API Foundations
in Go

by Tit Petric

API foundations in Go

You've mastered PHP and are looking at Node.js? Skip it and try Go.

Tit Petric

This book is for sale at http://leanpub.com/api-foundations

This version was published on 2019-01-15

Tweet This Book!

Please help Tit Petric by spreading the word about this book on Twitter!

The suggested hashtag for this book is #apifoundations.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#apifoundations

I'm dedicating this book to my wife, Anastasia. Even if you say that you don't suffer, I'm sure that I'm killing your inner child by explaining what Docker is. I'm sorry for that.

Contents

CONTENTS

CONTENTS

Introduction

About me

I'm one of those people with about two decades of programming experience under my belt. I started optimizing code in the '90s, discovered PHP in the 2000s, and have built several large-scale projects to date, all while discovering other programming language families like Java, Node.js and ultimately, Go.

I have built numerous APIs for my own content management products. Several products I've been involved with as a lead developer have been sold and are used in multiple countries. I've written a professional, dedicated API framework, which doubles as a software development kit for the Slovenian national TV and Radio station website, RTV Slovenia. I've also been a speaker at several local PHP user group events and conferences.

Why Go?

Go has been on my list for a while now. About a year or two ago, my coworker created a protocol converter that emulates a Memcache server, using Redis as the backend datastore. Since our biggest project has a code base built up over some 13 years, it was preferable to keep the working code as-is, and just replace the software around it as needed. Go made this possible.

One of the reasons for choosing Go was the constant comparison of Go with Node. Node has a much more vocal community, which seems to religiously advocate it. We have carried out several tests in recent months, and Node, while not very hard to start development with, had poorer performance than pretty much everything except PHP. I'm not saying that Go is better than Node, or that anything is better than anything else, but from what we've seen, it seems advisable to skip Node.js and go straight to Go. This might change as ES6 and ES7 get more traction - but there are immediate benefits of switching to Go. If you don't want to move from Node.js, this book is not for you. If you have an open mind - find out what Go can do.

Who is this book for?

This book is for senior developers who might not have had a chance to try Go, but are familiar with concepts of API development in languages like PHP or Node. Any reader must have a good understanding of REST APIs and server architecture. While I'll try to make everything as clear as possible, realize that if you're a novice programmer, there might be a gap between what you know, and what I'm trying to explain here.

I'm not going to be explaining the Go programming language in detail. I'm going to be diving head first into using it with just a note here and there. This is why familiarity and strong knowledge of programming concepts are required.

In the book, I will cover these subjects:

1. Setting up your environment
2. Data structures
3. Organizing your code
4. Encoding and decoding JSON
5. Serving HTTP requests
6. Parallel fetching of data
7. Using external services (Redis)
8. Using external services (MySQL)
9. Test driven API development
10. Your first API
11. Running your API in production
12. Resources

Covering these concepts should give you a strong foundation for your API implementation. The book doesn't try to teach you Go; the book tries to give you a strong software foundation for APIs, using Go.

How should I study it?

Through the book, I will present several examples on how to do common things when developing APIs. The examples are published on GitHub, you can find the link in the last chapter of the book.

You should follow the examples in the book, or you can look at each chapter individually, just to cover the knowledge of that chapter. The examples are stand-alone, but generally build on work from previous chapters.

Setting up your environment

Setting up a development environment, as well as a production environment, is an important topic today. While spinning up a virtual machine and installing software by hand is perhaps the accepted way of doing things, recently I've learned to systematize my development by using Docker containers.

The biggest benefit of Docker containers is a "zero install" way of running software - as you're about to run a container, it downloads the image containing all the software dependencies you need. You can take this image and copy it to your production server, where you can run it without any change in environment.

Networking

When you have docker set up, we will need to create a network so the services that we're going to use can talk to each other. Creating a network is simple, all you need to do is run the following command:

```
$ docker network create -d bridge --subnet 172.25.0.0/24 party
```

This command will create a network named **party** on the specified subnet. All docker containers which will run on this network will have connectivity to each other. That means that when we run our Go container, it will be able to connect to another Redis container on the same network.

Setting up a runner for your Go programs

There is an official Go image available on Docker. Getting a Docker image and running it is very simple, requiring just one line:

```
$ docker run --net=party -p 8080:80 --rm=true -it -v `pwd`:/go/src/app -w /go/src/ap\
p golang go "$@"
```

Save this code snippet as the file 'go', make it executable and copy it to your execution path (usually /usr/local/bin is reserved for things like this).

Let's quickly go over the arguments provided:

- **–net=party** - runs the container on the shared network

- **-p** - network forwarding from host:8080 to container:80 (HTTP server)
- **−rm=true** - when the container stops, clean up after it (saves disk space)
- **-v option** - creates a volume from the current path (pwd) in the container
- **"$@"** - passes all arguments to go to the container application

Very simply, what this command does is run a Docker container in your current folder, execute the go binary in the container, and clean up after itself when the program finishes.

> *Note*: as we only expose the current working path to the container, it limits access to the host machine - if you have code outside the current path, for example in ".." or "/usr/share", this code will not be available to the container.

An example of running go would be:

```
$ go version
go version go1.6 linux/amd64
```

And, what we will do through most of this book is run the following command:

- **go run [file]** - compile and run Go program

```
$ go run hello_world.go
Hello world!
```

A minimal example of a Go program

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Printf("Hello world!\n")
7  }
```

Setting up Redis

We will also use Docker to run Redis, which is as simple as setting up Go. Redis is an in-memory data structure store. It provides functionality to store and retrieve data in various structures, beyond a simple key-value database like Memcache. We will use this service later in the book when implementing our example API endpoints.

To run an instance of Redis named 'redis':

```
$ docker run --restart=always -h redis --name redis --net=party -d redis
```

Just like Go, Redis provides an official build on the Docker Hub. Interacting with Redis will be covered in detail in later chapters.

Other services

In addition to Go and Redis, other services are also available on the Docker Hub[1]. Depending on your use case, you might want to install additional services via docker.

Popular projects which I use from Docker on a daily basis:

- nginx (and nginx-extras flavours)
- Percona (MySQL, MariaDB) - also covered later in the book
- letsencrypt
- redis
- samba
- php

Docker is a very powerful tool which gives you all the software you might need. It's very useful also for testing, as you can use it to set up a database, populate it with test data, and then tear down and clean up after it. It's a very convenient way to run programs that are isolated from your actual environment, and may only be active temporary.

---

[1]https://hub.docker.com/

Data structures

Defining and handling data structures is a key activity in any programming language. When talking about object oriented programming, it's worth noting some differences between Go and other programming languages.

- Go doesn't have classes, it has structs
- Methods are bound to a struct, not declared within one
- The "interface" type can be an intersection of many or all types
- Packages behave like namespaces, but everything in a package is available

In short, it means that the functions you'll define for your structs will be declared outside of the struct they are working on. If you want to declare several types, the only way to assign them to a same variable, without many complications, is to declare a common interface.

Declaring structs

When we define a structure in Go, we set types for every member. Let's say we would like to define the usual "Petstore", which in turn has a list of pets. You could define the structure like this:

```
1  type Petstore struct {
2    Name string
3    Location string
4    Dogs []*Pet
5    Cats []*Pet
6  }
7
8  type Pet struct {
9    Name string
10   Breed string
11 }
```

The example is simple, in the sense that I'm not defining a "Pet", which can be a "Dog", or can be a "Cat". I'm just using one type for all.

> *Note*: Members of defined structures begin with an upper case letter. This means that they are visible outside the package they are defined in. All members, which you will encode to JSON in the next chapter, need to be public, that is - start with an upper case letter. The same applies to functions that you expose in packages.

Casting structs

We could declare a Dog and Cat struct, and could cast one to one from the other. A requirement for this is that the Dog and Cat types *must have identical underlying types*. So, if I was to extend the above example, I would copy the Pet struct into Dog and Cat so they are identical.

If they are not identical, the value can't be cast.

Error when casting

```
1   type Dog struct {
2     name string
3   }
4   type Cat struct {
5     name          string
6     hypoallergenic bool
7   }
8
9   func main() {
10    dog := Dog{name: "Rex"}
11    cat := Cat(dog)
12  }
```

The above example results in:

```
./type1.go:12: cannot convert dog (type Dog) to type Cat
```

Even if "Cat" has all the properties of "Dog", type conversion is not possible. It makes no sense to declare strong types of individual animals, if casting to a generic type is not possible. But we can assign these kind of strong types to a common interface type.

Declaring interfaces

An interface defines zero or more methods that need to be implemented on types/objects that can be assigned to that interface. If we extend the above example to declare explicit types for Cat and Dog, we will end up with something like this:

Structure for an agnostic pet list

```go
1   type Dog struct {
2     name  string
3     breed string
4   }
5   type Cat struct {
6     name          string
7     hypoallergenic bool
8   }
9   type Petstore struct {
10    name string
11    pets []interface{}
12  }
```

We are using the declaration of `pets` as "interface{}". The interface doesn't define any common methods, so anything can be assigned to it. Usually, one would implement getter or setter methods which should be common to all the types an interface can hold. With our example, we could have declared a function `GetName` that would return the name of the Pet:

Structure for an agnostic pet list

```go
1   type Pet interface {
2     getName() string
3   }
4
5   func (d Dog) getName() string {
6     return d.name
7   }
8   func (c Cat) getName() string {
9     return c.name
10  }
```

Abusing interfaces

Interfaces are a powerful and flexible way to work with objects of varying types. You should however tend to prefer static type declarations when possible - it makes the code faster and more readable. The Go compiler catches a lot of your common errors with static analysis. If you're using interfaces, you're exposing yourself to risks and errors which the compiler cannot catch.

All the variables which have been declared in the object are unreachable from the interface and will result in an error. But that doesn't mean that you can't access them.

If you absolutely must, you can use a feature called "reflection". Reflection exposes the members of any interface via an API.

"Hello world!" example using Reflection

```go
1   package main
2
3   import "fmt"
4   import "reflect"
5
6   type Test1 struct {
7     A string
8   }
9
10  func main() {
11    var t interface{}
12    t = Test1{"Hello world!"}
13
14    data := reflect.ValueOf(t)
15    fmt.Printf("%s\n", data.FieldByName("A"))
16  }
```

Interfaces can be a powerful part of Go, but care should be taken not to overuse them. They make many tasks simpler and can be a powerful tool when they are used correctly.

> Reflection is used by many packages to provide generic function interfaces, like the encoding/json package does, which we will use in a later chapter.

Embedding and composition

You can embed types into a struct by listing them in the struct declaration. This makes all the values and functions of the embedded struct available for use on your struct. For example, the sync.Mutex struct implements Lock and Unlock functions.

```
 1   type Services struct {
 2     sync.Mutex
 3     ServiceList map[string]Service
 4   }
 5
 6   func (r *Services) Add(name string, service Service) {
 7     r.Lock()
 8     r.ServiceList[name] = service
 9     r.Unlock()
10   }
```

This pattern enables composition. You can build your APIs for a larger program from smaller structures that can deal with specific problem domains. The example also shows a beneficial side-effect: you can reason that the embedded `sync.Mutex` is used to lock your structure fields below the embed.

You can use embedding to your advantage with, for example, `sync.Mutex`, `sync.RWMutex` or `sync.WaitGroup`. You can actually embed *many* structs, so your structure may perform the functions of them all.

An example from a project I'm working on uses two embedded structs:

```
 1   type RunQueue struct {
 2     sync.RWMutex
 3     sync.WaitGroup
 4     // ...
 5     flagIsDone   bool
 6   }
 7   func (r *RunQueue) Close() {
 8     r.Lock()
 9     defer r.Unlock()
10     r.flagIsDone = true
11   }
12   func (r *RunQueue) IsDone() bool {
13     r.RLock()
14     defer r.RUnlock()
15     return r.flagIsDone
16   }
```

Leveraging `sync.RWMutex`

The declaration of the RunQueue struct above leverages the `sync.RWMutex` to provide synchronous access to the object from many goroutines. A goroutine may use `Close` to finish the execution of the goroutine queue. Each worker in the queue would call `IsDone` to check if the queue is still active.

Leveraging `sync.WaitGroup`

The RunQueue struct leverages a `sync.WaitGroup` to provide queue clean up and statistics, such as elapsed time. While I can't provide all the code, the basic usage is as follows:

```
1   func (r *RunQueue) Runner() {
2     fmt.Printf("Starting %d runners\n", runtime.NumCPU())
3     for idx := 1; idx <= runtime.NumCPU(); idx++ {
4       go r.runOnce(idx)
5     }
6   }
7   func NewRunQueue(jobs []Command) RunQueue {
8     q := RunQueue{}
9     for idx, job := range jobs {
10      if job.SelfId == 0 {
11        q.Dispatch(&jobs[idx])
12      }
13    }
14    q.Add(len(q.jobs)) // sync.WaitGroup
15    return q
16  }
17  runnerQueue := NewRunQueue(commands)
18  go runnerQueue.Finisher()
19  go runnerQueue.Runner()
20  runnerQueue.Wait() // sync.WaitGroup
```

The main idea of the program I'm building is that it starts `runtime.NumCPU()` runners, which handle execution of a fixed number of commands. The `WaitGroup` comes into play very simply:

```
1   NewRunQueue calls *wg.Add(number of jobs)
2   Individual jobs are processed with RunQueue.runOnce, and they call *wg.Done()
3   RunnerQueue.Wait() (*wg.Wait()) will wait until all jobs have been processed
```

Limiting goroutine parallelization

At one point I struggled to create a queue manager, which would parallelize workloads to a fixed limit of parallel goroutines. My idea was to register a slot manager, which would provide a pool of running tasks. If no pool slot is available, it'd sleep for a few seconds before trying to get a slot again. It was frustrating.

Just look at the loop from the `Runner` function above:

```
1  for idx := 1; idx <= runtime.NumCPU(); idx++ {
2    go r.runOnce(idx)
3  }
```

This is an elegant way to limit parallelization to N routines. There is no need to bother yourself with some kind of routine allocation pool structs. The `runOnce` function should only do a few things:

1. Listen for new jobs in an infinite loop, reading jobs from a channel
2. Perform the job without new goroutines

The reason to read the jobs from a channel is that the read from a channel is a blocking operation. The function will just wait there until a new job appears on the channel it's reading from.

