   └── files in package gordle
└── main.go
```

The package `gordle` will expose a structure `Game` to which we will progressively add the needed methods to build a full game.

5.1.1 Mini main

Lucio begins simply, with an empty structure named `Game` in the `gordle` package. We know that this program will need more than 50 lines of code, so it's a good idea to split responsibilities over several files. Anything that relates to the game will be in the `gordle` package. We know, for instance, that at some point we'll have to keep the secret word somewhere. This leads us to the creation of a structure that will contain game information.

Listing 5.1 game.go: `Game` structure

```go
// Game holds all the information we need to play a game of gordl
type Game struct{}
```

As we've seen in Chapter 4 with the logger, there are several ways to create an object. Here, we expose a `New()` method, which will be the recommended entry point into the library, guaranteeing the creation of the `Game` object with all its dependencies. Note that it is a good habit to ensure proper behaviour of your library.

By convention, `New()` will return a pointer on `Game`, similarly to Go's built-in function `new`, which returns a pointer too.

Listing 5.2 game.go: `New` to create a `Game` structure

```go
// New returns a Game, which can be used to Play!
func New() *Game {
    g := &Game{}

    return g
}
```

We voluntarily did not write `return &Game{}` because we will add some code before this `return g` line to configure our game.

Then, we attach a `Play` method to the `Game` type. `Play` will run the game. In our first implementation, let’s simply print the instructions. Creating a method on an object, in Go, is achieved by writing a pointer receiver on the `Game` structure.

Listing 5.3 game.go: `Play` to run the game

```
// Play runs the game.
func (g *Game) Play() {
    fmt.Println("Welcome to Gordle!")

    fmt.Printf("Enter a guess:\n")
}
```

This is enough for a very first version. Let's call these new methods in the `main` function. For this, we need to import the `gordle` package in the `main.go` file.

```
import (
   "learngo-pockets/gordle/gordle"
)
```

In the `main` function, we need two steps to start the game: create a new Gordle game and launch it!

```
func main() {
   g := gordle.New()
   g.Play()
}
```

The aggregated main file looks like this:

Listing 5.4 main.go: main function and package

```
package main

import (
   "learngo-pockets/gordle/gordle"
)

func main() {
   g := gordle.New()
   g.Play()
}
```

After these initial steps, we can run our program and verify that it behaves as expected. Now is also a good time to commit these files to your favourite version control system, before you add some contents into the `Game` structure. We are now ready to wait for Claudio’s guess of a secret word!

5.1.2 Read player’s input

Since this is a game, we have a player, Claudio. Let’s ask him for a suggestion. Claudio has access to the keyboard, and we’ll be reading his attempts through the standard input. After reading it, we can check it against the solution.

Game structure

There are several ways of reading from the standard input, depending mostly on what we want to read. Some functions read a slice of bytes. Some read strings. In this case, the player will type characters and then press the Return key. We, therefore, want to read a line until we hit the first end of the line character.

The `bufio` package has a useful method to achieve this on its `Reader` structure: *“ReadLine tries to return a single line, not including the end-of-line bytes”*, reads the documentation. It also states that it’s not the best reader in the world for most reading use cases, but for one word from the standard input, it is perfect. The good thing is that the `bufio.Reader` implements the `io.Reader` interface! We don’t want to over-engineer our solution.

Our `Game` object will hold a pointer to a `bufio.Reader`. Why a pointer and not a simple object? Simply because the `bufio` package exposes a `NewReader` function that returns a pointer to a `bufio.Reader`. Also, since we’ll be calling `ReadLine` a lot, it’s useful to immediately have a variable of the type of that method’s receiver - a pointer.

Listing 5.5 game.go: `Game` with a reader

```
// Game holds all the information we need to play a game of gordl
type Game struct {
    reader *bufio.Reader
}
```

But wait, how do we initialise this reader? Should we do it as part of the `New()` function? Although this is a valid option, we soon realise that `NewReader` itself requires a `io.Reader` parameter - where should that parameter come from? Since `io.Reader` is a very simple interface, we can pass a variable implementing it to our `New` function, and create the `bufio.Reader` inside `New`:

Listing 5.6 game.go: `New` with the reader as parameter

```
// New returns a Game variable, which can be used to Play!
func New(playerInput io.Reader) *Game {
    g := &Game{
        reader: bufio.NewReader(playerInput),
    }

    return g
}
```

Before we dive into reading a player’s input, we need to answer a question: how does Go deal with characters?

If we want to play using another language, or even play in English with words that come from another language, we need Unicode for a full support of writable characters. Would it be fair to repudiate words such as *canapé, façade*, *dürüm* or even the old-fashioned *rememberèd*?

Go natively uses Unicode. All the source files need to be encoded in UTF and it even has a specific primitive type called rune that serves to encode a Unicode codepoint.

If we take for example the default line that appears on the Go playground and look at the length of the string (including the comma and the whitespace):

```
fmt.Println(len("Hello, 世界"))
```

This prints out 13. Indeed, UTF-8 requires 3 bytes to encode each of these non-latin characters. On the other hand,

```
fmt.Println(len([]rune("Hello, 世界")))
```

This outputs 9. We are measuring the number of runes and not the number of bytes necessary to encode them. Keep that in mind whenever iterating over a string's elements: you can either access its byte representation with `[]byte(str)`, or access its rune representation with `[]rune(str)`, which is the default behaviour.

Listing 5.7 Print the string and rune lengths

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
    fmt.Println(len("Hello, 世界"))
    fmt.Println(len([]rune("Hello, 世界")))
}
```

```
Console Output:

Hello, 世界
13
9
```

The ask method

We have a variable that allows us to read from the standard input once the game is set. Let's politely ask Claudio for his next word. Since the feature of retrieving an attempt provided by the player through the reader is something we can summarise in a sentence without having to explain how it works, it's a great candidate for a function! We'll call it `ask`, for clarity and simplicity. This method will accept a `Game` receiver, since it needs to read from its reader, and will return a slice of runes - the word proposed by the player. It will guarantee we have a valid suggestion.

Listing 5.8 game.go: `ask` method signature

```
// ask reads input until a valid suggestion is made (and returned
func (g *Game) ask() []rune {
// ...
}
```

The experienced reader will have noticed that we use a pointer receiver here. There are two reasons for this. The first is simple: we'll be modifying the state of our `Game` structure via many of its methods, so they will all require a pointer receiver. It is good Go practice to avoid having both pointer and non-pointer receiver methods on a type, for consistency. The second is a bit more complex, and is motivated by the fact that the `Game` structure has a field that is a pointer: the `output` field. Appendix E covers the issues that can happen when using copy-receivers with pointer fields.

Inside this `ask` method, we read the line using the reader. Should an error occur, we'll print it using `Fprintf`, but we decide to continue anyways and wait for a new attempt. That is: a jammed line won't cause the game to crash, but merely to ask for another word. `Fprintf` will allow us to write to the standard error.

We'll see, when completing the `Play` method, how to better deal with errors. The hard truth is that they shouldn't be ignored, most of the time. However, deciding that an error is not blocking is a good moment to leave a note for future-self explaining this decision, in the form of a comment.

We can add an easy check on the length of the word. For the moment, we play with the same parameters as the original Wordle, with 5-character long words. We can define a constant at package level and use it everywhere we need it.

A constant serves two purposes. The first one is to address a developer's laziness: by re-using a constant, we make sure that we don't have to update several lines of code should the value change. The second is to make the code clearer by giving a name and a purpose to a value (just like we do with variables): `connectionTimeout` is more explicit than `5*time.Minute`. So please don't call your constant `time5minutes`.

Making it a constant is an unambiguous way of telling the reader of the code that this value isn't expected to change.

Listing 5.9 game.go: `ask` method with the reader

```
const solutionLength = 5

// ask reads input until a valid suggestion is made (and returned
func (g *Game) ask() []rune {
    fmt.Printf("Enter a %d-character guess:\n", solutionLength)

    for {
        playerInput, _, err := g.reader.ReadLine() #A
        if err != nil {
            _, _ = fmt.Fprintf(os.Stderr, "Gordle failed to read
            continue #B
        }

        guess := []rune(string(playerInput))

        // TODO Verify the suggestion has a valid length.
    }
}
```

The `ReadLine` method will give us the user's input as a slice of bytes. We will then need to convert this byte slice into a rune slice. Converting each byte into the rune representing that byte would be a very bad mistake. Everything not ASCII would break. To properly convert a slice of bytes that we know represents a string to a slice of runes, we need to first convert the byte slice into a string and then into a rune slice.

The built-in method `len()` returns the length of a slice (or array). We can use it to compare the length of Claudio's word against the `solutionLength` constant. We should be polite and return a message if it fails. However, in order to make it clear that we aren't returning "happy path" information, we'll use the standard error output - available via `os.Stderr`.

Listing 5.10 game.go: `ask` method with the reader

```
guess := []rune(string(playerInput)) #A

if len(guess) != solutionLength { #B
    _, _ = fmt.Fprintf(os.Stderr, "Your attempt is invalid with G
} else {
    return guess
}
```

Let’s test the ask method

Did you think we’d forget?

The `ask` method uses its receiver's reader and returns a slice of runes. Let’s declare these in the test case definition.

As we are using the standard library’s `bufio.Reader`, we can use a reader to any stub mimicking the player’s input. A stub is a very simple way of implementing a dependency over a third party (in our case, Claudio).

Think of a few original test cases that use your favourite alphabet, abjad, syllabary, or even emoji from the Unicode list of supported characters.

Listing 5.11 game_internal_test.go

```
package gordle

import (
    "errors"
    "strings"
    "testing"

    "golang.org/x/exp/slices"
)

func TestGameAsk(t *testing.T) {
    tt := map[string]struct {
        input string
        want  []rune
    }{
        "5 characters in english": {
            input: "HELLO",
            want:  []rune("HELLO"),
        },
        "5 characters in arabic": {
            input: "مرحبا",
want: []rune("مرحبا"),
        },
        "5 characters in japanese": {
            input: "こんにちは",
            want:  []rune("こんにちは"),
        },
        "3 characters in japanese": {
            input: "こんに\nこんにちは",
            want:  []rune("こんにちは"),
        },
    }

    for name, tc := range tt {
        t.Run(name, func(t *testing.T) {
            g := New(strings.NewReader(tc.input))

            got := g.ask()
            if !slices.Equal(got, tc.want) {
                t.Errorf("got = %v, want %v", string(got), string
            }
        })
    }
}
```

You might have noticed that the first line of our test function is somewhat different from those in the previous chapters. Indeed, we used to declare a `testCase` structure, that would encapsulate all the fields we needed - and, now, it’s gone! Or, rather, it’s been replaced with what is called an anonymous structure. This implementation is very common in Go tests, and we’ll be using it from now on. However, if you prefer the previous way of declaring the `testCase` structure, that’s also perfectly valid. For comparison, here are both:

```
type testCase struct {            testCases := map[string]struct
input string                input string
want  []rune                want  []rune
}                    }
testCases := map[string]testCase
```

Note that with the current implementation of `ask`, if we have an input of only 3 runes, the `ReadLine` method waits forever, after ignoring the invalid 3-character-long line and waiting for more. What’s happening here is that

`ReadLine`, when hitting the end of the input, will return a specific error to let the caller know that there is nothing to be read.

Why are we not using == to compare slices? We can for arrays, after all! Remember that slices hold a pointer to their underlying array. Array values are comparable if values of the array element type are comparable. Two array values are equal if their corresponding elements are equal. But when it comes to slices, structs and maps, == will simply not work. It’s not that it will produce random results - no! Instead, Go will simply not let you compare two slices. Not even a slice with itself. The only entity that we can compare with a slice is the nil keyword.

This might sound a bit harsh, but it should be considered a safeguard rather than a restriction. It is possible, in tests, to use the method `reflect.DeepEqual`, but it was not designed for performance; you should avoid it in production code. Instead, write the simple loop.

Now, that does not compile. We added a dependency on an external library. The developers of the Go language will typically write their new libraries in `golang.org/x/` in order to let the community test them out. Once they are stable, they can be moved to the standard library.

Remember your module? To add the dependency, let the go tool look for the latest version with the following command:

```
go get golang.org/x/exp/slices
```

You can see that a `go.sum` file has appeared. This is typically something that should not be committed to your version control, but generated locally. You can also notice that your `go.mod` has changed, and it now refers to the new dependency.

Is your code compiling and your test passing now? We can move on.

Play

We are now able to read Claudio’s guess. Let’s make great use of this ability, and plug it in the `Play()` method which looks like this:

Listing 5.12 game.go: `Play` method

```
// Play runs the game.
func (g *Game) Play() {
    fmt.Println("Welcome to Gordle!")

    // ask for a valid word
    guess := g.ask()

    fmt.Printf("Your guess is: %s\n", string(guess))
}
```

There is one missing update in the main, can you spot it? Since New() method takes a reader as parameter, we need to pass os.Stdin to wait for the player’s input.

Listing 5.13 main.go: main function updated with os.Stdin

```
package main

import (
    "os"

    "learngo-pockets/gordle/gordle"
)

func main() {
    g := gordle.New(os.Stdin)
    g.Play()
}
```

You can test it manually in your console. Here is an example of a game:

```
$ go run main.go
Welcome to Gordle!
Enter a 5-character guess:
four #A
Your attempt is invalid with Gordle's solution! Expected 5 charac
apple #C
Your guess: apple #D
```

You may have noticed the `ask()` method is now responsible for both reading the input, standardising it and validating the guess. It is best to separate the concerns, let’s refactor!

5.1.3 Isolate the check

There is no specific rule concerning the responsibilities of a method, but when you start having multiple operations of different natures, it might be best to have one function for one action. Small functions are also easier to test and maintain, while making sure our code is robust!

Let’s move the word length validation to another method, adequately named `validateGuess`. Notice that we did say method, and not function. This `validateGuess` will have a receiver over the `Game` type. The reason for this won’t be visible here, but in the next pages, we’ll want to get rid of that `solutionLength` constant, in favour of a test against the real secret word’s length, which will be part of the `Game` structure. This `validateGuess` method is in charge of the validation, it takes the guess as a parameter and returns whether the word is valid.

There are two common ways of informing of the success of a check - either we can return a boolean value, or an error which is in Go a value, representing the issue we found (or `nil`, if everything was fine). Returning a boolean is simple, but it doesn’t allow for fine behaviour. What if we need to specify that we faced an unrecoverable (at least for this `validateGuess`’s concern) error? While there is no granularity with booleans, errors offer a lot more variations that will allow the caller - in our case, the `ask` method - to decide the behaviour if there is any error. We will also see how it makes the code easier to test.

Error propagation

Unfortunately, most programs will face errors. A file could be missing. A connection could be closed. A value could be an unexpected zero. Go’s take on error handling is to use functions that return one or more values, the last one (if any) being an error. The first line that follows the call to such a function is the most common line of any Go source code: `if err != nil {`. It’s so common we have a macro to paste it in our IDEs.

When we retrieve an error, the best thing we can do is to handle it as much as we can, and, if there’s nothing this layer can do about it, then propagate it to the upper layer, nicely wrapped. Wrapping it will provide context to the layer that can finally decide to handle the error. The simplest way of wrapping an error is to call `fmt.Errorf("... %w …", …, err, …)`, where the `%w` stands for “wrap the error”.

`fmt.Errorf` returns the wrapped error.

The following snippet of code holds the new method and its attached error. It is declared outside of the `validateGuess` method to enlarge its scope and use it in unit tests later, validating we retrieve the proper error.

Listing 5.14 game.go: `validateGuess` method

```
// errInvalidWordLength is returned when the guess has the wrong
var errInvalidWordLength = fmt.Errorf("invalid guess, word doesn'

// validateGuess ensures the guess is valid enough.
func (g *Game) validateGuess(guess []rune) error {
    if len(guess) != solutionLength {
        return fmt.Errorf("expected %d, got %d, %w", solutionLeng
    }

    return nil
}
```

We decided to keep the validating function simple. Every implementation of Wordle will feature its own validator - some will ensure that the attempt has the right length, others that the attempt is a word that exists in a dictionary, or in a list provided by the developers… There are as many implementations of this as one’s mind can think of.

Make sure to replace the validation with the call to `validateGuess` in the `ask` method like below. :

Listing 5.15 game.go: Call to `validateGuess` in `ask` method

```
err = g.validateGuess(guess)
if err != nil {
   _, _ = fmt.Fprintf(os.Stderr, "Your attempt is invalid with Go
} else {
    return guess
}
```

Testing validateGuess()

Extracting the validation into a dedicated method is one way to test unitary behaviour. As we did previously, we will use Table-Driven Tests to cover several cases without too much repetition. Let’s test this new function!

First, we need in our test a new `Game` object to be able to call the `validateGuess` method. Then, we build the structure holding all the parameters needed for our execution and validation phase, in this case, the attempted word and the expected error. Then it’ll be time to add test scenarios to our table. Finally, in the execution phase, we call `validateGuess` with the test case word and verify the error is as expected.

The `errors` package provides an important function, `errors.Is(err, target error) bool`, which reports whether any error in `err`'s chain matches the `target` error. `errors.Is` is very handy when dealing with wrapped errors, as it will unwrap all the errors and the chains to verify the presence of a specific error. Wrapped errors are similar to matryoshkas, and `errors.Is` lets you know if a layer is wearing a blue dress.

Now you are familiar with Test-Driven Tables, you should be able to write the first test scenario without checking the solution, which we provide anyway:

Listing 5.16 game_internal_test.go: Test `validateGuess` method

```
package gordle

import (
   "errors"
   "testing"
)

func TestGameValidateGuess(t *testing.T) {
   tt := map[string]struct { #A
      word     []rune
      expected error
   }{
      "nominal": { #B
         word:     []rune("GUESS"),
         expected: nil,
      },
      "too long": {
         word:     []rune("POCKET"),
         expected: errInvalidWordLength,
      }
   }

   for name, tc := range tt {
      t.Run(name, func(t *testing.T) {
         g := New(nil) #C

         err := g.validateGuess(tc.word) #D
         if !errors.Is(err, tc.expected) { #E
            t.Errorf("%c, expected %q, got %q", tc.word, tc.expec
         }
      })
   }
}
```

Exercise 5.1: Here we cover the happy path case with a five-character word and one unhappy path when the word is too long. Add new test cases to cover more invalid paths. What happens if the attempt has fewer characters, is empty, or is `nil`?

Input normalisation

There is a line left in this ask method that could be reused later.

```
guess := []rune(string(suggestion))
```

We accept all kinds of upper and lowercase mixes and it will later be simple to take care of the not-yet-supported writing systems if we put this into a small function.

Listing 5.17 game.go: Split characters

```
// splitToUppercaseCharacters is a naive implementation to turn a
func splitToUppercaseCharacters(input string) []rune {
    return []rune(strings.ToUpper(input))
}
```

Replace the long line above with a call to your new function. You can also check that the input was correctly normalised by writing a test over the function.

5.1.4 Check for victory

We have built the foundations of your game. The next step is to verify if Claudio's attempt is the solution. If it isn't, he gets to try again. We will also limit the number of attempts to make the game more challenging. Indeed, there are two ways to end a game of Gordle: either the word was found, or the maximum number of attempts was reached.

In order to give Claudio more attempts at finding the solution, we need to enrich the `Game` structure, as it holds all the information required to play a game. We keep the solution in the same type as the attempt, a slice of runes, in order to make the comparison and the manipulation easy. We can either have the `gordle` package select that solution, or, for now, have it as a parameter of the `New` function. To make sure the player can still try new words, we need to store the maximum number of attempts in a variable somewhere. We could have it a constant of the package, but having it embedded within the `Game` structure will avoid creating unnecessary constants, and will eventually keep the door open if we want to allow for more attempts.

Listing 5.18 game.go: Add solution and max attempts

```
// Game holds all the information we need to play a game of Gordl
type Game struct {
    reader      *bufio.Reader
    solution    []rune #A
    maxAttempts int #B
}
```

Let's update the `New` function by passing both the solution and the maximum number of attempts as parameters, for the time being.

```
// New returns a Game variable, which can be used to Play!
func New(playerInput io.Reader, solution string, maxAttempts int)
    g := &Game{
        reader:      bufio.NewReader(playerInput),
        solution:    splitToUppercaseCharacters(solution),
        maxAttempts: maxAttempts,
    }

    return g
}
```

We take the solution as a string, which is easier to use, and reuse the function we just wrote before. We are also normalising the solution given to our package by setting all letters to uppercase, something which again only makes sense in a limited number of alphabets.

In the Play method, we can add a loop to let Claudio suggest a second word, a third word, and so on. The criterion to end the loop will be that Gordle has received a number of attempts equal to the maximum allowed. That loop starts by asking for a word, ensures its validity, and then checks if the attempt is equal to the solution.

Here is the new version of the `Play()` method:

Listing 5.19 game.go: Add check on victory

```
// Play runs the game.
func (g *Game) Play() {
    fmt.Println("Welcome to Gordle!")

    for currentAttempt := 1; currentAttempt <= g.maxAttempts; cur
        guess := g.ask() #B

        if slices.Equal(guess, g.solution) {
            fmt.Printf("🎉 You won! You found it in %d guess(es)!
            return
        }
    }

    fmt.Printf("😞 You've lost! The solution was: %s. \n", string
}
```

Using emojis

To insert emojis, use Ctrl-Cmd-Space on Mac, Win-period on Windows. Typing Ctrl-Shift-U on Linux will let you enter unicode typing mode; write the hexadecimal code value and press enter to see it appear. `1F984` is the code of a unicorn. You can find the list of all available codes here: https://unicode.org/emoji/charts/full-emoji-list.html.

With the solution now embedded in a `Game` object, we can remove the constant `solutionLength` everywhere and replace it with the length of the solution - `len(g.solution)`.

Listing 5.20 game.go: Example of replacement

```
// Before the replacement
fmt.Printf("Enter a %d-character guess:\n", solutionLength)
// After the replacement
fmt.Printf("Enter a %d-character guess:\n", len(g.solution))
```

Are our tests still passing?

It’s been a long time… are our tests still passing?

Well, for now, they don’t even compile, because we changed the signature of `New`. As you can see, it forces the user to provide the mandatory fields:

```
g := New(strings.NewReader(tc.input), string(tc.want), 0)
```

The `ask` method does not use the max number of attempts, so we can give the zero value as parameter and tell the next maintainer that it is useless in this context. It makes our call a bit wacky, but this weird zero will be fixed in part 4 when we make it optional.

This should do it! We can continue to update the rest of the code, starting with the main. Indeed, as we mentioned earlier, we need a solution word to play. For now, we’ll hardcode this in the `main` like the following snippet of code

Listing 5.21 main.go: Main with hardcoded soluton and updated New()

```
package main

import (
    "os"
    "learngo-pockets/gordle/gordle"
)

const maxAttempts = 6

func main() {
    solution := "hello"

    g := gordle.New(os.Stdin, solution, maxAttempts)

    g.Play()
}
```

Let’s have a round of Gordle!

Here is an example of the game when the player finds the solution. This illustrates the game when Lucio plays his game - an easy win on the first attempt! Remembering what he wrote and hardcoded in `main` a few minutes earlier did help here…

```
$ go run main.go
Welcome to Gordle!
Enter a 5-character guess:
hello
🎉 You won! You found it in 1 attempt(s)! The word was: HELLO.
```

However, if Lucio lets his friend play the game, it’s a lot more difficult to win! With no hints to guide Claudio towards the solution, this game is almost impossible to win (unless one plays it twice, but changing the solution every time the game is played is work for later).

```
$ go run main.go
Welcome to Gordle!
Enter a 5-character guess:
sauna
Enter a 5-character guess:
pocket
Your attempt is invalid with Gordle's solution: expected 5, got 6
[...]
Enter a 5-character guess:
phone
😞 You've lost! The solution was: HELLO.
```

The game would be quite a bore if it didn't give the player some information about how close they are to the solution, in the form of hints as to which characters are properly located, and which are misplaced. It's time to give Claudio some feedback!

5.2 Providing feedback

Claudio just submitted a word. And our task is now to let him know which characters of that word are in the correct position, which are in the wrong position, and which simply don't appear in the solution. This will help him find the secret word that Gordle initially chose.

A good feedback should return a clear hint for every character of the input word, explicit about the correctness of the character in this or that position. The initial Wordle uses background colour, behind each character of the player's input. While this was great for most of us, an application that provides feedback to the user should take into account user accessibility. A common impairment is colour vision deficiency, where making a difference between green and orange isn't as obvious as it would seem. An option was added to Wordle that would allow players to use colours with high contrast instead of the default ones. Let's see what we can do here!

5.2.1 Define character status

We've determined that a feedback will be a list of indications that can have three values - correct, misplaced, and absent. In order to easily manipulate the feedback for a character, we create the type `hint` to represent these hints, of the type `byte` - the smallest type Go offers, regarding memory usage. The `iota` keyword allows us to automatically number them from 0 to 2. Using underlying numbers will make it easier for us when it comes to finding the best hint we can provide the player. Define this `hint` type in a new file, `hint.go`, in the package `gordle`.

Listing 5.22 hint.go: Hint character type and enum

```
// hint describes the validity of a character in a word.
type hint byte
```

```
const (
   absentCharacter hint = iota    #A
   wrongPosition                 #B
   correctPosition        #C
)
```

Ordering values in an enum

In our example, we have 3 values that we want to list in an enum. There are 3! ("factorial 3", equal to 3 * 2 * 1) overall possible permutations of 3 elements, which is 6 ways of ordering them. In Go, the best practice is always to make best use of the zero-value, and to sort the elements of the enum in a logical way - in our case, from worst to best. We could have had an `unknownStatus` as the zero-value of our enum, but as we'll see later, using the zero-value for `absentCharacter` will come in handy.

These hints will be printed on the screen to help Claudio make the best guess he can on his next attempt. We need to find a representation of these hints that is both simple and explicit. Since this is the 21st century, what better than emojis to convey a message that we can all understand and agree upon? We want to attach one emoji to each hint, and the Go way of implementing this is through a `switch` statement.

Let's now think about how this method that will provide a `string` representation of a `hint` is to be called. Both literally and practically: how do we want to name it, and how do we want to make calls to it.

The Stringer interface

One of the important interfaces to keep in mind while writing Go code is the `Stringer` interface defined in the `fmt` package. Its definition is simple: `String() string`. This means any type that exposes a parameterless method named `String` that returns a `string` implements this interface. So far, so good - but there is a key aspect that still has to be mentioned here. If we have a look at the `fmt.Printf` functions, we can read that "*Types that implement Stringer are printed the same as strings*". This means, in order to print a variable of a type that implements Stringer, we only need to use `%s`, `%q`, or `%v` in a `Printf` call, and this will, itself, call the `String()` method.

Implementing the `Stringer` interface will save a lot of time - reusing a well-known convention is better than trying to be smart, and it won’t require an extra layer of knowledge from future developers who will later work on this code.

Listing 5.23 hint.go: `String()` method

```
// String implements the Stringer interface.
func (h hint) String() string {
   switch h {
   case absentCharacter:
      return "⬜" // grey square
   case wrongPosition:
      return "🟡" // yellow circle
   case correctPosition:
      return "💚" // green heart
   default:
      // This should never happen.
      return "💔" // red broken heart
   }
}
```

Note that if your terminal does not display emojis properly, you can replace them with numbers or regular characters such as "`.`" for absent, "`x`" for misplaced, and "`O`" for correctly placed characters. It is less fun, but, at least, more readable than squares.

Providing a hint for a single character is good, but we’ll need to do so for every character of the word. We’ll represent the feedback of a word as a structure. It will hold the hint for each attempted character compared to its position in the solution. We name this new type `feedback`. We could place the definition of a `feedback` in a `feedback.go` file, but since it’ll be very tightly linked to a `hint`, and that these two types won’t have more than one method over them, we can place them in the same file.

Listing 5.24 hint.go: `feedback` type

```
// feedback is a list of hints, one per character of the word.
type feedback []hint
```

One can wonder what the benefit of having an alias for a slice of status really is. This is an interesting question, and its answer is simple: we can define methods over that alias. In particular, here, we'll want to print the feedback so Claudio can next make an informed guess. And, as we've seen a few lines earlier, the best way to provide a nice string from a structure is to have its type implement the `Stringer` interface. All we have to do is write a small function that will print the feedback of each character.

Benefits of a strings builder

Our first and naive implementation of the `String` method on the `feedback` type is to create a `string`, and append the status representation as we go through the feedback's statuses.

Listing 5.25 Naive implementation of building a string

```
// StringConcat is a naive implementation to build feedback as a
// It is used only to benchmark it against the strings.Builder ve
func (fb feedback) StringConcat() string {
   var output string
   for _, h := range fb {
      output += h.String()
   }
   return output
}
```

However, there is an important lesson here: one should never concatenate Go strings in a loop. We've written this function here only for teaching purposes.

In Go, strings are immutable. Constant. We cannot alter them. We can't even replace a character in a string without casting something to a slice, and something back to a string. This makes string manipulation quite painful, especially for what would seem the simplest task - sticking two strings together. When we use the + operator on two strings, Go will allocate memory for a new string of the correct size and copy the bytes of each operand into that new string.

While this is simple and clear when concatenating two strings together, it becomes slower as soon as we have several strings to merge. Keep in mind that, when the number of strings to connect exceeds two, there are two quite common alternatives that are worth checking:

- strings.Join(elems []string, sep string) string : returns a string of the elements separated by the separator (usually a whitespace or a comma). Works only if you already have a slice of strings, which is not our case here.
- strings.Builder: Slightly more complex, but also a lot more versatile. Under the hood, a strings.Builder stores the characters in a slice of runes, which is a lot easier to grow than a rock-solid string. This is the option we use in our example.

The `strings` package provides the type `Builder` that lets you build a `string` by appending pieces of the final string, while minimising the number of memory allocations and reallocations every time we add some characters.

In order to use the `Builder`, we declare a new variable and to fill the string. This type exposes several methods that can be used to append characters to the string being built: `WriteString`, `WriteRune`, `WriteByte` and the basic `Write`, which takes a slice of bytes. In our case the `WriteString` method is the most appropriate, since we know how to make a `string` from a `status`. Once we're done feeding data to the builder, calling `String()` on it will return the final string.

Listing 5.26 hint.go: `String()` on `feedback` type

```
// String implements the Stringer interface for a slice of hints.
func (fb feedback) String() string {
   sb := strings.Builder{}
   for _, h := range fb {
      sb.WriteString(h.String())
   }
   return sb.String()
}
```

Want to check the difference? See Appendix D.1 for how to benchmark your code. Once we've selected which implementation we'd rather use, let's not forget to test this method. Testing `feedback.String()` will cover `hint.String()`, which will be enough - therefore, no need to also test the `hint.String()` method.

We are now ready to send feedback to Claudio - but we are missing a small piece of information here. We don't know yet which characters are correctly - or incorrectly - located! This will be our next task before the game can be enjoyed.

5.2.2 Checking a guess against the solution

This section is about approaching a new problem. Whatever the language you use, there will be times when you need to roll away from the screen, take a piece of paper and pen, and think about the best way to solve your problem. In our case, we want to make sure the hints we give are accurate. A letter in the correct position should always be marked as in the correct position. A letter in the wrong position should only be marked as such if it appears unmatched elsewhere in the word. We need to make sure we cover double letters properly - for instance, what should be the feedback to the word "SMALL" if the solution is "HELLO" ?

As this book is not about algorithms, we'll start with the pseudo-code of the check function that implements our solution. Feel free to think about it yourself before jumping to our solution.

Pseudo-code is an intermediate representation of the code's logic with sentences and words rather than instructions. Pseudo-code doesn't have an official grammar - sometimes, the loops end with `END FOR`, sometimes with a curly brace. It's up to you to decide how you want to write it. Reading your own pseudo-code should be at least as clear as reading code, and you'll have access to operators that might not exist for a given programming language. Pseudo-code magically offers any function that you can dream of (although you might have to implement them later on). Here, we want to highlight the use of a "fake" operator such as `mark`.

Our pseudo-code's syntax will be close to that of Go, with curly braces, because we think it makes more sense in this book. We had previous drafts of pseudo-code that used boxes, arrows and loops.

Listing 5.27 Pseudo-code of `computeFeedback()`

```
func computeFeedback(guess, solution) feedback {

   for all characters of the guess { #A
      mark character absent
   }

   for each character of the attempt {                  #B
      if the character in solution and guess the same {
         mark character as seen in the solution
         mark character with correct position status
      }
   }

   for each character of the guess {                     #C
      if current character already has a hint {
         skip to the next character
      }

      if character is in the solution and not yet seen {
         mark character as seen in the solution
         mark character with correct position status
      }
   }

   return the hints
}
```

Once we’ve written the pseudo-code, we can shoot some examples at it and see how it behaves. By first iterating over correctly placed characters, and then over those that are misplaced, we get the expected output for “SMALL” vs “HELLO”:

```
S M A L L
H E L L O
□□◍♡□
```

We need to think about how to implement the different parts that are still “pseudo-code magic”. How do we `mark` a character with a hint? How do we `mark` a character as seen in the solution? There are lots of ways of implementing this, we’ll go with a simple approach here: we’ll use a slice of hints to mark characters of the guess with their appropriate hint, and we’ll use a slice of boolean to mark characters of the solution as either seen or not yet seen.

We recommend you give it a try before checking the solution.

Listing 5.28 game.go: `computeFeedback()` method

```
// computeFeedback verifies every character of the guess against
func computeFeedback(guess, solution []rune) feedback {
   // initialise holders for marks
   result := make(feedback, len(guess)) #A
   used := make([]bool, len(solution))  #B

   if len(guess) != len(solution) {
      _, _ = fmt.Fprintf(os.Stderr, "Internal error! Guess and so
      return result #C
   }

   // check for correct letters
   for posInGuess, character := range guess {
      if character == solution[posInGuess] {
         result[posInGuess] = correctPosition
         used[posInGuess] = true
      }
   }

   // look for letters in the wrong position
   for posInGuess, character := range guess {
      if result[posInGuess] != absentCharacter {
         // The character has already been marked, ignore it.
         continue
      }

      for posInSolution, target := range solution {
         if used[posInSolution] {
            // The letter of the solution is already assigned to
            // Skip to the next letter of the solution.
            continue #D
         }
         if character == target {
            result[posInGuess] = wrongPosition
            used[posInSolutionj] = true
            // Skip to the next letter of the guess.
            break #E
         }
      }
   }

   return result
}
```

A tricky part here is handling the case if the guess and the solution have different lengths. Since this is our code, we know this can’t happen - because it’s been checked earlier. But if somebody changes the code later (including future Lucio who forgot everything he wrote and why), it will end in a segfault, during runtime; we can’t even warn him with a unit test. For this reason, we decide to re-check the length of the guess against the length of the solution here.

Another option would have been to return a feedback and an error in the `computeFeedback` function. These assumptions are tolerable in internal functions, but they would absolutely not be accepted in exposed functions, because we don’t control the range of values that can be passed to functions available to the rest of the world.

Congratulations, you implemented the most difficult part! Now, let’s add some tests.

Testing computeFeedback()

To properly test the method `computeFeedback`, we need to provide a `guess`, a `solution` and the expected `feedback`. Once we have these, we can call `computeFeedback` and verify that the received `feedback` is the expected one.

To easily compare two `feedbacks`, we write a helper method next to the `feedback` definition. The package `github.com/google/go-cmp/cmp"` from Google provides some insights as to how we should name this method: “Types with an `Equal` method may use that method to determine equality”.

Listing 5.29 hint.go: `Equal()` helper

```
// Equal determines equality of two feedbacks.
func (fb feedback) Equal(other feedback) bool {
   if len(fb) != len(other) {
      return false
   }

   for index, value := range fb {
      if value != other[index] {
         return false
      }
   }

   return true
}
```

Earlier, to compare two slices, we used the `golang.com/x/exp/slices` package. As always, when writing code, there is more than one way of doing things. Here, we offer a different take, which is as valid as the previous one. You will find arguments for both over the internet. Our recommendation is to use whichever is clearer to you. If you’re curious, checking the implementation of `slices.Equal` is worth the time, but it requires some understanding of generics, which is a topic for later.

You now know how to write a test in a table-driven way. First, we define our structure holding the required elements for our test case, inputs and expected outputs. Then, we write our use cases, and finally we call the method and check the solution. Here, for the sake of clarity and to avoid unnecessary clumsiness, we’ve decided to use strings instead of slices of runes in the structure of our test case for the guess and the solution. The conversion from a string to a slice of runes is simple and safe enough to be performed in execution of the test. On the other side, we want to explicitly check the contents of the returned feedback, and, for this reason, we have a `feedback` field in the test case.

Listing 5.30 game_internal_test.go: `computeFeedback()` tests

```
package gordle

import "testing"

func TestComputeFeedback(t *testing.T) {
   tt := map[string]struct {
      guess            string
      solution         string
      expectedFeedback feedback
   }{
      "nominal": {...},
      "double character": {...},
      "double character with wrong answer": {...},
        "two identical, but not in the right position (from left to
            guess:            "hlleo",
            solution:         "hello",
            expectedFeedback: feedback{correctPosition, wrongPositio
        },
    }

    for name, tc := range tt {
        t.Run(name, func(t *testing.T) {
            fb := computeFeedback([]rune(tc.guess), []rune(tc.soluti
            if !tc.expectedFeedback.Equal(fb) {
                t.Errorf("guess: %q, got the wrong feedback, expected
            }
        })
    }
}
```

Exercise: Part of the fun of a project is to come up with some edge case scenarios. Try and find some that would push the logic to its limits.

Finally, we need to integrate the `computeFeedback` call in the `Play` function. This isn't too difficult, especially as we only want to print the feedback to Claudio.

Listing 5.31 game.go: Update the `Play()` function to display the feedback

```
[...]
    for currentAttempt := 1; currentAttempt <= g.maxAttempts; curr
        guess := g.ask()

        fb := computeFeedback(guess, g.solution) #A

        fmt.Println(fb.String()) #B

        if slices.Equal(guess, g.solution) {
            fmt.Printf("🎉 You won! You found it in %d guess(es)! Th
            return
        }
    }
[...]
```

And finally this is what playing the game looks like!

```
$ go run main.go
```

```
Welcome to Gordle!
Enter a 5-character guess:
hairy
💚⬜⬜⬜⬜ #A
Enter a 5-character guess:
holly
💚🟡💚💚⬜ #B
Enter a 5-character guess:
hello
💚💚💚💚💚 #C
🎉 You won! You found it in 3 attempt(s)! The word was: hello.
```

We now have a solution checker and we are able to give Claudio some well-deserved feedback! Feedback makes it a lot easier for the player to find the solution. However, there is a small final detail we still need to address that will provide even more fun: how about Claudio getting a different word every time he plays Gordle? We proved that our implementation works with a hardcoded solution, it is time to add a corpus and add randomisation to our game.

5.3 Corpus

In linguistics, a corpus is a collection of sentences or words assumed to be representative of and used for lexical, grammatical, or other linguistic analysis. Our corpus will be a list of words with the same number of characters.

Until now, we have been using a hardcoded solution and ensured our algorithm was working as expected. In this section, we will focus on adding randomisation to our game by picking a word from a given list. Let’s first retrieve a list of words and then pick a random word in it as the solution of the game.

5.3.1 Create a list of words

First, we create a `corpus` directory with a file named `english.txt`. This file contains a list of uppercase English words, one per line. Our corpus was built while playing other versions of the game. Feel free to use the adequate list for your own game. Adding a new corpus for a different language, or for a different list of words (6-character long, for instance) is now simple: all we have to do is add a file here and have the program load it.

5.3.2 Read the corpus

Parsing a file is a very common task that most programs face. It could be a configuration file with default values to load, an input file as we have here, a database query, an image, a video file, or anything that comes to your mind. If it exists on a disk, a program is going to read it. In our case, we want to read the corpus file as a list of words that we will store in a slice of strings.

Start by creating a new file, `corpus.go`, where all methods related to the corpus will live.

How can we read the corpus? As it happens, the `os` package provides a `ReadFile` method which takes the path to a file on disk as a parameter, reads it and returns its contents as a slice of bytes. It reads the whole file or returns an error if something bad happened.

The signature of the method `os.ReadFile`:

```
func ReadFile(name string) ([]byte, error)
```

It’s good to keep in mind that files, when written on disk, are nothing but a chunk of bytes. Nice characters, spaces, tabulation, tables, etc. are rendered by file editors. This book was saved as some 0’s and 1’s. This is why we don’t immediately have a list of lines out of the `ReadFile` function. That logic has to be implemented by us, at reading time. We know that some of these bytes are the new line character - but let’s not rush to an easy solution that would be to split this slice of bytes on \n. Indeed, how if the byte representation for \n (`0x0a`) was in fact a byte part of a representation of a non-ASCII longer character? Or, what if the file was encoded differently, with a new line character not only represented by \n, but rather a \r\n ?

Manipulating an array of bytes in our case is not very practical, so we will convert it to a string in order to split on any whitespace, including the new line characters. The `strings` package exposes `Split` and its siblings `SplitAfterN`, `SplitN`, `Cut`, and Fields. These functions come in handy when the need to split strings arises. In our case, the basic `Fields` is enough, as it will split the string into a slice of its substrings delimited by all default whitespaces, which relieves us from the trouble of knowing them. This slice of substrings is our list of words eligible to become a solution.

The signature of the method `strings.Fields` is:

```
func Fields(s string) []string
```

The full code of the `ReadCorpus` method looks like this:

Listing 5.32 corpus.go: `ReadCorpus` method

```
const ErrCorpusIsEmpty = corpusError("corpus is empty")

// ReadCorpus reads the file located at the given path
// and returns a list of words.
func ReadCorpus(path string) ([]string, error) {
   data, err := os.ReadFile(path)
   if err != nil {
      return nil,  fmt.Errorf("unable to open %q for reading: %w"
   }

   if len(data) == 0 {
      return nil, ErrCorpusIsEmpty
   }

   // we expect the corpus to be a line- or space-separated list
   words := strings.Fields(string(data))

   return words, nil
}
```

Sentinel errors

Error management is at the heart of software development, whatever your chosen language and whatever application you are making. Say you try to read a file line by line: it could be that the file doesn’t exist, or you might be missing the adequate rights to read it, or it could be empty or incomplete, or, finally, accessible, and you could read it to the end. In all these cases, you get an error back; your program’s reaction will be different depending on which error case you fall into.

You would like to check the error that was returned, with a line of code such as `err == ErrNoSuchFile` or `err == EOF`.

Sentinel errors are a type of recognisable errors. In Go, “errors are values”, meaning that they carry a meaning. Sentinel errors must behave like constants, but Go will only accept primitive types as constants, and not method calls. Unfortunately for us, the two default ways to build an error are by calling `fmt.Errorf` or `errors.New`. And these don’t produce constant values - they produce the output of a function, which isn’t known at compile time, only at execution time. This implies that errors generated by `fmt.Errorf` or `errors.New` will always be variable. So, how do we get the constant errors we’d like? We declare our own type and have it implement the `error` interface:

Listing 5.33 errors.go: Sentinel error `corpusError`

```
package gordle

// corpusError defines a sentinel error.
type corpusError string

// Error is the implementation of the error interface by corpusEr
func (e corpusError) Error() string {
   return string(e)
}
```

Here, we can declare a `corpusError` that is a constant (it is as primitive as a string) and still implements the `error` interface. Yes, we wish this type were in the standard library. Maybe in a future version of Go.

Small note: if you look at `io.EOF` in the code, you’ll realise it is a global and exposed variable - it was generated at execution time by a call to `errors.New`. Don’t do that at home. Imagine a pesky colleague were to do this:

```
io.EOF = nil
...
if err == io.EOF { // oops
```

Test the reading

Now, we can test if we can actually read a file full of words into a slice of string. Let’s add a nominal case reading from the corpus we created for the English list of words, verifying the length and the associated error, if any.

Listing 5.34 corpus_test.go: Test `readCorpus()`

```
package gordle_test

func TestReadCorpus(t *testing.T) {
   tt := map[string]struct { #A
      file   string
      length int
      err    error
   }{
      "English corpus": { #B
         file:   "../corpus/english.txt",
         length: 35,
         err:    nil,
      },
      "empty corpus": {
         file:   "../corpus/empty.txt",
         length: 0,
         err:    gordle.ErrCorpusIsEmpty,
      },
   }

   for name, tc := range tt {
      t.Run(name, func(t *testing.T) {
         words, err := gordle.ReadCorpus(tc.file) #C
         if tc.err != err {
            t.Errorf("expected err %v, got %v", tc.err, err)
         }

         if tc.length != len(words) {
            t.Errorf("expected %d, got %d", tc.length, len(words))
         }
      })
   }
}
```

We are now happy, we have our corpus in a handy form, it is reading from a file that can be updated in the simplest way possible - just add a new word to it as a new line. Gordle now knows a list of words. If we pick one - and try to make it different every time - Claudio will face a different challenge every time he plays the game!

5.3.3 Pick a word

Every game of Gordle needs a random word for the player to guess. We have a corpus, all that's left is to select one word from our list.

Libraries implementing random number generators are under a lot of pressure, as they need to comply with very strict requirements. One would expect, for instance, a random number generator to generate numbers with the same probability and amount of time.

Go’s `math/rand` package provides a random number generator - but there is another package that also achieves this in Go’s standard packages - the `crypto/rand` package. The main difference is that the crypto package guarantees truly random numbers, while the `math` package generates pseudo-random numbers. Oh, and the crypto package is a lot more expensive. As a rule of thumb, for small non-critical applications, using the `math` package is perfectly fine. When it comes to passwords, tokens, or security-related objects, using the crypto package is recommended.

Both of Go’s `rand` packages expose, amongst others, an `Intn(n int)` function that returns a number between 0 (included) and n (not included). These packages are built on algorithms using a base value called source that can be overridden. Overriding it with something that changes every time will ensure we get a random number out of the library. Earlier versions of Go (prior to 1.20) required the `rand` package to be seeded, with a call to `rand.Seed(seed)`. The random number generator is now seeded randomly when the program starts - there is no real point in calling it. We’ll keep this `rand.Seed` in the code, for those who aren’t using the latest version of Go.

Now that we know how to get a random number, picking a random word in a list is straightforward - simply get the word at the random index.

```
index := rand.Intn(len(corpus))
```

The `pickWord` function is implemented as follows:

Listing 5.35 corpus.go: `pickWord()` method

```
// pickWord returns a random word from the corpus
func pickWord(corpus []string) string {
   index := rand.Intn(len(corpus))

   return corpus[index]
}
```

How to test a random func?

You know the importance of testing the core methods to make sure they are working properly before calling them into higher methods. `pickWord` will follow that trend, but there is a minor issue. When we execute tests, usually, we want to compare an output to a reference. `pickWord`, by design, has a non-deterministic output. When this happens, we have two solutions. We can change the behaviour of the random number generator from the test (but then we're not testing anything). Or we can assert a truth about the output: what we really want to test is whether the method returns a word from the list, or the results we get when calling "a lot" of times the random function follows the expected distribution. So, we will go for the second approach, and ensure that the word `pickWord` returns was indeed in the initial list. For this, we won't use Table-Driven Test, as we won't have a wide variety of cases.

Let's first write a helper function to verify a word is present in a list of words. There is no special trick here, we have to range over the list and, if the word corresponds to the input, immediately return true. Otherwise, we return false. This function, similarly to the previous two that compared slices, is also a very common one that we can make more *generic*.

Listing 5.36 corpus_internal_test.go: Helper `inCorpus()`

```
func inCorpus(corpus []string, word string) bool {
   for _, corpusWord := range corpus {
      if corpusWord == word {
         return true
      }
   }
   return false
}
```

Earlier, we shyly ventured into the world of the `golang.org/x/exp/slices` to use the `Equal` function. That `slices` package also offers a `Contains` function with a very similar signature to our `inCorpus`. We believe that practice makes perfect and that it doesn’t hurt to have your tiny implementation closeby.

With the help of this small function, we can now add a test that will ensure `pickWord` returns a word from the input slice.

Listing 5.37 corpus_internal_test.go: Test `pickWord`

```
func TestPickWord(t *testing.T) {
   corpus := []string{"HELLO", "SALUT", "ПРИВЕТ", "ΧΑΙΡΕ"}
   word := pickWord(corpus)

   if !inCorpus(corpus, word) { #A
      t.Errorf("expected a word in the corpus, got %q", word)
   }
}
```

Now we have done the implementation and covered the testing, we are ready to wrap it up! Do you remember that nasty hardcoded solution in the `Game` structure creation? It’s time to replace it by calling the `pickWord` method and passing the corpus as a parameter of `New()`.

We want Gordle to be independent and reusable by anyone with a list of words. For this reason, we'll pick the solution in the `New` rather than have it provided by the rest of the world. Even the `main` function doesn't know the hidden word! However, we must now ensure that the corpus is valid - it should have at least one word. If the corpus is empty, the `New` function won’t be able to create a playable game, and we need to return an error. This will change the `New()` function’s signature.

We are now also reaching the moment where `New()` does a lot. Not only does it create a `Game`, but it also initialises it. We won’t push it any further, and instead consider that it might be time to split it into two distinct functions, each with its responsibilities. For now, let’s just add that final cherry on top of the `New()` function:

Listing 5.38 game.go: Update `New()`with a corpus

```
// New returns a Game variable, which can be used to Play!
func New(reader io.Reader, corpus []string, maxAttempts int) (*Ga
   if len(corpus) == 0 {
      return nil, ErrCorpusIsEmpty
   }
   g := &Game{
      reader:      bufio.NewReader(reader),
      solution:    []rune(strings.ToUpper(pickWord(corpus))), //
      maxAttempts: maxAttempts,
   }

   return g, nil
}
```

Now, we have everything ready. Claudio's been waiting a long time to play, let’s adjust the call in the `main` function and give him the keyboard!

5.3.4 Let’s play!

There is very little left to do before the game is complete. Only a few changes in the `main` function remain to apply - we’ve got a corpus, and we need to parse it and feed it to Gordle’s `New()` function. Since this `New()` function now returns an error, we should take care of it. Let’s write a message on the error output and leave the `main()` function with a `return`.

Listing 5.39 main.go: Calling `readCorpus()` in main

```
package main

import (
   "bufio"
   "fmt"
   "os"

   "learngo-pockets/gordle/gordle"
)

const maxAttempts = 6

func main() {
   corpus, err := gordle.ReadCorpus("corpus/english.txt") #A
   if err != nil {
      _, _ = fmt.Fprintf(os.Stderr, "unable to read corpus: %s",
      return
   }

   // Create the game.
   g, err := gordle.New(bufio.NewReader(os.Stdin), corpus, maxAtt
   if err != nil {
      _, _ = fmt.Fprintf(os.Stderr, "unable to start game: %s", e
      return
   }

   // Run the game ! It will end when it's over.
   g.Play()
}
```

That’s enough typing from our side, time to let Claudio smash these keys frenetically, in search of one of Gordle’s secret words.

```
$ go run main.go
Welcome to Gordle!
Enter a 5-character guess:
sauna
⬜⬜⬜⬜⬜
Enter a 5-character guess:
waste
⬜⬜⬜⬜🟡
Enter a 5-character guess:
hello
⬜🟡⬜⬜⬜
Enter a 5-character guess:
terse
⬜🟡🟡⬜🟡
Enter a 5-character guess:
crept
⬜💚💚⬜⬜
Enter a 5-character guess:
freed
💚💚💚💚💚
🎉 You won! You found it in 6 attempt(s)! The word was: FREED.
```

5.4 The limit of runes

Claudio enjoyed this so much he wants to submit his list of words! He wants to share Gordle with his friend Mithali, who lives in India. He writes a small list of words, to make sure the program behaves as expected, using the Devanagari characters - which are used to write in Hindi, just like the Latin characters are used to write in English. The first word he writes is नमस्ते - “Namaste”, meaning “Hello”. It’s composed of four characters, but after updating the `main` function and reading from this new `hindi.txt` file, Claudio gets prompted “`Enter a 6-character guess:`”. He comes back to you, unhappy with the program. What’s happening?

Devanagari, as opposed to Latin, isn’t an alphabet. Instead, it is an abugida - a system in which (simply put) vowels alter consonants. If we look at the word नमस्ते, we can split it into its different sections: न is pronounced “na”, म is pronounced “ma”, and स्ते is pronounced “ste”. This last section is actually the combination of the “sa” letter, written स, without its “a” part, and the “ta” section, written त, which is here written ते, because the sound “e” must be present, as represented by the *matra* - a descending bar above the *shirorekhā*, the horizontal line.

The word combination here also is important - even though the spelling न + म + स् + ते would pronounce the same sound, the rules of Devanagari combine the last two symbols, “s” and “te”, into one: स्ते “ste”.

Now, let’s see how Go deals with the “नमस्ते” string:

Listing 5.40 Understanding the नमस्ते case

```
func main() {
   s := "नमस्ते"
   for _, r := range []rune(s) {
      fmt.Print(string(r)+" ")
   }
}
```

Running this small program produces the following output:

```
न म स ् त े
```

We can see that Go did indeed split the string into six runes. We’ve already seen four of them, those with the *shirorekhā*: they are called *swars* in Hindi. The other two are a bit cryptic - they represent a dotted circle with a decoration - something called a diacritic. In Devanagari, this is one way of representing *matras* (which include, but aren’t restricted to, vowels).

Diacritics are alterations to existing characters, they don’t have an existence on their own. English has some diacritics, mostly in borrowed words such as *déjà-vu* or *señor*: the accents on the first word’s vowels can’t be written without their supporting vowel, and the same goes for the tilde, which needs a letter to sit on.

As we can see, Go won’t merge the diacritic ् with the character स, when splitting the string into runes. This character remains two different runes for Go. So, what can Lucio do to help Claudio? Unfortunately, this is the limit of what the native `rune` type of the Go language can support. But this is precisely what the `golang.org/x` packages are for - extending the limits of what Go natively accepts. In our case, the package `golang.org/x/text/unicode/norm` provides a type `Iter` that can be used for these strings. With a bit more work on the code, Mithali will be able to play Gordle too!

5.5 Conclusion

Finally, we’ve completed our objective! We’ve written a command-line game that lets a user interact with it via the standard input. Our game reads data from a file containing words, selects one at random, and has the player guess the word. After each attempt, we provide visual feedback to help the player towards the solution. Whatever happens, we’ve tried to print clear messages to the player so they don’t get lost with what to do next.

5.6 Summary

- A `switch` / `case` statement is a lot more readable than a long sequence of `if` / `else if` / `else` statements. A `switch` can even be used instead of an `if` statement. We think that, if you need an `else` statement, you’re better off with a `switch` block.
- A command-line tool often needs to read from the console input. Go offers different ways of doing it, in this chapter, we used the `bufio.ReadLine` method, which reads an input line by line.
- Sentinel errors are a simple way of creating domain errors that can be exposed for other packages to check. It is a cleaner implementation than

creating exposed errors with `errors.New()` or `fmt.Errorf()`. To declare a new sentinel error type, declare a new type that is defined as a `string` (this makes creating new errors simple).

- Propagating an error to the caller is the way Go handles anything that steps out of the happy path. Functions that propagate errors have their last return value of their signature be an error. In the implementation of these functions, `fmt.Errorf("... %w", … err)` is the default way of wrapping errors. The `w` in `%w` stands for “wrap”.
- Any structure with a method with the following signature: `String() string` implements the `fmt.Stringer` interface. Any structure that implements the `Stringer` interface will be nicely printed by `fmt.Print*` functions.
- The `os` package provides a `ReadFile` function that loads a file’s contents as a slice of bytes. This function can be used for plain-text files, media files, files in XML or HTML format, etc.
- The `golang.com/x/exp/slices` package contains useful tools such as the `Equal` function or the `Contains` function. However, the documentation mentions that they could move out of `/x/exp` at any point. As we’ve seen, implementing the function for a specific use case isn’t too complex.
- A Go `string` can be parsed as either a slice of bytes, or as a slice of runes. The latter is recommended when iterating through the characters that compose it. Use `[]rune(str)` to convert the `str` string to a slice of runes. However, even this solution isn’t perfect and won’t always work. Best to first check the language you’re dealing with to select the best libraries to parse any text.
- All receivers of a specific type should be either pointer or value receivers. Using value receivers is only interesting if the structure is small in memory, as it will copy it. When in doubt, use pointer-receiver declarations.
- When writing table-driven tests, it is a very common practice to use a `map[string]struct{...}`. The key of the map, the `string`, describes the test case, and the `struct` is an anonymous structure that contains the fields necessary for your test case.
- Getting a random number can be achieved by both the `math/rand` and the `crypto/rand` packages. Anything related to security, cyphering, or cryptographic data should use the `crypto/rand` package, while the

`math/rand` is cheaper to use.

- When working with random numbers, make sure you're using Go 1.20 or more. Otherwise, be explicit about setting the seed with a call to `rand.Seed()`. An usual value for the seed used to be the current nanosecond, retrieved with `time.Now().Nanosecond()`.
- Taking a step back, away from a screen, and writing pseudo-code with potatoes and arrows is valuable, and helps to see the bigger picture and imagine tricky scenarios that might prove or disprove an algorithm.

6 Money converter: CLI around an HTTP call

This chapter covers

- Writing a CLI
- Making an HTTP call to an external URL
- Mocking an HTTP call for unit tests
- Grasping floating-point precision errors
- Parsing an XML-structured string
- Inspecting error types

A long list of websites nowadays exposes useful APIs that can be called via HTTP. Common examples are the famous open-source system for automating deployment Kubernetes, weather forecast services, international clocks, social networks, online databases such as BoardGameGeek or the Internet Movie Database, content managers like WordPress, the list is long. A small number of them also provide a command-line tool that calls these APIs. Why? Even though nice and clickable interfaces are wonderful, they are still very slow. Here's an example: when we look up a sentence in our favourite search engine, it still takes an extra click to access the first link that isn't an advert, or the first one we haven't opened yet. The terminal shell, on the other hand, allows us to manipulate inputs and outputs of programs - and even to combine them, which reduces the number of command lines and helps automate more of our work.

In this chapter, we will create a CLI tool that can convert amounts of money. Starting with a broad view of what we want our tool to achieve, we will begin by defining the main concepts: what is a currency, and how do we represent it? What does it mean to convert money? We'll need a change rate, how do we get it? What should our input and our output be? How do we parse the input? As we'll see, some precaution is required when manipulating floating-point precision numbers. An early disclaimer is required here: this project is a tutorial project and shouldn't be used for real-life transactions.

Requirements

- Write a CLI tool
- Takes 2 currencies and an amount
- Returns the converted amount
- Safely rounds to the precision of the given currencies
- Currency should be provided according to the ISO-4217 standard

Limitations

- The input amount must be defined with digits only, and one optional dot as a decimal separator. We could extend later with spaces, underscores, or apostrophes.
- We only support decimal currencies. Sorry, ariaries and ouguiyas.

Usage example: change -from EUR -to USD 413.9

6.1 Business definitions

One approach to building software is to start with the business definitions. Indeed, understanding what you are trying to achieve is a good way to avoid solving a different problem. In our case, the big picture is that we want a command-line tool that takes an amount of money expressed as a quantity and its currency, and another currency as the target, and that will present the converted amount as its output. Our business words include “amount”, “quantity” (a decimal value), “currency”, and “convert”.

As we want to convert a certain amount of money between two currencies, we will start by defining what a currency is, what an amount is, and export a `Convert` function that takes these as input.

New project, new module

By now, you know how to initialise a new module. Create your folder, initialise it:

```
go mod init learngo-pockets/moneyconverter
```

Let’s start simple and create a `main.go` file at the root of the project, with the package name `main` and the usual `func main()`.

