YOUR FAST TRACK FOR GOLANG

EVERYDAY GO

BY ALEX ELLIS

Learn tools, techniques and patterns from real tools used in production.

Everyday Golang - The Fast Track

Alex Ellis

2021

Everyday Golang - The Fast Track

Introduction

"Everyday Go" is the fast way to learn tools, techniques and patterns from real tools used in production.

About the author: Alex is a software engineer and an avid technical writer. There's nothing he likes more than learning a new technology, and then writing it up in a clear, practical way. He's well known for his blog posts on containers, Kubernetes, cloud computing and Go. He founded the OpenFaaS project in 2016, and since then has gone on to publish training courses, eBooks and dozens of other tools for developers.

How to use the book

This book is a compilation of practical examples, lessons and techniques for Go developers. They are the kind of things that you may need to learn and apply in your everyday journey. The topics cover the software lifecycle from learning the fundamentals, to software testing, to distribution and monitoring.

If you feel like you may have seen some of these techniques before, you would be right. I've been writing about Go since 2015 and wanted to take everything that I've needed on a regular basis, and to lay it out for you to benefit from. It's not a reference book or a complete guide to the language, but if you like learning by example and aren't afraid to get your feet wet, then this style is probably for you.

The chapter "Writing unit-tests in Go" was originally part of my top ranking and performing blog post. It was even featured in the Kubernetes documentation to help new Go developers get up to speed. In Everyday Go, I've rewritten, updated and extended the tutorial with more content and techniques that you can use in your work.

You could say that this is the *fast track* to get the results you'll need in your everyday writing of applications in Go.

- How to use the book
- Why Go?
- Getting started with Go
- Managing packages and paths

- Understand Go modules and vendoring
- Creating your first program in Go
- Adding external dependencies to your program
- Cross-compile your code for different Operating Systems
- Inject version information into your binaries
- Merge Go objects together into one
- Deal with HTTP and JSON parsing
- Learn how to use concurrency with Goroutines, channels, Mutexes, error groups and WaitGroups
- Avoid a common mistake with Goroutines
- Write unit tests and understand everything that's available to you
- Learn party tricks of the testing in Go
- Isolate dependencies and unit test a HTTP server
- Embed data and files into your application
- Create dynamic content with templates
- Build a CLI that you'll enjoy using
- Load and write to config files in YAML
- Write your own HTTP microservices
- Integrate with Prometheus for metrics
- Learn to build and release Go binaries
- Build Docker container images
- Release your code on GitHub
- Get set for your onward journey

Help the book to evolve

Just as my knowledge of Go has evolved over time, this book will also evolve with updates and new examples. This book should be a place that you can come to for a pattern or sample that you can use as a basis of a new program or feature.

Check the final chapter for how to submit comments, suggestions or corrections.

Changelog

- 5 August 2021 - Clarify usage of RWMutex and locking behaviour
- 19 July 2021 - Fix typo on X509 certificates.
- 14 July 2021 - Correct git clone URL, explain Marshal and Unmarshal, remove an import.
- 28 June 2021 - Update wording on concurrency vs parallelism, introduce new locking example and Mutex/RWMutex.

- 27 June 2021 - Update to Go modules section and clarification of paths in package cache directory
- 24 June 2021 - MIT license added for example apps
- 24 June 2021 - Add section on multiple packages and new example app "multiple-packages"
- 23 June 2021 - Add new section on Go modules usage and vendoring
- 22 June 2021 - Update Go release date in intro, add "merge-objects" example app

Why Go?

Go's new logo

According to [Wikipedia](#), Go (or Golang) is a programming language that was developed by a team at Google and first appeared in 2009. It's synonymous with

server-side development and tooling for infrastructure projects.

> "Go is an open source programming language that enables the production of simple, efficient and reliable software at scale"
>
> • Go branding guidelines

Here are some projects that come to mind, which are entirely written in Go:

- Docker - a container engine written by Docker Inc
- Kubernetes - a clustering system for containers built by Google and open source contributors
- Traefik - a load-balancer built by Traefik Labs
- Consul, Terraform - tools for managing infrastructure built by Hashicorp
- Caddy - a high performance HTTP server written by Matt Holt
- OpenFaaS - serverless framework for hosting your own functions, built by OpenFaaS Ltd and open source contributors

These are just a few of hundreds of thousands of projects which have appeared on [GitHub's trending page for Go](#).

As you can see from the above, Go is very popular for server-side projects, but it is also very popular for writing Command Line Interfaces (CLIs) because the binaries can be statically linked. Statically-linked Go binaries are both fast and small, and do not require an additional runtime making them a very popular choice.

My personal highlights for Go are:

- portability - static compilation for easy distribution
- cross-compilation - being able to build binaries for Linux, Windows, MacOS Operating Systems (OS) and Arm computers with a single command (`go build`)
- concurrency model - Go has built-in mechanisms for concurrency that make it well suited to tasks both small and large
- ecosystem - knowing Go means being able to contribute to almost any other open source Go project or product
- tooling - Go includes its own unit-testing runner and profiler, both of which are simple and easy to use
- opinions - Go has strong opinions on formatting, style and typing that make it easy to move between codebases

Above everything else, Go has an impressive set of packages available within its standard libraries or "stdlib". There are many useful packages and utilities that would need to be installed separately in a programming language like JavaScript, Ruby or Java.

- `crypto` - libraries for X.509 certificates and cryptography
- `compress` - work with zip files and archives
- `http` - a very powerful and simple HTTP client and server package including things like reverse proxies
- `net` - work directly with sockets, URLS and DNS
- `encoding/json` - work directly with JSON files
- `text/template` - a powerful templating engine for replacing tokens in files, and generating text and HTML
- `os` - work with low level OS primitives

You can find a complete package list here.

Through the next few chapters we will explore some of the above highlights and also solve various practical problems. Even if you're a seasoned Go developer, I'd recommend going through each of the exercises, because even after 5 years I still learn something new each week.

Getting started with Go

There are three ways to install Go.

- Install Go using a system package - such as an msi (Windows) or pkg (MacOS)
- Install Go using a package manager - such as `apt-get` (Debian), `pacman` (Arch Linux), `brew` (MacOS)
- Download a tar archive and unpack it to a set location (Linux only)

If you're a new user and running on Windows or MacOS, then go ahead with the first option and download Go from the homepage. The version changes frequently, so download the latest stable version.

If you like to use package managers, you should be aware that the version they offer often lags behind. You may be installing a very old version of Go, and some features may not be available.

A note for Windows users

Whilst the Go commands can be run `cmd.exe` and PowerShell, some of the example commands expect a UNIX-style shell which is present on MacOS and Linux hosts.

For ease of use, it's recommended that you try the content using Windows Subsystem for Linux (WSL), if it is available for your system. Or if not, make use of [Git Bash](#).

Git bash provides a terminal where you can run bash, have access to the `git` command and various other UNIX-style commands such as `export, ls, mkdir` and `curl`.

Direct installation on Linux

As a Linux user, I often need to install Go on a server, within a Docker image, or on my Raspberry Pi. I use the third approach of downloading and unpacking a tar for this.

For a PC:

```
curl -fLS https://golang.org/dl/go1.16.linux-amd64.tar.gz \
 -o /tmp/go.tgz
```

For a Raspberry Pi:

```
curl -fLS https://golang.org/dl/go1.16.linux-armv6l.tar.gz \
 -o /tmp/go.tgz
```

Then on either platform, unpack the archive:

```
sudo mkdir -p /usr/local/go/
sudo tar -xvf /tmp/go.tgz --strip-components=1 -C /usr/local/go/
```

After the installation, you will need to set your PATH and GOPATH variables in your bash file, so that they load every time you log in.

```
    echo "export PATH=\$PATH:/usr/local/go/bin/" | tee -a ~/.bash_pro
    echo "export GOPATH=\$HOME/go" | tee -a ~/.bash_profile
```

Now disconnect from your server or Raspberry Pi, or close the terminal and connect again.

```
pi@raspberrypi:~$ go version
go version go1.16 linux/arm
```

Packages and paths

Go is usually installed to `/usr/local/go/`, with the `bin` folder containing the main binary and any other built-in commands like `gofmt`.

You should also make yourself familiar with the `GOPATH` variable. Whenever packages are fetched, they will usually be downloaded there. In the section on Go modules, we will clarify its use further.

Whenever you run `go install path/to/project`, you'll find the binaries there also.

Imagine your GOPATH is `$HOME/go/`

- `GOPATH/pkg/` - Go modules downloaded as dependencies
- `GOPATH/bin/` - any CLIs that you build or install via `go get`
- `GOPATH/src/` - the default location for source code

Here is an example of how to download a HTTP server called hash-browns that I wrote to generate hashes based upon a HTTP POST body.

```
$ export GO111MODULE=on
$ go get github.com/alexellis/hash-browns

go: downloading github.com/alexellis/hash-browns v0.0.0-2020012220141
go: github.com/alexellis/hash-browns upgrade => v0.0.0-2020012220141!
go: downloading github.com/prometheus/procfs v0.0.0-20180920065004-41
◀                                                                     ▶
```

> Note: For the rest of the book, you should assume that `GO111MODULE` is set to `on` and that Go modules are being used everywhere.

The source for the specific package will be downloaded to:

```
$GOPATH/pkg/mod/github.com/alexellis/hash-browns@v0.0.0-
20200122201415-86d5dd6d7faa
```

Note that the specific suffix after the `@` could change, depending on the latest version of the package. When Go programs are released with a `v` prefix, the import path will look more like `v0.1.0` instead of `v0.0.0-` followed by a SHA.

For example, the release for openfaas/faas-provider is prefixed with a `v`, therefore a "prettier" path is provided:

```
$ go get github.com/openfaas/faas-provider
```

And the path is set as:

```
$GOPATH/pkg/mod/github.com/openfaas/faas-provider@v0.18.5
```

The other time that the more verbose path might be used, is if you are trying to reference a commit that isn't attached to a release or a tag on GitHub.

The dependencies for it will be downloaded as follows:

```
$GOPATH/pkg/mod/github.com/prometheus/procfs@v0.0.0-20180920065004-
418d78d0b9a7
```

Then it will be compiled and the binary will be created at:

```
$GOPATH/bin/hash-browns
```

You'll see a similar technique used in project READMEs for tools written in Go, so you may want to add `$GOPATH/bin/` to your PATH environment variable.

> Note: Whenever `GO111MODULE` is set to `off`, then the code is downloaded to `$GOPATH/src/` instead of `$GOPATH/pkg`. Building without Go modules may be disabled in a future release of Go.

Vendoring and Go modules

In the early days of the Go ecosystem, the source code of any dependencies being used by your program was placed a `vendor` folder. The vendor folder would be committed alongside your application code, giving you a copy and back-up of anything required to build your application.

In the previous example, hash-browns, the `github.com/prometheus/procfs` module was required, which would be cloned into the source tree as follows:

```
$GOPATH/src/github.com/alexellis/hash-browns
$GOPATH/src/github.com/alexellis/hash-browns/vendor/github.com/promet
```

Various vendoring tools such as the now deprecated [dep](), were used to download and manage these dependencies. One file would specify which packages were required, and another "a lock file" would specify the versions. So at any time you could `rm -rf` the `vendor` folder and run something like `dep ensure` to get a fresh copy of all the various files.

In Go 1.13 the concept of Go modules was introduced which largely mirrors vendoring, with one big change. The source for dependencies no longer needed to be cloned and committed to source control with your application. The transition for OpenFaaS' 40+ projects was relatively smooth, with just a few projects causing headaches and needing to be deferred.

Another change with Go modules, was the ability to compile source code outside of the `$GOPATH/src` convention. For instance, you could have your source code at `$HOME/dev/` instead of `$HOME/go/src/` or wherever you'd set it.

Whenever you create a new Go project, it's best to run `go mod init` first. If your code falls within the conventional GOPATH structure, then it will have its name determined automatically, but if not, you can pass it as an argument.

Compare the following:

```
export GH_USERNAME=alexellis

$ mkdir -p $GOPATH/src/github.com/$GH_USERNAME/hash-gen
$ cd $GOPATH/src/github.com/$GH_USERNAME/hash-gen
$ go mod init
go: creating new go.mod: module github.com/alexellis/hash-gen

$ cat go.mod
module github.com/alexellis/hash-gen

go 1.15
```

And when run outside of GOPATH:

```
$ mkdir /tmp/hash-gen
$ cd /tmp/hash-gen
$ go mod init
```

```
go: cannot determine module path for source directory /tmp/hash-gen

Example usage:
        'go mod init example.com/m' to initialize a v0 or v1 module
        'go mod init example.com/m/v2' to initialize a v2 module

Run 'go help mod init' for more information.
```

Try one of the following instead:

```
$ go mod init hash-gen
$ go mod init github.com/alexellis/hash-gen
```

It's also possible to use Go modules and vendoring at the same time:

```
$ cd /tmp/
$ git clone https://github.com/alexellis/hash-browns
$ cd hash-brown
$ rm -rf vendor
$ go mod vendor
$ ls vendor/
```

Now see that the `vendor` folder has been created.

You can ask Go to build using the vendor folder instead of its local packages cache at $GOPATH/pkg by adding -mod vendor to the go build command.

```
$ go build -mod vendor
```

So which approach should you use?

Vendoring has the advantage of keeping all your source and dependencies in one place, it also doesn't require anything to be downloaded to execute a build. With some projects such as Kubernetes controllers this means saving time and bandwidth. If you have any private dependencies, vendoring can work out as the easiest option, especially when building inside a container or CI pipeline.

Using modules without vendoring seems to be trendy, with many projects removing their vendor folder. It has the advantage of making code reviews easier since there are far fewer files that change in a pull request or commit. Working with private Go modules can be a source of contention, so you have been warned.

See also: [Go modules reference docs](#)

Create your first Go program

You can create a Go program from memory, but I find it more useful to use an IDE like [Visual Studio Code](#). Visual Studio Code has a number of plugins that it can install to format your code, tidy it up and add any paths for modules or packages that you import. The automation is not as comprehensive as a paid IDE like [Jetbrains' GoLand](#), but is fast and free.

Create a folder `first-go` then, save the following as `main.go`:

```go
package main

func main() {

}
```

Now you have two ways to run the program.

1. Run `go run main.go`
2. Run `go build` and then run the resulting binary `./first-go`

Now print a console statement using `fmt.Printf`. This will import the `fmt` package at the top of the file, and VSCode will automatically run a number of Go plugins to update the `main.go` file for you.
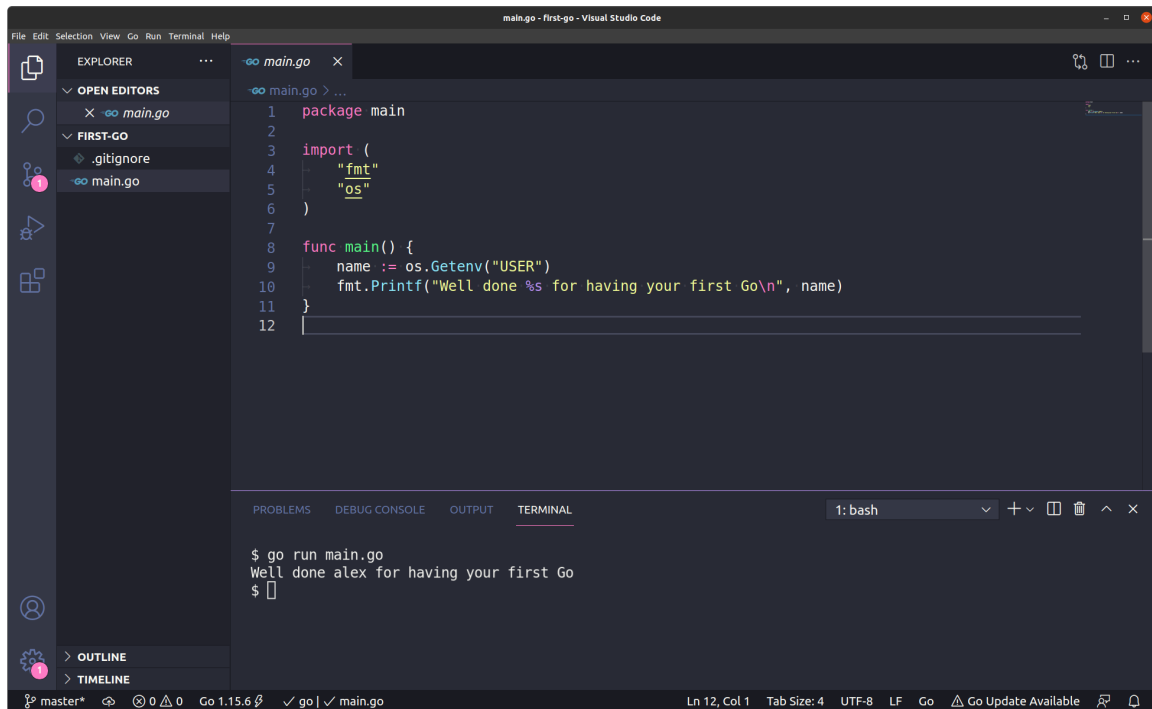
```go
package main

import (
    "fmt"
    "os"
)

func main() {
    name := os.Getenv("USER")
    fmt.Printf("Well done %s for having your first Go\n", name)
}
```

One of the programs run by VScode is `gofmt` and it's an example of how Go's opinionated design makes it easy to move between code-bases without learning new conventions. You can run it against a text file using `gofmt -w -s main.go`.

Note the use of the `os.Getenv` call to look-up your username from the `USER` environment variable.

Note for Windows users: try replacing "USER" with "USERNAME"

A screenshot of Visual Studio Code and our first-go app

Now, run the program again and you should see output as per above.

Add an external dependency

Whilst the standard library is extensive, it's likely that you will need to use an external dependency. Dependencies in Go are made available as source code that can be linked to your program, which differs from C# or Java where compiled assets (DLLs or JARs) are downloaded. For this reason, you'll usually see that a Go package has a path to where it can be downloaded with git.

You can use webhooks to receive events from third-party APIs such as GitHub.com or Stripe. Most of the time the webhooks are sent with a digest in the header, that you can use to verify that the message hasn't been tampered with.

HMAC uses a symmetric key that both sender/receiver share ahead of time. The sender will generate a hash when wanting to transmit a message - this

data is sent along with the payload. The recipient will then sign the payload with the shared key again. And if the hash matches then the payload is assumed to be from the sender.

Quote from [alexellis/hmac](#)

Let's use my library to create a digest of our own message, and then show how to verify it after.

To begin with, we should initialise a Go module for our application. The Go modules file `go.mod` lists external dependencies that we want our program to depend upon.

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/validate-hmac/

go mod init
go: creating new go.mod: module github.com/alexellis/validate-hmac
```

Review the contents of `go.mod`:

```
module github.com/alexellis/validate-hmac

go 1.15
```

It shows the path or name of this module, along with the minimum Go version required. In the past all code needed to be placed with the `$GOPATH` i.e. `$HOME/go/src/github.com/alexellis/validate-hmac`, but with go modules, the go.mod file can make this explicit, instead of implicit (read from the path).

Create main.go:

```go
package main

import (
    "fmt"

    "github.com/alexellis/hmac"
)

func main() {
```

```
    input := []byte(`input message from API`)
    secret := []byte(`so secret`)

    digest := hmac.Sign(input, secret)
    fmt.Printf("Digest: %x\n", digest)
}
```

According to the Go docs, `%x` used in `fmt.Printf()` uses a base 16, with lower-case letters for a-f, and is a way that we can share the raw computed bytes in a human readable format.

Now run the program:

```
$ go run main.go
go: finding module for package github.com/alexellis/hmac
go: found github.com/alexellis/hmac in github.com/alexellis/hmac v0.(

Digest: 17074131772d763bc4a360a6e4cb1a5ad1a98764
```

The digest `17074131772d763bc4a360a6e4cb1a5ad1a98764` was printed which is a hash that can be sent with the original input. Any user with the secret can compute another hash and if they match, that user will know the message was from us.

You'll see that the Go module is resolved, then downloaded from GitHub, and updated in go.sum and go.mod:

```
$ cat go.mod

module github.com/alexellis/validate-hmac

go 1.15

require github.com/alexellis/hmac v0.0.0-20180624211220-5c52ab81c0de

$ cat go.sum
github.com/alexellis/hmac v0.0.0-20180624211220-5c52ab81c0de h1:jiPEv
github.com/alexellis/hmac v0.0.0-20180624211220-5c52ab81c0de/go.mod h
```

The go.sum pins the specific version of the module for later use.

Now let's verify the digest we received using the library:

```go
package main

import (
    "fmt"

    "github.com/alexellis/hmac"
)

func main() {

    input := []byte(`input message from API`)
    secret := []byte(`so secret`)

    digest := hmac.Sign(input, secret)
    fmt.Printf("Digest: %x\n", digest)

    err := hmac.Validate(input, fmt.Sprintf("sha1=%x", digest), strin
    if err != nil {
        panic(err)
    }

    fmt.Printf("Digest validated.\n")
}
```

Run the program with `go run main.go` and you'll see it pass as expected.

Make use of flags

Go has a built-in flag parsing library. We will explore an alternative library, Cobra, in a later chapter. I've used Cobra in several projects but even now, I always start with the simplest thing that will work. Then later, if it is warranted, I'll move to a more complex, but powerful library than the standard library's flags package.

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/validate-hmac-flags/

go mod init
go: creating new go.mod: module github.com/alexellis/validate-hmac-fl
```

In our previous example we created hashes by hard-coding the input and secret, but that isn't going to be useful for our users. Let's use flags instead, so that we can run:

```
go run main.go
  -message="my message" \
  -secret="terces os"
```

Adapt the solution to use `flag.StringVar` to define each flag, followed by `flag.Parse`:

```go
package main

import (
    "flag"
    "fmt"
)

func main() {

    var inputVar string
    var secretVar string

    flag.StringVar(&inputVar, "message", "", "message to create a dig
    flag.StringVar(&secretVar, "secret", "", "secret for the digest")
    flag.Parse()

    fmt.Printf("Computing hash for: %q\nSecret: %q\n", inputVar, secr
}
```

This is a good time to let you know about the `%q` variable, which adds quotes around any string we print. I use it to make sure there are no extra lines or whitespace within a value.

```
$ go run main.go -message="my message" -secret="terces os"

Computing hash for: "my message"
Secret: "terces os"
```

Now let's take the value from the flag, and pass it into the hash generation code.

```go
        digest := hmac.Sign([]byte(inputVar), []byte(secretVar))
        fmt.Printf("Digest: %x\n", digest)
```

Run the whole program and see the digest generated:

```
$ go run main.go -message="my message" -secret="terces os"
Computing hash for: "my message"
Secret: "terces os"
Digest: 2aefe2f570c0e61bc346b9fd3b6fbb3107ebfdd9
```

Try the same message with a different secret:

```
$ go run main.go -message="my message" -secret="so secret"
Computing hash for: "my message"
Secret: "terces os"
Digest: d82655d25497765dc774e8e9cde8795858cb9f8e
```

Now, what if no secret is given as a flag value? How are you going to validate that?

```go
if len(secretVar) == 0 {
    panic("--secret is required")
}
```

Try adding the above, which looks at the length of the string:

```
$ go run main.go -message="my message"
panic: --secret is required

goroutine 1 [running]:
main.main()
        /home/alex/go/src/github.com/alexellis/validate-hmac-flags/ma
exit status 2
```

But we can get around this by setting the secret to " " whitespace, so how do you handle that?

```go
if len(strings.TrimSpace(secretVar)) == 0 {
    panic("--secret is required")
}
```

We can simply wrap the value with `strings.TrimSpace` which will remove any whitespace and handle that case for us.