```
1  func (r *RunQueue) runOnce(idx int) {
2    for {
3      queueJob, ok := <-r.runQueue
4      if !ok {
5        return
6      }
7      // run tasks
8  [...]
```

The job needs to be executed without a goroutine, or with nested `*WaitGroup.Wait()` call. The reason for this should be obvious - as soon as you start a new goroutine, it gets executed in parallel and the `runOnce` function reads the next job from the queue. This means that the limit on the number of tasks running in parallel would not be enforced.

Slices

A Slice is a Go-specific data type, which consists of:

1. a pointer to an array of values,
2. the capacity of the array,
3. the length of the array

If you're working with slices, which you mostly are, you'll inevitably find yourself in a situation where you'll have to merge two slices into one. The naïve way of doing this is something like the following:

```
1  alice := []string{"foo", "bar"}
2  bob := []string{"verdana", "tahoma", "arial"}
3  for _, val := range bob {
4      alice = append(alice, val)
5  }
```

A slightly more idiomatic way of doing this is:

```
1  alice = append(alice, bob...)
```

The `ellipsis` or variadic argument expands `bob` in this case to all its members, effectively achieving the same result, without that loop. You can read more about it in the official documentation: [Appending to and copying slices](#)[2]

You could use it to wrap logic around `log.Printf` that, for example, trims and appends a newline at the end of the format string.

```
1  import "log"
2  import "strings"
3
4  func Log(format string, v ...interface{}) {
5      format = strings.TrimSuffix(format, "\n") + "\n"
6      log.Printf(format, v...)
7  }
```

Since slices are basically a special type of pointer, this means there are a number of scenarios where you may inadvertently modify a slice without intending to. Thus, it's important to know how to manipulate slices.

The slice operator

The first thing one should be aware of is that the slice operator, *does not copy the data to a newly-created slice.* Not being aware of this fact can lead to unexpected results:

```
1  a := []string{"r","u","n"}
2  b := a[1:2]
3  b[0] = "a"
4  fmt.Println(a)
```

---

[2]https://golang.org/ref/spec#Appending_and_copying_slices

Does it print [r u n] or [r a n]? Since the slice b is not a copy of the slice, but just a slice with a modified pointer to the array, the above will print [r a n].

As explained, this is because the slice operator just provides a new slice with an updated reference to the same array as used by the original slice. From the official reference:

> Slicing does not copy the slice's data. It creates a new slice value that points to the original array. This makes slice operations as efficient as manipulating array indices. Therefore, modifying the elements (not the slice itself) of a re-slice modifies the elements of the original slice.

Source: Go Blog - Slices usage and internals[3].

Allocation by append

Appending to a slice is simple enough. As mentioned, the slice has a length which you can get with a call to len(), and a capacity which you can get with a call to cap().

```
1  a := []string{"r", "u", "n"}
2  fmt.Println(a, len(a), cap(a))
3  a = append(a, []string{"e"}...)
4  fmt.Println(a, len(a), cap(a))
5  a = a[1:len(a)]
6  fmt.Println(a, len(a), cap(a))
```

The expected output would be:

```
1  [r u n] 3 3
2  [r u n e] 4 4
3  [u n e] 3 3
```

Of course, that's not how append works. Append will double the existing capacity. This means you'll end up with output like this:

```
1  [r u n] 3 3
2  [r u n e] 4 6
3  [u n e] 3 5
```

If you wanted the previous result, you would have to create your own function, which would allocate only the required amount of items into a new slice, and then copy over the data from the source slice. This function might look something like this:

---

[3]https://blog.golang.org/go-slices-usage-and-internals

```
 1   func suffix(source []string, vars ...string) []string {
 2     length := len(source) + len(vars)
 3     ret := make([]string, length, length)
 4     copy(ret, source)
 5     index := len(source)
 6     for k, v := range vars {
 7       ret[index+k] = v
 8     }
 9     return ret
10   }
11
12   func main() {
13     a := []string{"r", "u", "n"}
14     fmt.Println(a, len(a), cap(a))
15     a = suffix(a, []string{"e"}...)
16     fmt.Println(a, len(a), cap(a))
17     a = a[1:len(a)]
18     fmt.Println(a, len(a), cap(a))
19   }
```

Copying slices

As explained above, "slicing does not copy the slice's data". This may sometimes have unintended consequences. This applies not only to slicing, but also to passing slices into functions.

```
 1   func flip(source []string) {
 2     source[1] = "a"
 3   }
 4
 5   func main() {
 6     a := []string{"r", "u", "n"}
 7     flip(a)
 8     fmt.Println(a)
 9   }
```

The above example will print `[r a n]`. This is unfortunate, because people intuitively expect slices to behave much like structs do. Passing a struct will create a copy. A slice will still point at the same memory that holds the data. Even if you pass a slice within a struct, you should be aware of this:

```
1   type X struct {
2     c string
3     source []string
4   }
5
6   func flip(x X) {
7     x.c = "b"
8     x.source[1] = "a"
9   }
10
11  func main() {
12    a := X{"a", []string{"r", "u", "n"}}
13    flip(a)
14    fmt.Println(a)
15  }
```

The above will print out a {a [r a n]}. Slices always behave as if they are passed by reference. In a way they are, as one of the parts of the slice is a pointer to the array of data it holds.

Using slices in channels

If you're using buffered channels, and are trying to be smart with slices, reusing a slice to keep allocations down, then you might find that this buffered channel is basically filled with the same slice over and over. As the slice contains the pointer to the array holding the data, the following example will have unexpected results:

```
1   func main() {
2     c := make(chan []string, 5)
3
4     go func() {
5       item := []string{"hello"}
6       for i := 0; i < 5; i++ {
7         item[0] = fmt.Sprintf("hello %d", i)
8         c <- item
9         //time.Sleep(100 * time.Millisecond)
10      }
11    }()
12
13    for i := 0; i < 5; i++ {
14      item := <-c
15      fmt.Println(item)
```

```
16      }
17  }
```

The output is:

```
1  [hello 4]
2  [hello 4]
3  [hello 4]
4  [hello 4]
5  [hello 4]
```

If you uncomment the `time.Sleep` command in the goroutine, you will most likely get the correct result:

```
1  [hello 0]
2  [hello 1]
3  [hello 2]
4  [hello 3]
5  [hello 4]
```

At any time when the consumer is reading from the channel, the retrieved slice will be identical, because only the item in the underlying array is changing. The only solution for this, I believe, is either finding a way to work with an unbuffered channel (a consumer must *always* be waiting to read from the channel), or to explicitly copy the slice which is being put into the channel, like in this playground example[4].

---

[4]https://play.golang.org/p/ORPyzf0zwY

Organizing your code

Whenever you start a project, there's always the question of how you will organize your code. Go gives you the option to organize your code in packages. You can install these packages with the command `gvt fetch [package]`. You will use several packages during the book, and the examples will also create some packages, which we will explain in this section.

Gvt stands for "Go vendoring tool". Alternatively, you may use `go get` to install packages into the global Go namespace. Vendoring your dependencies is a better option, so use `gvt`, or decide on which other vendoring tool you would like to use.

It's perfectly possible to run `gvt` from Docker. I use this:

```
1   function gvt {
2       echo "== gvt" "$@" "=="
3       if [ $1 == "fetch" ]; then
4           BASE="vendor/"
5           if [ -d "$BASE$2" ]; then
6               return
7           fi
8       fi
9       docker run --dns=8.8.8.8 \
10               --dns=8.8.4.4 \
11               --rm=true -i \
12               -v $(pwd):/go/src \
13               justincormack/gvt "$@"
14  }
```

Suggested package structure

Depending on how you will organize your API structure, you can create one or many packages. You create packages in folders. I'm going to call our project "foundations". I will create one folder:

- `foundations/bootstrap` - this folder will keep various utility functions

You can create additional packages for whichever API you need, from which you can now import the `foundations/bootstrap` package as needed.

What to put there?

The bootstrap package should contain functions that you often need. A very simple function would be this:

Example bootstrap file - now.go

```go
package bootstrap

import "time"

var StartTime float64

func Now() float64 {
  myTime := float64(time.Now().UnixNano()) / 1000000.0
  if StartTime < 0.000001 {
    StartTime = myTime
  }
  return myTime - StartTime
}
```

If you need to time how long an operation takes, then this function can come in very handy. Since we're dealing with writing API calls, we will use this function in the following chapters, and we will add some additional functions along the way.

The function Now will return the time in milliseconds since the last time the function was called. We also define a public variable StartTime. We can set this variable to 0 to reset the count.

While you save your functions or function groups to individual files, all of the files together compose the package. You can separate the functions by their intent or responsibility, instead of creating just one bootstrap.go file and having everything in there.

Example usage of bootstrap package

```go
package main

import "foundations/bootstrap"
import "fmt"

func main() {
  fmt.Printf("Time: %.4f\n", bootstrap.Now())
  fmt.Printf("Hello world!\n")
  fmt.Printf("Time: %.4f\n", bootstrap.Now())

  bootstrap.StartTime = 0
```

```
12    fmt.Printf("Time after reset: %.4f\n", bootstrap.Now())
13  }
```

The example above produces this output:

```
Time: 0.0000
Hello world!
Time: 0.0850
Time after reset: 0.0000
```

> Go supports vendoring, where packages can be downloaded from their VCS locations. This means that you can use `import` with the location of the package on GitHub or other code hosting services. There are a number of tools available for vendoring. I prefer `gvt`, which I also use in the samples available on GitHub.

Format your source code

One utility that comes with Go is the `go fmt` command. It will rewrite your code to use the commonly-accepted indentation (tabs) and remove semicolons at the end of lines.

There are benefits to this as:

1. You don't need to worry about some minor formatting when writing code,
2. Reading code from other people becomes easier as the formatting is the same,
3. You need never again discuss spacing or brace position with other devs

The sooner you learn to use it the better. There is a git hook[5] available which checks if your files have been formatted before allowing you to commit them into git.

---

[5] https://golang.org/misc/git/pre-commit

Encoding and decoding JSON

Encoding and decoding JSON documents is a typical requirement of API services. Your RESTful API endpoints should provide JSON so they can be consumed by HTTP clients and a lot of public APIs are available in this format as well. It is crucial when writing APIs to familiarize yourself with parsing and writing JSON data.

The Go standard library provides encoding/json[6] to provide the functionality of recoding and encoding JSON contents. The first thing to specify when encoding structures to JSON is the names of the fields they will be exported as. This is done inside the definition of the structure, using a backtick character. These annotations are called tags.

> A field declaration may be followed by an optional string literal tag, which becomes an attribute for all the fields in the corresponding field declaration. An empty tag string is equivalent to an absent tag.

These tags are used by various packages, in addition to `encoding/json`.

Encoding structs into JSON

I've already added the json export options, which are recognized by the library `encoding/json`, which we will use for encoding to JSON. Please review the code sample and pay attention to the definition of structures.

The full example

```go
type Petstore struct {
    Name     string `json:"name"`
    Location string `json:"location"`
    Dogs     []*Pet `json:"dogs"`
    Cats     []*Pet `json:"cats"`
}

type Pet struct {
    Name  string `json:"name"`
    Breed string `json:"breed"`
}

```

```go
13  type PetStoreList []*Petstore
14
15  func main() {
16    petstorelist := PetStoreList{}
17    petstore := &Petstore{
18      Name:     "Fuzzy's",
19      Location: "New York, 5th and Broadway",
20    }
21    petstore.Dogs = append(petstore.Dogs,
22      &Pet{
23        Name:  "Whiskers",
24        Breed: "Pomeranian",
25      },
26    )
27    petstore.Dogs = append(petstore.Dogs,
28      &Pet{Name: "Trinity"},
29    )
30    petstorelist = append(petstorelist, petstore)
31
32    jsonString, _ := json.MarshalIndent(petstorelist, "", "\t")
33    fmt.Printf("%s", jsonString)
34  }
```

This will result in a JSON like this:

```json
[
  {
    "name": "Fuzzy's",
    "location": "New York, 5th and Broadway",
    "dogs": [
      {
        "name": "Whiskers",
        "breed": "Pomeranian"
      },
      {
        "name": "Trinity",
        "breed": ""
      }
    ],
    "cats": null
  }
]
```

There are two usual problems which still need solving - the cats array is empty, and some pets don't have a breed. We want to remove this data from JSON. For this, there exists an option in encoding/json, called 'omitempty'. We can update our struct definition to include this option.

```
1  Dogs []*Pet `json:"dogs,omitempty"`
2  Cats []*Pet `json:"cats,omitempty"`
```

Note: this option was added within the double quotes, not after.

These hints specify if the fields should be present in the JSON encoded string if they are empty. Keep in mind, `empty` in this case means `nil`, `0`, `false` and even empty strings, maps and arrays.

It's good to consult the documentation[7] for explanation of additional options, like completely removing a field from encoding.

As you can see below, the empty array `cats`, and missing values for `breed` have been omitted from the resulting JSON result.

The encoded JSON with omitempty fields

```
[
  {
    "name": "Fuzzy's",
    "location": "New York, 5th and Broadway",
    "dogs": [
      {
        "name": "Whiskers",
        "breed": "Pomeranian"
      },
      {
        "name": "Trinity"
      }
    ]
  }
]
```

There are other more specific options available when encoding to JSON. Some field types can't be encoded to JSON, and you can force the encoding to skip some fields completely.

> **Notes:** skipping empty fields is useful mainly for technical reasons. Skipping empty fields reduces the amount of data being sent from APIs, which make a nice speed difference on slower connections. Data is also nicer to inspect visually, as you don't have to skip over empty data structures.

---

[7]https://golang.org/pkg/encoding/json/#Marshal

Decoding JSON contents into structs

Decoding of JSON is fairly straightforward. We will decode the same output we created in the previous step into a Go struct. We will also use the brilliant spew library[8] to "dump" this structure for inspection, so we can see that all the data was decoded without errors.

Example: Decoding JSON contents into structs