As long as we're only having a single binary, it's fine to have the main.go file at the root directory. For tools that expose several binaries, the common place for main function is in `cmd/{binary_name}/main.go`.

6.1.1 `money.Convert` converts money

While the `main` function is responsible for running the executable in a terminal, reading input and writing output, most of the logic will reside inside a subpackage: the `money` package’s scope will be the heart of our domain logic. The `main` package will be in charge of calling this `money` package. As a general rule, imagine that the package can be reused, for example if you start writing a fancy user interface, but don’t over-engineer it until you know what you actually need.

We now create a folder named `money` containing one file that will expose the converter’s entrypoint of the package, the `Convert` function. We can start writing the contents of `convert.go`: it has to be an exposed function.

Listing 6.1 convert.go: signature of the converter’s entrypoint

```
package money

// Convert applies the change rate to convert an amount to a targ
func Convert(amount Amount, to Currency) (Amount, error) {
    return Amount{}, nil #A
}
```

The two parameters are hopefully self-explanatory. We want to convert a given amount into the currency `to`. The function will return an amount of money or, if something goes wrong, an error.

In order to make our project compile, we need to define the two custom types: `Amount` and `Currency`. We can already anticipate that they will hold a few methods, e.g. `String()` to print them out. This calls for a file for each of the types, ready to hold their future methods.

Currency

The ISO-4217 standard associates a three-letter code to every currency used out there in the real world, for example USD or EUR. As this will be our input, we can start by using that three-letter code to define our `Currency` type that will represent the currency code with a field of type `string`.

Create a `currency.go` file in the same package and add the following structure.

Listing 6.2 currency.go: Currency definition

```
// Currency defines the code of a currency.
type Currency struct {
    code string #A
}
```

Immutability

The code string is hidden inside the struct for any external user. We will continue building all of our types so that they stay immutable, meaning that once they are constructed, they cannot be changed.

We do that to make the code more secure for the package's users (that is, us): if we have 10 euros, they will not suddenly become 19.56 Deutsche Mark because we called a function on them. Immutability also makes the objects inherently thread-safe.

Amount and Decimal

As our tool will convert money, we need to be able to represent a quantity of money in a given currency to convert. This is what the `Amount` type needs: a decimal value and a currency. Let's create the `amount.go` file in the package `money` and add the following structure.

Listing 6.3 amount.go: `Amount` struct

```
// Amount defines a quantity of money in a given Currency.
```

```
type Amount struct {
    quantity Decimal #A
    currency Currency #B
}
```

Why is `quantity` not simply a float? For a start, if we want to attach some methods to it, it needs to be a custom type. Second, we will see in the next part that floats are dangerous - there are many possible ways to save this number and if we want to make room for later optimisation, we need to hide the entrails behind a custom type.

As we don't know yet how we'll write these internal details, let's leave it empty for now. It is not necessary to have one struct per file, or file named after the main struct they contain, but it is a good way for maintainers to find what they are looking for.

Listing 6.4 decimal.go: `Decimal` struct

```
// Decimal is capable of storing a floating-point value.
type Decimal struct {
}
```

At this point your project's tree should have one directory and a total of 5 go files:

```
$ tree
.
├── go.mod
├── main.go
└── money
    ├── amount.go
    ├── convert.go
    ├── currency.go
    └── decimal.go
```

Everything should now compile with the `go build -o convert main.go` command. Congratulations, we've defined the business entities of our library.

Before we start filling it up, let's write a test.

Testing `Convert`

Testing a function that does nothing is pretty preposterous, you might think. We would like to argue that if you can’t write a test that is easy to understand and to maintain, your architectural choices are on the wrong path. If writing the test is a mess, rethink your code organisation even before starting to work on the business logic. Unfortunately, easy testing is not a guarantee of a good architecture, or the world would be a better place.

More for learning reasons than anything else, we chose to use a validation function here.

Validation function

In Chapter 2, we learned about writing Table-Driven Tests, where you define a list of test cases, each in its instance of a custom structure. The expected return value is directly in the structure in each case. Usually, you will see a function returning a pair of a value and an error. A validation function can ensure that a returned result is valid, on a case-by-case basis. Sometimes, you want to dig deep into the returned value. Sometimes, you want to ensure the error is the expected one. Most of the time, it's pointless to fully check both the returned value AND the error - only one of them will be set.

A validation function is a field from the test case structure and takes as parameter `*testing.T` and all necessary parameters for the check. It does not return an error but fails directly if something wrong happens. For our `Convert` function, we will need the value we got and the error.

```
    tt := map[string]struct {
        // input    fields
        validate func(t *testing.T, got money.Amount, err error)
    }{
```

Now wait a second. Is that field actually a function? Yes! Go allows for the definition of variables of many types, including functions of specific signatures. You can see examples of what it looks like in test cases below.

Listing 6.5 convert_test.go: Check the testability of the design choices

```
package money_test
```

```
import (
    "testing"

    "learngo-pockets/moneyconverter/money"
)

func TestConvert(t *testing.T) {
    tt := map[string]struct {
        amount   money.Amount
        to       money.Currency
        validate func(t *testing.T, got money.Amount, err error)
    }{
        "34.98 USD to EUR": {
            amount: money.Amount{}, #B
            to:     money.Currency{}, #B
            validate: func(t *testing.T, got money.Amount, err er
                if err != nil {
                    t.Errorf("expected no error, got %s", err.Err
                }
                expected := money.Amount{} #D
                if !reflect.DeepEqual(got, expected) {
                    t.Errorf("expected %v, got %v", expected, got
                }
            },
        },
    }

    for name, tc := range tt {
        t.Run(name, func(t *testing.T) {
            got, err := money.Convert(tc.amount, tc.to)
            tc.validate(t, got, err) #E
        })
    }
}
```

Enough suspense, let’s code this `Decimal`.

6.2 How to represent money

How should we represent a given amount of money? Say, 86.33 Canadian dollars. Or, when using the ISO-4217 standard, 86.33 CAD.

A first idea could be to simply use a float. Unfortunately, there are two problems in this naive approach.

First, we are not preventing anyone from declaring 86.32456 CAD, which bears no real-world meaning. The smallest subunit of this Canadian dollar is the cent, a hundredth of a dollar. Anything smaller than 0.01 CAD must be rounded one way or another. We want to prevent this nonsense from happening and prevent it *by design*. This means that the way we build this `Decimal` struct should prevent it from ever happening, not because of safeguards that we may accidentally remove, but because it should simply be impossible.

Second, the precision of the floating point numbers is worth diving into.

Floating-point numbers

Using integers in computer programming is straightforward - all you need to pay attention to is whether they're not too big. Using floating-point numbers is a very different story, and, whenever using floating-point numbers one should always assume one won't get exactly what one expects.

In computer science, the IEEE-754 standard, adopted by Go, defines an implementation of floating-point numbers arithmetic. Go offers two flavours of floating-point numbers: `float32` (encoded as 4 bytes) and `float64` (encoded as 8 bytes). Due to the implementation of IEEE-754, these two types have a precision - a number of guaranteed correct digits - when written in base 10.

`float32` guarantees a precision of only six digits. This means anything farther down the line from the first non-zero digit, in a `float32` variable, can safely be considered gibberish. Here are some examples:

`123_456_789` (around a hundred million) - the first non-zero digit is the leading `1`, the seventh digit is the `7`. If we write `fmt.Printf("%.f", float32(123_456_789))`, we get the output `123456792`. As we can see, we've lost the correct digits after the `7`.

`0.0123456789` - the first non-zero digit is the `1` in the hundredth (second after the decimal separator) position. If we write `fmt.Printf("%.10f", float32(0.0123456789))`, we get the output `0.0123456791`. Again, only the first seven non-zero digits were safely encoded, the rest is lost.

When using `float64` variables, the precision is 15 guaranteed digits, which would be around a few seconds if the dinosaurs had started a `float64` clock when they went extinct, 33 million years ago. You might think this is way too accurate to ever be imprecise - sometimes, it simply isn't: using a `float64` to represent the mass of Earth would make the weight of all the gold on Earth a negligible part of these gibberish numbers.

Some numbers will have an exact representation in IEEE 754 - numbers that are combinations of inverses of powers of 2 - up to a certain point. For instance, 0.625, which is ½+⅛, prints as 0.625000… - and all digits after the 5 are zeroes. But most fractions can't be written as sums of inverses of powers of two, and thus, most decimal numbers will be incorrectly represented, when using `float32` or `float64`.

We can reach the limits of `float32` rather early: the following line doesn't print the expected 1.00000000. Even though the first 7 digits are correct (0.9999… is equal to 1), the eighth isn't.

```
fmt.Printf("%.8f", float32(1)/float32(41)*float32(41))
```

Sometimes, a precision of seven significant digits will be enough. When averaging grades, using `float32` works perfectly. Similarly, an error of a millionth of a dollar would seem tolerable, if we were to use `float32`s in our project, wouldn't it? But what if the amount to convert is not 1 dollar, but ten million dollars? In this case, the error we introduce by using `float32` would already be a few dollars. Would that still be acceptable?

Back to money

In order to represent an amount of money, it is therefore always preferable to default to fixed precision, unless you know for certain that the floating point will not cause any harm - arithmetic operations on integers are correct to the unit - as long as they're "not too big". If we want to represent billions, which is close to the maximum value of an `uint32` - 2^32, or around 4 billion - the package `big` has a few types to represent really big numbers, for example `big.Int` or `big.Rat`, respectively for integer and rational. In our case, let's keep it simple and use regular integers: we'll accept the fact that we are not decillionaires as a limitation.

When it comes to operations, there are a few things we can take for granted, and others that we should not.

Floating-point number operations

Multiplying or dividing floating-point numbers is usually fine. The problems start happening when adding and subtracting one to another. Here's a simple scenario: a bank has encoded the money in their customers' accounts with `float32`. A very, very rich customer decides to buy ice cream with their credit card. The ice cream costs 1 euro. At that moment, the customer had a hundred million euros in their account. On the bank's side, the operation `float32(100_000_000) - float32(1)` is executed. At their surprise, the customer realises they still have a hundred million euros, as if the payment of 1 euro had never happened. This is due to the fact that we have 8 significant digits in the customer's bank account before we reach the unit, and the subtraction of 1 is lost in the noise of `float32`'s precision, which guarantees only 7 correct digits.

One of the most common mistakes programmers do when using floating-point numbers is using the == operator as a comparator. Since a floating-point number isn't properly represented, how can we hope it will be equal to another badly represented number? The safe way of comparing floating-point numbers is to take into account the precision: if two floating-point numbers are within the precision range of the largest, they should be considered equal, and, otherwise, they should be considered different. Here's a quick trigonometric example. Don't worry, we won't go too far. The sine function returns 0 when evaluated on any multiple of π. Let's see how this looks like when calling the `math.Sin` function (it returns a `float64`) in Golang:

```
fmt.Println(math.Sin(math.Pi))
```

This returns a very small, but clearly not null, value - 1.2246467991473515e-16. Any mathematician would be offended by this result. However, as computer scientists, we know that, instead of comparing this number to the exact 0, we should compare it to 0 within the range of the precision of a `float64`. This is how we could check if sin(π) is close enough (to the precision of 15 digits) to 0 that we can consider them non-distinguishable:

```
fmt.Println(math.Abs(math.Sin(math.Pi)-0) < math.Pow(10, -15))
```

6.2.1 Decimal implementation

That was a lot of theory. Knowing all this, there are a number of different possibilities for implementing this `Decimal` struct. What we chose to do here was to split the integer and decimal parts. Fortunately, as the contents of the struct are private to the package, users don't depend on our implementation and it should be possible to come back on this decision anytime without breaking our exposed API.

This is another reason why hiding the internal details behind a custom type is generally a good idea. In practice, we could start with imprecise floats and refactor later. Let's not, though - we already know that floats can introduce imprecision, and we wouldn't want that, right?

Integer and decimal parts are two different numbers. But how do we know what this decimal represents in the currency? The satoshi is currently the smallest unit of the bitcoin currency recorded on the blockchain and it is one hundred millionth of a single bitcoin (0.00000001 BTC), far from the generally accepted hundredth of euros, francs, hryvni or rupees. We will keep this precision of the decimal part as a power of ten. This precision is a number that will range between 0 (we'll always want to be able to represent 1.0) and a value that isn't too big. Since we don't need to represent numbers bigger than 10^30, we don't need to store an exponent of 10 that is bigger than 30. For small numbers such as this, using a `byte` is a safe choice. A `byte`'s maximum value is 255, and we're definitely not going to need that power of 10.

Listing 6.6 decimal.go: `Decimal` struct implementation

```
// Decimal can represent a floating-point number with a fixed pre
// example: 1.52 = 152 * 10^(-2) will be stored as {152, 2} #A
type Decimal struct {
    // subunits is the amount of subunits. Multiply it by the pre
    subunits int64 #B
    // Number of "subunits" in a unit, expressed as a power of 10
    precision byte #C
}
```

We should be able to update the test to add real quantity (as a `Decimal`) and currency (as a `Currency`) values to it. But can we?

Constructing the Decimal

The fields of the structs are not exposed, and we really want to keep it that way. We need a building function for `Decimal` and `Amount`.

Let’s start with the one with no dependency: `Decimal`.

What will it take as parameters, though? If we ask for three ints for integer part, decimal part and precision, there will be no way of changing this int-based implementation later. We can expect the amount to be expressed as a string in the caller’s input, in order to avoid floating point imprecision from the start.

One common way of creating a struct in Go is the `New` function, as seen in Chapter 4. Another, when everyone carries strings around, is the `Parse` keyword, as found, for example, in `time.ParseDate` or `url.Parse`. Our `Decimal` is a good candidate for this pattern. Let’s write a function that will take a string as its parameter and return a `Decimal` that the string represents.

Parse a decimal number

Write a `ParseDecimal` function in the `decimal.go` file that returns a `Decimal` or an `error`. Don’t forget to write a test. You will need `strconv.ParseInt` to convert strings to integers, and `strings.Cut`, to split a string on a separator. In a terminal, you can run `go doc strconv.ParseInt` and `go doc strings.Cut` for some inspiration. Remember: We want to use an `int64` to represent the value, as this is the largest of the basic types. Here’s a short description of the various steps we need to go through in `ParseDecimal`:

```
ParseDecimal(string) (Decimal, error) {
// 1 - find the position of the . and split on it.
// 2 - convert the string without the . to an integer. This could
// 3 - add some consistency check
// 4 - return the result
```

There are several ways of splitting the string “`18.95`” into “`18`” and “`95`”, and the `strings` package offers two: `Cut` and `Split`. Why are we using `strings.Cut` and not `strings.Split`? We appreciate the simplicity of the former, and it is a lot more convenient to use when the separator is not present in the input string.

On one hand, `Cut` will break the string into two parts, right after the first instance of the given separator, and return a boolean telling whether the separator was found. If the string does not contain the separator, the function returns the full string, an empty string and `false`.

On the other hand, `Split` breaks the string into substrings, delimited by separators, and returns a slice of these substrings. If the separator does not appear, `Split` returns a slice with only one string. Finally, if the separator or the string is empty, it returns an empty slice.

Here is an example of the behaviour of `strings.Cut` and `strings.Split` on three different strings: one where the separator doesn’t appear, one where the separator appears once, and one where the separator appears more than once.

Table 6.1 Examples of strings.Cut and strings.Split

Value	Output of fmt.Println(strings.Cut("{{Value}}", "p")	Output of fmt.Println(strings.Split("{{Value}}", "p")
banana	`"banana", "",  false`	`[]string{"banana"}`
grape	`"gra", "e", true`	`[]string{"gra", "e"}`

apple	`"a", "ple", true`	`[]string{"a", "", "le"}`

You can see on the grape example for `strings.Split` that the length of the resulting slice is the number of “p” +1. It is also interesting to notice that `Strings.Split` will not discard empty strings, as you can note on the “apple” example.

Since `ParseDecimal` can return an error, let’s take some time to go through what are error types and how to check them.

Error types

It is nearly part of the definition of parsing: there might be problems. If the user sends us letters, what can we do? Return an error.

The errors package exposes a useful method:

```
errors.As(err error, target any) bool
```

It reports whether err's concrete value is assignable to the value pointed to by the target. It becomes very handy when you are using a library and want to compare the type of error with the domain error, for example, is this error coming from the money package? When you are the writer of the library, it is polite to expose a domain error for the users so that they can adapt the behaviour in their code if the error is coming from your library.

In our case, we are the writers of the library and as polite people, we will expose a domain error type in the package money and implement the interface `Error` from the standard errors package.

Let’s create a package-specific error type.

Listing 6.7 errors.go: Custom error type for the package `money`

```
package money
```

```
// Error defines an error.
type Error string

// Error implements the error interface.
func (e Error) Error() string {
    return string(e)
}
```

Not a lot of effort, and worth the simplicity in usage. The consumer can now check whether a returned error is from this package.

```
var moneyErr money.Error
if errors.As(err, &moneyErr) {
// ...
}
```

Finally, before we start writing code, let’s think of errors that consumers will be able to understand. The first would be returned if the string to parse is not a valid number. The second will be raised if we try to deal with values that are too big. Having a limit is a good idea. It will help ensure that we don’t exceed the maximum value of an `int64`, especially when multiplying to `Decimal` variables together, which is bound to happen.

Listing 6.8 decimal.go: Expose errors

```
const (
    // ErrInvalidDecimal is returned if the decimal is malformed.
    ErrInvalidDecimal = Error("unable to convert the decimal")

    // ErrTooLarge is returned if the quantity is too large - thi
    ErrTooLarge = Error("quantity over 10^12 is too large")
)
```

Now that we’ve exposed the errors we could think of, let’s write the function.

Listing 6.9 decimal.go: `ParseDecimal` function

```
// ParseDecimal converts a string into its Decimal representation
// It assumes there is up to one decimal separator, and that the
func ParseDecimal(value string) (Decimal, error) {
    intPart, fracPart, _ := strings.Cut(value, ".") #A

    // maxDecimal is the number of digits in a thousand billion.
    const maxDecimal = 12

    if len(intPart) > maxDecimal {
        return Decimal{}, ErrTooLarge
    }

    subunits, err := strconv.ParseInt(intPart+fracPart, 10, 64) #
    if err != nil {
        return Decimal{}, fmt.Errorf("%w: %s", ErrInvalidDecimal,
    }

    precision := byte(len(fracPart)) #D

    return Decimal{subunits: cents, precision: precision}, nil
}
```

How does our precision variable work? Let’s look at a few examples.

Table 6.2 Precision examples

value (string)	precision
5.23	2
2.15497	5
1	0

As you can see, the precision of the parsed number is simply the number of digits after the decimal separator. If the user gives us 1.1 dollars, it’s a bit weird but it’s valid. Note that converting it to dollars will give $1.10 back, with one more digit, because dollars are divided in hundredths.

Testing ParseDecimal

In order to check the results, we need to access the unexposed fields of the `Decimal` structure. To achieve this, our test needs to reside in the same package and be aware of the implementation.

The test can look like this.

Listing 6.10 decimal_internal_test.go: Test `ParseDecimal`

```
func TestParseDecimal(t *testing.T) {
    tt := map[string]struct {
        decimal  string #A
        expected Decimal #B
        err      error #B
    }{
        "2 decimal digits": {
            decimal:   "1.52",
            expected: Decimal{
                integerPart: 1, decimalPart: 52, precision: 2,
            },
            err:      nil, #A
        },
        "no decimal digits": {...}, #B
        "suffix 0 as decimal digits": {...}, #B
        "prefix 0 as decimal digits": {...}, #B
        "multiple of 10": {...}, #B
        "invalid decimal part": {...},
        "Not a number": { #C
            decimal: "NaN",
            err:    ErrInvalidValue,
        },
        "empty string": { #C
            decimal: "",
            err:    ErrInvalidValue,
        },
        "too large": { #C
            decimal: "1234567890123",
            err:    ErrTooLarge,
        },
    }

    for name, tc := range tt {
        t.Run(name, func(t *testing.T) {
            got, err := ParseDecimal(tc.decimal)
            if !errors.Is(err, tc.err) {
                t.Errorf("expected error %v, got %v", tc.err, err
            }
            if got != tc.expected {
                t.Errorf("expected %v, got %v", tc.expected, got)
            }
        })
    }
}
```

Let’s have a look, in particular, at the test named “suffix 0 as decimal digits”. In it, we parse the value 1.50, and it gets converted to a Decimal with 150 subunits, and a precision of 2. This is correct, but is it really the best we can do? We’re dealing with a decimal number here, there is no point in keeping these extra zeroes, as they don’t bring any information. Let’s simplify a decimal, with the use of a new method for the `Decimal` type, called `simplify`. This method will be tested via the tests on `ParseDecimal`. `simplify` will remove zeroes in the rightmost position as long as this doesn’t affect the value of the Decimal. 32.0 should be simplified to 32, but 320 should remain 320.

Listing 6.11 decimal.go: Method `simplify`

```
func (d *Decimal) simplify() {
    // Using %10 returns the last digit in base 10 of a number.
    // If the precision is positive, that digit belongs to the ri
    for d.subunits%10 == 0 && d.precision > 0 {
        d.precision--
        d.subunits /= 10
    }
}
```

That was an important first step. Remember to run tests and commit - with explicit messages - regularly, especially upon completion of a piece of the deliverable. After this complex logic, writing the `Currency` builder is going to be easy.

6.2.2 Currency value object

In order to understand what this input number means, we need a currency. As mentioned before, each currency has a fixed precision and cannot express any value smaller than this precision: 0.001 CAD isn’t an amount that we want to represent, as it doesn’t exist in real life (it may be used in very particular use cases such as statistics, but not in transactions). There is no way to guess each currency’s precision. We could retrieve this information through a service, read a database, read a file... or we could simply keep a hard-coded list of the exceptional currencies and default to the hundredth, which most use. Hopefully, currencies won’t start using new subunits too often. After all, this is a pet project, we are not planning (yet) to support funky historical currencies.

First, add the currency’s precision to the struct. It’s going to be a value between 0 and 3. A `byte` is again a good choice here.

Listing 6.12 currency.go: Add `precision`

```
// Currency defines the code of a money and its decimal precision
type Currency struct {
    code      string
    precision byte #B
}
```

We can use the `Parse` prefix, as we take a `string` in and return a valid object or an error. Let’s have a thought about the error(s) we might have to return. If an invalid code currency is given, we should be able to return an error. Let’s create a new constant, `ErrInvalidCurrencyCode`, of `moneyError` type. As you can see, the proposed name of the error begins with a capital, meaning it is exposed. This allows this package’s consumer to check against it. Then, we can create a function named `ParseCurrency` that will take the given currency code, as a string, and return a `Currency` object and an `error`. The first validation consists of checking if the code is composed of 3 letters; if it isn’t, we can directly return our new error. Otherwise, we’ll switch on the possible code currencies and return the `Currency` object with their respective precisions. We will assume that the default case is 2, as most of the currencies have a precision of 2 digits.

Listing 6.13 currency.go: Function for supported currencies

```
// ErrInvalidCurrencyCode is returned when the currency to parse
const ErrInvalidCurrencyCode = moneyError("invalid currency code"

// ParseCurrency returns the currency associated to a name and ma
func ParseCurrency(code string) (Currency, error) {
    if len(code) != 3 { #B
        return Currency{}, ErrInvalidCurrencyCode
    }

    switch code {
    case "IRR":
        return Currency{code: code, precision: 0}, nil
    case "CNY", "VND": #C
        return Currency{code: code, precision: 1}, nil
    case "BHD", "IQD", "KWD", "LYD", "OMR", "TND": #C
        return Currency{code: code, precision: 3}, nil
    default:
        return Currency{code: code, precision: 2}, nil #D
    }
}
```

Again, don't trust this tool in production. Validating the currency in real life should be done against a list that can be updated without touching the code. What we could do without making this project too big to fit in a pocket: we could go further and make sure the letters are actually capitals of the English alphabet. Actually, let's make it an exercise.

Exercise 6.1 Make sure the currency code is made of 3 letters between A and Z. You can use the `regex` package if you want to make things complicated, or check that each of the 3 bytes is between 'A' and 'Z' included.

Have we properly tested everything? Not yet. Try writing a test for the parser yourself before looking at our version.

Listing 6.14 currency_internal_test.go: `TestParseCurrency` function

```
package money

import (
    "errors"
    "testing"
)


func TestParseCurrency_Success(t *testing.T) { #A
    tt := map[string]struct {
        in       string
        expected Currency
    }{
        "hundredth EUR":   {in: "EUR", expected: Currency{code: "
        "thousandth BHD": {...},
        "tenth VND":      {...},
        "integer IRR":    {...},
    }

    for name, tc := range tt {
        t.Run(name, func(t *testing.T) {
            got, err := ParseCurrency(tc.in)
            if err != nil {
                t.Errorf("expected no error, got %s", err.Error()
            }

            if got != tc.expected {
                t.Errorf("expected %v, got %v", tc.expected, got)
            }
        })
    }
}


func TestParseCurrency_UnknownCurrency(t *testing.T) { #B
    _, err := ParseCurrency("INVALID")
    if !errors.Is(err, ErrInvalidCurrencyCode) {
        t.Errorf("expected error %s, got %v", ErrInvalidCurrencyC
    }
}
```

This time we are not using validation functions but separating the success cases from the one error case. It is mostly a matter of taste - the only criteria, as usual, are whether the next reader will understand what we are testing and find it easy to add or change a test case. By calling `t.Run` we also make sure that all test cases can be run separately.

If your tests pass, don't forget to commit.

We have a number, we have a currency, we can put them together and make an `Amount` of money.

6.2.3 NewAmount

An amount is a decimal quantity of a currency. As mentioned before, a decimal can be incompatible with a currency, if its precision is too large, for instance, we shouldn't allow for the creation of an Amount of decimal quantity 19.8875 (a number with a precision of 4) Canadian dollars, since a Canadian dollar's subunit is a cent (a precision of 2). Building an object that is not valid makes no sense: it is the role of the `New` function to return either something valid or an error.

Listing 6.15 amount.go: `NewAmount` function

```
const (
    // ErrTooPrecise is returned if the number is too precise for
    ErrTooPrecise = moneyError("quantity is too precise")
)

// NewAmount returns an Amount of money.
func NewAmount(quantity Decimal, currency Currency) (Amount, erro
    if quantity.precision > currency.precision { #A
        // In order to avoid converting 0.00001 cent, let's exit
        return Amount{}, ErrTooPrecise
    }
    quantity.precision = currency.precision #B

    return Amount{quantity: quantity, currency: currency}, nil
}
```

The test should be quite straightforward, so we won't give you our version here. Of course you can still find it in the book's code repository. Don't forget to cover the error case and run the test with coverage to validate you did not miss anything.

```
go test ./... -cover
```

Update the external test

Now, the last step before writing the actual conversion is to update the test of `Convert`. As it is testing `Convert` and neither `ParseDecimal`, `ParseCurrency` nor `NewAmount` (which are already covered by their own internal tests), what will we do with the potential errors? It is not the role of the `TestConvert` function to check the different values that `ParseNumber` and friends can return, but it needs to deal with the errors.

One option to avoid dealing with these errors would be to write a function that builds the required structures without checking anything, because you know that your test cases are valid. In order to build them, it needs to live in the `money` package and be exposed to the `money_test` package. But then what would prevent consumers from using that test utility function and send you invalid values? It completely invalidates the actual builders that we wrote in this chapter section. Let’s not choose this dangerous option.

An alternative is to call the function and ignore the error:

```
number, _ := money.ParseDecimal("23.52")
```

This is practical and doesn’t expose anything dangerous, but it raises a problem: if we give our test an invalid value by mistake, there’s no way we detect it. We would be testing on the zero values returned by the function alongside the error without knowing it, resulting in flaky tests. Let’s, again, not choose this option.

As we need to call a chain of different builders, we will instead write a test helper function, and we will tell Go about it. The `testing.T` object that we use for unit testing has a `Helper` function: according to the documentation (which can be obtained with `go doc testing.T.Helper`), *Helper marks the calling function as a test helper function. When printing file and line information, that function will be skipped.* It means that you will be able to see which test broke, rather than this helper function’s line number.

Listing 6.16 convert_test.go: Helpers to parse `Currency` and `Amount`

```
package money_test

import (
    "testing"

    "learngo-pockets/moneyconverter/money"
)

func mustParseCurrency(t *testing.T, code string) money.Currency
    t.Helper() #A

    currency, err := money.ParseCurrency(code)
    if err != nil {
        t.Fatalf("cannot parse currency %s code", code) #B
    }

    return currency
}

func mustParseAmount(t *testing.T, value string, code string) mon
    t.Helper() #A

    n, err := money.ParseDecimal(value)
    if err != nil {
        t.Fatalf("invalid number: %s", value)
    }

    currency, err := money.ParseCurrency(code)
    if err != nil {
        t.Fatalf("invalid currency code: %s", code)
    }

    amount, err := money.NewAmount(n, currency)
    if err != nil {
        t.Fatalf("cannot create amount with value %v and currency
    }

    return amount
}
```

As you can see we are not using `t.Fail` but `t.Fatal`, which stops the test run immediately.

We can now give actual values to the `Convert` function's test. The return value is still nothing, though.

Listing 6.17 convert_test.go: Calling `mustParse` in the test case

```
"34.98 USD to EUR": {
    amount:          mustParseAmount(t, "11.22", "USD"), #A
    to:              mustParseCurrency(t, "EUR"), #B
    validate: func(t *testing.T, got money.Amount, err error) {
        if err != nil {
            t.Errorf("expected no error, got %s", err.Error())
        }
        expected := money.Amount{}
        if !reflect.DeepEqual(got, expected) {
            t.Errorf("expected %q, got %q", expected, got)
        }
    },
},
```

Does it compile?

Does it run?

Does it pass tests?

Good job. This is a good time to commit your work.

6.3 Conversion logic

We are happy (or at least OK) with the API of this package, and the objects that we have in hand are guaranteed valid and supported. Let’s now have it really convert the money. For the first version, until we can actually run the tool, we will hardcode an exchange rate. After validating the base logic, we’ll be able to call a distant server holding the truth. For this distant server, we’ve decided to use the European Central Bank, which is free of use, doesn’t require authentication, and is likely to still be around in a couple of years.

6.3.1 Apply change rate

Of all the different entities that we built, which is responsible for applying a change rate?

This logic could belong to the `Amount` structure. It would know how to create a new `Amount` with a new value. Of course, amounts should be immutable and we need to make sure that the input amount is not modified by the operation. But does this option make sense conceptually? Would you expect a sum of money in a given currency to tell you what it is worth in another? Would you expect your 10 Sterling pound note to tell you “Hey, I’m worth roughly 10 US dollars today”? Probably not. You would go to an exchange office, give it your note and expect another back with a handful of coins. If it doesn’t make sense conceptually, then it will be harder to understand for future maintainers.

Instead, let’s write the exchange office as a function that will be called by

`Convert`.

Implement applyExchangeRate

As mentioned earlier, we don’t want to use float64 for this piece of the logic. It’s the most sensitive, and we want to ensure we do exact maths, without losing any precision on the values we handle. On one side, we have our `Amount`’s `quantity` field of type `Decimal`, and on the other side, we have an exchange rate that will be retrieved in a remote call. We must also provide the target currency, as that is where the precision of the output amount is stored.

How should we express that rate? Exchange rates published by the European Central Bank have up to 7 figures, which means we could safely store it in a `float64` variable. A `float32` might not be enough for currencies that use 7 digits - who knows why an eighth digit wouldn’t be added. We’ve already created a specific type for floating-point numbers with high precision, `Decimal`. It would be better to use that. Since this variable won’t represent a “normal” decimal number, we might as well use a specific type for it, in order to best describe its purpose. We can even push the zeal to the point of creating a new file for it, but at this point even the most adamant advocate for small files among these authors will admit that it can also live in the `convert.go` file, just after the exposed method.

A note on code organisation: you always want to have the exposed method first in a file, as it is easier to read code from its entrypoint. If you have multiple exposed functions in the same file, you may want to start with an exposed function and keep the private functions it calls just after.

Listing 6.18 convert.go: ExchangeRate type

```
// ExchangeRate represents a rate to convert from a currency to a
type ExchangeRate Decimal
```

This leads to an explicit signature for the function. `applyExchangeRate` is in charge of multiplying the input quantity by the change rate and returning an `Amount` compatible with the target `Currency`. We will first need a function to multiply a `Decimal` with an `ExchangeRate`, and then we’ll have to adjust the precision of that product to match the `Currency`’s.

To multiply a decimal with an exchange rate, we converted the exchange rate to a `Decimal`, and performed some arithmetics: the result of the multiplication is the product of the values, and the precision of the returned decimal is the sum of the precisions. We’ve done the following:

```
20.00*4.0 = {subunits: 2000, precision:  2}*{subunits: 40, precis
          = {subunits: 2000*40, precision: 2+1}
          = {subunits: 80_000, precision: 3}
```

The first point to notice, here, is that we have obtained “80” by using a lot more digits than necessary. Indeed, 80 is equal to 80.000, but we don’t really need this precision. We can make use of the method `simplify` here again, when performing the multiplication

The second point to notice is that we have a precision that doesn’t yet take into account any information about currencies. All we’ve done so far is multiplying decimal numbers. In `applyExchangeRate`, we’ll therefore need to adjust the result of `multiply` to give it the precision of the target currency. For this, we’ll have to multiply (or divide) by the difference of precision between our target currency and the result of the exchange rate multiplication. Of course, we could have a direct call to `math.Pow(10., precisionDelta)` here, but this would be costly, with lots of casting to and from floats or integers. Instead, we’ll delegate that task to a function named `pow10`. In the function, we’ll hardcode some common powers of 10 as quick-win solutions, and default to the expensive call to `math.Pow` only for values out of the expected range of exponents. Overall, this `pow10` function could be implemented with an exhaustive `map` or a `switch` statement. We decided to go with the latter, but both options are valid.

Listing 6.19 decimal.go: `pow10()`

```
// pow10 is a quick implementation of how to raise 10 to a given
// It's optimised for small powers, and slow for unusually high p
func pow10(power int) int {
    switch power {
    case 0:
        return 1
    case 1:
        return 10
    case 2:
        return 100
    case 3:
        return 1000
    default:
        return int(math.Pow(10, float64(power))) #A
    }
}
```

Let’s write the code for this first part, and then we can implement the mysterious `multiply` function. The `switch` is here to adjust the result with the precision of the target currency.

Listing 6.20 convert.go: `applyExchangeRate`

```
// applyExchangeRate returns a new Amount representing the input
// The precision of the returned value is that of the target Curr
// This function does not guarantee that the output amount is sup
func applyExchangeRate(a Amount, target Currency, rate ExchangeRa
    converted, err := multiply(a.quantity, rate) #A
    if err != nil {
        return Amount{}, err
    }

    switch { #B
    case converted.precision > target.precision: #C
        converted.subunits = converted.subunits / pow10(converted
    case converted.precision < target.precision: #D
        converted.subunits = converted.subunits * pow10(target.pr
    }

    converted.precision = target.precision

    return Amount{
        currency: target,
        quantity: converted,
    }, nil
}
```

The returned `Amount` is not constructed using the function that validates it. Instead, we prefer to return an amount that the `Convert` function has to validate before returning it to the external consumer. Note that we are being explicit in the documentation: if a future maintainer (you included) wants to start exposing this function for some reason, they will need to refactor it to return an error if needed.

Finally, the core of this chapter resides in the multiplication function, so let’s implement it! Remember, we don’t want to multiply floats together, as this could lead to floating-point errors. This means we’ll have to convert our `ExchangeRate` into a `Decimal`. The rest is quite straightforward.

Listing 6.21 convert.go: `multiply`

```
// multiply a Decimal with an ExchangeRate and returns the produc
func multiply(d Decimal, r ExchangeRate) (Decimal, error) {
    // first, convert the ExchangeRate to a Decimal
    rate, err := ParseDecimal(fmt.Sprintf("%g", r)) #A
    if err != nil {
        return Decimal{}, fmt.Errorf("%w: exchange rate is %f", E
    }

dec :=  Decimal{
        subunits:  d.subunits * rate.subunits,
        precision: d.precision + rate.precision,
    }
// Let's clean the representation a bit. Remove trailing zeroes.
dec.simplify()

return dec, nil
}
```

Now that we have a function to convert an amount to a new currency, where should we call it? Before we answer that question, let’s test what we’ve written.

Most importantly, testing it!

This is the heart of the logic. It requires a lot of testing to make sure that everything works fine and keeps working fine if we ever decide to change any implementation.

Before writing the test, you can start thinking about all the test cases imaginable. Here are a few examples:

Table 6.3 Possible test cases

Amount	Rate	TargetCurrency precision	What are we checking?
1.52	1	2	The output must be exactly identical to the input.
2.50	4	2	The decimal part becomes 0, but precision should be that of target
4	2.5	0	Same as above, but switched around, and the precision is 0.
3.14	2.52678	2	A real-life exchange rate
1.1	10	1	Keeping the precision of 1
1_000_000_000.01	2	2	Keep the precision in large numbers
265_413.87	5.05935e-5	2	Very small rate
265_413	1	3	Adding extra precision in the output when there was

			none
2	1.337	5	Increasing the precision in the output
2	1.33 * 10^16	5	Rate is too high

The number of different test cases, and how fast we can think of new corner cases, calls for a table or map-based test.

Of course, as the function is not exposed, the test will have to be internal and you need a new file for that. The implementation of the test can look like this.

Listing 6.22 convert_internal_test.go: Testing `applyExchangeRate`

```
package money

import (
    "reflect"
    "testing"
)

func TestApplyExchangeRate(t *testing.T) {
    tt := map[string]struct {
        in             Amount
        rate           ExchangeRate
        targetCurrency Currency
        expected       Amount
    }{
        "Amount(1.52) * rate(1)": { #A
            in: Amount{
                quantity: Decimal{
                    cents: 152,
                    exp:   2,
                },
                currency: Currency{code: "TST", precision: 2},
        },
        // add test cases
    }

    for name, tc := range tt {
        t.Run(name, func(t *testing.T) {
            got := applyExchangeRate(tc.in, tc.targetCurrency, tc
            if !reflect.DeepEqual(got.number, tc.expected) {
                t.Errorf("expected %v, got %v", tc.expected, got)
            }
        })
    }

}
```

Finally! `Convert` can return something useful to the consumer. We don't have exchange rates right now, so let's hardcode a rate of 2 for now and keep the fetching of exchange rates for later in this chapter.

Listing 6.23 convert.go: First implementation of `Convert`

```
// Convert applies the change rate to convert an amount to a targ
func Convert(amount Amount, to Currency) (Amount, error) {
    // Convert to the target currency applying the fetched change
    convertedValue := applyExchangeRate(amount, to, 2) #A

    return convertedValue, nil #B
}
```

Congratulations, you broke the test for this function! We are now returning something so you can fix it by calling `mustParseAmount` to define the expected output.

We now trust the heart of the conversion, we can make sure that what we return to the consumer can be used again by our own library.

6.3.2 Validate result

Because we have so many limitations in the supported values, it seems wise to check that the output is something expected.

What are the limitations exactly?

- The number cannot be higher than 10^{12} or it will lose precision because of the floats;
- The number's precision must be compatible with the currency's.

Contrary to the conversion logic, this can be the responsibility of the `Amount` structure. An amount can be valid or not, and it should know about it. After all, pound notes and the dollar bills have the necessary fancy decorations on them to attest their authenticity.

Listing 6.24 amount.go: `validate` method implementation

```
// validate returns an error if and only if an Amount is unsafe t
func (a Amount) validate() error {
    switch {
    case a.number.integerPart > maxAmount: #A
        return ErrTooLarge #A
    case a.number.precision > a.currency.precision:
        return ErrTooPrecise #A
    }

    return nil
}
```

This is what the `Convert` function finally looks like. It is pretty small: not much would need to be tested internally.

Listing 6.25 convert.go: `Convert` implementation

```
// Convert applies the change rate to convert an amount to a targ
func Convert(amount Amount, to Currency) (Amount, error) {
    // Convert to the target currency applying the fetched change
    convertedValue := applyExchangeRate(amount, to, 2)

    // Validate the converted amount is in the handled bounded ra
    if err := convertedValue.validate(); err != nil {
        return Amount{}, err
    }

    return convertedValue, nil
}
```

Check your tests: do you have a convincing coverage of the finalised library? You can check the coverage of your test.

Now that we have the whole structure and logic of our library, now that it is tested, let’s plug it in, because how is code fun if you don’t run it? After that, we will fetch some real-life change rates and finish the tool. Having an executable in which we keep adding features allows us to showcase an early version of our product that we can improve.

6.4 Command-Line Interface

In this section, we will write the `main` function: let’s not forget that we are not writing a library, we are writing a CLI. This means we will be parsing input parameters, validating them and passing them on to the `Convert` function.

But before we implement all these safety nets, we want to run our application!

6.4.1 Flags and arguments

Take a step back. What should our program do? Let’s look at the Usage we wrote in the requirements.

```
change -from EUR -to USD 413.98
```

Note that we have 2 flags - the in and out currencies - and an argument - the amount we want to change.

Currency flags

In order to read flags from a command line, as we saw in Chapter 2, Go has the quite explicit `flag` package. After importing it in our main.go file, we can start with the first few lines that will ensure we properly read from the command line.

As mentioned in Chapter 2, the `flag` package exposes useful methods to retrieve values from flag parameters. Here, we will use the `flag.String` method to retrieve the two currencies' source and target: `from` and `to`.

`flag.String` takes as an argument the name of the flag, the default value (which can be empty) and a brief description. It returns the contents of the flag `-from` as a variable of type `*string`. As we've already mentioned in Chapter 2, calling the `Parse` method is necessary after all flags are defined and before values are accessed. Here, we leave the default value empty for the `-from` flag, but we set it for the `-to` flag to the string `EUR`. Should the user not provide the `-from` flag on the command-line, the value of the `from` variable will be an empty string. Similarly, if the `-to` flag is absent, the value will be `EUR`.

Listing 6.26 main.go: Parsing the flags

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    from := flag.String("from", "", "source currency, required")
    to := flag.String("to", "EUR", "target currency") #A

    flag.Parse()

    fmt.Println(*from, *to) #B
}
```

Now, we can run it. Do you remember how to run a program from the terminal after all this library development?

```
go run . -from EUR -to CHF 10.50
```

With this first implementation of the main, we're not calling the `Convert` function, but we're printing the source and destination currencies. They should appear on the screen.

Value argument

The next step in implementing our command-line interface is to retrieve the value that we have to convert. If it's absent from the command line, we'll exit with an error.

Retrieving arguments from the command-line

When running an executable, most of the time, we need to specify the input, the behaviour, the output, etc. These parameters can be provided either explicitly, via the command line, or implicitly, via pre-set environment variables, or configuration files at known locations. When it comes to explicit settings, there are two ways of passing user-defined values to the program: arguments, and flags.

Arguments are a sequence of parameters that starts at 0 and proceeds. Arguments are anonymous and ordered. Exchanging their position can completely change the behaviour of the program. Arguments, most of the time, are mandatory, and have no default value. The only argument of the `go build` command is the directory, file, or package containing the `main` function.

Flag parameters, on the other hand, aren't sorted. They can appear in any order in the command-line without altering the behaviour of the program. They can have default values (used when the flag is absent from the command-line), as our `-to` has. An example of a flag that controls behaviour that you might have been using is the `-o {binaryPath}` option of the `go build`.

In Go, the parameters of the command line can be retrieved with `os.Args` or with the `flag` package. Let's have an example to see the differences between these two:

```
./convert -from EUR -to JPY 15.23
```

In this line, the `os.Args` would return a list of 6 strings, each entry representing a word of the command: `{"./convert", "-from", "EUR", "-to", "JPY", "15.23"}`. The `flag.Args`, on the other hand, would return only the arguments to the command line that weren't in flags: `{"15.23"}`.

Depending on which information we want to access, using `flag.Args` or `os.Args` is more meaningful. In our case, we only want to access the command line parameters, and we don't care which flags were set. Using the first parameter that isn't part of a flag is simple: we can use `flag.Arg(0)`.

Listing 6.27 main.go: Retrieve the first argument of the program

```
func main() {
    from := flag.String("from", "", "source currency, required")
    to := flag.String("to", "EUR", "target currency")

    flag.Parse()

    value := flag.Arg(0) #A
    if value == "" {
        _, _ = fmt.Fprintln(os.Stderr, "missing amount to convert
        flag.Usage() #C
        os.Exit(1)
    }

    fmt.Println(*from, *to, value) #D
}
```

The inputs are in. They are strings, and we are not sure that the values are valid. Fortunately, we have the perfect functions for that already.

6.4.2 Parse into business types

The `Convert` function is taking as parameters values that are already typed for its usage, and the package exposes ways to build them. This strategy optimises flexibility in the consumer’s logic, as `main` is free to use the type through any other logic that it could add, or use its own different types and `Parse` at the last minute, or use strings and parse whenever it needs it.

We are not doing much more than converting in this chapter (feel free to add to it later). We just need to parse them all.

Listing 6.28 main.go: Parse currencies and amount

```
package main

import (
    "flag"
    "fmt"
    "os"

    "learngo-pockets/moneyconverter/money"
)
```

```
func main() {
    from := flag.String("from", "", "source currency, required")
    to := flag.String("to", "EUR", "target currency")

    flag.Parse()

    fromCurrency, err := money.ParseCurrency(*from) #A
    if err != nil {
        _, _ = fmt.Fprintf(os.Stderr, "unable to parse source cur
        os.Exit(1)
    }

    // TODO: repeat for target currency

    // TODO: read the argument, as seen above

    quantity, err := money.ParseDecimal(value) #B
    if err != nil {
        _, _ = fmt.Fprintf(os.Stderr, "unable to parse value %q: 
        os.Exit(1)
    }

    amount, err := money.NewAmount(quantity, fromCurrency) #C
    if err != nil {
        _, _ = fmt.Fprintf(os.Stderr, err.Error())
        os.Exit(1)
    }

    fmt.Println("Amount:", amount, "; Currency:", toCurrency) #D
}
```

Run it and enjoy the show. You should have some gibberish, something like this:

```
$ go run . -from EUR -to CHF 10.50
Amount: {{10 50 2} {EUR 2}}; Currency: {CHF 2}
```

For someone who doesn’t know the structures we use, this is hard to understand. It is therefore polite for the library to expose some Stringers on its types.

6.4.3 Stringer

If you look into the `fmt` package, you can find a very useful interface that all of the package’s formatting and printing functions understand: the `Stringer`. It follows a Go pattern where interfaces with only one method are named with this method followed by -er, as in `Reader`, `Writer`, etc. Let’s look at how it is defined:

```
type Stringer interface {
    String() string
}
```

`fmt.Stringer` is implemented by any type that has a String method, which defines the “native” formatting for that value. The String method is used to print values passed as an operand to any format that accepts a string or to an unformatted printing function such as Print.

In order to implement an interface in Go, a type only needs to have the right method(s) attached to it.

Listing 6.29 currency.go: Implement Currency `Stringer`

```
// String implements Stringer.
func (c Currency) String() string {
    return c.code
}
```

Magically, your `Currency` is now a `Stringer`. Anyone calling a printing function with a currency as parameter will be calling this. Try it:

```
$ go run . -from EUR -to CHF 10.50
Amount: {{10 50 2} {EUR 2}}; Currency: CHF
```

The target currency is now properly readable. Let’s do the same with `Decimal`. We already have a method on the type Decimal, and it receives a pointer - the `simplify` method. Go doesn’t really like having both pointer and non-pointer receivers for methods of a type, so let’s have `String()` accept a pointer receiver - we need a pointer receiver for `simplify`.

Here we chose to use a double-formatting trick: we are using the precision to create the formatting string, then using this to format the number. For example, for a precision of 2 digits, the format variable will be `%d.%02d`, which pads the decimal part with zeros: we don’t want to print 12.5 when the currency has cents, we want to print 12.50. This trailing 0 is added by padding it in the `%02` formatting string.

However, not all currencies have a precision of 2 digits, and we must build this `%02` string using the precision of the currency. For this, we can use a function provided by the package in charge of string conversions - adequately named `strconv`. The function we use is `strconv.Itoa`, which you can think of as the reverse `strconv.Atoi`.

```
decimalFormat := "%d.%0" + strconv.Itoa(int(d.precision)) + "d"
```

Immediately, we notice that an edge-case is when the precision is 0. Since this is a simple test, and that the output is quite simple, we will start our `String()` function by checking this scenario.

We have all the bricks to write the implementation of the `Stringer` interface for `Decimal` type. The output of `pow10` gives us the number of subunits in a unit of the currency, which means we can retrieve the fractional part and the integer part by simply dividing by the number of subunits. Finally, we can return the printed output using the formatting `decimalFormat`.

Listing 6.30 decimal.go: Implement Decimal Stringer

```
// String implements stringer and returns the Decimal formatted a
// digits and optionally a decimal point followed by digits.
func (d *Decimal) String() string {
    // Quick-win, no need to do maths.
    if d.precision == 0 {
        return fmt.Sprintf("%d", d.subunits)   #A
    }

    centsPerUnit := pow10(d.precision)    #B
    frac := d.subunits % centsPerUnit
    integer := d.subunits / centsPerUnit

    // We always want to print the correct number of digits - eve
    decimalFormat := "%d.%0" + strconv.Itoa(int(d.precision)) + "
    return fmt.Sprintf(decimalFormat, integer, frac)

}
```

Even if `Currency`’s `String` method can arguably skip the unit test requirement, this one needs one. Take a minute to write it and check your coverage. Remember that coverage does not check that you are covered, you could have perfect coverage and still miss a lot of cases; instead, it tells you where you are not, and you can decide whether it is worth the effort to extend coverage.

Finally, `Amount` should also implement the `Stringer` interface. This could be adapted to different language standards but we chose `22.368 KWD` as the output format.

Listing 6.31 amount.go: Implement Amount Stringer

```
// String implements stringer.
func (a Amount) String() string {
    return a.number.String() + " " + a.currency.code
}
```

Is your output more legible now?

```
$ go run . -from EUR -to CHF 10.50
10.50 EUR CHF
```

Now we have all the Stringer implemented, we can call the Convert function.

6.4.4 Convert

The only thing left to do in the main function is to call `Convert`.

Listing 6.32 main.go: End of the main function

```
func main() {
    // ...

    convertedAmount, err := money.Convert(amount, toCurrency)
    if err != nil {
        _, _ = fmt.Fprintf(os.Stderr, "unable to convert %s to %s
        os.Exit(1)
    }

    fmt.Printf("%s = %s\n", amount, convertedAmount))
}
```

This code compiles. This code can be run. It’s beautiful.

Did you spot the bug?

There is a final issue we need to address here. Despite our heavy testing, we’ve missed something quite obvious. If you’ve tried running the tool, you might’ve noticed it. When we pass the input amount with a lower number of decimal digits than the currency’s precision, we display that amount with its input number of digits, and not its currency’s!

Here’s an example:

```
$go run . -from BHD -to CHF 12.5
12.5 BHD = 25.00 CHF
```

If we check the switch in the `ParseCurrency` code, we see that there are 1000 fulūs in 1 Bahraini dinar - we should be writing `12.500 BHD = 25.00 CHF`.

The root for this problem resides in the `NewAmount` function. Let’s fix it by taking into account the currency’s precision, and add a test to cover this bug.

Listing 6.33 amount.go: Fixing the NewAmount

```
// NewAmount returns an Amount of money.
func NewAmount(quantity Decimal, currency Currency) (Amount, erro
    switch {
    case quantity.precision > currency.precision:
        // In order to avoid converting 0.00001 cent, let's exit
        return Amount{}, ErrTooPrecise
    case quantity.precision < currency.precision:
        quantity.subunits *= pow10(currency.precision - quantity.
        quantity.precision = currency.precision
    }
    return Amount{quantity: quantity, currency: currency}, nil

}
```

There’s a teeny tiny problem, though. This code doesn’t work properly: it applies a constant conversion rate of 2, regardless of currencies we set on the command-line. We need real exchange rates. We are ready to call the bank.

6.5 Call the bank

We have a working solution, but for one problem: we are not using the real exchange rates. We need to call an external authority to get them. Here, the authors chose to implement a solution based on the API of the European Central Bank, because it is free of charge, it does not need any identification protocol, and it is very likely to still be running with the same API in a year or even two. An unreliable API from a data provider is something that we don’t want to face.

Fetching and using the data are two separable concerns and any separable logic should be indeed separated in order to make testing and evolving easier. The bank is going to be a dependency of our program: an external resource on which it relies in order to work. It is an accepted best practice in software design, whatever the language you use, to use inversion of control (IoC). Inversion of control serves multiple design purposes:

- decoupling the execution of a task from implementation,
- focusing a module or package on the task it is designed for,
- freeing systems from assumptions about how other systems do what they do and instead rely on contracts,
- and finally preventing side effects when replacing a module.

More concretely, the `money` package should not know where the exchange rate is coming from, this is beyond its scope. Another package will be responsible for calling the bank when needed, deal with the bank-specific logic and return the required info. This other package is therefore a dependency that the consumer (here our `main` function) is giving to it, via a contract in the shape of an interface. This way, the consumer decides what source of data is the best, and the money conversion is not touched.

Let’s take an example. Let’s say that while you are writing the tool, somebody else in another team is writing the banking service. You cannot access it yet. What you can do is create a dependency that `Convert` can understand, where you simply return hard-coded values. And when the service is finally here, you just need to replace the plug with a call to the new API. Everything else is already tested and runs smoothly. Replacing dependencies with stubs during development is just one use of this pattern. Another could be adding a cache: replace a call to the API with a similar function that checks in memory whether the value is already known and avoids a network call.

In our case, the dependency's role is to fetch the exchange rate between two currencies. Think of an errand boy cycling to the bank a few streets away and returning with the info, while the clerk responsible for computations is waiting. Even though the API returns the whole list of currencies that it knows and exchange rates for 1 euro, the tool doesn't need the full list, only the `to` and `from` currencies; knowledge about the details of the API should stay inside the dependency's package.

6.5.1 Dependency injection - the theory

There are two ways in Go to provide a dependency: one is more object-oriented, the other looks like functional programming.

Object dependency

The first option requires the consumer to have in hand a variable of a type that implements an interface. If you know any object-oriented language, such as the Java family, you will be familiar with this approach.

In this version, we create a structure with a function `FetchRates` attached to it, and we pass a variable of this type to `Convert`. `Convert` is expecting any variable that implements the expected interface.

Listing 6.34 Dependency injection via an interface

```
type ratesFetcher interface {
    FetchRates(from, to Currency) (ExchangeRate, error) #A
}

func Convert(..., rates ratesFetcher) { #B
    ...
    rate, err := rates.FetchRates(from, to) #B
    ...
}
```

```
...

func main() {
    ratesRepo := newRatesRepository() #C
    money.Convert(..., ratesRepo)
}
```

In this implementation, the main function is in charge of creating the variable that implements the interface.

Function dependency

The second option is more verbose but it also works and can be preferred in some cases. The `Convert` function's last parameter is a function's definition rather than an object implementing an interface. The rest is relatively similar.

Listing 6.35 Dependency injection via a function

```
func Convert(..., rates func(from, to Currency) (ExchangeRate, er
    ...
    rate, err := rates(from, to) #B
    ...
}

func main() {
    ratesRepo := newRatesRepository() #C
    money.Convert(..., ratesRepo.FetchRates) #D
}
```

The main function is passing directly the `FetchRates` method that `Convert` will be calling.

You can even name the function's signature by declaring a type.

```
type getExchangeRatesFunc func(from, to Currency) (ExchangeRate,

func Convert(..., rates getExchangeRatesFunc) (Amount, error) {
...
}
```

Alternatively, the consumer is free to create any function on the fly, relying on variables of the outside scope if needed:

Listing 6.36 Dependency injection via a local function

```
func main() {
    config := ...

    fetcher := func(from, to Currency) (ExchangeRate, error){ #A
        return config.MockRate, nil #B
    }

    money.Convert(..., fetcher)
}
```

As you can see the function dependency option is a bit less intuitive for beginners - why would a function be a parameter? - but also leaves more room to the consumer to implement the dependency. It doesn’t fix the name of the function, nor does it require it to be a method on an object, and mocking it for tests is slightly easier. As often, it leaves more freedom for the implementation, which means that mistakes are easier to make.

For example, you can have two different methods on one object and pick the one you want to use depending on the context. Imagine an API where you can have daily exchange rates for free, or rates updated every minute when you are logged in. Both functions have the same signature with different names, and they are methods of the same object, which contains configurations valid for both calls.

Listing 6.37 Dependency injection via a local function

```
func main() {
    ratesRepo := newRatesRepository()
     apiKey := getAPIKey() #A

    fetcher := ratesRepo.FreeRates #B
    if apiKey != "" {
        fetcher = ratesRepo.WithAPIKey(apiKey).LoggedInRates #C
    }

    money.Convert(..., fetcher)
}
```

In the rest of the chapter, we choose to implement the rate retriever with the first approach presented, the interface dependency, mostly because it is easier to read, explain, and understand.

6.5.2 ECB package

Let’s create a new package that will be responsible for the call to the bank’s API. There is no point in trying to make it sound generic: it will only know how to call this one API from the European Central Bank, so let’s call it `ecbank`.

API of the package

As we’ve seen, the new package should expose a struct with one method attached, and probably a way to build it.

Let’s talk a little about the method’s signature. We assume that it will take two currencies and return the rate or an error. It should not return a `Decimal`, because `Decimal` represents money values and has the associated constraint that nothing exists below the cent (or agora or qəpik). It should return an exchange rate, for which we happen to already have a business type.

What are we going to call the structure? In real physical life, in order to get the information, your errand boy would walk to the bank and ask. In our code, this object does what the bank does, so we can safely call it a bank.

Listing 6.38 ecb.go: EuroCentralBank struct and its principal method

```
// EuroCentralBank can call the bank to retrieve exchange rates.
type EuroCentralBank struct {
}

// FetchExchangeRate fetches the ExchangeRate for the day and ret
func (ecb EuroCentralBank) FetchExchangeRate(source, target money
    return 0, nil
}
```

Arguably, we could build completely independent packages and not rely on `money`’s types. The architectural decision instead is to base everything on the autonomous `money` package. Others are allowed to rely on it, but it needs to rely on nothing else. In Go, if package A relies on package B and B on A, the compiler will stop you right there: import cycles, also known as cyclic dependencies, are not allowed. Cyclic dependencies are a compiler's version of our chicken or egg dilemma. Even though some languages might manage how to build with cyclic dependencies, Go forbids it from the start. This makes the architecture cleaner, in our opinion.

What will this `FetchExchangeRate` method do? Build the request for the API, call it, check whether it worked and, if it did, read the response and return the exchange rate between the given currencies. The whole logic is articulated around the HTTP call. Let's see how Go natively deals with these.

6.5.3 HTTP call, easy version

The European Central Bank exposes an endpoint that lists daily exchange rates. You can first try calling the API in your favourite terminal to see what it looks like.