Here's the complete example:

```go
package main

import (
    "flag"
    "fmt"
    "strings"

    "github.com/alexellis/hmac"
)

func main() {

    var inputVar string
    var secretVar string

    flag.StringVar(&inputVar, "message", "", "message to create a dig
    flag.StringVar(&secretVar, "secret", "", "secret for the digest")
    flag.Parse()

    if len(strings.TrimSpace(secretVar)) == 0 {
        panic("--secret is required")
    }

    fmt.Printf("Computing hash for: %q\nSecret: %q\n", inputVar, secr

    digest := hmac.Sign([]byte(inputVar), []byte(secretVar))
    fmt.Printf("Digest: %x\n", digest)
}
```

As an extra-work task, why don't you add a mode, so that your CLI can be used for both generating and validating hashes?

You could either add a flag with `flag.BoolVar` such as `-generate=true/false`, or a string mode like `-mode=generate/validate`.

Define a separate package or library

Until now, we have written code within a single package called `main`. The `main` package is required to make a program runnable on its own, however sometimes

you'll want to write a library so that code can be shared between programs.

You may also have a project with a `main` package and an additional named package for code organisation. Packages can be tested separately.

Generally, starting a `variable`, `function` or `struct` with a capital letter means that it will be available outside the package to other consumers.

We'll create a package that can perform a similar task to the `dir` or `ls` shell commands.

Create a new project:

```
$ export GH_USERNAME="alexellis"
$ mkdir -p $GOPATH/src/github.com/$GH_USERNAME/multiple-packages/
$ cd $GOPATH/src/github.com/$GH_USERNAME/multiple-packages/

$ go mod init
go: creating new go.mod: module github.com/alexellis/multiple-package
```

Now create a folder for the package called `cmd`:

```
$ mkdir -p $GOPATH/src/github.com/$GH_USERNAME/multiple-packages/cmd
```

Then create `ls.go` inside it:

```go
package cmd
```

Now you have defined a package called `cmd`, but it doesn't have any functions or structs exported.

```go
package cmd

func ExecuteLs(path string) (string, error) {

    return "", nil
}
```

The command can then be imported by a new `main` package using the path `github.com/alexellis/multiple-packages/cmd`. In some cases, like the HMAC library we used earlier, there is no need for the project to have its own `main` entrypoint because it will only be consumed by third-parties as a function.

Create `main.go` in the root folder:

```go
package main

import (
    "fmt"
    "os"

    "github.com/alexellis/multiple-packages/cmd"
)

func main() {
    wd, err := os.Getwd()
    if err != nil {
        fmt.Fprintf(os.Stderr, "unable to get working directory: %s",
            err.Error())
    }

    res, err := cmd.ExecuteLs(wd)
    if err != nil {
        fmt.Fprintf(os.Stderr, "unable list files in %s, error: %s",
            wd, err.Error())
    }

    fmt.Printf("%s\n", res)
}
```

Now populate the `ls.go` program with the rest of its code:

```go
package cmd

import (
    "fmt"
    "os"
)

func ExecuteLs(path string) (string, error) {

    entries, err := os.ReadDir(path)
    if err != nil {
        return "", err
    }
    output := fmt.Sprintf("Files in %s\n", path)
```

```
    output += "Name\tDirectory\t\n"

    for _, e := range entries {
        output += fmt.Sprintf("%s\t%v\n", e.Name(), e.IsDir())
    }

    return output, nil
}
```

Did you know? Unit tests are written in the same package as the source code, so if you have tests for `main`, you'd place them in the root folder, and any tests for the `cmd` package would be placed within the `cmd/` folder. In a later chapter, you'll learn about unit testing in Go.

And then try it out:

```
$ go run .
Files in /Users/alex/src/github.com/alexellis/multiple-packages
Name    Directory
cmd     true
go.mod  false
main.go false
multiple-packages        false
```

If you'd like to make use of this program on your system, you can copy it to your `$PATH` variable or run `go install`, to have it placed in `$GOPATH/bin/`

```
$ export GH_USERNAME="alexellis"
$ cd $GOPATH/src/github.com/$GH_USERNAME/multiple-packages
$ go install
$ cd /tmp/
$ multiple-packages
Files in /tmp
Name    Directory
...
```

If you receive a "not found" error here, then you need to add the `$GOPATH/bin/` to your `$PATH` variable on your system.

Taking it further:

- If you'd like to align the text into a table, why not try the [TabWriter](#)?
- How would you use flags and packages to add other commands and execute them according to the user's input?

[Derek](#) is a GitHub bot that I wrote to make it easier to automate open source maintenance and to delegate permissions to other contributors through comments. Derek is a good example of multiple packages, why not read the source code and see how his packages are exported and consumed in the `main.go` file.

Cross-compile for different operating systems

In the previous chapter we built a simple Go program that used an environment variable to print output, but the binary we produced could only be run on the same operating system and CPU architecture.

If you have another computer with the same operating system, you can copy the binary to that computer and run it without any changes. If you are using a Mac, but want to run the program on a Raspberry Pi or Windows desktop, you will need to use Go's built-in cross-compilation options to produce different binaries.

You need to be aware of the target Operating System and the target CPU architecture.

Go supports a number of Operating Systems, however these are the most used:

- Windows
- MacOS
- Linux

Go programs also need to be built for a specific CPU architecture and version. Most of the time this is a detail you can avoid, but if you expect your programs to be run on an Arm processor like a Raspberry Pi or an AWS Graviton server, you will need to bear this in mind too.

- Regular PCs and cloud, covering both Intel and AMD CPUs - `x86_64`
- 32-bit Arm devices like the Raspberry Pi are called `armhf`, `armv7` or `armv6l`
- 64-bit Arm servers and devices are called `arm64` or `aarch64`

If you don't specify any options, your binary will be built for the computer you are using, otherwise you can customise the options and output filename.

Go build arguments

Here's how to build our `first-go` program for Windows, Linux and MacOS:

```
GOOS=linux go build -o first-go
GOOS=darwin go build -o first-go-darwin
GOOS=windows go build -o first-go.exe
```

To build for an Arm CPU, just add the following:

```
GOARCH=arm GOOS=linux go build -o first-go-arm
GOARCH=arm64 GOOS=linux go build -o first-go-arm64
```

If you want to debug an issue with your binary, you can use the bash command `file` to see how it was built:

```
file first-go
first-go: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), s
linked, not stripped

file first-go-arm
first-go-arm: ELF 32-bit LSB executable, ARM, EABI5 version 1 (S
statically linked, not stripped

file first-go-arm64
first-go-arm64: ELF 64-bit LSB executable, ARM aarch64, version
statically linked, not stripped
```

If you use the bash command `du` you can see the size of the binary, which includes debugging symbols.

```
du -h ./first-go
```

```
2.0M    ./first-go
```

This file can sometimes be made smaller by stripping the debug symbols out.

```
GOOS=linux go build  -a -ldflags "-s -w"
```

```
du -h first-go
1.4M    first-go
```

```
first-go: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), s
stripped
```
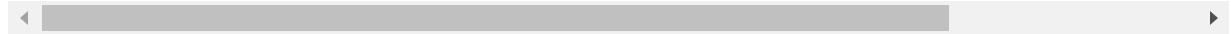
Note the smaller size, and the "stripped" output from the "file" command.

CGO and portability

CGO enables the creation of Go packages that call C code.

"CGO" is Go's ability to link to C/C++ libraries and is sometimes required to work with certain libraries like SQLite and certain Linux cryptographic packages which are not available in Go's standard library.

CGO makes portability more of a challenge and is beyond the scope of this book. My personal recommendation is to avoid it, all costs, if you plan to cross-compile your applications.

A good way to ensure that you are not using any CGO is through the `CGO_ENABLED=0` environment variable that you can prefix to your builds.

See also: [Go docs on CGO](#) and [cgo is not Go by Dave Cheney](#)

Fetch JSON from a HTTP endpoint

This has to be one of the most common tasks for a new Go programmer. You have installed Go, and have started to develop a CLI, now you need to interface with an external API and make a HTTP "GET" call. It's very likely that the API will be using JSON.

We will be querying an endpoint provided for free that tells us how many astronauts are currently in space and what their names are.

We'll be using HTTP and JSON from Go's standard library, and unlike with other languages, we will not need to install any external packages.

Get started

Make yourself a folder for a new project (you can use your Github username here):

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/get-json/
```

Now save a new file named `main.go`:

```
package main

func main() {

}
```

This program is effectively hello-world without any print statements.

Go ahead and run the program, you have two ways to do this:

```
go run main.go
```

Or build a binary, then run it.

```
go build
./get-json
```

Now add a print statement:

```go
package main

import "fmt"

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

Now you can build and run the code again, note that you can also change the output of the binary when compiled:

```
go build -o get-json
./get-json
```

Go offers a convention-based formatting of files, you can format a file and its indentation with:

```
gofmt -s -w main.go
```

The -w command writes the change to the file.

As we explored in an earlier chapter, Go applications can also be cross-compiled for other operating systems without any further changes, here's how you can build the above code for Windows:

```
GOOS=windows go build -o get-json.exe
```

Learn to parse JSON

Let's go on to parse some JSON, in Go we can turn a JSON document into a struct which is useful for accessing the data in a structured way. If a document doesn't fit into the structure it will throw an error.

Let's take an API from the Open Notify group - it shows the number of people in space along with their names.

[People in Space JSON](#)

It looks a bit like this:

```
{"people": [{"craft": "ISS", "name": "Sergey Rizhikov"}, {"craft
```

According to Wikipedia:

> In computer science, marshalling or marshaling (US spelling) is the process of transforming the memory representation of an object into a data format suitable for storage or transmission. It is typically used when data must be moved between different parts of a computer program or from one program to another.

If you're coming from Python, Ruby or Node, then the "Unmarshal" behaviour in Go is usually used whenever you would have parsed text to JSON.

Go takes a reference to a struct, and then tries to marshal the input into the datastructure. The opposite operation is "Marshal" which serializes and object into a text or binary representation.

We will be importing the `fmt` package for the `Println` function and the `encoding/json` package so that we can `Unmarshal` the JSON text into a struct.

Create `main.go`:

```go
package main

import (
    "encoding/json"
    "fmt"
)

type people struct {
```

```go
        Number int `json:"number"`
}

func main() {
    text := `{"people": [{"craft": "ISS", "name": "Sergey Rizhik
    textBytes := []byte(text)

    p := people{}
    err := json.Unmarshal(textBytes, &p)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(p.Number)
}
```

We have hard-coded the JSON response from Open Notify so that we can work on one chunk of behaviour at a time.

The `json.Unmarshal` function works with a `[]byte` type instead of a string so we use `[]byte(stringHere)` to create the format we need.

In order to make use of a `struct` to unmarshal the JSON text we normally need to decorate it with some tags that help the std library understand how to map the properties:

```go
type people struct {
    Number int `json:"number"`
}
```

> The property names need to begin with a capital letter which marks them as *exportable* or *public*. If your struct's property is the same in the JSON you should be able to skip the annotation.

Another thing to notice is that we pass the address of the new / empty people struct into the method. You can try removing the `&` symbol, but the value will be set in a different copy of the empty struct. This may seem odd if you are coming from languages that pass parameters *by reference*.

Retrieve the JSON via HTTP

Now we can parse a JSON document matching that of our API, let's go on and write a HTTP client to fetch the text from the Internet.

Go has a built-in HTTP client in the `net/http` package, but it has a problem with long timeouts and there are [some well-known articles](#) recommending that you set a timeout on your request explicitly.

There are more concise ways of creating a HTTP request in Go, but by adding a custom timeout we will harden our application.

Now create main.go:

```go
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "time"
)

type people struct {
    Number int `json:"number"`
}

func main() {

    url := "http://api.open-notify.org/astros.json"

    spaceClient := http.Client{
        Timeout: time.Second * 2, // Timeout after 2 seconds
    }

    req, err := http.NewRequest(http.MethodGet, url, nil)
    if err != nil {
```

```go
        log.Fatal(err)
    }

    req.Header.Set("User-Agent", "spacecount-tutorial")

    res, err := spaceClient.Do(req)
    if err != nil {
        log.Fatal(err)
    }

    if res.Body != nil {
        defer res.Body.Close()
    }

    body, err := ioutil.ReadAll(res.Body)
    if err != nil {
        log.Fatal(err)
    }

    p := people{}
    err = json.Unmarshal(body, &p)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(p.Number)
}
```

If you're used to writing Node.js, then you will be used to most I/O operations returning a nullable error. The same is the case in Go and checking for errors here has made the code quite verbose. You may want to refactor elements of the code into separate methods - one to download and one unmarshal the JSON data.

It may not always make sense to attempt to parse the HTTP response, one case is if the HTTP code returned is a non-200 number. Here's how you can access the HTTP code from the response:

```go
    fmt.Printf("HTTP status: %s\n", res.Status)
```

If you are unsure whether the JSON schema is stable and expect it, at times, to return invalid JSON, then you could extend the parsing code so that it prints out the value received in the error. That way, you can understand not only that parsing failed, but the reason why.

```go
err := json.Unmarshal(body, &p)
if err != nil {
    log.Fatalf("unable to parse value: %q, error: %s",
  string(body), err.Error())
}
```

There are many bots and automated applications that scrape websites and access APIs. In order to be a good citizen on the Internet, you should be aware of the accepted rules and standards.

There's one more important line I wanted to highlight. I've set a User-Agent in the HTTP request's header. This lets remote servers understand what kind of traffic it is receiving. Some sites will even reject empty or generic User-Agent strings.

```go
req.Header.Set("User-Agent", "spacecount-tutorial")
```

With some APIs, you may need to use a specific User-Agent, or to simulate the value a browser would pass.

Learn more about the User-Agent header: [Mozilla on User-Agents](#)

If you decide to access HTML pages using your Go program, you should read up on other mechanisms [like robots.txt](#):

> A robots.txt file tells search engine crawlers which pages or files the crawler can or can't request from your site. This is used mainly to avoid overloading your site with requests; it is not a mechanism for keeping a web page out of Google.

Here is what the code may look like when extracted to its own function and then called from the `main()` function.

```go
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
)

type people struct {
    Number int `json:"number"`
}

func main() {

    apiURL := "http://api.open-notify.org/astros.json"

    people, err := getAstros(apiURL)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%d people found in space.\n", people.Number)
}
```

Then our function would change as follows:

```go
func getAstros(apiURL string) (people, error) {
    p := people{}
    req, err := http.NewRequest(http.MethodGet, apiURL, nil)
    if err != nil {
        return p, err
    }

    req.Header.Set("User-Agent", "spacecount-tutorial")

    res, err := http.DefaultClient.Do(req)
    if err != nil {
        return p, err
    }
```

```go
    if res.Body != nil {
        defer res.Body.Close()
    }

    body, err := ioutil.ReadAll(res.Body)
    if err != nil {
        return p, err
    }

    err = json.Unmarshal(body, &p)
    if err != nil {
        return p, err
    }

    return p, nil
}
```

Note that we have the same amount of `return` statements, but there is only one log statement. Our code has become more useful to others, since not every caller may want to call `log.Fatal` when there is an error, for instance if the server returned an error, the caller may have wanted to retry the request instead.

One other thing you can do to reduce the line-count, is to "in-line" the `if` statement:

```go
    p := people{}
    if err := json.Unmarshal(body, &p); err != nil {
        return err
    }
```

This makes for more succinct code, but also means that the scope of the `err` variable is limited to only the `if` statement and won't leak into the function's scope.

Extracting additional data from the JSON

The JSON also has an element or object for each person aboard a craft.

Define a struct for the person element:

```go
type person struct {
  Name  string `json:"name"`
}
```

Then add it to your `people` struct

```go
type people struct {
    Number int        `json:"number"`
    Person []person `json:"people"`
}
```

Update your `main.go` to print the new fields:

```go
func main() {
    apiURL := "http://api.open-notify.org/astros.json"

    people, err := getAstros(apiURL)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%d people found in space.\n", people.Number)

    for _, p := range people.Person {
        fmt.Printf("Let's wave to: %s\n", p.Name)
    }
}
```

Here's the result on 5th March 2021:

```
7 people found in space.
Let's wave to: Sergey Ryzhikov
Let's wave to: Kate Rubins
Let's wave to: Sergey Kud-Sverchkov
Let's wave to: Mike Hopkins
Let's wave to: Victor Glover
Let's wave to: Shannon Walker
Let's wave to: Soichi Noguchi
```

We have now written some code to fetch a resource over HTTP, parse it as JSON, handle errors and to make it reusable. We then went on to harden

and make the code more useful by extracting a function.

Five ways to write an awesome CLI

We're having a renaissance of (Command Line Interfaces) CLIs - every programming language from Node.js to Go to less fashionable ones like .NET all have CLIs and developers love them.

A command-line interface should be something that feels awesome to use, and not something that is jarring or in any way surprising. You should pay attention to the style, user-experience and approach used by the most popular CLIs, and learn lessons.

> CLIs (command-line interfaces) are text-based interfaces for your applications which are easily automated, fast to work with and can be combined with other CLIs to create workflows.

Concepts carry across: almost every productivity application has a picture of a floppy disk. What does that do? There's no need to guess or read the docs. It's a save button. In the same way, `-v` means *print verbose* output for this command.

When done well, a CLI: * Is easy to get started with * Has just enough to get the job done * Interoperates with other CLIs (through pipes or files) * Doesn't surprise the user

So here are five ways that you can make sure that your next CLI is one that developers love to use, and tell their friends about.

Pick Go

It's probably no surprise that when reading a book about Go, that I'm advocating for writing CLIs using Go, but here's why.

I prefer the OpenFaaS CLI which was written in Go over the Serverless Framework Inc CLI which was written in JavaScript with Node.js:

- Compiles to a single static binary

With Go you can provide a single static binary that contains your whole application or CLI for your chosen platform. To target a different CPU or OS you just pass in an environmental override when building. With JavaScript, you have to install Node.js, then install hundreds or thousands of tiny packages from `npm`.

Here's a binary for Linux, Mac, Windows and for a Raspberry Pi:

```
GOOS=linux go build -o cli
GOOS=darwin go build -o cli-darwin
GOOS=windows go build -o cli.exe
GOARCH=armv7 GOOS=linux go build -o cli-rpi
```

That's it - and there are more platforms available too - like FreeBSD. You won't need to install any dependencies and the final output binary can be tiny.

See the initial few chapters of the eBook for a deeper-dive into static binaries and how to support multiple architectures.

- Consistent style

Go is an opinionated language and while there may be some differences between which editors a project prefers - you will encounter a consistent standard for styling, formatting and build tools. Something like Node.js could involve any number of "task runners" or "transpilers" - or a more esoteric flavour of the language like TypeScript or CoffeeScript.

Go has a consistent style and was deliberately designed to be unambiguous. Why is that important? It makes it easy for developers to contribute, without having to invest much time in learning the code-base, toolchain or style chosen. At time of writing, the CLI for OpenFaaS has 810 commits, and the whole project has over 8000 commits from 340 individual developers.

- Fast on every platform

Go is rapid on all platforms, without an initial lag. If you've ever used a CLI written in Python or Node.js, then you are probably used to seeing an initial lag every time you use the CLI. This is made worse on platforms like the Raspberry Pi, where just running `node hello-world.js` can take 1.5-3.0 seconds to start running the first instruction.

- Easy to work with REST APIs

Go includes a no-nonsense `http` client and has built-in support for working with `xml`, `json` and binary formats. There's also a very good library for working with YAML which is used in OpenFaaS here: stack.go

There may be reasons why Node.js or another language is more suitable for your use-case. If you've already decided to build a CLI in Go - then you can reap the benefits of a fast binary that's small and easy to distribute.

Did you know that you can distribute an `npm` package that installs a Go CLI? I first saw this being done by Cloudflare, and we adapted the same approach for the openfaas CLI `npm install -g @openfaas/faas-cli`.

The code downloads the latest version of the `faas-cli` binary from the GitHub releases page and then places it in the correct location in the filesystem.

Parse flags & arguments

The standard Go library includes a `flags` package that can be used to parse flags or arguments in a couple of lines of code.

```go
package main

import (

    "flag"
    "fmt"
    "os"
)


func main() {
```

```go
    var image string
    flag.StringVar(&image, "image", "", "Docker image")
    flag.Parse()

    if len(image) == 0 {
        fmt.Fprintf(os.Stderr, "You must specify a Docker image
    }

    fmt.Printf("Your Docker image was: %s", image)
}
```

From my experiences in the world of enterprise development - the average C# developer would have written his own string parser and then created a ConsoleApp as an entry-point then a DLL Library for the parsing and one more for the application code.

> When you contrast that to the snippet above, Go is unpretentious. You can have a single file that compiles to a tiny binary and only uses the standard library. And be done with that.

I suggest you start with the `flags` package with each new CLI, and when you reach its limits, only then consider migrating to something more modular like [Cobra](#).

Cobra is used by Docker, Kubernetes and the OpenFaaS projects and means that handlers/commands can live in separate files or modules. It also makes documenting each command really simple. [Checkout the code here for our 'list functions' command](#).

Another advantage of using "cobra" is that it has a verb noun syntax. This helped us when designing our user experience. We went from a somewhat jarring experience to something more fluid:

```
faas-cli -deploy \
    -image=functions/alpine \
    -name=cat \
    -fprocess=/bin/cat
```

To:

```
faas-cli deploy \
    --image=functions/alpine \
    --name=cat \
    --fprocess=/bin/cat
```

The `-deploy` flag, became a sub-command called `deploy` and allowed us to move the code into its own package. Cobra handles the parsing and help messages for all the sub-commands you want to add.