```go
type Petstore struct {
  Name     string `json:"name"`
  Location string `json:"location"`
  Dogs     []*Pet `json:"dogs,omitempty"`
  Cats     []*Pet `json:"cats,omitempty"`
}

type Pet struct {
  Name  string `json:"name"`
  Breed string `json:"breed,omitempty"`
}

type PetStoreList []*Petstore

func main() {
  petstorelist := PetStoreList{}

  jsonBlob, err := ioutil.ReadFile("example2.json")
  if err != nil {
    fmt.Printf("Error reading file: %s\n", err)
  }

  err = json.Unmarshal(jsonBlob, &petstorelist)
  if err != nil {
    fmt.Printf("Error decoding json: %s\n", err)
  }

  spew.Dump(petstorelist)
}
```

After executing the example, we can see the imported data:

[8]https://github.com/davecgh/go-spew

Imported data in Go structures

```
(main.PetStoreList) (len=1 cap=4) {
 (*main.Petstore)(0xc820012370)({
  Name: (string) (len=7) "Fuzzy's",
  Location: (string) (len=26) "New York, 5th and Broadway",
  Dogs: ([]*main.Pet) (len=2 cap=4) {
   (*main.Pet)(0xc82000a360)({
    Name: (string) (len=8) "Whiskers",
    Breed: (string) (len=10) "Pomeranian"
   }),
   (*main.Pet)(0xc82000a3c0)({
    Name: (string) (len=7) "Trinity",
    Breed: (string) ""
   })
  },
  Cats: ([]*main.Pet) <nil>
 })
}
```

Nested structures

Contrary to what you might believe, there's no need to declare structures individually, you may also declare and use them in nested form. When dealing with more complex JSON documents, this has a number of advantages.

Let's consider this simple JSON file:

```
1  {
2    "id": 1,
3    "name": "Tit Petric",
4    "address": {
5      "street": "Viska cesta 49c",
6      "zip": "1000",
7      "city": "Ljubljana",
8      "country": "Slovenia"
9    }
10 }
```

The type for this JSON document can be declared as:

```
1  type Person struct {
2    Id      int    `json:"id"`
3    Name    string `json:"name"`
4    Address struct {
5      City    string `json:"city"`
6      Country string `json:"country"`
7    } `json:"address"`
8  }
```

By using this form of declaration, you clearly and explicitly define the hierarchy of the JSON document you are parsing. Keep in mind that the declaration doesn't need to define all fields, but just the ones you will be using. In skipping some fields, you are making the JSON parsing more resilient to changes in the JSON document.

Anonymous structs

The `Address` property in the previous section is called an anonymous struct. You may declare the complete structure as anonymous by explicitly assigning it to a variable:

```
1  person := struct {
2    Id int `json:"id"`
3    Name string `json:"name"`
4    Address struct {
5      City string `json:"city"`
6      Country string `json:"country"`
7    } `json:"address"`
8  }{}
```

And you can use this variable the same you would a `Person{}` from the previous example.

```
1  err = json.Unmarshal(contents, &person)
```

With this, you have mastered the basics of encoding and decoding objects to and from JSON. You will use this knowledge when returning data from your API services, and when consuming data from external sources.

Serving HTTP requests

RESTful API principles dictate the way applications send and retrieve data from API services. We will be implementing our RESTful API service using HTTP as the transportation mechanism. We will take a look at how to provide this with the Go standard library, and then review the usage with an update to go-chi/chi[9].

Setting up a simple web server

Scalability is usually a thing which is achieved by using a specialized load balancer or reverse proxy. Commonly deployed load balancers might be nginx[10], haproxy[11] or traefik[12] (the latter written in Go). These load balancers are used to distribute incoming traffic between two or mode back-end services, as much for redundancy in case of outages, as well as scaling in the case where a single instance wouldn't be able to handle the load.

The approach how to handle back-end services varies. PHP implements a FastCGI interface, with which a web serve communicates with PHP. If you would be using Node, most likely you'd start your own HTTP server, and use the excellent express.js[13] to route request endpoints to individual API implementations. Much like with Node, you would also implement a HTTP server in Go to do the same thing. The principles don't change much.

Let's use the standard library to create a miniature HTTP server, that will print out the current time on any request issued to it.

Example: a minimal HTTP microservice

```go
package main

import (
  "fmt"
  "net/http"
  "time"
)

func requestHandler(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "The current time is: %s\n", time.Now())
```

---

[9]https://github.com/go-chi/chi
[10]https://www.nginx.com
[11]http://www.haproxy.org/
[12]https://traefik.io/
[13]https://expressjs.com/

```
}

func main() {
  fmt.Println("Starting server on port :3000")
  http.HandleFunc("/", requestHandler)
  err := http.ListenAndServe(":3000", nil)
  if err != nil {
    fmt.Println("ListenAndServe:", err)
  }
}
```

Run the server with `go run server1.go` to start the server, press CTRL+C to terminate it. When you start the server, open up another terminal and issue a request to see the output:

Verifying that our microservice works

```
# curl -sS http://localhost:3000/
The current time is: 2018-03-12 17:31:38.890281205 +0000 UTC m=+149.061759235
```

In theory, you would write your individual endpoints as such - microservices. But in practice, it's common to group requests from the same problem domain into a single HTTP service. In order to achieve that, we have to add routing to our application.

Routing logic

When writing an API service, we have to think about routing logic. Routing is basically a way to map the request URL to a handler which provides the response. With this, we will implement different response logic for defined routes. For example:

1. a `/time` entry point which will return the current time,
2. a `/say` entry point which will respond with a Hello based on a parameter "name"

While the standard library doesn't give us a lot of features, it's fairly easy to set up the HTTP handlers to respond to these individual endpoints.

In the `/say` endpoint, we want to read the parameter "name", print a greeting if it's supplied, or just print `Hello ... you.` if the parameter is omitted.

Extended microservice with API parameters

```go
package main

import (
  "fmt"
  "net/http"
  "time"
)

func requestTime(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "The current time is: %s\n", time.Now())
}

func requestSay(w http.ResponseWriter, r *http.Request) {
  val := r.FormValue("name")
  if val != "" {
    fmt.Fprintf(w, "Hello %s!", val)
  } else {
    fmt.Fprintf(w, "Hello ... you.")
  }
}

func main() {
  fmt.Println("Starting server on port :3000")

  http.HandleFunc("/time", requestTime)
  http.HandleFunc("/say", requestSay)
```

```go
  err := http.ListenAndServe(":3000", nil)
  if err != nil {
    fmt.Println("ListenAndServe:", err)
  }
}
```

In `requestSay` we retrieve the parameter "name", using the r.FormValue[14]. If this parameter isn't present, the returned value will be an empty string.

Verifying that our microservice works

```
# curl -sS http://localhost:3000/say
Hello ... you.
# curl -sS http://localhost:3000/say?name=Tit%20Petric
Hello Tit Petric!
```

> *Note*: The HandleFunc is very explicit about the URL. For example, the server will respond to requests that are issued to `/time`, but not `/time/` (trailing slash added). In the same way, it will respond with a 404 error even to a `/` (index) request, because there is no handler matching that URL. If we would declare a handler for "/", it would catch all requests not matched elsewhere.

---

[14]https://golang.org/pkg/net/http/#Request.FormValue

Routing and middleware with go-chi

There are several frameworks available for HTTP routing in Go, adding various features that you'd have to bolt onto the standard library. The most often used feature, in addition to routing, is providing middleware that will add CORS response headers, or print out logging information for issued requests.

The current favourite go-chi/chi[15] provides both a router, and optional handlers in the form of subpackages. The router is expressive and allows for much more flexibility than the standard library - adding support for URL parameters, and grouping of routes.

Add the following imports:

```
"github.com/go-chi/chi"
"github.com/go-chi/chi/middleware"
```

And we can modify our example, to add logging middleware and to create a sub-route with an URL parameter. We want our microservice to respond to requests matching /say/{name}. URL parameters in chi are defined by encapsulating them in curly braces.

A microservice with nested routes and logging

```
func main() {
  fmt.Println("Starting server on port :3000")

  r := chi.NewRouter()
  r.Use(middleware.Logger)

  r.Get("/time", requestTime)
  r.Route("/say", func(r chi.Router) {
    r.Get("/{name}", requestSay)
    r.Get("/", requestSay)
  })

  err := http.ListenAndServe(":3000", r)
  if err != nil {
    fmt.Println("ListenAndServe:", err)
  }
}
```

When it comes to URL parameters in chi, they are mandatory. If we would just create a route with r.Get("/say/{name}",..., the route wouldn't match /say or /say/ requests. We resort to nested routes in this case, where we register handlers for both the parametrized route (/{name}) and the default handler.

---

[15]https://github.com/go-chi/chi

Verifying that our microservice works

```
# curl -sS http://localhost:3000/say
Hello ... you.
# curl -sS http://localhost:3000/say/Tit%20Petric
Hello Tit Petric!
```

As chi contains a number of [middleware implementations](#)[16], we can immediately resort to `middleware.Logger`, which will log the request details to the standard output.

Logging output from our microservice

```
# go run server3.go
Starting server on port :3000
2018/03/12 18:36:03 "GET http://localhost:3000/say HTTP/1.1" from [::1]:33454 - 200 \
15B in 14.7µs
2018/03/12 18:36:09 "GET http://localhost:3000/say/Tit%20Petric HTTP/1.1" from [::1]\
:33456 - 200 18B in 13.499µs
```

In order to make this work, we had to make a small change to our `requestSay` handler. In order to read the URL parameter value, we have to call `chi.URLParam`:

Updated request handler for URL parameters in chi

```go
func requestSay(w http.ResponseWriter, r *http.Request) {
  val := chi.URLParam(r, "name")
  if val != "" {
    fmt.Fprintf(w, "Hello %s!\n", val)
  } else {
    fmt.Fprintf(w, "Hello ... you.\n")
  }
}
```

As you see, the rest of the function stays unchanged. This demonstrates a good characteristic of chi itself, the fact that it's compatible with the Go standard library. This means, if you find yourself in a need to migrate from one to the other, you can do this with only minor refactoring.

---

[16]https://github.com/go-chi/chi#middlewares

Advanced middleware - CORS

A common requirement of API implementations is to provide CORS headers in the HTTP response. We can use special CORS auxiliary middleware[17] from the go-chi project, in order to add them to our microservice.

Add the following import:

```
"github.com/go-chi/cors"
```

After creating the router, you can use the CORS middleware like so:

Example CORS middleware

```
cors := cors.New(cors.Options{
  AllowedOrigins:   []string{"*"},
  AllowedMethods:   []string{"GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS"},
  AllowedHeaders:   []string{"Accept", "Authorization", "Content-Type", "X-CSRF-Toke\
n"},
  AllowCredentials: true,
  MaxAge:           300, // Maximum value not ignored by any of major browsers
})
r.Use(cors.Handler)
```

After restarting the server, we can verify that the response includes the expected headers:

Verifying that the CORS middleware works

```
# telnet localhost 3000
Trying ::1...
Connected to localhost.
Escape character is '^]'.
GET /say/Tit HTTP/1.0
Origin: my.dev.hostname.local

HTTP/1.0 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: my.dev.hostname.local
Vary: Origin
Date: Mon, 12 Mar 2018 18:53:20 GMT
Content-Length: 11
Content-Type: text/plain; charset=utf-8
```

---

[17]https://github.com/go-chi/cors

```
Hello Tit!
Connection closed by foreign host.
```

CORS headers are added to every response, for requests that include the required `Origin` header. There are other auxiliary middleware available, handling JWT authentication for example[18]. You should familiarize yourself with the complete list[19], as it might save you a lot of development time.

---

[18]https://github.com/go-chi/jwtauth
[19]https://github.com/go-chi/chi#auxiliary-middlewares--packages

JWT authentication middleware

A common use case for APIs is to provide authentication middleware, which will let a client make authorized requests to your APIs. Generally, your client performs some sort of authentication, and a session token is issued. Recently, JWT (JSON Web Tokens) are a popular method of providing a session token with an expire time, which doesn't require some sort of storage to perform validation.

So what is JWT exactly?

> JSON Web Token (JWT) is an open standard (RFC 7519[20]) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

A token is constructed from three pieces of information:

1. the header, specifying the signature algorithm,
2. the claims, which may also include an expire time,
3. the signature based on header settings

Using these three pieces, it's possible for your API service to re-create the signature based on data in the header and in the claims pieces of the JWT. This is possible because this signature is created using a secret signing key, which is only known to the server.

Creating a signing key for JWT

```
jwtauth.New("HS256", []byte("K8UeMDPyb9AwFkzS"), nil)
```

In case your signing key is compromised, you should immediately change it. This will *invalidate* all existing issued JWT's, forcing your clients to re-authenticate to your service.

Claims

JWT claims are the facility with which you state that the client using your API services might be `"user_id": "1337"` or similar. Think of it as a `map[string]interface{}`, with some casting required. When your client authenticates against your API, by performing a login with a client ID and Secret, you issue a new JWT that doesn't require you re-authenticating the client in the database, until the token expires.

It's a good thing to issue a debug token from your application and write it to the log. This way you'll have a valid token to perform testing against your application.

---

[20]https://tools.ietf.org/html/rfc7519

Encoding a testing token

```go
type JWT struct {
  tokenClaim string
  tokenAuth  *jwtauth.JWTAuth
}

func (JWT) new() *JWT {
  jwt := &JWT{
    tokenClaim: "user_id",
    tokenAuth:  jwtauth.New("HS256", []byte("K8UeMDPyb9AwFkzS"), nil),
  }
  log.Println("DEBUG JWT:", jwt.Encode("1"))
  return jwt
}

func (jwt *JWT) Encode(id string) string {
  claims := jwtauth.Claims{}.
    Set(jwt.tokenClaim, id).
    SetExpiryIn(30 * time.Second).
    SetIssuedNow()
  _, tokenString, _ := jwt.tokenAuth.Encode(claims)
  return tokenString
}
```

When you'll create a new JWT object with `JWT{}.new()`, a debug token will be printed to the log.