```
curl "http://www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xm
```

As you can see it returns a large XML response. We will have to parse it and find the desired value. But first, let's retrieve this message in our code!

Go's `net/http` package provides server and client utilities for HTTP(S) calls. It uses a struct called `Client` to manage the internals of communicating over HTTP and HTTPS. Clients are thread-safe objects that contain configuration, manage TCP state, handle cookies, etc. Some of the package's functions don't require a client, for example, the simple `Get` function, and use a default one:

```
http.Get("http://example.com/")
```

We will start with this easy call.

Making calls to external resources

Anything that calls something that isn't your code should be handled with extreme care and precaution. Hope for the best, but prepare for the worst. Here is your chance to be creative: what’s the worst that could possibly happen when calling someone else’s code? Making a call to a library could, potentially, lead to a panic, in the worst case scenario.

When it comes to network calls, we could get errors from the network (for instance, we couldn’t resolve the URL of the resource), or server-related problems (timeouts, unavailability).

It is your responsibility, as developer, to decide which issues should be handled by your code, and which won’t be. Be explicit in your documentation about what is covered, and what isn’t. For instance, if you decide to not set a timeout for your request, you implicitly accept that the call you make could hang on to your connection forever - resulting in your application being frozen.

Design is about what you allow and what you prevent.

Where to declare the path

We declared the path to the resource as a constant. This is OK for now - maintainers can change it in one single place if the API changes (imagine the case of a version change in the path, even if it is not the case here). Ideally, the client package should know about the relative path and the consumer should tell it, as a configuration, what URL to call: this leaves space for test environments. This configuration is out of the scope of our chapter but feel free to think up a cleaner solution.

You can declare the constant just before the function, but you can also reduce its visibility and prevent anything else inside the package from reaching it by declaring it inside the `FetchExchangeRates` function. The compiler will still replace it wherever it finds it.

Listing 6.39 ecb.go: FetchExchangeRates first lines

```
const euroxrefURL := "http://www.ecb.europa.eu/stats/eurofxref/eu

resp, err := http.Get(euroxrefURL)
```

```
if err != nil {
    return money.ExchangeRate(0), fmt.Errorf("%w: %s", ErrServerS
}
```

In case of an error, we return the zero value of the first return value’s type. This type is `ExchangeRate`, which is based on a `float64`, so its zero value is simply 0. We add the dot afterwards to specify to the compiler that we are declaring a float and not an integer. The compiler will know that it should actually return an `ExchangeRate`.

Note that in production code, it is considered a bad idea to stick to the default client which `http.Get` is using. See more about clients at the end of this chapter.

Errors, again

Keeping in mind that the consumer - the `main` function - should not have to deal with implementation details, we should not directly propagate the `net/http` package’s error to our consumer: if they want to check what type of error is returned, they would also have to rely on the `net/http` package - then, if we change the implementation and use another protocol, we break the consumer’s code. Not nice.

We will instead declare the same 4 lines as the money package’s errors, but these will be specific to our `ecbank` package: consumers will be able to check the value of the error with errors.Is() and know its meaning.

Here, we don’t really need to expose the error type we define - it brings no value to the customers. We only need to expose the sentinel errors, as we want to allow for error checking.

Listing 6.40 errors.go: Possible errors

```
// ecbankError defines a sentinel error.
type ecbankError string

// ecbankError implements the error interface.
func (e ecbankError) Error() string {
    return string(e)
}
```

We then declare our constant and exposed errors close to where they can be returned. This list will be enriched as we add more code.

```
const (
    ErrCallingServer = ecbankError("error calling server")
)
```

The `http.Get` function returns an `http.Response`. One of `Reponse`’s exposed fields is its `Body`, which implements `io.ReadCloser`. Before even looking at the documentation, you can see from its name that it exposes a `Read` and a `Close` method. Whatever you do, don’t forget to close it, or you will create all kinds of memory leaks. To clear your mind from that as soon as possible, Go has the `defer` keyword: what you put after `defer` will be done just before any return. It means you can use the response, read it, have fun, return errors when you have to, and whichever branch the code runs through will close the response body before returning.

```
defer resp.Body.Close()
```

Now we can look at what we received from the bank.

6.5.4 Parse the response

Before parsing the body of the response, we first want to check the status code. The status code describes how the remote server handled our query. There is no point in reading the response if we know that the call was unsuccessful.

Check status

The standard of the hypertext transfer protocol defines a long list of possible status codes distributed in five classes:

- 1xx (100 to 199) informational response – the request was received, continuing process
- 2xx (200 to 299) successful – the request was successfully received, understood, and accepted
- 3xx (300 to 399) redirection – further action needs to be taken in order

to complete the request
- 4xx (400 to 499) client error – the request contains bad syntax or cannot be fulfilled
- 5xx (500 to 599) server error – the server failed to fulfil an apparently valid request

In order to carry on, we need something that starts with 2 - more specifically, we know that we want a 200. But we also want to check for 4xx and 5xx: in the first case we made a mistake with our query, in the second it's not our fault.

Because we currently only really care about the class of the status code, in case of a problem, we can use a division to check just the first figure. It's perfectly fine to have a function dedicated for only this division.

Listing 6.41 ecb.go: Function handling HTTP status code

```
const (
    clientErrorClass = 4
    serverErrorClass = 5
)

// checkStatusCode returns a different error depending on the ret
func checkStatusCode(statusCode int) error {
    switch {
    case statusCode == http.StatusOK: #A
        return nil
    case httpStatusClass(statusCode) == clientErrorClass: #B
        return fmt.Errorf("%w: %d", ErrClientSide, statusCode)
    case httpStatusClass(statusCode) == serverErrorClass: #C
        return fmt.Errorf("%w: %d", ErrServerSide, statusCode)
    default: #D
        return fmt.Errorf("%w: %d", ErrUnknownStatusCode, statusC
    }
}

// httpStatusClass returns the class of a http status code.
func httpStatusClass(statusCode int) int {
    const httpErrorClassSize = 100
    return statusCode / httpErrorClassSize
}
```

The `FetchExchangeRate` function can call this checker and forward the error without wrapping it: we already made sure we knew what type of error we were returning. When calling functions from the same package, it is your responsibility to decide whether you want to wrap the error or not. Errors coming out of exposed functions should all be documented and of known types, but you have the choice of where you create them.

We now know that the http call caused no error, and we can ensure that the server returned a valid response. Let’s now have a look at the XML contained in this response.

XML parsing

In order to parse XML, we will use the `encoding/xml` package of Go. As we saw in Chapter 3, there is a good list of different encodings supported next to it, including JSON, CSV, base64…

Decoding and encoding XML (or JSON)

Both encoding/json and encoding/xml packages offer two ways of decoding a message. They both expose an `Unmarshal` function that can convert a slice of bytes into an object. They also both allow for the creation of a `Decoder`, through a function called `NewDecoder`. This constructor takes a `io.Reader` as its parameter, from which calls to `Decoder` will read and convert data to the desired object. The answer to the “which should I use?” question is simple: if you have an `io.Reader`, use a `Decoder`. If you have a `[]byte`, then using `Unmarshal` or `NewDecoder` is fine.

Similarly, when encoding JSON or XML, if you have access to a `io.Writer`, use an `Encoder`. Otherwise, use `Marshal`.

Here are two examples of how to parse a slice of bytes :

```
type person struct {
    Age  int    `json:"age"`  # Must be exposed to be decoded
    Name string `json:"name"`
}

data := []byte(`{"age": 23, "name": "Yoko"}`)
```

```
p := person{}

err := json.Unmarshal(data, &p) # This requires the whole data sl

if err != nil {

    panic(err)

}
```

```
p := person{}

dec := json.NewDecoder(bytes.NewReader(data)) # Or use an io.Rea

err := dec.Decode(&p)

if err != nil {

    panic(err)

}
```

The `response.Body` is of type `io.Reader`, isn't that convenient! It therefore makes complete sense to use a `Decoder` here. We can then `Decode` into the right structure. In order to do this, we first define a type (which will be covered in the next paragraph), and pass a pointer to a variable of that type to the decoder. Indeed, declaring the variable will allow it to exist in memory, the decoder will access its various fields to fill them with what can be found in the `Reader`. It is paramount to provide a pointer to the `Decoder` - see Appendix E for a more detailed explanation.

```
decoder := xml.NewDecoder(resp.Body)

var xrefMessage theRightStructure #A
err := decoder.Decode(&xrefMessage)
```

What exactly is this "right" structure? Something that looks like the response we got, and states what XML field should be unmarshalled into what Go field, using tags.

To define this structure, let’s start by looking at the response. The European Central Bank being responsible for euros, everything is based on euros.

Listing 6.42 XML response from the API

```
<?xml version="1.0" encoding="UTF-8"?>
<gesmes:Envelope xmlns:gesmes="http://www.gesmes.org/xml/2002-08-
    <gesmes:subject>Reference rates</gesmes:subject>
    <gesmes:Sender>
        <gesmes:name>European Central Bank</gesmes:name>
    </gesmes:Sender>
    <Cube>
        <Cube time='2023-02-20'> #B
            <Cube currency='USD' rate='1.0674'/> #C
            <Cube currency='JPY' rate='143.09'/>
            <Cube currency='BGN' rate='1.9558'/>
            <Cube currency='CZK' rate='23.693'/>
            <Cube currency='DKK' rate='7.4461'/>
            [...]
        </Cube>
    </Cube>
</gesmes:Envelope>
```

We can keep the naming of the response and create a structure called `envelope`. However, the XML node name Cube is not explicit enough, so we’ll use `currencyRates`.

While the parsing objects themselves, `envelope` and `currencyRate`, do not need to be exposed, their fields must be accessible to the `encoding/xml` package, and therefore have to be exposed.

The way Go tells the `encoding/*` packages from and to which node a field should be decoded or encoded is by defining a tag at the end of the line declaring this field’s name in the structured language. Tags are always declared between back-quotes and composed of their name followed by a column and a value in double quotes. If you need multiple tags on the same field, separate them with commas inside the backquotes. For example:

```
type Movie struct {
    Title       string `xml:"Title",json:"title"`#A
    ReleaseYear int    `json:"year"` #B
}
```

To retrieve the attributes of an XML node you just need to tell the decoder to look for an attribute. Go offers the possibility to "unnest" nodes by using the > syntax. Here, we don't want to retrieve the `time` attribute of the intermediate `Cube` node, only its inner `Cube` nodes. We "skip" from the root to the level that contains the data we want with `Cube>Cube>Cube`, where the first one is a child of the `Envelope`, and the last one contains our exchange rate.

Listing 6.43 envelope.go: structures used for XML decoding

```
type envelope struct {
    Rates []currencyRate `xml:"Cube>Cube>Cube"` #A
}

type currencyRate struct {
    Currency string             `xml:"currency,attr"` #B
    Rate     money.ExchangeRate `xml:"rate,attr"`
}
```

There are a couple of fields that we don't need, such as the `time` or the `subject`. Be minimal when you are declaring tags and only retrieve what you need.

Compute exchange rate

Now, we need to compute the exchange rate between the source and target currencies. Remember that the European Central Bank's exchange rates are all answers to "which quantity of currency X do I get for 1 euro?". This means that the euro can be used as a "transition" currency, or, even better, that the rates to convert from a currency to another is simply computed with a hop in euro-world. The rate from CAD to ZAR is, by transitivity, the rate from CAD to EUR multiplied by the rate from EUR to ZAR. We only have access to the EUR to CAD exchange rate, but we'll assume, in this project, that the CAD to EUR exchange rate is the inverse of the EUR to CAD exchange rate.

How do we retrieve our two change rates from the decoded list? One approach would be to go through the list and retrieve them. We would need to stop as soon as we found both of them. If, when we reach the end of the list, we didn't find them, then we can send an error. This implementation would work, but it doesn't make the easiest code to read.

Considering the very low-performance requirements in our situation, we prefer to optimise for maintainability rather than memory footprint and go with another solution: store all the decoded currencies and their change rates in a `map`, and add the euro - as it's absent in the payload from the European Central Bank. Then, when need be, we can get the interesting values from the map. Registering the values has an O(N) time and memory complexity, because we go through the list once; and getting a value from a map is, in our case, an O(1) in time complexity. It adds a little memory footprint than only storing two values, but there are not millions of currencies in the world, even if you choose to consider all of human history. We should be fine.

The map key is the currency and the value is the rate. Note that we could improve readability by naming the currency code something else than `string`, but as the money package did not deem it necessary to expose the type, let's follow suit and roll with a simple `string`.

We could write a function that takes an envelope as its input and returns the map, or we could write a method. Both implementations would be clear here. Using a method implies that we may be changing the object that holds it, whereas using a function should not. It is more a convention than a real constraint.

Listing 6.44 envelope.go: Map the rates for easy search

```
const baseCurrencyCode = "EUR"

// exchangeRates builds a map of all the supported exchange rates
func (e envelope) exchangeRates() map[string]money.ExchangeRate {
    rates := make(map[string]money.ExchangeRate, len(e.Rates)+1)

    for _, c := range e.Rates {
        rates[c.Currency] = c.Rate
    }

    rates[baseCurrencyCode] = 1. #B
    return rates
}
```

From there it becomes easy to compute the desired change rate:

Listing 6.45 envelope.go: Compute change rate

```
// exchangeRate reads the change rate from the Envelope's content
func (e envelope) exchangeRate(source, target string) (money.Exch
    if source == target {
        return 1., nil #A
    }

    rates := e.mappedChangeRates()

    sourceFactor, sourceFound := rates[source] #B
    if !sourceFound {
        return 0, fmt.Errorf("failed to find the source currency 
    }

    targetFactor, targetFound := rates[target] #D
    if !targetFound {
        return 0, fmt.Errorf("failed to find target currency %s",
    }

    return targetFactor / sourceFactor, nil #E
}
```

We are using the shortened syntax for currencies in input, where multiple parameters have the same type: this type is only declared once at the end of the list. Compare:

```
source, target string
```

```
source string, target string
```

Don't forget to test! You don't need us for this one. While sometimes it is ok to skip a unit test on some intermediate layers, this kind of computation should raise a test flag and sirens: coming back to change an implementation detail may result in this division being switched around and the next thing you know the FBI is after you for illegal money. But switching a division around, who would do that, it never happens! You would be surprised.

Once you are done, let’s write the function that reads from the response body and returns the exchange rate.

Listing 6.46 envelope.go: Read change rate

```
func readRateFromResponse(source, target string, respBody io.Read
    // read the response
    decoder := xml.NewDecoder(respBody)

    var ecbMessage envelope
    err := decoder.Decode(&ecbMessage)
    if err != nil {
        return 0., fmt.Errorf("%w: %s", ErrUnexpectedFormat, err)
    }

    rate, err := ecbMessage.exchangeRate(source, target)
    if err != nil {
        return 0., fmt.Errorf("%w: %s", ErrChangeRateNotFound, er
    }
    return rate, nil
}
```

As you can see, we are limiting the scope of the arguments to `string`s and `io.Reader`. It could be tempting to send the full `money.Currency` and `*http.Response` that the main function actually has in hand, but it makes testing harder and blocks future changes for no good reason.

The last thing we need to do for the exposed method of the package is, call this last function. Easy, right?

```
readRateFromResponse(source.ISOCode(), target.ISOCode(), resp.Bod
```

The ISO codes of `source` and `target` are not exposed, though. They are accessible via the `String()` method, so it would be tempting to use that, but what is the guarantee that the stringer will always return the ISO code? What if somebody wants to make the CLI nicer and print the full name in English, they would just have to change the stringer. Boom, nothing works anymore.

We can instead add an `ISOCode` method in the `money` package, one that provides the ISO code and whose behaviour is not going to change for the sake of the presentation.

The final exposed function should look something like this.

Listing 6.47 ecb.go: Fetch exchange rate

```
// FetchExchangeRate fetches the ExchangeRate for the day and ret
func (ecb EuroCentralBank) FetchExchangeRate(source, target money
    const path = "http://www.ecb.europa.eu/stats/eurofxref/eurofx

    resp, err := http.Get(path)
    if err != nil {
        return 0., fmt.Errorf("%w: %s", ErrServerSide, err.Error(
    }

    // don't forget to close the response's body
    defer resp.Body.Close()

    if err = checkStatusCode(resp.StatusCode); err != nil {
        return 0., err
    }

    rate, err := readRateFromResponse(source.Code(), target.Code(
    if err != nil {
        return 0., err
    }

    return rate, nil
}
```

Testing around HTTP calls

Now, this part is pretty important for our tool, so how do we test it? The code is explicitly calling a hard-coded URL. Do we really want to make an HTTP call every time we run a unit test? What if the remote server is not responding, what if we lost the connection? Unit tests should be local and fast. We certainly don’t want to have a real call during unit testing.

The `httptest` package exposes the infrastructure to set up a small HTTP server for the tests. It can run a very tiny mock HTTP server for the milliseconds when your test requires it. Then you just need to pass the server’s URL to the caller and define what you expect as a response. Any query to that server’s URL will always return a specific message, as we’ll see below.

But, wait, in our code, the URL is a constant. As mentioned before, in a production environment, we would want to make this configurable and make it part of the `Client` object. Let’s do that.

Add a field path to the object. Ideally, a `New` function would be tasked with taking this path as a parameter and creating the object, but we can take a small shortcut.

Listing 6.48 ecb.go: Fetch exchange rate with mockable path

```
// Client can call the bank to retrieve exchange rates.
type Client struct {
    url string #A
}

// FetchExchangeRate fetches today's ExchangeRate and returns it.
func (c Client) FetchExchangeRate(source, target money.Currency)
    const euroxrefURL = "http://www.ecb.europa.eu/stats/eurofxref

    if c.url == "" { #B
        c.url = euroxrefURL
    }

    resp, err := http.Get(c.url)
// ...
```

This should work the exact same way if you test it manually. The only difference is that now you can automate the test.

In your test, start by creating a server using the `httptest` facilities. `NewServer` takes a function of the type `HandlerFunc`, which is the standard HTTP handler in Go, defined not in the httptest but in the real http package.

```
ts := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWri
    fmt.Fprintln(w, `...`)
}))
defer ts.Close() #A
```

Here the parameter is an anonymous function that we cast into the `HandlerFunc` type. What the server does is just write the expected response into the `ResponseWriter` every time a query is sent to its URL.

We can then pass this server's URL to our EuroCentralBank and the rest of the test is quite easy for you by now:

Listing 6.49 ecb_internal_test.go: Testing the Fetch function

```
func TestEuroCentralBank_FetchExchangeRate_Success(t *testing.T)
    ts := httptest.NewServer(http.HandlerFunc(func(w http.Respons
        fmt.Fprintln(w, `<?xml...>`)
    }))
    defer ts.Close() #A

    ecb := Client{
        path: ts.URL, #B
    }

    got, err := c.FetchExchangeRate(mustParseCurrency(t, "USD"),

//...
```

Note: why are we copying `mustParseCurrency` in 2 places? Many times, a small copy is better than a big dependency. This way, both can evolve independently. To keep only one, you would need to expose it in a non-test file… let's stop there and copy a handful of lines.

Of course the example test that we are giving here is just one test case. Don't forget to add more cases, including error cases. Instead of writing a pretty XML response into the ResponseWriter, try this line:

```
w.WriteHeader(http.StatusInternalServerError)
```

What would you expect if the XML is broken?

6.5.5 Use in the money package

Good. Now, we can retrieve the rate. Back in `main`, how do we use the previous section with `Convert`?

Interface definition

In some common languages, an interface is a contract that every implementation must satisfy. In Go, it's quite the opposite. We don't write interfaces as long as we don't need them. And we start needing them when they make our life simpler. We have a saying in Go that could be counterintuitive if you come from a strongly object-oriented language like Java or C++ where interfaces are implicit.

Interfaces

should be discovered, not designed.

What does it mean? We have written a dependency for `Convert` without first defining the contract between the two packages: `money.Convert` has to use something from the `ecbank` package. Is it not too late now - it never is - how are we going to tell `Convert` what rates provider to expect?

Well, we just do. We already have a function signature that is generic enough to be mocked in tests. Let's put it in an interface for `Convert` to use. As usual, put it where you need it: you can declare it next to `Convert` itself.

Listing 6.50 exchangerates.go: Interface definition

```
type exchangeRates interface { #A
    FetchExchangeRate(source, target Currency) (ExchangeRate, err
}
```

Why is it not exposed, how can other packages use the interface? Actually, we don't want anyone else to rely on this interface, it's ours, in this package. If someone else needs to call the same API, they will define their own one-line interface and mock it the way they want. It reduces coupling quite drastically.

Let's see how we can use the interface.

Use in Convert

We can now add the dependency to Convert's signature, and call it to retrieve the current rate. As the caller is responsible for providing the implementation of the rates provider, it will know about all the kinds of errors that it can return. If anything wrong happens, we can simply wrap the error that we get and bubble it up.

Listing 6.51 convert.go: Final implementation

```
// Convert applies the change rate to convert an amount to a targ
func Convert(amount Amount, to Currency, rates exchangeRates) (Am
    // fetch the change rate for the day
    r, err := rates.FetchExchangeRate(amount.currency, to) #A
    if err != nil {
        return Amount{}, fmt.Errorf("cannot get change rate: %w",
    }

    convertedValue := applyExchangeRate(amount, to, r)

    if err := convertedValue.validate(); err != nil {
        return Amount{}, err
    }

    return convertedValue, nil
}
```

Fix the test

Time to fix your test. If you want to be fast, the smallest thing that implements your local interface is `nil`. Let’s try:

```
got, err := money.Convert(tc.amount, tc.to, nil)
```

It compiles. But if you try to run it, you will get a full-fledged panic attack:

```
panic: runtime error: invalid memory address or nil pointer deref
```

This message means you are trying to access a field or a method on a pointer that is clearly wrong, which is the case of our `nil` value. It is what the C family of languages calls a segmentation fault and what the Java family calls NullPointerException. At runtime, your machine is trying to access a function on an object whose address is nothing (recognisable by the value zero). Result: boom.

Wasn’t that fun to watch? At least, this happened in a test environment, and not on a production platform. Meeting with these runtime errors is always valuable. Once you’ve experienced a few, you know precisely what you’re looking for when investigating an issue. Did we access a slice past its bounds? Did we dereference an invalid pointer? Did we just divide by 0? Let’s fix this by implementing a stub in the test file: a very minimal struct that implements our interface and returns values that we require for testing. If in a bigger project you require a mock, there are a handful of tools out there that can generate one from the interface definition, e.g. minimock or mockgen (https://github.com/golang/mock). The difference between a mock and a stub is that the mock will check whether it has been called and make the test fail if the expected calls don’t match the actual ones. A stub will only imitate the expected behaviour when called, but cannot validate anything.

Listing 6.52 convert_test.go: Stub the dependency

```
// stubRate is a very simple stub for the exchangeRates.
type stubRate struct {
    rate money.ExchangeRate
    err  error
}

// FetchExchangeRate implements the interface exchangeRates with
func (m stubRate) FetchExchangeRate(_, _ money.Currency) (money.E
    return m.rate, m.err
}
```

Exercise: Update the rest of the unit test. You need to add the stub to the test case scenario and give the rate that you expect to get from the dependency.

6.5.6 Dependency injection in main

The last missing piece consists in building the EuroCentralBank and passing it to `Convert`.

Listing 6.53 main.go: Use the conversion rate

```
[...]
rates := ecbank.EuroCentralBank{}

convertedAmount, err := money.Convert(amount, toCurrency, rates)
```

```
[...]
```

And tada ! Try it, go on. Just because we're all tired by now, here is an example command again.

```
go run . -from EUR -to JPY 15.23
```

You can try invalid currencies and numbers, have fun. Play, it's not your money anyway.

6.5.7 Sharing the executable

While "`go run .`" is nice, sometimes you don't want to share the source code with other people, and only the compiled binary. Generating the executable file in Go is achieved with the following command:

```
go build -o convert main.go
```

This command generates an executable binary file, `convert`, in the `bin` directory. Of course, the location can be changed. `-o` is a flag that has a default value, as all flags do - its default value is "`.`", which is the current directory. Then, we can execute it:

```
./convert -from EUR -to JPY 15.23
```

6.6 Improvements

There are a lot of tiny problems with the implementation we presented here. The goal of the chapter was to reach a working solution, not a perfect one. Let's go over a few ideas and implement one.

6.6.1 Caching

First, we are calling the bank for every run of the tool, which is a waste of resources. For example, you could be tempted to write a script that reads a long list of amounts from a file, and, for each line, changes the amount to Philippine pesos. Currently, there would be an identical HTTP call for each line. This is quite time-consuming.

A solution would be to dump the rates in a temporary file with the date in the name. The `ecbank.Client` struct would have a pointer to that file. If the file doesn't exist, fetch the rates and dump them. If the file is too old, same. Otherwise load from the file.

You would then need to provide a way to flush the cache with a different flag.

6.6.2 Timeout

You might someday be in the situation of one of the authors of this book: trying to get the change rate between euros and British pounds, but there's some sea above your head and you've lost the 4G signal. Fun fact, the world's longest undersea section for trains is 37.9 km and the fastest train can only go at 160km/h inside. What happens when you make a http call and the server never answers? Nothing, unless you plan for it by the means of a timeout.

In this code, we're calling `http.Get`. Under the hood, this makes use of the default client available in the net/http package. While this is perfectly fine for a small example such as this chapter's, it is certainly not good enough for production code. Running go doc `http.Client` shows that one of the fields of the `Client` structure is Timeout. As you would expect, setting this will take care of interrupting calls exceeding a given amount of time. The default value of this field, which is the default value of the `http.DefaultClient`, is zero, which, as the doc reads, means "no timeout". Using `http.Client{Timeout: 5*time.Second}` would, for instance, create a client with a specific timeout, which can be safely used instead of the default client.

If you look at how the client is defined in the code, you will see a lot of default zero values:

Listing 6.54 net/http/client.go: Implementation of the DefaultClient

```
// DefaultClient is the default Client and is used by Get, Head,
var DefaultClient = &Client{}
```

It is a pointer, so anyone changing it will change it for the whole program.

This is actually a design choice: you can start the program by setting a timeout on this variable and the rest of the program can rely on it and use that timeout value. But it’s a global variable that the rest of the program can change - you have to hope that none of your libraries sets a different timeout than your timeout inside the `DefaultClient`. If you want to fathom how annoying this would be, simply imagine that your smartphone has unfortunately inverted the phone numbers of two of your contacts, and you have to figure out a way to find out.

Instead, we can declare our own `http.Client`, only to be used in our `EuroCentralBank` structure, and use the `Get` method of the client. This lets us set a timeout that will be used during, and only during, our calls to the European Central Bank.

Listing 6.55 ecb.go: Timeout example

```
// Client can call the bank to retrieve exchange rates.
type Client struct {
    client http.Client #A
}

// NewBank builds a Client that can fetch exchange rates within a
func NewClient(timeout time.Duration) Client { #B
    return Client{
        client: http.Client{Timeout: timeout},
    }
}

// FetchExchangeRate fetches the ExchangeRate for the day and ret
func (c Client) FetchExchangeRate(source, target money.Currency)
    const path = "http://www.ecb.europa.eu/stats/eurofxref/eurofx

    resp, err := c.client.Get(path) #C
    [...]
```

The main function will have to adapt:

```
rates := ecbank.NewBank(30 * time.Second)
```

We are now ready to handle undersea tunnels. The `http.Get` function will immediately return an error (and an unusable response) if the timeout is reached, and it’ll be up to the caller to decide what to do. The `net/http` package warns us, in the documentation of `http.Get`, that the errors returned are of type `*url.Error` (the pointer information is very important), and that we can use that to determine whether the call timed out. This is a nice opportunity to discover a useful function of the `errors` package.

We've already seen that we can test if an error is of a specific flavour, with `errors.Is`. Sometimes, we want to inspect the error a bit further, especially when we know there is something more than an error message that can be extracted from the error. In this case, we are informed that the error returned is, in fact, of a specific type. This means we could cast it to that type:

```
urlErr, ok := err.(*url.Error)
```

This would then allow us, provided the `ok` variable is true, to access fields and methods of the `*url.Error` structure. Let's have a look at what's over there: `go doc url.Error`.

As we can see, there are several exposed fields in that structure - the operation that was attempted, the URL that was requested, and the error itself. But what is interesting for us, here, is that we can call a `Timeout` method that returns a boolean value. This is how we can ensure we did indeed reach a timeout.

```
if urlErr.Timeout() {
```

This is nice, but there is a nicer and more idiomatic way of performing this operation: we can make use of the `errors.As` function. Its signature is simple: it takes an error, a target, and it returns a boolean - whether it succeeded. When it did, the target now contains the value of the original error.

Because `errors.As` is writing to its `target` parameter, we need to provide it as a pointer to a variable that will receive the value, just as we had to do earlier with the `Decode` method of the `encoding/xml` package. In our case, we want to pass a pointer to a variable of type `*url.Error`. Yes, that's a pointer to a pointer. But it's important to understand that we merely pass the address of a variable, and the fact that this variable is itself a pointer has nothing to do with it - we only retrieved this information from the `http.Get` documentation. Then, if everything went as expected, we can use the (now populated) `*url.Error` to check if it is indeed a timeout!

Listing 6.56 ecb.go: Checking for timeout

```
func (c Client) FetchExchangeRate(source, target money.Currency)
    [...]
    resp, err := c.client.Get(path)
if err != nil {
        var urlErr *url.Error #A
        if ok := errors.As(err, &urlErr); ok && urlErr.Timeout()
            // This is a timeout!
```

It’s now up to you to decide what should be done when a timeout is reached. It could be interesting to retry after a few moments - maybe the 4G coverage is now better and we’re out of that undersea tunnel. Or we could decide that any error we face is fatal for the process of converting money, and it’s not our converter’s responsibility to choose how to deal with network errors.

Beyond timeouts

As we’ve seen, our `http.Client` structure can be tuned with a timeout. But a timeout isn’t the only value we can set for our client - for instance, here, we’ve not overwritten the `Transport` field of our `http.Client`, which means we’ll be using the `http.DefaultTransport` in our client. The arguments for using a specific `http.Client` applies here again - we might also want to tune the `Transport` within our `Client`.

Testing our implementation

We can’t re-use the same test as previously, since we were only passing the URL of the bank to the `Client`. This time, we need to use a client that will proxy to the server’s URL. We can do this with the `Transport.Proxy` field of the `http.Client` structure. Here’s the implementation for this:

Listing 6.57 ecb_internal_test.go: Testing exchange rates

```
func TestEuroCentralBank_FetchExchangeRate_Success(t *testing.T)
    ts := httptest.NewServer(...)
    defer ts.Close()

    proxyURL, err := url.Parse(ts.URL)
    if err != nil {
        t.Fatalf("failed to parse proxy URL: %v", err)
    }

    ecb := Client{
        client: &http.Client{
            Transport: &http.Transport{
                Proxy: http.ProxyURL(proxyURL), #A
            },
            Timeout: time.Second, #B
        },
    }

    got, err := ecb.FetchExchangeRate(mustParseCurrency(t, "USD")
    ...
```

The rest of the test is the same as before. But this is only testing the happy path, let’s also test the case where a timeout occurs! For this, we’ll change the behaviour of the `NewServer` we build in the test, and, instead of writing an XML to the response, we’ll instead simulate a long wait with `time.Sleep`. We’ll re-use a similar client as in the successful test, and this time, we’ll check the error that is hopefully returned!

Listing 6.58 ecb_internal_test.go: Testing timeout

```
func TestEuroCentralBank_FetchExchangeRate_Timeout(t *testing.T)
    ts := httptest.NewServer(http.HandlerFunc(func(w http.Respons
        time.Sleep(time.Second * 2) #A
    }))
    defer ts.Close()

    proxyURL, err := url.Parse(ts.URL)
    if err != nil {
        t.Fatalf("failed to parse proxy URL: %v", err)
    }

    ecb := Client{
        client: &http.Client{
            Transport: &http.Transport{
                Proxy: http.ProxyURL(proxyURL),
            },
            Timeout: time.Second, #B
        },
    }

    _, err = ecb.FetchExchangeRate(mustParseCurrency(t, "USD"), m
    if !errors.Is(err, ErrTimeout) { #C
        t.Errorf("unexpected error: %v, expected %v", err, ErrTim
    }
}
```

Timeout’s value

When it comes to choosing a “good” timeout value, you want to think about the call you’re executing. Your timeout is your patience, you don’t want to think about what the remote service has to execute your query, because you shouldn’t know. Their implementation and performance might change without you having to change your code. A rule of thumb, though, when making calls to external resources is that a call that leaves your environment - be it a local network, or a cloud platform - should be allowed up to a few seconds. You need to account for all these time-consuming network handshakes. When running locally, a few seconds are only tolerable for big processes, and the usual value is less than a second. These numbers aren’t set in stone, as they need to be tuned for your own use cases. Allowing only 20 milliseconds for a call over the internet is too short, and if your timeout for a local query is 30 minutes, there is something fishy in the architecture.

6.6.3 Alternative tree

In your everyday developer life, you will be led to import external libraries from the open-source world. This is very frequent for most libraries. In general, they are organised with the exposed types at the root of the module as it minimises the path to reach the required package for users. Compare `github.com/learngo-pockets/money` instead of `github.com/learngo-pockets/moneyconverter/money`.

We have created a folder named `money` containing one file, exposing all the methods and types to the users and have everything at the root.

If you need a main, it is common to create it in a folder `cmd` for command. Here is an example of a project importing our library.

```
$ tree
.
├── cmd/main.go
└── github.com/learngo-pockets/money
└── money.go
└── go.mod
```

Congrats! You are done! It was a tough chapter with different concepts that we will practice again over the following chapters.

6.7 Summary

- The `flag` package exposes functions that allow us to retrieve both arguments and optional parameters from the command line. We can even set default values to our flags. Remember to call `flag.Parse()` before checking the values of the flags!
- When implementing a functionality, it is good to declare types that mirror the core entities that we will have to handle. In our case, we created a `Currency`, a `Decimal`, an `Amount`, and we defined what `Convert` should do before writing a single mathematical operation or calling a single function. We also knew from the start how to organise our code, which types and functions should be exposed.
- The `fmt.Stringer` interface is a simple interface that will make printing complex structures nicely.
- Floating point numbers are inaccurate, at best. There should always be room for a margin when comparing two floating point numbers. Operations on floats will sometimes seem nonsensical, because of the precision limitation. Knowing their precision and the precision of the computation they're used for will help make a choice of float32, float64, big.Float - or to go for something else.
- A package is most often built starting with its API, and finishing with its unexposed functions.
- Go's net/http package offers functions to perform HTTP calls, such as GET or POST requests, over the network. It offers a default client, which can be used for prototyping, but shouldn't be kept in production-level code, for security reasons.
- A HTTP call will return a response that contains a status code. Checking this status code is mandatory - the code informs how meaningful the

body of the response is. Status codes are divided into 5 classes, but the most important are 200 (and 2xx), which means everything went fine, 400 (and 4xx) which means the request might be incorrect - some fields could be missing, some authentication could be wrong, … -, and 500 (and 5xx), which mean the server faced an issue and couldn't process the request.

- Testing HTTP calls should be agnostic from external behaviour. Go provides a httptest package to mock HTTP calls by providing an infrastructure to set up an HTTP server for tests. a useful NewServer function.
- A clean code separates the responsibilities of each package. Retrieving data isn't the same task as computing data, and should be handled in a separate piece of code. Go offers extremely simple interfaces that allow for dependency injection. Dependency injection makes code simple, and tests even simpler.
- Stubs are a very nice way of implementing interfaces. Simply declare a struct where you need to implement an interface - usually in a _test.go file - and have it implement the interface. A stub is very useful when trying to improve test coverage, but it can only be used for unit tests, as they don't check the whole logic of the call.
- Exposed functions, in a package, should return sentinel errors. This makes using a package clean and simple. Within the implementation of the package, the decision of creating the sentinel errors in exposed functions or in unexposed functions is left to the developer.
- Using `errors.Is` how we test if an error is a sentinel. Using `errors.As` is how we access an error's fields and methods.
- Go's `encoding/xml` package provides a function and a method to decode an XML message. In order to be able to decode some bytes into a structured variable, Go's syntax requires that the fields we want to decode be described with the XML path where their value can be read. It is even possible to skip layers by using the > character in that path.
- The toolchain offers the `go build -o path/to/exec .` command, which generates an executable file at the specified location.

7 Caching with generics

This chapter covers

- Using generics in Go
- Not using generics when they are not needed
- Creating type constraints
- Goroutines, parallelism and concurrency
- Race conditions
- Mutexes
- Some Go proverbs

Imaging being in school, when there was an important piece of home assignment that you realised was due tomorrow, and you hadn’t even started? How nice was it to phone a friend and ask if, maybe, they had the answer to question 2.b? Of course, you would have their answer, not yours, but getting this solution early also meant you could spend time on some other part of the homework. Of course, teachers will discourage you from being lazy and cheating - the whole point of these assignments is to have you use your brain power, to either learn or understand a lesson.

Computers, on the other hand, won’t judge you for taking shortcuts. A cache is such a shortcut: it’s a key-value storage which can access data that has at least been computed once in the past - as long as it makes sense to access it. Caches are often used when we know a slow function returning a value will be called several times with the same input - and that the output value should be the same every time. There are cases when we want to use a cache - for instance, you know that “the city with the longest name is: Bangkok”. This isn't something that you need to check every morning. There are cases when we specifically don’t want to use a cache - “The current exchange rate of the Algerian dinar to Euros is: ??” - in these cases, an outdated response could lead to confusion. And sometimes, there are situations where using a cache could be acceptable - “The total population of Ethiopia is: 123,967,194” doesn’t really require an instantaneous answer, the value from last week having the same magnitude as that of this week.

In this chapter, we will present generics and implement a naive cache. Through tests, we will show our first approach isn't good enough and needs to be strengthened. Then, we will add a "time to live" to values in our cache, to make sure we don't store outdated information. Finally, we'll cover some good practices of using caches.

Requirements:

- Write a cache of type key, value
- We should be able to store data, read and delete it
- It should be concurrency-safe

7.1 A naive cache

Let's start with a short definition. A cache is a storage for retrieving values that were previously saved. In order to access these values, the user of the cache will use the same key as they did when they registered the value the first time.

There are two important notions regarding a cache that need to be evoked here.

- A cache should be able to store any new pair of key and value;
- When retrieving a value using a key, the cache should return what was previously stored in it;
- Usually the whole reason for it is speed: a cache needs to be fast.

Here are some examples of pairs of keys and values a cache could hold:

- Phone number (`string`) - Name (`string`)
- Year (`uint`) - All the medallists, per country, at these Olympics (`map[Country][]Athlete`)

Through these examples we would like to show that a key can take many forms, and that a value can take even more. We'll cover this in detail later, but for now, we need to understand generics and how to write them in Go, so we can write our first implementation of a cache.

We need to start with a bit of theory, but we will try to keep it short. We are here primarily for hands-on projects, after all.

7.1.1 Introduction to generics

Go is strongly typed by design. It means every variable has to have an explicit type. You can know by looking at the code which variable is a string, which is an `int`, and therefore you can understand how they interact and what the code does. If your function takes a `float64`, there is nothing else you can give it than a `float64`.

```
func prettyPrint(f float64){
    fmt.Printf("> %f", f)
}

prettyPrint(.25)
```

This is nice and clear, but how can we write a function that prints an `int`? Do we have to copy the tree lines and change a few characters? Well, believe it or not, ancient generations of Gophers remember the time when yes, you indeed had to copy all of these little functions around. But generic types, also known as generics, arrived and saved us from so much boilerplate.

Generics let you write code without explicitly providing the type of the data a function takes or returns. The types are instead defined later when you use the function. Functions are just an example. You cannot fathom the amount of boilerplate code (and bugs) that got swept away by this great feature. But like all good things, we should not overuse them, and we should know what we are doing with them.

How does it work? Instead of specifying a real type, like `float64` above, we define a generic type and give it a *constraint*. The least-constraining type constraint is the keyword `any`, which really means what it says: any type at all. Here is an example of a single function that first accepts a `float64` as its parameter `t`, and then accepts a `string` as the same parameter `t`. You can notice that the signature of the function is somewhat different - we've used square brackets between the name of the function and its parameters. We've declared in these square brackets that `T` is a placeholder for the type constraint `any` for the scope of this function's declaration. The type `T` is set when we call the function.

```
func prettyPrint[T any](t T){
    fmt.Printf("> %v", t) #A
}

prettyPrint[float64](.25) #B
prettyPrint[string]("pockets")
```

In this example, we can no longer use the `%f` verb in `Printf`, as this is only usable with floating point numbers. We specify, when we call the function, what the type `T` should be.

But what is type `T`, specifically? It depends on what we decide to pass to the function when it is called. The function is *parameterised*: at compilation time, the required flavours, here a `float32` and a `string`, will both be generated and compiled like any concretely typed function. The only difference is the reusability of your code, which is now way better.

Type inference

If the type `T` appears in the parameters of the function, the Go compiler is able to detect which type it represents each time the function is called and doesn't need the hint we're giving it by using square brackets when we call the function. For instance, in the previous example, since the parameter of the function is precisely of type `T`, the Go compiler will notice that we call this function with a `float64` and with a `string` parameter. The lines above can be simplified to the following:

```
func prettyPrint[T any](t T){
    fmt.Printf("> %v", t)
}

prettyPrint(.25) #A
prettyPrint("pockets") #B
```

Type inference might make generics seem a bit magical. It's worth noting type inference will only happen on input parameters, not on returned types.

Type constraints

We saw that `any`, introduced by Go 1.18 along with generics, is a keyword that can be used as a type constraint for, actually, no constraint at all. It is equivalent to `interface{}`. There is only one other built-in constraint: `comparable` is implemented by all comparable types, that is all types that support comparison of two elements using == or `!=`. Into this category fall booleans, numbers, strings, pointers, channels, arrays of comparable types, structs whose fields are all comparable types. Slices, maps, and functions are not comparable. If you create a map and want its key to be generic, it will have to be `comparable`. More on that soon.

We have seen in the previous chapters how interfaces work, and how any structure can implement an interface as long as it has all methods attached with the right signature.

Type constraints are interfaces, and, of course, you can declare your own. As `%v` is not exactly formatting prettily, here is a different version:

```
func prettyPrint[T fmt.Stringer](t T){
    fmt.Printf("> %s", t) #A
}
```

The difference here is that now we cannot give floats and integers to our function anymore because they don't implement the `Stringer` interface, but we can give any structure that does implement it. If you remember the `Amount` from our previous chapter's money converter - it did implement this interface, and we can therefore call:

```
amount, err := money.NewAmount(...)
prettyPrint(amount)
```

Generic types

Let's say we want to define a group of `Amounts` so that we can perform some specific operations on them.

```
type Group []Amount

func (g Group) PrettyPrint() {
    for _, v := range g {
        prettyPrint(v)
    }
}

func main() {
    var g Group
    g.PrettyPrint()
}
```

But of course, suppose we wanted to do the same to pencils and clouds and dresses. So, we would decide to make this group generic:

```
type Group[T any] []T
```

`Group` is parameterised: it is a group of `T`, and it is an alias for a slice of `T`. Go won't allow us to write a method with a receiver of type slice - `func (s []string) Do()` - the compiler would complain with a message saying "invalid receiver type". But using our `Group`, we are able to write a method on that parameterised type:

```
func (g Group[T]) PrettyPrint() {
    for _, v := range g {
        prettyPrint(v)
    }
}
```

The receiver of the method needs to be parameterised too. Indeed, the variable `v` in the loop will be of type `T`, and the compiler needs to know where to look for what `T` means. Until we instantiate a `Group`, it means nothing.

```
var g Group[Cloud]
g.PrettyPrint()
```

Here we are: the compiler can now create a version of the `Group` that supports Clouds, and only clouds.

Declaring your own constraints

One last thing you need: what if you want to support multiple integer types? Or support `int` and your own `PocketInt` but nothing else? You cannot make primary types implement an interface, but you can define union interfaces.

```
type summable interface {
    int | int32 | int64
}
```

This means that any function that can take a summable as parameter will accept any of int, int32 or int64, and will be able to use the + operator on it.

However, now, if you define a new type that is an `int` (in other words, a new type whose underlying type is int), it won't be included. For example:

```
type age int
```

To support all things that are actually `ints`, we use the `~int` (with a tilde) syntax to include all types whose underlying type is `int`. The following interface includes the type `age` above.

```
type summable interface {
    ~int | ~int32 | ~int64
}
```

If you are talking about stars and specifically need an int64 for ages, you can change your type and everything will fall into place.

Enough with theory, let's write this cache.

7.1.2 Project initialisation

Since this project is about creating a library, we'll use a common organisation of our files. In Chapter 4, we exposed a module that contained a package `pocketlog`. Although this was nice when we needed to introduce packages, most open source libraries will expose their types at the root of the module, as this prevents having cumbersome import paths. Here, we want our users to import our cache package with the minimum effort and this means placing our cache as early as possible.

Our file organisation will be as follows:

```
learngo-pockets/genericcache
├── cache.go
└── go.mod
```

This will allow anyone to use our library by importing our module, and then using `genericcache.Cache`. We will, of course, require other files - but these are the bare necessities that our users will need.

Start by creating a `genericcache` directory, and in there run the following line to initialise the module:

```
go mod init learngo-pockets/genericcache
```

7.1.3 Implementation

As we've seen earlier, a cache is a place to store data in a way we can retrieve it easily. In our case we decided to have a key value storage that could be used for almost any type of key, and any type of value. We need our keys to be comparable - that is, we need to know if two keys are considered equal. This is achieved with the constraint `comparable`.

In our tests, we use simple cases where the key is an integer and the value a string - but feel free to use other types instead.

Create a type `Cache` in your package, in a `cache.go` file. It is a `struct` holding one field: a map. The type of the keys and values is parameterised.

Listing 7.1 cache.go

```
// Cache is key-value storage.
type Cache[K comparable, V any] struct {
    data  map[K]V
}
```

As you can see, we have defined two types, giving them one-capital-letter names as per convention. `K` for key and `V` for value seem as self-explanatory as we can get.

Already we face a decision: should we store a `map[K]V` or a `map[K]*V`? In human terms, should our cache store copies of the user's values, or should we only store pointers to them? There are pros and cons to each choice, and a tradeoff has to be found between using more memory (with copies of the values) and using more CPU (because of pointer indirection having to be resolved). We decided to go with the implementation above - after all, if a user wants to use a type `V` that is a pointer to their values, they can still do it! This also means that our cache is fully responsible for its memory, and that once a value has been added to the cache, it can’t be updated without a call to the cache.

New

Since our `cache` contains a map, we need to initialise it. A side effect of using a map in a structure is that the zero value of variables of that type should be treated with caution. Not being able to use a zero value is an argument to keep a type unexposed.

Listing 7.2 cache.go

```
// New creates a usable Cache.
func New[K comparable, V any]() Cache[K, V] {
    return Cache[K, V]{
        data: make(map[K]V),
    }
}
```

Read

The most common operation executed on a cache is usually to read from it, as that’s the whole point of our cache. Let’s write a method to achieve this. This method accepts a key of the adequate type, and returns the value - also of the adequate type. It is up to us to decide what to return when the key isn’t found in the cache, which is the case when the value hasn’t been stored there yet. As a reminder, Go’s default map implementation returns a boolean and a value, and that’s what we’ll be using here. Should you want to use errors, you’ll have to decide how to return the error: via a constant and a local type, or via an exposed variable. You can also, as usual, call `errors.New` directly in your return line, but it will be harder for users to compare with a known value and decide what to do next. We simply think having the same interface as a map makes things clearer for the end user.

Listing 7.3 cache.go: Implement Read method on the cache

```
// Read returns the associated value for a key,
// and a boolean to true if the key is absent.
func (c *Cache[K, V]) Read(key K) (V, bool) {
    v, found := c.data[key]
    return v, found
}
```

The most-used function is written. We could unit-test it, but in this situation, an integration test involving multiple operations seems a better idea. If we write a unit test now, it will be extremely tied to the implementation choices and will not help us in any future refactoring. We would end up testing whether Go can read from a map, which is already covered by the Go developers.

In order to write the integration tests, in order to read from the cache, we need to first be able to write something into it.

Upsert

If we want to read a value from our cache, we need to expose a way of adding it in there.

A question to be raised early here is “should we let the user insert the same key several times ?”. In most caches, a “recent” value is usually more interesting than an “old” value. For this reason, we decided to silently overwrite any previously existing value in our cache - but other implementations might decide to return an error if the key is already present when we try to add it in the map.

Since we’re overwriting any potentially existing data, we can name our method `Upsert` - a combination of “insert” and “update”. It guarantees the key will be present in the cache, associated with the specified value.

`Upsert` could return an error. For instance, we might want to limit the number of elements in our cache - hitting a limit would be a valid reason to divert from the happy path. Let’s keep this door open from the start. After all, returning an error is perfectly normal in Go.

Listing 7.4 cache.go: Implement Upsert function on the cache

```
// Upsert overrides the value for a given key.
func (c *Cache[K, V]) Upsert(key K, value V) error {
    c.data[key] = value

    // Do not return an error for the moment,
    // but it can happen in the near future.
    return nil
}
```

Nearly there. You can start writing a unit test that writes, reads, checks the returned type, checks the returned value for an absent key, writes another value for the same key, etc. There are a lot of different situations that can already be covered.

Delete

The last operation we need is deleting a key from the cache, for when we know that this value is stale. For example, say we are pre-computing the list of group conversations that each user is part of. Somebody creates a new conversation and invites 5 people. Each of them will need a new computation of the listing, but only when they open the messaging app. We can invalidate their keys and let the system re-compute next time the list is required.

Most caches grow with no real limit. At the end of this chapter, we'll expose a few ways to keep the cache manageable.

Let's expose a method to ensure an item is no longer present in a cache. This method will take a key, and remove the entry from the cache. But what if the key isn't present in our cache to begin with?

Go's answer to this question - at least for maps - is to be idempotent: rather than considering that we're trying to remove an entry, we think of this action as ensuring that this entry is not in the map after the execution of `delete`. We decided to follow the same philosophical approach with our `Delete` method. If the key is not in the cache, our method performs no operation - commonly shortened as “no-op”.

Listing 7.5 cache.go: Implement Delete method on the cache

```
// Delete removes the entry for the given key.
func (c *Cache[K, V]) Delete(key K) {
    delete(c.data, key)
}
```

To test - or not to test

Unit-testing a one-liner like this is a question that a dev team needs to solve as a group: what is the level of testing we want to have on this, are we testing our own code or the Go map itself? Since we didn't add any logic on top of the map, we decided that our code - so far - didn't need unit tests. This doesn't prevent us from writing some small functional tests - a list of calls that ensures that we indeed insert values in our cache, and that we're able to retrieve them.

Our first implementation of the cache seems to cover our needs - we can store data, we can retrieve values using the keys that were used to insert them, we can remove some data, if need be. The world seems perfect. That's precisely the moment when someone in your team makes a comment in the code review - "Is this thread-safe?"

This is an excellent question, and, in order to answer it, we need to understand how we'll be able to prove it is (or isn't). "Thread-safety" is invoked when several threads - parallelised parts of a program - access the same resource simultaneously and try to alter it. To imagine a real-world parallel, suppose you're having dinner with a friend, and suddenly, both of you are thirsty. You both want to grab the bottle of water to fill your glass. If you were to grab the bottle at the same time, and pour in different glasses at the same time, there would probably be water all around the place, and you wouldn't be sure your glass is full by the end of it. For our cache, this would mean, for instance, having two "threads" try and write a different value for the same key. How do we know this won't break anything?

7.2 Introducing goroutines

Let's return for a moment to the basics of what a computer is - a set of devices connected together. We've got a processor (CPU), the central unit in charge of performing the actual computing, some memory bars, used to store values used by the computing module, a power source, and many extra parts such as a hard drive to store persistent data, a motherboard to connect everything together, etc. Here, we’ll focus on the processor.

A processor is in charge of running the binary code that was generated by the compiler. Each program, when launched, is loaded in the memory, and then executed on the processor. But, wait, does that mean that a processor can only run a single program at a time? The answer is no, for two reasons. The first one is that programs run on cores, which are parts executing the binary code in the processor. In the 2000s, processor manufacturers started shipping their processors with 2 or more cores. Each core can dedicate its activity to only one task at a time. If you have more cores, they can run multiple tasks at the same time independently on a single computer. The other reason is that our operating system, which is also a program, coordinates different tasks and programs to run on these cores. The user interface has to run somewhere. There must be some running piece of code that reads input from the keyboard. There must be something that communicates with your hard drive.

In order to prevent a computer from freezing because a core would be running a program that wouldn’t end, computer scientists have implemented schedulers for CPUs - a way of “pausing” a program to let another one run. This is how we could have multiple programs run at once before the democratisation of cores.

For many programmers, the fact that several cores were present on a machine meant that there were more resources that could be used to run a program. After all, if the load could be balanced on two cores instead of one, maybe the program could run twice as fast! Let’s douse your hopes right now: in most cases, this doesn’t work.

How can we use this feature? Pieces of a program that run independently at the same time are called threads, coroutines, fibers, or in Go, goroutines.

In this section, we’ll see what goroutines are, how to create them, and how to manage them.

7.2.1 What’s a goroutine?

Many other programming languages use the term “thread” when they describe a task that is launched for parallel execution. In Go, we see things differently. First, we don’t use system “threads” directly. Instead, we use Go’s goroutines. They are managed by the Go runtime layer that runs along your Go program. There are many differences between goroutines and threads, but this isn’t a topic for this book. Instead, let’s remember that a goroutine is a way of launching a piece of code in the background.

```
a = taskA()
b = taskB() #A
```

Of course, most of the time, we want our program to execute sequentially - we want the second task to be run after the first. But sometimes, we don’t *need* the first task to have successfully returned before we run the second one.

Let’s make a real-world comparison: suppose you’re preparing a curry. You have your pot with curry sauce, and your pot with rice. The recipe tells you that each one should cook for 10 minutes. You could first cook the sauce for 10 minutes, and, when it’s ready, cook the rice for 10 minutes and the sauce is getting cold. You’d spend 20 minutes preparing your dish, when you could have cooked both pots at the same time - provided you had enough burners - reducing that total time to around 10 minutes.

This is what goroutines address. They allow you to run several tasks simultaneously - provided you can launch them. This last bit is usually not an issue - goroutines are really light to handle, and unless you start creating millions, you should be fine.

Now, there is a word that has been used in this section that needs a closer look. We’ve used “in parallel”, “simultaneously”, “in the background” or “concurrently” to represent the idea that a goroutine doesn’t block its caller. Over the years, these words have sometimes been used interchangeably. Fortunately for us, Rob Pike wrote some proverbs for Go (https://go-proverbs.github.io/), and one of them deals with this specific topic. It also helps us getting clear definitions of what each of these words mean, as we explain right after:

Go proverb

Concurrency is not parallelism.

This proverb highlights that having two (or more) goroutines does not guarantee any simultaneous execution on parallel cores, but that they will be executed independently, for better or for worse. Concurrency should focus on how to write code to support goroutines, while parallelism is what happens when the code is executed.

7.2.2 How to launch a goroutine

Let’s remember that Go was created with, in mind, the idea that running goroutines should be simple. The creators of Go made it extremely straightforward: if you want a function to run in the background, you simply prefix its call with `go`. That’s it. It doesn’t require any specific import or compilation options. Here is a simple example:

Listing 7.6 parallel.go: An example of a program running goroutines

```
package main

import (
    "fmt"
    "time"
)

func printEverySecond(msg string) {
    for i := 0; i < 10; i++ {
        fmt.Println(msg)
        time.Sleep(time.Second)
    }
}

func main() {
    // Run two goroutines
    go printEverySecond("Hello") #A
    go printEverySecond("World") #B

    var input string
    fmt.Scanln(&input) #C
}
```

When the execution reaches the `fmt.Scanln` line, we have our three routines running at the same time - the main one, and those printing messages every second. But there is a small drawback to using goroutines - they're launched "in the background" - but this means that they finish without letting the caller know! This is what the problem looks like for our previous example, in lines of code:

```
go cookCurrySauce()
go cookRice()
// how do I know the food is ready?
```

There are two major ways of dealing with this - the first one is to use channels, and the second one is to use a library that solves the problem.

7.2.3 Getting notified a goroutine has ended using channels

Go has a specific type called "channels" that it can use for communication between goroutines.

Go proverb

Don't communicate by sharing memory, share memory by communicating.

A channel is how we communicate in Go. We can send information such as triggers, new data, results, errors, etc. between goroutines through channels.

A channel can be seen as a conduit to which data can be sent - and from which data can be retrieved. A channel, in Go, is declared for a specific type of message it will contain. For instance, if a channel were to be used to convey integers, we'd write the following line:

```
var c chan int
```

Channels, like maps and slices, need to be instantiated with the `make` function. When instantiating them, we can decide whether we want a channel to be buffered - it will only be able to contain up to X elements - or unbuffered - with no limit to the number of elements it contains at any given time.

```
c := make(chan int, 10) #A
c := make(chan int) #B
```

A buffered channel that has reached its capacity becomes “blocking” on writing attempts - as long as no message is read from the channel when it has reached its capacity, any sending to the channel will wait for a spot in the queue, blocking the goroutine.

The syntax to write to and read from a channel uses arrows:

```
c := make(chan int) #A
c <- 4 #B
i := <- c #C
```

The power of channels, in Go, comes from the fact that items are read from the channel in the same chronological order they were sent. In other words, *first in, first out*.

Finally, when no new messages are expected, a channel should be marked as closed for writing. For this, we use the built-in function `close`. It is still possible to read from a channel after it is closed.

```
c := make(chan int)
c <- 4
close(c) #A
i := <- c
```

Reading from a channel is a blocking call. If there are no messages in the channel, the execution waits till we find one. As a result, we can use a channel to notify that a goroutine is done:

```
c := make(chan struct{}, 1) #A

go func(doneChan chan <- struct{}) {
    defer func() { #B
        log.Println("done")
        doneChan <- struct{}{}
        close(doneChan) #C
    }
    // run task
}(c) #D

_ = <- c #E
```

We introduced two commonly used notions in this example. First, a channel can be used to notify its listeners. Here, we only want to notify that we're done - and for this, we use the Go trick of empty structures: `struct{}`, because empty structures are very light (they have a memory footprint of 0 bytes). We don't need a convoluted structure that would transport data around, and so we don't use one. There's no point in overdoing it here.

The second interesting part is the signature of the function we run as our goroutine. A small arrow `<-` squeezed its way between the words `chan` and `struct{}`. When we declare a function, we can be a bit more specific than "here's a channel for you to use": we can specify in the signature of the function whether a channel should be used for reading messages from it, for writing messages to it, or for both. If a function should only read from a channel of strings, its signature can be written as `func read(c <- chan string)`. Visually, the arrow points out of the channel, an indication that messages will be read from the channel. If we want to specify that we want to write to a channel in a function, we can use the `func write(c chan <- string)` syntax. Visually, the arrow helps us understand that strings will be sent into the channel.

If we wanted to both read and write from a channel, the syntax is simply `func rw(c chan string)`. No arrows this time. However, we discourage passing a channel for both reading and writing to a function - this suggests the function's scope is too big, and we should be able to extract the reading and the writing into two different functions.

Finally, a channel should be closed when the job is done, to tell listening goroutines that no more data will arrive. When a single function is in charge of writing to a channel, that function should be in charge of closing the channel. Leaving it open is not a problem if you don't want to signal listeners that you're done.

Let's have a final look at how we'd write our synchronisation point if we have to handle several goroutines:

```
numRoutines := 2
c := make(chan struct{}, numRoutines) #A

go cookRice(c) #B
go cookCurry(c) #B

for i := 0; i < numRoutines; i++ { #C
    _ = <- c
}
```

Bon appétit.

7.2.4 Running goroutines and having a synchronisation point

While using channels works perfectly fine, it always feels like reinventing the wheel. While this is fine when your road needs specific wheels, it so happens that Golang provides two libraries that replace these channels nicely. One is present in the standard library, while the other is (still) in the experimental packages of the Go sources.

Using sync.WaitGroup

Let's have a look at the `sync` package, in particular its `WaitGroup` type. `go doc sync.WaitGroup` tells us that `WaitGroups` can be used to wait for goroutines to finish - which is exactly what we're trying to do here. The `WaitGroup` type exposes three methods:

- `Add(delta int)`: Registers a number of new goroutines to wait for. This can be called several times.
- `Done()`: Used by a goroutine to notify the `WaitGroup` that it has completed its task. Should be called in a `defer` statement.
- `Wait()`: The synchronisation point, called after `Add()` and after the goroutines have been launched.

Let's give these a try with our cooking example:

Listing 7.7 Cooking example using sync.WaitGroup

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    wg := &sync.WaitGroup{} #A
    wg.Add(2) #B

    go cookRice(wg) #C
    go cookCurry(wg) #C

    wg.Wait() #D
}

func cookRice(wg *sync.WaitGroup) {
    defer wg.Done() #E
fmt.Println("Cooking rice...")
// prepare rice
}

func cookCurry(wg *sync.WaitGroup) {
    defer wg.Done() #E
    fmt.Println("Preparing curry sauce...")
    // prepare curry
}
```

In this example, we created a default `WaitGroup`. Because `WaitGroups` don't expose any fields, they will always be created with the exact same line: `wg := &sync.WaitGroup{}`. Well, not absolutely always - you could name yours differently - but `wg` is a common name for a `WaitGroup`.