We also took feedback about managing multiple functions and created a YAML format which meant the CLI command could be as simple as:

```
faas deploy -f stack.yml
```

Or

```
faas deploy -f \
              https://raw.githubusercontent.com/openfaas/faas-
cli/master/stack.yml
```

> Tip: pick verbs / commands carefully and ask other people if they make sense. If something jars with you then it's probably wrong. It can take several iterations but what you're aiming for is an intuitive experience.

Automate everything

Create an automated build using a free and public CI platform like [GitHub Actions](#) so that contributors or collaborators know whether their changes can be integrated.

Use GitHub releases to track changes in the project and milestones. You can set up a post-build action in GitHub to publish binary artifacts automatically for every platform you target.

If you have a Docker image - publish that on the Docker store at the same time as pushing a new release artifact. Tools like GitHub can obfuscate credentials and keys so that they do not show up in logs.

Go projects are very easy to build in Docker. Here's an example from the [inlets-operator](#), a Kubernetes controller which creates tunnel servers for Kubernetes services you want to expose in a private cluster:

- [Dockerfile inlets-operator](#)

Make sure you use [multi-stage builds](#) so you ship a lean image.

For the various projects I maintain, we tend to split the build into a "CI" or "test" action:

- [ci-only.yaml for the inlets-operator](#)

And a publish action:

- [publish.yaml for the inlets-operator](#)

While it's easy to start with - you do not want to be building 3-5 binary files on your own machine for every release then uploading them to a web-form every time you need an update.

Integrate with package managers

If you want to make it easy for your target audience to be able to get hold of your CLI then you need to make that as easy as possible. This means integrating with package managers.

- `brew` - if your target audience is the developer - then a large percentage of them may have a Mac. brew is a package manage which means most CLIs are only one command away `brew install faas` for instance
- Linux - for Linux there are many package managers so I've chosen to build a download utility which can be run as root or as a regular user. To get the OpenFaaS CLI: `curl -sL https://cli.openfaas.com | sh` - or to get hold of Docker: `curl -sL https://get.docker.com | sh`.
- Windows - Windows users have good options available for shells including Git Bash (my preferred option) and the Windows Subsystem

for Linux. WSL can use the Linux `curl` utility, but if you are targeting "point and click" developers you may want to create a "double-click" installer.

Whatever you do - make sure it's automated and evaluate each package manager you support. Will you have time to maintain the integration? Is it provided by the project or by the community? What if it breaks? For instance - we maintain a guide on how to [upgrade our brew formula](#).

Tip: Make sure the update/release cycle of chosen packages can match the cadence of your project. Whilst package managers are seen as a best practice, the versions of applications you'll find there can get out of date very quickly. Direct downloads over HTTPS using your GitHub project's release page is probably the easiest and most convenient approach.

You can also offer an auto-update or a help message to show users how to keep up to date. Here's an example from [inletsctl](#), an open source tool for creating ad-hoc tunnel servers.

```
inletsctl update
You can update inletsctl with the following:

# For Linux/MacOS:
curl -SLfs https://inletsctl.inlets.dev | sudo sh

# For Windows (using Git Bash)
curl -SLfs https://inletsctl.inlets.dev | sh

# Or download from GitHub: https://github.com/inlets/inletsctl/
Thanks for using inletsctl!
```
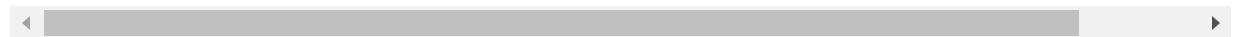
The faas-cli has code that can inform you whether your version is out of date, feel free to browse the repository to see how it works.

```
faas-cli version

  ___                _____              ____
 /   \              |     |            /    |
```

```
 _   _           ___       ___   ____            _  ___
| | | |'_ \ / _ \ '_ \| |_ / _` |/ _` \  __ \
| |_| | |_) | __/ | | |  _| (_| | (_| |__) |
 \___/| .__/ \___|_| |_|_|  \__,_|\__,_|____/
      |_|
```
```
CLI:
 commit:  6b5e7a14a598527063c7c05fb4187739b6eb6d38
 version: 0.13.6
Your faas-cli version (0.13.6) may be out of date. Version: 0.13
available on GitHub.
```

Accept contributions and gather feedback

Provide an easy way for people to provide feedback and contributions. A CLI should be designed for its operators - so make it easy for them to submit code changes or suggestions.

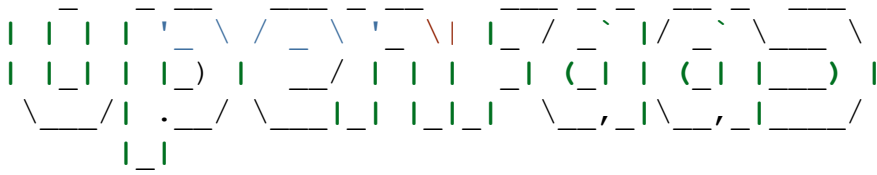User feedback is essential, but when running an Open Source Software project I often hear people struggle to understand how or if their software is being used.

Basic feedback can be gathered from the download statistics on GitHub or `brew`. Several key projects have started gathering usage data automatically using an analytics platform - examples include: `brew`, Visual Studio Code, Atom and Docker for Mac/Windows. If you go down this route make sure you provide a privacy policy and comply with any potential data-protection regulations.

We once considered using Google Analytics to collect information about OpenFaaS CLI usage, but never committed to it. You can read more in: [issue #108](). It was going to show us:

- which commands were used
- what programming languages functions were used
- operating system / CLI version / location in the world

Even if you don't decide to add analytics to your CLI, you can still gather feedback whether your tool is for internal and private work use, or open-source and freely available.

Now, Go and write your CLI

There are many reasons to make your next CLI awesome with Go. From the speed of compilation and execution, to the availability of built-in or high-quality packages, to the ease of automation.

If you already have a CLI, and want to take it to the next level, check out a couple of CLIs that I maintain which have been built up with experience and lessons learned over the past 5-6 years:

- [faas-cli](#) - OpenFaaS CLI for creating, deploying and managing serverless functions
- [arkade](#) - open source marketplace for Kubernetes, and tool downloader

Even if your CLI is only going to be used at work for internal use, it's worth thinking about how you're going to accept feedback or code contributions from colleagues and teammates.

Writing unit-tests in Golang

In a previous chapter we explored how to interact with a HTTP endpoint. We downloaded a response over the Internet, and then parsed the JSON response into a struct.

But how would we go about testing code that we write in Go? What can Go's own test runner offer us? And what kind of opinions does it prescribe?

Testing opinions

Just like Go has strong opinions and conventions for formatting, naming and package layout, it has the same for unit-testing.

Go actually has its own built-in testing command called `go test` and a package `testing` which combine to give a minimal but complete testing experience. The standard tool-chain also includes benchmarking `go test -bench` and statement-based code coverage `go test -cover` similar to [NCover](.NET) or [Istanbul](Node.js).

If you are coming to Go from a language like Ruby, Java or .NET, then you may be surprised to find that there is no assertions library, and [certain books](#) go as far as to discourage any use of them.

To a JavaScript developer, writing in a BDD style, testing a Sum method would look something like this:

```javascript
var expect = require('expect');

describe('Calculator', function() {
  describe('#multiply()', function() {
    it('should return -10 when 10 is multiplied by -1', function
      const calc = new Calculator()
      expect(calc.multiply(10, -1)).to.eql(-10)
    });
  });
})
```

```
});
```

An alternative to the above may be to use an `assert.AreEqual()` type of method from the Java / C# world. The Go developers do not take this approach, and if you want to see some examples of unit testing in Go, check out this test from the OpenFaaS CLI which parses a YAML file: stack_test.go.

In Go, the same test would look more like this:

```go
func TestMultiply_ForANegativeNumber(t *testing.T) {
    c:= Calculator{}
    want := -10
    got := c.Multiply(10, -1)

    if want != got {
        t.Fatalf("want %d, but got %d", want, got)
    }
}
```

In some languages, additional libraries add support for annotations that allow a test to run for various values.

See this example from NUnit:

```csharp
[Test]
[TestCase("hello@example.com")]
[TestCase("hello+plus@example.com")]
[TestCase("hello.with.dots@example.com")]
public void ValidateEmail_supports_real_emails(string emailAddre
{
    // ARRANGE
    var sut = new EmailValidator();

    // ACT
    var result = sut.ValidateEmail(emailAddress);

    // ASSERT
    Assert.That(result.IsValid, Is.True);
}
```

In Go, this would be achieved through a test-table:

```go
func TestMultiply_VariousNumbers(t *testing.T) {
    c:= Calculator{}

    cases := []struct {
        a int
        b int
        want int
        name string
    }{
        {-10, 1, -10, "negative case"},
        {10, 1, 10, "positive case"},
        {10, 0, 0, "zero case"},
    }

    for _, tc := range cases {
        t.Run(tc.name, func(t *testing.T) {
            want := tc.want
            got := c.Multiply(tc.a, tc.b)

            if want != got {
                t.Fatalf("want %d, but got %d", want, got)
            }
        })
    }
}
```

If you'd like to see a more complex example, the OpenFaaS CLI where it determines the correct choice of a number of variables: priority_test.go

When using assertions instead of pure Go:

- tests can feel like they're written in a different language (RSpec/Mocha for instance)
- errors can be cryptic "assert: 0 == 1"
- pages of stack traces can be generated
- tests stop executing after the first assert fails - masking patterns of failure

There are third-party libraries that replicate the feel of RSpec or Assert. See also stretchr/testify.

Disclaimer: I want you to feel comfortable with Go's approach, even if you are not used to it. Do us both a favour, and don't reach for your nearest BDD library for Go. Why? Because one of the strongest properties of Go is consistency, the fact that any Go developer should be able to pick up another Go codebase without learning new libraries and styles.

Testing the Sum method

Create a folder for the exercise:

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/sum-testing
```

Here is an example of a method we want to test in the `main` package. We have defined an exported function called `Sum` which takes in two integers and adds them together.

Save `main.go`:

```go
package main

import "fmt"

func sum(x int, y int) int {
    // This code has a bug that adds 1
    return x + y + 1
}

func main() {
    x := 5
    y := 5
    fmt.Printf("%d + %d = %d\n", x, y, sum(x, y))
}
```

Try the application::

```
go run main.go
5 + 5 = 11
```

Notice there is a bug, we will need to unit test this code.

We need to write our unit test in a separate file, this is usually in the same package and folder as the code being tested, but it doesn't have to be. The reason why keeping the tests with the code is useful is because:

1. They can be imported as a single package
2. It means you can unit-test private methods, without making them public

Save `main_test.go`:

```go
package main

import "testing"

func Test_sum(t *testing.T) {
    x := 5
    y := 5

    want := 10
    got := sum(x, y)
    if want != got {
        t.Logf("Testing a sum of: %d %d", x, y)
        t.Errorf("Sum was incorrect want: %d, but got: %d", want
    }
}
```

Then run `go test ./` to run all the tests in the current package:

```
go test ./

--- FAIL: Test_sum (0.00s)
    /home/alex/go/src/github.com/alexellis/sum-testing/main_test
    /home/alex/go/src/github.com/alexellis/sum-testing/main_test
```

Notice the following about the unit test:

- The only parameter to the test method is: `t *testing.T`
- It begins with the word `Test` followed by a word or phrase, either with an underscore, or the function under test.
- The `t` variable is used to signal a failure.
- `t.Log` can be used to provide debug information for any test
- The file must be saved in a file named `something_test.go` such as: `addition_test.go`

Let's fix the test, edit the code in `sum` so that it doesn't add a rogue `1` to any number and run the tests again:

```
go test ./
ok      github.com/alexellis/sum-testing      0.001s
```

There are two ways to launch tests for a package. These methods work for unit tests and integration tests alike.

1. Within the same directory as the test:

```
go test
```

Or with a given path:

```
go test ./pkg/parser
```

This picks up any files matching packagename_test.go

or

2. By fully-qualified package name

```
go test github.com/alexellis/golangbasics1
```

You can run this from whatever folder you like.

You can add an additional test by using a unique name.

```
package main
```

```go
import "testing"

func Test_sum_odd_numbers(t *testing.T) {
    x := 5
    y := 5

    want := 10
    got := sum(x, y)
    t.Logf("Testing a sum of: %d %d", x, y)
    if want != got {
        t.Errorf("Sum was incorrect want: %d, but got: %d", want
    }
}

func Test_sum_negative_numbers(t *testing.T) {
    x := -5
    y := -5

    want := -10
    got := sum(x, y)
    t.Logf("Testing a sum of: %d %d", x, y)
    if want != got {
        t.Errorf("Sum was incorrect want: %d, but got: %d", want
    }
}
```

Do you notice that there's some repetition between both tests? We will tackle that in the next section.

You have now run a unit test in Go, for a more verbose output type in `go test -v` and you will see the PASS/FAIL result of each test including any extra logging produced by `t.Log`.

```
go test -v
=== RUN   Test_sum_odd_numbers
    main_test.go:11: Testing a sum of: 5 5
--- PASS: Test_sum_odd_numbers (0.00s)
=== RUN   Test_sum_negative_numbers
    main_test.go:23: Testing a sum of: -5 -5
```

```
--- PASS: Test_sum_negative_numbers (0.00s)
PASS
ok      github.com/alexellis/sum-testing      0.001s
```

I would advise you to remove as much logging from unit tests as possible, since when you have many, it will become noisy and distracting.

Adding a test table

The concept of "test tables" is a set (slice array) of test input and output values. Here is an example for the `Sum` function:

```go
package main

import "testing"

func Test_sum(t *testing.T) {
    tables := []struct {
        x int
        y int
        want int
    }{
        {1, 1, 2},
        {1, 2, 3},
        {2, 2, 4},
        {5, 2, 7},
    }

    for _, table := range tables {
        got := sum(table.x, table.y)
        if got != table.want {
            t.Errorf("Sum of (%d+%d) was incorrect, got: %d, war
                table.x, table.y, got, table.want)
        }

    }
}
```

If you want to trigger the errors to break the test then alter the `Sum` function to return `x * y`.

```
$ go test -v
=== RUN   Test_sum
--- FAIL: Test_sum (0.00s)
    table_test.go:19: sum of (1+1) was incorrect, got: 1, want:
    table_test.go:19: sum of (1+2) was incorrect, got: 2, want:
    table_test.go:19: sum of (5+2) was incorrect, got: 10, want:
FAIL
exit status 1
FAIL    github.com/alexellis/t6 0.013s
```

When should you use a test table? My opinion is after you have enough duplication of single unit tests that it's becoming difficult to maintain. Perhaps after 2-3 very similar tests, you may want to invest 10 minutes in converting them all to a test table. When tests have duplication, they can be easier to maintain in a table. There are pros and cons, so weigh it up case by case.

Party tricks for "go test"

In this section we'll explore a couple of party tricks that `go test` brings along. Then we'll put everything together that we've learned so far in a larger example, where we isolate a dependency.

Statement coverage

The `go test` tool has built-in code-coverage for statements. To try it with out example above type in:

```
$ go test -cover
PASS
coverage: 50.0% of statements
ok      github.com/alexellis/golangbasics1  0.009s
```

High statement coverage is better than lower or no coverage, but metrics can be misleading. We want to make sure that we're not only executing statements, but that we're verifying behaviour and output values and raising errors for discrepancies. If you delete the "if" statement from our previous test it will retain 50% test coverage but lose its usefulness in verifying the behaviour of the "Sum" method.

Generating an HTML coverage report

If you use the following two commands you can visualise which parts of your program have been covered by the tests and which statements are lacking:

```
go test -cover -coverprofile=c.out
go tool cover -html=c.out -o coverage.html
```

You can now open coverage.html in a web-browser.

Benchmarking your code

Benchmarking is another part of the `go test` workflow that can be useful for advanced use-cases.

In the following code copied from the [stdlib](), a hash is generated with bcrypt.

```go
package main

import (
    "bytes"
    "crypto/aes"
    "crypto/cipher"
    "encoding/hex"
    "fmt"
    "io"
)

func Hash(keyValue, secret string) string {
    key, _ := hex.DecodeString(keyValue)
```

```go
    bReader := bytes.NewReader([]byte(secret))

    block, err := aes.NewCipher(key)
    if err != nil {
        panic(err)
    }

    var iv [aes.BlockSize]byte
    stream := cipher.NewOFB(block, iv[:])

    var out bytes.Buffer

    writer := &cipher.StreamWriter{S: stream, W: &out}
    if _, err := io.Copy(writer, bReader); err != nil {
        panic(err)
    }

    return fmt.Sprintf("%x\n", out.Bytes())
}
```

A benchmark can be written for it as follows:

```go
package main

import (
    "testing"
)

func BenchmarkHash10(b *testing.B) {
    for n := 0; n < b.N; n++ {
        key := "1234567890"
        Hash("6368616e67652074686973207061737373", key)
    }
}

func BenchmarkHash20(b *testing.B) {
    for n := 0; n < b.N; n++ {
        key := "12345678901234567890"
        Hash("6368616e67652074686973207061737373", key)
```

```
    }
}
```

This defines two tests for keys of differing lengths (10) and (20) characters. You can then run the benchmarks as follows:

```
$ go test -bench=.
goos: linux
goarch: amd64
pkg: github.com/alexellis/bench-comp
BenchmarkHash10-8           816800                 1314 ns/op
BenchmarkHash20-8           876960                 1347 ns/op
PASS
ok      github.com/alexellis/bench-comp         2.288s
```

Benchmarking can also be useful for comparing the performance go code on a certain CPU. Arm CPUs have traditionally lacked optimizations for cryptography, so this benchmark should run much slower on a Raspberry Pi.

Go doesn't ship your tests

In addition, it may feel unnatural to leave files named `addition_test.go` in the middle of your package. Rest assured that the Go compiler and linker will not ship your test files in any binaries it produces.

Here is an example of finding the production vs test code in the net/http package we used in the previous Golang basics tutorial.

```
$ go list -f={{.GoFiles}} net/http
[client.go cookie.go doc.go filetransport.go fs.go h2_bundle.go

$ go list -f={{.TestGoFiles}} net/http
[cookie_test.go export_test.go filetransport_test.go header_test
```

Stress testing your tests

Your tests can also be run more than once, I've found this to show up issues with race conditions or shared state in code that I've written.

Simply add `-test.count=100` to run your tests 100 times.

At times, you'll also see the tests refuse to run because the results are `(cached)`. In this instance, adding `-test.count=1` can get around that problem.

Running tests in parallel

It may appear that your unit tests run well and in a deterministic order, but often when they are run in parallel problems can be found before they hit production. This is because when we test our code locally, it's usually with a lower load, or serially. On a production system, we may have many users doing things in parallel.

You may also have a suite of tests that are slow to run, in that case running in parallel may save time. 10x tests that take 1s to complete could run in 10 seconds serially, or in 1 second if run with a parallelism of 10.

Consider this example:

```go
package main

import (
    "testing"
    "time"
)

func Test_Slow1(t *testing.T) {
    time.Sleep(1 * time.Second)
}

func Test_Slow2(t *testing.T) {
    time.Sleep(1 * time.Second)
}

func Test_Slow3(t *testing.T) {
```

```
        time.Sleep(1 * time.Second)
}


func Test_Slow4(t *testing.T) {
        time.Sleep(1 * time.Second)
}


func Test_Slow5(t *testing.T) {
        time.Sleep(1 * time.Second)
}

$ go test ./ -v
=== RUN   Test_Slow1
--- PASS: Test_Slow1 (1.00s)
=== RUN   Test_Slow2
--- PASS: Test_Slow2 (1.01s)
=== RUN   Test_Slow3
--- PASS: Test_Slow3 (1.00s)
=== RUN   Test_Slow4
--- PASS: Test_Slow4 (1.01s)
=== RUN   Test_Slow5
--- PASS: Test_Slow5 (1.00s)
PASS
ok      github.com/alexellis/go-slow    5.343s
```

To run the tests in parallel, `t.Parallel()` has to be added to the test function, then the `-test.parallel=N` parameter has to be passed in along with it.

```
package main


import (
        "testing"
        "time"
)


func Test_Slow1(t *testing.T) {
        t.Parallel()
        time.Sleep(1 * time.Second)
}
```

```go
func Test_Slow2(t *testing.T) {
    t.Parallel()
    time.Sleep(1 * time.Second)
}


func Test_Slow3(t *testing.T) {
    t.Parallel()
    time.Sleep(1 * time.Second)
}


func Test_Slow4(t *testing.T) {
    t.Parallel()
    time.Sleep(1 * time.Second)
}


func Test_Slow5(t *testing.T) {
    t.Parallel()
    time.Sleep(1 * time.Second)
}
```

Then run:

```
$ go test -v . -test.count=1 \
  -test.parallel 5
=== RUN   Test_Slow1
=== PAUSE Test_Slow1
=== RUN   Test_Slow2
=== PAUSE Test_Slow2
=== RUN   Test_Slow3
=== PAUSE Test_Slow3
=== RUN   Test_Slow4
=== PAUSE Test_Slow4
=== RUN   Test_Slow5
=== PAUSE Test_Slow5
=== CONT  Test_Slow1
=== CONT  Test_Slow5
=== CONT  Test_Slow4
=== CONT  Test_Slow2
=== CONT  Test_Slow3
--- PASS: Test_Slow3 (1.00s)
--- PASS: Test_Slow1 (1.00s)
--- PASS: Test_Slow5 (1.00s)
```

```
--- PASS: Test_Slow4 (1.00s)
--- PASS: Test_Slow2 (1.00s)
PASS
ok      github.com/alexellis/go-slow    1.342s
```

This test could also have been written as a test table.

```go
package main

import (
    "testing"
    "time"
)

func Test_Slow1(t *testing.T) {
    t.Parallel()
    cases := []struct {
        Name     string
        Duration time.Duration
    }{
        {Name: "Case 1", Duration: time.Second * 1},
        {Name: "Case 2", Duration: time.Second * 1},
        {Name: "Case 3", Duration: time.Second * 1},
        {Name: "Case 4", Duration: time.Second * 1},
        {Name: "Case 5", Duration: time.Second * 1},
    }

    for _, c := range cases {
        tc := c
        t.Run(tc.Name, func(t *testing.T) {
            t.Parallel()
            t.Logf("%s sleeping..", tc.Name)
            sleepFn(tc.Duration)
            t.Logf("%s slept", tc.Name)
        })
    }
}

func sleepFn(duration time.Duration) {
    time.Sleep(duration)
}
```

Note the use of the `tc` variable which captures the value of the test case in into the `t.Run` closure.

As a bonus, it's more readable and it took marginally less time.