```
2018/04/19 11:35:18 DEBUG JWT: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoi\
MSJ9.ZEBtFVPPLaT1YxsNpIzVGSnM4Vo7ZrEvp77jKgfN66s
```

You can pass this token via URL query parameter to test GET requests, or use it in your test suite in order to issue more complex API requests, passing it with the `Authorization` header, or as a Cookie.

> **Note**: Be very careful that you generate your tokens with an expire time. If you don't do this, they will be valid until you change your signing key. Another option is to invalidate individual tokens on the server side, which requires some sort of database for logging and revoking them. For example: instead of issuing actual user IDs as the example shows, you'd create a session ID, which you can additionally validate for expiry/logout.
>
> The example already sets the needed parameters in order so you'll validate tokens with an expiry time. If you issue requests to a protected endpoint after the token expires, an error message is returned, hinting you to re-authenticate on some API call.

Protecting API access with JWT

Each request that comes to the API can include a JWT Verifier. This works similarly to CORS headers - it tests the presence of a JWT in either the HTTP query string, cookie or Authorization HTTP header. The result of the verifier are new context variables for the JWT and a possible parsing errors. The verifier doesn't break out of the request in case of a missing or invalid token, it's job is to provide this information to the Authenticator.

The `jwtauth` package provides a default verifier, which you can use out of the gate. Let's add an utility function to our JWT struct that will return it:

```
func (jwt *JWT) Verifier() func(http.Handler) http.Handler {
  return jwtauth.Verifier(jwt.tokenAuth)
}
```

We include this verifier on every request. This makes it possible to pass tokens to API endpoints which don't explicitly require an authenticated user. In such cases, you can retrieve and handle any claims, still providing a valid response in cases where a JWT is not present.

Example setting up a JWT verifier

```
login := JWT{}.new()

mux := chi.NewRouter()
mux.Use(cors.Handler)
mux.Use(middleware.Logger)
mux.Use(login.Verifier())
```

Instead of using `mux.Route` as we did before, we should use [mux.Group()](https://godoc.org/github.com/go-chi/chi#Mux.Group)[21] to split requests into authenticated and public endpoints. Using Group() allows us to add new handlers, to the existing global ones. This way we avoid certain API call prefixes like `/api/private/*`.

> Group creates a new inline-Mux with a fresh middleware stack. It's useful for a group of handlers along the same routing path that use an additional set of middleware.

---

[21]https://godoc.org/github.com/go-chi/chi#Mux.Group

Example of using mux.Group

```go
// Protected API endpoints
mux.Group(func(mux chi.Router) {
  // Error out on invalid/empty JWT here
  mux.Use(login.Authenticator())
  {
    mux.Get("/time", requestTime)
    mux.Route("/say", func(mux chi.Router) {
      mux.Get("/{name}", requestSay)
      mux.Get("/", requestSay)
    })
  }
})

// Public API endpoints
mux.Group(func(mux chi.Router) {
  // Print info about claim
  mux.Get("/api/info", func(w http.ResponseWriter, r *http.Request) {
    owner := login.Decode(r)
    resputil.JSON(w, owner, errors.New("Not logged in"))
  })
})
```

The API endpoints /time and /say will now be accessible only with a valid token. For example, if we issue a request against this endpoint with an expired token, we might end up with something like:

```
{
  "error": {
    "message": "Error validating JWT: jwtauth: token is expired"
  }
}
```

But if we issue a request against /info, we can end up with:

Example response with valid JWT

```
{
  "response": "1"
}
```

Or if the token already expired:

Example response when JWT expired

```
{
  "error": {
    "message": "Not logged in"
  }
}
```

This is because we implemented full validation in Decode, or more accurately, our Authenticate function where we provide a valid claim or an error:

```go
func (jwt *JWT) Decode(r *http.Request) string {
  val, _ := jwt.Authenticate(r)
  return val
}

func (jwt *JWT) Authenticate(r *http.Request) (string, error) {
  token, claims, err := jwtauth.FromContext(r.Context())
  if err != nil || token == nil {
    return "", errors.Wrap(err, "Empty or invalid JWT")
  }
  if !token.Valid {
    return "", errors.New("Invalid JWT")
  }
  return claims[jwt.tokenClaim].(string), nil
}
```

We use the same Authenticate function to provide the Authenticator() middleware that enforces JWT usage on private API endpoints. The error in Decode() is ignored, as the called function already enforces an empty string return. The Authenticator middleware however, returns the error in full:

```go
func (jwt *JWT) Authenticator() func(http.Handler) http.Handler {
  return func(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
      _, err := jwt.Authenticate(r)
      if err != nil {
        resputil.JSON(w, err)
        return
      }
      next.ServeHTTP(w, r)
    })
  }
}
```

The full code sample for the above JWT authenticated microservice is available on GitHub[22]. Feel free to check it out and take it for a spin.

---

[22]https://github.com/titpetric/books/tree/master/api-foundations/chapter4b-jwt

Parallel fetching of data

When you're writing a service, you usually don't get the benefit of starting exactly from scratch. Mostly, you build upon existing work of those before you, those next to you and public services which provide value from the start.

There are a few considerations to make, when you're consuming external services. Is the service slow? Is the service reliable? Is the connection to the service reliable? What is the worst case scenario, if the service goes offline?

A simple API service

We will layout a simple API service. We will create the following API endpoints:

- `/fullname`
- `/firstname`
- `/lastname`

All three API calls will accept parameters "firstname" and "lastname". The endpoint `/fullname` will query individual endpoints `/firstname` and `/lastname`, getting and decoding JSON data, and creating a JSON response with both. We will not use advanced routing from the previous chapter, just to make the example more concise.

For those of you a bit familiar with PHP, we will define a function named `json_encode`. The function uses the trick from the chapter of data structures to accept any argument and passes the data to json.Marshal to return a string. We don't package this function, but include it directly into our program. The name of the function starts with a lower-case, which only makes it visible in the package in which it's declared (main).

We are also using the `foundations/bootstrap` package, with the method `Now` (now.go), declared in the "Organizing your code" chapter.

Our utility function: json_encode

```
1  func json_encode(r interface{}) string {
2      jsonString, _ := json.MarshalIndent(r, "", "\t")
3      return string(jsonString[:])
4  }
```

With these functions we can implement individual firstname and lastname endpoints. I've added a 200ms and 300ms delay to each one of them respectively, and print out when the function execution starts, and when it finishes with a call to fmt.Printf.

Basic API endpoint: getFirstname

```go
type FirstName struct {
  Firstname string `json:"firstname"`
}

func getFirstName(w http.ResponseWriter, r *http.Request) {
  time.Sleep(200 * time.Millisecond)
  value := r.FormValue("firstname")
  response := FirstName{Firstname: value}
  response_json := json_encode(response)
  fmt.Fprintf(w, response_json)
}
```

This is a synchronous (blocking) way to request the resources:

Basic API endpoint setup

```go
type FullName struct {
  Firstname string `json:"firstname"`
  Lastname  string `json:"lastname"`
}

func getFullName(w http.ResponseWriter, r *http.Request) {
  bootstrap.StartTime = 0

  firstname_value := r.FormValue("firstname")
  lastname_value := r.FormValue("lastname")

  var firstname FirstName
  var lastname LastName
  var fullname FullName

  data := url.Values{}
  data.Add("firstname", firstname_value)
  data.Add("lastname", lastname_value)