The second step is to set the number of goroutines that this WaitGroup will be in charge of. Here, we made a single call to Add, but it's perfectly fine to call Add(1) several times. This is quite common when you have to deal with loops. We could have written our code this way, which makes it easier to refactor, if you want bland rice or just the sauce:

```
    wg.Add(1)
    go cookRice(wg)

    wg.Add(1)
    go cookCurry(wg)
```

Then, the important part is to defer a call to `wg.Done()` in each function we call. This is why we need to pass a pointer to the WaitGroup in the signature of each of these functions. Indeed, if we had passed a copy, each function would call `Done()` on a copy of the WaitGroup `wg`, and the original `wg` (in `main`, in our code) would never be notified. In this case, a call to `Wait()` will eventually result in a `panic`. For more details on passing values by copy or by reference, see Appendix E.

Finally, we call `wg.Wait()`, which will return after the same number of `Done()` have been called as the sum of all the `Add(n)` we've performed on this `WaitGroup`.

`WaitGroup` is a very commonly used wait of synchronising goroutines that we've launched into the wild. Under the hood, in order to keep track of how many goroutines aren't completed yet, it uses a field of type `atomic.Uint64`. It's interesting to know that Go exposes types that can serve for atomic operations - but we won't dive into this world here. They work great for functions that do their thing on their own. However, if anything goes wrong and an error needs to be captured, the only way is through an error channel that we pass to each goroutine, and from which we read after the call to `Wait`:

```
wg := &sync.WaitGroup{}
wg.Add(2)
errChan := make(chan error, 2)

go cookCurry(wg, errChan)
go cookRice(wg, errChan)

wg.Wait()

// handle the error, if any
select {
case err := <- errChan:
    // deal with the error
default:
    continue
}
```

As you can see, we can retrieve some errors from the goroutines with an error channel. Unfortunately for us, we had to pass a channel around to read errors, and the whole point of using a `WaitGroup` was to not have to use channels in the first place… Well, guess what? There is a library that allows us to handle errors when we're using goroutines.

Using golang.org/x/sync/errgroup

`errgroup` is, as you can see, a package that is not in the standard Go library. This means that, if we want to use it, we need to start by importing it as a dependency of our module: `go get golang.org/x/sync/errgroup`.

Now, let’s have a look at what this package exposes:

```
go doc golang.org/x/sync/errgroup.
```

We can find a type `Group` in there, and four methods - we’ll only cover three of them here, as they’re the most commonly used. But, first, how do we create a `Group`? Well, we can either use a zero value - `eg := errgroup.Group{}`, or we can use the `errgroup.WithContext(ctx)` function. In our simple example, we don’t have contexts and we will go with the first option, but, in the vast majority of cases, using the second option is recommended, as you’ll have a variable of type `context.Context` closeby. We will cover contexts in a later chapter. Internally, an `errgroup.Group` is a `sync.WaitGroup` with extra fields to handle - mostly - context and errors.

Now, what can our `Group` do? It has a `SetLimit(n)` method, which reminds us of the `Add(n)` method of the `WaitGroup`. They are different, though, in that when we called `Add(n)`, we needed to have `n` equal to the number of goroutines we were launching (and for which we’d later call `Done()`). `SetLimit` doesn’t work the same way: instead of immediately defining how many goroutines will be launched (the `errgroup.Group` tracks this internally), we specify a maximum number of goroutines allowed to be running at the same time. Most of the time, you will want this value equal to the number of goroutines you are running, which is the default value, but sometimes your goroutines make use of a resource that doesn’t scale well with load - maybe each of your goroutines calls the database, and the database can only handle 10 calls at a time. In such cases, it’s perfectly valid to have a hardcoded limit in your `Group`.

It has a `Wait()` method, also quite similar to that of the `WaitGroup` type, except that it returns an error. This is quite important, as we’ll soon see. And finally, it has a `Go` method that takes, as its parameter, a function returning an error. This `Go` method is in charge of launching the goroutine - it is also in charge of letting the `Group` know when this function finishes.

As we know, many functions written in Go can return an error. In our example, the `cookCurry` could, for instance, return an `ErrIngredientNotFound` error. All of our functions could be returning an error, and we don't want to deal with the problems of retrieving all of them. The `Wait()` method of the `errgroup.Group` type returns an error that happened in one of the goroutines. It doesn't return any error that happened there - it returns the (chronologically) last one.

Now we know how to use an `errgroup.Group`, let's use it in our cooking example:

Listing 7.8 Cooking example using errgroup.Group

```
package main

import "golang.org/x/sync/errgroup"

func main() {
    var g errgroup.Group #A
    g.SetLimit(2) #B

    g.Go(func() error { #C
        cookRice()
        return nil
    })
    g.Go(cookCurry) #C

    err := g.Wait() #D
    if err != nil {
        // handle error
    }
}

func cookRice() {
    // cook rice here
}

func cookCurry() error {
    // cook curry here - this may return an error
    return nil
}
```

That’s it! We’ve now seen three ways of controlling the synchronisation of goroutines. While we can use channels to notify that a function is returning, it’s quite common to use `sync.WaitGroup` when we want to launch any number of simultaneous calls, or to use `errgroup.Group` when we also want to retrieve any error from these calls.

7.3 A more thread-safe cache

But let’s get back to the initial question - is our cache thread-safe? Now we know how to run goroutines, let’s test it! But before we run any test, let’s keep a very important quote regarding testing by Edsger Dijkstra in mind.

Edsger Dijkstra

Program testing can be used to show the presence of bugs, but never to show their absence !

First, let’s have a look at the test we currently have and notice one thing: it’s extremely linear. It validates that, if we do a specific operation before another one, then the output is predictable. Does it run anything in goroutines? No - which means it proves absolutely nothing about thread-safety.

Our cache could possibly be used by several goroutines during the execution of a program - for instance, several incoming requests could be processed at the same time, causing the cache to be updated in a very short window. Let’s start by writing a test that simulates these “simultaneous” calls.

Using goroutines

For this, we’ll use the `sync.WaitGroup` - we need to run goroutines and we want to make sure they’ve all finished before we can return from the test. In order to make things “problematic”, let’s have each of the goroutines write a different value in the same cache, every time for the same key. Here is what we write:

Listing 7.9 Testing the cache with goroutines

```
func TestCache_Parallel_goroutines(t *testing.T) {
    c := cache.New[int, string]() #A

    const parallelTasks = 10 #B
    wg := sync.WaitGroup{}
    wg.Add(parallelTasks) #B

    for i := 0; i < parallelTasks; i++ {
        go func(j int) { #C
            defer wg.Done()
            c.Upsert(4, fmt.Sprint(j)) #D
        }(i)
    }

    wg.Wait() #E
}
```

In this test, we launch 10 goroutines, and each one is in charge of writing a different value for the same key in our cache.

Using t.Parallel()

Alternatively, we can make use of the `testing` package to execute parallel tests. This feature is particularly useful in two scenarios: when you want to reduce the time your tests will take, because you know some steps are independant and can be run simultaneously, and when you want to make sure you don't have data races.

The gist is as follows: if a test function contains the line `t.Parallel()`, the Go test framework will run it along with other functions in the same scope that also have the `t.Parallel()` line. In other words, the execution of this function won't be blocking for the execution of other test functions.

Let's write a test using the `t.Parallel()` feature. In our test, we want the same index of our cache to be written at by two different calls, with different values in each case.

Listing 7.10 Cooking example using t.Parallel()

```
func TestCache_Parallel(t *testing.T) {
    c := cache.New[int, string]() #A
    t.Run("write six", func(t *testing.T) {
        t.Parallel() #B
        c.Upsert(6, "six")
    })

    t.Run("write kuus", func(t *testing.T) {
        t.Parallel() #C
        c.Upsert(6, "kuus")
    })
}
```

Now let’s run it and see what happens: `go test .` - everything seems fine. However, we’re cheating here - we’ve written this test because we know something should go wrong. We know that upserting two different values “at the same time” is precisely a data race, and we want it to be caught. But how can we achieve this?

7.3.1 Using go test -race .

The `go test` command comes with several flags, here’s how to find them. `go help test` returns a short list - namely, `-args`, `-c`, `-exec`, `-json` and `-o` - but it also informs us that the flags from the `build` command are inherited by the `test` command. Let’s have a look at the output of `go help build`, then. One of the first flags provided is `-race`, which “enables race detection” - precisely what we’re looking for.

Let’s run our test again, but this time with the `-race` flag: `go test -race .`: we get the following output

```
$ go test --trimpath -race .
==================
WARNING: DATA RACE
Write at 0x00c0000a53e0 by goroutine 13:
  runtime.mapassign_fast64()
      runtime/map_fast64.go:93 +0x0
  learngo-pockets/genericcache.(*Cache[...]).Upsert()
      learngo-pockets/genericcache/cache.go:28 +0x124
  learngo-pockets/genericcache_test.TestCache_Parallel.func1()
      learngo-pockets/genericcache/cache_test.go:73 +0x97
  learngo-pockets/genericcache_test.TestCache_Parallel.func2()
      learngo-pockets/genericcache/cache_test.go:74 +0x47
```

```
Previous write at 0x00c0000a53e0 by goroutine 20:
  runtime.mapassign_fast64()
      runtime/map_fast64.go:93 +0x0
  learngo-pockets/genericcache.(*Cache[...]).Upsert()
      learngo-pockets/genericcache/cache.go:28 +0x124
  learngo-pockets/genericcache_test.TestCache_Parallel.func1()
      learngo-pockets/genericcache/cache_test.go:73 +0x97
  learngo-pockets/genericcache_test.TestCache_Parallel.func2()
      learngo-pockets/genericcache/cache_test.go:74 +0x47
```

As you can see, Go was able to detect that we were writing at the same index twice, at the same time. This constitutes a data race, and this is what would make our cache not thread-safe.

You might notice that we've eluded describing the `--trimpath` flag here. The default behaviour of Go's test framework is to output the absolute path of failing tests (and the stack that leads there). Using `-trimpath`, we tell Go to only output the path from the root of our module. This makes the output clearer when sharing it.

We can now answer our collegue's remark: our implementation of the cache is not thread-safe. This is a severe flaw in design and security. We need to work on it.

7.3.2 Add a mutex

Go proverb

Channels orchestrate; mutexes serialize.

When it comes to restricting synchronised access to a resource, computer scientists - namely, Edsger Dijkstra - introduced the notion of semaphores. A semaphore is a counter that keeps track of a number of threads accessing a given resource. Semaphores are used to allow a specific number of threads to simultaneously a variable, a connection, a socket… We can push the semaphore to the extreme and allow up to exactly one thread to access a resource. A semaphore that ensures **MUT**ual **EX**clusion to a resource is suitably named "mutex". Go allows us to use mutexes through the type `Mutex`, defined in the standard library's `sync` package.

The sync.Mutex type

Here’s how to declare a simple mutex in Go:

```
var mu sync.Mutex
```

Before we dig into how to use our mutex, it is important to remember that a mutex is always used to protect the access to a resource. Place it in your code as close as possible to the resource the mutex protects.

Let’s have a look at `go doc sync.Mutex`. We see there that a `Mutex` exposes `Lock()`, `Unlock()`, and `TryLock()`. While the first two methods are quite explicit, one might be tempted to use `TryLock`. A quick glance at its documentation, through `go doc sync.Mutex.TryLock` tells us that if we resort to using this method, we have a deeper problem.

We can lock our mutex when we want a piece of code to have exclusive access to the resource, and unlock it afterwards. We’re almost ready to use our mutex - there is a final line of the documentation that is worth engraving: “a `Mutex` must not be copied after first use”. Copying a mutex by passing it as a parameter to a function is a mistake that usually leads to unexpected behaviours when locking or unlocking the mutex. They and the structures containing them need to be passed as pointers.

The zero value of a `Mutex` is to be unlocked. Using mutexes requires paying special attention to the structure of the code. Indeed, it is a “common” source of error to forget to unlock a mutex because the function exits early. As a best practice, we recommend always `defer`-ring the `Unlock()` call right after calling `Lock()`. There will be a few cases when this isn’t exactly what you need, but these will be the exceptions to the general rule of deferring unlocking.

Let’s return to the code and add a mutex to our cache. First, we’ll add a mutex next to the resource we want to protect - the `data` map, within the `Cache` structure.

Listing 7.11 cache.go : cache with a mutex

```
type Cache[K comparable, V any] struct {
mu   sync.Mutex
    data map[K]V
    }
```

Each method on the `Cache` type will ensure only a single goroutine can enter it at a time by having the same two lines:

```
c.mu.Lock()
defer c.mu.Unlock()
```

We can now re-run `go test -race .`: we should no longer see any data race detected. The mutex seems to have done the job. However, using mutexes isn't free - there is a cost in time execution every time we lock (and unlock). For this reason, it is worth checking we weren't overzealous in our usage of mutexes. In our example, while we are ensuring that no two goroutines update the contents of the cache "simultaneously", we're also preventing two goroutines from reading from our cache, which is not a conflicting operation.

The sync.RWMutex type

In order to address this specific need, the standard library exposes another mutex: the `RWMutex`, a read-write mutex, also in the `sync` package. This mutex is very similar to the basic `Mutex` - it also exposes `Lock()` and `Unlock()` - but on top of that, it also has a `RLock()` and a `RUnlock()` methods that are used when we only want to use the mutex to read data. Any number of goroutines can call `RLock()` without blocking each other, but as soon as `Lock()` is called, no goroutine can access the resource - neither for reading, nor for writing.

We can update our code - the mutex in the cache should be a `RWMutex`. The `Read` method should only call `RLock` and `RUnlock`, as it doesn't modify the contents of the cache. `Upsert` and `Delete` will still need a regular `Lock` and `Unlock` call. As a general rule, `sync.Mutex` is the way to go, and `sync.RWMutex` should only be considered if you are facing performance issues - and even then, caution should be the rule. Because of its richer interface, accidentally calling `RLock` instead of `Lock` will have a disastrous impact on the code - and the compiler won't tell you. Don't blindly believe that `RWMutex` is faster than `Mutex` - instead, benchmark it for your specific usecase, and use the appropriate one.

Running `go test -race .` once more should give us confidence we didn't add a data race. We now have a fully operational, thread-safe, generic cache!

7.4 Possible improvements

Even though our cache looks perfect, there are a couple of optimisations we could add. The first one we present here represents the idea that no value is frozen in time forever. After all, "the last person to have walked on the Moon" could very much not be Gene Cernan in the near future. Sometimes, it's best to ignore outdated values, and the cache should tell us whether a value has reached its expiration date. The second optimisation we'll present is about handling the cache's memory footprint. Indeed, if the user doesn't call `Delete()`, the cache will only grow, storing more and more items. Not having a limit on the size of the cache is dangerous - it could end up using too much memory, causing some slowing down in the application.

7.4.1 Add TTL

As we mentioned previously, values retrieved in the past can become outdated. One way of ensuring our values are never too ancient is to give each one of them a "best before" date. Once this timestamp has passed, we shouldn't trust the value any longer. In computer science, this timestamp is called a "time to live" - or TTL. In order to implement it in our cache, we need to attach a "best before" to each of our values.

Add the timestamp

Thanks to generics, we can add an expiration date to any value by defining a new type - an entry with timeout:

```
type entryWithTimeout[V any] struct {
    value   V
    expires time.Time // After that time, the value is useless.
}
```

Our cache is in charge of setting the `expires` value when we upsert an item in our cache. We'll provide a TTL to our cache as a field. This TTL could be a hardcoded parameter of the cache - but this isn't very user-friendly. When writing a library, you don't know what use will be made of it. Our cache can be used for varying values such as "most trending posts on social media", or for stable values such as the list of capitals of countries of the world. It's best to expose this TTL as a mandatory parameter of our `New` function.

Listing 7.12 cache.go : creating a cache with a TTL

```
type Cache[K comparable, V any] struct {
    ttl  time.Duration

    mu   sync.Mutex
    data map[K]entryWithTimeout[V]
}

func New[K comparable, V any](ttl time.Duration) Cache[K, V] {
    return Cache[K, V]{
        ttl:        ttl,
        data:       make(map[K]entryWithTimeout[V]),
    }
}
```

Update the methods

Let's have a thought about what will happen in our `Read()`, `Upsert()`, and `Delete()` methods. The easiest one is `Delete`: there's nothing to change there. A key can be removed, regardless of whether the associated value has reached its expiration date. Then, let's have a look at `Upsert`. We used to either insert the data, or override the value. Well, things aren't very different now - upon insertion, we'll add the data with the correct `expire` value, and upon updating, we'll not only override the value, but also its `expire` field.

Listing 7.13 cache.go : Upsert with a TTL

```
func (c *Cache[K, V]) Upsert(key K, value V) {
    c.mu.Lock()
    defer c.mu.Unlock()

    c.data[key] = entryWithTimeout[V]{
        value:   value,
        expires: time.Now().Add(c.ttl), #A
    }
}
```

Finally, we’re left with the trickier `Read()` method. This is where we’ll check whether an entry is no longer valid. We need to add a second check on top of the present one that verifies our cache has a value for the requested key. If the value is still valid, we can return it. But what if it’s not? In this case, in our implementation, we decided that the user doesn’t need to know why the value isn’t in the cache - after all, what matters is that it couldn’t be found.

Listing 7.14 cache.go: Read with a TTL

```
func (c *Cache[K, V]) Read(key K) (V, bool) {
    c.mu.Lock()
    defer c.mu.Unlock()

    var zeroV V #A

    e, ok := c.data[key]

    switch {
    case !ok:
        return zeroV, false
    case e.expires.Before(time.Now()):
        // The value has expired.
        delete(c.data, key)
        return zeroV, false
    default:
        return e.value, true
    }
}
```

By implementation, our `Read()` method now has to alter the contents of the map. As a result, we can’t rely on a `RWMutex` as we did in section 3. Instead, we use a regular `sync.Mutex`. This will have a small impact on performance - two `Read()` can no longer be executed simultaneously.

In the implementation of our `Read()` method, we start by defining a non-initialised value of type `V`. This is very common in generic functions that return a constraint - indeed, we can’t `return V{}`, as this would require `V` to have a concrete type representation at runtime.

Now that we've written the code, we should test it. Our scenario, here, is to create a cache with a rather small TTL, to insert an item, and then to wait more than our cache's TTL. Checking immediately if the item is available shouldn't return an error, but checking after a while should.

Listing 7.15 cache_test.go: Testing Read with TTL

```
func TestCache_TTL(t *testing.T) {
    t.Parallel()

    c := cache.New[string, string](5, time.Millisecond*100)
    c.Upsert("Norwegian", "Blue")

    // Check the item is there.
    got, found := c.Read("Norwegian")
    assert.True(t, found)
    assert.Equal(t, "Blue", got)

    time.Sleep(time.Millisecond * 200)

    got, found = c.Read("Norwegian")

    assert.False(t, found)
    assert.Equal(t, "", got)
}
```

We start our test with a call to `t.Parallel()`. Indeed, we're fine running this test along with others. We recommend using this in every "light" test. If a test requires a lot of resources - CPU, RAM, disk, network, then you might not want to have it run with others. In our case, we're absolutely fine.

Schrödinger's conundrum

You might have noticed that we discard expired items only when we try to access them via `Read()`. This means that items could expire way before we look at them, unbeknownst to us. The side effect is that our cache might be using chunks of memory for useless data. How do we deal with that?

Well, bluntly put, we decided not to. If we were to implement "something that regularly checks each item and gets rid of it if we know it's no longer usable", we'd basically be writing a garbage collector for our cache. We'd need to start a goroutine in `New()`, and that goroutine’s only task would be to endlessly scan the map and delete items that have reached their TTL. Instead of implementing this, we have decided to address a slightly related issue - controlling the size of our cache.

7.4.2 Add a maximum number of items in the cache

In order to prevent too many items from being added to the cache, we will set a limit to our cache’s size. This will be a property of the cache, an unexposed unsigned integer keeping track of how many items were added and removed.

Architectural decisions

We’ll need to make a decision when we try to add a new value into the cache and the maximum number of items is reached.

In our implementation, we decided to allow this operation - and discard another entry. There are lots of interesting choices to determine which entry to remove from the map in this case - eligible candidates could be “the oldest entry in the cache”, “the most recent entry in the cache”, “the least read entry in the cache”, “the entry that hasn’t been read for the longest time”, etc. Each of these implementations requires storing extra information in our cache. The choice of which one to use is highly dependent on the information stored in the cache. Here, we’ll decide to remove the entry that is the oldest in the cache, and we consider that overriding a value should reset its timestamp - as it does for the TTL.

For this, we need to keep track of the order in which items were inserted. Let’s look at which options Go offers to implement this:

- Maybe use a channel? This is the most intuitive implementation of a “first in, first out” list in Go. When we `Upsert` a new entry, we register the key in the channel. The first item in the channel would be the oldest. However, this wouldn’t work, because we don’t cover the cases where the user would call `Delete` or when `Upsert`. Indeed, in these two cases, we’d have to move an item from “somewhere” in our channel to its tail. Since channels in Go don’t support suppression of an element, this implementation is not good enough.

- Maybe we could use a slice? After all, slices can handle suppression of some element in the middle of the slice without too much effort. When we `Upsert` an element, if it was present, we add it to the slice, and when we want to override it with another `Upsert`, we can move it to the end, and finally `Delete` removes it from the slice.
- Other options are available - using a binary search tree, for instance - but we'll go with the slice, as it covers most of our needs.

We already know, since we want our cache to hold up to a maximum number of items, that we can give an upper bound to our slice's capacity. We'll initialise it with the following syntax:

```
chronologicalKeys := make([]K, 0, maxSize)
```

In order to check whether we've reached the maximum number of items, we can either store the `maxSize` value as a field of our cache, or we can use the `cap` built-in function, on the `chronologicalKeys` slice. In this book, we decided to go with the former for the sake of clarity, but this adds the cost of storing this value in our structure.

```
if len(c.data) == maxSize
if len(c.data) == cap(c.chronologicalKeys)
```

This last parameter is here to tell the capacity of our slice at execution. When an element is appended to a slice, if that slice's length is equal to its capacity, the whole slice needs to be reallocated elsewhere in memory. Setting the correct capacity to our slice prevents these reallocations.

Now, just as we did for the TTL, let's have a look at the impact of having this slice in our cache for each of our exposed functions.

Implementation

`New()` should take another parameter: the maximum size of the cache. Having a default value doesn't really make sense here - a cache of 10 integers wouldn't be the same size as a cache of 10 extremely complex structures with lots of fields. The package `reflect` could help us set a maximum memory size to our cache based on the memory footprint of a single item, but this would be overkill. Instead, let’s have the user specify a size they think is good enough. Then, any memory consideration is left to them.

Listing 7.16 cache.g : New with a maximum size

```
type Cache[K comparable, V any] struct {
    ttl             time.Duration

    mu              sync.Mutex
    data            map[K]entryWithTimeout[V]

    maxSize         int
    chronologicalKeys []K
}

// New creates a new Cache with an initialised data.
func New[K comparable, V any](maxSize int, ttl time.Duration) Cac
    return Cache[K, V]{
        ttl:               ttl,
        data:              make(map[K]entryWithTimeout[V]),
        maxSize:           maxSize,
        chronologicalKeys: make([]K, 0, maxSize),
    }
}
```

Next, we notice that adding an entry to our cache will no longer be as simple as adding a key-value pair to a map: indeed, we now need to update the `chronologicalKeys` slice - either by adding, removing, or moving one of its elements, everytime we update the map - respectively by inserting, deleting, or updating one item.

As a result, we refactor our code to avoid duplicating logic. We need a small function that adds a key-value pair to our cache, and one that removes a key from it. Both functions should be in charge of updating both our map and our slice. Let’s start with these - and it’s also a good opportunity to use a feature added in Go 1.21 - the `slices` package. This package is a helper for most common operations on slices. Here, we’ll use it to delete all items from the slice that have a specific value, with its `DeleteFunc` function. This function returns a slice that has dropped all items that returned true in the provided callback (it doesn't update the slice).

Listing 7.17 cache.go: Utility functions to replace the map calls

```
// addKeyValue inserts a key and its value into the cache.
func (c *Cache[K, V]) addKeyValue(key K, value V) {
    c.data[key] = entryWithTimeout[V]{
        value:   value,
        expires: time.Now().Add(c.ttl),
    }
    c.chronologicalKeys = append(c.chronologicalKeys, key)
}

// deleteKeyValue removes a key and its associated value from the
func (c *Cache[K, V]) deleteKeyValue(key K) {
    c.chronologicalKeys = slices.DeleteFunc(c.chronologicalKeys,
    delete(c.data, key)
}
```

Now that we've got these helping functions, we can update the code in `Read()` first - all we have to do is update how we remove an entry when it had reached its TTL:

Listing 7.18 cache.go: Read with the new helper functions

```
func (c *Cache[K, V]) Read(key K) (V, bool) {
    ...
    case e.expires.Before(time.Now()):
        // The value has expired.
        c.deleteKeyValue(key) #A
        return zeroV, false
    ...
```

Listing 7.19 cache.go: Delete with the new helper functions

```
func (c *Cache[K, V]) Delete(key K) {
    // Lock the deletion on the map
    c.mu.Lock()
    defer c.mu.Unlock()

    c.deleteKeyValue(key)
}
```

And finally, we have to update the `Upsert()` function. This one is slightly tricker, as, this time, this is where the core of the feature we want to implement resides - we want to limit the number of items that are stored in our cache at a given time. Since this number only grows when we upsert items, it makes sense that this function will be the most affected one. Let’s have a look at all possibilities when the user calls `Upsert()`:

- The cache already has a value for that key: in this case, we want to reset the whole entry with the new value - and the new TTL. We need to also update the position of the key in our chronological slice. We can achieve this by deleting the old pair and adding the new one.
- The cache doesn’t have a value for this key: in this case, if we’ve not reached the maximum capacity of our cache, then we can simply insert the new pair. However, if we have reached the maximum capacity, we need to clear some space for the new entry - this means discarding the item that has been there for the longest. This item is at the beginning of our slice of keys.

Now that we know how our method should behave, let’s implement it:

Listing 7.20 cache.go: Upsert with the new helper functions

```
func (c *Cache[K, V]) Upsert(key K, value V) {
    c.mu.Lock()
    defer c.mu.Unlock()

    _, alreadyPresent := c.data[key]
    switch {
    case alreadyPresent:
        c.deleteKeyValue(key) #A
    case len(c.data) == c.maxSize: #B
        c.deleteKeyValue(c.chronologicalKeys[0])
    }

    c.addKeyValue(key, value) #C
}
```

There is one last chance for optimisation here. When we need to replace an existing entry, but the cache is at maximum capacity, we know we don’t need to discard the oldest entry - we can discard the value we’re about to replace to create enough room for the new entry. Go’s `switch/case` statement has a very specific behaviour that we used in our implementation: when several `case` statements are valid, only the first eligible one will be executed. That’s an implicit rule that most people know without knowing it - it also applies to `default`: if we enter a `case` statement, we won’t execute the `default` block. We used that behaviour here to delete only the pair we need to update. Should you ever need to enter more than one case statement, you could consider using the keyword `fallthrough`. But, in our opinion, it would probably be clearer to write a list of `if` statements, in that case.

Test it

Our cache is no longer a plain map. We;ve added logic with our list of items by age, and this new logic is invisible to the end-user. As a result, it’s worth adding a few internal tests to just make sure we’re doing everything right.

Finally, let’s think of an end-user test scenario for our new feature. We can validate it by adding items to our cache beyond its limit. It would also make sense to check that updated items have their insertion timestamp updated. For this, we’ll create a cache with a small maximum capacity, insert items to the brim, upsert the oldest, and then insert a new key. We should then be able to retrieve the upserted value, and we should no longer be able to retrieve the second value we added in our cache.

Listing 7.21 cache_test.go: Test the maximum capacity of the Cache

```
// TestCache_MaxSize tests the maximum capacity feature of a cach
// It checks that update items are properly requeued as "new" ite
// and that we make room by removing the most ancient item for th
func TestCache_MaxSize(t *testing.T) {
    t.Parallel() #A

    // Give it a TTL long enough to survive this test
    c := cache.New[int, int](3, time.Minute) #B

    c.Upsert(1, 1)
    c.Upsert(2, 2)
    c.Upsert(3, 3)

    got, found := c.Read(1)
    assert.True(t, found)
    assert.Equal(t, 1, got)

    // Update 1, which will no longer make it the oldest
    c.Upsert(1, 10)
    // Adding a fourth element will discard the oldest - 2 in thi
    c.Upsert(4, 4)

    // Trying to retrieve an element that should've been discarde
    got, found = c.Read(2)
    assert.False(t, found)
    assert.Equal(t, 0, got)
}
```

Congratulations! We’ve now written a generic library that we can share with other developers. We started with a naive implementation that covered our needs, and then we strengthened it by adding thread-safety on it. Even though there was quite a lot of theory presented in this chapter, we managed to cover practical requirements for a cache.

7.5 Common mistakes

In *100 Go mistakes and how to avoid them*, Teiva Harsanyi dedicates 20 of the hundred teachings to concurrency and parallelism. We highly recommend the book to dive deeper into Go. In the meantime, here is our own, shorter, list of common mistakes to avoid.

When to use channels in a concurrency situation

Channels are something very specific to Go, which means developers new to the language do not get them as easily as the rest of the language. They are, arguably, the one feature that requires a learning curve and some practice in the entire language.

Because of this, because they can be tricky at first, do not use them if you don’t need them. They might seem shiny, your situation might look like a good place to use them, but think twice. Channels should be used when you need to communicate in or out of a goroutine.

Concurrency impact of a workload type

Given a concurrency situation, you first need to determine the type of workload: it will not lead to the same solution. Loads can be CPU, memory or IO-bound. Running a merge sort algorithm is typically high on CPU, whereas making REST API calls is a lot of input/output.

Take for example a program that counts the number of lines in a bunch of files. Opening and closing files would be the bottleneck here: it is IO-bound. The number of goroutines that can work in parallel will be determined by the operating system’s limit, or the rules of your server if the files are remote - you don’t want to crash it or get banned by hitting it too much.

On the other hand, if you are encoding a video on a single machine, a task that is typically high on CPU, then you need to look at the architecture of your machine. The `GOMAXPROCS` environment variable is an interesting hint. Its default value is the number of cores of your CPU. It represents the maximum number of goroutines that could actually run simultaneously. Any extra goroutine will have to share a CPU with existing goroutines. It can (too often) happen that parallelising the work actually makes things worse, because you have already hit the max load of your CPU.

The size of a buffered channels can be a good thing to benchmark in your performance tests, which should be executed on a machine with a similar architecture as that where your code will be executed.

Finish your routines

Goroutines are easy to start, but don’t let them leak. As we have seen, a program should only exit when all of its child goroutines are finished. Once you’re done writing to your goroutines, close them so that their readers know when to stop listening.

Explore the sync package for tools to make your life easier. Most of the types there should not be copied, though, be careful.

7.6 Summary

- A cache is a key-value storage facility. They are commonly used when getting the value associated with a key is costly (timewise or in the amount of resources) and when getting a previously retrieved ro computed value is OK.

- When writing a library, one question that should always be raised is “is this library thread-safe ?”. The answer is either “yes, and I know why”, or “No, it’s not”. There is no middle ground - the worst case scenario is usually also the most dangerous.
- Go implements genericity with, well, generics. Structures, variables, and functions can be declared using generics.
- A constraint is a requirement for a generic type. Common constraints are `any` (quite explicit), `comparable`, which allows the use of == between two values, or `golang.org/x/exp/constraints.Ordered`., which allows the use of >, <, >=, <= between two values.
- Constraints are passed in square brackets after the name of the generic entity:
  - `type fieldWithName[T any] struct { value T, name string }`
  - `var hashIndex[T myConstraint] map[uint]T`
  - `func sortSlice[T constraints.Ordered](t []T) ([]T, error)`
- Constraints can be omitted in the declaration of functions when the compiler can infer which type it should be using: `sortSlice([]int{1,4,3,2})`.
- Go uses goroutines for concurrency. Goroutines are similar to what other languages usually call threads, except that they’re not. Threads live at an OS-level, while goroutines live farther from your silicon - they exist in the runtime environment of Go.
- In Go, goroutines are launched with the keyword `go`: `go do()` runs the `do` function in a new goroutine.
- Channels to communicate data between different goroutines. When passing a channel to a function, make it explicit in the signature that the function will either read from the channel, with the syntax `func f(c <-chan string)`, or that it will write to the channel, with the syntax `func f(c chan <- string)`. If you need to both read from and write to a channel in a single function, there is probably a design flaw.
- Two types are commonly used when we need to synchronise goroutines: `sync.WaitGroup` and `errgroup.Group`. When using `sync.WaitGroup`, start by calling `Add(n)` with the number of goroutines that will be executed. Each goroutine is in charge of calling `Done`. The

synchronisation is achieved by calling `Wait()`. When using `errgroup.Group`, start by setting a maximum number of parallel goroutines with `SetLimit(n)`. Launch each goroutine with a call to `Go(...)`. The synchronisation is achieved with a call to `Wait()`, which returns an error if one of the goroutines returns an error.

- The choice of `sync.WaitGroup` or `errgroup.Group` is often driven by the necessity to check for errors in at least one goroutine: use `sync.WaitGroup` when errors don't need a specific treatment. Use `errgroup.Group` if you want to handle errors.
- Mutexes are used whenever we want to protect a variable from concurrent writing - or reading. In Go, we can create a mutex variable by using the `sync.Mutex` type: `var myMutex sync.Mutex`. A mutex should never be exposed - instead, use it in exposed functions. A mutex should always be written close to the variable or field it protects.
- You can call `mu.Lock()` on a mutex, but we highly recommend immediately following this with `defer mu.Unlock()`. Debugging locked mutexes is a pain.
- Use `t.Parallel()` in your tests to let the framework know that a test is not blocking for the execution of other tests.
- Use the `-race` flag when testing to try and detect data races - but remember, the failure to detect a data race doesn't mean there are no data races.
- Use the `-trimpath` flag when testing to only output paths relatively to the root of the module.
- A `switch/case` statement will only execute the first valid `case` - any subsequent case will be ignored.

8 Gordle as a service

This chapter covers

- Creating and running an HTTP server that listens to messages on a given port
- Listening to endpoints with different verbs (GET, POST)
- Building a response with a status code
- Decoding different sources of data: path and query parameters, bodies and headers
- Testing using regular expressions

In 1962, J. C. R. Licklider mentioned the possibility of having computers communicate one with another over a network. Since then, computer science has travelled a long way, first through this “intergalactic computer network”, then ARPANet, and, today, the Internet. Networks are now used on a daily basis - when you pick up your phone to check the weather, the news, or even the time. The possibility of using a server from a remote location was paramount when, in 2020, the whole planet went into lockdown.

A server, in the end, is really just a machine that listens to communications on a given set of ports and is able to answer messages that it receives. In this chapter, we’ll implement such a server. In order to make things interesting, our server will have an application public interface - usually referred to as APIs - that will allow a user to play a game of Gordle. This will allow us to focus the implementation of this chapter on the server side rather than on the algorithmic aspect, which has already been covered in Chapter 5.

In the first part of this chapter, we’ll create the REST API, and test it with simple tools such as an internet browser or a command. In the second part, we’ll integrate the game of Gordle - which will require a few updates to comply with how we want to use it in the server. Finally, we’ll mention a few security tips.

[[Disclaimer]] Further steps, such as containerization and deployment, are not the topic of this chapter.

Requirements:

Play Gordle on a web service. Run a service that exposes at least the following endpoints:

- `NewGame` creates a session and returns a gamer ID. This will be used for counting the number of attempts.
- `GetStatus` returns, well, a game's status: how many guesses are still allowed, what were the previous hints, etc.
- `Guess` takes a word as a parameter and returns the feedback and the status of the game.

We will want to include, in a second step, some tracking of the players' sessions. Many online resources use some kind of user identifier - most of the time, authentication. We'll see how we can convey this information here.

Non-requirements: monitoring, logging strategy, scalability (yet)

8.1 Empty shell for the new service

A web service can be thought of as a daemon - it is always running, waiting for queries that are sent to a port of the computer hosting it. In this chapter, we will start our development from the outside in: we will first create a service that listens to a port but does nothing, then add empty endpoints, one per feature, and finally, we will add their logic. This strategy is best when some other team members, e.g. front-end developers or other teams altogether, are waiting for your work. It is possible to return a static mocked response that other people can use while you develop and where they can give you feedback. Having a service that returns something, even when it's constant or irrelevant, is often enough to help other people design, develop, test, or deploy their solutions.

Before we begin writing a few lines of code, it's important to introduce some vocabulary and understand a few theoretical notions.

8.1.1 Server, service, web service, endpoints, and HTTP handlers

A server can be thought of as a computer, running somewhere. We usually keep servers running, as they host services: applications that expose an API (Application Programming Interface). Usually, we want services to be permanently running - a stopped service is of no use. A web service is a particular kind of service that makes use of the web protocol to receive and send communications with the outer world.

It is important to keep in mind, for later steps of this chapter, that a service isn’t supposed to end its execution, in other words, it is running until the user decides to stop it.

Endpoints are the access points for the exterior into a service. For web services, endpoints are mapped to specific URLs as we’ll soon see. A web service, behind the scenes, will use an HTTP handler to deal with requests. HTTP handlers receive requests and generate responses.

You should be aware that the terms webservice and endpoint are used in different contexts with slightly different definitions, but we will use the one above in this chapter.

8.1.2 Let’s code

Let’s start by creating the module for our service. Keeping in mind that we will be running Gordle in an HTTP server, we can come up with a relevant name. Remember that if you are pushing your code to a code repository, it is always better to declare the full path of your repository as a module.

```
$ go mod init learngo/httpgordle
```

Once we’ve created the `go.mod` file, we can start writing the `main` function. It will be responsible for creating a server and running it. For this, we’ll need some help from the `net/http` package that we’ve already seen in Chapter 6. Its documentation is quite long, but if we read only the first lines of `go doc net/http`, we see that *Package http provides HTTP client and server* *implementations*. For now, we are only interested in the server side.

A server listens to a specific port, so find any free port on your host machine. The default port for HTTP is 80, but for development purposes, we prefer to use another, such as 8000 or 8080, as 80 will very probably be used on your machine by something else.

A function in the `net/http` package seems to achieve exactly our need - `ListenAndServe`.

Listing 8.1 main.go: Create the server

```
package main

import "net/http"

func main() {
// Start the server.
err := http.ListenAndServe(":8080", nil) #A
if err != nil {
    panic(err) #B
    }
}
```

Right, we hope this wasn't too frightening. We've given `ListenAndServe` two parameters. The first one is the address we want our web service to be listening to - `localhost:8080` (the `localhost` can be omitted) is a popular choice - and the second parameter is, well, the topic of the next pages: it's the handler that can deal with requests. If you are on Windows, consider using another port than 8080 to avoid the Windows Firewall popup. For now, we can keep it nil, but that's where the logic will be implemented. We said panicking was OK in the body of the `main` function — actually, yes and no. `panic` dumps the whole stack trace, which could be confusing for users. It would be more polite to dump the error and exit with a proper code like the snippet below, the downside being not having the guarantee that all the `defer` calls have been executed.

```
if err != nil {
fmt.Fprintln(os.Stderr, err)
os.Exit(1)
}
```

Let's run it!

```
$ go run .
```

You might notice that the execution hangs. That's because the `ListenAndServe` function never returns - after all, that's exactly how we want it to behave: the service is running!

How do we test it manually? There are many tools that can be used to test an HTTP server, we'll mention four of them:

- An Internet browser: they are designed to send HTTP requests. It will be a bit tricky to set some parameters such as the body of the request, its headers, or the verb we want to use, but it should do the trick for this first implementation.
- Postman: this tool has a graphical user interface that allows for finer usage than a Web browser when it comes to sending formatted messages over the network.
- `curl`: this is a command-line tool that exposes everything we want to use. This is our preferred option: not only is it simpler to share the execution command line in a book, but also using command-line tools rather than clickable interfaces will make testing a lot more automatable. `curl` is shipped with every version of Linux or Mac OS, and with Windows 10 and above.
- Write a program in Go that creates a client to speak with our server. We'll cover this section in a later chapter.

The `nil` handler we provided can't really do much, but still, we can see it in action. If you open your favourite web browser and enter the URL `localhost:8080`, you'll see a response from the default HTTP handler - a 404 message.

Should you want to use `curl`, here is the same request as a command line: it will also return a 404 message. We'll make more extensive use of `curl` in our next tests.

```
$ curl http://localhost:8080
```

We have successfully implemented our first HTTP server. Before we move on to the next section, we need to kill our server. In the shell where we executed `go run .`, let’s press control and C simultaneously. This sends an interrupt signal to the current process, which terminates it. There are other ways of terminating a running program, either via your computer’s task monitor or by using the `kill` command, if you know the process ID of the program you want to terminate. Once this is done, we can commit our work before moving on to the next part - getting rid of this `nil` handler.

8.2 Adding endpoints

We’re implementing a web service that allows people to play a game of Gordle. Let’s have a look again at the requirements: we need to be able to create a game, to play a guess, and to retrieve the status of a game. We can immediately see our service will be dealing with “game” entities - creating them, using them, and displaying them. We’ll also need to be able to access these games - either to play a guess or display their status. For this, we’ll need some way of identifying a game and a way to store it. Storing will come later, as we work from the outside in.

An endpoint, on an HTTP server, is a pair of a path and a HTTP method. The path should reflect which resource is being used. In our case, we will be dealing with games, which makes the path `/games` a good start. When we need to identify a single game, we can use `/games/{game_id}`.

An **HTTP method** (or verb) describes the action we want to execute as we call an address. There are several methods defined, but we’ll focus on the following:

- GET: is used when accessing a resource;
- POST: is used when creating a resource or asking for data to be processed;
- PUT: is used to update a resource;
- DELETE: is used to delete a resource.

Some of these endpoints - GET, PUT and DELETE, in this list - should be idempotent - that is, several consecutive calls should all return the same response, and should all leave the resources in the same state as calling them just once. If your API is not idempotent, you need to explicitly tell your users in the documentation. GET, PUT and DELETE should simply not be used for non-idempotent endpoints - use POST instead.

We’re now ready to implement our first endpoint.

8.2.1 Create a new game

Creating a game is the first thing a player will do. Let’s start by having a look at what goals we need to achieve. We don’t really need any input to create a game of Gordle. If you remember Chapter 5, we launched a game with `go run main.go`. As we’re adding a new endpoint, we need a pair of a path and an HTTP method. The resources we’ll want to use are the games - the `/games` path seems perfect.

Which HTTP method should we use? In this case, since we are creating a game, we should use a `POST`. It could happen that you already know the identifier of the resource to create. For instance, if we were dealing with books, we could use the ISBN to create a book resource with the method `PUT` on the following address: `/books/9781633438804`.

For Gordle, we only need to create an empty game. However, in order to keep track of it, it is paramount that we return the game’s identifier - which will be used in the `GetStatus` and `Guess` endpoints. An identifier can take the form of a series of digits - a phone number is an example of a digit-only identifier - or characters - for example, a registration plate is a car’s identifier. There are some good libraries that provide unique identifiers - for example `github.com/google/uuid`. We’ll stick to random integers, as we’ve already covered this topic in Chapter 5 when we needed to get a random word from a list.

We’ve now defined the API for this endpoint. We know which path we want to use, which verb should be associated with it, and what the response should be - an identifier. The documentation would start like this:

```
POST /games - creates a new game and returns its ID.
```

Back to the code!

A few words about project organisation

There is no official rule on how to organise files within a module, nor modules within a project. However, there are common practices that are worth mentioning. The first important point is that a few folder names have a specific behaviour in Go.

- testdata:
  We've already mentioned in Chapter 3 that directories named `testdata` won't be examined by the go tool - code inside it won't be compiled by `go build` or `go run`, tests written inside won't be executed by `go test`, and documentation won't be visible through `go doc`.
- internal:
  It's now time to introduce another special name for a directory: `internal`. An `internal` directory can contain code for the current module to use, but this code won't be visible to other modules. For instance, the module `golang.org/x/text` has an `internal` package, where the type `InheritanceMatcher` is defined. However, even though this type is exposed (because of its capital letter), we can't create a variable of this type in our module: the scope of types and functions defined in a directory named `internal` - or a subdirectory of an `internal` directory - is limited to the current module (in our example, the `golang.org/x/text` module). An `internal` directory is a good place to put code you don't want other people to use. In the case of a service, most of the code will reside there.
- vendor:
  We'll mention this one for historical reasons - there is a third directory name to know about: `vendor`. We won't go through the whole history of the language, but earlier versions of Go used to have "versioned" dependencies - copies local to each module. These copies would be placed inside a `vendor` directory - and it was a good idea to always ignore the contents of that directory in your favourite versioning tool. It's best to simply never name a directory `vendor`, for compatibility matters. If you really must, use `vendors` instead.
- pkg:
  You might encounter packages located in a `pkg` package, at the root of the module : `module/pkg/my_package`. In `pkg`, you can expect to find

libraries that could be used outside of your project. We do not encourage the use of `pkg`, since it is not a Go standard. It is rather a historical artefact or a golang-standards/project-layout, which is not the official standard from the Go team.

These were strict rules, and we can add some suggestions that you are free to follow. We like to expose the API of a service in an `api` package - a directory at the root of the module.

File organisation of the service

We are now prepared to organise our code and can create an `api` directory at the root of our module, and an `internal` directory into which we'll write all sorts of things, including our HTTP handlers in a subdirectory. Indeed, how we implement an endpoint won't be of any use to external developers, and that's why we might as well hide this within an `internal/handlers` directory.

We choose to create a package for each handler. In our case, a simple package for all of them would be perfectly fine as well, however, by having multiple packages, we can show how we would structure a larger project -- without actually writing a large project. Depending on the situation, you can sometimes do everything in the same package (albeit in different files for clarity) or spread the logic across different packages. Here we chose to keep the logic (validations, calling the storage, etc.) inside the handlers (who are responsible for everything related to the HTTP API), which means we prefer to have a package per endpoint. As usual, think about how each package will scale and grow when you add functionalities as you make this type of decision. Finally, it's important to take into account the notion of coupling. Some structures and functions are coupled by nature and will need to be updated together. The more coupled pieces of code are spread into different packages, the more difficult it is to maintain code quality. As usual, it is a balance to find, and later updates will provide opportunities to refactor the organisation if it no longer fits your needs.

The HTTP API of our service can be exposed in an `http.go` file, while the initial handler for a new game will be in a `newgame/handler.go` file. We'll bind the API to the handler in the router.go file.

Here’s our file organisation at this point:

```
.
├── go.mod
├── internal
│   ├── api
│   │   ├── doc.go
│   │   └── http.go
│   └── handlers
│       ├── doc.go
│       ├── newgame
│       │   └── handler.go
│       └── router.go
└── main.go
```

Defining the REST API

REST (REpresentational State Transfer) is a set of conventions that help define an API on an HTTP server. It defines collections of resources and ways to interact with them.

Let’s start with the `http.go` file. It should contain everything that we need to expose to allow someone else to use the `NewGame` endpoint that we’ll next implement. By everything, we mean which URL should be used and which method. If there is anything more, such as parameters to the query, we include them in this file. Let’s create the `http.go` file in the package `internal/api`.

Listing 8.2 api/http.go: Define what is needed for NewGame endpoint

```
package api

import "net/http"

const (
    NewGameRoute  = "/games"
    NewGameMethod = http.MethodPost #A
)
```

We have the needed constant for the `NewGame` endpoint that we’re about to implement.

What should they expect in return? Sometimes creation endpoints only return an ID. Here we would like to be more verbose and return the full game that we created: the client of our Gordle game needs to know the number of characters in the secret word and the maximum number of attempts allowed. As we’re defining what a Game is in the API, we should think of every field that we want to include. We can also tell the status of the game to let them know whether they can keep playing and whether they won or lost already. Finally, having a list of the previous attempts will help in the display.

Let’s define the shape of this JSON game.

```
{
  "id": "1225482481867118141",
  "attempts_left": 4,
  "word_length": 5,
  "status": "Playing",
  "guesses": [
    {"word":"slice","feedback":""}
  ],
}
```

This translates easily into a Go struct, with JSON tags as seen in previous chapters.

Listing 8.3 internal/api/http.go: Define the API structure for a game

```
package api

// ...

// GameResponse contains the information about a game.
type GameResponse struct {
    ID string `json:"id"`
    AttemptsLeft byte `json:"attempts_left"`
    Guesses []Guess `json:"guesses"` #A
    WordLength byte `json:"word_length"`
    Solution string `json:"solution,omitempty"` #B
    Status string `json:"status"`
}

// Guess is a pair of a word (submitted by the player) and its fe
```

```
type Guess struct {
    Word     string `json:"word"`
    Feedback string `json:"feedback"`
}
```

Note that all types are primary types, we are not imposing any strong typing to our consumers.

We chose to express the feedback as a string. More on this when we start filling it up.

We now have the structure, it is officially published to consumers, time for the server to actually expose the endpoint.

HTTP multiplexer, handle, and handler

If you remember the first section, we provided a `nil` handler to the `ListenAndServe` function. Let’s have a closer look at this function’s second parameter - it’s a `http.Handler`. This type is declared as follows:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

There are several ways of implementing a `Handler` that can be passed to `ListenAndServe`. For instance, we could define a structure and have it implement the interface:

```
type newGameHandler struct {}

func (h *newGameHandler) ServeHTTP(w *http.ResponseWriter, req *h
    ...
}
```

And in the main function:

```
err := http.ListenAndServe(":8080”, newGameHandler{})
```

This would do the trick. However, there is a minor issue here: creating a handler this way only allows for one endpoint to be defined. As we know we want to implement several endpoints - at least 3.

Have a look at another type provided by the `net/http` package: the `ServeMux`. A quick `go doc http.ServeMux` command shows that it is a “request multiplexer”. This means that a `ServeMux` is in charge of routing requests based on the URL they were sent to. Multiplexers - also commonly referred to as “muxes” - are the founding stone of an HTTP service.

There are two other important points to highlight with the `ServeMux`: first, it allows the registration of endpoints with the `HandleFunc` method, which is what we want to achieve. Second, `ServeMux` has a method `ServeHTTP`, with the correct signature: it implements the `Handler` interface, and we can pass a `ServeMux` to the `ListenAndServe` function!

Multiplexer

Let’s start by writing the multiplexer. We will then look at the signature of what it accepts to register an endpoint.

We need to build a `http.Handler`. We know that building it will at first only take a few lines, but as soon as the service grows, it will grow fast. This is why we write a function that builds it and returns it. All the logic of building the mux will be isolated here.

What the function does is simply to create a new instance, make it listen to our future endpoint and return it. We have not defined yet what the endpoint will look like, so let’s put a placeholder first. If you want to compile to check that everything else makes sense, `nil` is perfectly acceptable; although if you use `nil`, don’t expect a request to your service to do anything but panic.

Listing 8.4 internal/handlers/router.go: Associate a handler to a URL

```
package handlers

import (
    "net/http"

    "learngo-pockets/httpgordle/internal/api"
    "learngo-pockets/httpgordle/internal/newgame"
)

// Mux creates a multiplexer with all the endpoints for our servi
func Mux() *http.ServeMux {
    mux := http.NewServeMux()
    mux.HandleFunc(api.NewGameRoute, newgame.Handle) #A
    return mux
}
```

We can finally use this `Mux` in the main function, replacing the previous `nil` handler:

Listing 8.5 main.go: Use the new Mux() function

```
package main

import (
    "net/http"

    "learngo-pockets/httpgordle/internal/handlers"
)

func main() {
    err := http.ListenAndServe(":8080", handlers.Mux())
    if err != nil {
        panic(err)
    }
}
```

Now, what’s left is to implement that `newgame.Handle` handler that we passed in the `mux`.

Handler for New Game

`mux.HandleFunc` has the following signature:

```
func (mux *ServeMux) HandleFunc(pattern string, handler func(Resp
```

This method registers a handler - the anonymous function we pass as the second parameter - for the provided path. The benefit of this method over `http.Handle` is that we don’t have to write a new `http.Handler` - we simply have to provide the handler itself, the function in charge of dealing with the request and writing the response.

For the simplest implementation, we can simply call the `Write` function on the `ResponseWriter` and say something to the client. Let’s write the handler in the file `internal/handlers/newgame/handler.go`.

Listing 8.6 handler.go: Empty newGame handler

```
package newgame

import "net/http"

func Handle(w http.ResponseWriter, req *http.Request) {
    _, _ = w.Write([]byte("Creating a new game"))
}
```

This is the very first version. We’re not even checking errors - but we will, as we get closer to our final version.

Right, we’re getting there! We can now `go run .` this code and check how it behaves. There is in fact a final step we need to achieve before we can move on to the next section. Let’s discover it together.

If you open a browser to `localhost:8080/games` or run `curl http://localhost:8080/games` while the service is running, you should get a message letting you know that a game is being created. There is an invisible parameter that is passed by these tools - we didn’t specify which HTTP method to use, and both tools chose and sent a `GET` by default. We want our `NewGame` endpoint to only accept `POST` requests. Let’s implement this.

What happens when a request is received?

Even though it might sound evident, a server should be able to serve several clients at a time. The fact that someone is using an endpoint should not prevent others from also calling it. Behind the scenes, this means that the server must be able to not wait till a task is complete before serving a new call. In Go, this is achieved with goroutines. Goroutines are Go’s version of concurrent programming - the closest equivalent notion, in other languages, is usually called threads, coroutines, fibers or green threads.

Goroutines, however, are different from threads, and we will cover goroutines more extensively in a later chapter.

When a server receives a request, it starts a goroutine that will execute the handler. Even though this might seem wonderful and extremely handy, we will see that it comes with limitations. The last section of this chapter, covering correctness, will present two important topics - race conditions, and ensuring the server doesn’t explode when attempting to serve two requests at the same time. As seen in the previous chapter, instead of running simply `go test .` we will add the flag `-race`.

HTTP codes and header

A game of Gordle should only be created when a `POST` request is received. But what should we do when we receive the wrong method, and how do we implement this?

The answer to the first question is clear: we should reject the message. There is actually a specific HTTP code for wrong verbs, so we might as well use it: `http.StatusMethodNotAllowed` (see Table 8.1). And as to where we should make this check, the only logical place is within the `Handle` function.

HTTP response headers are set by a call to `WriteHeader` with an adequate status code. If we receive a request for anything other than what we declared in the API, we can terminate the call there and now.

Here is the implementation of the check in the `newGameHandler` function, in the file `internal/handlers/newgame/handler.go`.

Listing 8.7 handler.go: Reject requests using the wrong method

```
func Handle(w http.ResponseWriter, req *http.Request) {
    if req.Method != api.NewGameMethod {
        w.WriteHeader(http.StatusMethodNotAllowed)
        return
    }
    _, _ = w.Write([]byte("Creating a new game"))
}
```

As you can see, this is not really the same error handling as in “regular” Go. We have to adapt to the HTTP protocol, and this means communicating errors through status codes. Since there is no point in trying to process a request that was invalid, we can safely return from our handler. The error - invalid method - has been dealt with and there’s nothing else we want to do.