```
$ go test -v . -parallel 5
=== RUN   Test_Slow1
=== PAUSE Test_Slow1
=== CONT  Test_Slow1
=== RUN   Test_Slow1/Case_1
=== PAUSE Test_Slow1/Case_1
=== RUN   Test_Slow1/Case_2
=== PAUSE Test_Slow1/Case_2
=== RUN   Test_Slow1/Case_3
=== PAUSE Test_Slow1/Case_3
=== RUN   Test_Slow1/Case_4
=== PAUSE Test_Slow1/Case_4
=== RUN   Test_Slow1/Case_5
=== PAUSE Test_Slow1/Case_5
=== CONT  Test_Slow1/Case_1
=== CONT  Test_Slow1/Case_4
    main_test.go:25: Case 4 sleeping..
=== CONT  Test_Slow1/Case_5
    main_test.go:25: Case 5 sleeping..
=== CONT  Test_Slow1/Case_3
    main_test.go:25: Case 3 sleeping..
=== CONT  Test_Slow1/Case_2
    main_test.go:25: Case 2 sleeping..
=== CONT  Test_Slow1/Case_1
    main_test.go:25: Case 1 sleeping..
    main_test.go:27: Case 1 slept
=== CONT  Test_Slow1/Case_2
    main_test.go:27: Case 2 slept
=== CONT  Test_Slow1/Case_5
    main_test.go:27: Case 5 slept
=== CONT  Test_Slow1/Case_4
    main_test.go:27: Case 4 slept
=== CONT  Test_Slow1/Case_3
    main_test.go:27: Case 3 slept
--- PASS: Test_Slow1 (0.00s)
    --- PASS: Test_Slow1/Case_2 (1.01s)
    --- PASS: Test_Slow1/Case_1 (1.01s)
```

```
    --- PASS: Test_Slow1/Case_5 (1.01s)
    --- PASS: Test_Slow1/Case_4 (1.01s)
    --- PASS: Test_Slow1/Case_3 (1.01s)
PASS
ok      github.com/alexellis/go-slow    1.260s
```

So we went from just over 5 seconds to just under 1.5 seconds. This may
seem marginal, but if you have lots of tests or run them often, it can be
more efficient.

Compile the tests to a binary

Something that I learned only after using Go for several years, was that the
tests can be compiled to a binary, just like your CLI or HTTP server can be.

Why would this be useful?

One scenario is if you have tests that run against external resources, such as
a validator or certifier tool. You can version, release, and distribute that
without losing the capabilities of `go test`.

Let's try an example?

```
$ git clone https://github.com/alexellis/go-execute
$ cd go-execute
```

Then test it the "normal" way:

```
$ go test -v ./...
=== RUN   TestExec_WithShell
--- PASS: TestExec_WithShell (0.00s)
=== RUN   TestExec_CatTransformString
--- PASS: TestExec_CatTransformString (0.00s)
=== RUN   TestExec_CatWC
--- PASS: TestExec_CatWC (0.00s)
=== RUN   TestExec_WithEnvVars
--- PASS: TestExec_WithEnvVars (0.00s)
=== RUN   TestExec_WithEnvVarsInheritedFromParent
--- PASS: TestExec_WithEnvVarsInheritedFromParent (0.00s)
=== RUN   TestExec_WithEnvVarsAndShell
```

```
--- PASS: TestExec_WithEnvVarsAndShell (0.00s)
PASS
ok      github.com/alexellis/go-execute/pkg/v1  0.798s
```

Now let's create a binary that can be distributed and run on another machine. We'll just add the `-c` (compile) flag.

```
$ go test ./... -c
$ v1.test
```

Then you can run the test binary and pass in similar flags to what we saw earlier.

```
$ ./v1.test
PASS

$ ./v1.test -test.v
# Produces the same output as "go test -v"
```

See `./v1.test --help` for more.

For more on the basics read the [Golang testing docs](#).

Isolating dependencies

What's the difference between a unit test and an integration test?

> The difference between unit and integration tests is that unit tests usually isolate dependencies that communicate with network, disk etc. Unit tests normally test only one thing such as a function.

Isolation in Go can be achieved through interfaces, but if you're coming from a C# or Java background, they look a little different in Go. Interfaces are implied rather than enforced which means that concrete classes don't need to know about the interface ahead of time.

That means we can have very small interfaces such as [io.ReadCloser](#) which has only two methods made up of the Reader and Closer interfaces:

```
        Read(p []byte) (n int, err error)
```

Reader interface

```
        Close() error
```

Closer interface

If you are designing a package to be consumed by a third-party then it makes sense to design interfaces so that others can write unit tests to isolate your package when needed. When these do not exist, it can create a lot of work and maintenance for a consumer.

An interface can be substituted in a function call. So if we wanted to test this method, we'd just have to supply a fake / test-double class that implemented the Reader interface.

Create a folder to work in:

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/fake-1/
```

Save main.go:

```go
package main

import (
    "fmt"
    "io"
    "io/ioutil"
    "log"
)

type FakeReader struct {
    payload  []byte
    position int
}

func (f *FakeReader) Read(p []byte) (n int, err error) {
    p = f.payload
```

```go
        f.position += len(f.payload)
        if f.position > len(f.payload) {
            return 0, io.EOF
        }

        return len(f.payload), nil
}


func (f *FakeReader) SetFakeBytes(p []byte) {
        f.position = 0
        f.payload = p
}


func ReadAllTheBytes(reader io.Reader) []byte {
        data, err := ioutil.ReadAll(reader)
        if err != nil {
            log.Fatal(err)
        }
        return data
}


func main() {
        fakeReader := &FakeReader{}
        want := []byte("when called, return this data")
        fakeReader.SetFakeBytes(want)
        got := ReadAllTheBytes(fakeReader)
        fmt.Printf("%d/%d bytes read.\n", len(got), len(want))
}
```

Before implementing your own abstractions (as above) it is a good idea to search the Golang docs to see if there is already something you can use. In the case above we could also use the standard library in the bytes package:

```go
        func NewReader(b []byte) *Reader
```

The testing/iotest package provides some Reader implementations which are slow or which cause errors to be thrown half way through reading. These are ideal for testing the resilience of your system. You should be prepared to ask questions like: "What would happen if there was a timeout?"

Example of isolating dependencies

Let's refactor the code that we used to fetch JSON data from a HTTP endpoint. If you haven't done this exercise yet, you can go back and try it now.

First we will create a new folder, with a sub-package called astros which will contain the refactored code and its interface.

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/isolate-astros/astr
```

We need to create an interface so that we can substitute the HTTP call with our own data.

The `GetWebRequest` interface below specifies the following function to return data, for a given URL.

```go
type GetWebRequest interface {
    FetchBytes(url string) ([]byte, error)
}
```

> Note: interfaces are inferred rather than explicitly decorated onto a struct. This is different from languages like C# or Java.

Why did we create an interface? Because we want to be able to "fake" or swap the HTTP call with test data.

Save `isolate-astros/astros/types.go`

```go
package astros

type People struct {
    Number int `json:"number"`
}

type GetWebRequest interface {
    FetchBytes(url string) ([]byte, error)
}
```

Now save the implementation as isolate-astros/astros/http_astros.go:

```go
package astros

import (
    "io/ioutil"
    "net/http"
)

type LiveGetWebRequest struct {
}

func (LiveGetWebRequest) FetchBytes(url string) ([]byte, error) {
    req, err := http.NewRequest(http.MethodGet, url, nil)
    if err != nil {
        return []byte{}, err
    }

    req.Header.Set("User-Agent", "spacecount-tutorial")

    res, err := http.DefaultClient.Do(req)
    if err != nil {
        return []byte{}, err
    }
    defer res.Body.Close()

    body, err := ioutil.ReadAll(res.Body)
    if err != nil {
        return []byte{}, err
    }

    return body, nil
}
```

Next, let's write the new main.go file that uses the interface and the real LiveGetWebRequest implementation:

```go
package main
```

```go
import (
    "encoding/json"
    "fmt"

    "github.com/alexellis/isolate-astros/astros"
)

func getAstronauts(getWebRequest astros.GetWebRequest) (int, err
    url := "http://api.open-notify.org/astros.json"
    body, err := getWebRequest.FetchBytes(url)
    if err != nil {
        return 0, err
    }
    peopleResult := astros.People{}

    if err := json.Unmarshal(body, &peopleResult); err != nil {
        return 0, err
    }
    return peopleResult.Number, err
}

func main() {
    liveClient := astros.LiveGetWebRequest{}
    number, err := getAstronauts(liveClient)
    if err != nil {
        panic(err)
    }
    fmt.Println(number)
}
```

Test it out to make sure it all works:

```
go run .
7
```

Next create the unit test `main_test.go`:
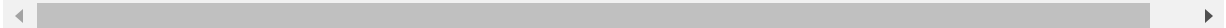
```go
package main

import "testing"
```

```go
type testWebRequest struct {
}

func (testWebRequest) FetchBytes(url string) ([]byte, error) {
    return []byte(`{"number": 2}`), nil
}

func TestGetAstronauts(t *testing.T) {
    want := 3
    got, err := getAstronauts(testWebRequest{})
    if err != nil {
        t.Fatal(err)
    }
    if want != got {
        t.Errorf("People in space, want: %d, got: %d.", want, go
    }
}
```

Note the struct is defined in the same package as the tests. This is making it private, because it does not need to be accessed anywhere else.

The test then goes on to verify the count of people using a new instance of the struct as its parameter. We see hard-coded JSON will be returned with "2" people being in space.

Run the unit tests:

```
go test ./
--- FAIL: TestGetAstronauts (0.00s)
    main_test.go:19: People in space, want: 3, got: 2.
FAIL
FAIL    github.com/alexellis/isolate-astros   0.002s
FAIL
```

Now go ahead and fix the test data so that it returns three instead of 2 people, and run the test again. Remember, to always start with a test that fails, so that you know that your test is doing something.

Choosing what to abstract

The above unit test is effectively only testing the `json.Unmarshal` function and our assumptions about what a valid HTTP response body would look like. This abstracting may be OK for our example, but our code coverage score will be low.

It is also possible to do lower level testing to make sure that the HTTP request being made is correct, or that we created a GET request instead of a POST.

Fortunately Go has a set of helper functions for creating fake HTTP servers and clients.

Going further:

- explore the http/httptest package
- and refactor the test above to use a fake HTTP client.
- what is the test coverage percentage like before and after?

Unit-testing a HTTP server

Unit-testing a HTTP handler is slightly more involved than using an interface, and took me a while to get to grips with.

Let's explore the HTTP handler we want to test:

```go
// Copyright (c) OpenFaaS Author(s). All rights reserved.
// Licensed under the MIT license.

package auth


import (
    "crypto/subtle"
    "net/http"
)

type BasicAuthCredentials struct {
    User      string
```

```
        Password string
}

// DecorateWithBasicAuth enforces basic auth as a middleware wit
func DecorateWithBasicAuth(next http.HandlerFunc, credentials *E
    return func(w http.ResponseWriter, r *http.Request) {

        user, password, ok := r.BasicAuth()

        const noMatch = 0
        if !ok ||
            user != credentials.User ||
            subtle.ConstantTimeCompare([]byte(credentials.Passwc

            w.Header().Set("WWW-Authenticate", `Basic realm="Res
            w.WriteHeader(http.StatusUnauthorized)
            w.Write([]byte("invalid credentials"))
            return
        }

        next.ServeHTTP(w, r)
    }
}
```

This code applies a HTTP basic-auth header to a server, so that you cannot log in, unless you have the credential.

We need to somehow create a fake `http.ResponseWriter` and receive what is written to it, and also create a fake `http.Request` so that we can pass in different values to test whether the authentication check works.

The `httptest.NewRecorder()` can be used to record the response being sent back by our handler.

We can then use the `httptest.NewRequest` function to create a fake request that won't go to a real HTTP server.

```
func Test_AuthWithValidPassword_Gives200(t *testing.T) {

    handler := func(w http.ResponseWriter, r *http.Request) {
```

```go
    handler :=    func(w http.ResponseWriter, r *http.Request) {
        io.WriteString(w, "<html><body>Hello World!</body></html
    }
    w := httptest.NewRecorder()

    wantUser := "admin"
    wantPassword := "password"
    r := httptest.NewRequest(http.MethodGet, "http://localhost:8

    r.SetBasicAuth(wantUser, wantPassword)
    wantCredentials := &BasicAuthCredentials{
        User:     wantUser,
        Password: wantPassword,
    }

    decorated := DecorateWithBasicAuth(handler, wantCredentials)
    decorated.ServeHTTP(w, r)

    wantCode := http.StatusOK

    if w.Code != wantCode {
        t.Fatalf("status code, want: %d, got: %d", wantCode, w.C
    }

    gotAuth := w.Header().Get("WWW-Authenticate")
    wantAuth := ``
    if gotAuth != wantAuth {
        t.Fatalf("WWW-Authenticate, want: %s, got: %s", wantAuth
    }
}
```

To test for the forbidden HTTP code (403), we can then do the following:

```go
func Test_AuthWithInvalidPassword_Gives403(t *testing.T) {

    handler := func(w http.ResponseWriter, r *http.Request) {
        io.WriteString(w, "<html><body>Hello World!</body></html
    }

    w := httptest.NewRecorder()
```

```go
    wantUser := "admin"
    wantPassword := "test"
    r := httptest.NewRequest(http.MethodGet, "http://localhost:8
    r.SetBasicAuth(wantUser, wantPassword)

    wantCredentials := &BasicAuthCredentials{
        User:     wantUser,
        Password: "",
    }

    decorated := DecorateWithBasicAuth(handler, wantCredentials)
    decorated.ServeHTTP(w, r)

    wantCode := http.StatusUnauthorized
    if w.Code != wantCode {
        t.Fatalf("status code, want: %d, got: %d", wantCode, w.C
    }

    gotAuth := w.Header().Get("WWW-Authenticate")
    wantAuth := `Basic realm="Restricted"`
    if gotAuth != wantAuth {
        t.Fatalf("WWW-Authenticate, want: %s, got: %s", wantAuth
    }
}
```

You can clone the Git repository and run the tests like this:

```
mkdir -p $GOPATH/src/github.com/openfaas/
cd $GOPATH/src/github.com/openfaas/
git clone --depth=1 https://github.com/openfaas/faas-provider
cd faas-provider

go test -v ./auth/
=== RUN   Test_AuthWithValidPassword_Gives200
--- PASS: Test_AuthWithValidPassword_Gives200 (0.00s)
=== RUN   Test_AuthWithInvalidPassword_Gives403
--- PASS: Test_AuthWithInvalidPassword_Gives403 (0.00s)
=== RUN   Test_ReadFromCustomLocation_AndNames
--- PASS: Test_ReadFromCustomLocation_AndNames (0.00s)
=== RUN   Test_ReadFromCustomLocation_DefaultNames
```

```
--- PASS: Test_ReadFromCustomLocation_DefaultNames (0.00s)
PASS
ok      github.com/openfaas/faas-provider/auth  0.002s
```

Work concurrently work with Goroutines

In computing, concurrency means being able to handle multiple tasks at once, but not necessarily working on them at the exact same time. This concept is often confused with parallel computing, the act of making progress on multiple tasks at the same time.

> Go's answer to concurrency is Goroutines. The documentation defines a "goroutine" as: "a lightweight thread managed by the Go runtime."

Why is that important? Operating System (OS) threads are typically expensive to create and coordinate. Go's approach is described as a "lightweight" thread, meaning that there's not necessarily a 1:1 relationship between Operating System (OS) threads and Goroutines, making them faster and more scalable.

For a deep dive into Goroutines and synchronisation techniques, you may also like [The Go Programming Language (Addison-Wesley)](#).

In this chapter, I want to cover some practical examples to show you how to get started today. I'll also mention a common pitfall that can catch even the most experienced Golang programmers.

We'll be covering: writing your first Goroutine, creating Goroutines within a loop, synchronisation through WaitGroups, error handling with Error groups, using channels and limiting the amount of work in progress.

A simple example with Goroutines

In this example, we have a `printLater` function which simulates a slow running call, slow enough that we may want to run several calls concurrently and collect the results at the end, rather than running each in serial and waiting longer.

```go
package main

import (
    "fmt"
    "os"
    "time"
)

func main() {
    printLater("Hello\n", time.Millisecond*100)
    printLater("World\n", time.Millisecond*100)
    printLater(os.Getenv("USER")+"\n", time.Millisecond*100)


}

func printLater(msg string, duration time.Duration) {
    time.Sleep(duration)
    fmt.Printf(msg)
}
```

When run, this program gives the following, and takes roughly 300ms to execute:

```
$ go run main.go
Hello
World
alex
```

Now imagine that the printLater method was actually the previous example of fetching and extracting JSON from a remote HTTP server. We can optimise the total running time, from 300ms to around 100ms by executing the HTTP calls concurrently using Goroutines.

```go
package main

import (
    "fmt"
    "os"
    "time"
)

func main() {
```

```
func main() {
    go func() {
        printLater("Hello\n", time.Millisecond*100)
    }()
    go func() {
        printLater("World\n", time.Millisecond*100)
    }()
    go func() {
        printLater(os.Getenv("USER")+"\n", time.Millisecond*100)
    }()
}

func printLater(msg string, duration time.Duration) {
    time.Sleep(duration)
    fmt.Printf(msg)
}
```

Now when you run this program, it will exit without any output. That is because the Goroutines execute asynchronously, and therefore need to be synchronised with the main program's execution flow.

You can synchronise Goroutines through the use of channels, or a number of other primitives in the sync package of the standard library. The WaitGroup is one of my preferred examples.

The WaitGroup primary purpose is to block and wait for a number of tasks to complete.

To use a WaitGroup, first, call the Add function with a number of tasks, then set up your Goroutines. Within each Goroutine, you will need to have it call Done when it has completed. Finally, within the function synchronising the work, call wg.Wait() and it will block execution there until all the routines have called Done and the outstanding tasks are set to 0.

```
package main

import (
    "fmt"
    "os"
    "sync"
```

```go
        "time"
)

func main() {
    wg := sync.WaitGroup{}
    wg.Add(3)

    go func() {
        defer wg.Done()
        printLater("Hello\n", time.Millisecond*100)
    }()
    go func() {
        defer wg.Done()
        printLater("World\n", time.Millisecond*100)
    }()
    go func() {
        defer wg.Done()
        printLater(os.Getenv("USER")+"\n", time.Millisecond*100)
    }()

    wg.Wait()
}

func printLater(msg string, duration time.Duration) {
    time.Sleep(duration)
    fmt.Printf(msg)
}
```

Now we see the code complete, and print all the messages. We can tell that the work is being executed concurrently because the messages are seemingly out of order.

```
$ go run main.go
World
Hello
alex
```

This example cannot handle errors, and it's important to note that if one Goroutine calls `panic()` then the whole application will fail. We will explore two approaches for error handling in a few moments.

Avoid a common mistake with Goroutines

Now let's explore a common mistake that is made with Goroutines, one which I also make from time to time.

Now that you have understood an approach to defining and synchronising Goroutines, it is tempting and sometimes more efficient to introduce a loop.

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    wg := sync.WaitGroup{}
    workItems := 5
    wg.Add(workItems)

    for i := 1; i <= workItems; i++ {
        go func() {
            defer wg.Done()
            printLater(
                fmt.Sprintf("Hello from %d\n", i),
                time.Millisecond*100)
        }()
    }

    wg.Wait()
}

func printLater(msg string, duration time.Duration) {
    time.Sleep(duration)
    fmt.Printf(msg)
}
```

Now run the example:

```
$ go run ./
Hello from 6
Hello from 6
Hello from 6
Hello from 6
Hello from 6
```

Note that the message reads `Hello from 6` for each of the 5 responses? The loop captured the final value of the `i` variable and used it for each Goroutine.

To solve the problem, we can either pass the `i` variable into the Goroutine as a parameter, or capture it within a closure using a new local variable.

Here's the first approach:

```
for i := 1; i <= workItems; i++ {
    go func(j int) {
        defer wg.Done()
        printLater(
            fmt.Sprintf("Hello from %d\n", j),
            time.Millisecond*100)
    }(i)
}
```

Notice the signature for our Goroutine changed as follows:

```
i := 1
go func(j int) {
  fmt.Printf("Value is: %d", j)
}(i)
```

In the second approach, we define a new local variable `j` to capture the value of `i`:

```
for i := 1; i <= workItems; i++ {
    j := i
    go func() {
        defer wg.Done()
        printLater(
            fmt.Sprintf("Hello from %d\n", j),
```

```
            time.Millisecond*100)
        }()
    }
```

In both cases, we get the output as expected:

```
Hello from 4
Hello from 3
Hello from 5
Hello from 1
Hello from 2
```

Error groups

Another way to synchronise work in parallel is through the unofficial "errgroup" package from the Golang team.

This becomes even more useful in the example above because any of our Goroutines can return an error and be captured without having to write code to synchronise messages over channels.

Not only can the errgroup run a number of Goroutines concurrently, but it can also run them serially in a pipeline, checking the error of each task before moving onto the next.

To use an *error group*, Import the `"golang.org/x/sync/errgroup"` package, then define a new `errgroup` with `g := errgroup.Group{}`, then call `g.Go(func() error { })` for each of the work items. You'll notice that the `g.Go()` function takes a function which can return an error. If there is no error, then return `nil`, otherwise return the error and the whole execution will exit and bubble up the error to the synchronising function.

```go
package main

import (
    "fmt"
    "os"
    "time"

    "golang.org/x/sync/errgroup"
```

```go
	golang.org/x/sync/errgroup
)

func main() {
	g := errgroup.Group{}
	g.Go(func() error {
		printLater("Hello\n", time.Millisecond*100)
		return nil
	})
	g.Go(func() error {
		// printLater("World\n", time.Millisecond*100)
		return fmt.Errorf("world failed")
	})
	g.Go(func() error {
		printLater(os.Getenv("USER")+"\n", time.Millisecond*100)
		return nil
	})

	err := g.Wait()
	if err != nil {

		fmt.Fprintf(os.Stderr, "error found: %s", err.Error())
		os.Exit(1)
	}
	fmt.Print("All work completed as expected")
}


func printLater(msg string, duration time.Duration) {
	time.Sleep(duration)
	fmt.Printf(msg)
}
```

When running the code, you'll see the following output:

```
$ go run ./
Hello
alex
error found: world failed
```

You can find out more about *error groups* in the package's docs

Sharing data between Goroutines

There are two approaches to sharing data between Goroutines in Go, whilst staying within the bounds of memory. The first would be to create a type with a locking mechanism, so that a reader or writer could obtain a lock to prevent two Goroutines trying to update the same data and the resulting race condition.