Let’s run our previous test of starting the service and checking the `http://localhost:8080/games` page through various tools. Depending on your browser, you could see a 405 error. However, this could also not appear - Firefox didn’t display anything in our tests, while Google Chrome did. Let’s have a look with `curl`:

```
$ curl localhost:8080/games
```

Nothing. A completely empty response. This is quite problematic, as it makes checking our implementation more difficult. Fortunately for us, `curl` also comes with options, and an interesting one is `--verbose`, or `-v`, which prints a lot more information. Let’s give it a try - you should get a somewhat similar output:

```
$ curl -v localhost:8080/games
```

Output:

```
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /games HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.81.0
> Accept: */*
> < HTTP/1.1 405 Method Not Allowed
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

Now that’s more like it. Lines starting with > are header data sent by the client. Lines starting with < are header data returned by the service. The first line to notice here is the first header data sent - we did send a `GET` method. Indeed, `curl` uses a default verb when sending a request, if none was explicitly provided - in this case, a `GET` method. The other interesting line from this output is the first found in the response section: we can see the server returned a 405 status code and its explicit meaning “Method Not Allowed”- which is what we are expecting.

`curl` allows for specific methods to be used via the `--request`, or `-X`, option. In Postman, you’d simply change the method used by using the dropdown box. From a web browser, things are getting a bit tricky. Sometimes, it’s achievable to send a `POST` - or a `PUT`, a `DELETE`, or anything you’d like - using the developer’s settings, but, in most cases, we’ve reached the limits of what browsers offer for our purpose. From now on, we’ll limit the scope of testing with external tools to `curl` - Postman is quite intuitive for all basic uses and doesn’t need guidelines. Let’s shoot a `POST` on our endpoint:

```
$ curl -v -X POST localhost:8080/games
```

Output:

```
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> POST /games HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.81.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 19
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host localhost left intact
Creating a new game
```

This is almost exactly what we wanted. Almost, because there’s a rule in the HTTP protocol: an action that creates a resource should return a status describing that the resource was created. Here’s what raised our attention in the pair of exchanged headers: the status code of the response is `200`, which stands for “OK”. This status code shouldn’t be used when creating a game; instead, according to HTTP standards, we should be using `201`, which stands for “Created”.

Here is a table of the most common HTTP status codes, as described in the the RFC9110 documentation at https://datatracker.ietf.org/doc/html/rfc9110#name-status-codes. This documentation contains the full details regarding HyperText Transfer Protocol (HTTP) protocols, standards and status codes. Make sure to look up the meaning of 418 and where it comes from.

Table 8.1 Most common HTTP status codes

Code	Meaning
200	OK - the server is returning what the client asked for
201	Created - server processed the request and sent the newly created resource as response
202	Accepted - the server will treat the client’s demand later - typically used for asynchronous processes
204	OK, nothing to return - the server correctly fulfilled the request and there is no content to return. It is used by POST, PUT and DELETE commands
400	Bad Request - the server cannot process due to something that's perceived to be a client error (e.g. malformed request, missing mandatory fields)
401	Unauthorised - the client is not allowed to do that action, the request lacks valid authentication credentials for the target resource
403	Forbidden - the client needs to authenticate in order to access this resource, in other words, the server understood the request but refuses to fulfil it
404	Not Found - the server did not find a current representation for the target resource or is not willing to disclose that one exists
500	Internal Error - the server encountered an error and doesn’t know how to deal with it

Let’s bring this final change to the code before we can wrap it up and move on to the next endpoint.

The function in charge of writing this status code is the handler. So, let’s open `newgame/handler.go` and include the call to write the status code. This code appears in the response, and we’ll use the `WriteHeader` method, which only takes a single parameter - the status code to be carried with the response body, in the file `internal/handlers/newgame/handler.go`.

Listing 8.8 handler.go: Set the status code of the successful response

```
func Handle(w http.ResponseWriter, req *http.Request) {
    [...]
    _, _ = w.Write([]byte("Creating a new game"))
    w.WriteHeader(http.StatusCreated)
}
```

Now, if we restart our server, and we run:

```
$ curl -v -X POST http://localhost:8080/games
```

We should see that the response status code is now 201, shouldn’t we? Well, unfortunately, it’s not. It’s still 200. What’s happening? Well, if we have a look at the terminal in which the server is running, we should see a line similar to these lines:

```
http: superfluous response.WriteHeader call from learngo-pockets/
```

What does this mean? We only have one call to `WriteHeader`, how can it be superfluous? It turns out that the `w.Write` call is already setting a header with a default `http.StatusOK` code. The good thing is that `WriteHeader` will ignore any subsequent calls once the header is set, which means it’s basically a regular ordering bug between `w.Write` and `w.WriteHeader` to determine which code will be set. It’s a rigged race, because we’re in charge of it, and we control which one is called first. Let’s adapt the code above to complete this section in the same file, `internal/handlers/newgame/handler.go`:

Listing 8.9 handler.go: Set the response message

```
func Handle(w http.ResponseWriter, req *http.Request) {
    if req.Method != api.NewGameMethod {
        w.WriteHeader(http.StatusMethodNotAllowed)
        return
    }
    w.WriteHeader(http.StatusCreated)
    _, _ = w.Write([]byte("Creating a new game"))
}
```

Great! After restarting it, we can observe that the server now behaves as expected: it returns the correct status code and tells us it's creating a game - shouldn't we believe it? We've checked that it only accepts POST requests. The pesky line about superfluous calls to `WriteHeader` is now gone.

We can happily commit and move forward, but there is a way to make our code shorter.

Open-source multiplexers

In the previous section, we made an assumption. We assumed that the list of endpoints was set in stone and that we would never have to implement anything else. This meant that we were able to associate a path to each endpoint. But suppose we are now informed that we need a new endpoint, something to be able to track the games currently being played. The best URL for this would be `/games`, and the method we'd want to use would be `GET`. There's the rub. We already have a handler for this path, and our handler ensures the method is `POST`. If we were to implement this new `ListGames` endpoint, we'd have to change how our `newGameHandler` is implemented. As a matter of fact, it simply wouldn't be a `newGameHandler` at all! It would listen to a given path.

This is one of the comments that can be made against the `http.ServeMux` type: creating several endpoints (for different verbs) for the same path is cumbersome. For this reason (and a few others that we'll cover as we meet them), writing a personal implementation of the `http.Server` interface is quite common, and many of the most-starred Go projects on github are about this. At the time we are writing, there is an open proposition by the Go team to add this feature into the standard libraries (https://github.com/golang/go/discussions/60227). Here are some popular picks:

- github.com/go-chi/chi: it allows for simple implementation of endpoints

- `github.com/gorilla/mux`: it is very complete and robust
- `github.com/gin-gonic/gin`: this is the most popular mux on github

We picked `chi` for its conciseness and the rate of maintenance by the community. Let’s quickly rewrite the router with this library. First, we need to get the dependency. For this, we need to tell the go tool to add it to our `go.mod` file with the following command:

```
$ go get -u "github.com/go-chi/chi/v5"
```

You might notice that there is a `v5` trailing here. If you try to access this URL in a browser, it will return an error. However, this github repository has been tagged with v5.0.0 at some point (and with more tags as time passes), and using `/v5` in the go get command is how we ensure we use a version compatible with v5 - it could be v5.0.0 or v5.0.8, both offer the same API.

We can then change the router:

Listing 8.10 internal/handlers/router.go: Use the chi library

```
package handlers

import (
    "github.com/go-chi/chi/v5" #A

    "learngo-pockets/httpgordle/internal/api"
    "learngo-pockets/httpgordle/internal/handlers/newgame"
)

// NewRouter returns a router that listens for requests to the fo
//   - Create a new game;
//
// The provided router is ready to serve.
func NewRouter() chi.Router { #B
    r := chi.NewRouter()

    r.Post(api.NewGameRoute, newgame.Handle) #C

    return r
}
```

Alternatively, if you want to make use of the `NewGameMethod` defined in the

API for the sake of your users, instead of calling the method `Post`, you can use another function:

```
r.Method(api.NewGameMethod, api.NewGameRoute, newgame.Handle)
```

This is more verbose but asserts that the server uses what its API exposes. Otherwise, just remove the constant and let users read the documentation.

Exercise 1: Use a walk function to print for each handler the method, and route.

We can remove the check for the method in our handler because `chi` makes sure we never get called with any other method. `Handle` is now done in 2 lines. We can even use the occasion to actually return a game, as defined by the API.

Listing 8.11 newgame/handler.go: Return a Game response

```
func Handle(w http.ResponseWriter, req *http.Request) {
    w.Header().Set("Content-Type", "application/json") #A
    w.WriteHeader(http.StatusCreated)

    apiGame := api.GameResponse{} #B
    err := json.NewEncoder(w).Encode(apiGame) #C
    if err != nil { #D
        // The header has already been set. Nothing much we can d
        log.Printf("failed to write response: %s", err)
    }
}
```

Try it. Now we need to test it.

Testing the game creation

This new shorter version of the handler should be easier to test: fewer lines of code means fewer bugs. What could be blocking is that it takes two rather complicated parameters that we need to mock or stub.

Lots of people are writing services, and this is why they wrote libraries for making the job faster. Lots of people are testing these libraries, so of course we don’t need to stub these ourselves. It’s always good to limit the tests to the code you’ve written.

We can easily create a request with `http.NewRequest`. For the writer, Go has the built-in package `httptest`, with a `Recorder` type and `NewRecorder` build function.

We are making use here of two well-used testing packages `require` and `assert`. Their use can be controversial since some purist will recommend to call only the standard library nevertheless we will use them so you can familiarise with them. They are found in the module `github.com/stretchr/testify`, so let’s start by adding this module to our project with `go get` `github.com/stretchr/testify`.

testify’s assert and require:

testify is a very popular library for testing Go code. It offers a lot of validation tools, the most commonly used ones being its two packages `assert` and `require`. Each of these packages offers similar functions - check whether an error is nil or if two values or equal, but they have a very different behaviour:

- If a function in the `assert` package notices something wrong, a message will be displayed, and the execution of the test carries on.
- If a function in the require package notices something wrong, a message will be displayed, and the execution of the test is immediately terminated.

This helps drive which of assert or require we want to use: the former when we need to check several values, and the latter when we know there is no point in continuing the test if something is wrong.

You should have everything in hand now to write the test to the nominal behaviour.

Listing 8.12 newgame/handler_test.go: Testing `Handle`

```
func TestHandle(t *testing.T) {
    req, err := http.NewRequest(http.MethodPost, "/games", nil) #
    require.NoError(t, err)

    recorder := httptest.NewRecorder() #B

    Handle(recorder, req) #C

    assert.Equal(t, http.StatusCreated, recorder.Code) #D
    assert.Equal(t, "application/json", recorder.Header().Get("Co
    assert.JSONEq(t, `{"id":"",...}`, recorder.Body.String()) #D
}
```

This was our first endpoint - our service now supports the creation of (empty) games. That was a big step, but we've covered a good many important aspects of web services. Before we start playing, our next task is to ensure we can get the status of a game given its identifier.

8.2.2 Get the game status

Here's the picture so far: we have a service that allows for the creation of Gordle games. Of course, the end goal is to have players make guesses, but the second endpoint we'll describe here is the `GetStatus` one. It contains a tiny bit more than the first one, as it introduces only a single new notion: reading a variable input from the user. This time, we can't just always return "a game" - we need to be able to identify which game the user wants to view.

Providing parameters to an HTTP API

There are four main ways for a user to communicate parameters (or variables) to an HTTP web service. We will see that they are each appropriate for specific use cases.

- Path parameters;
- Query parameters;
- Request bodies ;
- Headers .

Path parameters are used when we want to target a single resource. In our Gordle server, an identifier can be used to target an instance of a game. The path to target a game should be `/games/{gameID}`.

Indeed, it’s a common practice to use `/items/{itemID}` in REST APIs (more than the singular version, `/item/{itemID}`). Sometimes, we can accept more than one path parameter. Twitter, for instance, uses `https://twitter.com/{user}/status/{messageID}` to display a message by a user. Similarly, Wikipedia uses path parameters to access its articles: http://jp.wikipedia.org/wiki/金継ぎ., where the identifier of the article is 金継ぎ. Our `GetStatus` endpoint will implement this for the Gordle service.

Query parameters are used to filter the resources we want to target with our request. A filter is a list of pairs of keys and their associated value. There could be 0, 1, or many results - we don’t know, and we can’t make any assumptions. These query parameters are passed in the URI of the request, but at the end of it, separated from the URL by a `?` character. These parameters aren’t specific to an endpoint and could be used in several places of the API. Their syntax is `{{path}}?key1=value1&key2=value2`. The most common example is Google’s search engine: as there can’t (reasonably) be a dedicated resource per possible query that people ask Google, each query is sent to their servers as a query parameter, where the key is `q` and the value was keyed in by the user: https://www.google.com/search?q=金継ぎ . We will show how to use a query parameter at the end of this chapter to improve the NewGame endpoint by allowing the caller to specify which language they want to use.

While path and query parameters should be used to specify which resources we want to operate on (retrieve, delete, update), or what characteristics these resources should have, we sometimes need to provide parameters inherent to the request itself. When we need to send data to the service, we use *body parameters*. This name derives from the fact that they will be transmitted to the service as part of the request’s body. So far, we haven’t seen request bodies, but this will be the point of the third endpoint - `Guess`.

Finally, some parameters are “meta”-parameters - they don’t affect the execution, but, for instance, the format of the output or describe some information about the caller. These parameters are passed in the *headers* of the query - just as the status code is passed in the headers of the response. Headers are usually the place where we store authentication information.

That’s enough theory, let’s start implementing our new endpoint!

Define the HTTP API for GetStatus

As we did earlier, we need to declare the path to this new endpoint, and the method that we expect when it is called. We place these two values in the `api` package to make them visible to other users. We want to retrieve the status of the Game resource without changing it - a `GET` will do. But the path is a bit more complex! In a REST API, every request must contain all the information to identify which resource is targeting. In our case, this means the request needs to contain the ID of the game. As we’ve seen previously, a path parameter is a common way of providing this identifier:

```
/games/{game_id}
```

Indeed, how do we represent a path that is not constant? That’s where the default `net/http` package is a bit too strict - and this is one of the reasons that pushed developers into writing their own routing libraries, such as `chi`. We want to be able to access a game’s status via the path `/games/8476516`, where `8476516` would be the game’s identifier. Obviously, we can’t create billions of routes - one per identifier - so, instead, we’ll let the `chi` library determine how a path parameter should be handled.

Let’s have a look at `chi`’s documentation. Since `chi` isn’t a package of Go’s SDK, we need to be in a module to be able to read it. If we run `go doc github.com/go-chi/chi/v5`, we can read that curly braces around a word are used to represent placeholders in a path. This means we can use `/games/{game_id}` and `chi` will be able to extract the identifier in the handler. Achieving this with Go’s SDK would require lots of security checks, as we’d be splitting the full path into bits separated by slashes. It’s doable, but it would take a lot more lines than what these libraries offer.

Listing 8.13 api/http.go: Add constants for GetStatus endpoint

```
const (
    NewGameRoute = "/games"

    GameID = "id"
    GetStatusRoute = "/games/{" + GameID + "}" #A
)
```

Defining a constant for the `GameID` placeholder will be useful when we want to extract it from the request, in the handler. This is our next step.

Note that we are concatenating strings with `+` here. As `GetStatusRoute` is a constant, the concatenation happens only once, during compilation. For this reason, we don’t need to use a string buffer.

Before we start listening to this route, there is one more definition we want to specify in our documentation: what is the expected status code? This request is asking for a resource, so the possible responses should be "200 here it is", or "404 not found" if the game doesn’t exist. Of course, 500 internal server error" is always a possibility but we want to avoid it as much as possible.

Implement getStatus

Create a package for the new handler, ideally `internal/handlers/getstatus`, with a file for the principal `Handle` function. If you decide that copy/pasting from the `newgame` package is a good option, don’t forget to rename the package in both new files - in all three files, if you dutifully added a `doc.go`.

For now, the implementation of the `GetStatus` endpoint will be limited to only printing the game identifier that the caller passes as a path parameter: we don’t have any storage yet, we just want to make sure that we know how to parse the ID.

Add the path to the router. There is no specific priority when adding handles to a mux. Grouping things based on resources and relater behaviour is usually what makes the most sense.

```
...
r.Post(api.NewGamePath, newgame.Handle) #A
r.Get(api.GetStatusPath, getstatus.Handle) #B
...
```

Now we need to write this `getstatus.Handle` function. It must have the same signature as in the first endpoint (because it’s a `HandlerFunc`). The `chi` library exposes a useful `URLParam` function: take a look at the documentation before using it. If anything goes wrong, we'll call `http.Error`, which writes a message and a status code to the response writer. Remember to always `return` after a call to `http.Error` - this is to prevent any writing to the response. Let's write in the file `internal/handlers/getstatus/handle.go`:

Listing 8.14 handle.go: Status endpoint handler

```
func Handle(writer http.ResponseWriter, request *http.Request) {
    id := chi.URLParam(request, api.GameID) #B
    if id == "" { #C
        http.Error(writer, "missing the id of the game", http.Sta
        return
    }
    log.Printf("retrieve status of game with id: %v", id)

apiGame := api.GameResponse{
        ID: id,
    }
    // ... encode into JSON
}
```

For the sake of simplicity, we decided to use the standard log package which is not thread-safe. So keep in mind that it can lead to unordered logs and complicate later testing.

The validation of the ID could be more thorough: we could check that it is only digits, or that it follows whatever constraints we have put in for security. We will forget this for the sake of keeping our project pocket-sized, but keep it in mind if you push something to production.

Run the server and call the endpoint. Does it return the ID you passed in the URL? Congratulations. You can commit. But wait, what about the test? Good thing you asked.

Testing getStatus

The test here is not much different from the `newGame` version. We only need to add the ID to the list of URL parameters. This requires dipping our first toes into the notion of context, which will be covered in more detail in a later chapter. For now, what needs to be understood is that a context is where `chi` reads URL parameters - and this means it’s how we need to add them.

Listing 8.15 getstatus/handler_test.go: Testing `Handle`

```
func TestHandle(t *testing.T) {
    req, err := http.NewRequest(http.MethodPost, "/games/", nil)
    require.NoError(t, err)

    // add path parameters
    rctx := chi.NewRouteContext()
    rctx.URLParams.Add(api.GameID, "123456") #B
    req = req.WithContext(context.WithValue(req.Context(), chi.Ro

    recorder := httptest.NewRecorder() #C

    handle(recorder, req)

    assert.Equal(t, http.StatusOK, recorder.Code)
    assert.JSONEq(t, `{"id":"123456","attempts_left":0,"guesses":
}
```

The rest, you can guess. Now take the time to use `chi` library by updating the function `Handle()` on `NewGame` endpoint in the file `internal/handler/newgame/handler.go`.

We can now create games and retrieve them. This allows us to make sure everything is now ready for our third and last endpoint - guessing!

8.2.3 Guess

Finally, in order to play, the player must be able to send a query with their word and get a feedback message. What will the API of the third endpoint be?

Request definition

Adding a new endpoint means choosing a new pair of path and method. What method are we going to use? We are changing an already-existing resource, so `PUT` is in order. The path is fairly straightforward - it’ll be

`/games/{game_id}`, similarly to the `getStatus` endpoint, as that's the resource that we'll be interacting with.

But then, the endpoint needs to receive parameters. It will read the guess from the request body, encoded in JSON because we are following HTTP and REST standards. In some cases, updating a resource requires sending its full description - in that case, we would have the same JSON structure for the response of the POST and GET and the request of the PUT. Here, changing the status of a game requires only sending a word, as long as we're providing the ID, so we will go for the simplest JSON object.

```
{"guess":"hello"}
```

This translates into a Request that we can define in the `api` package for others to use.

Listing 8.16 api/guess.go: Request definition

```
// GuessRequest is the structure of the message used when submitt
type GuessRequest struct {
    Guess string `json:"guess"` #A
}
```

Add the path to the endpoint to this file. The path is the same as for GetStatus, but nothing proves that it will always be, so we need another constant. Using the same path constant means forcing them to always be identical, by design. We don't want that - each endpoint deserves to have its path as a constant.

```
GuessRoute = "/games/{" + GameID + "}"
```

What we do want to force by design is the consistency of the `GameResponse` structure. Both `GetStatus` and `Guess` endpoints return a full game, and we don't want the definitions to diverge. For this, we can simply use the same structure as the response in both endpoints.

Decoding a request body

We have an API. Time to write a new handler in a new package and plug it into the router. Nothing needs new explanations, so we can wait while you prepare the handle function. You can even run it with a simple `log.Printf` to make sure it works as expected.

What will this new handler do? First, it should parse the ID of the game, exactly like we did in the previous endpoint. No surprise here.

Second, it should parse the body of the request. You already know from previous chapters how to decode JSON messages into a Go structure. In a flash of genius, somebody thought that the `Body` field of a `http.Request` should implement the `io.Reader` interface - let's make use of that!

Listing 8.17 guess/handler.go: How to read a request body

```
// Read the request, containing the guess, from the body of the i
r := api.GuessRequest{} #A
err := json.NewDecoder(request.Body).Decode(&r) #B
if err != nil {
    http.Error(writer, err.Error(), http.StatusBadRequest) #C
    return
}
```

Print out your findings into the logs and check what happens when you shoot a `curl` at the service:

```
$ curl -v -X PUT "http://localhost:8080/games/123456" -d ‘{"guess
```

Note the `-d` flag for passing a body. You can also use `-d@file.json` if your request body is saved in a file.

Does everything show properly on the logs? Did you write a test?

Testing with a request body

If we run the same test on this handle function as we have in GetStatus, it will panic: the body of our request is `nil`, so we are trying to decode from a `nil` reader, and this doesn't end well. The only change that needs to be made is quite short: the `http.NewRequest` function that we've used to create requests so far takes, as its third parameter, an `io.Reader`, which is quite simple to create from a string in Go (remember to use backticks ` to wrap a string that contains double quotes " without having to escape them):

```
body := strings.NewReader(`{"guess":"pocket"}`)
req, err := http.NewRequest(http.MethodPut, "/games/123456", body
```

We have the full structure of our service. We have 3 endpoints that return something. This point is a good time to deploy the service to a testing environment and have other people play with it. It’s the situation where you send a link to your shiny new service to the rest of the team with a long message asking them to try it, warning them that it does nothing yet, and somebody will reply saying “Hey, nice work but I found a bug: how come it always returns an empty game?” .

Well, let’s fix that.

8.3 Domain objects

Each player’s game must be stored somewhere, a concept which is commonly referred to as “repository” - or “repo”. Out of the very many options, the cheapest and fastest is to store it in memory. This has a lot of downsides: if you decide to scale up a little and deploy more than one instance, the game will only be stored in one, meaning that if your `Guess` query hits an instance on which your game isn’t registered, you get a 404; also if the instance goes down for any reason, your game is lost.

We want this project to fit in a pocket, so we will go for this option for now. In a bigger project, we’d use a proper database.

Separation of concerns

One of the main ways to tell whether a project’s code is *clean* is the *separation of concerns*. In theory, each package, each structure, should have a defined role that can be explained in one sentence, and this sentence is the first line of its documentation. Most of the time, one sentence is enough to cover everything. In practice, this rule can introduce complicated communication between highly coupled ideas, as mentioned before there is always a tradeoff to find between separating and keeping things together.

If the responsibilities of a package don't fit into a handful of words, it is generally a red flag: future maintainers will not know where to find what piece of logic and will end up throwing everything away and rewriting it, not necessarily better. Sometimes we rewrite code only in order to understand what is going on.

Whether you choose hexagonal architecture, lasagne model-view-controller, or any other chimaera that fits your needs, most of the time you want to isolate the data storage management from the API details. You will see some cases where data-storage and api design need to co-evolve to be performant, but they are not the majority.

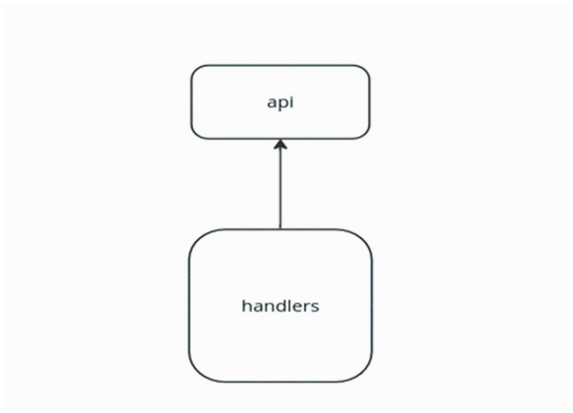
Software theorists often give an example of the perfect data-storing package: "Look, you are using MariaDB, and you just need to change this package import and boom, you are using DynamoDB". This is nice, but it never happens: one does not simply change their storage system. What one does, though, a lot, is maintain it, and knowing that all the DB-related things are here and all the non-DB-related things are not-here will help everyone a lot in the long run.

8.3.1 Domain types

Create a new package for the data repository. Do you want other modules - other developers - to use it? No. That means `internal/repository` will do. This package is responsible for storing and retrieving games. Here, that was a one-sentence documentation.

Here is the structure of our code so far:

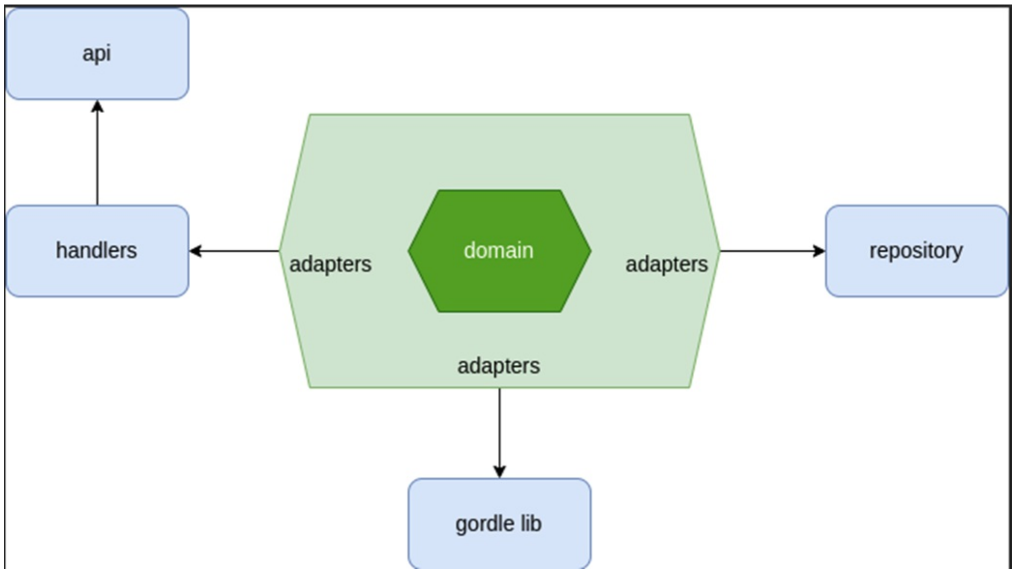
Figure 8.1 Structure of the service



We have an `api` package that other modules can import, and an internal `handlers` package directly called by our `main`.

If we go towards a hexagonal architecture, also known as ports and adapters, and add to it the code domain, the library that we will copy from Chapter 4, and the new data storage, we should aim for something like this:

Figure 8.2. Target structure of the service



This is a bit too complex for the size of the service, so we will keep the domain logic inside the handlers package, but keep in mind that the core logic and the API details should be two distinct things, with an easy-to-draw boundary. Each different layer of our design should serve a specific purpose.

What we call the *domain* here is the core of the service, the business logic of our program. All adapters should be able to rely on it, and it should rely on none of them. This way, we prevent circular dependencies in the code and circular knots in our brains.

The package can be simply called “domain”, “core”, or with a more specific but limiting name. In our case, we find that it deals with a player’s session.

In an `internal/session` package, create a `Game` struct. This should contain everything that the service needs in order to interact with Gordle games.

We know we want to respondto the API’s needs with a `Game` structure:

Listing 8.18 internal/session/game.go: Define the Game structure

```
// Game contains the information about a game.
type Game struct {
    ID        GameID
    AttemptsLeft byte
    Guesses         []Guess
    Status       Status
}
```

As seen above, we need new types, let’s define what is a `GameID` and what is a `Status` in the same file that above.

Listing 8.19 game.go: Define the identifier and status types

```
// A GameID represents the ID of a game.
type GameID string #A

// Status is the current status of the game and tells what operat
type Status string #B

const (
    StatusPlaying = "Playing" #B
    StatusWon     = "Won"
    StatusLost    = "Lost"
)
```

We also want to expose the guesses, as we need to carry them around, store them, and return them.

Listing 8.20 game.go: Define the type Guess

```
// A Guess is a pair of a word (submitted by the player) and its
type Guess struct {
    Word     string
    Feedback string
}
```

Good start. It is not yet enough to play, but that will happen in the next section.

One thing we can already add to the domain is a business error: if the player sends a new game after the game is over, we should be explicit about the problem. You can either define a custom error type or do it the short way:

```
// ErrGameOver is returned when a play is made but the game is ov
var ErrGameOver = errors.New("game over")
```

One of your authors really doesn’t like exposing global variables. You are old enough to decide for yourself.

8.3.2 API adapters

On the API side, we can start manipulating our new `Game` object.

NewGame

Take for example the first endpoint: `NewGame`. This is how the handler currently creates the API version of the game:

```
func Handle(w http.ResponseWriter, req *http.Request) {
    w.Header().Set("Content-Type", "application/json") #B
w.WriteHeader(http.StatusCreated) #B

apiGame := api.GameResponse{} #A
    err := json.NewEncoder(w).Encode(apiGame) #B
    if err != nil //...
}
```

What is the responsibility of the `Handle` function? Dealing with the API details: if the client needs a different flag or a different format, this is where it should happen.

We decided to keep the business logic in the same package, but it doesn’t mean we should keep it in the same function. Instead, we want a function that creates and saves the game somewhere, and another to reshape it into what the clients want.

Listing 8.21 newgame/handler.go: Using session.Game

```
func Handle(w http.ResponseWriter, req *http.Request) {
    game, err := createGame() #A
    if err != nil {
        log.Printf("unable to create a new game: %s", err) #B
        http.Error(w, "failed to create a new game", http.StatusI
        return #C
    }

    w.Header().Set("Content-Type", "application/json")
w.WriteHeader(http.StatusCreated) #E

apiGame := response(game) #D
    // ...
}

func createGame() (session.Game, error) {
    return session.Game{}, nil
}

func response(game session.Game) api.GameResponse {
    return api.GameResponse{}
}
```

This way, each function has a well-defined job, and we can test units rather than big blobs. It also becomes easier to parallelise the work inside a team, merge changes made by multiple people, and most importantly, understand what we are reading.

GetStatus

Let’s move on to the `GetStatus` endpoint. The response format is identical, which means the response function can be reused. Should it live in the `newgame` package and be exposed to the `getstatus` package? That would make very little sense to the next person looking for it in order to change it.

The more straightforward solution here is to have it in an `internal/api` package with this function. We avoid import cycles and whoever comes next will understand what is inside.

Now the name `response` doesn't make sense anymore. It converts a `session.Game` into a `GameResponse`. We can already write most of its logic too, and even start testing it.

Listing 8.22 convert.go: Adapter between API and domain

```
// ToGameResponse converts a session.Game into an GameResponse.
func ToGameResponse(g session.Game) GameResponse {
    apiGame := api.GameResponse{
        ID:           string(g.ID),
        AttemptsLeft: g.AttemptsLeft,
        Guesses:      make([]Guess, len(g.Guesses)), #A
        Status:       string(g.Status),
        // TODO WordLength #D
    }

    for index := 0; index < len(g.Guesses); index++ {
        apiGame.Guesses[index].Word = g.Guesses[index].Word #B
        apiGame.Guesses[index].Feedback = g.Guesses[index].Feedba
    }

if g.AttemptsLeft == 0 { #C
        apiGame.Solution = ""// TODO solution #D
    }

    return apiGame
}
```

Testing this presents no trick: we need to check that an input game would come out as expected in a new shape. You can use `go test -cover` or your IDE's UI to make sure that you are covering all code branches.

Update the `GetStatus` handler and run its tests. Still happy? If in the previous version, your guess slice was `nil` and it is now initialised, it should fail. You should be able to fix it by replacing the json value `null` by an empty array `[]`.

Guess

Finally, the last endpoint. What does the handler do? Let’s have a look.

```
func Handle(w http.ResponseWriter, req *http.Request) {
    // ... decode request

    apiGame := api.GameResponse{
        ID: id,
    }

// ... encode response
}
```

We can already replace this naive initialisation of an `api.Game` with two calls: a new local (unexposed) function with the business logic, and a conversion into the API structure.

Listing 8.23 guess/handler.go: Using session.Game

```
// Handler returns the handler for the game creation endpoint.
func Handle(w http.ResponseWriter, req *http.Request) {
    id := chi.URLParam(req, api.GameID)
    // ...
    r := api.GuessRequest{}
    // ...

    game, err := guess(id, r) #A
    if err != nil {...}

    apiGame := api.ToGameResponse(game) #B

    w.Header().Set("Content-Type", "application/json")
    // ...
}

func guess(id string, r api.GuessRequest) (session.Game, error) {
    return session.Game{
        ID: session.GameID(id),
    }
}
```

The order of things is always the same: decode the request, validate any input that requires validation, call the business logic, convert the returned domain type into API-readable structures and encode the response.

If your tests pass, you can commit. Try running the service and shooting a few curls at it to see how it behaves.

If you deploy again to the testing environment, that unhappy teammate who did not read your warning will still be unhappy: the games are still as empty as before.

8.4 Repository

At this point, we can ask the question of priorities: we still have two main tasks at hand: saving the game and allowing clients to play. Which one should we tackle first?

One way of answering is, which one will show the most progress? If we start with the storage, the first two endpoints will be finished, we will see how they integrate together in an end-to-end test (or at least end-to-midway, because we won't be able to play). If we start with playing, we will be playing on a new empty game at every guess, so testing will be difficult and flaky. Storage it is, then.

If we were using some proven technology for data storage, for example, a SQL database, we would need some extra fields related to storage that the domain does not need: think of fields like created at, deleted at, versioning… In that case, we would create a new `Game` structure, and adapters between the domain and the repository, just like we did for the API. That way, the schema of our database would be able to evolve independently from the domain or API.

Because we are aiming for the fastest storage option, we don't have any technology-related constraints. It means we can keep the domain structures.

8.4.1 In-memory database

There are loads of database options out there, all of them ideal for a limited set of situations. As we explained before, in-memory is ideal for no situation, but fast to write.

What it means is that we will keep a variable to store the games. All the operations we do on games rely on their ID, so a key/value storage is perfect. In Go, this takes the form of a `map`.

Let’s create a package for it. Same question as before: do you want external modules to use your repository? Please, no. It would invalidate the whole point of the service and its API if clients went directly to the DB. Any additional security or logic that you would add (sending events, leaderboards, etc.) would be immediately buggy.

Simplest repository

We have covered in previous chapters how to create an object that will work as a dependency. If we were using an external database, we would initialise a connection when our server starts and keep it as a dependency of the whole service. Here we will initialise the map instead and keep it in the same way.

Let’s create the repository structure. It will hold methods such as `Find` and `Update`.

Listing 8.24 repository/memory.go: Declare the structure

```
// GameRepository holds all the current games.
type GameRepository struct {
    storage map[session.GameID]session.Game #A
}
```

Of course this requires initialisation, so we need a `New()` function.

Listing 8.25 memory.go: New function for the repository

```
// New creates an empty game repository.
func New() *GameRepository {
    return &GameRepository{
        storage: make(map[session.GameID]session.Game), #A
    }
}
```

We could have a method `Upsert`, for both creating and updating. But in our case, we know that some specific use cases create new games and others are only working on existing ones, so we prefer to separate them. It will allow us to validate that we are not trying to create the same entity twice or inserting a game where we have already been playing a few guesses.

Listing 8.26 memory.go: Add a game to the storage

```
// Add inserts for the first time a game in memory.
func (gr *GameRepository) Add(game session.Game) error {
    _, ok := gr.storage[game.ID]
    if ok {
        return fmt.Errorf("gameID %s already exists", game.ID) #A
    }

    gr.storage[game.ID] = game

    return nil
}
```

We will come back to this error. If we want to check for it specifically in the calling code, it is currently difficult.

On Your Own: You can write the `Find` and `Update` methods. The former needs to retrieve from the map and return some kind of error if nothing is there with the given ID. The latter should also prevent insertion and only accept overwriting an already-existing value.

You can also write unit tests on the four functions. Here we would use `New` in the other 3 tests and consider the job done for it: there is no particular trick to it.

The package works, it is tested. How do we use it, though?

8.4.2 Service-level dependency

As we said, we want the repository to be initialised on startup and passed to the service router as a dependency. Let’s look back at what our `main` does.

```
func main() {
    err := http.ListenAndServe(":8080", handlers.NewRouter())
    if err != nil {
        panic(err)
    }
}
```

We can easily add the initialisation and pass the new variable to the router.

Listing 8.27 main.go: Using the repository

```
func main() {
    db := repository.New() #A

    err := http.ListenAndServe(":8080", handlers.NewRouter(db)) #
    if err != nil {
        panic(err)
    }
}
```

How does the router pass this to the handlers? Our `NewRouter` function does not call the handlers, it only gives the router a reference to them, so we cannot simply add a parameter. What we can do instead is turn our Handle functions into anonymous functions that are created on startup.

Let’s anonymise the `Handle` functions and wrap them instead in a `Handler` function that takes a repository as a parameter and returns the previous `http.HandleFunc`.

Listing 8.28 newgame/handler.go: Using the repository

```
// Handler returns the handler for the game creation endpoint.
func Handler(db *repository.GameRepository) http.HandlerFunc { #A
    return func(w http.ResponseWriter, _ *http.Request) { #A
        game, err := createGame(db)

        // ...
    }
}
```

The contents are the same so far. We can now update the router.

Listing 8.29 router.go: Using the repository

```
func NewRouter(db *repository.GameRepository) chi.Router {
    r := chi.NewRouter()

    r.Post(api.NewGameRoute, newgame.Handler(db)) #A
    r.Get(api.GetStatusRoute, getstatus.Handler(db))
    r.Put(api.GuessRoute, guess.Handler(db))

    return r
}
```

How do we test this? We can only pass a concrete repository to our `Handler` function. As soon as we use a real external database, this means we need to spin an instance and connect to it to run unit tests. That is absolutely not sustainable. Let's abstract it with an interface.

The `NewGame` endpoint only needs to add a game to the repository, nothing else. We can actually prevent it from doing anything else by defining a minimal interface.

Listing 8.30 newgame/handler.go: Minimal interface

```
type gameAdder interface { #A
    Add(game session.Game) error
}

// Handler returns the handler for the game creation endpoint.
func Handler(db gameAdder) http.HandlerFunc { #B
    return func(w http.ResponseWriter, _ *http.Request) {
        // ...
    }
}
```

The router's `db` variable automatically implements this little interface, and now it becomes easy to create a stub with one single method for the unit test. Adapting the test requires only 2 changes.

Listing 8.31 newgame/handler_test.go: Stubbing the repository

```
func TestHandle(t *testing.T) {
    handleFunc := Handler(gameAdderStub{}) #A

    req, err := ...
    // ...
    handleFunc(recorder, req) #B

    assert...
}

type gameAdderStub struct { #C
    err error
}

func (g gameAdderStub) Add(_ session.Game) error {
    return g.err
}
```

You can adapt this logic to the other two endpoints. `GetStatus` only needs a `finder`, and `Guess` needs to call two methods.

Now before we rejoice, there is one thing: remember how our server accepts requests in different goroutines and treats them concurrently? Writing into a map is not thread-safe: it means that if two different routines write in the same map, we cannot guarantee which one will win, if any. To fix this problem, we can use a concept we saw in the previous chapter, a mutex. The goal is to avoid concurrently accessing the map and to ensure the sanity of our server.

8.4.3 Add mutex to the repository

The motivation is the same as seen in the previous chapter: we want to protect our repository - a map, here - from concurrent accesses. Let’s keep it simple and use a `sync.Mutex` in `Add()`, `Find()` and `Update()` methods in `internal/repository/memory.go`. Reminder: a mutex should be placed as close as possible to a thread-unsafe variable. We theoretically could have a mutex on the server, preventing two requests from being handled at a time - but that would completely defeat the purpose of having a microservice. Imagine if your favourite web site was only accessible to a single person at a time! On the other hand, if the storage solution was already thread-safe - a database that, for instance, would only accept a single query at a time - we wouldn’t need a mutex at all.

First, we need to add the mutex next to the resource we want to protect, the storage map, inside the `GameRepository` structure.

Listing 8.32 memory.go: Add mutex to GameRepository

```
// GameRepository holds all the current games.
type GameRepository struct {
    mutex     sync.Mutex #A
    storage   map[session.GameID]session.Game
}
```

Then we are able to access the `mutex` from the receiver on each method, here is the sample of code for the `Add` method.

Listing 8.33 memory.go: Call Lock and Unlock mutex on Add()

```
// Add inserts for the first time a game in memory.
func (gr *GameRepository) Add(game session.Game) error {
    log.Print("Adding a game...")

    // Lock the reading and the writing of the game.
    gr.mutex.Lock() #A
    defer gr.mutex.Unlock() #B

    _, ok := gr.storage[game.ID]
    if ok {
        return fmt.Errorf("%w (%s)", ErrConflictingID, game.ID)
    }

    gr.storage[game.ID] = game

    return nil
}
```

You can now update the other methods by yourself and run the tests! If the tests pass, your code is safely committed and you’ve had a good glass of clear water, let’s play!

8.5 Adapting the Gordle library

From chapter 5, or from our repository, copy the `gordle` library. It is far more complex than what we need: it wraps the whole session. We need to refactor it and simplify it for our needs here.

What should it do? We know that we need the following use cases:

- Create and return a new Gordle game;
- Accept a guess and return the feedback.

We could choose to have the Gordle library be in charge of the number of attempts a player has, but we decided to have this logic in the session. Let’s split the responsibilities:

Table 8.2 Package requirements

Package `session`’s `Game`	Package `gordle`’s `Game`
· Store number of attempts left · Store list of attempts and feedbacks · Is able to play a guess by calling Gordle	· Can create a game using a corpus for its solution · Accepts a guess and computes the feedback · A feedback can tell whether a game is over

Now we need to separate the concerns of the `session` and `gordle` packages. The `gordle` library does not need an ID, but the `session` does. The library does not need the previous guesses. It must tell the status of the game.

8.5.1 API of the library

Considering our previous decisions about the distribution of responsibilities, we want to be able to create a game and play. This is what we need to expose:

```
var g gordle.Game = gordle.NewGame(corpus)
var feedback gordle.Feedback = g.Play(guess)

var solution string = g.ShowAnswer()
var won bool = feedback.GameWon()
```

That should cover it. Add a `fmt.Stringer` on the feedback, and we are good.

More design questions:

Who is responsible to tell us how many attempts are allowed, and how many are left? Who is responsible for reading the corpus file, and for picking a random word from it? This is open to discussion: we could decide that the library takes the solution as a string parameter, after all, it does not need anything else. On the other hand, we could say that any use of the library requires this behaviour, so we might as well expose it. On a third hand (?), we could create a corpus-reading package, independent and testable.

In our case, because we copied most of the logic from a previous chapter, we chose option 2, which keeps the old code together until we prove that it should be refactored further. Therefore, we expose a `ReadCorpus` function that returns a list of strings, and this is what the `NewGame` function takes to randomly select a word. Note that `New` taking a single solution simplifies the tests, because we don't have to go through randomisation.

We didn't want to weigh this book with lots of copy-pasted code from a previous chapter. You can find the resulting simplified package in our repository, or play around to see what you need. Have fun going through the exercise of reducing code yourself. For the sake of continuity, and to make sure that we are working with the same code base, here is the final API of our package:

Listing 8.34 API of the package

```
$ go doc internal/gordle

package gordle // import "learngo-pockets/httpgordle/internal/gor

const ErrInaccessibleCorpus = corpusError("corpus can't be opened
const ErrInvalidGuess = gameError("invalid guess length") #B
func ReadCorpus(path string) ([]string, error) #A
type Feedback []hint
type Game struct{ ... }
    func New(corpus []string) (*Game, error) #B
```

Reading the doc shows that option 3, having a corpus reader somewhere else, would have made this API easier to understand. Feel free to refactor this way.

If you are happy with the test coverage of the library, it is time to commit and use it.

8.5.2 Usage in the endpoints

At this point, we have everything we need to write the logic of the endpoints. We have already created a function to isolate that logic in each of the three endpoints, so we just need to fill this function up in each location.

The first thing to do is to keep a `gordle.Game` in our domain: if we add a field of this type to `session.Game`, we can use the field to play.

Listing 8.35 session/game.go: Using gordle.Game in the session

```
// Game contains the information about a game.
type Game struct {
    ID          GameID
    Gordle          gordle.Game #A
    AttemptsLeft byte
    Guesses      []Guess
    Status          Status
}
```

With this new type ready (and tests passing), we can complete the endpoints.

NewGame

Here, we need to create a game, generate a random ID for it and save it, then return it.

Why random? Incremental IDs are a terrible security flaw, as anyone can create a game and play around to mess up with other people's games (note that authentication is mentioned later in this chapter).

We could use an integer. As we mentioned before, generating a random integer can be done quite fast with the `math/rand` package, with a rather poor randomisation, or with better distribution but higher costs with the `crypto/rand` package.

There are other alternatives, like Universal Unique ID (uuid), for which Google's library is most generally used in Go, or Universally Unique Lexicographically Sortable Identifier (ULID). We picked this last one, and we will be using the generative library found here: `github.com/oklog/ulid`.

If you are using a relative path to the corpus, it needs to be defined relative to the compiled binary, not the file where it is defined, not necessarily `main.go` and not the path of execution.

Using the embed package

Go's standard library comes with a very peculiar package: `embed`. This package implements one very useful feature: it allows regular files' contents to be included in the source code of our program. In order to achieve this, we use a specific instruction sent to the compiler at compilation time - they're called *directives*, or *pragmas*. These come in the form of a comment, starting something like `//go:{something}`. Directives are quite common to indicate specific parts of the code to code analysis tools such as linters. The syntax for our need is the following:

```
//go:embed corpus/english.txt
var englishCorpus string
```

This `englishCorpus` variable needs to be defined outside of any function (placing it inside a function would be considered a regular comment by the compiler - and ignored).

In order to use this, first, we need to import the `embed` package. However, as we're not going to use any constant, variable, type, or function from that package, your IDE might get rid of this line completely, and it wouldn't be unfair from it - after all, why import a package if we don't use it?

There is a trick that can be used here: we can force the import of a package by aliasing it to _ (underscore). This way, the package won't be dropped and will be available where we import it. Importing a package as _ is a neat trick which is usually done when we need to call the `init` functions of some libraries. Here, it ensures that the embed package will properly load the contents of the file located at the path `corpus/english.txt`, this location is relative to the source `.go` file. We now have loaded the contents of the file into a variable.

Listing 8.36 newgame/handler.go: Endpoint logic

```
func createGame(db gameAdder) (session.Game, error) {
    corpus, err := gordle.ReadCorpus("corpus/english.txt") #A
    if err != nil {
        return session.Game{}, fmt.Errorf("unable to read corpus:
    }

    game, err := gordle.New(corpus)
    if err != nil {
        return session.Game{}, fmt.Errorf("failed to create a new
    }

    g := session.Game{
        ID:           session.GameID(ulid.Make().String()), #B
        Gordle:       *game,
        AttemptsLeft: maxAttempts, #C
        Guesses:      []session.Guess{},
        Status:       session.StatusPlaying,
    }

    err = db.Add(g) #D
    if err != nil {
        return session.Game{}, fmt.Errorf("failed to save the new
    }

    return g, nil
}
```

We chose to keep a constant for the maximum number of attempts, but if your corpus has words with different lengths, you can be more creative and derive this max number from the length of the word, the difficulty settings, the ELO of the player or any other variable.

Additional optimisation: here we are reading the corpus every time the endpoint is called. This looks like a waste of resources. What would be ideal would be to load it on startup, deal with any error at this point (e.g. file not found) and fail to start if we have nothing. If the service cannot access any list of words, it might as well not start at all.

Now that we have access to the solution, we can also update the API adapter to add the `WordLength`, and other fields that we may have left out so far.

Additionally, this hard-coded path to the corpus will become a pain as soon as we start testing.

Test it with regular expressions:

How do we test this? There is only one pitfall: randomisation. We cannot expect the output of this `createGame` function to be identical when it is called multiple times.

In this situation, it is important to determine what we want to test. We could use a trick, passing an ID generator interface as a parameter, using one that generates a fixed value in tests and a ULID in real life. We can also agree that all we need to assert is that the ID is composed of that many alphanumeric characters and if so we are happy. Whether it is our job to check whether consecutive calls return different IDs is also arguable, we can trust that the lib does it - but can we trust that we will always call the lib?

To validate that "the ID is composed of that many alphanumeric characters", the trick is to use regular expressions. Look into the `regexp` package, there are a lot of options.

When testing the handler itself, we can replace the ID with a known string. For this, we isolate the generated ID in the JSON output and replace it using `strings.Replace`.

We need to look for `"id":"<somenumbersandletters>"` in the output body, so the regular expression we will match against is `` `.+"id":"([a-zA-Z0-9]+)".+` ``. Let's break down this regular expression:

- `.+` : indicates a list of one or more (`+`) of any (`.`) characters, followed by:
- `"id":"` : the very string composed of a double quote, the letter `i`, the letter `d`, a colon, and a double quote, followed by:
- `([a-zA-Z0-9]+)` : this is a captured block - that's what the parentheses represent. They aren't matched by the regular expression, which only matches one or more `(+)` of any letter or number (characters in the range from `a` to `z`, `A` to `Z` or `0` to `9`), followed by:
- `".+"` : a string starting with a double quote, followed by one or more (`+`) of any (`.`) characters.

This string is enclosed in backticks `, which is how we avoid having to escape double quotes in a Go string.

Listing 8.37 newgame/handler_test.go: Replace ID in JSON output

```
// idFinderRegexp is a regular expression that will ensure the bo
// only letters (uppercase and/or lowercase) and/or digits.
idFinderRegexp := regexp.MustCompile(`.+"id":"([a-zA-Z0-9]+)".+`)

id := idFinderRegexp.FindStringSubmatch(body) #B
if len(id) != 2 { #C
    t.Fatal("cannot find one id in the json output")
}
body = strings.Replace(body, id[1], "123456", 1) #D

assert.JSONEq(t, testCase.wantBody, body) #E
```

The expected body contains the known string, so we can now use `assert.JSONEq` or some equivalent. If the ID does not match the expected format, `FindStringSubmatch` will not find it and return only one item: the full string. The test will fail.

Now, when we are testing the `create` function itself, we do not get an encoded body. We can use `FindStringIndex` instead, which finds *the location of the leftmost match of the regular expression* in a string. If the index is there, we're good. This is wrapped by `testify`'s `Regexp` function.

Listing 8.38 newgame/handler_test.go: Validating ID with a regexp

```
func Test_createGame(t *testing.T) {
    corpusPath = "testdata/corpus.txt"

    g, err := createGame(gameCreatorStub{nil}) #A
    require.NoError(t, err)

    assert.Regexp(t, "[A-Z0-9]+", g.ID) #B
    assert.Equal(t, uint8(5), g.AttemptsLeft) #C
    assert.Equal(t, 0, len(g.Guesses)) #C
}
```

Regular expressions are extremely powerful, but also very hard to understand when you don't know what you are looking at. Whenever you write one, do not expect the next maintainer (including yourself) to find it easy to parse: add a comment to tell them what it is looking for. Systematically.

Check that your tests are passing and properly covering your code, and you can move on to the second endpoint.

GetStatus

Here we only need to call the DB and return the game. That’s it. And, of course, deal with any error.

Ah. How do we deal with the errors? How do we know if the game was not found or if there was another unexpected error (e.g., connection error in the situation of a real database)? We want to return a `Status Not Found` if the game doesn’t exist, but `Internal Error` otherwise.

Fortunately, the repository exposes a specific error against which we can check.

Listing 8.39 getstatus/handler.go: Dealing with errors

```
game, err := db.Find(session.GameID(id)) #A
if err != nil {
    if errors.Is(err, repository.ErrNotFound) { #B
        http.Error(w, "this game does not exist", http.StatusNotF
        return
    }

    log.Printf("cannot fetch game %s: %s", id, err)
    http.Error(w, "failed to fetch game", http.StatusInternalServ
    return
}
```

Note that here we choose not to bubble up the errors: the `http.Error` message doesn’t contain the `err` value. Indeed, this would expose the internals of our service to clients, and this is rarely a good idea. The words sent back along with the error are hiding the true error’s details, so we need to log it for debugging purposes.

Guess

Finally, let’s play!

Here we need to fetch the game, play the word, save the result and return it.

Possible errors:

What can possibly go wrong? Problematic scenarios could be: the game is not found, the storage is not responding, the proposed word is not valid, and the game is over, either lost or won. One thing we didn’t add yet was a sentinel error in the domain (the `session` package), to tell us that no, you cannot play a game that you already won (or lost).

Let’s first see what the function of achieving all the work must do. We will omit the errors first, then think about each situation.

Listing 8.40 guess/handler.go: Endpoint logic

```
func guess(id session.GameID, guess string, db gameGuesser) (sess
    game, err := db.Find(id) #A

    if game.AttemptsLeft == 0 || game.Status == session.StatusWon
        return session.Game{}, session.ErrGameOver
    }

    feedback, err := game.Gordle.Play(guess) #C

    game.Guesses = append(game.Guesses, session.Guess{ #D
        Word:     guess,
        Feedback: feedback.String(),
    })

    game.AttemptsLeft -= 1 #E

    switch { #F
    case feedback.GameWon():
        game.Status = session.StatusWon
    case game.AttemptsLeft == 0:
        game.Status = session.StatusLost
    default:
        game.Status = session.StatusPlaying
    }

    err = db.Update(game) #G
    return game, nil
}
```

That’s a long function. In each case, what should the error be?

When we look for the game, we can simply wrap the error with some context. Not much context here, but it is an example. This error will always be of type `repository.Error`, and this is where the `repository.ErrNotFound` can be returned.

```
game, err := db.Find(id)
if err != nil {
    return session.Game{}, fmt.Errorf("unable to find game: %w",
}
```

We can also get an error while playing, and it will be a `gordle.Error`.

```
feedback, err := game.Gordle.Play(guess)
if err != nil {
    return session.Game{}, fmt.Errorf("unable to play move: %w",
}
```

Finally, we are calling the storage again

```
err = db.Update(game)
if err != nil {
    return session.Game{}, fmt.Errorf("unable to save game: %w",
}
```

Because errors are values, we know what happened when we receive an error, and we can adapt the status code and message of the HTTP response.

Listing 8.41 guess/handler.go: Using domain.Game

```
game, err := guess(id, r, db)
if err != nil {
    switch {
    case errors.Is(err, repository.ErrNotFound):
        http.Error(w, err.Error(), http.StatusNotFound)
    case errors.Is(err, gordle.ErrInvalidGuess):
        http.Error(w, err.Error(), http.StatusBadRequest)
    case errors.Is(err, session.ErrGameOver):
        http.Error(w, err.Error(), http.StatusForbidden)
    default:
        http.Error(w, err.Error(), http.StatusInternalServerError
    }
    return
}
```

Don’t forget to test, there is no trick but there are a lot of edge cases.

You have a functioning service! Congratulations. Do you want to write a front-end client now? Playing the game with `curl` only is not user-friendly… You can even write a CLI in Go that calls the service.

Before we leave you, though, we need to list a few warnings about the shortcuts we took.

8.6 A few security notions and improvements

When writing a server, we need to keep in mind that the objective is to deploy it somewhere so that it can serve requests. However, we don’t always know how many users a single server will be in charge of, nor how many queries they’ll send. When it comes to security, we have to bend our minds and think of all “unhappy” paths. We need to identify what could possibly go wrong, and prevent worst-case scenarios from happening. In this case, it’s sometimes useful to imagine ourselves as people wanting to find flaws and break the system.

8.6.1 Limiting the number of requests served at a time

One of the most frequent attacks against a server is called a DDoS attack - it’s a process in which the goal is to overload the server with too many requests. This kind of attack doesn’t extract any information from the server, but it causes it to crash, which makes it unavailable for other users. This attack is usually performed by having lots of computers send thousands of requests to a server. Each request will cause the server to allocate memory to process it - a parallel task (thread or routine), some stack allocation, etc. Since servers have limited resources, at some point, a vast number of requests will cause these resources to be depleted, and the server won’t be able to handle anything at all, in the best cases.

Fortunately for us, `chi` offers a simple way to control how many concurrent requests can be processed simultaneously on our web service with the `Throttle` function. This function takes, as its parameter, the maximum number of requests allowed to be processed simultaneously. It should be called as we declare the mux.

```
r := chi.NewRouter()
r.Use(middleware.Throttle(10))
r.Post(...
```

Of course this number must be declared as a constant at the very least, but ideally configurable.

8.6.2 User authentication

We have mentioned that clients calling the service should be authenticated so that we can make sure a game created by one player will only be played by that one person. It can also help in limiting the number of requests per user and therefore the load on the service.

How do we authenticate a user? One very common protocol is OAuth (Open Authorization) - often written as the latest version of OAuth 2.0. It is used to authenticate a user via an authentication server. Typically, authenticating a user would be the job of one service, which would deal with all the security and provide a signed token. Our Gordle service would then receive this token via an HTTP header, validate it, decode it and find the user identifier in it.

Depending on your needs, you can decide the level where you want to authenticate the clients: indeed you can require an identification for each player, or authenticate each application connecting to your service. In the second option, a website and a mobile app would have different IDs and keys, and each would have a request rate limit, regardless of the number of players that they serve.

Authentication and the security problems that it solves could be the subject of a full chapter but not in the scope of our book. Read *OAuth2 in Action*, by Justin Richer and Antonio Sanso, for more.

8.6.3 Logging

In this chapter, we used the native and very basic `log` package. This is a terrible idea in production: it mangles the log output in a concurrent environment, typically a service. There are lots of great logging libraries out there that protect your output and offer formatting options. With Go 1.21 came a standard library structured logger in the form of the `slog` package - we recommend using it over the `log` package.

8.6.4 Error formatting

When we are returning an error, it is a simple string. Good practice teaches us that when an endpoint’s output is formatted in JSON, the errors should also be formatted. Take an example:

```
{
  "error": "game over"
}
```

This way, your clients can use the same decoder whatever the status code of the response.

8.6.5 Decode query parameters

As we saw before, query parameters are used in API as optional parameters. They always come as pairs of key and value. They are used as filters to specify the resource to be created, updated, or deleted.

Syntax

Query parameters are appended after the path and separated from it by a question mark `?`. To use them, you put the key first, followed by =, the sign equal, and the associated value with which you want to filter. In case we have multiple parameters, we add an ampersand `&` sign between each pair. The whole list of pairs of `key=value` after the ? sign forms the query string.

Let’s implement an example

We want to add the possibility to choose the language in which we want to play Gordle. To do so, let’s add the language as a query parameter. The URL to create the game will look like this:

http://localhost:8080/game?lang=en

In this example, the key is `lang` and the value is `en`.

First, we define the constant for the key, and we declare it next to the path parameter constant:

Listing 8.42 internal/api/http.go: Add query parameter key as constant

```
const (
    // GameID is the name of the field that stores the game's ide
    GameID = "id"
    // Lang is the language in which Gordle is played.
    Lang   = "lang" #A
    // ...
)
```

That was the interesting and very easy part. We will now see how to decode the query parameter from the request. All the query parameters are retrievable from the URL thanks to the `url` library’s `URL.Query()` method, which we can access from the `request.URL` field in our handlers. If you check the return type, you will see that it is an `url.Values`, which exposes (amongst others) a `Get(key string) string` method.

```
$ go doc url.Values
// Values maps a string key to a list of values.
// It is typically used for query parameters and form values.
// Unlike in the http.Header map, the keys in a Values map
// are case-sensitive.
type Values map[string][]string
...
```

So let’s use the `Get` method on the `Values` to retrieve the language in the file `internal/handlers/newgame/handler.go`

Listing 8.43 newgame/handler.go: Update handler to retrieve query parameter

```
// Handler returns the handler for the game creation endpoint.
```

```
func Handler(db gameAdder) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) { #A
        lang := r.URL.Query().Get(api.Lang) #B
        if len(lang) > 0 {
            // TODO create a game in the chosen language
            fmt.Println(lang) #C
        }
```

Congratulations, you now know pretty much everything about REST APIs and how to decode all kinds of parameters!

8.7 Summary

- A web service is a program that continuously listens to a port and knows what to do with requests based on a set of handlers.
- We can use `http.NewServeMux()` to create a multiplexer that will route requests, based on the URL they were sent to, to specific handlers. Otherwise, use open-source libraries.
- A handler's task is to fill an `http.ResponseWriter`. It should `Write` to it, and, sometimes, set headers with `WriteHeader`.
- The default status code set by `ResponseWriter.Write` is `http.StatusOK`. If we want to return a different code, we need to call `WriteHeader` before we start writing the contents of the response via `Write`.
- If an error happens in an endpoint, the handler should not return this error. If you really want to know what's gone wrong, a log is the place where the error is last displayed.
- Some directory names have specific meanings in Go. Files inside `testdata/` will not be compiled by `go build` or `go run`. Files inside `internal/` will not be `import`-able by other modules. `vendor/` is a name that should be avoided for historical reasons.
- Types that we define for the API should only be used in the endpoint handlers. The rest of the time, we should be using types of our domain. Domain - or model - types shouldn't be visible from outside the service's module. Usually, they hide inside an `internal` directory.
- A "repository" - oftentimes called "repo" - offers access to the data, which can be stored in a physical database, in memory, or in any form at all.

- It is always a good idea to check whether your code is thread-safe. Write tests and make use of mutexes when necessary.
- The `embed` package can be used to load the contents of a file (or directory) at compilation time. This is useful, for instance, if you want to keep your SQL queries in .sql files, or when, as we did, you want to load a set of hardcoded values.
- Some, if not most, open-source packages use semantic versioning. Go will natively use the latest v1.x.y version of a package if nothing is specified. In order to enforce using v4.m.n, one should explicitly `"go get path/to/package/v4"` and use `import "path/to/package/v4"` in the .go files.
- Regular expressions are a very powerful way to match patterns. You will find yourself using them in various situations and validating randomised values is one of them. As it can be quickly unreadable, do not forget to explain in a comment any regular expressions you write.
- REST API (REpresentational state transfer) or RESTful API is a set of constraints defining an interface between two systems. REST APIs communicate through HTTP and can exchange data through JSON, HTML or even plain text.
- HTTP statuses are useful to communicate precisely what happened on the server side to the client. HTTP status code is a three-digits from 1xx to 5xx, which could mean everything went well like 200, or resources are not found, such as the well-known 404. Returning the proper code means that the client can better understand what happened. When in doubt, return 500. When at a loss, return 418.

9 Concurrent maze solver

This chapter covers

- Spinning up goroutines as we need them
- Communicating between different goroutines
- Loading and writing a PNG image
- Manipulating images and colours using a go library
- Writing a GIF image
- Using linked lists

The oldest representation of a maze found by archaeologists, from palaeolithic times, was engraved in a piece of mammoth ivory. In Indo-European mythology, mazes are often associated with engineers, like Daedalus in Greece. They are also used as a symbol for the difficult path of a life towards a God figure in O'odham tradition in North America, in India in the Chakra-vyuha style or in Europe on the floors of mediaeval churches to represent the way to salvation.

Solving a maze has been an interesting engineering exercise for ages, including physically with an autonomous robotic mouse (see Micromouse competitions), or virtually, using graph theory. There are countless algorithms, each optimised for different constraints: are there loops? Is the target inside the maze, like a treasure, or on another side, like a liberating way out? Are there curves or only right-angled corners? In the case of multiple possible paths, do we need the shortest or the fastest, or the one that goes through a collection of bonus stars?

In this chapter we want to find the treasure in a maze, starting from an entrance position. Each intersection we meet raises a question: which branch should we explore - to the left, to the right, or straight ahead? We'll answer that question by exploring all branches concurrently, spinning up goroutines each time we have an intersection.

Requirements

- Find the path through a maze that has no loops (there is only one path to reach any pixel).
- The maze is a PNG RGBA image.
- The command-line tool should take an input image’s path and write another image with the pixels from entrance to the treasure highlighted.
- As a bonus, it should also generate a GIF image of the exploration process.

9.1 Maze generation

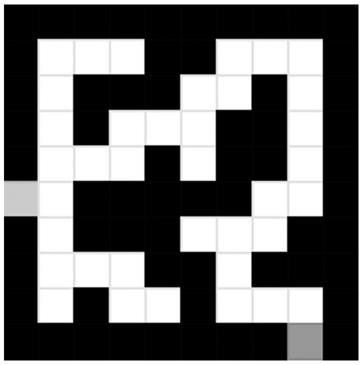
If we want to solve mazes, we need mazes to solve. Because we are developers, we decided to quickly code a maze generator as a side project.

A handful of recognised algorithms for maze generation are available online. By now you should have enough understanding of the Go language to be able to code one yourself. Note that ours is available in the book’s repository, in a `builder` folder, for those most in a hurry. Here are a few important points for a maze generator.

A maze can be represented as a grid in which elements can be either walls or paths. Two special path elements can be found in the maze: the entrance and the treasure. The goal of the maze is to find a list of positions in the grid that links the entrance to the treasure. These positions need to be adjacent - teleportation isn’t allowed in a maze.

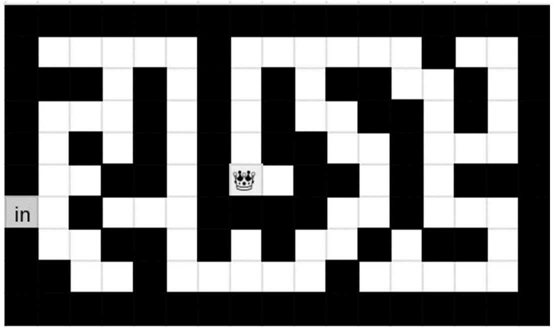
Below you can see an example of a small maze with a treasure corresponding to an exit on the bottom edge.

Figure 9.1 Example of a small maze with a treasure (exit) on the edge



But the treasure does not have to be on an edge, the following figure shows an example of a bigger maze with a treasure inside.

Figure 9.2 Example of a maze with a treasure inside



Since it’s nice to have a preview of the maze, we decided to encode ours as an image. An image is a very convenient way of representing a two-dimensional grid - it has a fixed size, and we can encode the information of whether a grid element is a wall, a path, an entrance or a treasure by using colour values. In the examples above, walls are painted black, while paths are painted white.

9.1.1 What is an image?

In computer science, 2D images are mostly of two kinds - vector images and bitmap images. Vector images are similar to mathematical entities - regardless of how much you zoom in, lines have no thickness, points don’t look bigger on your screen, etc. SVG (Scalable Vector Graphics) is a common format for vector images.

On the other hand, bitmap images, also called raster images, contain a 2D grid of picture elements. These picture elements, or “pixels”, each bear a colour. When zooming in a raster image, pixels are simply displayed larger. Common formats for raster images are PNG (Portable Network Graphics) and JPEG (Joint Photographic Experts Group). JPEG images offer lossy compression, which means they will usually require fewer bytes to store the information - but they might also modify the image while compressing it. The information encoded at the pixel positions is usually colour - but sometimes, we use pixels for something else, such as heat map or density (this is how MRI uses images to represent internal tissues), or for palettes, where each colour has a specific meaning (for instance, a map of the world in which each country is represented with a different colour).

The choice of an image format

Since we want to encode a 2D grid, a raster image format seems perfectly adequate. JPEG’s lossy compression can result in a pixel’s colour being altered, and we want exact values to represent our walls, paths, entrance and treasure. For this reason, we have decided to encode our image as a PNG image.

Of course, Go has a package for image manipulation - and also has a package for most common image formats.

```
$ go doc image
package image // import "image"

[...]

type Image interface{ ... }
    func Decode(r io.Reader) (Image, string, error)

type Point struct{ ... }

type RGBA struct{ ... }
    func NewRGBA(r Rectangle) *RGBA

$ go do image.Point
type Point struct {
    X, Y int
}
    A Point is an X, Y coordinate pair. The axes increase right a
```

One of the types we see in the `image` package is the `RGBAImage`. This offers access to the `RGBAAt(x, y)` method, which allows us to retrieve the colour of a pixel at a given position in the image.

9.1.2 Maze constraints

Remember the constraints we have on the initial version of the solver:

- no loops - there is exactly one path from the entrance to any given point of the maze
- the generated image should be a PNG image using the RGBA colour model

When writing the maze generator, you can also add a complexity constraint on the length of the path from entrance to treasure to avoid straightforward answers. In our implementation for example, that path - the solution - must have a length of at least the height of the image plus its width.

We decided to use the following colours, but feel free to be more artistic and colour-blind friendly:

- Entrance - deep sky blue
- Treasure - pink
- Wall - black
- Path - white

Now we have a generated maze, we can start solving it!

9.2 Maze solver

To solve the maze, we will start by opening the maze image we generated, and we will explore the possible paths, recording them at the same time. By the end, we will have a first hacky version, finishing when the treasure is found.

9.2.1 Setup

As usual, start by setting up your module and creating a `main.go` file at the root. This project being a simple command-line tool, we can have the `main.go` file at the root of the module and the rest will live in the `internal` folder.

The first step to solving the maze will be to open it.

9.2.2 Loading the maze image

As we mentioned, we want the input PNG to be passed as argument to the tool, with the output path:

```
$ maze-solver maze_10x10.png solution.png
```

This means the first thing our `main()` will do is read these 2 arguments.

Listing 9.1 main.go: Reading the arguments

```
package main

import (
    "fmt"
    "log"
    "os"
)

func main() {
    if len(os.Args) != 3 { #A
        usage()
    }

    inputFile := os.Args[1] #B
    outputFile := os.Args[2]

    log.Printf("Solving maze %q and saving it as %q", inputFile,

}

// usage displays the usage of the binary and exits the program.
func usage() {
    _, _ = fmt.Fprintln(os.Stderr, "Usage: maze_solver input.png
    os.Exit(1) #C
}
```

You can already run it and check various scenarios.

Open the input image

We then open the first image, containing the maze. What kind of errors can happen at this point? There could be no file at all, or it could be a non-PNG image. In each case, we want to print an explicit error.

The operation can be summarised in one sentence, so, of course, we put it in one function. It takes the string and returns an image of the `*image.RGBA` type. We want a pointer, because, eventually, we’ll want to modify the image when we write the path to the treasure. We write that function into a new file, one that will contain the file IO operations.

```
func openMaze(imagePath string) (*image.RGBA, error)
```

Let’s call our new function `openMaze`. It will check that the file exists, open it - don’t forget to defer the call to `Close` - and decode the PNG. The last step is done by calling `Decode` from the `image/png` package, which takes an `io.Reader` and returns an `image.Image`, which is an interface.

Now, unfortunately, the `image.Image` interface offers a single method to access a pixel’s value - `At(x, y)` - which returns the `color.Color` of a pixel at the intersection of the `x`-th column and the `y`-th row. The `color.Color` returned by `At` from a regular `image.Image` needs to be converted to the RGBA colour model to be usable. Instead of having to call `At(x, y).RGBA()` everytime we want to access a pixel’s value, we can use an `image.RGBA` - a type that offers a very convenient method `RGBAAt(x, y)`. For this, we’ll simply try to type assert the `image.Image` we decoded from the file into an `image.RGBA` variable.

Go offers `image.RGBA`, but doesn’t offer `image.RGB`. For this reason, it’s simpler to consider RGBA images in this chapter, even though we only chose colours with 100% opacity.

Create a file `imagefile.go` next to `main.go`.

Listing 9.2 imagefile.go: Open the maze image

```
package internal

import (
    "fmt"
    "image"
    "image/png"
    "os"
)

// openMaze opens a RGBA png image from a path.
func openMaze(imagePath string) (*image.RGBA, error) {  #A
    f, err := os.Open(imagePath) #B
    if err != nil {
        return nil, fmt.Errorf("unable to open image %s: %w", ima
    }
    defer f.Close()

    img, err := png.Decode(f) #C
    if err != nil {
        return nil, fmt.Errorf("unable to load input image from %
    }

    rgbaImage, ok := img.(*image.RGBA) #D
    if !ok {
        return nil, fmt.Errorf("expected RGBA image, got %T", img
    }

    return rgbaImage, nil
}
```

We have the image. Don’t forget to call the function in your main, handle the error properly, and you can test manually what happens in different scenarios.

In the code above, there are 2 error cases that are easy to test automatically: unable to open the file, and unable to load the PNG image. Testing whether the function is unable to type assert it as a RGBA image requires a PNG image that wasn’t encoded as a RGBA image. We’ve provided such an image in our repository: `mazes/rgb.png`. You should expect an output like this:

```
go run . mazes/rgb.png solution.png
2023/10/16 18:26:28 INFO Solving maze "mazes/rgb.png" and saving
ERROR: expected RGBA image, got *image.Paletted
exit status 1
```

Finally, we need to handle the `os.Open` error when we can’t open a file that we are able to detect. This case is quite rare - on Unix, it requires execution rights on a directory and no read rights on a file in that directory. Still, it may happen and we will be happy to know if it does.

Write a test, and then we can set up the solving part.

9.2.3 Add the solver

Solver structure

Solving the maze will be done by a dedicated object, one that can be constructed by giving it the image and that carries a `Solve()` method. Why? The object will be able (later) to hold settings such as the colours of the path, walls, entrance, treasure and solution (the path from entrance to treasure). As you will quickly see, it will also hold the channels for communication between the goroutines and the solution at the end. For now, let's keep it simple.

The `Solver` is the heart of the tool and would benefit from living in a dedicated package: `internal/solver`. Remember - packages inside `internal` can't be used by anyone else than your module. Create a file `solver.go` in the `internal/solver` package.

Listing 9.3 solver.go: The Solver structure

```
package solver

import "image"

// Solver is capable of finding the path from the entrance to the
// The maze has to be a RGBA image.
type Solver struct {
    maze *image.RGBA
}
```

Before we go on, let's define the API of this object. It needs to solve the maze and write the solution image. That makes 2 operations, so that makes 2 exposed methods.

Listing 9.4 solver.go: Solve API

```
// Solve finds the path from the entrance to the treasure.
func (s *Solver) Solve() error {
    return nil
}
```

The `SaveSolution` method lives in `imagefile.go` since it is in the image manipulation scope.

Listing 9.5 imagefile.go: Save solution API

```
// SaveSolution saves the image as a PNG file with the solution p
func (s *Solver) SaveSolution(outputPath string) error {
    return nil
}
```

`New` function

Actually, our implementation is highly tied to the image package. Why not delegate the opening of the PNG image to a `New` function? We will need that function anyway.

Move the file `imagefile.go` containing the `openImage` function along with its test to the `solver` package and call the function in a new function called `New`, do not forget to remove it from the `main` function. It takes the path as a parameter and returns a pointer to a `Solver` and an error. Note that, in a file, we tend to write `New` functions and other such constructors after the structure definition.

Listing 9.6 solver.go: New Solver

```
// New builds a Solver by taking the path to the PNG maze, encode
func New(imagePath string) (*Solver, error) {
    img, err := openMaze(imagePath)
    if err != nil {
        return nil, fmt.Errorf("cannot open maze image: %w", err)
    }

    return &Solver{
        maze: img,
    }, nil
}
```

Your current tree should look like this:

```
$ tree
├── go.mod
├── internal
│   └── solver
│       ├── imagefile.go
│       └── solver.go
└── main.go
```

At this point you can even finish writing the main function by building your `Solver` and calling its public API in the proper order. Deal with the various errors in the way you prefer, but don’t forget that CLI tools are expected to return a status code 1 when there is an error, via `os.Exit(1)`. If you have a doubt, you can have a look at the code in the `09-maze_solver/2_solver/2_3_add_solver/main.go` folder.

Run the tests that you have written, commit and have a cup of tea, the next section is the heart of the project.

9.3 Let’s go exploring!

We’ve loaded the maze. Our next objective is to find the path from entrance to treasure, but first we need to find the entrance.

9.3.1 Find the entrance

The first step is quite straightforward, but of course we would not be here if there were nothing to learn on the way.

Colour palette

In order to encode information, the maze generator used pixels to store specific values at specific positions. In our maze, these values will represent walls, paths, entrance, and treasure. We want to compare the colours of the pixels against these specific values. RGBA colours are expressed as structures in the `image/color` package, and structures cannot be constants - which means our reference colours for paths, walls, entrance, and treasure can't be constants. So how do we refer to them? We have a few solutions.

One is to declare the colours as global variables in the `solver` package. Well, we don't like global variables because they can be modified by mistake, undetected by any test, and then the whole behaviour becomes completely unexplainable. It can be a good first step, but we prefer not to have it in production.

An interesting alternative is to create a `palette` structure to hold all the different values. It would be possible to give the tool a palette configuration file with these colours, but this is beyond the scope of our chapter. We don't need to expose the structure as long as the `main` package doesn't need to change the values.

Create a file named `palette.go` in the `internal/solver` package.

Listing 9.7 palette.go: Declare the list of colours

```
// palette contains the colours of the different types of pixels
type palette struct {
    wall        color.RGBA
    path        color.RGBA #A
    entrance    color.RGBA #B
    treasure    color.RGBA
    solution    color.RGBA #C
}
```

A `palette` structure populated with the values that we picked can be returned by a `defaultPalette()` function, with the advantage over global variables that nothing can change the values that a function will return. Unfortunately, it would create a new structure every time you need it, allocating precious memory that needs to be garbage-collected. It can become costly as soon as we start exploring bigger mazes.

Listing 9.8 palette.go: Default colours function

```
// defaultPalette returns the colour palette of our maze.
func defaultPalette() palette {
    return palette{
        wall:     color.RGBA{R: 0,   G: 0,   B: 0,   A: 255}, #A
        path:     color.RGBA{R: 255, G: 255, B: 255, A: 255}, #B
        entrance: color.RGBA{R: 0,   G: 191, B: 255, A: 255}, #C
        treasure: color.RGBA{R: 255, G: 0,   B: 128, A: 255}, #D
        solution: color.RGBA{R: 225, G: 140, B: 0,   A: 255}, #E
    }
}
```

What we did instead was to save these in the `Solver` structure, as settings to solve the picture. It makes sense because another solver with another picture can use a different set of colours.

Listing 9.9 solver.go: Solver with palette

```
type Solver struct {
    maze     *image.RGBA
    palette   palette
}
```

For the moment you can simply set these colours to some default values in the `New` function of the solver package using the function we just defined, `defaultPalette()`.

Let’s follow the signs to find the entrance.

Pixel definition

We will be going through the maze by exploring pixels, identified by their coordinates on the 2-dimensional image. As we’ll need to navigate in these 2D grids, let’s use `image.Point` that Go provides.

```
type Point struct {
    X, Y int
}
```

One thing we can easily anticipate with a pixel is that we will need to find its open neighbours: the pixels bearing the Path colour that are orthogonally connected to it. This will be easier if the pixel can give us the coordinates of its own neighbours. Because of how the maze is implemented, we don’t want to include diagonally-adjacent neighbours.

Create a file named `neighbours.go` in the `internal/solver` package and write the `neighbours` function.

Listing 9.10 neighbours.go: Coordinates of a pixel

```
package solver

import "image"

// neighbours returns an array of the 4 neighbours of a pixel.
// Some returned positions may be outside the image.
func neighbours(p image.Point) [...]image.Point {
    return [...]image.Point{
        {p.X, p.Y + 1}, #A
        {p.X, p.Y - 1}, #B
        {p.X + 1, p.Y}, #C
        {p.X - 1, p.Y}, #D
    }
}
```

Slice or array?

In this function, we returned an array. `[...]image.Point` is equivalent to `[4]image.Point`, which is an array (the length of the array is computed at compilation). Using an array is a minor optimisation, as it will more likely be allocated on the stack than in the heap.

Nothing in this function guarantees that the neighbours are inside the image. For instance, two neighbours of the top left corner, at position {0, 0}, are outside the image. We could add a safety net here, or we could require that none of the edges of the image should be explored.

An alternative is to consider how we’ll use these neighbours. In our code later, we’ll want to check whether they represent an explorable section of the maze, or a wall, or the treasure, or some other information that we code. For this, we’ll have to look at the value returned `RGBAAt(position)`, for each neighbour of a pixel. After checking its implementation, it is clear that it returns a zero value when the position is outside the image. This means it is safe to have `neighbours` return points that would not be within the bounds of our image.

Another thing that the API doesn't guarantee is the order of the neighbours. We could start from the top and go clockwise or be wild and just scramble them. This means the test is the perfect occasion to use stretchr's `testify` framework and its `assert.ElementsMatch` function.

Find the pixel of the entrance

Go back to the file `solver.go` file and create the function `findEntrance`. We'll have to scan the whole image to find one pixel that has the entrance colour. In order to check each pixel's value, a common practice in image processing is to follow the row-major order with two nested loops: an outer loop that will iterate over the rows of the image, and an inner one that will iterate over the rows, just as some languages such as English or Tifinagh write text from the leftmost position to the rightmost, and then to the next line, from the leftmost again.

The reason for this specific pattern is that image formats tend to store pixel values in "scanline" format, where horizontally adjacent pixels of the image are stored in adjacent memory locations. Understandable when we remember that most image format developers are English speakers.

Most of the time, our maze's first pixel will be at position (0, 0) - in the top left corner. But if we're looking at a subsection of an image, our "top left" corner might be at another position. Here, we can access our maze's bounds via the `Bounds()` method on the `image.RGBA` type. This returns two points, that define the bounding box of our image: the `Min` and a `Max` fields

Listing 9.11 solver.go: Find entrance

```
// findEntrance returns the position of the maze entrance on the
func (s *Solver) findEntrance() (image.Point, error) {
    for row := s.maze.Bounds().Min.Y; row < s.maze.Bounds().Max.Y
        for col := s.maze.Bounds().Min.X; col < s.maze.Bounds().M
            if s.maze.RGBAAt(col, row) == s.palette.entrance {
                return image.Point{X: col, Y: row}, nil #A
            }
        }
    }

    return image.Point{}, fmt.Errorf("entrance position not found
}
```

Back in the `Solve` method, we can call this to know where to start.

Listing 9.12 solver.go: Call to findEntrance in Solve

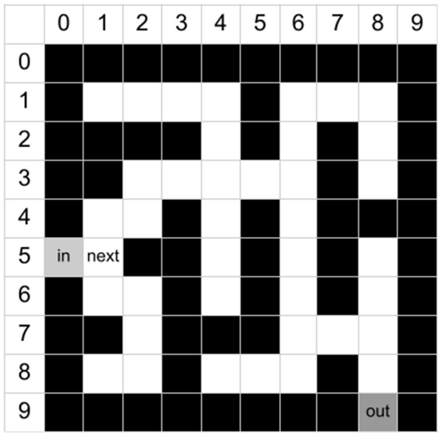
```
// Solve finds the path from the entrance to the treasure.
func (s *Solver) Solve() error {
    entrance, err := s.findEntrance()
    if err != nil {
        return fmt.Errorf("unable to find entrance: %w", err)
    }

    log.Printf("starting at %v", entrance))

    return nil
}
```

What if there is no entrance? Make sure to cover all kinds of situations in your tests. Maybe we want a maze to have a single entrance. If you have trouble writing tests, you can find our test cases in the file `3_exploring/3_1_find_entrance/internal/solver/solver_internal_tes` and all maze images used for the scenarios are located under `internal/solver/testdata`.

Figure 9.3 Maze with entrance and next pixel



We’ve stepped into the maze at the entrance and we now have explored one pixel. The next pixel(s) now need to be explored so that we can reach the treasure!

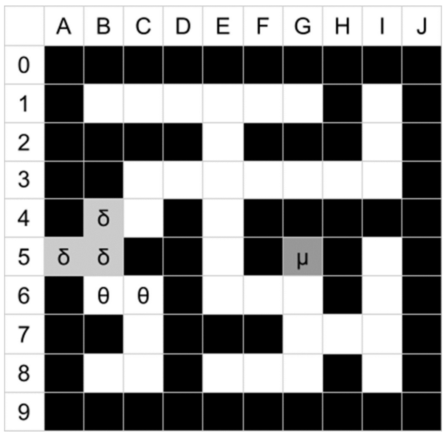
9.3.2 Communicating new possible paths

How do we solve a maze? There are lots of optimised algorithms available, but most of them will offer an iterative approach - turn left at every corner, go to the location that is both nearest to the entrance and unexplored, etc. Each time an intersection is met, a decision must be made - should we start with the left side, then the right side, and the straight ahead? In this chapter, we answer this question by saying: why not try all at the same time? This calls for parallel programming, which in Go is implemented with goroutines. Every time we’ll find a branching in the path, we’ll want to continue in one of the possible directions and start a goroutine with the other(s). Using goroutines means we can delegate the exploration of these other directions and focus on our current branch till we either reach the treasure or a dead end.

Using goroutines can increase the performance - making finding the solution faster - but this isn’t guaranteed. Starting goroutines takes time, and communicating data with them adds on top of that. Usually, goroutines aren’t necessary for very quick tasks. In our case, each goroutine has an undetermined scope: we don’t know when it could end, so we might as well give it its chance. What is certain is that using goroutines increases CPU usage.

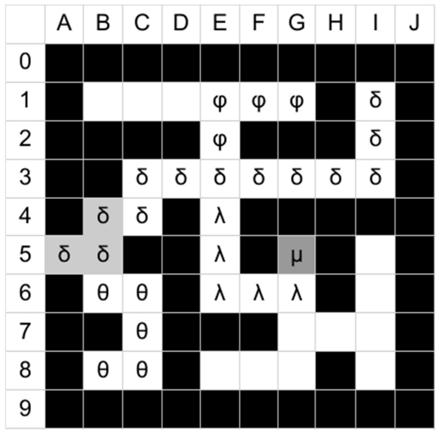
In the example below, we are looking for the Treasure (μ). Goroutine Daedalus (δ) starts in A5, then goes to B5, and needs to branch. A second goroutine, Theseus (θ), picks up at B6 while δ continues in B4, C4, etc. As long as our maze contains no loop, Daedalus won’t meet Theseus as they explore the maze, and this means Daedalus doesn’t need to know about Theseus at all.

Figure 9.4 exploration by different goroutines



From there, both goroutines keep exploring and spinning new explorers. The exploration to reach the monster could look like this (each goroutine’s explored path is represented by a different Greek letter).

Figure 9.5 Exploration representing a possible sequence of events



How does the δ goroutine communicate that a new exploration should be started from B6? And what should be in charge of listening to that notification? In other words, how do goroutines communicate? Via channels, of course.

Communication

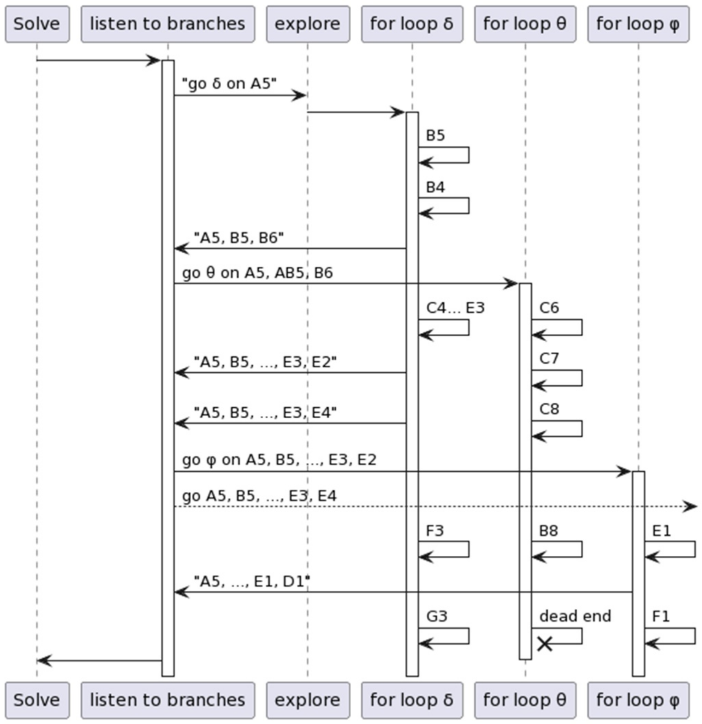
between goroutines is the purpose of channels.

After finding the position of the entrance, our `Solve` function is in charge of initiating the exploration of our maze, starting from there. As soon as a new path should be explored, we want the solver to start the exploration of that branch. Each explorer will be in charge of notifying new branches to our solver with a channel, which will be listening to these notifications. We understand from this that we need two new methods on our solver - the first one will explore a path - we can call it `explore` - and the second one will be in charge of listening to branches - let’s call it `listenToBranches`. Our solver initiates the exploration of the maze by sending a message to the channel that the `listenToBranches` method is listening.

The function `listenToBranches` reads the very first message and creates a goroutine, the one we called Daedalus (δ), for the path starting at A5. Daedalus looks at the neighbours of A5: only B5 is eligible. It integrates B5 to its explored path and checks the neighbours of B5. B4 and B6 are eligible candidates for exploration. Our exploring goroutine, Daedalus, sends the path to B6 to the channel and keeps exploring B4, C4, and so on. Meanwhile, the listener reads the message sent by Daedalus and spins a new goroutine, Theseus (θ), which goes on to C6, C7 and so on, until it finds a dead end and finishes. Meanwhile, Daedalus has sent the path up to E2 to the channel, and the path up to E4, and the listener has spun two new goroutines, λ and φ.

Of course in this scenario, take the grammatical tenses with a dash of salt, because depending on your architecture and the random reassignments decided by the CPU, the future, the present and the past can vary from one run to the next.

Figure 9.6 The sequence of events in `Solve`



Implementing it will help us understand the details better.

Create the channel

What does Daedalus need to communicate in order to start an autonomous goroutine that will be able to explore the path from B6 onwards?

Let’s ask another question: what does Theseus need to know? What does the goroutine that gets to the treasure need to know? The path so far. That’s all. The path through pixels that have been explored to reach this point, which we can express as a linked list of `image.Point`.

Create a file `internal/solver/path.go` and add the `path` structure.

Listing 9.13 path.go: Using a linked list to store the path so far

```
package solver

import "image"

// path represents a route from the entrance of the maze up to a
type path struct {
    previousStep *path
    at           image.Point
}
```

Add a field to the `Solver`: a channel whose messages are pointers to a `path`.

Listing 9.14 solver.go: Add a channel to the Solver structure

```
type Solver struct {
    maze           *image.RGBA
    palette        palette
    pathsToExplore chan *path #A
}
```

This is where goroutines will publish new paths to explore, and where the `listenToBranches` method will listen in order to spin up new exploration goroutines. That method will be added in 9.3.4: common sense dictates that you cannot read from a channel into which nothing has been written yet (we will see that it's actually not so simple).

9.3.3 Record the explored path

Considering that the aim of the program is to tell us how to go from one point to another, we need to know, whenever we explore a new pixel, how we got here.

Behaviour of each goroutine

Let's write the `explore` function, which takes a path as its parameter and explores it. It will go on until it either finds a dead end or the treasure and will publish to the channel any branch it does not take. This function is what each goroutine will do.

For a first version, if we find the treasure, let's only print a message and stop exploring with a `return`. We will come back to that later.

We chose to put this function in a dedicated file `explore.go` in the `solver` package: it is important enough, and we will regularly come back to debug it.

Listing 9.15 explore.go: Explore one path

```
package solver

import (
    "image"
    "log"
)

// explore one path and publish to the s.pathsToExplore channel
// any branch we discover that we don't take.
func (s *Solver) explore(pathToBranch *path) {
    if pathToBranch == nil {
        // This is a safety net. It should be used, but when it's
        return
    }



    pos := pathToBranch.at #A

    for { #B
// We know we'll have up to 3 new neighbours to explore.
        candidates := make([]image.Point, 0, 3)
        for _, n := range neighbours(pos) { #C
    if  pathToBranch.isPreviousStep(n){ #D
                // Let's not return to the previous position
                continue
            }
            // Look at the colour of this pixel.
            // RGBAAt returns a color.RGBA{} zero value if the pi
            switch s.maze.RGBAAt(n.X, n.Y) { #E
            case s.palette.treasure:
                log.Printf("Treasure found at %v!", n)
                return
            case s.palette.path: #F
                candidates = append(candidates, n)
            }
        }


        if len(candidates) == 0 {
            log.Printf("I must have taken the wrong turn at posit
            return
        }

        // See below
    }
}

// isPreviousStep returns true if the given point is the previous
func (p path) isPreviousStep(n image.Point) bool {
    return p.previousStep != nil && p.previousStep.at == n
}
```

Note that, so far, neighbours can only be Wall, Path, Entrance, or Treasure. Entrance has already been skipped because it was the previous pixel, and there is nothing we can do about a Wall. This is why we only have 2 cases in the `switch` - and we didn't add an empty `default` case: we either find the treasure, or another position to explore. Anything else is not interesting. It is important to note that, as we look for eligible neighbours, we don't want to go back on our steps and return to the previous position. This could lead to an endless creation of goroutines and a crash of the program. We'll explore ways of preventing this in section 9.6.1.

Then, there are two cases to consider: either we have no next pixel to explore, in which case it was a dead end, or we do have next pixels to explore - which is when we'll have to send messages to the listening goroutine.. In the case of a dead end, we print a log message to understand what is going on, but there is nothing else to do anymore. We exit the loop and the goroutine ends its execution.

Branching out

What happens when we do have one or more pixel candidates for exploration? We keep one for ourselves (we decided it'd be the first, `candidates[0]`) and send the path to the others on the channel. Then we continue the exploration one step further.

Listing 9.16 explore.go: Keep exploring

```
for _, candidate := range candidates[1:] { #A
    branch := &path{previousStep: pathToBranch, at: candidate}
    s.pathsToExplore <- branch #B
}
```

```
pathToBranch = &path{previousStep: pathToBranch, at: candidates[0
pos = candidates[0]
```

This is a rather long function. We could refactor by extracting logical pieces of code. Loops are usually a good candidate, because they can be summarised in one sentence, which is logically a unit. Let’s have a look at our options:

- Can we extract the inside of the big infinite loop? There would be too many variables to return: the next position, the next “previous” position, a signal about whether to exit, and possibly an error.
- Can we extract the first part, where we look for candidates? This is where we chose to exit in the case of a success. Possible but not too easy.
- Can we extract the second switch, where we look at the candidates? In this situation, we also exit in the case of a dead end. We could pull out the publishing loop, but would the code really become easier to understand for 3 lines?

Let’s keep it this way and see whether writing a test is overly complicated or not. As often, because we want to isolate the logic as much as possible, the test is a bit more complicated than the code itself and requires at least the same notions, so we will keep it for just a bit later. Don’t make it a habit, though.

9.3.4 Wait for unexplored paths and start a goroutine

As we explore, we need a function that listens to the channel and starts a new goroutine for each message in the channel. It doesn’t need to be complex: for each message in the channel, call `explore`.

The keyword in Go for listening to all the messages published to a channel is `range`, exactly like with slices or maps. Using `range` over a channel is blocking, meaning that this function will wait for a message to be published on the channel as long as `close(s.pathsToExplore)` wasn’t called.

Listing 9.17 explore.go: Listen to the channel

```
// listenToBranches creates a new goroutine for each branch publi
```

```
func (s *Solver) listenToBranches() {
    for p := range s.pathsToExplore {
        go s.explore(p)
    }
}
```

This is a very short implementation; it can work but it has a catch with goroutines. We know when they start, but we don't know when they end. Here, we are not keeping track of the different goroutines (nor their amount), which means the program can end while some of them are still running and keep using memory and CPU. We will need to fix this before considering our code correct.

But let's first make our program work. The last thing we need to do in order to kickstart the exploration is to publish the first message.

Start the first goroutine - buffers on channels

How do we start the first goroutine, the one we called Daedalus in our example? We only need to publish the entrance to the channel and start listening.

Let's come back to the `Solve` function. It knows the position of the entrance pixel. We can publish that.

```
s.pathsToExplore <- &path{previousStep: nil, at: entrance}
```

Don't forget to call the listening function `listenTobranches` after that and try running the program. Do you get an error? You should. What we have is this:

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send (nil chan)]:
learngo/09/maze/internal/solver.(*Solver).Solve(0x1400009c180)
```

We are trying to write to a nil channel and to read from it. Initialise it in the `New` function, and try again.

```
pathsToExplore: make(chan *path),
```

Same problem. We have chosen an unbuffered channel and sending a message to it before reading from it will keep causing a fatal error.

Unbuffered channels

A send operation on an unbuffered channel blocks the sending goroutine until a corresponding receive on the same channel at which point the value is transmitted and both goroutines may continue. *The Go Programming Language*, *A. A. A. Donovan & B. W. Kernighan*

We can’t write to an unbuffered channel before we start reading from it. What we can do instead is give a one-value buffer to the channel. One value is enough because after writing this value our main goroutine will listen forever.

```
pathsToExplore: make(chan *point, 1),
```

Alternatively, if we had built an unbuffered channel with `make(chan *point)`, the writing to that channel in the `Solve` function could have been done in a goroutine:

```
go func() { s.pathsToExplore <- &path{previousStep: nil, at: entr
```

As you see if you have logs everywhere, a size of one is enough to get to the solution of a small maze, but we still finish in a deadlock. As we said, the main goroutine listens forever, even when all subroutines have stopped publishing. Go is able to notice that and ends the execution with a deadlock error after a little while. We need a way to tell the listening method to stop.

9.3.5 Stop listening, we found it: short version

When one goroutine finds the treasure, it needs to save the path leading to it somewhere, and somehow tell all the other goroutines to stop looking, as well as tell the listener to stop listening. We will start by implementing a quick version, so that we can get something pretty as soon as possible, see its limitations and find a better solution.

We took a small shortcut a few pages ago, at the point where we found the treasure. At that point, we need to save the path of pixels somewhere where the `SaveSolution` function can find it. But where? Most of the time, the straightforward answer is good enough: we can put it inside the `Solver`. Additionally, it can serve as a flag to tell different goroutines that the treasure has been found and that they can stop looking for it.

Listing 9.18 solver.go: Add the solution

```
type Solver struct {
    maze         *image.RGBA
    palette       palette
    pathsToExplore   chan *path

    solution *path #A
}
```

In the `explore` function, writing to the field is done in just one line. Let’s go back to the switch:

Listing 9.19 explore.go: Save the solution

```
case s.palette.treasure:
    s.solution = &path{previousStep: pathToBranch, at: n} #A
    log.Printf("Treasure found at %v!", n))
    return
```

Wait - any goroutine could be writing to `s.solution`. How do we make sure that we don’t create a race condition? Let’s add a mutex to protect ourselves against this. The mutex is a new field of the `Solver` structure. Here’s how we use it in our case:

Listing 9.20 explore.go: Save the solution with a mutex

```
case s.palette.treasure:
    s.mutex.Lock()
    defer s.mutex.Unlock() #A
    if s.solution == nil {
        s.solution = &path{previousStep: pathToBranch, at: n}
        log.Printf("Treasure found at %v!", n))
    }
    return
```

Now how do we tell other goroutines to stop? Let’s start by using this `solution` field as a flag. Not a great solution, but a fast one. Since we’ll need to check in several places whether the solution was found, we can write a function:

Listing 9.21 explore.go: Stop listening to new messages

```
func (s *Solver) listenToBranches() {
    for p := range s.pathsToExplore {
        go s.explore(p)
        if s.solutionFound() { #A
            return
        }
    }
}

// solutionFound returns whether the solution was found.
func (s *Solver) solutionFound() bool {
    s.mutex.Lock()
    defer s.mutex.Unlock()
    return s.solution == nil
}
```

We have another infinite loop that could use a stop in the `explore` function. We can change the infinite loop so that it stops when the solution is found.

Listing 9.22 explore.go: Stop exploring a path

```
func (s *Solver) explore(pathToBranch []image.Point) {
    //...

    for !s.solutionFound() { #A
        candidates := ...
```

Let’s run it. Has it stopped deadlocking? Depending on the complexity of our input maze, maybe yes, maybe no, because our solution is hacky. Before we fix it properly, it’s time to start automating the test.

9.3.6 Test one goroutine’s logic

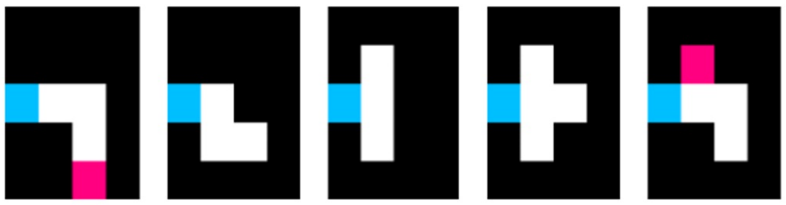
We can test this on an image that is only 4 pixels wide and 5 high: we need a

2x3 grid plus some mandatory walls on 3 sides.

Here are a few test cases:

- Only a path to the treasure
- A maze with one path leading to a deadend
- A maze with two branches
- A maze with a cross
- A maze with a treasure and a deadend

Figure 9.7 A few cases of mazes



If we send the first pixel as parameters, we can count the number of branches that have been published to the channel. For this, we'll create a `Solver`, but we won't listen to its channel. At the end of the run, each branch will have been published to the channel. We can check the number of messages inside a channel with the `len` built-in function, just as we would for slices or maps. Since we won't be listening to the channel, we need to build it with enough capacity to store all the messages that will be published there.

Create a test file `explore_internal_test.go` and the test function `TestSolver_explore`. Feel free to copy from the book repository `internal/solver/testdata` folder.

Listing 9.23 explore_internal_test.go: Test explore function

```
func TestSolver_explore(t *testing.T) {
    tests := map[string]struct {
        inputImage string #A
        wantSize   int #B
    }{
        "cross": {
            inputImage: "testdata/explore_cross.png",
            wantSize:   2,
        },
        // ...
    }
    for name, tt := range tests {
        name, tt := name, tt

        t.Run(name, func(t *testing.T) {
            t.Parallel() #C

            maze, err := openMaze(tt.inputImage) #D
            require.NoError(t, err)

            s := &Solver{
                maze:            maze,
                palette:         defaultPalette(),
                pathsToExplore: make(chan *path, 3), #E
            }

            s.explore(&path{at: image.Point{0, 2}}) #F

            assert.Equal(t, tt.wantSize, len(s.pathsToExplore)) #
        })
    }
}
```

Feel free to add more possibilities. You can also write a second test function for the cases where `len(s.pathsToExplore) > 0`, listen to all the messages and check that we get what you expect. Be careful not to rely on the order of the neighbours sent by the `neighbours()` function, because it is not guaranteed by the implementation. Currently, the behaviour is to always continue exploring in this order of preference: above, below, right, and left. Imagine a future developer reordering the neighbours and breaking this seemingly unrelated test.

Why are we not using `New`? We want to control the size of the channel in the situation of our test, because we are not reading from it. A standard `Solver` has an unbuffered channel; here we want a buffer of 3 paths for all the potential candidates.

We found the treasure, now we need to show how to get there!

9.4 Show the result

We have a solution by now, printable on the terminal, but it’s not exactly human-friendly. The program takes a path in parameter for the output image, writing it should present no particular traps.

Listing 9.24 imagefile.go: Save the output image

```
// SaveSolution saves the image as a PNG file with the solution p
func (s *Solver) SaveSolution(outputPath string) (err error) {
    f, err := os.Create(outputPath) #C
    if err != nil {
        return fmt.Errorf("unable to create output image file at 
    }
    defer func() {
        if closeErr := f.Close(); closeErr != nil {
            err = errors.Join(err, fmt.Errorf("unable to close fi
        }
    }()


    stepsFromTreasure := s.solution
    // Paint the path from last position (treasure) back to first
    for stepsFromTreasure != nil {
        s.maze.Set(stepsFromTreasure.at.X, stepsFromTreasure.at.Y
        stepsFromTreasure = stepsFromTreasure.previousStep #A
    }


    err = png.Encode(f, s.maze) #B
    if err != nil {
        return fmt.Errorf("unable to write output image at %s: %w
    }

    return nil
}
```

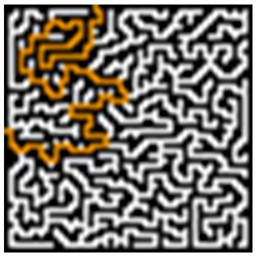
There is a small piece of code that needs explaining here. We start by creating a file, which should always be followed by a `defer f.Close()`. However, `Close()` returns an error, and if we don’t check it, we lose it. So, how can we return both the error that could happen if the deferred call to `Close()` fails and any other error that `Encode` could return? If we have a close look at the signature of the method, we see that we named the output error. This allows us to override `err` in the deferred anonymous function and return an error that would be both the error returned by `Encode` and the one returned by `Close`.

Run the program on a small maze.

```
$ go run . mazes/maze50_50.png sol.png
2023/10/18 09:56:40 INFO Solving maze "mazes/maze50_50.png" and s
2023/10/18 09:56:40 INFO starting at (0,25)
2023/10/18 09:56:40 INFO Treasure found: (18,0)!
```

You should observe a new file in your project, sol.png, which looks somewhat like this:

Figure 9.8 Example of a resolved maze 50px-wide



Then try something bigger. It still ends with a deadlock if your maze is complex enough, doesn't it?

The reason is as follows: when one goroutine finds the solution and saves it in our `Solver` object, in the next nanoseconds, the listener stops listening to the channel, but some other explorers are still looping through their neighbours and publishing to that same channel. As soon as that channel has received as many messages as its capacity allows, writing to it causes a deadlock - and the program exits.

How do we fix this?

9.5 Notify the treasure is found

We have a working solution that is flaky for two reasons: the most obvious one, a technical issue, is that we deadlock when we work on big enough mazes. The other and more urgent reason, a design issue, is that we are not waiting on all of our goroutines before we end the program! Since our maze has only one solution, there is no point for other goroutines to keep searching once we've found the treasure. Fixing the latter will solve the former - if the goroutines stop exploring, they'll stop publishing new paths to explore and the channel won't be full, and we won't have deadlocks any more.

9.5.1 Keep track of all the goroutines

The `listenToBranches` method, responsible for listening to the communication channel and starting goroutines, is the one that knows how many goroutines it started, and how many are still running. It should be the one keeping track of them and waiting for them to finish. The easiest way to keep track of goroutines is to use a `sync.WaitGroup`.

Add a wait group to the `listenToBranches` function. Every time a message is received, it should add one tracker to the wait group before spinning a new goroutine. That goroutine should then tell the wait group when it is done with its work. We don't want to pollute the `explore` method with this logic or spread it across multiple functions, so we can make good use of an anonymous function to call both `explore` and `Done`.

Listing 9.25 explore.go: Stop listening to new messages

```
func (s *Solver) listenToBranches() {
    wg := sync.WaitGroup{}
    defer wg.Wait() #A

    for p := range s.pathsToExplore {
        wg.Add(1) #B
        go func(path *path) {
            defer wg.Done() #C
            s.explore(path)
        }(p) #D
        if s.solution != nil {
            return
        }
    }
}
```

This way, we can be sure that the program will only end when all the goroutines are finished.

You might have noticed that we used an anonymous function with a parameter. Why did we do this? The reason is both important and a bit complicated. All versions of Go, up to 1.22, suffer from the way `for` loops are handled: versions 1.21 and prior would overwrite the variable used to iterate - in our case, the `p` pointer. While this would be fine if we didn't run concurrent activities in our loop's body, things are different here.

Consider the following piece of code:

```
for p := range s.pathsToExplore {
    wg.Add(1)
    go func() {
        defer wg.Done()
        s.explore(p)
    }()
}
```

This is equivalent to the following, as of Go 1.21:

```
var p *path
for {
    p = <-s.pathsToExplore
    wg.Add(1)
    go func() {
        defer wg.Done()
        s.explore(p)
    }()
}
```

Before Go 1.22, we would have no guarantee that the value passed to the first call to explore is indeed the value we first read from the channel - it might have been overridden by the second value by the time the code execution reaches explore. There are two common and useful tricks to prevent that value from being overridden. The first one is the one we presented above - by passing the pointer `p` as a parameter of our anonymous function that starts the goroutine (rather than somewhere within the goroutine), we ensure that it isn't overridden: indeed, the for loop can't read the next message from the channel as long as the goroutine hasn't be started.

```
The other common trick that is frequently used is to simply manua
for p := range s.pathsToExplore {
    p := p
    wg.Add(1)
    go func() {
        defer wg.Done()
        // use the local p
        s.explore(p)
    }()
    ...
```

We aren't simply replacing `p` by itself - here, we are sending the `for` loop variable to the shadow, and we make a safe copy of it that we can send to the goroutine. This trick is very commonly found when using test tables, in which we usually call `t.Run()` on a function that has a `t.Parallel()` in it.

We are now sure that our `listenToBranches` function won't return while some goroutines are still exploring, thanks to the call to `Wait()`. We don't want the explorers to reach dead ends (which they will, eventually). We also don't want them to start new goroutines at intersections - this would cost CPU and memory for no reason. It'd be nice if we could kindly ask them to stop exploring as soon as we know the treasure was found.

9.5.2 Send a quit signal

We already have a stop-exploration condition in our `explore` function: the "infinite" `for` loop keeps exploring as long as the solution isn't found. But we don't like this check on `solution`: the field is used both to save a value and as a flag. This makes the code complex to understand, hard to refactor, and it introduces a data race. There must be a better way to communicate between goroutines that the job is done. Wait… Communication between goroutines? A job for a channel, of course!

Add a quit channel: the select keyword

Let's replace the check of the value of solution in the `listenToBranches` method. We'll be expecting the explorer that found the solution to send a message in a new channel. We want `listenToBranches` to read from that channel in order to know that it should stop launching exploration goroutines. However, our `listenToBranches` function is already listening to a channel - how could it be listening to two at a time, if listening to one is blocking?

This is where the keyword `select` is useful in Go: it accepts several `case` statements (similarly to a `switch`), which can either be “read from a channel” or “write to a channel”, and whichever is validated first gets to be executed - and the others are skipped. If several `case` statements are eligible at the same time, Go will pick a random one.

However, most of the time we aren’t interested in only the first message received from a channel - we want to process all of them. For this, we can use the for-select combination, which allows us to listen to several channels at the same time. It can be seen as an extension of the `for msg := range myChannel` loop that we used to listen forever to one channel.

In our case, we can replace the for-range loop with a for-select loop, which will have the same behaviour.

```
for {
    select {
    case p := <-s.pathsToExplore:
        wg.Add(1)
        go func(p *path) {
            defer wg.Done()
            s.explore(p)
        }(p)
    }
}
```

`select` also accepts a `default` entry, which is mostly used when there is something else than the `select` in the infinite `for` loop.

Let’s add a channel to our `Solver` structure, which we will use to inform `listenToBranches`, and later the explorer goroutines, that the solution was found and that it’s time to stop. We can call it `quit`. Since this channel won’t carry any meaningful messages, we can declare it as a channel of empty structures: `chan struct{}`. Empty structures are a nice feature of Go, as they are extremely lightweight (they use 0 bytes) and quick to create or copy. We need to initialise the channel in the `New` function, when creating a `Solver`.

```
quit: make(chan struct{})
```

We don’t need a buffered channel, as we will listen to it in the `listenToBranches` function. Let’s see what this looks like.

Listing 9.26 explore.go: Stop listening to new messages

```
func (s *Solver) listenToBranches() {
    wg := sync.WaitGroup{}
    defer wg.Wait()

    for {
        select {
        case <-s.quit: #A
log.Println("the treasure has been found, stopping worker")
            return
        case p := <-s.pathsToExplore: #B
            wg.Add(1)
            go func(p *path) {
                defer wg.Done()

                s.explore(p)
            }(p)
        }
    }
}
```

We’re listening to the `quit` channel - but we need to write a message to this channel for it to be useful. Let’s do this in the `explore` method, when we find the treasure.

Listing 9.27 explore.go: Notify that the solution was found

```
func (s *Solver) explore(pathToBranch *path) {
    ...
    switch s.maze.RGBAAt(n.X, n.Y) {
    case s.palette.treasure:
        s.mutex.Lock()
        defer s.mutex.Unlock()
        if s.solution == nil {
            s.solution = &path{previousStep: pathToBranch, at: n}
            log.Printf("Treasure found at %v!", n)
            s.quit <- struct{}{} #A
        }

        return
```

Let’s run this on a few mazes. Does it work? Since we’re working with goroutines, nothing is absolutely deterministic, but we did have errors on our side when running this. Indeed, as soon as our goroutine listening to the new paths exits, we still have some goroutines trying to write in that channel - and that’s a blocking action. Some of our explorers will go into deadlock mode, trying to write to a channel that nothing reads. So far, we’ve just changed the way we reach the same issue. But fortunately for us, there is hope.

To solve this, we need to make sure the explorers stop exploring as soon as the solution is found. Could we read from the `quit` channel? Well, we could, but there is a small conundrum: we don’t know how many explorers are still running at this moment. And we would need to have one message per explorer goroutine if we want each explorer to quit when we find the treasure. Which means we would need to broadcast a potentially huge number of messages in the `quit` channel to ensure every explorer receives its own.

But there is a more interesting way of solving this issue. We can `close` the quit channel. A closed channel is a channel to which writing is impossible, but reading is still possible. A closed channel can’t be reopened, it’s a final action to take. The interesting part of a closed channel is that we can always read from it, and this will always return a value - either one that was previously written in there, or the zero value of the type of messages the channel transmits if there are no written values left to read.

In order to know whether the value read from a channel was written there in the first place, or if it’s kindly returned because we’re trying to read from a closed channel, we can use the second value returned by the `<-` operator:

```
msg, ok := <- myChannel
```

For our business, we don’t really need to care where the empty structure comes from, we only want to try and read from that `quit` channel. Indeed, if we make sure nothing writes to this channel, the only moment when we could read from it will be when it’s closed. And several goroutines can try to read from a closed channel without stealing each other’s message - which means several goroutines can now know if it’s time to stop working.

Let’s replace the `s.quit <- struct{}{}` line with a `close(s.quit)`. This doesn’t change the code in the `listenToBranches` method.

But how can we best use this `quit` channel when exploring?

Stop explorers

The `explore` method performs two tasks: it is in charge of advancing through the maze and of notifying our listener of paths it doesn’t explore. We would like both of these tasks to be ended whenever the solution is found.

Let’s start by thinking about what’s happening with the `pathsToExplore` channel once the solution is found. When the `quit` channel is closed by an exploring goroutine, the `listenToBranches` function returns, which means nothing is listening to the `pathsToExplore` channel any longer, which implies we cannot write to that channel when the solution is found.

How do we make sure we only write there when allowed? We can’t first check `quit` and then write to `pathsToExplore` - this would leave a tiny gap during which another explorer could close `quit`:

```
select {
case <-s.quit:
    log.Printf("I'm an unlucky branch, someone else found the tre
    return
default:
    # A goroutine could close quit between the line above and the
    s.pathsToExplore <- branch
}
```

This solution isn’t secure enough because of that gap. Instead, we want the same logic as we have in the `listenToBranches` method:

```
select {
case <-s.quit:
    log.Printf("I'm an unlucky branch, someone else found the tre
    return
case s.pathsToExplore <- branch:
    //continue execution after the select block

}
```

In this piece of code, we first check whether `quit` is closed (it's the only case when we can read a message from it). If it is, we return, otherwise, we publish our new branch.

Listing 9.28 explore.go: Explore only if the treasure is not found

```
func (s *Solver) explore(pathToBranch *path) {
// ...

for _, candidate := range candidates[1:] {
        branch := &path{previousStep: pathToBranch, at: candidate
        select {
        // s.quit returns a zero value only when the channel was
            log.Printf("I'm an unlucky branch, someone else found
            return
        case s.pathsToExplore <- branch:
            // continue execution after the select block
            }
        }
```

Finally, we want to stop the exploring goroutines when the solution is found. We can do this as the first operation of our infinite loop in the explore method:

Listing 9.29 explore.go: First, check if the solution was found

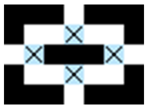
```
func (s *Solver) explore(pathToBranch *path) {
    for {
        // Let's first check whether we should quit.
        select {
        case <-s.quit:
            return #A
        default:
            // Continue the exploration.
        }
```

It is a common pattern to dedicate a channel to communicate that everything should stop. Do not hesitate to have a look at the full method in this book's dedicated repository, at `5_notify_treasure_found/5_2_send_quit_signal/explore.go`.

You now have a working maze explorer, with a few known limitations. If you want to extend it, we have a few ideas to present.

9.6 Visualisation

There are numerous ways to go further with this pocket project. One of the issues we haven't raised yet is that of mazes that contain loops. Imagine a maze containing the following extract:



The four X-marked spaces are paths, but they all go around a "pillar" - a piece of wall not connected to any other wall. At each intersection, a goroutine would either stay close to the pillar or exit the room - but it would still create a branch that would explore around the room. This means that such a room would create an unlimited number of goroutines, something very, very harmful for the wellbeing of our computers. We could ask for the user to provide a maze with no loops, but we could also try and handle it as part of the exploration.

Finally, we did solve the maze, but wouldn't it be nice if we could also show the intermediary steps? Here, we'll animate our progression through the maze and produce a nice GIF file of the exploration.

9.6.1 Overcome the loop constraint

We set ourselves a constraint in the beginning: the maze should never have loops - if you see it as a graph, it is a tree, where each node only has one path leading to it.

If we remove that constraint, there can be multiple possible paths to the treasure. The goal of our chapter is not to find the shortest: we would need to keep track of the distance of each pixel to the entrance by giving it a weight, which could be achieved with increasing values of RGBA pixels.

What we want to try instead in this chapter is just to find one of the solutions and avoid going through the same pixels multiple times.

Strategy

Since we don’t want a goroutine to explore pixels that were previously explored - by either this goroutine or another one - we want to be able to treat them as non-candidate neighbours of pixels being explored. For this, an easy trick is to mark them as explored as we explore them. First, define an `explored` value in the `palette` structure:

```
type palette struct {
    ...
    solution    color.RGBA
    explored    color.RGBA
}
```

And have our `defaultPalette` function return a value for this field (a different value than from the `palette.path`).

Then, in the `explore` function, all we have to do is paint the pixel we’re exploring, and *voil*→!

Listing 9.30 explore.go: Stop exploring a path

```
func (s *Solver) explore(pathToBranch *path) {
    if pathToBranch == nil {
        // This is a safety net. It should be used, but when it's
        return
    }

    pos := pathToBranch.at

    for {
        s.maze.Set(pos.X, pos.Y, s.palette.explored) #A
        select {
```

From now on, a pixel that was explored will not be eligible in our search for candidates - as it won’t be of the `s.palette.path` colour.

Let’s run the program, you should see an image with the solution and the path explored coloured such as the image below:

Figure 9.9 Example of a maze solved with the explored pixels coloured blue

Implementation traps

But wait - we’re working with several goroutines, and we want each one to modify the contents of a pixel in our image - this is a door wide open for race conditions. Is this a problem in this scenario? One could argue that in this case it isn’t: whichever goroutine gets there first, the result will always be the same - each of the goroutines writing the same contents at that pixel, we’re fine with any overwritten or partially written value. But this is totally wrong.

Race conditions

are undefined behavior. Avoid at all costs.

There is no such thing as a benign race condition. Even in a situation when an innocent developer could think well, whichever goroutines writes first, it will be the same result, no. It will not, because undefined behaviour means that the memory could be corrupted somewhere else, or the code could accidentally trigger a bomb. We don’t know. It is undefined. The compiler makes a lot of assumptions when turning our code into machine language, and one of these assumptions is that there is no data race.

9.6.2 Animate the exploration

We know our solution reaches the treasure. We have some logs that tell us which dead ends we managed to find. But this isn’t very visual, and since this is a chapter about images, let’s make it more fun!

The objective of this section is to generate a list of frames as we progress through the maze. Each frame should display the state of exploration at a given moment. We’ll use another image format for this - a GIF, Graphics Image Format - which can be used for animations (even though it wasn’t the initial design). Without having to debate on the pronunciation of this format’s name (at least until the audiobook version of this chapter), it’s interesting to know that a GIF can contain more than a single raster frame. We can encode several frames within a single GIF file, and we can specify a duration for each of them to appear when the GIF file is displayed. These durations are expressed in hundredths of a second, as this is closest to our screen’s refresh rate a unit can be.

Now we’ve decided we want to show the state of the exploration, how do we do it?

Adding frames to the GIF

Well, first, we need to be able to keep track of the pixels we’ve explored so far - which is precisely what we did in 9.5.1. Second, we need to add the frames - the image with its currently explored pixels - at specific moments of our exploration. Let’s start by adding to our Solver structure a new field in charge of holding the GIF using the type from the standard library `image/gif`.

```
type Solver struct {
    ...
    animation      *gif.GIF
}
```

When do we want to take snapshots of our exploration? If we answer this question with a time unit - such as every millisecond - we might face different outputs depending on how fast the program runs on a computer. Otherwise, it could be worth considering that we want to display the status after 10 new pixels have been discovered. Although this would work, we’d face severe problems as our maze grows. Suppose our maze contains 40% of path pixels: on a 10 x 10 maze, there are about 40 path pixels to explore - which would make a 4-frame GIF. However, on a 1000 x 1000 maze, there would be about 400,000 pixels to explore - resulting in a 40,000-frame GIF. Such a file would, first, be very heavy, and second, if we gave each frame one hundredth of a second to be displayed, it’d take more than 6 minutes, in the worst case, to display the exploration.

Instead, we can decide to go with a different approach: let’s decide that we want our final GIF to be 30 frames long if we explore the whole maze. That's an arbitrary number, but it will make for an animation that won't be too long. This means we need to print the state of exploration after *total_explorable_pixels / 30* pixels were explored. We need to count all explorable pixels for our animation, let's write a function for that in a new file, `animation.go`:

Listing 9.31 animation.go: Counting all explorable pixels

```
// countExplorablePixels scans the maze and counts the number
// of pixels that are not walls.
func (s *Solver) countExplorablePixels() int {
    explorablePixels := 0
    for row := 0; row < s.maze.Bounds().Dy(); row++ { #A
        for col := 0; col < s.maze.Bounds().Dx(); col++ { #B
            if s.maze.RGBAAt(col, row) != s.palette.wall { #C
                explorablePixels++
            }
        }
    }
    return explorablePixels
}
```

In section 9.6.1, we added an operation when we encountered a new unexplored pixel - we painted it. Here, we want to do something else when we meet a new pixel. This calls for refactoring these actions into a single method, on the `Solver` structure, that we can call `registerExploredPixel`. This function will paint explored pixels and, depending on how many were explored, it will also be in charge of adding the frame to our animation. However, while painting a pixel with a colour doesn't take too long, "adding the frame to the animation" will mean copying the whole image, something that might take a long time. We don't want that copying to block any exploration process, which means we want the explorers to asynchronously send notifications that a new pixel is to be marked as registered. We wrote this method in a file named `animation.go`.