In the first example, our code may look similar to:

```go
package main

import (
    "sync"
)

type ScrapeRun struct {
    Result []ScrapeResult
    Lock   *sync.Mutex
}

type ScrapeResult struct {
    HTTPCode   int
    HTTPLength int
    Error      error
}
```

Within the main program flow, you would write something similar to:

```go
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "sync"
)

type ScrapeRun struct {
    Results map[string]ScrapeResult
```

```go
    Lock     *sync.Mutex
}

type ScrapeResult struct {
    HTTPCode   int
    HTTPLength int64
    Error      error
}

func scrape(url string, sr *ScrapeRun) {
    res, err := http.Get(url)
    if err != nil {
        sr.Lock.Lock()
        defer sr.Lock.Unlock()

        sr.Results[url] = ScrapeResult{
            Error:    err,
            HTTPCode: http.StatusBadGateway,
        }
        return
    }

    defer res.Body.Close()

    sr.Lock.Lock()
    defer sr.Lock.Unlock()

    length := res.ContentLength
    // Read whole body to find length, if not returned in
    // response
    if length == -1 {
        body, _ := ioutil.ReadAll(res.Body)
        length = int64(len(body))
    }
    sr.Results[url] = ScrapeResult{
        HTTPCode:   res.StatusCode,
        HTTPLength: length,
    }
}

func main() {
```

```go
    url1 := "https://inlets.dev/"
    url2 := "https://openfaas.com/"
    url3 := "https://openfas.com/"

    sr := ScrapeRun{
        Lock:    &sync.Mutex{},
        Results: make(map[string]ScrapeResult),
    }

    wg := sync.WaitGroup{}
    wg.Add(3)

    go func(u string) {
        defer wg.Done()
        scrape(url1, &sr)
    }(url1)

    go func(u string) {
        defer wg.Done()
        scrape(url2, &sr)
    }(url2)

    go func(u string) {
        defer wg.Done()
        scrape(url3, &sr)
    }(url3)

    wg.Wait()

    for k, v := range sr.Results {
        if v.Error != nil {
            fmt.Printf("- %s = error: %s\n", k, v.Error.Error())
        } else {
            fmt.Printf("- %s = [%d] %d bytes\n", k, v.HTTPCode,
        }
    }
}
```

This is the output from running the program:

```
$ go run .
- https://openfas.com/ = error: Get "https://openfas.com/": dial
- https://openfaas.com/ = [200] 32696 bytes
- https://inlets.dev/ = [200] 69805 bytes
```

Did you know? The `sync.Mutex` makes an exclusive lock on the object,
meaning that no other reads or writes are possible until the lock is released.
This can cause a bottleneck if there are both readers and writers of an
object.

The `sync.RWMutex` type can be used as a drop-in replacement for
`sync.Mutex`, where an arbitary number of readers can obtain a lock or a
single writer.

```go
package main

import (
    "sync"
)

func main() {
    lock := sync.RWMutex{}
    go readOperation(data, "key", &lock)
    go writeOperation(data, "key", "value", &lock)

    // Add synchronisation for the two routines here.
}

func readOperation(data map[string]string, key string, lock *syr
    lock.RLock()
    defer lock.RUnlock()
    fmt.Println(data["key"])
}

func writeOperation(data map[string]string, key, value string, l
    lock.Lock()
    defer lock.Unlock()
    data[key] = value
}
```

In this case, the read and write operations can happen at the same time, because the write lock will not block the function which reads the value.

What's a real-world use-case for this pattern? Perhaps you have an in-memory cache holding a list of active customers. The list needs to be downloaded over the network, which can be an expensive operation. The list is then shared with a number of Goroutines or handlers in your HTTP server. The read lock would be obtained for accessing the data, and the write lock would be used by something like a timer or a long-running loop that updates items within the cache.

The second approach would be for the Goroutines to simply output data, without sharing a type. The calling code can then aggregate the results, without needing to create a separate type with a lock. You'll see an example of this in the following section *Using channels for synchronisation*.

Using channels for synchronisation

If you have existing experience with Go, then you may have been surprised that I showed you the WaitGroup first, followed by an *Error group* without covering channels. An incomplete understanding of channels can lead to leaking Goroutines or deadlocks within your application, and in some cases a memory leak.

That said, I want to show you an example of how to run an asynchronous task, where it can either return a successful message or an error using a channel.

The `titleOf` function will download the HTML from a URI passed to it, and then look for the `<title>` line and return that to you.

Go has two types of channels:

By default channels are unbuffered, meaning that they will only accept sends (chan <-) if there is a corresponding receive (<- chan) ready to receive the sent value. Buffered channels accept a limited number of values without a corresponding receiver for those values.

From the [docs](#)

You can define a channel by using `make(chan type)`, so we create a channel for `string` and one for `error` in the sample.

Both channels are passed into the `titleOf` function, followed by a `select` statement, which will block until either of the channels has a message to be read.

We can pass the channels as follows: `go titleOf(uri, titleCh, errorCh)`

Then block with the following select statement:

```go
select {
case title := <-titleCh:
    fmt.Printf("Title: %s\n", title)
case err := <-errorCh:
    fmt.Printf("Error: %s\n", err.Error())
    os.Exit(1)
}
```

If you only had a single channel, then you could bypass the `select` statement, and use the `<-` receiver syntax.

```go
err := <-errorCh
fmt.Printf("Error: %s\n", err.Error())
os.Exit(1)
```

Here is the complete example:

```go
package main

import (
    "flag"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "strings"
)
```

```go
func main() {

    var uri string
    flag.StringVar(&uri, "uri", "", "uri to fetch")
    flag.Parse()

    titleCh := make(chan string)
    errorCh := make(chan error)

    go titleOf(uri, titleCh, errorCh)

    select {
    case title := <-titleCh:
        fmt.Printf("Title: %s\n", title)
    case err := <-errorCh:
        fmt.Printf("Error: %s\n", err.Error())
        os.Exit(1)
    }
}

func titleOf(uri string, titleCh chan string, errCh chan error)
    req, err := http.NewRequest(http.MethodGet, uri, nil)
    if err != nil {
        errCh <- err
        return
    }

    res, err := http.DefaultClient.Do(req)
    if err != nil {
        errCh <- err
        return
    }
    if res.Body != nil {
        defer res.Body.Close()

        res, _ := ioutil.ReadAll(res.Body)
        for _, l := range strings.Split(string(res), "\n") {
            if strings.Contains(l, "<title>") {
                titleCh <- l
                return
```

```
            }
        }
    }

    errCh <- fmt.Errorf("no title found")
}
```

Here's an example of running the workload, with three different websites and then a couple of error scenarios.

```
$ go run . -uri https://inlets.dev
Title:     <title>Inlets - The Cloud Native Tunnel</title>

$ go run . -uri https://github.com/alexellis/
Title:   <title>alexellis (Alex Ellis) · GitHub</title>

$ go run .
Error: Get "": unsupported protocol scheme ""
exit status 1

$ go run . -uri https://gitzub.com
Error: Get "https://gitzub.com": dial tcp: lookup gitzub.com: no
exit status 1
```

Using a context to handle timeouts

Now, there's also another potential problem here. Some websites may hang or keep a connection open indefinitely. Go provides a concept called "context" that can be passed into calls that make use of external resources like with our HTTP request from above.

> A Context carries a deadline, a cancellation signal, and other values across API boundaries.

You can view the docs for the context package here: [context](#).

Some library functions allow a Context to be passed via a parameter, but if one is not present, you can get the background context with:

```go
ctx := context.Background()
```

You may also see `context.TODO()` used which can be used when no context is available, or it's unclear which context to use.

Its API also has several utility methods for defining a cancellation policy:

```go
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

A parent context needs to be passed in, when the `cancel` function is called, any in-progress operations must be stopped and resources should be released.

```go
func WithDeadline(parent Context, d time.Time) (Context, CancelF
```

This function is similar to `WithCancel`, except there is a deadline time/date for some future date at which time the operation should be cancelled. This is useful with HTTP requests or when accessing an external system.

```go
func WithTimeout(parent Context, timeout time.Duration) (Context
```

The `WithTimeout` function is also similar, but provides a helper so that a timeout value can be passed in rather than an actual date in the future.

In the example below, you'll see how `WithTimeout` works.

```go
package main

import (
    "context"
    "flag"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "strings"
    "time"
)
```

```go
func main() {

    var uri string
    var timeout time.Duration

    flag.StringVar(&uri, "uri", "", "uri to fetch")
    flag.DurationVar(&timeout, "timeout", time.Second*5, "timeou

    flag.Parse()

    titleCh := make(chan string)
    errorCh := make(chan error)

    go titleOf(uri, timeout, titleCh, errorCh)

    select {
    case title := <-titleCh:
        fmt.Printf("Title: %s\n", title)
    case err := <-errorCh:
        fmt.Printf("Error: %s\n", err.Error())
        os.Exit(1)
    }
}

func titleOf(uri string, timeout time.Duration, titleCh chan str
    ctx, cancel := context.WithTimeout(context.Background(), tim
    defer cancel()

    req, err := http.NewRequestWithContext(ctx, http.MethodGet,
    if err != nil {
        errCh <- err
        return
    }

    res, err := http.DefaultClient.Do(req)
    if err != nil {
        errCh <- err
        return
    }

    if res.Body != nil {
```

```
if res.Body != nil {
    defer res.Body.Close()
    res, _ := ioutil.ReadAll(res.Body)
    for _, l := range strings.Split(string(res), "\n") {
        if strings.Contains(l, "<title>") {
            titleCh <- l
            return
        }
    }
}

errCh <- fmt.Errorf("no title found")
}
```

Now let's set the timeout to be a little more aggressive, like `10ms` and introduce a flag so that we can easily configure the value.

```
$ go run . -uri https://github.com/alexellis --timeout 10ms
Error: Get "https://github.com/alexellis": context deadline exce
exit status 1

$ go run . -uri https://github.com/alexellis --timeout 1000ms
Error: Get "https://github.com/alexellis": context deadline exce
exit status 1

$ go run . -uri https://github.com/alexellis --timeout 3000ms
Title:   <title>alexellis (Alex Ellis) · GitHub</title>
```

So it looks like we need a rather generous timeout to fetch my GitHub profile from a client in the UK.

Limiting concurrent work

So far, all the techniques that we've seen do not allow us to limit resources. At some point, you may exceed a limit like the number of files or HTTP connections that are allowed on your machine, or the number of database connections that your Database as a Service allows.

In those cases, it's useful to think of limiting work. A similar technique is implemented in the OpenFaaS CLI for building, pushing and deploying functions.

By default, the level of concurrency is 1, but it can be overridden, so that if you need to build 8 functions, you could have four of them being processed at the same time. You can think of this approach as a "window", 4 Goroutines will start, and receive 4/8 pieces of the work. When the work in progress drops to 3, another item from the remaining queue of 4 items will be processed until all the work is done and nothing is remaining.

The following pattern updates our `titleOf` method to run without a channel, and to return its results as `(string, error)`.

```go
func titleOf(uri string, timeout time.Duration) (string, error)
    ctx, cancel := context.WithTimeout(context.Background(), tim

    defer cancel()

    req, err := http.NewRequestWithContext(ctx, http.MethodGet,
    if err != nil {
        return "", err
    }

    res, err := http.DefaultClient.Do(req)
    if err != nil {
        return "", err
    }

    if res.Body != nil {
        defer res.Body.Close()
        res, _ := ioutil.ReadAll(res.Body)
        for _, l := range strings.Split(string(res), "\n") {
            if strings.Contains(l, "<title>") {
                return uri, nil
            }
        }
    }

    return "", fmt.Errorf("no title found")
}
```

We then add a workQueue concept and a number of workers (Goroutines) which read from the queue. A WaitGroup then allows each worker to signal when it has completed processing.

```go
package main

import (
    "context"
    "flag"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "os"
    "strings"
    "sync"
    "time"
)

func main() {

    uris := []string{"https://github.com/alexellis",
        "https://github.com/openfaas",
        "https://github.com/inlets",
        "https://github.com/docker",
        "https://github.com/kubernetes",
        "https://github.com/moby",
        "https://github.com/hashicorp",
        "https://github.com/civo",
    }

    var w int
    flag.IntVar(&w, "w", 1, "amount of workers")
    flag.Parse()

    titleMap, errs := fetch(uris, w)
    if len(errs) > 0 {
        for _, err := range errs {
            fmt.Fprintf(os.Stderr, "error %s", err.Error())
        }
```

```go
        os.Exit(1)
    }

    for k, v := range titleMap {
        fmt.Printf("%s has %s\n", k, v)
    }
}


func fetch(uris []string, workers int) (map[string]string, []err
    titles := make(map[string]string)

    var errs []error

    workQueue := make(chan string)

    wg := sync.WaitGroup{}
    wg.Add(workers)
    for i := 0; i < workers; i++ {
        worker := i
        go func(worker int, workQueue chan string) {
            log.Printf("[Worker: %d] Started work.", worker)

            for uri := range workQueue {
                log.Printf("[Worker: %d] Getting: %s", worker, u

                start := time.Now()
                title, err := titleOf(uri, time.Second*5)
                if err != nil {
                    errs = append(errs, err)
                }
                log.Printf("[Worker: %d] Got: %s (%.2fs)",
                    worker,
                    uri,
                    time.Since(start).Seconds())

                titles[uri] = title
            }
            log.Printf("[Worker: %d] Finished work.", worker)
            wg.Done()
        }(worker, workQueue)
    }
```

```
        go func() {
            for _, u := range uris {
                workQueue <- u
            }
            close(workQueue)


        }()


    wg.Wait()
    return titles, errs
}
```

The `-w` flag controls concurrency.

Observe the difference in output and time taken with a single worker and with 4.

```
$ time go run main.go -w 1
2021/06/18 18:42:30 [Worker: 0] Started work.
2021/06/18 18:42:30 [Worker: 0] Getting: https://github.com/alex
2021/06/18 18:42:32 [Worker: 0] Got: https://github.com/alexelli
2021/06/18 18:42:32 [Worker: 0] Getting: https://github.com/oper
2021/06/18 18:42:33 [Worker: 0] Got: https://github.com/openfaas
2021/06/18 18:42:33 [Worker: 0] Getting: https://github.com/inle
2021/06/18 18:42:33 [Worker: 0] Got: https://github.com/inlets (
2021/06/18 18:42:33 [Worker: 0] Getting: https://github.com/dock
2021/06/18 18:42:34 [Worker: 0] Got: https://github.com/docker (
2021/06/18 18:42:34 [Worker: 0] Getting: https://github.com/kube
2021/06/18 18:42:34 [Worker: 0] Got: https://github.com/kubernet
2021/06/18 18:42:34 [Worker: 0] Getting: https://github.com/moby
2021/06/18 18:42:35 [Worker: 0] Got: https://github.com/moby (0.
2021/06/18 18:42:35 [Worker: 0] Getting: https://github.com/hash
2021/06/18 18:42:35 [Worker: 0] Got: https://github.com/hashicor
2021/06/18 18:42:35 [Worker: 0] Getting: https://github.com/civc
2021/06/18 18:42:36 [Worker: 0] Got: https://github.com/civo (0.
2021/06/18 18:42:36 [Worker: 0] Finished work.
https://github.com/openfaas has   <title>OpenFaaS · GitHub</titl
https://github.com/inlets has   <title>inlets · GitHub</title>
https://github.com/docker has   <title>Docker · GitHub</title>
https://github.com/kubernetes has   <title>Kubernetes · GitHub</
https://github.com/moby has   <title>Moby · GitHub</title>
```

```
https://github.com/moby has    <title>Moby · GitHub</title>
https://github.com/hashicorp has    <title>HashiCorp · GitHub</ti
https://github.com/civo has    <title>Civo · GitHub</title>
https://github.com/alexellis has    <title>alexellis (Alex Ellis)


real    0m6.206s
user    0m0.556s
sys     0m0.188s
```

```
$ time go run main.go -w 4
2021/06/18 18:41:56 [Worker: 0] Started work.
2021/06/18 18:41:56 [Worker: 0] Getting: https://github.com/alex
2021/06/18 18:41:56 [Worker: 3] Started work.
2021/06/18 18:41:56 [Worker: 3] Getting: https://github.com/open
2021/06/18 18:41:56 [Worker: 2] Started work.
2021/06/18 18:41:56 [Worker: 1] Started work.
2021/06/18 18:41:56 [Worker: 1] Getting: https://github.com/dock
2021/06/18 18:41:56 [Worker: 2] Getting: https://github.com/inle
2021/06/18 18:41:57 [Worker: 2] Got: https://github.com/inlets (
2021/06/18 18:41:57 [Worker: 2] Getting: https://github.com/kube
2021/06/18 18:41:57 [Worker: 3] Got: https://github.com/openfaas
2021/06/18 18:41:57 [Worker: 3] Getting: https://github.com/moby
2021/06/18 18:41:57 [Worker: 1] Got: https://github.com/docker (

2021/06/18 18:41:57 [Worker: 1] Getting: https://github.com/hash
2021/06/18 18:41:57 [Worker: 2] Got: https://github.com/kubernet
2021/06/18 18:41:57 [Worker: 2] Getting: https://github.com/civo
2021/06/18 18:41:58 [Worker: 3] Got: https://github.com/moby (0.
2021/06/18 18:41:58 [Worker: 3] Finished work.
2021/06/18 18:41:58 [Worker: 0] Got: https://github.com/alexelli
2021/06/18 18:41:58 [Worker: 0] Finished work.
2021/06/18 18:41:58 [Worker: 2] Got: https://github.com/civo (0.
2021/06/18 18:41:58 [Worker: 2] Finished work.
2021/06/18 18:41:58 [Worker: 1] Got: https://github.com/hashicor
2021/06/18 18:41:58 [Worker: 1] Finished work.
https://github.com/docker has    <title>Docker · GitHub</title>
https://github.com/kubernetes has    <title>Kubernetes · GitHub</
https://github.com/moby has    <title>Moby · GitHub</title>
https://github.com/alexellis has    <title>alexellis (Alex Ellis)
https://github.com/civo has    <title>Civo · GitHub</title>
https://github.com/hashicorp has    <title>HashiCorp · GitHub</ti
https://github.com/inlets has    <title>inlets · GitHub</title>
https://github.com/openfaas has    <title>OpenFaaS · GitHub</titl
```

```
real    0m2.319s
user    0m0.545s
sys     0m0.101s
```

That's a difference of around 4 seconds. We gained that speed simply by implementing concurrency. The work queue then limits the amount of open requests.

To see this pattern in action, within the faas-cli, checkout the [build.go command](#).

Store data with a database

Almost all line of business applications will require a form of storage, so in this chapter I'll show you how to connect to a Postgresql database. For other databases, you'll need to find a different library, but the approach will be similar.

> Not all Go libraries are equal. Some use pure Go code, and others link to a rich ecosystem of existing C/C++ codebases. The ones which reference Go require CGO, which can introduce a number of other issues and portability problems.

One of the most helpful tools I've found for writing this eBook has been a todo list, so let's start with that example. You can then customise the schema, the fields and operations however you like to suit your own needs.

Create a directory for the exercise:

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/todo-database/
```

Create `schema.sql`:

```sql
CREATE TABLE todo (
  id              INT GENERATED ALWAYS AS IDENTITY,
  description     text NOT NULL,
  created_date    timestamp NOT NULL,
  completed_date  timestamp NULL
);
```

We're using a flat schema, with a single table. The `id` field is generated, and then we have a longer field for the todo text and a created/completed date field. Only the `completed_date` field can accept a null entry.

Once you have defined a schema, it's time to provision a database or to install Postgresql so that you can obtain credentials or a connection string and create the schema.

A managed Postgresql database created on DigitalOcean

I decided to create a database using Postgresql 11 on DigitalOcean. This is a managed service, and you'll find similar services are available from other cloud providers such as AWS and GCP.

You can use the `psql` tool to connect and create a database. Alternatively, you could input a `CREATE` statement at the start of your application. Some authors write code to automate maintenance like database creation and schema migration upon application startup.

My connection string from the dashboard was: `postgresql://doadmin:ildkg2my7g7lqqxh@todo-pg11-do-user-2197152-0.b.db.ondigitalocean.com:25060/defaultdb?sslmode=require`

When you deploy your application, be careful that you do not hard-code any confidential data. You may want to use a configuration file with strict permissions, or environment variables to read the password and address.

```go
package main

import (
    "database/sql"
    "flag"
    "fmt"
    "time"
```

```
        _ "github.com/lib/pq"
)


type Todo struct {
    ID              int
    Description     string
    CreatedDate     time.Time
    CompletedDate   *time.Time
}
```

Here, we define a `Todo` struct to be used to load entries from the database into memory. We also introduce the [database/sql](#) package which can be used with different libraries, but provides a single interface for operations.

In our case, we'll be using the [lib/pq](#) library, which is "pure Go" meaning that we do not need to enable CGO to make use of it.

Next, define all the flags that are required to connect to the database, and those which are required for the application. If you're writing server-side software, you may read these values from an environment variable, or if you're using OpenFaaS on Kubernetes, from a file mounted at `/var/openfaas/secrets/NAME`.

```
func main() {
    var (
        user, password, host, dbName, sslMode, todo, action string
    )
    var port int

    flag.StringVar(&user, "user", "", "Postgresql username")
    flag.StringVar(&password, "password", "", "Postgresql password")
    flag.StringVar(&host, "host", "", "Postgresql host")
    flag.IntVar(&port, "port", 0, "Postgresql port")
    flag.StringVar(&dbName, "dbname", "", "Postgresql host")
    flag.StringVar(&sslMode, "sslmode", "require", "Postgresql host")

    flag.StringVar(&todo, "todo", "", "Todo item")
    flag.StringVar(&action, "action", "create", "Action for todo iten

    flag.Parse()
```

Next, we'll define the flow which attempts to make a database connection, followed by the `create` and `list` operations. Perhaps you could write the `update`

and `done` actions yourself as an extension to the exercise?

```go
db, err := connect(user, password, host, dbName, sslMode, port)
if err != nil {
    panic(err)
}

switch action {
case "create":
    err := create(db, todo)
    if err != nil {
        panic(err)
    }
case "list":
    todos, err := list(db)
    if err != nil {
        panic(err)
    }
    for _, t := range todos {
        fmt.Printf("- %q\t%s\n", t.Description, t.CreatedDate.Str
    }
default:
    panic("Supported actions are: list, create")
}
}
```

Now on to the `connect` method. Here we construct a connection-string, which is often provided by a managed SQL service. If not, you can construct it from the individual parts. The Postgresql service on DigitalOcean also needs a "strict" `sslmode` flag.

```go
func connect(user, password, host, dbName, sslMode string, port int)
    connStr := "postgres://" + user + ":" + password + "@" + host + '

    db, err := sql.Open("postgres", connStr)
    if err != nil {
        return nil, err
    }

    err = db.Ping()
    if err != nil {
        return nil, err
    }
```

```
    return db, nil
}
```

Here's how you can perform an insert. Note that the `$1` is substituted for `todo` `string`. By using this syntax instead of `fmt.Sprintf` or string concatenation, we get some level of protection against SQL injection attacks where a user may insert additional SQL statements in a value.

```go
func create(db *sql.DB, todo string) error {
    res, err := db.Query(`INSERT INTO
        todo
        (id, description, created_date)
        VALUES
        (DEFAULT, $1, now());`,
        todo)
    if err != nil {
        return err
    }
    defer res.Close()
    return nil
}
```

The `list` function returns a slice of the `Todo` structs. You can use the same `$1` technique to specify keys or sort orders for the `SELECT` statement.

Finally the `Scan` function is used to populate the values from the row into the type using the proper typing. See how we didn't have to parse a date string into a `time.Time` for instance?

```go
func list(db *sql.DB) ([]Todo, error) {
    rows, err := db.Query(`SELECT
        id,
        description,
        created_date,
        completed_date
        FROM todo;`)

    if err != nil {
        return []Todo{},
            fmt.Errorf("unable to get from todo table: %w", err)
    }

    todos := []Todo{}
```

```go
		defer rows.Close()
		for rows.Next() {
			result := Todo{}
			err := rows.Scan(&result.ID,
				&result.Description,
				&result.CreatedDate,
				&result.CompletedDate)
			if err != nil {
				return []Todo{}, fmt.Errorf("row scan error: %w", err)
			}
			todos = append(todos, result)
		}
	return todos, nil
}
```

Then it's over to you, to try it out:

```
$ go build && ./todo \
  -user doadmin \
  -password ildkg2my7g7lqqxh \
  -host todo-pg11-do-user-2197152-0.b.db.ondigitalocean.com \
  -port 25060 \
  -dbname defaultdb \
  -todo "Publish this book" \
  -action create
```

Now list your todo items:

```
$ go build && ./todo \
  -user doadmin \
  -password ildkg2my7g7lqqxh \
  -host todo-pg11-do-user-2197152-0.b.db.ondigitalocean.com \
  -port 25060 \
  -dbname defaultdb \
  -action list

- "Publish this book"   2021-06-21 10:57:04.374785 +0000 +0000
```

Next, you could explore adding the missing functionality, and consider using a table and adding some colour. You may also want to filter out all items which have a completed date. Check out the tabwriter package for formatting your CLI output.

A CLI framework like Cobra, could improve usability by making the actions into verbs such as:

```
$ ./todo list
$ ./todo create "Publish the next version of this eBook"
$ ./todo close "Weekly shopping"
```

Find out more about the two libraries we used below:

- [lib/pq](lib/pq)
- database/sql

For a more complete example of data-access including a schema with a foreign key relationship and custom database functions, see my example on the OpenFaaS blog: How to build a Serverless Single Page App

Write your own config files in YAML

The trend with [12 Factor Apps](#) is to make all configuration available through environment variables instead of through files. This makes them easy to deploy with containers because files tend to require additional permissions or filesystems to be created and mounted.

However environment variables can have drawbacks like length, and complexity when there are many of them. Nesting is also difficult since they are all inputted at the same level. You may find that a configuration file is the answer.

A practical example of YAML in action

In the OpenFaaS CLI we added a configuration file to store the credentials for different installations that you log into. The CLI also has a configuration file for functions that you deploy called stack.yml.

```yaml
version: 1.0
provider:
  name: openfaas
  gateway: http://127.0.0.1:8080

functions:
  gumroad-upgrade:
    lang: node12
    handler: ./gumroad-upgrade
    image: alexellis2/gumroad-upgrade:0.2.1
    environment:
      subject: "Your bonus upgrade to my video workshop"
      sender: sales@openfaas.com
      region: "eu-west-1"
    secrets:
    - ses-access-key-id
```

```
    - ses-access-token
    - seller-id
    - secret-url
```

Example of an OpenFaaS YAML file to configure building and deploying functions

The library that I've used most is gopkg.in/yaml.v2. It suited the use-case that it's my go-to library for parsing or saving YAML.

Read and write your own YAML schema

Create a new folder and initialise it for the sample:

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/yaml-config/
```

To load a configuration file, you define a number of structs and annotate them, just like we did with the JSON examples.

```go
type Spec struct {
    // Name name of the function
    Name string `yaml:"name"`

    // Image docker image name of the function
    Image string `yaml:"image"`

    Environment map[string]string `yaml:"environment,omitempty"`

    // Limits for the function
    Limits *FunctionResources `yaml:"limits,omitempty"`
}


// FunctionResources Memory and CPU
type FunctionResources struct {
    Memory string `yaml:"memory"`
    CPU    string `yaml:"cpu"`
}
```

As you can see from above, where we would have put `json:"field"`, we just write `yaml` instead and then go on to use a very similar interface to unmarshal the text into an object.

Create main.go:

```go
package main

import (
    "fmt"
    "io/ioutil"

    yaml "gopkg.in/yaml.v2"
)

// Add above structs here

func main() {
    bytesOut, err := ioutil.ReadFile("config.yaml")
    if err != nil {
        panic(err)
    }

    spec := Spec{}
    if err := yaml.Unmarshal(bytesOut, &spec); err != nil {
        panic(err)
    }

    fmt.Printf("Function name: %s\tImage: %s\tEnvs: %d\n", spec.
}
```

Test it out:

```
go run main.go
Function    name:     gumroad-bot                              Image:
ghcr.io/alexellis/gumroad-bot     Envs: 1
```

You can write a configuration file or YAML file to disk using the Marshal function. Adapt the example:

```go
func main() {
    spec := Spec{
        Image: "docker.io/functions/figlet:latest",
        Name:  "figlet",
    }

    bytesOut, err := yaml.Marshal(spec)
    if err != nil {
        panic(err)
    }

    err = ioutil.WriteFile("figlet.yaml", bytesOut, os.ModePerm)
    if err != nil {
        panic(err)
    }
    fmt.Printf("Wrote: figlet.yaml.. OK.\n")
}
```

Test it out:

```
$ go run main.go

Wrote: figlet.yaml.. OK.

$ cat figlet.yaml

name: figlet
image: docker.io/functions/figlet:latest
```

In a previous role, I found that I could create huge amounts of CI jobs very quickly by reading an input file and generating YAML using this library. It saved me a lot of time overall.

Merging objects

If you create a large schema with many different fields, it may be desirable to maintain a single example file and then allow users to provide overrides for any values that need customisation.

We've used the [mergo](#) library [in the OpenFaaS project](#) and had good experiences, however you may find other libraries that fulfil a similar role.

This approach works with any structs that you have in memory, and isn't specific to YAML.

Create a new folder for the example:

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/merge-objects
```

This is how the API works:

```
merged := Spec{}

overrides := Spec{}

err := mergo.Merge(merged, override, mergo.WithOverride)
if err != nil {
    return merged, err
}
```

Here's an example of what it would look like:

```go
package main

import (
    "fmt"

    "github.com/imdario/mergo"
    yaml "gopkg.in/yaml.v2"
)

// Paste the struct definitions in here.

func main() {
    base := Spec{
        Image: "docker.io/functions/figlet:latest",
        Name:  "figlet",
    }
```

```go
    production := Spec{
        Environment: map[string]string{"stage": "production"},
        Limits:      &FunctionResources{Memory: "1Gi", CPU: "100
    }

    overrides := []Spec{
        base,
        production,
    }

    merged := Spec{}
    for _, override := range overrides {
        err := mergo.Merge(&merged, override, mergo.WithOverride
        if err != nil {
            panic(err)
        }
    }

    bytesOut, err := yaml.Marshal(merged)
    if err != nil {
        panic(err)
    }

    fmt.Printf("Merged content:\n\n%s\n", string(bytesOut))
}
```

Output:

```
$ go run main.go

Merged content:

name: figlet
image: docker.io/functions/figlet:latest
environment:
  stage: production
limits:
  memory: 1Gi
  cpu: 100Mi
```

See how the production configuration for the "stage" field and limits were overlaid on top of the base configuration?

Where could you use object merging in your code?

Inject version information into your binaries

In this chapter, I'll show you how to inject variables into your [Golang](#) executable at build-time.

The two ways we have used this in the OpenFaaS community are:

- To inject version information such as a Git commit ID (`77b4dce`) and Git tag (`0.1.0`)
- To add platform information such as whether the binary is for Raspberry Pi or a desktop PC
- To add a public key for validating licenses for open-core features

This information becomes part of the compiled application, but is determined once at build-time, instead of at run-time.

Here's an example where the Docker project's CLI contains a commit ID of `77b4dce` along with the Go version it was built with.

```
$ docker version
Client: Docker Engine - Community
 Version:           19.03.13
 API version:       1.40
 Go version:        go1.13.15
 Git commit:        4484c46d9d
 Built:             Wed Sep 16 17:02:52 2020
 OS/Arch:           linux/amd64
 Experimental:      false
```

Why add versioning information?

I can't state how useful and valuable it is to have a version number in your application when it comes to bug reports from users. It can also help you spot outdated versions, and show users verify what version they are using by running the program's `version` command.

Capture useful variable(s)

Before you get started, think about what makes sense to be injected at build-time? It could be anything from the hostname of the CI machine to the result of a web-service call or more commonly the last commit ID from your Git log.

Let's use the last Git commit id.

Create a test Git project with a README

```
$ mkdir /tmp/git-tester && \
 cd /tmp/git-tester && \
 git init && \
 echo "Let's work with Git" > README && \
 git add . && \
 git commit -m "Initial" && \
 echo "Let's keep working" >> README && \
 git add . && \
 git commit -m "First update"
```

You'll now see two commits in your Git log:

```
$ git log
commit 67b05a31758848e1e5237ad5ae1dc11c22d4e71e
Author: Alex Ellis <alexellis2@gmail.com>
Date:   Tue Aug 8 08:37:20 2017 +0100

    First update

commit 9b906b6d02d803111250a974ed8042c4760cde7f
Author: Alex Ellis <alexellis2@gmail.com>
Date:   Tue Aug 8 08:37:20 2017 +0100

    Initial
```

Here's how you find the ID of your last commit:

```
$ git rev-list -1 HEAD
67b05a31758848e1e5237ad5ae1dc11c22d4e71e
```

Next we can capture that into an environmental variable:

```
$ export GIT_COMMIT=$(git rev-list -1 HEAD) && \
  echo $GIT_COMMIT

67b05a31758848e1e5237ad5ae1dc11c22d4e71e
```

Prepare your code

Let's build a sample that prints a version that we injected at build-time.

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/inject-commit/
```

Edit `main.go`:

```go
package main

import (
        "fmt"
)

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

In order to pass a build-time variable we need to create a variable within our main package. We'll call it `var GitCommit string`.

```go
package main

import (
    "fmt"
)

// GitCommit is set at build-time
var GitCommit string

func main() {
```

```
    fmt.Printf("Hello world, version: %s\n", GitCommit)
}
```

Test it out with `go build`:

```
go build
./inject-commit


Hello world, version:
```

The version is empty, but we're now ready to start injecting a variable.

Make sure you have at least one commit in place:

```
git init

echo "/inject-commit" > ".gitignore"

echo "# Sample Commits versions in Go" >> README.md
git add .
git commit -m "Initial"
```

Override the `go build` parameters using `ldflags`

Now we need an additional override to our `go build` command to pass information to the linker via the `-ldflags` flag.

```
export GIT_COMMIT=$(git rev-list -1 HEAD)
go build -ldflags "-X main.GitCommit=$GIT_COMMIT"
```

Now we see our application has a hard-coded version which we injected at build-time.

```
$ ./inject-commit
Hello world, version: a38eef50779c6e659b405a14e80b51148e6d4cb1
```

You can have your version printed out every time you start the application, or add code so that it is only printed when the `-version` flag or `version` command is used.

```
package main
```

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

// GitCommit is set at build-time

var GitCommit string

func main() {
    var version bool
    flag.BoolVar(&version, "version", false, "Print the version"
    flag.Parse()

    if version {
        fmt.Printf("Version: %s\n", GitCommit)
    }
    fmt.Printf("Hello world\n")
}
```

```
$ go build -ldflags "-X main.GitCommit=$GIT_COMMIT" && ./inject-
Version: a38eef50779c6e659b405a14e80b51148e6d4cb1

Hello world


$ go build -ldflags "-X main.GitCommit=$GIT_COMMIT" && ./inject-
Hello world
```

Do it with Docker

Once you have worked out your build-time variables it's likely you will want to update your Dockerfile.

Write a Dockerfile:

```
FROM golang:1.15.0

WORKDIR /go/src/github.com/alexellis/inject-commit
COPY .git      .
COPY app.go    .

RUN GIT_COMMIT=$(git rev-list -1 HEAD) && \
  go build -ldflags "-X main.GitCommit=$GIT_COMMIT"

CMD ["./git-tester", "-version"]
```

Dockerfile

Run a build and then test it out:

```
$ docker build -t inject-commit .
$ docker run -ti inject-commit
Hello world, version: 67b05a31758848e1e5237ad5ae1dc11c22d4e71e
```

When it comes to shipping Docker images with Go, you need to use a multi-stage build, to ship just your binary instead of the `golang:1.15.0` image which is an SDK and comparatively large in comparison to your program. You also need to switch to a non-root user, so that the container is able to run without unnecessary privileges.

Now this is just a minimal example to show you how to get started with Docker and Go build-time variables. You can take it further and optimize the size of the Docker image by using Multi-stage builds.

Examples in production

Check out alexellis/k3sup to find a static binary that is shipped with version information. For a binary that is also shipped with a Docker image, check out openfaas/of-watchdog or openfaas/faas-cli.

```
$ k3sup version

  _         _____
 | |   _|___  / ___  _   _ _ __
 | |/ / |_ \/ __| | | | | '_ \
 |   < __) \__ \ |_| | | |_) |
 |_|\_\____/|___/\__,_| .__/
```

```
                             |_|
Version: 0.9.13
Git Commit: 95fc8b074a6e0ea48ea03a695491e955e32452ea
```

You can see that I went a little further here and added an ASCII logo, and if
you run the example yourself, you'll see that the logo is coloured red also.
View the code to see how.

Embed data in your application

It can be useful to ship additional resources other than code with your applications. A common method for doing this is embedding files or data.

In the "Store data with a database" chapter we created a todo app, which needed a schema to be created manually to define a table. But what if our application could create the table at start-up, if it didn't exist already?

Fortunately in Go 1.16 the embed package has made this simple, and before that many third-party libraries existed to do very similar things.

The Go docs show three ways we could achieve this:

```go
import _ "embed"

//go:embed schema.sql
var tableCreate string
print(tableCreate)
```

At build time, the contents of `schema.sql` will be built into the application, and made available through the `tableCreate` string. This is very similar to the technique we saw for injecting a version into an application.

If you had an image, then you would want to encode the bytes as a binary `[]byte` slice.

```go
import _ "embed"

//go:embed logo.png
var logo []byte
fmt.Printf("Total bytes: %d\n", len(logo))
```

Once in memory, the logo can then be served to a client via a HTTP connection, modified, or written to disk for instance.

Using the `embed.FS` interface means that unlike in the previous example, the data is not kept in memory in a variable, and can be loaded on demand. This approach would be useful for large files.

```go
import "embed"


//go:embed ubuntu-20-cloudinit.iso
var f embed.FS
data, _ := f.ReadFile("ubuntu-20-cloudinit.iso")
fmt.Printf("Total bytes: %d\n", len(data))
```

Go is a perfectly good systems programming language, and you'll find many packages for managing filesystems and networks. You could potentially ship something like an ISO image inside your application that can be written out to a disk at any time.

The fourth option we'll look at is what I would use if I wanted to ship something like a React.js front-end along with a HTTP server or API written in Go.

```go
import "embed"


// frontend holds our static web server content.
//go:embed image/* template/*
//go:embed public/index.html
//go:embed css/*
var frontend embed.FS
```

Now you can simply link the `embed.FS` to a custom HTTP middleware or handler and serve the files to users from a specific route or path. You'll learn more about HTTP servers later in the book, however this is the example given in the Go documentation:

```go
http.Handle("/public/",
    http.StripPrefix("/public/",
        http.FileServer(http.FS(frontend))))
```

Your API could be bound to the path `/api/v1/` and then any static content you have embedded would be served from `/public/`.

Create dynamic content with templates

Templates are used to replace tokens or expressions in a file or block of text (template) with inputs provided at runtime.

They look similar to expressions found in front-end templating technology like [Embedded Ruby (ERb)](#), for this reason they may seem inituitive to you already.

A template can be as simple as a single line of text, or a multi-line block. The template could be read from disk, over the Internet, or kept within a const in memory.

Here's an example based upon the [Golang docs](#):

```
Dear {{.Name}},

{{if .Attended}}
It was a pleasure to see you at the wedding.
{{- else}}
It is a shame you couldn't make it to the wedding.
{{- end}}

Best wishes,
Alex
```

First, we need to load the template into memory, then construct a typed struct or an inline struct to pass in the values to be placed within the file, then we execute it to get the result. `.Name` will be replaced directly, but `.Attended` is going to cause one of the two following sentences to be added into the letter.

```go
package main

import (
    "bytes"
    "fmt"
    "text/template"
)

const letter = `Dear {{.Name}},
```

```go
{{if .Attended}}
It was a pleasure to see you at the wedding.
{{- else}}
It is a shame you couldn't make it to the wedding.
{{- end}}
Best wishes,
Alex`

func main() {

    people := []struct {
        Name     string
        Attended bool
    }{
        {
            Name:     "Alex",
            Attended: true,
        },
        {
            Name:     "Mick",
            Attended: false,
        },
    }

    t := template.Must(template.New("letter").Parse(letter))

    for _, person := range people {
        buffer := bytes.Buffer{}
        err := t.Execute(&buffer, person)
        if err != nil {
            panic(err)
        }
        fmt.Printf("Letter for %s:\n\n%s\n\n", person.Name, buffer.St
    }
}
```

Running the example we see the following output:

```
$ go run main.go
Letter for Alex:

Dear Alex,

It was a pleasure to see you at the wedding.
```

```
Best wishes,
Alex


Letter for Mick:


Dear Mick,


It is a shame you couldn't make it to the wedding.
Best wishes,
Alex
```
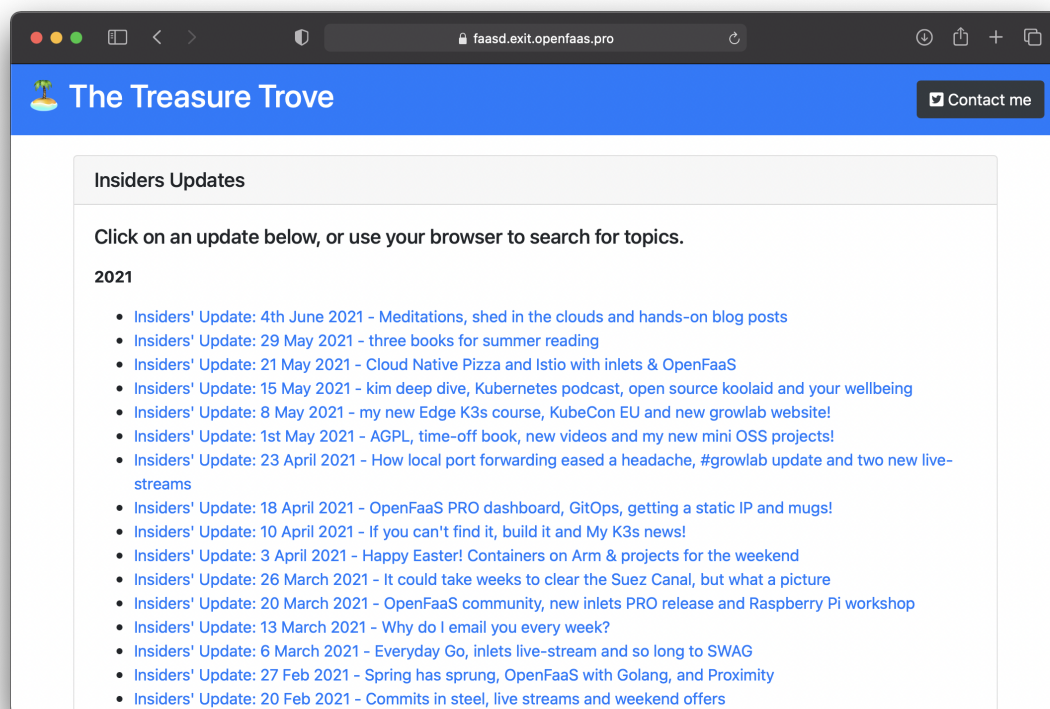
Whenever I want to add a HTML front-end to an application, I'll turn to Go templates and embed the HTML template files along with the code.

Advanced templates: ranges and custom functions

I use templates with [my Sponsors' Portal](#) in order to list and print out all the previous editions of my newsletter.

A list of previous newsletter emails

Here is a snippet of the code that prints out the list:

```html
<div class="card">
  <h5 class="card-header">Insiders Updates</h5>
    <div class="card-body">
      <h5 class="card-title">Click on an update below, or use
      <ul>
      <!-- Be a good person, and do not scrape this HTML -->
      {{ range $in, $el:= .Updates }}
        {{ if ($.Updates.YearBreak $in) }}
        </ul>
        <p><b>{{$el.Date.Year}}</b></p>
        <ul>
        {{ end }}
        <li id="{{$el.ID}}"><a href="{{$.Path "/update"}}/?id
      {{ end }}
      </ul>
    </div>
  </div>
</div>
```

In the example I used the `range` keyword to iterate around the `Updates` slice, which contains a number of Update structs.

Things then get slightly more involved because I wanted to break the list according to the year of the post, so that they would be broken up into blocks or groups.

On the line `{{ if ($.Updates.YearBreak $in) }}` you can see that I call a function to determine when to break the line. The simplest way to add custom functions to templates is to add functions to the type being accessed in the template.