There are mostly two ways in Go to make asynchronous calls. The first one is to make the call in a goroutine:

```
go s.registerExploredPixel(pos)
```

This is a perfectly valid option, but one has to ask themself if race conditions could happen. Ultimately, this method will require the explicit use of a mutex. But what did we say about communication between goroutines?

The second option, which we'll use here, is to use a channel into which explorers send pixels they want registered. This approach means we will have our `registerExploredPixel` receive pixels from a channel. There is no need for a mutex, as long as we process the pixels read from the channel one at a time. Let's add this channel to our `Solver` structure. Don't forget to initialise it in the `New` function.

```
type Solver struct {
    ...
    exploredPixels chan image.Point
    animation      *gif.GIF
}
```

The explorer's "infinite" `for` loop can be updated to either abort when the quit channel was closed because the solution was found, or send a pixel for registration and continue with the exploration.

Listing 9.32 explore.go: Registering pixels as explored

```
func (s *Solver) explore(pathToBranch *path) { #A
...
    for {
        // Let's first check whether we should quit.
        select {
        case <-s.quit:
            return
        case s.exploredPixels <- pos: #A
            // Continue the exploration.
        }
)
...
}
```

Now we can write the function responsible for registering explored pixels.

In order to know how often we should write a new frame, we define `totalExpectedFrames` as the number of frames we want in the output gif, let's say 30 max. We won't get exactly 30, because we won't be exploring every pixel. We then count the total number of explorable pixels and use the for/select pattern to keep going until we are told to quit.

Every time we receive the position of a newly-explored pixel, we paint it, increment the counter of explored pixels, and if we reached the threshold, paint a new frame.

Listing 9.33 animation.go: Implement registerExploredPixels

```
// registerExploredPixels registers positions as explored on the
// and, if we reach a threshold, adds the frame to the output GIF
func (s *Solver) registerExploredPixels() {
    const totalExpectedFrames = 30

    explorablePixels := s.countExplorablePixels() #A
    pixelsExplored := 0

    for {
        select {
        case <-s.quit: #B
            return
        case pos := <-s.exploredPixels: #C
            s.maze.Set(pos.X, pos.Y, s.palette.explored) #D
            pixelsExplored++
            if pixelsExplored%(explorablePixels/totalExpectedFram
                s.drawCurrentFrameToGIF()
            }
        }
    }
}
```

What is this `drawCurrentFrameToGIF` method? What does it do? How do we paint the frame? First, if we have a look at `go doc gif.GIF.Image`, we notice that the `GIF` structure uses a slice of paletted images. This is a compression algorithm by which each colour used in the image is stored in a palette, and each pixel, instead of being encoded with the classic RGBA values, is encoded with the key of its colour in the palette. Palettes usually have fewer pixels than the whole RGBA spectrum can offer - which sometimes leads to compression artefacts in resulting images. So, how do we create a paletted image? Well, it’s quite straightforward, Go has an `image/color/palette` package. This package only offers two palettes - `Plan9`, and `WebSafe` (with a sweet mention to “early versions of Nestcape Navigator”). Here, the choice is yours. We also have a decision to make - do we want our GIF animation to be the same size our initial maze was, or do we want it of fixed size? Using the same size as the input image is simpler, but it will make most GIFs too small, or too large. Having a frame of a different size than our maze will require pixel interpolation, as we’ll see in a few lines. For the purpose of this chapter, we’ll go with a constant width of 500 pixels, and height in same ratio as input image pixels for each frame:

```
const gifSize = 500
frame := image.NewPaletted(image.Rect(0, 0, gifSize, gifSize*s.ma
```

Using the `image/color/palette` in our code will cause a conflict! Indeed, we already have a type called palette in our package - it defines what colours the walls and the paths should be expected. We can easily resolve this conflict by aliasing the import.

Aliasing imports

In Go, it’s sometimes useful to alias an import. Here, we’ll use `import plt "image/color/palette"`. When aliasing imports, it’s best to use an alias that resembles the original package name to keep the code clear.

We’ve created an empty canvas, let’s draw the current state of the explored maze into it. Unfortunately, Go’s `image/draw` package doesn’t allow for scaling images - and therefore doesn’t allow for any interpolation whatsoever. Instead, we’ll have to use `golang.org/x/image/draw`, its more versatile version. This package offers a `golang.org/x/image/draw.Scaler` interface, which shrinks or expands a rectangle section of an input image to a rectangle section of an output image. `golang.org/x/image/draw` exposes three types that implement the `Scaler` interface: `NearestNeighbor`, `CatmullRom`, and `ApproxBiLinear`. For the purposes of this chapter, we’ll stick to `NearestNeighbor`, as it’s the one that won’t blur our pixels’ edges.

```
draw.NearestNeighbor.Scale(frame, frame.Rect, s.maze, s.maze.Boun
```

Finally, we can add the frame to our GIF image. All three operations can be written into a single method called by markPixelExplored:

Listing 9.34 animation.go: Drawing the frame to the GIF

```
package solver

import (
    "image"
    plt "image/color/palette" #A

    "golang.org/x/image/draw"
)

// ...

// drawCurrentFrameToGIF adds the current state of the maze as a
func (s *Solver) drawCurrentFrameToGIF() {
    const (
        // gifWidth is the width of the generated GIF.
        gifWidth = 500
        // frameDuration is the duration in hundredth of a second
        // 20 hundredths of a second per frame means 5 frames per
        frameDuration = 20
    )

    // Create a paletted frame that has the same ratio as the inp
    frame := image.NewPaletted(image.Rect(0, 0, gifSize, gifWidth

    // Convert RGBA to paletted
    draw.NearestNeighbor.Scale(frame, frame.Rect, s.maze, s.maze.

    s.animation.Image = append(s.animation.Image, frame)
    s.animation.Delay = append(s.animation.Delay, frameDuration)
}
```

We now have a single goroutine in charge of updating the values of the pixel of our image, which does it pixel per pixel, as they come through the channel. Let’s not forget to start this `registerExploredPixels` method in `Solve`. We now have two “listening” goroutines we want to start - `listenToBranches` and `registerExploredPixels`. To launch both and synchronise after they’ve returned, we can use a `sync.WaitGroup`:

Listing 9.35 solver.go: Launch listeners in Solve

```
func (s *Solver) Solve() error {
    // ...
log.Printf("starting at %v", entrance)

    s.pathsToExplore <- &path{previousStep: nil, at: entrance}
    wg := sync.WaitGroup{}
    wg.Add(2)

    defer wg.Wait() #A

    go func() { #B
        defer wg.Done()
        // Launch the goroutine in charge of drawing the GIF imag
        s.registerExploredPixels()
    }()

    go func() { #C
        defer wg.Done()
        // Listen for new paths to explore. This only returns whe
        s.listenToBranches()
    }()



    return nil
}
```

Generating the GIF file

We’ve now added frames to our GIF. Each of them was copied, pixel by pixel, from the maze being explored.

Let’s draw the GIF file! For this, we’ll simply plug somewhere in our code when we know we’re ready to print it. The current `SaveSolution` function is a good choice, since it’s already in charge of writing an output file. Let’s call a new method in there to draw our final GIF.

Listing 9.36 imagefile.go: Generate the GIF file

```
func (s *Solver) SaveSolution(outputPath string) error {
     // ...
gifPath := strings.Replace(outputPath, "png", "gif", -1)
err = s.saveAnimation(gifPath)
if err != nil {
    return fmt.Errorf(...)
}

return nil
```

```
}

// saveAnimation writes the gif file.
func (s *Solver) saveAnimation(gifPath string) error {
    outputImage, err := os.Create(gifPath)
    if err != nil {
        return fmt.Errorf(...)
    }

    defer func() {
        if closeErr := outputImage.Close(); closeErr != nil {
            // Return err and closeErr, in worst case scenario.
            err = errors.Join(err, fmt.Errorf("unable to close fi
        }
    }()

    log.Printf("animation contains %d frames\n", len(s.animation.
    err = gif.EncodeAll(outputImage, s.animation)
    if err != nil {
        return fmt.Errorf("unable to encode gif: %w", err)
    }

    return nil

}
```

This code is very similar to that of the encoding of the PNG image.

Now, let’s run the program:

```
$ go run . mazes/maze50_50.png solution.png
2023/10/18 11:42:57 INFO Solving maze "mazes/maze50_50.png" and s
2023/10/18 11:42:57 INFO starting at (0,25)
2023/10/18 11:43:00 INFO Treasure found: (18,0)!
2023/10/18 11:43:00 INFO the treasure has been found, worker goin
2023/10/18 11:43:00 INFO animation contains 30 frames
```

This should generate the solution.png image, but also a solution.gif file. Open this file to see how the maze was explored! Do you notice anything? The solution doesn’t appear very clearly - if it is at all displayed - and the loop restarts immediately. It’d be nice to make sure the solution is added to the list of frames, and that this final frame is printed for a longer duration. In 9.4, we added the painting of the solution to the `SaveSolution` method. Now that we need to do something on the GIF, we might want a dedicated method for this and move the logic out of the code that writes files into the solver. Let’s write the final lines of code for this chapter. First, paint the pixels between the entrance and the solution in the image stored in the solver, and then add a final frame (which will include the painted solution pixels) to the GIF. By setting a longer value, we ensure that the final frame will be displayed long enough to be admired!

Listing 9.37 solver.go: Finalise exploration by saving solution

```
func (s *Solver) Solve() error {
    // ...
    wg.Wait()

    s.writeLastFrame()

    return nil
}

// writeLastFrame writes the last frame of the gif, with the solu
func (s *Solver) writeLastFrame() {
    stepsFromTreasure := s.solution
    // Paint the path from entrance to the treasure.
    for stepsFromTreasure != nil {  #A
        s.maze.Set(stepsFromTreasure.at.X, stepsFromTreasure.at.Y
        stepsFromTreasure = stepsFromTreasure.previousStep
    }

const solutionFrameDuration = 300 // 3 seconds
    // Add the solution frame, with the coloured path, to the out
    s.drawCurrentFrameToGIF()  #B
    s.animation.Delay[len(s.animation.Delay)-1] = solutionFrameDu
}
```

Rerun the program and open the GIF. You can adjust the values of the frame durations, or the number of frames, to get the look and feel you really want! Unfortunately, we can’t include the GIF in this book, but share yours with your friends!

9.7 Summary

- In computer science, the main type of two-dimensional images are raster images and vector images. Vector images are used in fonts and logos, in infographics, or in icons. Vector images are very scalable - you can zoom in and not see any artefacts.

- The other half of the images we use are raster images - two-dimensional grids of pixels. Each pixel of an image has a colour which can be expressed in the RGBA colour model (but it might be encoded in another colour model, such as the YCbCr, for JPEG images). The value of the colour can be used to encore either a physical information, such as the amount of light of red, green, and blue frequencies that is emitted by an object (as in the picture of a flower), any numerical information, such as the density of population, or finally a palette can be used to represent areas of same category, such as in a map, where each country has its own colour.
- The `image/png` package is used to `Decode` a file into an `image.Image`. This `Image` will frequently be type asserted to a `RGBA` or `NRGBA`. To encode an image, use the `Encode` function from the package you wish to encode your image - available options are `gif`, `jpeg`, and `png`. Other formats require third-party libraries.
- Images usually have their pixel at position (0, 0) in the upper-left. However, some images might have (0, 0) in any other corner. It all depends on the image format and the image’s metadata. Use what the image package returns to iterate over the pixels of an image.
- You can access a pixel’s value in an `image.Image` with the `.At()` method. This returns a `color.Color()` that you have to convert to `color.RGBA`. When using an `image.RGBA`, you can use `RGBAAt()` instead, which will return a `color.RGBA` that can then be compared to known values.
- In order to write a pixel to an `image.RGBA`, use the `Set(x, y, rgba)` method.
- When scanning a whole image, use two nested loops, the outermost one iterating over the rows, and the innermost one iterating over the columns. This is beneficial, performance-wise, for all “scanline” formats.
- When you can’t have global constants, it’s slightly cleaner to have a function that returns configuration values rather than using global variables. Avoid exposing global variables for safety reasons: other pieces of code might change them.
- Writing to an unbuffered channel that isn’t read from is blocking. Either

write to it in a goroutine, or use a buffered channel, whose size should be the maximum number of elements that will be written there before the reading starts.

- When starting goroutines in loops, make sure your loop variables are protected. The loop variables can be the messages you read from a channel, the keys or values of a map you iterate through, or the elements of a slice.
- There are three common ways of protecting iterators of a for-loop when a goroutine is launched inside the loop:
    - you can either use a version of Go that guarantees that (currently, it’s considered for Go 1.22)
    - you can shadow the loop variable with another one in your loop (usually, we give the new variable the same name as the loop variable)
    - finally you can launch your goroutine with an anonymous function that takes the loop variable as a parameter.
- The `select` keyword allows a piece of code to listen to several channels. Whenever a message is published in any of the channels, the code written in the `case` statement will be executed.
- If several `case` statements in a `select` are eligible, Go will pick a random one.
- It is common to have one of the `case` statements of a `select` be a return condition. This is especially true in servers, where the processing of an input request should be ended as soon as the request is cancelled.
- The `for-select` “infinite loop of listening” pattern is very common. Usually, one of the cases of the `select` block will contain the condition to exit the loop.

10 Habits Tracker using gRPC

This chapter covers

- Writing a web service using Protobuf and generating the Go code of its gRPC definition
- The Context interface in Go
- Running the service with basic endpoints
- Testing with integration tests

As developers, we spend most of our day in front of a screen for work, on top of any leisure activity we might have. Unfortunately, the effects of a high number of hours watching these lit pixels - albeit sometimes positive for moral or psychological aspects - are mostly considered negative for eyesight, causing eye fatigue, dry eyes, or difficulty focusing. On the other hand, there are some activities that will alleviate these ophthalmic conditions - most of them include simply doing something else than watching a screen. Usually, recommendations go along the path of regularly taking a stroll, reading a book, or having a physical activity.

It’s never easy to pick up a new habit, and no one has ever gone from never jogging to running a marathon. The goal is always incremental. But the important point is to track how much of these habits one can get done in a week, and maybe adjust objectives for the next week.

In this chapter, we’ll write a service in charge of registering such habits. The user will be able to create habits, give them an expected frequency - a number of times per week they are expected to be completed - and list them. We’ve already written an HTTP service, this time we’ll focus on another popular network remote procedure call protocol, this one developed by Google: gRPC.

Functional requirements

- Create and delete a habit

- List the created habits
- Tick a created habit
- Get the status of a habit

Technical requirements

- gRPC service, with Protobuf in and out
- Run locally
- In-memory DB to start

10.1 API definition

In the same vein as we did in chapter 8, we are going to create a web service to track personal habits; that is, a Go program that runs indefinitely, ready to listen to requests and respond to them. Requests are sent by clients, who need to know what to send and how to understand the response. Such a set of definitions that is called an Application Programming Interface, or API for short. Here, the client is the user who wants to track her habits. She will do it by calling the endpoints such as `CreateHabit` exposed on an API we are going to build.

In this case, we want the communication between the clients and our service to use the gRPC framework, where messages are encoded using the Protocol Buffers (Protobuf) format and using the HTTP/2 network layer. Protocol Buffers are a programming language-independent description of how these messages are encoded.

Protocol Buffers

Protocol Buffer fields are a mechanism used for serializing structured data. While this can also be achieved by lots of other ways (JSON, XML, yaml, …), Protocol Buffers have an emphasis on two important points: versioning the serialized model, and reducing any non-data information. Invented by Google in 2001 and released to the public in 2008, they are perfect for high-network applications like microservices. Protocol Buffers is a way to describe communication between programs in a cross-language way. You can define what data is being sent via message definitions. You can also define endpoints for what is being communicated. Messages and service APIs, the endpoints are written in Protocol Buffers files (text files with, usually, the `.proto` file extension), which can then be compiled to generate clients for the programming language of your choice, as we'll explain below. Clients can be generated for many common languages, including Go. A few limitations: Protobuf messages are not self-describing – you need to know how to read them before you can access their contents. And this also means we can't simply use regular tools such as curl to send messages to a gRPC endpoint - testing will also be a bit trickier than with JSON APIs.

The first step in the development of a system, once we know the requirements, is generally to define the API: how the system will be used. Any language can be used, as long as we know our users will use this language - or that some tools can be used to generate adequate files to connect to our servers. In this section, we'll use Protobuf to declare our API. Our Protobuf files will be compiled into Go files that we can use to implement our service.

The final API will resemble the following, and throughout this chapter, we go through each step necessary to implement these endpoints:

Listing 10.1 Habits service API

```
// Habits is a service for registering and tracking habits.
type HabitsService interface {
  // CreateHabit is the endpoint that registers a habit.
  CreateHabit(CreateHabitRequest) (CreateHabitResponse, error)

  // ListHabits is the endpoint that returns all habits.
  ListHabits(ListHabitsRequest) (ListHabitsResponse, error)

  // TickHabit is the endpoint to tick a habit.
  TickHabit(TickHabitRequest) (TickHabitResponse, error)

  // GetHabitStatus is the endpoint to retrieve the status of tic
  GetHabitStatus(GetHabitStatusRequest) (GetHabitStatusResponse,
}
```

10.1.1 Protobuf declaration

While this is not a book about Protobuf, we need a few basics to define our API.

Initialise your go module the usual way: create a directory, and run:

```
go mod init learngo-pockets/habits
```

Even before creating a `main.go` or anything, create a folder at the root of the project, named `api/proto`, where we can store the Protobuf files. Their extension is `.proto`.

The service's job is to deal with habits, so create a file `habit.proto` where we can define what a habit is. For the moment, we will give it a name and a weekly frequency. For example, if I want to practise Go 5 times a week, I want to be able to send something along those lines:

```
{"practice Go", 5}
```

Habit entity

Let's start with a minimal API definition of what a habit is. It has a name and a weekly frequency. We can write a Protobuf file with the entity.

Each proto file starts with the version of the protocol, then defines a Protobuf package, and in our case, because we want to generate Go code, a Go package. Generated code will end up in the folder named after the package and situated inside the `go_package` module path. As often, a piece of code will make things clearer.

Listing 10.2 habit.proto: Headers of the proto file

```
syntax = "proto3"; #A

package habits; #B
option go_package = "learngo-pockets/habits/api"; #C
```

Every structure in Protobuf is a *message* and every field is given a number that will allow consumers to recognise it. If we decide that the name is 1, it will have to stay 1 forever, and future versions with different fields will still look for the name at index 1.

Listing 10.3 habit.proto: define the Habit message

```
// Habit represents an objective one wants to complete a given nu
message Habit { #A
  // Name of the habit, cannot be empty
  string name = 1; #B
  // Frequency, expressed in times per week.
  int32 weekly_frequency = 2;
}
```

In a Protobuf message, each field must have a unique identifier. There is no point in leaving gaps, just follow incremental order. The syntax is to list each field with its type followed by its name and identifier. You can find more examples and lists of supported types here: https://protobuf.dev/programming-guides/proto3/ .

Don’t hesitate to be extremely verbose in your comments: this is what users will read in order to figure out how to use what you made, not the generated Go code. Comments in the proto files will be carried over into the generated code.

Service definition

Once we have this simple `Habit` object, we can declare a service to manipulate it. In another file, define a service that will use this message.

It is good practice, for version compatibility, to define a Request and a Response, even when they are empty or when they contain only one field: it makes changes smaller and avoids breaking the API. The gRPC `Habits` service is in charge of registering and tracking habits. That is the place where we will add along the way all the needed endpoints to track the habits.

Listing 10.4 service.proto: define the Habits service

```
syntax = "proto3";

package habits;
option go_package = "learngo-pockets/habits/api"; #A
```

```
// Habits is a service for registering and tracking habits.
service Habits { #B
}
```

First endpoint: Create

This service exposes nothing, as you can see. The first endpoint that we need is for creating a habit to track.

A generally accepted best practice when naming inputs and outputs of endpoints is to have a dedicated *message* (a structure with fields) for each of them, called request and response, or input and output, even when they are empty or contain only one field. Consider the difference between these 2 signatures:

```
func CreateHabit(Habit)
func CreateHabit(CreateHabitRequest) CreateHabitResponse
```

In the first case, we give a habit and expect nothing in return. Simple. In the second case, we need to define two additional structures, it’s verbose, it’s annoying - the first one will just contain a `Habit` field and the other will be empty. What would be the point?

The point is version intercompatibility. Let’s say in the next version we want to add a user token to identify which user is creating the habit, and then return a habit identifier. In the more verbose case, we would just add a field in each structure, and if it is not mandatory, any code written for the initial version will still work, whereas in the first and straightforward case, we would break the whole API.

For this reason, the `CreateHabit` endpoint will use its `Request` and `Response`. You might wonder what happens in the case of errors - why wouldn’t they appear in the proto API? The answer is that the gRPC-compilation tool will be in charge of adding support for errors. This support differs from language to language - in Go, we can have several returned values, whereas in C++ or Java, the error needs to be returned differently - which means we don’t write errors in the proto file - but, don’t panic, the Golang interface compiled from this proto will allow us to return an error.

We can add the endpoint to the service with one line, then define 2 new messages.

Listing 10.5 service.proto: define the Habits service

```
service Habits {
  // CreateHabit is the endpoint that registers a habit.
  rpc CreateHabit(CreateHabitRequest) returns (CreateHabitRespons
}
```

In order to use the `Habit` message in the response, we need to import the neighbouring file.

Listing 10.6 service.proto: define the CreateHabit in and out

```
import "habit.proto"; #A

service Habits {
  ...
}

// CreateHabitRequest is the message sent to create a habit.
message CreateHabitRequest {
  // Name of the new habit. Cannot be empty.
  string name = 1;
  // Frequency of the new habit. Will default to once per week.
  optional int32 weekly_frequency = 2; #B
}

// CreateHabitResponse is the response of the create endpoint.
message CreateHabitResponse {
  Habit habit = 1;
}
```

And done. In a handful of lines, we have an API for the first step of the tracker, which is the creation of a habit. As you can see, there is no path and no verb: they are specific to HTTP. gRPC does not use them.

As we want to use it in Go, now is the time to generate the Go code.

10.1.2 Code generation

Generating code from Protobuf files is done using `protoc`, standing for proto compiler. We will also install two plugins, namely `protoc-gen-go` and `protoc-gen-go-grpc`.
https://grpc.io/docs/languages/go/quickstart/#prerequisites lists all steps for this, but we'll repeat them here.

Installation steps

Depending on your system, installing `protoc` might be achievable simply, through a package management tool such as Homebrew's `brew` (on Mac) or `apt` (on Linux). Unfortunately, there are a few more steps when installing it on Windows. Here are the commands you can run from a terminal:

```
$ apt install -y protobuf-compiler #Linux
$ brew install protobuf #Mac
```

Once `protoc` is installed, getting the Golang-specific dependencies is made easy by the fact that we can ask Go to do it:

```
$ go install
google.golang.org/protobuf/cmd/protoc-gen-go@latest
$ go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest
```

These two utilities are used to compile `.proto` files declaring messages and services into Golang files. They work as plugins for `protoc` – they will be called if `protoc` detects we want to compile Golang files.

Compilation

The compilation command is pretty long. We will try to decompose it step by step.

In your favourite terminal, navigate to the root of the go module and try the very minimal version:

```
protoc api/proto/habit.proto
```

The compiler complains: it needs output directives. For what language should it generate the compiled files? Where? The `go_out` parameter will tell the compiler both the requested output language and the location for compiled files by specifying the target folder. By specifying this option, we also tell `protoc` to use the `protoc-gen-go` plugin.

```
protoc --go_out=api/ api/proto/habit.proto
```

This generates a Habit structure, but it puts it in an impractical location: the whole module tree is created all over again:

```
.
├── api
│   ├── learngo-pockets
│   │   └── habits
│   │       └── api
│   │           └── habit.pb.go
│   └── proto
│       ├── habit.proto
│       └── service.proto
│
```

That’s not what we want, we would like the Go file to appear directly in the api folder. Fortunately for us, there is an option for that: `--go_opt=paths=source_relative`.

```
protoc --go_out=api/ --go_opt=paths=source_relative api/proto/hab
```

Now the tree looks like what we want. The last step is to compile all of the proto files, not only Habit.

```
protoc --go_out=api/ --go_opt=paths=source_relative api/proto/*.p
```

It doesn’t work. What the compiler is doing with this command is to take each file separately and generate a Go file for it. When it reaches `service.proto`, it cannot import `Habit` because we never told it where to look.

The `-I` option has the following documentation if you run `protoc --help`: Specify the directory in which to search for imports. May be specified multiple times; directories will be searched in order. If not given, the current working directory is used.

Perfect for our needs.

```
protoc -I=api/proto/ --go_out=api/ --go_opt=paths=source_relative
```

Once you have run this command, your tree should look like this:

```
.
├── api
│   ├── habit.pb.go
│   ├── proto
│   │   ├── habit.proto
│   │   └── service.proto
│   └── service.pb.go
├── go.mod
└── go.sum
```

All the messages exist as Go structures, but not the service yet. We also need to generate the gRPC part.

The options are quite similar to the pure Go ones: `go-grpc_out` and `go-grpc_opt`. Passing these options on the command line will silently tell `protoc` to use the `protoc-gen-go-grpc` plugin.

```
protoc -I=api/proto/
--go_out=api/ --go_opt=paths=source_relative
    --go-grpc_out=api/
--go-grpc_opt=paths=source_relative api/proto/*.proto
```

There is one final parameter that we must talk about, when it comes to the Go gRPC compiler, and this has to do with forward-compatibility. Suppose that we're happy with the current proto API, that we use it to compile the Golang files, and that we implement the server interface with a structure of our own. Then, let's assume we want to add a new endpoint - we'll have to update the proto file, and regenerate the Golang files. As mentioned on the go-grpc repository, "it is a requirement that adding methods to a service cannot break existing implementations of the service". So, how did they ensure this requirement is always met?

There are two options. The first one is to require that any implementation of the server embeds a type defined in the generated file. The other is to allow for the developers to not implement the required server interface. While this second option is not recommended, it is still available by passing another parameter to the command line:

```
--go-grpc_opt=paths=source_relative,require_unimplemented_servers
```

In the rest of this chapter, we will use files that were generated without this final option - and we'll remind you to embed the type when creating the server type.

Automated generation

Remember to put this massive command in a place where you and future maintainers will find it, typically in a Makefile or as part of a script. You might wonder why we wouldn't place this in a `generate.go` file with a `//go:generate` directive – the reason is that we were lazy in our command-line and used a `*` to send all the `.proto` files to `protoc`. Unfortunately, while shells understand how to expand `*.proto` into "every file with a `proto` expansion", `go generate` doesn't, which prevents us from using the same command-line directly in a `//go:generate` directive. However, if you have access to `bash` or `sh`, or any other shell you fancy, you can tell `go generate` to run a command in a shell with the following syntax (don't forget the double quotes around the command that you really want to run) :

```
//go:generate bash -c "protoc -I=api/proto/ {...} api/proto/*.pro
```

We provided an example of a `generate.go` (a common name for files that only contain `//go:generate` directives) file that contains a similar command. We slightly adapted it because we placed it directly into the `api` directory. It's up to you to decide whether you want a target in your Makefile or if you'd rather call `go generate` to produce these files.

Make sure to document it, like everything that is not considered general knowledge in the industry.

Your tree should now look something like this:

```
.
├── api
│   ├── proto
│   │   ├── habit.proto
│   │   └── service.proto
│   ├── generate.go
│   ├── habit.pb.go
│   ├── service_grpc.pb.go
│   └── service.pb.go
├── go.mod
└── go.sum
```

Ready to start coding in Go?

10.2 Empty service

Now that we have an API exposing the create habit endpoint for our user, we can write the code and make it run. We will first create an empty service, make it run, and then add the endpoints. After that will come the data layer, and finally integration tests, and we will be ready to start again with more functionalities.

10.2.1 Creating a small logger

A logger is often the first package that is written in a module, as it’ll likely be used by every other package. But loggers can sometimes be problematic - they’ll write to whatever output we tell them to write to. Sometimes, this causes issues - for instance, should the log messages always be printed when testing? And to what output? In this section, we’ll implement a small logger that will make it easier for us to both run and test our code with logs.

We can notice that the `testing.T` structure already implements a `(t *T) Logf(format string, args ...any)` method that will only print what we called it with when the current test fails. In order to be able to use a logger in our code and in our tests, let’s write a small logger that will only expose one method - the same as exposed by `testing.T`. This way, we will be able to create loggers in our code, and inject the `t` test variable in tests as the test logger. This will prevent output-jamming.

Listing 10.7 log/log.go: define a small logger

```
package log

import (
    "io"
    "log"
    "sync"
)

// A Logger that can log messages
type Logger struct {
    mutex  sync.Mutex #A
    logger *log.Logger
}

// New returns a logger.
func New(output io.Writer) *Logger {
    return &Logger{
        logger: log.New(output, "", log.Ldate|log.Ltime), #B
    }
}

// Logf sends a message to the log if the severity is high enough
func (l *Logger) Logf(format string, args ...any) {
    l.mutex.Lock()
    defer l.mutex.Unlock()
    l.logger.Printf(format, args...)
}
```

Now that we have our basic logger, we can start implementing the server package. We will need this logger there.

10.2.2 Server structure

First, we create a structure that will be our server. It would not make sense if it were to stay empty for long; it will soon contain a repository for data retention.

In a new folder `internal/server`, create a `server.go` file and add the struct with a `New` function. As we'll want to use a logger, let's declare a one-method interface that we will use as our logger.

Listing 10.8 server.go: define the web service

```
// Server is the implementation of the gRPC server.
type Server struct {
    lgr Logger #A
}
```

```
// New returns a Server that can ListenAndServe.
func New(lgr Logger) *Server { #B
    return &Server{
        lgr: lgr,
}
}

type Logger interface {
    Logf(format string, args ...any)
}
```

Not very interesting yet. We need to add a `ListenAndServe` method on the `Server`, so that it can start listening to and serving new requests sent on a given port. Ports are virtual “doors” through which messages transfer, either internally or with the rest of the world, so that only one application can listen to a given port on the same machine. Ports are identified with their port number - 80 is used by HTTP, 443 by HTTPS, etc. When listening to a specific port, either use one that is assigned by a standard, such as 80 for HTTP, or use a port number between 1024 and 49151 for internal usage.

A gRPC server, just as the HTTP server we saw in Chapter 8, is first and foremost a good listener. We give it a port to listen to, and start it with a call to `Serve`. This call will only return when the server shuts down.

But a gRPC server is a bit more than a HTTP server - it must implement the desired gRPC API. For this, we start by creating a barren server using the `grpc` package, and we then attach our implementation to that server by registering it.

Listing 10.9 server.go: listen to a given port

```
import (
...
    "google.golang.org/grpc"

    "learngo-pockets/habits/api"
)

// ListenAndServe starts listening to the port and serving reques
func (s *Server) ListenAndServe(port int) error {
    const addr = "127.0.0.1"
    listener, err := net.Listen("tcp", net.JoinHostPort(addr, str
    if err != nil {
        return fmt.Errorf("unable to listen to tcp port %d: %w",
    }

    grpcServer := grpc.NewServer() #B
    api.RegisterHabitsServer(grpcServer, s) #C

    s.lgr.Logf("starting server on port %d\n", port)

    err = grpcServer.Serve(listener) #D
    if err != nil {
        return fmt.Errorf("error while listening: %w", err)
    }

    // Stop or GracefulStop was called, no reason to be alarmed.
    return nil
}
```

There are better ways of starting the server to support graceful shutdown. We will improve this later in the chapter. Additionally, if you want to allocate a free port randomly, you can use port `0`. The documentation of `net.Listen` explains which networks are supported.

Wait… This does not compile. We cannot register a `HabitService` that does not know how to create a habit. As you can see, `api.RegisterHabitsServer` takes as a second parameter anything that implements the `HabitServer` interface, which was generated from our Protobuf service. We just need to implement that one method.

When trying to compile or run, we also faced an error mentioning that our `Server` type cannot be registered as a `HabitsServer` because it doesn't implement a method named `mustEmbedUnimplementedHabitsServer`. This is a reminder that, when we generated the Go files from the proto files, we used the recommended way, which requires embedding a structure, as the non-implemented method's name suggests. So, let's embed the required type:

```
// Server is the implementation of the gRPC server.
type Server struct {
    api.UnimplementedHabitsServer
lgr Logger
}
```

Composition and embedding

Both concepts extend the notion of a structure, but in a different way. While composition, which in Go is achieved by listing named fields of a structure, represents a “has-a” relationship between two types, embedding corresponds to a “is-a” relationship. In our case, our Server *being an* UnimplementedHabitsServer, it has an implementation for that required method.

As we know that this method will require tests and probably side functions, we can already put it in a `create.go` file in the server package. The signature of this function was generated by the `protoc` toolchain; we can’t alter it. As we’ll see, there is a mysterious first parameter, into which we’ll dive later in this chapter.

Listing 10.10 create.go: implement the HabitServer interface

```
// CreateHabit is the endpoint that registers a habit.
func (s *Server) CreateHabit(
_ context.Context, #A
request *api.CreateHabitRequest
) (*api.CreateHabitResponse, error) {
    s.lgr.Logf("CreateHabit request received: %s", request)

    return &api.CreateHabitResponse{
        Habit: request.Habit,
    }, nil
}
```

This should be enough for now. We will come back to it very quickly. Our endpoint is implemented - our whole service is implemented. It’s now time to spin it up.

10.2.3 Creating and running the server

We can call these `New` and `ListenAndServe` functions in `main`. As this is a web service, we prefer to put the `main.go` file in a `cmd/habits-server` folder, decluttering the root of the module. On some operating systems, such as Windows, `go run dir/main.go` will cause an executable file called `dir.exe` to be generated and executed - placing the `main.go` file in an aptly named directory is important in that regard.

What the `main` function does is create a new instance of our server and call `Listen`, which only returns if there is an error. Since we need to inject a logger into our server instance, we can create it in the main function and pass it via `server.New`. We can use that logger in the main function too.

Here is how we create a new server in our main package and run it:

Listing 10.11 main.go: Run it

```
package main

import
    "fmt"
    "os"

    "learngo-pockets/habits/internal/server"
    "learngo-pockets/habits/log"
)

const port = 28710 #A

func main() {
lgr := log.New(os.Stdout) #B

    srv := server.New(lgr) #C

    err := srv.Listen(port) #D
    if err != nil {
        lgr.Logf("Error while running the server: %s", err.Error(
os.Exit(1) #E
    }
}
```

There is basically no logic inside the main function. It means that our service will be easier to test: all the logic is in isolated packages.

Run it!

```
go run cmd/habits-server/main.go
```

It does absolutely nothing, but it runs. How wonderful! Add a few logs in

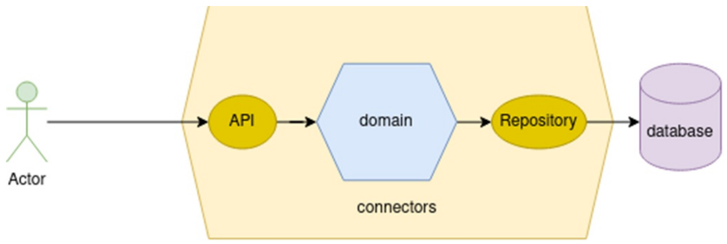
`ListenAndServe` to make sure.

10.3 First endpoint: create

We have a running gRPC server that implements the desired API. We might want to go a step beyond and have our endpoint do something else than print a pretty message. This is, after all, Chapter 10.

Before we start coding, let’s do a bit of thinking. The compiled code, generated from the Protobuf files, has defined a `Habit` structure. Should we reuse that structure or define a new one ? The answer is quite straightforward here: it's always best to not leak protocol definitions into the core business code, because it'll create problems when we start to add support for other protocols, such as XML or JSON. These data definitions that are used only for describing protocol are called Data Transfer Objects or DTO's for short. Instead, our core business code, often called “domain” or “model”, should have types for every entity that needs to be handled internally. Let’s look at a clean target architecture.

Figure 10.1 Architectural diagram with domain and connectors



For the exact same reasons that we saw in Chapter 8, the Go structures representing the transferable data, here the generated code, must be capable of evolving independently from the rest of the code.

10.3.1 Business layer

In Chapter 8, we created a `session` package with our logic. Create an `internal/habit` folder where we can define our domain `Habit` and the types that it needs. We want to keep the data that was received as input but also remember when the habit was created and give it an ID to find it again. These fields are not part of the input message - we are able to add them here because we’re not re-using the API structure.

Listing 10.12 internal/habit/habit.go: define the business types

```
// ID is the identifier of the Habit.
type ID string

// Name is a short string that represents the name of a Habit.
type Name string

// WeeklyFrequency is the number of times a Habit should happen e
type WeeklyFrequency uint

// Habit to track.
type Habit struct {
    ID              ID #A
    Name            Name
    WeeklyFrequency WeeklyFrequency
    CreationTime    time.Time #A
}
```

It is always good to create a specific type for each of the fields in our main entity, even though the usage might seem more verbose: functions and methods will take typed arguments that will serve as documentation and make the API clearer. For example, if a function takes the name and the ID and both are strings, it’s quite easy to mix them up, while if one is explicitly an ID and the other explicitly a name, casting the name into an ID type should raise a red flag to the developer writing the call.

Requirement: Create a habit

If we look at the requirements, the first thing we need to do is create a `Habit`. We defined some optional values in the Protobuf documentation, meaning that we must complete the fields if needed. The first question here is: is input validation the job of the API layer, or the domain layer? Both solutions make sense for different reasons. We decided that if some new feature needs to create a habit inside our service, it will call the domain directly and we want this `NewHabit` to always return a valid entity. We cannot rely on the API layer to always send what the domain needs.

If you are in a situation where validation on the API layer makes more sense, there are a few libraries out there that can do it for you with a few tags.

What if the input is invalid? Just like HTTP, gRPC uses different status codes to make sense of the response defined by the RPC API. These codes are included in the error that is returned alongside the response, by the endpoint.

Table 10.1 A few gRPC status codes

Code	Number	Description
OK	0	Not an error; returned on success.
INVALID_ARGUMENT	3	The client specified an invalid argument.
NOT_FOUND	5	Some requested entity (e.g., file or directory) was not found.
PERMISSION_DENIED	7	The caller does not have permission to execute the specified operation.
UNIMPLEMENTED	12	The operation is not implemented or is not supported/enabled in this service.
INTERNAL	13	An unspecified error occurred while processing the request.

These are only a few, the rest, with deeper explanations, can be found in the official documentation. You can run `go doc google.golang.org/grpc/codes` to have the list. Well, not exactly: `go doc` limits its output, which causes only the first few codes to be printed. To get the whole list, run:

```
go doc --all google.golang.org/grpc/codes
```

Knowing this, any invalid input will return a code 3 (`INVALID_ARGUMENT`). Should the business layer be in charge of returning such a code 3? Certainly not. Returning this code is the role of the API layer - the domain layer is not even aware that we’re implementing a gRPC server - and it needs to know what happened inside the domain layer. Perfect occasion to use a typed error.

Validate with typed error

Let’s start with the `validateAndComplete` function, in a new `create.go` file dedicated to this business logic. It must check that the name is not empty, set the frequency to one if empty, and also fill up the two internal fields. Arguably, it could be two functions: validate and complete.

Listing 10.13 internal/habit/create.go: validate and complete the entity

```
// validateAndCompleteHabit fills the habit with values that we w
// Returns InvalidInputError. #D
func validateAndCompleteHabit(h Habit) (Habit, error) {
    // name cannot be empty
    h.Name = Name(strings.TrimSpace(string(h.Name))) #A
    if h.Name == "" {
        return Habit{}, InvalidInputError{field: "name", reason:
    }

    if h.WeeklyFrequency == 0 { #B
        h.WeeklyFrequency = 1
    }

    if h.ID == "" { #C
        h.ID = ID(uuid.NewString())
    }

    if h.CreationTime.Equal(time.Time{}) {
        h.CreationTime = time.Now()
    }

    return h, nil
}
```

We now need to define this typed error, in a new `errors.go` file in the `habit` package:

Listing 10.14 errors.go: typed error for invalid input

```
// InvalidInputError is returned when user-input data is invalid.
```

```
type InvalidInputError struct {
    field  string
    reason string #A
}

// Error implements error.
func (e InvalidInputError) Error() string {
    return fmt.Sprintf("invalid input in field %s: %s", e.field,
}
```

We could expose the given value of the field too: it is very useful when we get an error to know what the server actually got - it differs from what we think we sent more often than we care to admit. But this error will be logged, copied around, and malevolent users could send in gigabytes of data and crash our system. There are some ways to avoid this (limiting the size of requests, truncating logs, etc.) but for now, let’s just avoid logging the field.

Testing the validation

We can already write an easy unit test for this `validateAndComplete` function. No need to mock any dependency, what a pleasure!

When writing this test, we found out that each test case had very different assertions, so we chose to write a named function for each. You can write several independent `TestXxx` functions, or group them inside a single one with an explicit name:

Listing 10.15 create_internal_test.go: test `completeHabit`

```
func Test_validateAndFillDetails(t *testing.T) {
    t.Parallel()

    t.Run("Full", testValidateAndFillDetailsFull)
    t.Run("Partial", testValidateAndFillDetailsPartial)
    t.Run("SpaceName", testValidateAndFillDetailsSpaceName)
}
```

The first function, `testValidateAndFillDetailsFull`, checks that if the habit is complete, nothing is changed.

```
func testValidateAndFillDetailsFull(t *testing.T) {
    t.Parallel()

    h := Habit{...all fields are filled...}

    got, err := validateAndCompleteHabit(h)
    require.NoError(t, err)
    assert.Equal(t, h, got)
}
```

The second function checks that if the habit is incomplete, ID and creation time are filled up, the rest did not change. Each run of the test will give us different values, so we are only checking for “not empty”. If you want to be more thorough, you can check that the ID follows a given format, using regular expressions and that the time is within the past second or so.

```
func testValidateAndFillDetailsPartial(t *testing.T) {
    t.Parallel()

    h := Habit{...}

    got, err := validateAndCompleteHabit(h)
    require.NoError(t, err)
    assert.Equal(t, h.Name, got.Name)
    assert.Equal(t, h.WeeklyFrequency, got.WeeklyFrequency)
    assert.NotEmpty(t, got.ID)
    assert.NotEmpty(t, got.CreationTime)
}
```

Then we check the name. If we send only spaces, we should get an `InvalidInputError`. At this point in the development, the exact content of the error might still change, so we just focus on the type.

```
func testValidateAndFillDetailsSpaceName(t *testing.T) {
    t.Parallel()

    h := Habit{Name:"    ",...}

    _, err := validateAndCompleteHabit(h)
    assert.ErrorAs(t, err, &InvalidInputError{})
}
```

Good. Run the test, make sure you are happy about your coverage.

Now let’s call the endpoint. The `Create` function in the business, or domain, layer, will fill the habit and will be ready to save it to a data storage.

Listing 10.16 create.go: business function to create a habit

```
// Create validates the Habit, saves it and returns it.
func Create(_ context.Context, h Habit) (Habit, error) {
    h, err := validateAndFillDetails(h)
    if err != nil {
        return err #A
    }

    // Need to add the habit to data storage...

    return nil
}
```

You can already write a closed-box test for this one, or at least the structure for the test.

We added a “context” as the first parameter, but we ignored it until now. Why? The answer to this is quite simple, and we’ll explain it fully in section 10.6. For now, let’s provide a `context.Context` variable, which, in Go, is almost always called `ctx`.

10.3.2 API layer

Now that we’ve implemented the validation in the domain layer, let’s move back to the server package and update the `CreateHabit` method on the server structure.

What should it do? This is the gRPC layer, where we transform an API-specific signature into domain objects, call the domain function and transform the response back into API-specific types.

Listing 10.17 create.go: API layer

```
// CreateHabit is the endpoint that registers a habit.
func (s *Server) CreateHabit(ctx context.Context, request *api.Cr
    var freq uint #A
    if request.WeeklyFrequency != nil {
        freq = uint(*request.WeeklyFrequency)
    }

    h := habit.Habit{
        Name:            habit.Name(request.Name),
        WeeklyFrequency: habit.WeeklyFrequency(freq),
    }

    createdHabit, err := habit.Create(ctx, h)
    if err != nil {
        ...
    }

    s.lgr.Logf("Habit %s successfully registered", createdHabit.I

    return &api.CreateHabitResponse{
        Habit: &api.Habit{
            Id:              string(createdHabit.ID),
            Name:            string(createdHabit.Name),
            WeeklyFrequency: int32(createdHabit.WeeklyFrequency),
        },
    }, nil #B
}
```

If we want the default value of a habit's frequency to be 1, why are we setting the `freq` to 0 (by using Go's default value) when it is absent? We decided that this default value was a business requirement and not an API definition. It is arguable and can only be decided case by case. Imagine what you expect if you call the domain method with an empty frequency, from somewhere else than the API layer, and act accordingly.

How do we manage the error returned by the domain layer? We made sure that if the error is caused by a bad input, it will have a specific type. We can use `errors.As` to cast it into the `InvalidInputError` type and check whether we should return a code 3. To be perfectly honest, we could use `errors.Is` instead because we are not using any field of method specific to the type `InvalidInputError`, but we chose to show you how `As` can be used.

But what if we receive something that is not an `InvalidInputError`? After all, our future implementation of the endpoint logic might have to face database calls, which could cause errors that would not be due to an input message validation.

A rule of thumb to remember, when implementing a gRPC endpoint, is that every return statement should either return a `nil` error, or an error built from the `status.Error` function (or `Errorf`). The `status` package is a neighbour of the `codes` package : `google.golang.org/grpc/status`. When in doubt of which error code we should return, the default choice of `codes.Internal`.

Listing 10.18 create.go: error management

```
got, err := habit.Create(ctx, h)
if err != nil {
    var invalidErr habit.InvalidInputError
    if errors.As(err, &invalidErr) {
        return nil, status.Error(codes.InvalidArgument, invalidEr
    }
    // other error
    return nil, status.Errorf(codes.Internal, "cannot save habit 
}
```

When a service ends up having several endpoints, checking the error and outputting the appropriate status code can be factored in a single function, `toAPIErrorf(err error, format string, args ...any)`. Feel free to implement it when the need arises.

Time to test our server manually.

Hand testing

The tool we used 2 chapters ago to call our service, `curl`, only does HTTP calls, but it has a cousin, `grpcurl`, which does the same job. There are alternative options to `grpcurl` - many providing a graphical user interface, but this one is the one we find most convenient. If you fancy a nice GUI, Postman supports gRPC and can send Protobuf messages to servers since its version 10.

First, start your server with `go run cmd/main.go`.

Next, you can install `grpcurl` with the following command:

```
go install github.com/fullstorydev/grpcurl/cmd/grpcurl@latest
```

Now, we can start using the tool to send requests to our server. There is a major difference between `curl` and `grpcurl`: the format of the message was, for `curl`, a regular JSON document, whereas for `grpcurl`, we need to provide a valid Protobuf entity. If you remember the beginning of this chapter, Protocol Buffers have indexed fields, which means the message we'll send via `grpcurl` will need to be properly written, with its fields in the correct positions. There are two options for us here - we can either provide the proto definition to `grpcurl`, or we could have it ask for that definition from the server. The second option is called reflection, and we won't be using it here. Indeed, reflection adds a small overhead to our server - something that usually we don't want to ship to production.

So, here is how we tell `grpcurl` how to structure our query (and understand the response): we simply pass it the proto files with the `-proto` parameter - we'll give it the `service.proto` file, as this is where the definition of the endpoints lie. Since some of the files include other files, we need to specify the "root" from which they refer, via the `-import-path` parameter. Finally, we need to tell which endpoint we want to aim at. This is passed as the final parameter of the request - in the form of `{package}.{service}/{endpoint}`.

Here is the command line that we are able to run:

```
grpcurl \
-import-path /path/to/learngo-pockets/habits/api/proto/ \ #A
-proto service.proto \ #B
-plaintext -d '{"name":"clean the kitchen"}' \ #C
localhost:28710 \ #D
habits.Habits/CreateHabit #E
```

If everything went fine, you should receive a response from the server (formatted in JSON). Does it contain an ID field? Is the weekly frequency set?

Did you also try with an "invalid" name for the habit?

Check the server's output - it should be logging a message every time a request is received. If your tests are conclusive, it's time to do a bit more than logging in our endpoint!

10.3.3 Data retention

The service tells its clients that it can create habits, but it doesn’t store them. We need to fix that.

Repository package

For the first version, we can use the same kind of in-memory repository that we used for games in Chapter 8, in a package called `internal/repository`. It has the same drawbacks: unscalable, probably unstable very soon, but it gives us something quickly, so it is ok for a proof of concept.

Write a `Repository` structure with a `New` function that builds it and initialises its map of data. Similarly to the `New` function of the `server` package, we want to inject a `Logger` in here too. For now, we will need one method on the `Repository` type, `Create`; but soon we’ll want to add `List`, which will return all the contents of our database.

If you have followed us through nine chapters, you should be able to create the package, expose the right functions, structures and methods, and of course cover them with some tests. Do not forget to add a mutex to lock the data when reading and writing on the repository storage.

```
$ go doc
package repository // import "learngo-pockets/habits/internal/rep

Package repository accesses the habits data.

type Error string
type Logger {...}
type HabitRepository struct{ ... }
    func New(lgr Logger) *HabitRepository

$ go doc HabitRepository
package repository // import "."

type HabitRepository struct {
        // Has unexported fields.
}
    HabitRepository holds all the current habits.
```

```
func New(lgr Logger) *HabitRepository
func (hr *HabitRepository) Create(_ context.Context, habit habit.
func (hr *HabitRepository) ListAll(_ context.Context) ([]habit.Ha
```

As you can see, by having a `context.Context` parameter in each of our methods, we anticipated that when we replace this with a real database, we will need a context to stop looking for data when a client interrupts the call.

If you need it, remember that an example of the code can be found in the book’s repository.

Dependency injection

Now, we didn’t really test this `repository` package. The main reason here is that all we do in it is write to a map, and that we list all values of a map. We can add the call to `Add` inside the domain function. Here is a flow of the call from the client to the database. You can imagine the same flow back with either errors or nils.

We have implemented the api-to-domain connection in the server package, but we lack the right-hand calls. For that, we need the server to have an instance of the Repository connector, and we need the domain’s `Create` function to expect a small interface with `Add` in it.

Let’s first inject a repository dependency to the server. But why don’t we simply call `repository.New()` in the server, rather than doing it in the `main` function? As we’ll see, this makes tests a lot simpler than having to rely on a hardcoded implementation of that dependency. This is one of Go’s best usages of its lightweight interfaces. We are using an interface here so that tests for the server can use mocks.

Listing 10.19 server.go: Adding a repository connector to the server

```
// Server is the implementation of the grpc server.
type Server struct {
    db Repository
    lgr Logger
}

type Repository interface { #A
```

```
    Add(ctx context.Context, habit habit.Habit) error
    FindAll(ctx context.Context) ([]habit.Habit, error)
}

// New returns a Server that can Listen.
func New(repo Repository, lgr Logger) *Server {
    return &Server{
        db: repo,
        lgr: lgr,
    }
}
```

Update the `main` function to comply with this new signature of `New` - we need to pass an entity that implements that interface, such as the output of `repository.New(...)`.

Second, the `Create` endpoint on the domain needs to take an interface as parameter, also for stubbing mocking purposes, but also to reduce the scope of problems: by using an interface with only the `Add` method, we assure that `Create` cannot use any other future method and mess with the logic. Imagine that you observe in your logs that calls to `FindEverything` are messing with the performance of the service. You know by seeing this interface that the culprit is not `Create` and you can move on.

Listing 10.20 create.go: Call the repository in the logic function

```
type habitCreator interface {
    Add(ctx context.Context, habit Habit) error
}

// Create adds a habit into the DB.
func Create(ctx context.Context, db habitCreator, h Habit) (Habit
    h, err := validateAndCompleteHabit(h)
    if err != nil {
        return Habit{}, err
    }

    err = db.Add(ctx, h)
    if err != nil {
        return Habit{}, fmt.Errorf("cannot save habit: %w", err)
    }

    return h, nil
}
```

Here we are, right? Can you see in the logs that your call goes all the way to the DB? Let’s write a couple of tests, to ensure we properly catch the errors. For this, we’ll start with a simple stub, as we did in Chapter 6, to implement the `habitCreator` interface.

But how can we update the tests of `Create` to make sure that `Add` is properly called? That’s what we are going to see in the next part.

10.4 Unit testing with generated mocks

In the last chapters, we have seen how you can write your own stubs when the interface is small enough and the logic simple. There are a few libraries out there capable of taking an interface and generating mocks instead.

Stubs vs Mocks

Stubbing and mocking are two very common ways of making use of an interface for tests. While stubbing consists in writing a structure that implements the interface and returns “hard-coded” values, in order to test the behaviour of your code when the stubbed dependencies returns this or that, mocking adds on top a check on how many times each dependency was called, and if it was with the correct parameters.

10.4.1 Generate mocks

The best known libraries are mockgen, `mockify` and `minimock`. They are based on different design decisions, so feel free to pick your favourite. In our example, we chose `minimock` because it provides mocked functions with typed parameters.

```
go install github.com/gojuno/minimock/v3/cmd/minimock@latest
```

Because the mocks are generated, this is a perfect occasion to use the `go:generate` syntax. Pick an interface, e.g. `habitCreator`, and add this line above:

```
//go:generate minimock -i habitCreator -s "_mock.go" -o "mocks"
```

We are asking `minimock` to generate a mock for the interface (`-i`) `habitCreator`, in a file with a specific suffix (`-s`) and in a specific output folder (`-o`). Create that folder before you can continue: from the root of the module, it will be `internal/habit/mocks`.

In your favourite terminal, navigate to the `habit` package and run

```
$ go generate .
```

or alternatively, navigate to the root of the module and run all the generate commands in the project with

```
$ go generate ./...
```

You can see a new file has appeared in the mocks folder. Check the contents with `go doc`.

```
$ go doc internal/habit/mocks
package mocks // import "learngo-pockets/habits/internal/habit/mo

type HabitCreatorMock struct{ ... }
    func NewHabitCreatorMock(t minimock.Tester) *HabitCreatorMock
type HabitCreatorMockAddExpectation struct{ ... }
type HabitCreatorMockAddParams struct{ ... }
type HabitCreatorMockAddResults struct{ ... }
```

10.4.2 Use the mocks

The closed-box test for `Create` does not compile anymore. Let’s fix it.

First, there are two imports that we need to add. One is pretty obvious but the second calls for a little explanation.

```
import (
    // ...
    "learngo-pockets/habits/internal/habit/mocks"
    "github.com/gojuno/minimock/v3"
)
```

The first import is here to access the mocks we just generated. The second, on the other hand, is about the `minimock` library.

If you pay extremely close attention, you’ll realise that the second import’s path ends with “`/v3`”. Are we really importing a package named `v3`? This would be a very strange name for a package…

Versioning modules in Go

Sometimes, a module needs to go through heavy changes that make the new version incompatible with the previous one. Interrupting the backward compatibility of a module requires a version change. When this happens, the `go.mod` file should be updated to reflect the version: the first line of minimock’s `go.mod` is `module github.com/gojuno/minimock/v3`.

Users who want to use `minimock` (or any other versioned module) have to remember to specify the version they want to use in the import path, right after the name of the module, for instance : `import "github.com/jackc/pgx/v5"` and `import "github.com/jackc/pgx/v5/pgxpool"`. When using functions or types defined in these packages, ignore the “`/v5`” part : `pgx.Connect(...)` or `pgxpool.New(...)`.

But why do we need to use the `minimock` library? Wouldn’t the generated code contain enough tools? It turns out it doesn’t. Indeed, the generated `mocks` package exposes a `NewHabitCreator(...)` function, which returns the type we want - `HabitCreatorMock`, an implementation of the `habitCreator` interface. But this function’s parameter is of type `*minimock.Controller`. Don’t be scared, there is no other package to import.

Next, we can now define a function that builds a mock for each of the test cases. It takes a controller and returns a mocked instance of the required interface, `habitCreator`. The test case structure will hold a new field whose type is a function - to be honest, if you look at the test with error cases that we have in the book’s repository, you will see that it would be far easier to read if we had written it as two separate functions, but we wanted to show you how functions can make your life better as fields of a test case struct.

The controller is created at the start of the test, it can be shared by all the test cases, and, if your version of minimock is recent enough (v3.3.0), it automatically registers a check at the end of the test to ensure each expected call was met with an actual call.

In the nominal test case, or happy flow, the mock should take the input habit, previously declared as a variable called `h`, and return no error.

Listing 10.21 create_test.go: Add a mock function to each test case

```
tests := map[string]struct {
    db          func(ctl *minimock.Controller) *mocks.HabitCreato
    expectedErr error
}{
    "nominal": {
        db: func(ctl *minimock.Controller) *mocks.HabitCreatorMoc
            db := mocks.NewHabitCreatorMock(ctl)
            db.AddMock.Expect(ctx, h).Return(nil) #A
            return db
        },
        expectedErr: nil,
    },
}
```

Finally, we can plug this into the `TestCreate` function.

Listing 10.22 create_test.go: use the mock when testing

```
t.Run(name, func(t *testing.T) {
    t.Parallel()

    ctrl := minimock.NewController(t)
    defer ctrl.Finish() #A

    db := tt.db(ctrl)

    got, err := habit.Create(ctx, db, h)
    assert.ErrorIs(t, err, tt.expectedErr)
    if tt.expectedErr == nil {
        assert.Equal(t, h.Name, got.Name)
    }

}
```

It runs and succeeds. You can commit to make sure you don’t forget this state, before playing around with the mocks. For example, what happens if you comment out the call to `Add`? Your test should tell you.

You can also read the documentation of the mocks package and how the minimock tool can best be used, in order to define your favourite style.

There is one more way of testing that is perfect for this kind of CRUD service: integration testing.

10.5 Integration testing

Where unit tests focus on one function, integration tests will check the behaviour of the entire service, with scenarios. Here are example scenarios for our service:

Scenario 1: Add and list

- Add a habit: walk in the forest 3 times a week
- Add a habit: water the plants twice a week
- List the habits: check the 2 habits that we get
- Add a habit: read a book 5 times a week
- List the habits: check the 3 habits that we get

Scenario 2: Add and delete

- Add a habit: walk in the forest 3 times a week
- List the habits: check the 1 habit that we got
- Add a habit: no name, expected error with code 3
- Add a habit: water the plants twice a week
- List the habits: check the 2 habits that we got
- Remove the first habit
- List the habits: check that we still have the second

And so on. Some people will intertwine this with API testing, others will separate testing the flows from testing the gRPC response for each endpoint’s error cases.

In order to test a whole flow, we need to be able to list the habits that we saved. Then, we will write the first scenario.

10.5.1 List habits

Adding an endpoint requires a few additions, but it should be quick enough:

1. Update the Protobuf file with a `ListHabits` endpoint. First, because this way you can already publish the interface for the rest of your team to use and mock.