```go
// Update represents an InsidersUpdate stored as a Gist
type Update struct {
    ID          string    `json:"id,omitempty"`
    Description string    `json:"description,omitempty"`
    Date        time.Time `json:"date,omitempty"`
    MarkdownURL string    `json:"markdownUrl,omitempty"`
    HTMLURL     string    `json:"htmlUrl,omitempty"`
}
```

```go
type Updates []Update

// YearBreak returns whether the previous item was in
// another year
func (u Updates) YearBreak(i int) bool {
    if i == 0 {
        return true
    }

    return u[i-1].Date.Year() != u[i].Date.Year()
}
```

By checking the index of the current item, and the previous one, we can determine whether we are at a breakpoint in the list of years and emails.

Real-world templates - arkade

A final example I'll share is the arkade project, where templates were useful for determining which version of a CLI to download.

If there was a convention that all maintainers and vendors used for CLI naming then we would only need one template for all downloads, however that's anything but the case.

> You may be wondering why these templates are compiled into the Go binary? The `Tool` struct could be Marshaled into a JSON file for dynamic updates, but having everything in one place, and being able to unit test quickly seems to be working for us.

```go
tools = append(tools,
Tool{
Owner:        "inlets",
Repo:         "inlets-pro",
Name:         "inlets-pro",
Description: "Cloud Native Tunnel for HTTP and TCP traffic.",
BinaryTemplate: `{{ if HasPrefix .OS "ming" -}}
{{.Name}}.exe
{{- else if eq .OS "darwin" -}}
{{.Name}}-darwin
{{- else if eq .Arch "armv6l" -}}
{{.Name}}-armhf
{{- else if eq .Arch "armv7l" -}}
{{.Name}}-armhf
```

```
{{- else if eq .Arch "aarch64" -}}
{{.Name}}-arm64
{{- else -}}
{{.Name}}
{{- end -}}`,
```

When evaluated, this template takes the information from running `uname -a` and then determines the correct filename for your Operating System and hardware type.

One of the downsides of using Go templates is that they cannot be validated statically. They are only validated when executed, and so for arkade we introduced a policy of writing unit tests to cover each code path.

As we learned in a previous chapter `go run -cover` can help us validate whether all code paths are being executed.

```go
func Test_DownloandInletsProCli(t *testing.T) {
    tools := MakeTools()
    name := "inlets-pro"

    tool := getTool(name, tools)

    tests := []test{
        {
            os:      "ming",
            arch:    arch64bit,
            version: "0.8.3",
            url:     `https://github.com/inlets/inlets-pro/releases/c
        },
        {
            os:      "linux",
            arch:    arch64bit,
            version: "0.8.3",
            url:     `https://github.com/inlets/inlets-pro/releases/c
        },
        {
            os:      "linux",
            arch:    archARM64,
            version: "0.8.3",
            url:     `https://github.com/inlets/inlets-pro/releases/c
        },
        {
            os:      "darwin",
```
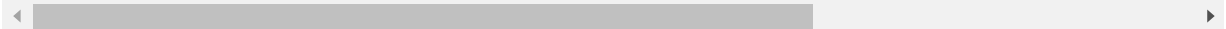
```go
            arch:       arch64bit,
            version: "0.8.3",
            url:        `https://github.com/inlets/inlets-pro/releases/d
        },
    }

    for _, tc := range tests {
        got, err := tool.GetURL(tc.os, tc.arch, tc.version)
        if err != nil {
            t.Fatal(err)
        }
        if got != tc.url {
            t.Errorf("want: %s, got: %s", tc.url, got)
        }
    }

}
```

The above code means we can run `arkade get inlets-pro` and then the latest version of the correct binary will be downloaded to our system.

Checkout the code for arkade, and run the tests for yourself: * alexellis/arkade

Write a HTTP server

One of the most useful projects you can write with Go is a HTTP server. It's also one of easiest to start with using files only from the standard library.

```go
package main

import (
    "fmt"
    "log"
    "net/http"
    "time"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Req
        log.Printf("Request received")
        w.Write([]byte("Hello world"))
    })

    port := 8080
    s := &http.Server{
        Addr:           fmt.Sprintf(":%d", port),
        ReadTimeout:    10 * time.Second,
        WriteTimeout:   10 * time.Second,
        MaxHeaderBytes: 1 << 20,
    }
    log.Printf("Listening on port: %d\n", port)
    log.Fatal(s.ListenAndServe())
}
```

Test it out:

```
# In terminal 1
go run main.go

# In terminal 2
```

```
$ curl http://localhost:8080 ; echo

Hello world
```

The http.Request has the context and content of the HTTP request, you can [explore the source code here](#) or with intelli-sense in your IDE.

Reading the request

A HTTP request has several ways to send data which you can use in your function to decide what to do: * Body * HTTP headers * Query string * Path * Host header

Create a new project and test it out:

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/yaml-config/
cd $GOPATH/src/github.com/$GH_USERNAME/yaml-config/
go mod init http-server

touch main.go
```

This is an example of how to read each of them:

```go
package main

import (
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "time"
)

func main() {
    http.HandleFunc("/info", func(w http.ResponseWriter, r *http
        log.Printf("=====================================
        log.Printf("Query-string: %s", r.URL.RawQuery)
        log.Printf("Path: %s", r.URL.Path)
        log.Printf("Method: %s", r.Method)
```

```go
        log.Printf("Path: %s", r.Host)
        for k, v := range r.Header {
            log.Printf("Header %s=%s", k, v)
        }

        if r.Body != nil {
            body, _ := ioutil.ReadAll(r.Body)
            log.Printf("Body: %s", string(body))
        }
        log.Printf("==========================================

        w.WriteHeader(http.StatusAccepted)
    })

    port := 8080
    s := &http.Server{
        Addr:           fmt.Sprintf(":%d", port),
        ReadTimeout:    10 * time.Second,
        WriteTimeout:   10 * time.Second,
        MaxHeaderBytes: 1 << 20,
    }
    log.Printf("Listening on port: %d\n", port)
    log.Fatal(s.ListenAndServe())
}
```

Let's try it out with a HTTP get and a HTTP post using `curl`:

```
# Terminal 1
$ go run main.go
2021/06/08 11:11:30 Listening on port: 8080


# Terminal 2
$ curl -i http://127.0.0.1:8080/info


2021/06/08 11:11:37 ==========================================
2021/06/08 11:11:37 Query-string:
2021/06/08 11:11:37 Path: /info
2021/06/08 11:11:37 Method: GET
2021/06/08 11:11:37 Path: localhost:8080
2021/06/08 11:11:37 Header User-Agent=[curl/7.68.0]
2021/06/08 11:11:37 Header Accept=[*/*]
2021/06/08 11:11:37 Body:
```

```
2021/06/08 11:11:37 Body:
2021/06/08 11:11:37 =========================================
```

Note the User-Agent sent by curl, and the lack of a HTTP body. We also get the path, which was used by the internal route handling to run the function we defined.

```
# Terminal 2
$ curl -i -H "Content-type: application/json" \
  -d '{"mode": "green"}' http://localhost:8080/info?insert=true

2021/06/08 11:13:28 =========================================
2021/06/08 11:13:28 Query-string: insert=true
2021/06/08 11:13:28 Path: /info
2021/06/08 11:13:28 Method: POST
2021/06/08 11:13:28 Path: localhost:8080
2021/06/08 11:13:28 Header Content-Length=[17]
2021/06/08 11:13:28 Header User-Agent=[curl/7.68.0]
2021/06/08 11:13:28 Header Accept=[*/*]
2021/06/08 11:13:28 Header Content-Type=[application/json]
2021/06/08 11:13:28 Body: {"mode": "green"}
2021/06/08 11:13:28 =========================================
```

Here we have passed JSON, so can see a content-type and body defined along with the POST method.

Advanced route handling

You can add logic to a HTTP handler to only respond to certain HTTP methods, or to take different actions according to the path, however there are also a number of libraries available that can reduce code duplication.

[Gorilla's mux](#) is one of the most popular and is used in OpenFaaS and the inlets project.

In this example, the `function` path can take a name as an argument which will be parsed and made available without any user action. The Regular

Expression allows alpha-numerics plus dashes and requires at least one character.

```
    r.HandleFunc("/function/{name:[-a-zA-Z_0-9.]+}", functionPro
```

Within the handler, the `mux.Vars()` helper function is used to query the name:

```go
    return func(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        name := vars["name"]
        log.Printf("Invoking function: %s", name)
    }
```

The library also supports a fluid-style for defining behaviour, where methods can be chained to define logic.

In this example, varying HTTP verbs are bound to different handlers to manage a REST endpoint to manage functions.

```
    r.HandleFunc("/system/functions", listFunctions).Methods(htt
    r.HandleFunc("/system/functions", deployFunction).Methods(ht
    r.HandleFunc("/system/functions", deleteFunction).Methods(ht
    r.HandleFunc("/system/functions", updateFunction).Methods(ht
```

Let's try a full example:

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/mux/
cd $GOPATH/src/github.com/$GH_USERNAME/mux/
go mod init mux

touch main.go
```

Then add this code to main.go:

```go
package main

import (
    "fmt"
```

```go
        "log"
        "net/http"
        "time"

        "github.com/gorilla/mux"
)

func main() {

    router := mux.NewRouter()
    router.HandleFunc("/customer/{id:[-a-zA-Z_0-9.]+}", func(w h
        v := mux.Vars(r)
        id := v["id"]
        if id == "alex" {
            w.Write([]byte(fmt.Sprintf("Found customer: %s", id)
        } else {
            http.Error(w, fmt.Sprintf("Customer %s not found", i
        }
    })

    tcpPort := 8080
    s := &http.Server{
        Addr:           fmt.Sprintf(":%d", tcpPort),
        ReadTimeout:    time.Second * 10,
        WriteTimeout:   time.Second * 10,
        MaxHeaderBytes: http.DefaultMaxHeaderBytes, // 1MB - can
        Handler:        router,
    }
    log.Printf("Listening on: %d", tcpPort)


    log.Fatal(s.ListenAndServe())
}
```

Then try it out:

```
# Terminal 1
$ go run main.go


# Terminal 2
```

```
$ curl -i http://127.0.0.1:8080/customer/alex

HTTP/1.1 200 OK
Date: Tue, 08 Jun 2021 10:32:16 GMT
Content-Length: 20
Content-Type: text/plain; charset=utf-8

Found customer: alex

$ curl -i http://127.0.0.1:8080/customer/john

HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Tue, 08 Jun 2021 10:32:55 GMT
Content-Length: 24

Customer john not found
```

Writing HTTP middleware

A HTTP middleware is a function that runs before or after the main HTTP handler for an endpoint has executed. Examples of why you may need to do this include: logging & tracing, authentication & authorization and applying other rules or parsing.

Let's try a full example which rejects any HTTP requests which do not have a Bearer header (often used for authentication):

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/http-bearer
cd $GOPATH/src/github.com/$GH_USERNAME/http-bearer
go mod init http-bearer

touch main.go
```

Here's the code:

```
package main
```

```go
import (
    "fmt"
    "log"
    "net/http"
    "os"
    "time"
)

func makeAuth(token string, next http.HandlerFunc) http.HandlerF
    return func(w http.ResponseWriter, r *http.Request) {
        if b := r.Header.Get("Authorization"); len(b) == 0 || b
            http.Error(w, "Not authorized", http.StatusUnauthori
        } else {
            next(w, r)
        }
    }
}

func main() {
    token := os.Getenv("TOKEN")
    next := func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Authorized"))
    }

    http.HandleFunc("/customer/", makeAuth(token, next))

    tcpPort := 8080
    s := &http.Server{
        Addr:           fmt.Sprintf(":%d", tcpPort),
        ReadTimeout:    time.Second * 10,
        WriteTimeout:   time.Second * 10,
        MaxHeaderBytes: http.DefaultMaxHeaderBytes,
    }
    log.Printf("Listening on: %d", tcpPort)
    log.Fatal(s.ListenAndServe())
}
```

Note that the `makeAuth` function adds an if guard around the HTTP authorization header, and if not found or not having the matching token, then the "next" HTTP call won't be made. In this way, you can chain as

many HTTP middleware as you like, for authorization then logging, and whatever else you need.

Try it out:

```
# Terminal 1
go run main.go

# Terminal 2

curl http://127.0.0.1:8080/customer/alex -H "Authorization: Bear
Authorized


curl http://127.0.0.1:8080/customer/alex
Not authorized
```

Add Prometheus metrics to your microservices

The [Prometheus](#) project was originally developed at Soundcloud, before being donated to the [Cloud Native Computing Foundation (CNCF)](#), where it is now considered to be a "graduated" project. For this reason, it's common to find Prometheus used for collecting HTTP metrics in many different microservices within open source and enterprise.

Prometheus is deployed as a HTTP server alongside your applications. It then connects to each known endpoint to gather metrics from a HTTP endpoint. It could be deployed on the same HTTP port as your application or on a separate port usually under `/metrics`. The interval is known as a `scrape_interval` and each application is a `scrape_target`.

To add metrics to your application, you can use the [prom-http](#) package, which exposes a `/metrics` endpoint.

The simplest thing you can do is to add promhttp's Handler which will then expose information on memory usage. Later, we'll see how to define custom metrics.

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/metrics/
cd $GOPATH/src/github.com/$GH_USERNAME/metrics/
go mod init metrics

touch main.go
```

Create main.go:

```go
package main

import (
    "fmt"
    "net/http"
    "time"
```

```go
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

func main() {

    port := 8080
    readTimeout := time.Millisecond * 500
    writeTimeout := time.Millisecond * 500

    metricsMux := http.NewServeMux()
    metricsMux.Handle("/metrics", promhttp.Handler())
    metricsMux.HandleFunc("/", func(w http.ResponseWriter, r *http.Re
        fmt.Fprintf(w, "No tracking for this request")
    })

    s := &http.Server{
        Addr:           fmt.Sprintf(":%d", port),
        ReadTimeout:    readTimeout,
        WriteTimeout:   writeTimeout,
        MaxHeaderBytes: 1 << 20, // Max header of 1MB
        Handler:        metricsMux,
    }

    s.ListenAndServe()
}
```

Now:

```
# Terminal 1
go run main.go

# Terminal 2
curl http://127.0.0.1:8080/metrics
```

You'll see HTTP metrics about the `/metrics` endpoint itself:

```
# HELP promhttp_metric_handler_requests_in_flight Current number of
scrapes being served.
# TYPE promhttp_metric_handler_requests_in_flight gauge
promhttp_metric_handler_requests_in_flight 1
# HELP promhttp_metric_handler_requests_total Total number of
scrapes by HTTP status code.
# TYPE promhttp_metric_handler_requests_total counter
promhttp_metric_handler_requests_total{code="200"} 1
```

```
promhttp_metric_handler_requests_total{code="500"} 0
promhttp_metric_handler_requests_total{code="503"} 0
```

Then you'll also see memory information, the Go version, the number of Goroutines active, and many other details.

```
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 8

# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.15.6"} 1

# HELP go_memstats_alloc_bytes Number of bytes allocated and still
in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 1.152704e+06
#  HELP  go_memstats_alloc_bytes_total  Total  number  of  bytes
allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 1.152704e+06
```

Once you have Prometheus deployed, you can configure it to scrape the Go application. Prometheus can be run as a normal executable, if you download it from its [GitHub releases page](#).

You'll see the `prometheus` and `prometheus.yml` files in the archive. Extract the files and edit `prometheus.yaml`.

```yaml
# my global config
global:
  scrape_interval:     15s # Set the scrape interval to every 15 seco
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The de:
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      # - alertmanager:9093

# Load rules once and periodically evaluate them according to the gl
rule_files:
  # - "first_rules.yml"
```

```
  # - "second_rules.yml"


# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeser:
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.
    static_configs:
    - targets: ['localhost:9090']
```

Under `scrape_configs` add a job name and scrape target for your microservice.

```
  # The job name is added as a label `job=<job_name>` to any timeser:
  - job_name: 'service'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.
    static_configs:
    - targets: ['localhost:8080']
```

If deployed in a cluster or on a network, the target would change from `localhost` to the IP address or DNS name that the service can be reached on, for instance: `gateway:8080`.

```
# In Terminal 1
$ ./prometheus --config.file=./prometheus.yml


# In Terminal 2
$ go run main.go
```

You can navigate to the Prometheus UI at:

http://127.0.0.1:9090

Under "Status" and "Targets" you'll be able to view the two targets and when they were last scraped. If there are any issues with accessing the endpoints, you'll be able to see them here too.

The scrape configuration

The scrape configuration

The metrics explorer (globe) button shows all metrics available for browsing.

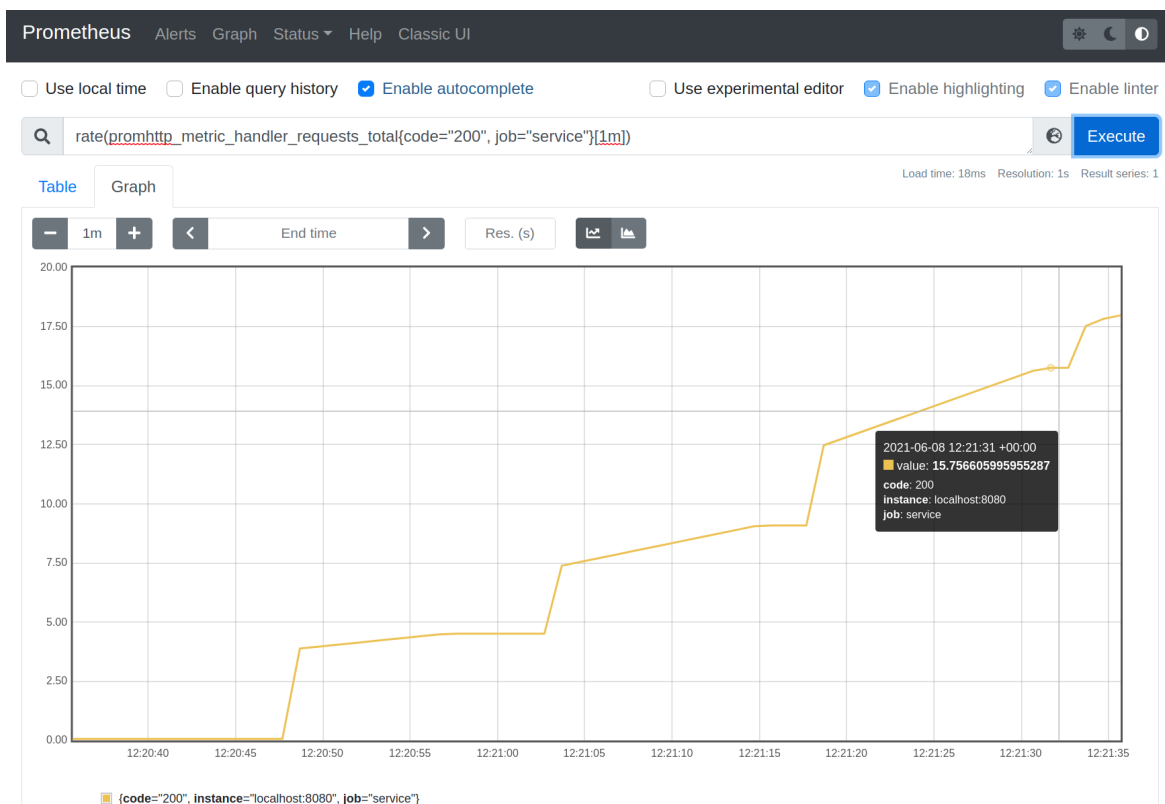

The metrics explorer

The table view can also be useful to view numbers
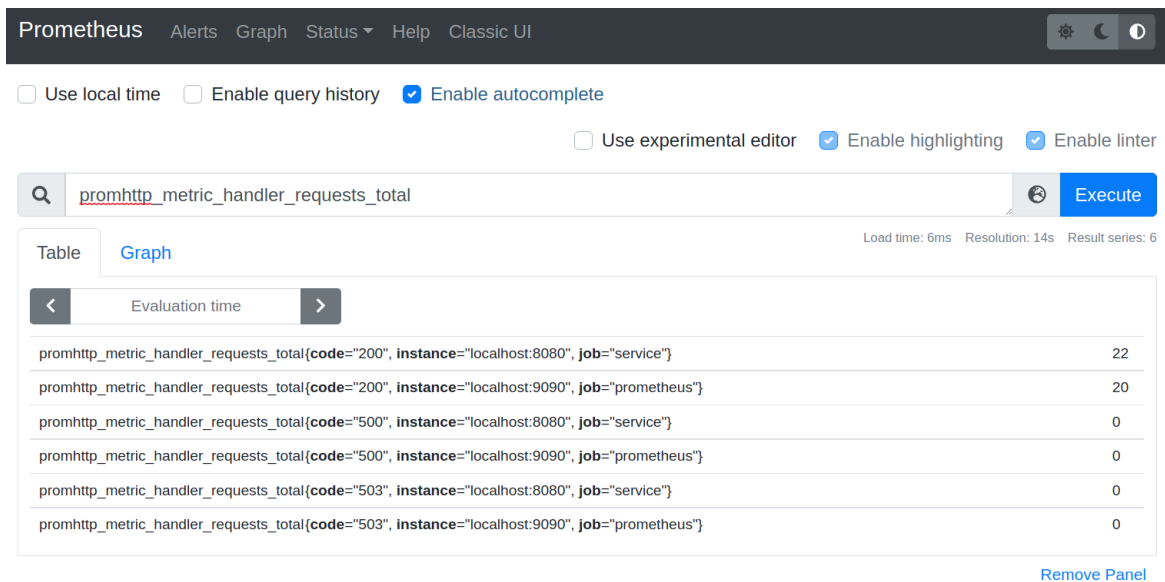
Prometheus table view

One of the most useful queries you can write is to check the rate of certain HTTP requests against your service. This query looks for 200 / OK responses, but you may want to pick out specific error codes to detect issues.

The rate graph

A graph showing the rate from the query

```
rate(
    promhttp_metric_handler_requests_total{
        code="200",
        job="service"
    }[1m]
)
```

The rate of invocations on the `promhttp_metric_handler_requests_total` metric are being tracked for HTTP 200 status codes for the service job over a 1-minute window.

Remember that this is just the status code for the Prometheus endpoint itself, so it does not yet instrument any API calls that our service may have. There is value in this though, since it can also track leaking Goroutines which haven't been cleared up with a similar query:

```
rate(
    go_goroutines{
        job="service"
    }[1m]
)
```

Note the "job" is looking at the scrape endpoint for our HTTP server. If you want, you can also review the amount of Goroutines Prometheus itself is running, since it instruments itself under the "prometheus" job.

Tracking HTTP metrics for our microservice

We will now build a microservice that can generate a SHA256 hash of any input body, and will return a 200 status code for valid inputs. When a body isn't present and the HTTP method isn't POST, then the service will return a 400 error.

When complete, we'll be able to measure Rate Error Duration (RED) metrics for any HTTP middleware or handler you wish to instrument.

```
export GH_USERNAME="alexellis"
mkdir -p $GOPATH/src/github.com/$GH_USERNAME/custom-metrics/
cd $GOPATH/src/github.com/$GH_USERNAME/custom-metrics/
go mod init metrics
```

```
touch main.go
touch metrics.go
```

Create metrics.go:

```go
package main

import (
    "net/http"

    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promauto"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)


type HttpMetrics struct {
    RequestsTotal            *prometheus.CounterVec
    RequestDurationHistogram *prometheus.HistogramVec
}


func NewHttpMetrics() HttpMetrics {
    return HttpMetrics{
        RequestsTotal: promauto.NewCounterVec(prometheus.CounterOpts{
            Subsystem: "http",
            Name:      "requests_total",
            Help:      "total HTTP requests processed",
        }, []string{"code", "method"}),
        RequestDurationHistogram: promauto.NewHistogramVec(prometheus
            Subsystem: "http",
            Name:      "request_duration_seconds",
            Help:      "Seconds spent serving HTTP requests.",
            Buckets:   prometheus.DefBuckets,
        }, []string{"code", "method"}),
    }
}


// InstrumentHandler instruments any HTTP handler for the request
// total and request duration metric
func InstrumentHandler(next http.HandlerFunc, _http HttpMetrics) http
    return promhttp.InstrumentHandlerCounter(_http.RequestsTotal,
        promhttp.InstrumentHandlerDuration(_http.RequestDurationHisto
}
```

Now create main.go:

```go
package main

import (
    "crypto/sha256"
    "encoding/hex"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "time"

    "github.com/prometheus/client_golang/prometheus/promhttp"
)


// computeSum generates a SHA256 hash of the body
func computeSum(body []byte) []byte {
    h := sha256.New()
    h.Write(body)
    hashed := hex.EncodeToString(h.Sum(nil))
    return []byte(hashed)
}

func main() {
    port := 8080
    readTimeout := time.Millisecond * 500
    writeTimeout := time.Millisecond * 500

    mux := http.NewServeMux()

    httpMetrics := NewHttpMetrics()
    instrumentedHash := InstrumentHandler(func(w http.ResponseWriter,
        if r.Method != http.MethodPost {
            w.WriteHeader(http.StatusBadRequest)
            return
        }
        if r.Body != nil {
            body, _ := ioutil.ReadAll(r.Body)
            fmt.Fprintf(w, "%s", computeSum(body))
        }
    }, httpMetrics)

    mux.HandleFunc("/hash", instrumentedHash)

    mux.Handle("/metrics", promhttp.Handler())
```

```go
    s := &http.Server{
        Addr:           fmt.Sprintf(":%d", port),
        ReadTimeout:    readTimeout,
        WriteTimeout:   writeTimeout,
        MaxHeaderBytes: 1 << 20, // Max header of 1MB
        Handler:        mux,
    }
    log.Printf("Listening on: %d", port)
    s.ListenAndServe()
}
```

Run the example and make sure Prometheus is running also.

```
# Terminal 1
$ go run .

# Terminal 2

$ curl -i http://127.0.0.1:8080/hash
HTTP/1.1 400 Bad Request
Date: Tue, 08 Jun 2021 14:30:29 GMT
Content-Length: 0

$ curl -i http://127.0.0.1:8080/hash -d "Golang"

HTTP/1.1 200 OK
Date: Tue, 08 Jun 2021 14:30:51 GMT
Content-Length: 64
Content-Type: text/plain; charset=utf-8

50e56e797c4ac89f7994a37480fce29a8a0f0f123a695e2dc32d5632197e2318
```

Now view the metrics endpoint, or visit the Prometheus dashboard.

```
$ curl http://localhost:8080/metrics

# HELP http_request_duration_seconds Seconds spent serving HTTP reque
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{code="200",method="post",le="1"}
http_request_duration_seconds_sum{code="200",method="post"} 0.0001308
http_request_duration_seconds_count{code="200",method="post"} 1
```

```
http_request_duration_seconds_bucket{code="400",method="get",le="1"}
http_request_duration_seconds_sum{code="400",method="get"} 5.4304e-05
http_request_duration_seconds_count{code="400",method="get"} 1
# HELP http_requests_total total HTTP requests processed
# TYPE http_requests_total counter
http_requests_total{code="200",method="post"} 1
http_requests_total{code="400",method="get"} 1
```

You'll see the histogram values in `http_request_duration_seconds_bucket/sum/count` and counts for each status in `http_requests_total`.

Adding custom metrics to your Prometheus data

At this point, adding your own additional custom metrics is relatively simple.

1. Define the metric and name it, make sure that it stays within the scope of your handler
2. Call `prometheus.MustRegister` on the metric
3. Call the metric's method within your handler to increment/decrement or record the custom data

Edit the previous example (custom-metrics). I've removed the RED metrics for brevity, but you can combine the two.

```go
package main

import (
    "crypto/sha256"
    "encoding/hex"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "time"

    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

// computeSum generates a SHA256 hash of the body
func computeSum(body []byte) []byte {
```

```go
    h := sha256.New()
    h.Write(body)
    hashed := hex.EncodeToString(h.Sum(nil))
    time.Sleep(time.Second * 5)
    return []byte(hashed)
}
```

Note the 5 second sleep added to show the concurrent requests. It may be hard to see the gauge going up unless we add some latency.

```go
func main() {
    port := 8080
    readTimeout := time.Second * 10
    writeTimeout := time.Second * 10

    mux := http.NewServeMux()

    hashesInflight := prometheus.NewGauge(prometheus.GaugeOpts{
        Subsystem: "http",
        Name:      "hashes_inflight",
        Help:      "total hashes inflight",
    })
    prometheus.MustRegister(hashesInflight)
```

The metric such as a Gauge needs to be defined within scope of the HTTP handler where it's going to be used. And `prometheus.MustRegister` is also required.

Then simply call `metric.Inc()` when starting and `metric.Dec()` when done.

```go
    customHash := func(w http.ResponseWriter, r *http.Request) {
        if r.Method != http.MethodPost {
            w.WriteHeader(http.StatusBadRequest)
            return
        }
        if r.Body != nil {
            defer r.Body.Close()

            hashesInflight.Inc()
            defer hashesInflight.Dec()

            body, _ := ioutil.ReadAll(r.Body)
            fmt.Fprintf(w, "%s", computeSum(body))
        }
    }
```

The rest of the code is as before:

```
    mux.HandleFunc("/hash", customHash)
    mux.Handle("/metrics", promhttp.Handler())

    s := &http.Server{
        Addr:           fmt.Sprintf(":%d", port),
        ReadTimeout:    readTimeout,
        WriteTimeout:   writeTimeout,
        MaxHeaderBytes: 1 << 20, // Max header of 1MB
        Handler:        mux,
    }
    log.Printf("Listening on: %d", port)
    s.ListenAndServe()
}
```

Run the example:

```
# Terminal 1:
$ go run main.go

# Terminal 2:

# Run this 3-5 times quickly hitting up in between each
$ curl localhost:8080/hash -d test &

# Then quickly run:

$ curl -s localhost:8080/metrics | grep inflight
# HELP promhttp_metric_handler_requests_in_flight Current number of s
# TYPE promhttp_metric_handler_requests_in_flight gauge
promhttp_metric_handler_requests_in_flight 5
```

Now you have learned how to add custom metrics and standard HTTP RED metrics for tracking your functions.

Building and releasing Go binaries

Whether you've written a CLI or a HTTP server, most of the time Go code can be compiled down to a static binary which is self-contained and can run on the target system you specify.

In an earlier chapter we covered cross-compilation, at this point you should consider whether you need to target one environment such as Windows, Linux or MacOS, or multiple.

The release-it example

Release-it repo is an example of a static Go binary that can be built and released through GitHub Actions. It uses a Makefile, so it can also be run in other CI pipelines or manually as and when you create a new release.

The Makefile has the following properties, all of which were covered in previous chapters already:

- It builds binaries for multiple platforms and Operating Systems
- It injects the Commit and Version from git into the final binary

```
Version := $(shell git describe --tags --dirty)
# Version := "dev"
GitCommit := $(shell git rev-parse HEAD)
LDFLAGS := "-s -w -X main.Version=$(Version) -X main.GitCommit=$(GitC

.PHONY: all
all: gofmt dist

.PHONY: gofmt
gofmt:
	@test -z $(shell gofmt -l ./ | tee /dev/stderr) || (echo "[WARN]

.PHONY: dist
dist:
	mkdir -p bin/
	CGO_ENABLED=0 GOOS=linux go build -mod=vendor -a -ldflags $(LDFLA
	CGO_ENABLED=0 GOOS=darwin go build -mod=vendor -a -ldflags $(LDFL
```

```
GOARM=7 GOARCH=arm CGO_ENABLED=0 GOOS=linux go build -mod=vendor
GOARCH=arm64 CGO_ENABLED=0 GOOS=linux go build -mod=vendor -a -ld
GOOS=windows CGO_ENABLED=0 go build -mod=vendor -a -ldflags $(LDF
```

In addition, it runs `gofmt` for good measure.

If there were any unit tests in the project, then we could add an additional target
for them, such as:

```
.PHONY: gotest
gotest:
    go test ./...
```

Then update the `all` line to `gofmt gotest dist` which would run each of the
tasks for us.

Clone the example and build it:

```
$ mkdir $GOPATH/src/github.com/alexellis/
$ cd $GOPATH/src/github.com/alexellis/
$ git clone https://github.com/alexellis/release-it
$ cd release-it
$ make
```

By default `make` will try to run the `all` target, but you can also run `make gofmt` if
you want to test formatting before committing to a build.

You'll find each binary in the `bin/` folder and can run them on your system if it
matches the OS/architecture pair, or copy it to another system to run it there.
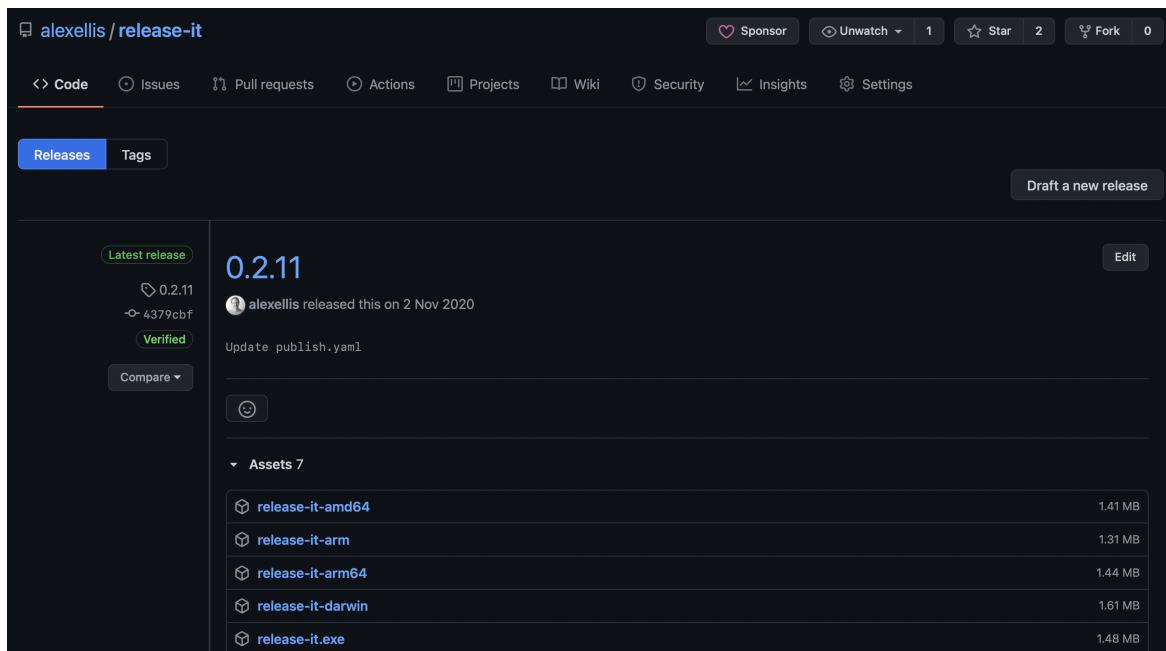
GitHub's release page

When you release binaries to users, GitHub's release functionality can come in handy because it allows you to upload the binaries manually or via automation.

Releasing your binaries with a GitHub Action

[GitHub Actions](#) is free to use for open source repositories (i.e public ones) and comes with a number of free minutes for private ones also. This example should be adaptable to other platforms like Jenkins or GitLab CI, since the concepts are largely the same.

In our example, we'll first checkout the code from GitHub for the release tag, then run `make all`, followed by uploading the release artifacts to the GitHub release.

```
name: publish

on:
  push:
    tags:
      - '*'

jobs:
  publish:
```

```
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master
        with:
          fetch-depth: 1
      - name: Make all
        run: make all
      - name: Upload release binaries
        uses: alexellis/upload-assets@0.2.2
        env:
          GITHUB_TOKEN: ${{ github.token }}
        with:
          asset_paths: '["./bin/release-it*"]'
```

Each step in the workflow file can use a specific GitHub Action like `alexellis/upload-assets@0.2.2` or `actions/checkout@master`, but when you leave that field blank, it uses the base system such as `ubuntu-latest`. On the base Ubuntu runner you can execute standard shell and build commands like `make all`.

The alexellis/upload-assets action is one that I built to use on all the various OSS projects I maintain and allows a number of files to be uploaded.

Users can then go ahead and download your binary and run it on their system.

For instance:

```
$ cd /tmp/
$ curl -sLSf \
  -o ./release-it \
  https://github.com/alexellis/release-it/releases/download/0.2.11/re
    && chmod +x ./release-it \
    && ./release-it

release-it: 0.2.11, commit: 4379cbf694c55bd98f5080d76ac5972fd89dcc65
```

Releasing a Docker container image

I'll use the term Docker image and container image interchangeably, since many people only know containers by [Docker](#).

First of all, you'll need a Dockerfile which has a similar purpose to the Makefile from earlier: to build a binary that can be shipped to users.

This Dockerfile is going to be multi-stage, and also multi-architecture.

- multi-stage - the SDKs used to build the code are not shipped in the final image
- multi-archi(tecture) - the output can run on more than one target platform

Multi-arch builds are important when your target platform is an Arm server and your development machine is Intel or AMD, and visa-versa if you're an Apple M1 user.

We'll continue to use the [release-it](#) repo from the previous section.

```
FROM --platform=${BUILDPLATFORM:-linux/amd64} golang:1.15 as bui

ARG TARGETPLATFORM
ARG BUILDPLATFORM
ARG TARGETOS
ARG TARGETARCH

ARG Version
ARG GitCommit

ENV CGO_ENABLED=0
ENV GO111MODULE=on

WORKDIR /go/src/github.com/alexellis/release-it
```

```
COPY .  .

RUN CGO_ENABLED=${CGO_ENABLED} GOOS=${TARGETOS} GOARCH=${TARGET
  go test -v ./...

RUN CGO_ENABLED=${CGO_ENABLED} GOOS=${TARGETOS} GOARCH=${TARGET
  go build -ldflags \
  "-s -w -X 'main.Version=${Version}' -X 'main.GitCommit=${GitCc
  -a -installsuffix cgo -o /usr/bin/release-it .

FROM --platform=${BUILDPLATFORM:-linux/amd64} gcr.io/distroless/
WORKDIR /
COPY --from=builder /usr/bin/release-it /
USER nonroot:nonroot

CMD ["/release-it"]
```

To build the container for your current platform, run:

```
export DOCKER_BUILDKIT=1

docker build -t release-it:0.1.0 .
```

In the output, you'll see that I'm using an Apple M1 today, which means
Docker Deskop is running as an ARM64 Linux VM, so this output can only
be run on an ARM64 server or another M1 with Docker Desktop.

```
=> [builder 4/5] RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm64   gc
=> [builder 5/5] RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm64   gc
```

Then try the container out:

```
docker run --rm --name release-it \
  -ti release-it:0.1.0
release-it: , commit:
```

Notice how the version information is lacking?

We need to inject it into the build via what Docker calls *build arguments*.

```
$ docker build \
 --build-arg Version=0.1.0 \
 --build-arg GitCommit=b9bfc0508 \
 -t release-it:0.1.0 .

$ docker run docker.io/library/release-it:0.1.0
release-it: 0.1.0, commit: b9bfc0508
```

Going multi-arch

Now that we have a multi-stage image, which is efficient and only ships what is necessary to users, it's time to build for multiple platforms. This technique relies on the buildx addition which is preinstalled for Docker Desktop, for Linux users, see the [Docker documentation](#) on how to add it to your system.

```
$ docker buildx build \
 --build-arg Version=0.1.0 \
 --build-arg GitCommit=b9bfc0508 \
 --platform linux/arm64 \
 --platform linux/amd64 \
 --push \
 -t alexellis2/release-it:0.1.0 .
```

Note the new `--platform` flag which specifies a target of a Linux OS for both ARM64 servers and AMD64 (Intel/AMD) servers. You can also specify other architectures like a 32-bit Raspberry Pi.

The `--push` flag is required for multi-arch images, since they cannot be kept in your library at this time, they must be pushed directly to a registry.

You can however, now log into both an ARM64 server or an Intel/AMD server and run the container:

```
$ docker run -ti alexellis2/release-it:0.1.0
release-it: 0.1.0, commit: b9bfc0508
```

Congratulations, you've now built a static Golang binary, and shipped it to a GitHub releases page and shipped your first multi-arch Docker that can run on two different target platforms.

Shipping a Docker container from GitHub Actions

Now, the final step if you want to release a container image for your GitHub repository is to define a GitHub action for it.

Create the following file as `.github/workflows/publish.yaml` and commit it. It will run a release build upon every tag you create, so every GitHub Release created in the UI or API for your repo.

```yaml
name: publish

on:
  push:
    tags:
      - '*'

jobs:
  publish:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master
        with:

          fetch-depth: 1
      - name: Get TAG
        id: get_tag
        run: echo ::set-output name=TAG::${GITHUB_REF#refs/tags/
      - name: Set up QEMU
        uses: docker/setup-qemu-action@v1
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      - name: Login to Docker Registry
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
```

```
        password: ${{ secrets.DOCKER_PASSWORD }}
        registry: ghcr.io
  - name: Release build
    id: release_build
    uses: docker/build-push-action@v2
    with:
      outputs: "type=registry,push=true"
      platforms: linux/amd64,linux/arm/v6,linux/arm64
      build-args: |
        Version=${{ steps.get_tag.outputs.TAG }}
        GitCommit=${{ github.sha }}
      tags: |
        ghcr.io/alexellis/release-it:${{ github.sha }}
        ghcr.io/alexellis/release-it:${{ steps.get_tag.outpu
        ghcr.io/alexellis/release-it:latest
```

You can see that the `build-args` are populated just as when we ran `docker build` or buildx locally, and then we have some additional actions for setting up qemu. Qemu is used by buildx to cross compile your code to work on other machines.

The list of platforms (`linux/amd64,linux/arm/v6,linux/arm64`) means this container will work on Raspberry Pi OS, 64-bit Arm servers and regular Intel/AMD servers too.

Before triggering your first build, you'll need to create the `DOCKER_USERNAME` and `DOCKER_PASSWORD` secrets. Enter these on the Repository's Settings page under the *Secrets* tab. If you're using GitHub's container registry (GHCR), then the username is your GitHub login.

In a recent change to GitHub Actions, you can replace `password: ${{ secrets.DOCKER_PASSWORD }}` with `password: ${{ GITHUB_TOKEN }}`. The example is provided to be generic however, so that you can use it with the [Docker Hub](#), [Quay](#), or another registry.

> Note that with GHCR, your container image will be set to private by default. You'll also need to make it public to share it, or authenticate your Docker engine or Kubernetes cluster otherwise.

Your onward journey

I hope you've enjoyed reading my recipes and examples which have been developed over the past 5-6 years of my own journey with Go and the Open Source community. Of course I am still learning all the time and some of the examples in the book used to be blog posts, but are now out of date in comparison.

Giving feedback

If you have feedback, or have spotted errors, then feel free to reach out to me so that we can improve the eBook for everyone else. Email contact@openfaas.com.

Likewise, if your team or company is using Go at work and would like advice, or to speak to someone who is experienced with Go and distributed systems, feel free to reach out.

Contributing to Open Source

We're also looking for open source contributors to various Open Source projects that the community maintains. This can be a good way to cut your teeth and see some of what you've been learning into action.

- OpenFaaS - portable, open source serverless functions
- arkade - marketplace to install CLIs directly from GitHub releases and Kubernetes apps
- k3sup - bootstrap Kubernetes clusters on PCs and cloud using K3s
- derek - a GitHub bot that automates maintainer tasks on open source projects, and delegates permissions to contributors through issue/PR comments
- inlets-operator - a Kubernetes Operator that gives you public IPs for any of your LoadBalancer services on private Kubernetes clusters

My other content

OpenFaaS itself is written in Go, it's a serverless platform that can run on Kubernetes, or on a single VM or Raspberry Pi. The eBook and video workshop [Serverless For Everyone Else](#) has sold 450 copies already, and is a great place to start learning about functions.

I also have a workshop on building Raspberry Pi Clusters with Kubernetes (k3s) which covers how to boot a Raspberry Pi from network storage. Network storage is faster and more efficient than SD cards. The eBook and video workshop are available separately or as a bundle on Gumroad: [Netbooting workshop for Raspberry Pi with K3s](#)

If you [follow me on Twitter @alexellisuk](https://twitter.com/alexellisuk), you'll be able to see when I have new videos, events and blog posts, which may be published in a number of places.

Finally, thank you for purchasing this eBook and I hope you enjoy your journey with Golang, wherever you are on that path.