Listing 10.23 service.proto: add the list endpoint

```
// Habits is a service for registering and tracking habits.
service Habits {
  // CreateHabit is the endpoint that registers a habit.
  rpc CreateHabit(CreateHabitRequest) returns (CreateHabitRespons

  // ListHabits is the endpoint that returns all habits.
  rpc ListHabits(ListHabitsRequest) returns (ListHabitsResponse);
}

// ListHabitsRequest is the request to list all the habits saved.
message ListHabitsRequest { #A
}

// ListHabitsResponse is the response with all the saved habits.
message ListHabitsResponse {
  repeated Habit habits = 1; #B
}
```

From there you can regenerate the corresponding go files with `go generate ./...`

2. Add the logic in the domain layer, following the pattern of `internal/habit/create.go`. Don't forget to use a tiny interface for the database, generate a mock for it, and test thoroughly. Note that it is not always necessary to have a mock framework to test. An alternative using a simple function replacing the database behaviour is to have a structure holding the output content you want to mock and have the database call returning it.

Listing 10.24 list_test.go: Test List without minimock

```
// MockList is a mock for FindAll method response.
```

```
type MockList struct { #A
    Items []habit.Habit
    Err   error
}

// FindAll is a mock which returns the passed list of items and e
func (l MockList) FindAll(context.Context) ([]habit.Habit, error)

func TestListHabits(t *testing.T) {

// TODO: Write the needed content for the tests cases

        "empty": {
            db:             MockList{Items: nil, Err: nil}, #B
            expectedErr:    nil,
            expectedHabits: nil,
        },
        "2 items": {
            db:             MockList{Items: habits, Err: nil},
            expectedErr:    nil,
            expectedHabits: habits,
        },
        "error case": {
            db:             MockList{Items: nil, Err: dbErr},
            expectedErr:    dbErr,
            expectedHabits: nil,
        },

    for name, tc := range tests {
        name, tc := name, tc

        t.Run(name, func(t *testing.T) {
            t.Parallel()

            got, err := habit.ListHabits(context.Background(), tc
            assert.ErrorIs(t, err, tc.expectedErr)
            assert.ElementsMatch(t, tc.expectedHabits, got)
        })
    }
}
```

3. If you don't have one already, write the repository function that lists all the saved habits. The repository should return a deterministic list of habits, sorted using a specific criterium such as the creation date of the habits.

Listing 10.25 memory.go: deterministic output of habits

```
// FindAll returns all habits sorted by creation time.
func (hr *HabitRepository) FindAll(_ context.Context) ([]habit.Ha
    log.Infof("Listing habits, sorted by creation time...")

    // Lock the reading and the writing of the habits.
    hr.mutex.Lock()
    defer hr.mutex.Unlock()

    habits := make([]habit.Habit, 0)
    for _, h := range hr.habits {
        habits = append(habits, h)
    }

    // Ensure the output is deterministic by sorting the habits.
    sort.Slice(habits, func(i, j int) bool { #A
        return habits[i].CreationTime.Before(habits[j].CreationTi
    })

    return habits, nil
}
```

4. Add the `ListHabits` method to the service, following the pattern of `internal/server/create.go`. Isolate the transformation of the generated structure into the domain structure in a separate function, unit-test it too. We decided that a repository containing no habits shouldn't be a problem and return an error, but feel free to make a different choice here. Also, think about determinism: if the repository contains two elements, should they always be returned in the same order by the endpoint? Determinism is very important, and we can only recommend enforcing it wherever possible. Testing deterministic endpoints is a lot simpler than testing non-deterministic endpoints!

5. Test manually with `grpcurl` or a similar tool.

Now that you trust that this new endpoint works the way you expect, we can write an integration test.

10.5.2 Integration with go test

We want to write a test that will go through every layer of the service, all the way to the network outside of it. Considering that our database is currently a hacky in-memory thing, there is no point in mocking it, but when we finally use a real database system, it will be necessary to either mock it or run an instance locally.

This test will run the service for real and call it as any client would.

Run a service

First, create a test file `internal/server/integration_test.go`. In it, add a `TestIntegration` function. There are several places an integration test file can be stored; since we are testing the API of the server, we placed it close to it.

First, we create a gRPC server instance and register it - something similar to what we already have in the `main` function. Second, we create a listener - by giving an empty string as the address parameter, we ask it to find a free port on the host and use it. Third, we run that server in a parallel thread, so that the rest of the test can keep running and calls can be made to it. Of course, we need the server to stop at the end of the test, whenever that is.

If we need to write any “utility” function, we can start their implementation with `t.Helper()`. This will tell the Go test suite to ignore this layer when an error is surfaced.

Listing 10.26 integration_test.go: start the service

```
func TestIntegration(t *testing.T) {
    grpcServ := newServer(t)
    listener, err := net.Listen("tcp", "")
    require.NoError(t, err)

    wg := sync.WaitGroup{}
    wg.Add(1)
    go func() {
        defer wg.Done()
        err = grpcServ.Serve(listener)
        require.NoError(t, err)
    }()
    defer func() {
        // terminate the GRPC server
        grpcServ.Stop()
        // when that is done, and no error were caught, we can en
        wg.Wait() #A
    }()

}

func newServer(t *testing.T) *grpc.Server {
    t.Helper() #B
s := server.New(repository.New(t), t) #C


    return s.registerGRPCServer()
}
```

The server is running, a client can enter the scene.

Create a client

A client needs to know to which address to send its requests and what the shape of the server is (what the endpoints are). We create a function to build that new client - it takes the address as a parameter.

Note that we need to pass some credentials to connect to the server. The `grpc` library kindly offers a function that generates credentials that disable transport security (TLS). While this is usually a security breach, we're running our server in a very restricted environment, and we can accept not having to pass credentials. Depending on which network the request will be sent through, you might have to use credentials, or you might be able to use the `insecure` package to generate some for you.

Listing 10.27 integration_test.go: create a client

```
func TestIntegration(t *testing.T) {
    ...
    // create client
    habitsCli, err := newClient(t, listener.Addr().String())
    require.NoError(t, err)
}

func newClient(t *testing.T, serverAddress string) (api.HabitsCli
    t.Helper()
    creds := grpc.WithTransportCredentials(insecure.NewCredential
    conn, err := grpc.Dial(serverAddress, creds)
    if err != nil {
        return nil, err
    }

    return api.NewHabitsClient(conn), nil
}
```

The scene is set, we can start the scenario.

Run scenario

As we are only testing two endpoints, we can create a function for the happy path for each of them. Then we create an error path function for `CreateHabit`, because `ListHabits` never returns a business error.

Here is an example with the list habits endpoints, which is the trickier one: it returns generated IDs whose value will change at every run. So, we overwrite it after checking it’s been filled.

Listing 10.28 integration_test.go: function to list a Habit

```
func listHabitsMatches(t *testing.T, habitsCli api.HabitsClient,
    list, err := habitsCli.ListHabits(context.Background(), &api.
    require.NoError(t, err)


    for i := range list.Habits {
        assert.NotEqual(t, "", list.Habits[i].Id)
        list.Habits[i].Id = "" #A
    }
    assert.Equal(t, list.Habits, expected) #B
}
```

Consider the option of basing your integration test on a struct that holds the client as a field and can hold methods that wrap calls to the client, to make your test easier to read. We decided to use functions only, but all of them will start with the same 2 arguments, which can become very verbose - and is usually a cue for refactoring.

With this kind of helper functions, the scenario can look fairly readable:

Listing 10.29 integration_test.go: scenario in the code

```
// add 2 habits
addHabit(t, habitsCli, nil, "walk in the forest")
addHabit(t, habitsCli, ptr(3), "read a few pages")
addHabitWithError(t, habitsCli, 5, "        ", codes.InvalidArgum

// check that the 2 habits are present
listHabitsMatches(t, habitsCli, []*api.Habit{...})

// ...

func ptr(i int32) *int32 {
    return &i
}

func addHabit(t *testing.T, habitsCli api.HabitsClient, freq *int
    _, err := habitsCli.CreateHabit(context.Background(), &api.Cr
        Name:            name,
        WeeklyFrequency: freq,
    })
    assert.NoError(t, err)
}
```

Make sure you isolate your different scenarios so that they can run in parallel - you can even run that many instances of the server in parallel, one for each integration scenario. You can also use this opportunity to play with concurrency and call `Add` a large number of times concurrently to check for performance.

Using test.short to only run lightweight tests

So far, our test resembles any other unit test that we’ve written - apart that it’s called “integration”. Sometimes, these integration tests can be quite intense, because they go through lots of features and cases, or because they include some benchmarks or run some load or performance tests. These tests usually take quite some time, and it isn’t advised to include them in continuous integration toolchains, as they might slow down the delivery process. For instance, it could be a requirement to have “light” tests run on pull requests, but “heavy” tests to run on tagging and image building. The `go test ./...` command accepts a `-short` flag. Setting this flag in the command line will change the output of the `testing.Short()` function, which we can invoke in any test.

Let’s start our TestIntegration with a check on that flag:

```
func TestIntegration(t *testing.T) {
    // Skip this test when running lightweight suites
    if testing.Short() {
        t.Skip()
    }

grpcServ := newServer(t)
```

Now, let’s have a look at the output of `go test -v ./...` . Using the -v flag makes the output verbose and lists each test function called. The output should contain `TestIntegration`:

```
> go test -v ./...
...
ok      learngo-pockets/habits/internal/habit    1.004s
=== RUN   TestIntegration
    create.go:17: Create request received: name:"walk in the fore
...
--- PASS: TestIntegration (0.00s)
...
```

Let’s try the same, but this time with the short flag: `go test -v -short ./...` . This time, the output should explicitly indicate that `TestIntegration` was skipped:

```
> go test -v -short ./...
...
ok      learngo-pockets/habits/internal/habit    1.004s
=== RUN   TestIntegration
    integration_test.go:23:
--- SKIP: TestIntegration (0.00s)
...
```

We now have a way of running only the “lightweight” tests in our CI pipelines - simply, when running a more expensive or consuming one, check for that `-short` flag. Its presence should be an indicator that we want a quick result.

Now, so far, we’ve been using our own in-memory database, which we quite trust. Indeed, if that database were to be unavailable, we’d have serious issues in our server itself, since the server and the database are part of the same program. But most of the time, the database is a remote entity, one that could behave erratically, because of network issues, or external load, or lots of other pesky bugs - or, worse, what if our own query crashes that database? We already handle the case when the database returns an error, but what if it doesn’t answer our query? How long should we wait before realising something is wrong?

10.6 Getting the best out of the context

When relying on remote services, a good practice is usually to allow the callee to provide a response within a specific time frame. There are two ways of expressing a time limit - either by providing a timeout, just as Mission Impossible’s “This message will self-destruct in 5 seconds”, or by providing a deadline, “You have until Friday, 11 AM”. The choice of which one to use depends on the activity being performed, but the former is more common than the latter, when it comes to calls to remote network entities.

10.6.1 What is a context?

Earlier in the chapter, we haven’t detailed what a `Context` really is, but now is the time to run `go doc context.Context` to find out. As we can read there, the purpose of a context is to carry around deadlines, cancellation signals, and values across API boundaries.

The documentation also tells us that a `Context` is an interface with the following methods:

```
Value(key any) any
Deadline() (deadline time.Time, ok bool)
Done() <-chan struct{}
Err() error
```

Before we go any further, it’s important to note that even though `context.Context` is an interface, it is one of the few that you should never need to implement.

The `Value` method available of a context is here to implement the “carrying values across API boundaries” requirement. While a context can be seen as a key-value storage, we highly recommend you don’t think of it that way. If you place important values inside context, then it's not clearly visible anymore what input different functions require. It's better to only stick to sending non-critical data, such as monitoring identifiers, or request identifiers in context. Business values shouldn’t be passed via the context. If you’re thinking of this as an option, we recommend going for an alternative. We’ll see an example below where the context’s storage feature is used.

The `Deadline` method returns when a context’s deadline is set - and if it is. The deadline is the time when the context will start saying it’s reached its expiration date to whomever might ask.

The `Done` method returns a channel that will be closed when the context has reached its end. Calling `Done()` is simpler than comparing `Deadline()` with `time.Now()`, so this is what is usually done.

Finally, the `Err` method returns an error describing why the channel returned by `Done` is closed, or `nil` if it isn’t closed yet.

Now, let’s have a look at how to create contexts.

10.6.2 Create a context

Golang’s `context` package offers several functions that allow for the creation of a context. They are mostly divided into two categories: those that create a child context from a parent one, and those that spring one out of the blue.

This latter set contains only `context.Background()` and `context.TODO()`. We recommend always creating your application’s context in your `main` function, and passing it around to any dependency that might need it. Create it with the `Background` method, and try to avoid calling `TODO`. Overall, your application should have a single parent context.

Now that we’ve got a context, we can create children. They can come in a variety of shapes, but the main difference lies in how we want to set their `Deadline` property. We can call `WithDeadline` to provide a specific timestamp at which time the child will be cancelled, or `WithTimeout` if we want to specify how long a context should “live”. The second option is by far the most common when it comes to making calls to remote services.

Most of the time, however, a function will receive a context as one of its parameters. There is a silent convention to always provide the context as the first parameter of a call to a function - and, just as we usually name the errors `err`, we similarly very often call our contexts `ctx`. And when a function receives a context, it should not try to create a new one with `Background()` or `TODO()`. In our gRPC generated code, we can observe that the endpoints’ signatures all start with an incoming context - that’s the one we should be using.

While it might seem repetitive to always have to pass the context as a parameter, we strongly encourage you to resist the urge of using composition to save a context into a variable.

10.6.3 Using a context

When should we create a child context, though? Why not provide the parent context - after all, it might have a deadline itself… The answer here is having good practices in coding. It’s about controlling with precision every call that is made across your network. If a service needs to call two other services to answer a request, we want to know which of these two is taking an awfully long time to return a response, if any. To achieve this, each remote call must be using its own context, with its own deadline. Depending on your application, a timeout value can range between 10 milliseconds and some days. Don’t be too strict and take into account network latency.

Let’s have an example with our database. Let’s say that we want to ensure that the repository call in our `Create` endpoint doesn’t take too long. We do this by creating a context with a timeout of 100 milliseconds, and we call our repository.

Listing 10.30 create.go: Adding a context around the db call

```
// Create adds a habit into the DB.
func Create(ctx context.Context, db habitCreator, h Habit) (Habit
    h, err := validateAndCompleteHabit(h)
    if err != nil {
        return Habit{}, err
    }

    dbCtx, cancel := context.WithTimeout(ctx, time.Second) #A
    defer cancel()
    err = db.Add(dbCtx, h) #B
    if err != nil {
        return Habit{}, fmt.Errorf("cannot save habit: %w", err)
    }

    return h, nil
}
```

If we run this, everything works fine. But it’s not tested, and we should test it. It’s even worse than not tested - it actually breaks our existing tests! Indeed, if you remember, we are using mocks, and mocks are very strict about what they expect. Our tests didn’t worry too much about the context - after all, we didn’t fiddle with it so far. But now, the context used to call `Add` in the `Create` endpoint is not the `Background` one any more. We should update the test, but how can we provide the exact same context? We would need to know exactly when the deadline is to be able to expect it properly.

Many mocking libraries face this issue at some point. Minimock has decided to expose a `minimock.AnyContext` variable that will match any `context.Context` variable. Some other mocking libraries go a step beyond, and propose a `mock.Anything` variable that can be used as a wildcard for any input parameter. We only need to use the context, so we’ll limit ourselves to this option.

Listing 10.31 create_test.go: Mocking the child context

```
func TestCreate(t *testing.T) {
    // ...
    "nominal": {
        db: func(ctl *minimock.Controller) *mocks.HabitCreatorMoc
            db := mocks.NewHabitCreatorMock(ctl)
            db.AddMock.Expect(minimock.AnyContext, h).Return(nil)
            return db
        },
        expectedErr: nil,
    },
    "error case": {
        db: func(ctl *minimock.Controller) *mocks.HabitCreatorMoc
            db := mocks.NewHabitCreatorMock(ctl)
            db.AddMock.Expect(minimock.AnyContext, h).Return(dbEr
            return db
        },
        expectedErr: dbErr,
    },
```

While this fixes the current tests, it doesn’t test the current feature of having a timeout on our database call. For this, we will need to improve our mock.

As you now know, a context is something that can reach its deadline, and, when this happens, the channel returned `Done()` is closed - which means reading from it starts returning a zero value instead of being a blocking call. This is how applications check for a cancelled / expired timeout. The following piece of code is present in various forms in most libraries that handle timeouts:

```
select {
// Read from channel used by backend to communicate response
case response := <-responseChan:
    return response, nil
// Check for deadline
case <-ctx.Done():
    return nil, ctx.Err()
}
```

In this select, whichever happens first causes the function to return - either we received a response, or the deadline was met. The line `case <- ctx.Done():` appears more than 60 times in the standard library alone - and it’s always followed by returning the cause of the deadline, via `ctx.Err()`.

Let’s add a test that implements this logic. For this, we can’t use `Expect`, as this immediately returns the specified values. Instead, we’ll have to overwrite the behaviour of the `Add` method, which minimock allows with the `Set` method:

Listing 10.32 create_test.go: Testing the timeout

```
func TestCreate(t *testing.T) {
    // ...
    "db timeout": {
        db: func(ctl *minimock.Controller) *mocks.HabitCreatorMoc
            db := mocks.NewHabitCreatorMock(ctl)
            db.AddMock.Set(
                func(ctx context.Context, habit habit.Habit) erro
                    select {
                    // This tick is longer than a database call
                    case <-time.Tick(2 * time.Second):
                        return nil
                    case <-ctx.Done():
                        return ctx.Err()
                    }
                })
            return db
        },
        expectedErr: context.DeadlineExceeded, #A
    },
```

So, as we've seen, contexts can be used to detect unexpectedly long remote calls. Some functions allow you to register pairs of key-values inside a context, but we recommend keeping that option as a last resort.

Instead, let's resume our habits server, and implement the final endpoint - one that allows us to keep track of what we do on a weekly basis.

10.7 Track your habits

Congratulations, at this point of the chapter, you know all the basics about gRPC in Go! This next section is a guided exercise where you will prove your autonomy and test your aggregated knowledge. If you encounter any struggle, you can find all the code in the repository.

First, let's quickly reset the final goal of this pocket project and define what tracking a habit means. We have the possibility to create a list of habits with a target weekly frequency - how about being able to tick one of our habits whenever we achieve it and retrieve the current status, so we can plan the rest of the week? Am I done for the week? Should I block a time slot to go for a walk? All these questions will you be able to answer!

We will build the following scenario:

1. Create several habits
2. List the created habits
3. Tick the habits that you achieved
4. Get the status of the habits

We are missing the bricks to fulfil step 3 and 4 so let’s go for the implementation. On the API, we will need two new endpoints `TickHabit` and `GetHabitStatus`.

10.7.1 Tick a habit

Let’s define a new endpoint `TickHabit` on the proto side with its associated request and response.

Listing 10.33 service.proto: TickHabit definition

```
  // TickHabit is the endpoint to tick a habit.
  rpc TickHabit(TickHabitRequest) returns (TickHabitResponse);

// TickHabitRequest holds the identifier of a habit to tick it.
message TickHabitRequest {
  // The identifier of the habit we want to tick.
  string habit_id = 1;
}

// TickHabitRequest is the response to TickHabit endpoint.
// Currently empty but open to grow.
message TickHabitResponse {   #A
}
```

Do not forget to regenerate the Go library!

Then, add the implementation on the server with the following signature:

```
TickHabit(ctx context.Context, request *api.TickHabitRequest) (*a
```

Ticks and habits being different notions, we would store them in different tables in an SQL database. In our memory implementation, we will store the ticks in a structure of its own, next to the habits. This will allow us, if we develop a UI, to retrieve only the list of habits or the full status of a habit for a week.

10.7.2 Store ticks per week

This is the same logic that we did previously for the habits in memory, the only tricky part is the data definition. We want to store all the ticks for each habit, and because we want to get a weekly status, we will store ticks grouped by week. The built-in Go library package provides a very useful method named `ISOWeek()` that returns the ISO 8601 year and week number in which that time occurs. Running `go doc time.ISOWeek` returns:

```
func (t Time) ISOWeek() (year, week int)
```

We will use the naming `ISOWeek` in our code. Let's create a new package called `isoweek` and a new file where we will define a `ISO8601` structure holding a `Year` and a `Week`.

Listing 10.34 isoweek/isoweek.go: ISO8681 structure

```
package isoweek

// ISO8601 holds the number of the week and the year.
type ISO8601 struct {
    Year int
    Week int
}
```

It is now easy to define the data storage type in `Repository`. In order to retrieve the current status of a habit for the current week, our storage will store for each habit id a map of the isoweek with its associated events which are timestamps:

```
storage map[habit.ID]map[isoweek.ISO8681][]time.Time
```

For more readability, we chose to have a custom type `ticksPerWeek` which is a map holding all the timestamps per ISO Week. It will now be very easy to retrieve the current status of a habit at the current time. If you want to extend the project, you can even have an endpoint retrieving the status for a given week or date.

Let's add a new type of storage to `HabitRepository` and rename `db` into `habits` to be more explicit.

Listing 10.35 repository/memory.go: ticks storage

```
// ticksPerWeek holds all the timestamps for a given week number.
type ticksPerWeek map[isoweek.ISO8601][]time.Time

// HabitRepository holds all the current habits.
type HabitRepository struct {
    habits map[habit.ID]habit.Habit #A
    ticks  map[habit.ID]ticksPerWeek #B
}
```

Do you feel confident in creating the needed methods? But let’s not anticipate and create only the `Add` for the moment. All the logic of `ISO8601` computation is done in a dedicated function and pass it directly as a parameter, it will be easy to reuse over the other endpoints and tests.

```
AddTick(_ context.Context, id habit.ID, t time.Time) error
```

Do not forget to verify if the habit and the ISOWeek exist in the storage before inserting a new tick.

The full implementation of `Tick` on the domain side should now be ready!

Listing 10.36 habit/tick.go: Tick implementation

```
package habit

import (
    "context"
    "fmt"
    "time"
)

//go:generate minimock -i habitFinder -s "_mock.go" -o "mocks"
type habitFinder interface {
    Find(ctx context.Context, id ID) (Habit, error)
}

//go:generate minimock -i tickAdder -s "_mock.go" -o "mocks"
type tickAdder interface {
    AddTick(ctx context.Context, id ID, t time.Time) error
}

// Tick inserts a new tick for a habit.
```

```
func Tick(ctx context.Context, habitDB habitFinder, tickDB tickAd
    // Check if the habit exists.
    _, err := habitDB.Find(ctx, id) #A
    if err != nil {
        return fmt.Errorf("cannot find habit %q: %w", id, err)
    }

    // AddTick adds a new tick for the habit.
    err = tickDB.AddTick(ctx, id, t)
    if err != nil {
        return fmt.Errorf("cannot insert tick for habit %q: %w",
    }

    return nil
}
```

We now just have to call it on the server side and transform the request and the response. Since we need to provide a timestamp when the habit was ticked, we could either have it passed by the caller or the gRPC endpoint. If the value is set in the server layer rather than read from the request, we need to remember that the server and the client might be in different timezones, and that the "current day" is only a relative notion.

Wait! What happens if I try to tick a habit that does not exist?

10.7.3 Handle corner cases

If the habit does not exist in the habits repository, we do not want to have inconsistent data and store a new tick for an unknown habit. Let's create a `Find` method on the habit repository:

```
Find(ctx context.Context, id habit.ID) (habit.Habit, error)
```

A good practice is to create a custom error that is checked on the server side to return the proper gRPC code. Here, we chose to switch on the domain error and convert in `codes.NotFound` for example if the habit does not exist in the database.

```
    err := habit.Tick(ctx, s.db, s.db, habit.ID(request.HabitId),
    if err != nil {
        switch {
        case errors.Is(err, r.ErrNotFound):
            return nil, status.Errorf(codes.NotFound, "couldn't f
        default:
            return nil, status.Errorf(codes.Internal, "cannot tic
        }
    }
```

You can test the endpoint manually using `grpcurl` on a created habit, for instance one with the following ID: 98ab1bbe-41d5-4ed3-8f33-e4f7bec448c8.

```
$ grpcurl \
-import-path api/proto/ \
-proto service.proto \
-plaintext -d '{"habit_id":"98ab1bbe-41d5-4ed3-8f33-e4f7bec448c8"
localhost:28710 \
habits.Habits/TickHabit
```

Upon inspection, no errors have been returned. Our interest now lies in determining the frequency of calls made to the Tick endpoint for a given habit. Let us proceed to retrieve this information.

10.7.4 Get habit status

The last task entails retrieving the count of habit ticks for a given week. This requires both the habit ID and a timestamp to specify the desired week. We shall proceed by implementing an endpoint capable of accepting the ID and timestamp parameters, which will then furnish habit details alongside the tick count. The proto definition looks like below:

Listing 10.37 api/proto/service.go: GetHabitStatus definition

```
// GetHabitStatus is the endpoint to retrieve the status of ticks
rpc GetHabitStatus(GetHabitStatusRequest) returns (GetHabitStatus

// GetHabitStatusRequest is the request to GetHabitStatus endpoin
message GetHabitStatusRequest {
  // The identifier of the habit we want to retrieve.
  string habit_id = 1;

  // The time for which we want to retrieve the status of a habit
  optional google.protobuf.Timestamp timestamp = 2;
}
```

```
// GetHabitStatusResponse is the response to retrieving the statu
message GetHabitStatusResponse {
  // All the information of a habit.
  Habit habit = 1;
  // The number of times the habit has been ticked for a given we
  int32 ticks_count = 2;
}
```

The remainder of the steps? You should be able to do it all alone!

1. Create a method on the server side
2. Isolate the logic on the domain
3. Retrieve the data on the repository side
4. Plug all the calls
5. Do not forget to test!

If you test with grpcurl, you should obtain something like below:

```
$ grpcurl \
-import-path api/proto/ \
-proto service.proto \
-plaintext -d '{"habit_id":"98ab1bbe-41d5-4ed3-8f33-e4f7bec448c8"
localhost:28710 \
habits.Habits/GetHabitStatus

{
  "habit": {
    "id": "98ab1bbe-41d5-4ed3-8f33-e4f7bec448c8",
    "name": "read a few pages",
    "weeklyFrequency": 3
  },
  "ticksCount": 2
}
```

10.7.5 Add a timestamp

It would be more fun to be able to dive into the past to tick a habit we forgot to update. To do so, we can extend the two last endpoints by adding a timestamp to the requests.

In proto, there are different types we can import that will be nicely serialised in the programming language you choose. You can always refer to the full list of well-known types in the Protocol Buffers documentation (https://protobuf.dev/reference/protobuf/google.protobuf/). The expected format is a RFC3339 date string such as “2024-01-25T10:05:08+00:00”. Let’s import the timestamp type by adding this line in the top imports of our file service.proto.

```
import "google/protobuf/timestamp.proto";
```

Here is an example of `GetHabitStatusRequest` on how to add the timestamp as a new field using the proto `Timestamp` type. By default, a field is optional in proto version 3, but you can ask for a field to be present by adding the `required` keyword:

Listing 10.38 api/proto/service.go: Import the timestamp

```
message GetHabitStatusRequest {
  string habit_id = 1;
  google.protobuf.Timestamp time = 2;
}
```

Update the two endpoints and test your code!

10.7.6 Habit Tracker in action

We are now able to play with the habit tracker, so let’s play a full scenario where we add habits, we tick them, retrieve their status, tick again with a timestamp and retrieve their status for the given date. We can do it manually by using `grpcurl` in your terminal or we can update the integration test. Let’s see first the `grpcurl` commands and compare the responses.

1. Create a habit “Write some Go code”
   Request:

```
$ grpcurl \
-import-path api/proto/ \
-proto service.proto \
-plaintext -d '{"name":"Write some Go code", "weekly_frequency":3
localhost:28710 \
habits.Habits/CreateHabit
```

Response:

```
{
  "habit": {
    "id": "94c573f1-df03-45ec-97fc-8b8fc9943472",
    "name": "Write some Go code",
    "weeklyFrequency": 3
  }
}
```

2. Create a habit “Read a few pages”
   Request:

```
$ grpcurl \
-import-path api/proto/ \
-proto service.proto \
-plaintext -d '{"name":"Read a few pages", "weekly_frequency":5}'
localhost:28710 \
habits.Habits/CreateHabit
```

   Response:

```
{
  "habit": {
    "id": "96b72dce-7a2e-43ce-9091-0f9fc447b8a1",
    "name": "Read a few pages",
    "weeklyFrequency": 5
  }
}
```

3. Retrieve the list of the habits
   Request:

```
$ grpcurl \
-import-path api/proto/ \
-proto service.proto \
-plaintext -d '{}' \
localhost:28710 \
habits.Habits/ListHabits
```

   Response:

```
{
  "habits": [
    {
      "id": "94c573f1-df03-45ec-97fc-8b8fc9943472",
      "name": "Write some Go code",
      "weeklyFrequency": 3
    },
    {
      "id": "96b72dce-7a2e-43ce-9091-0f9fc447b8a1",
      "name": "Read a few pages",
      "weeklyFrequency": 5
    }
  ]
}
```

4. Tick habit “Write some Go code” without a timestamp because you just did it
   Request:

```
$ grpcurl \
-import-path api/proto/ \
-proto service.proto \
-plaintext -d '{"habit_id":"94c573f1-df03-45ec-97fc-8b8fc9943472"
localhost:28710 \
habits.Habits/TickHabit
```

   Response:

```
{

}
```

5. Get the status of the habit “Write some Go code” for the current week
   Request:

```
$ grpcurl \
-import-path api/proto/ \
-proto service.proto \
-plaintext -d '{"habit_id":"94c573f1-df03-45ec-97fc-8b8fc9943472"
localhost:28710 \
habits.Habits/GetHabitStatus
```

   Response:

```
{
  "habit": {
    "id": "94c573f1-df03-45ec-97fc-8b8fc9943472",
    "name": "Write some Go code",
    "weeklyFrequency": 3
  },
  "ticksCount": 1
}
```

6. Tick habit “Read a few pages” without a timestamp because you are doing it
   Request:

```
$ grpcurl \
-import-path api/proto/ \
-proto service.proto \
-plaintext -d '{"habit_id":"96b72dce-7a2e-43ce-9091-0f9fc447b8a1"
localhost:28710 \
habits.Habits/TickHabit
```

   Response:

```
{

}
```

7. Get the status of the habit “Read a few pages” for the current week
   Request:

```
$ grpcurl \
-import-path api/proto/ \
-proto service.proto \
-plaintext -d '{"habit_id":"96b72dce-7a2e-43ce-9091-0f9fc447b8a1"
localhost:28710 \
habits.Habits/GetHabitStatus
```

   Response:

```
{
  "habit": {
    "id": "96b72dce-7a2e-43ce-9091-0f9fc447b8a1",
    "name": "Read a few pages",
    "weeklyFrequency": 5
  },
  "ticksCount": 1
}
```

8. Tick habit “Read a few pages” with a timestamp in the previous week

Request:

```
grpcurl \
-import-path api/proto/ \
-proto service.proto \
-plaintext -d '{"habit_id":"96b72dce-7a2e-43ce-9091-0f9fc447b8a1"
localhost:28710 \
habits.Habits/TickHabit
```

Response:

```
{

}
```

9. Get the status of the habit “Read a few pages” during the previous week
   Request:

```
grpcurl \
-import-path api/proto/ \
-proto service.proto \
-plaintext -d '{"habit_id":"96b72dce-7a2e-43ce-9091-0f9fc447b8a1"
localhost:28710 \
habits.Habits/GetHabitStatus
```

Response:

```
{
  "habit": {
    "id": "96b72dce-7a2e-43ce-9091-0f9fc447b8a1",
    "name": "Read a few pages",
    "weeklyFrequency": 5
  },
  "ticksCount": 1
}
```

Launching manually the commands can be a bit annoying and repetitive, so let’s automate this and update the integration test.

First, we can write helper functions as we did previously to tick a habit and to verify the habit status matches when calling `GetHabitStatus`. These functions will both make the call to the API and validate the returned values.

Listing 10.39 integration_test.go: Add TickHabit and GetStatus calls

```
func tickHabit(t *testing.T, habitsCli api.HabitsClient, id strin
    _, err := habitsCli.TickHabit(context.Background(), &api.Tick
        HabitId: id,
    })
    require.NoError(t, err) #A
}

func getHabitStatusMatches(t *testing.T, habitsCli api.HabitsClie
    h, err := habitsCli.GetHabitStatus(context.Background(), &api
    require.NoError(t, err)

    assert.Equal(t, expected.Habit, h.Habit) #B
    assert.Equal(t, expected.TicksCount, h.TicksCount)
}
```

The generated ID is needed to call TickHabit and GetHabitStatus endpoints, we can retrieve it from addHabit helper method.

Listing 10.40 integration_test.go: Update addHabit to retrieve the id

```
func addHabit(t *testing.T, habitsCli api.HabitsClient, freq *int
    resp, err := habitsCli.CreateHabit(context.Background(), &api
        Name:            name,
        WeeklyFrequency: freq,
    }) #B
    require.NoError(t, err)

    return resp.Habit.Id #C
}
```

You can now add the calls to the main test function by calling the two helpers above.

Listing 10.41 integration_test.go: Call tickHabit and verify the statuses

```
func addHabit(t *testing.T, habitsCli api.HabitsClient, freq *int
    // add 2 ticks for Walk habit
    tickHabit(t, habitsCli, idWalk)
    tickHabit(t, habitsCli, idWalk)

    // add 1 tick for Read habit
    tickHabit(t, habitsCli, idRead)
    // check that the right number of ticks are present
    getHabitStatusMatches(t, habitsCli, idWalk, &api.GetHabitStat
        Habit: &api.Habit{
            Id:             idWalk,
            Name:           "walk in the forest",
            WeeklyFrequency: 1,
        },
        TicksCount: 2,
    })

    getHabitStatusMatches(t, habitsCli, idRead, &api.GetHabitStat
        Habit: &api.Habit{
            Id:             idRead,
            Name:           "read a few pages",
            WeeklyFrequency: 3,
        },
        TicksCount: 1,
    })
```

Congratulations!You have built a solid habit tracker backend which can be easily reused for a frontend application. You can commit and enjoy your new project!

10.8 Summary

- The `go generate` command is a Go tool that gives the possibility to generate programs from the source code. The compiler will scan comments with the specific syntax //go:generate and will execute the following commands.
- gRPC, standing for Google Remote Procedure Call, is a framework to connect services, devices, applications and more. It is a powerful and efficient framework for transporting light-weight messages in Protobuf format.
- Protobuf, short for Protocol Buffers, provides serialisation for structured data while guaranteeing high performance. It comes with its own syntax composed of Protocol Buffer messages and services written in .proto files.
- Protobuf messages are language-neutral, thanks to the Protobuf compiler (`protoc`), you can generate interfaces and structures in many programming languages (Go, Java, C++ and more).

- gRPC clients and servers communications are standardised thanks to status codes defined by the RPC API. A status is composed of an integer code and a string message. While designing the API, you should pick the most appropriate return code for your use case and you can always refer to the documentation (https://grpc.github.io/grpc/core/md_doc_statuscodes.html).
- grpcurl is an open-source CLI tool that enables you to communicate with gRPC servers easily. It is basically like curl but for gRPC. Human-friendly, it allows you to define JSON requests instead of unreadable bytes. It is very handy when you need to test services manually.
- Declare small interfaces close to their use. It comes in handy when testing to mock your dependencies instead of counting on hardcoded behaviours.
- Dependency injection is a technique to give all the needed objects to a function instead of creating or building them internally. Note that it comes in very handy to mock the dependencies when testing the function.
- While testing, there are different ways of simulating dependency behaviours, mock tools are handy for describing expected results. The main well-known mock tools are mockgen coming with Go, `mockify` and `minimock`, which we used during the chapter.
- `context` is a standard library which provides the Context type and its associated methods to carry information along requests and responses. For example, when a user sends a request, you can store its identity in the context and retrieve it later in the chain of functions. It is safe to use methods like `WithCancel` or `WithDeadline` across multiple goroutines.
- A program should create only one context and provide it to its functions. Most of the time, the context will be coming from an external caller. Creating children for a context is perfectly fine and encouraged.
- Always pass a `context.Context` as the first parameter of a function, even if you are not using it (you can use the blank identifier in this rare case).
- A context will be necessary any time we make a remote call across the network, whichever protocol or framework is being used. Setting a deadline for the remote call is a good safety net - otherwise, your application calls might be hanging forever.
- Don't use a `context.Context` as a key-value storage inside an

application.

- Mocking functions that use a context is sometimes tricky, especially if the function being tested creates a context of its own. To solve this, many libraries offering mocks expose a variable that can be used as a wildcard for the context.
- Generated mocks are programmed to behave as specified. If they receive a call for a precise set of parameters, they will return the specified values. However, it is sometimes necessary to override this default behaviour of always returning something, especially when testing the behaviour when a deadline is reached.
- Using `testing.Short()` allows us to know if the `-short` flag was passed on the `go test` command line. Long or expensive tests should be skipped altogether when this flag is set, by using `t.Skip()`.

Appendix A. Installation steps

Any compiled language needs, first and foremost, a compiler.

The Go toolchain was initially written in C. Since Go 1.3, it is written directly in Go, following the principle of *eating your own dog food*. As everything is open source, you can at any time suggest improvements, or at least look into the standard library’s source code for how other developers write their Go.

A.1 Install

Start by visiting the Go website. It explains in a simple way (did we tell you that Go aims for simplicity?) how to download the installer and run it on either Linux, Mac or Windows. Follow the installation steps and do not forget to add go to your path.

There is no good reason to pick old versions. Just for the record, we are writing this book using Go 1.20.

https://go.dev/doc/install

A.2 Check

As mentioned on the online installation guide, you can check the version of Go that you are using and also verify that Go is properly installed by running this command in any directory at all:

Listing A.1 Check the installation in your console

```
$ go version
go version go1.19.0 darwin/arm64
```

A.3 Go’s environment variables

Go, under the hood, uses several variables without being explicit about it. In this section, we’ll be looking closely at two of these variables - namely `GOROOT` and `GOPATH`.

If you’ve just installed Go, these variables won’t be set in your sessions. “But how come Go uses them if they’re not set?” might you ask. A very wise question. Go is able to use default values for these (and more, as we’ll see) variables.

A.3.1 The go env command

The `go` command can access environment variables, just as any program could. However, Go comes with an extra layer of variables, which aren’t visible to you from a terminal. These variables can be listed with the `go env` command. `go env` will return all the Go environment variables it can access. Typically, you will paste the output of this command along with any question you post online or when you open a bug.

Alternatively, we can pass it a list of the variables we want to retrieve, which limits the output. Here is an example of the results of this command. Fear not, should your output differ - after all, we don’t share the same environment.

Listing A.2 Example go env output

```
$ go env -json GOBIN GOENV GOROOT GOPATH CGO_ENABLED
{
"GOBIN": "",
"GOENV": "/home/user/.config/go/env",
"GOPATH": "/home/user/go",
"GOROOT": "/usr/local/go",
"CGO_ENABLED": "1",
}
```

We won’t go through the long list of variables displayed by `go env`, as most are beyond the scope of this book. For instance, the variables towards the end of the list are related to CGo, the utility that allows integration of C code within Go code.

The values returned by `go env` here are the default values, which are based on your machine’s architecture and your installation directory of Go. We hardly ever need to modify any of these values, but, for your knowledge, they can be overridden with regular environment variables.

Listing A.3 Overriding a Go env variable with an env variable

```
# On Linux:
$ CGO_ENABLED=0 go env -json CGO_ENABLED
{
"CGO_ENABLED": "0",
}

# On Windows:
C:\> set "CGO_ENABLED=0" & go env -json CGO_ENABLED
```

They can also be written in Go’s configuration file (which is pointed by the `GOENV` variable) with the `go env -w VARIABLE=VALUE` command.

Listing A.4 Overriding a Go environment variable with go env

```
# On Linux:
$ go env -w GOBIN=/home/user/bin
$ go env GOBIN
{
"GOBIN": "/home/user/bin",
}
# On Windows
C:\> go env -w GOBIN=%LOCALAPPDATA%
```

A.3.2 The GOBIN variable

The `GOBIN` variable contains the path of a directory in which Go will download any tools you install with `go install url@version`. This is the standard way of retrieving utilities in Go. More on the next page.

```
$ go install golang.org/x/tools/cmd/godoc@latest
```

A.3.3 The GOPATH variable

The `GOPATH` variable contains a list of paths to directories in which Go will resolve its dependencies. Earlier versions of Go used a decentralised approach - should two projects require the same dependency, that dependency would be downloaded twice, and stored in the unregretted `vendors` directory of each project. This is no longer the case: now, when a dependency is needed, it is stored locally and any project you have will use the local version rather than re-download that dependency.

Make sure your workspace is contained in the `GOPATH` list of directories. If you're working in `${HOME}/go`, you'll be fine. Otherwise, you can use the following command – Windows users should use a semicolon to add an extra path:

Listing A.5 Add a directory to your go path - Unix-based system

```
# On Linux
$ go env -w GOPATH=${GOPATH}:/path/to/workspace
# On Windows
C:\> go env -w "GOPATH=%GOPATH%;C:\path\to\workspace"
```

A.3.4 The GOROOT variable

The `GOROOT` variable points to the directory which contains the installation of Go. We recommend not changing what is in the `GOROOT` tree, because installing a new version of Go would mean discarding any of your changes there. Similarly, it is not ideal to have any of the other Go environment variables point to somewhere within the `GOROOT`.

As part of your installation, you made sure the path `${GOROOT}/bin` was included in your `PATH` environment variable - that's how we can run go. This directory contains another executable - `gofmt` - which is in charge of formatting code.

A.4 Hello!

The Hello World instructions are detailed on Go's website, but here is a short version. You will find, at the very beginning of the first project, in chapter 2, explanations regarding each line of the typical hello world.

Create a hello folder in your `${GOPATH}` with a file named `hello.go` and paste the following:

Listing A.6 hello.go

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

To manage dependencies and versions in Go, we use modules. Run the following command to create your first module:

```
$ go mod init
```

A `go.mod` file appeared. It contains the path to your module and your Go version.

Then run your code in the same folder and wave at your screen: your machine is trying to communicate!

Listing A.7 run a go file

```
$ go run hello.go
Hello, World!
```

Funny how quickly we personify our computer friends.

A.5 Installing new dependencies

When developing new functionalities, we like to build upon the work of others. Go has 2 different tools to retrieve existing work, each with its specific objective: `go install` and `go get`. They work in a very similar way, but are used in different contexts.

Both commands accept the name of a repo, and will retrieve its contents at a specific version. The main difference is that `go get` only retrieves go files from that repo, when `go install` also compiles the retrieved package into an executable. Which one you use will depend on what you need - do you want the sources or the executable ?

`go install` is rather recent, and some public repositories will still list `go get` as the method to install their binaries. If you follow that path, you will be faced with a message suggesting using `go install` - in these cases, use the second option of `go install` listed below.

A.5.1 go install

If you need to retrieve a binary written in Go, use `go install`. It will fetch the sources and compile them locally for your machine's architecture. There are three different ways of calling the `go install` tool.

The first option lets you retrieve a specific version of a repository. This is very useful when writing automation tools, when you want a constant and deterministic flow.

The second option is very similar, and retrieves the code at the latest version of the repository, using its `main` (or `master`) branch. This is the most common way of using `go install` manually.

The last option will use the contents of your project's `go.mod` file to find which version to download and install. This will only work if you are running the command from within a Go project.

Listing A.8 go install examples

```
# Install a specific version.
$ go install golang.org/x/tools/cmd/godoc@v0.1.12

# Install the highest available version.
$ go install golang.org/x/tools/cmd/godoc@latest

# Install the highest available version.
$ go install golang.org/x/tools/cmd/godoc
missing go.sum entry for module providing package golang.org/x/to
        go mod download golang.org/x/tools
```

A.5.2 go get

If you need sources that your own code depends upon, `go get` will update your module file (see below) and download the sources into `${GOPATH}/pkg/mod`. You can then look into them in order to understand what the code does. Similarly to `go install`, `go get` can be used with different behaviours.

The first option you have is to run `go get` on an URL, without specifying a version or anything. This will retrieve the contents of that dependency, and its own dependencies to your `go.mod` file - you are telling go you need that repo in your project.

The second option you have is to retrieve the code by explicitly giving the name of a tag, branch, or commit. This is extremely useful when working on two projects at the same time, or when working with a project that hasn't been merged into main yet. This option will register that new package into your `go.mod` file, at the desired version.

Listing A.9 go get examples

```
# Retrieve the experimental slices package using the version defi
$ go get golang.org/x/exp/slices
# Retrieve the experimental slices package, latest commit on bran
$ go get golang.org/x/exp/slices@master
# Retrieve the experimental slices package at a specific commit o
$ go get golang.org/x/exp/slices@c99f07
```

A.6 Code editors

Go is supported by more and more code editors. As always in this situation, the best tool is the one you know how to use. The official Go website, as we write, lists three editors:

- GoLand, by JetBrains. JetBrains has a long list of editors for various languages, the most famous being IntelliJ for Java. You can install GoLand as a standalone editor or add the Go plugin to any other editor in the list.
- Visual Studio Code, by Microsoft, has a Go extension.
- vim-go is great if you already know vim

A quick search around the web will easily give you instructions to add Go support to your usual tool, if it is not already there, e.g. with GoSublime, Atom with GoPlus, LiteIDE…

Go is now installed on your machine and you can start using it and follow the book’s instructions. We learned about the Go environment variables and important paths. Your terminal knows how to greet you in English using Go, you can move to Chapter 2 and teach it to greet you in other human languages.

Appendix B. Formatting cheat sheet

Go offers several *verbs* that are passed to printing functions to format Go values. In this appendix, we present the most known verbs and special values that can be passed to these functions. You can refer to these tables all along the book. The result for each of the following entries was generated by `fmt.Printf("{{verb}}", value)`.

Table B.1. Default

Verb	Output for fmt.Printf("{{Verb}}", []int64{0, 1})	Description
`%v`	`[0 1]`	Default format
`%#v`	`[]int64{0, 1}`	Go-syntax format
`%T`	`[]int64`	Type of the value

Table B.2. Integers

Verb	Output for fmt.Printf("{{Verb}}", 15)	Description
`%d`	`15`	Base 10
`%+d`	`+15`	Always show the sign

`%4d`	`␣␣15`	Pad to 4 characters with spaces, right justified
`%-4d`	`15␣␣`	Pad to 4 characters with spaces, left justified
`%04d`	`0015`	Pad to 4 characters with prefixing zeros
`%b`	`1111`	Base 2 (binary)
`%o`	`17`	Base 8 (octal)
`%x`	`f`	Base 16, lowercase
`%X`	`F`	Base 16, uppercase
`%#x`	`0xf`	Base 16 with leading 0x

Table B.3. Floats

Verb	Output for fmt.Printf("{{Verb}}", 123.456)	Description

`%e`	`1.234560e+02`	Scientific notation
`%f`	`123.456000`	Decimal point, no exponent. The default precision is 6.
`%.2f`	`123.46`	Default width, precision 2 digits after the decimal point
`%8.2f`	`␣␣123.46`	Width 8 chars, precision 2 digits after the decimal point. Default padding character is space
`%08.2f`	`00123.46`	Width 8 chars, precision 2 digits after the decimal point. Left-padding with specified character (here, `0`)
`%g`	`123.456`	Exponent when needed, necessary digits only

Table B.4. Characters

Verb	Output for fmt.Printf("{{Verb}}", 'A')	Description
`%c`	`A`	Character
`%q`	`'A'`	Quoted character
`%U`	`U+0041`	Unicode

`%#U`	`U+0041 'A'`	Unicode with character

Table B.5. Strings or byte slices

Verb	Result for "gophers"	Description
`%s`	`gophers`	Plain string
`%8s`	`␣␣gophers`	Width 8, right justified
`%-8s`	`gophers␣␣`	Width 8, left justified
`%q`	`"gophers"`	Quoted string
`%x`	`676f7068657273`	Hex dump of byte value
`% x`	`67 6f 70 68 65 72 73`	Hex dump with spaces

Table B.6. Booleans

Verb	Output for fmt.Printf("{{Verb}}", true)	Description
		Equivalent to %v but only for

`%t`	`true`	booleans

Table B.7. Pointers

Verb	Output for fmt.Printf("{{Verb}}", new(int))	Description
`%p`	`0xc0000b2000`	Base 16 notation with leading 0x

Table B.8. Special values

Verb	Description
`\a`	U+0007 alert or bell
`\b`	U+0008 backspace
`\\`	U+005c backslash
`\t`	U+0009 horizontal tab
`\n`	U+000A line feed or newline
`\f`	U+000C form feed

`\r`	U+000D carriage return
`\v`	U+000b vertical tab
`%%`	The `%` character : `fmt.Printf("%05.2f%%", math.Pi)` prints `03.14%`

All Unicode values can be encoded with backslash escapes and can be used in string literals.

There are four different formats:

- `\x` followed by exactly two hexadecimal digits : \x64,
- `\` followed by exactly three octal digits: \144,
- `\u` followed by exactly four hexadecimal digits \u0064,
- `\U` followed by exactly eight hexadecimal digits \U00000064.

The escapes `\u` and `\U` represent Unicode code points. Here is an example of a Unicode value embedded in a string:

```
fmt.Println("Thy bosom is endear\u00e8d with all hearts")
```

Appendix C. Zero values

C.1 What is a zero value

Sometimes while coding, you will need the use of a variable without assigning a value. For example, a variable should be declared before a condition to exist outside of it:

```
var counter int
if readline(&buf) {
    counter += 1
}
fmt.Println(counter)
```

In this case, the variable `counter` is declared without an explicit initial value meaning it is given by default its zero value, which, for an integer, is `0`. Note that the initialisation to zero value is done recursively either for a slice, a map or a structure: each element or field will be set to its zero value according to its type.

C.2 The zero values of any types

Most zero-values are intuitive, but there are a few that are worth keeping in mind. Those that you should absolutely remember are:

- Booleans have a zero-value `false`;
- Slices and maps have a zero-value equal to the `nil` entity

You can find below a table of examples from the simplest to more complex types with their zero-values. Feel free to come back to this table through the book.

Table C.1. Zero values of any types

Variable declaration	Observed zero-value
`var r rune`	`r == 0`
`var f float32`	`f == 0.`
`var b bool`	`b == false`
`var i []int`	`i == nil (*)`
`var a [2]complex64`	`a == [2]complex64{0+0i, 0+0i}`
`var m map[string]int`	`m == nil (*)`
`type person struct`	`p` has been allocated in memory, it can’t be `nil` (`nil` is

{ age int name string } var p person	also not of type `person`) `p.age == 0` `p.name == ""`
`var i * int`	`i == nil`
`type Doer interface { Do() } var d Doer`	`d == nil`
`var c chan string`	`c == nil`
`type translate func (string) string var t translate`	`t == nil`

(*) Maps and slices should be declared with the `make()` function. If not, they take the zero-value of nil, as described here. There are a few things to know about slices and maps that can come in handy at any time.

C.3 Slices and maps specificities

Slices and maps have some specificities that should be noticed when manipulating zero values and nil entries.

The `len` function can be called on `nil` slices or maps, and returns the value 0. In a vast majority of cases, checking the length is better than checking if the structure is `nil`. Let’s have a look at an example:

Listing C.1 Checking the length of a slice

```
func main() {
    data := []string{}
    fmt.Println(data == nil)
    fmt.Println(len(data))
    fmt.Println(data[0])
}
======
false
0
panic: runtime error: index out of range [0] with length 0
```

As you can see in this example, declaring an empty slice doesn’t return a `nil` slice. In order to be able to check any of its elements, we should always check the length of a slice.

There is, however, one thing that we can do with uninitialised slices, and this is appending entries to them. This won’t cause any panic error, and will simply return a non-`nil` slice with the new elements, if there were any.

Listing C.2 Appending to a nil slice

```
func main() {
    var data []string
    fmt.Println(data == nil)
    data = append(data, "hello")
    fmt.Println(data)
}
======
true
[hello]
```

Maps follow the same logic: when declaring one without initialising it, the map will be `nil`. The important information is that you can’t write data in such a map.

Listing C.3 Trying to add elements in a nil map

```
func main() {
    var m map[string]int
    m["hello"] = 37
}
======
panic: assignment to entry in nil map
```

However, accessing items in a `nil` map will return the zero-value of this item (it’s obviously not present). This is useful information, because sometimes, you receive a map from a library. It’s safe to check for keys in the map, but it’s even safer to check for its length first.

Listing C.4 Trying to read elements from a nil map

```
func main() {
    var m map[string]int
    count, found := m["hello"]
    fmt.Printf("found: %v; count: %d\n", found, count)
}
======
found: false; count: 0
```

C.4 Benefiting from zero-values

Suppose we want to count the number of different words in a text, and keep track of their number of occurrences. One simple way of achieving this goal is to use a map, where the keys will be the different words, and the values will be their current count, as we iterate through the list of words.

Listing C.5 A structure to count different words in a text

```
wordCount := make(map[string]int)
```

When accessing an entry absent from this map - a word that we haven't seen so far - the returned value at the index of the new word will be the zero value of the integer tye: 0. This is extremely convenient, as it means we can consider words that haven't been seen so far as words that have been seen zero times. Recording an occurrence of a word doesn't need any extra effort if the word had or hadn't been registered before: we simply add 1 to the counter.

Listing C.6 Counting different words in a text

```
import (
    "fmt"
    "strings"
)

func countWords(s string) {
    wordCounter := make(map[string]int)
    for _, word := range strings.Fields(s) {
        wordCounter[word]++
    }

    // print results
    for word, count := range wordCounter {
        fmt.Printf("We recorded the word %q %d time(s).\n", word,
    }
}

func main() {
    countWords("to be or not to be")
}
======
We recorded the word "or" 1 time(s).
We recorded the word "not" 1 time(s).
We recorded the word "to" 2 time(s).
We recorded the word "be" 2 time(s).
```

Appendix D. Benchmarking

One of the great tools Go offers is a benchmarking command. Writing benchmarks to compare the allocation of memory and the execution time is extremely simple - it's very similar to writing a test over a function.

We'll use the type `B`, defined in the `testing` package. You'll never guess what `B` stands for…

The type `B` has one exposed field, and integer `N`, which counts the number of iterations the benchmark has executed. When running benchmarks, this field has an initial value that will allow at least a certain amount of iterations to ensure we have a steady result - no need to try and set it manually.

Test benchmarking functions follow a convention very similar to test functions: their name must start with `Benchmark`.

D.1 StringBuilder (from 4.2.1)

We explained that using concatenation to build long strings is not a good idea and you should use a Builder. Don't take our word for it, measure it yourself!

We are building a string that represents the `feedback` type, which is a slice of `statuses`.

Listing D.1 status_internal_test.go: Examples of benchmarks

```
// Benchmark the string concatenation with only one value in feed
func BenchmarkStringConcat1(b *testing.B) {
   fb := feedback{absentCharacter}
   for n := 0; n < b.N; n++ { #A
      _ = fb.StringConcat()
   }
}
```

EXERCISE: Instead of having a feedback of one status, write benchmark functions that will accept longer feedbacks. Since Gordle will mostly be used with words of 5 characters, that’s probably the length we want to benchmark.

As mentioned earlier, the benchmark can be run using our friend the `go test` tool, with specific options. In order to run benchmarks (just as we had for tests) for all files in subdirectories, we pass the `-bench=.` option, and, if we want to display details of memory operations, we can add `-benchmem`. Running benchmarks will, however, also run the tests. If we want to avoid that, and run only the benchmarks, we can add an extra parameter to the command-line, an indication to help Go find our tests by their name: a regular expression. We’ll cover this topic a bit more extensively, but, for now, we’ll use the (very) loose `^$`, which will match all benchmark test functions.

```
$ go test ./... -run=^$ -bench=. -benchmem
```

Listing D.2 Result of the benchmarks

```
$ go test ./... -run=^$ -bench=. -benchmem

goos: darwin
goarch: arm64
pkg: github.com/ablqk/tiny-go-projects/chapter-04/2_feedback/gord
BenchmarkStringConcat1-10      174882942       6.850 ns/op
BenchmarkStringConcat2-10       15633693      74.28 ns/op       2
BenchmarkStringConcat3-10        8609542     137.1 ns/op        5
BenchmarkStringConcat4-10        5873654     201.1 ns/op       10
BenchmarkStringConcat5-10        4455464    275.2 ns/op        16

BenchmarkStringBuilder1-10      71407850     16.69 ns/op
BenchmarkStringBuilder2-10      30721999     38.28 ns/op
BenchmarkStringBuilder3-10      27036134     45.64 ns/op
BenchmarkStringBuilder4-10      17278803     70.44 ns/op
BenchmarkStringBuilder5-10      16189770     73.27 ns/op
PASS
ok      github.com/ablqk/tiny-go-projects/chapter-04/2_feedback/g
*/
```

The output of this command can be a bit scary at first. After all, we only wrote a 5-line test! Let’s have a look. We can see several lines, and several columns. Each line corresponds to a function to benchmark that the go tool found in our code (respecting our earlier loose regexp). The columns represent metrics that were observed by the test tool during the execution of the benchmark.

- The first column is the name of the function, with a suffix indicating the number of processors on the machine.
- The second column indicates the number of loops that were executed (the `b.N` value, if you remember).
- The third column indicates the amount of time (usually in nanoseconds) each operation took.
- The fourth column indicates the number of bytes allocated per operation.
- The final column indicates the number of memory allocations per operation.

Some quick maths should show that the benchmark tool gave roughly the same amount of execution time to each line (second column multiplied by third column). The benchmark results are interesting as they are pretty simple to read.

String concatenation is three to four times slower than using the string builder when we need to append five times. Using the `a + b` string concatenation makes a number of memory allocations proportional to the number of strings to concatenate (which makes sense, since strings are immutable), and these operations cost more and more memory every time. On the other hand, the memory allocations of the string builder are scarcer and lighter. This benchmark confirms we definitely should be using the string builder to generate feedback!

D.2 Summary

In order to compare the relative efficiency of two implementations, Go's `test` tool allows for a simple implementation of benchmark functions. A `BenchmarkNameOfFunc(b *testing.B)` function will be considered as a benchmarking function and will be ran with `go test ./... -run=NameOfFunc -bench=. -benchmem`. The benchmarked function must be called inside a `for n := 0; n < b.N; n++ {` loop.

welcome

Thank you for purchasing the *Learn Go with Pocket-Sized Projects*! We hope you will have fun and make immediate use of your learnings.

This book is for developers who want to learn the language in a fun and interactive way, and be comfortable enough to use it professionally. Each chapter is an independent pocket-sized project. The book covers the specificities of the language, such as implicit interfaces and how they help in test design. Testing the code is included throughout the book. We want to help the reader become a good modern software developer while using the Go language.

This book also contains tutorials for command-line interfaces, and for both REST and gRPC microservices, showing how the language is great for cloud computing. It finishes with a project that uses TinyGo, the compiler for embedded systems.

Each pocket-sized project is written in a reasonable number of lines. Our goal is to provide various exercises so any developer who wants to begin with Go or to explore the specificities of the language can follow the steps described in each chapter. This is not a book to learn development from scratch and the chapters are graded.

We encourage you to ask your questions and post the feedback you have about the content in the liveBook Discussion forum. We want you to get the most out of your readings to increase your understanding of the projects.

— Aliénor Latour, Donia Chaiehloudj and Pascal Bertrand

In this book