  // fetch firstname

  fn_url := "http://localhost/firstname?" + data.Encode()
  fmt.Printf("[%.4f] Fetching url: %s\n", bootstrap.Now(), fn_url)
  fn_response, _ := http.Get(fn_url)

  fn_contents, _ := ioutil.ReadAll(fn_response.Body)
```

```
27
28     _ = json.Unmarshal(fn_contents, &firstname)
29     fullname.Firstname = firstname.Firstname
30
31     // fetch lastname
32
33     ln_url := "http://localhost/lastname?" + data.Encode()
34     fmt.Printf("[%.4f] Fetching url: %s\n", bootstrap.Now(), ln_url)
35     ln_response, _ := http.Get(ln_url)
36
37     ln_contents, _ := ioutil.ReadAll(ln_response.Body)
38
39     fmt.Printf("[%.4f] Done fetching\n", bootstrap.Now())
40
41     _ = json.Unmarshal(ln_contents, &lastname)
42     fullname.Lastname = lastname.Lastname
43
44     // return fullname response
45     response_json := json_encode(fullname)
46     fmt.Fprintf(w, response_json)
47     fmt.Printf("[%.4f] Done with response: %#v\n", bootstrap.Now(), fullname)
48 }
```

Sorry for the bit of a long code snippet - it does just this:

1. reads firstname and lastname from the query string,
2. creates a new query string for sub-requests,
3. fetches the firstname,
4. fetches the lastname,
5. constructs a full name response

With the logging and the utility method we have in place, we can time the API request. When we request /fullname?firstname=Tit&lastname=Petric, we will get something like this.

```
Starting server on port :80
[0.0000] Fetching url: http://localhost/firstname?firstname=Tit&lastname=Petric
[202.3157] Fetching url: http://localhost/lastname?firstname=Tit&lastname=Petric
[503.8228] Done fetching
[504.1355] Done with response: main.FullName{Firstname:"Tit", Lastname:"Petric"}
```

Making it parallel

So, by default, the calls are blocking, waiting for the previous to finish. The way to make them non-blocking is to wrap the request in a goroutine. We will also need a way to retrieve this data from the main thread, so we will create a `chan` (short for channel). This is a Go data type, which creates a "pipe" between goroutines in order to exchange data.

Goroutine which requests the firstname API

```
1   fn_chan := make(chan []byte, 1)
2   go func() {
3       fn_url := "http://localhost/firstname?" + data.Encode()
4       fmt.Printf("[%.4f] Fetching url: %s\n", bootstrap.Now(), fn_url)
5       fn_response, _ := http.Get(fn_url)
6       contents, _ := ioutil.ReadAll(fn_response.Body)
7       fn_chan <- contents
8   }()
```

When we write to a channel from a goroutine, we must also read from this channel. When we read from it the process again becomes blocking, so you should read after you've run all your goroutines.

Reading channels and processing the data

```
1   fn_contents := <-fn_chan
2   _ = json.Unmarshal(fn_contents, &firstname)
3   fullname.Firstname = firstname.Firstname
4
5   ln_contents := <-ln_chan
6   _ = json.Unmarshal(ln_contents, &lastname)
7   fullname.Lastname = lastname.Lastname
```

After running a server with requests wrapped in goroutines, the log is something like this:

```
[0.0000] Fetching url: http://localhost/lastname?firstname=Tit&lastname=Petric
[0.0444] Fetching url: http://localhost/firstname?firstname=Tit&lastname=Petric
[3.1770] Request with firstname
[5.3557] Request with lastname
[205.9363] Response with lastname
[303.9368] Response with firstname
[306.6057] Done fetching
[306.9421] Done with response: main.FullName{Firstname:"Tit", Lastname:"Petric"}
```

As we see, the first request and the second request start less than a millisecond apart. The main thread waits for the first request, and then for the second if it's still not done.

Some tips

There are a few tips I can share, so you might avoid common problems. I violated some of them in the examples above, but keep in mind - the examples read more like a guide to aid you in transferring some of your existing concepts of API development from PHP or other programming languages. They are not exactly best practice in various meanings. They demonstrate a concept by skipping out on some things like error handling, code organization, and proper execution flow - those things are up to the reader.

That being said, advice is always good, and I'll try to explain some in individual chapters.

Do hard processing in goroutines

As we saw with http.Get, by default concepts in Go can be blocking. Go itself uses non-blocking I/O to avoid the system blocking the thread, so different goroutines can be run during the time while the first is waiting for the I/O operation to complete. This means that you have to use goroutines if you want to optimize your own execution, like it was shown in the examples above.

I violate this rule a bit by performing json.Unmarshal in the main goroutine. While it's generally fast, imagine what would happen if you had to unserialize several 100MB worth of JSON? It's important to think about which operations would benefit from a goroutine. Some common examples:

1. Fetching data from multiple external API sources (multiple HTTP requests),
2. Independent processing (fetching contacts and calendar data - distinct sources),
3. Composition (Example by Twitter: Trends, Feed, Who to Follow)

The examples are plentiful when you're looking at web pages, and which APIs they provide. While some things need to be done sequentially, there are always opportunities to group this data into one large batch instead of many small, sequential ones.

A typical thing which might happen in an application is a fetching of a list, for which each list item needs some additional processing. Some program is going through every item in the twitter feed, detecting links and embedding images. Some program is going through a list of search results, pairing them with ratings data from TripAdvisor or some other site.

The next time you're doing an `O(N)` operation on a list of N items, perhaps consider if it's possible to do them in parallel. Having `N*O(1)` may mean an improvement of many times - for external data it shortens the time to the longest request, processing and calculation scales over your available CPUs, which might mean an improvement of 4x on a very common quad core CPU, if the workload is finely balanced.

> The `net/http` package actually uses goroutines for individual connections, so you might just be fine
> if you're processing 100MB of JSON. Understand your environment.

Create one or many packages for your endpoints

Think of packages as namespaces in PHP. Or just classes, because in many ways they behave similarly. A "Hello world" service would look something like this:

Example apiservice package, defining HelloWorld

```
1  package apiservice
2
3  import "fmt"
4
5  func HelloWorld() {
6    fmt.Printf("Hello world!\n")
7  }
```

Where you keep your `main.go` file, create the folder `apiservice/`, and in it save `apiservice.go` file with the above contents; Remember the data structures chapter - the upper case letter of a function means that it's public and you can use it from other packages. Using the package then is as simple as this:

Example of using the apiservice package

```
1  package main
2
3  import "foundations/apiservice"
4
5  func main() {
6    apiservice.HelloWorld()
7  }
```

We import the package with `foundations/` prefix, which is the name of our project. This is usually the folder name where your main.go file is located. We will return to this subject with a practical example in a later chapter.

Measure first

If you need to know how much time something lasts, you should measure it first. Ask yourself: how will you know which one of your functions is the slowest? How will you know if some times a function will take 10 seconds, while it usually completes in 10 milliseconds?

I have the very strong opinion that, especially for external resources, you have to measure and log everything in various level of detail, so you can act on what your data is telling you. If you measure how much time an SQL query is taking to complete, you have to log the query itself so you know where to look to fix it. If you have a generic http request object, be sure to log the URL along with the slow response time. The problems you find may not be the problems that you can fix, but without this data, it's mostly just stumbling in the dark.

Reverse proxy

Have a solid reverse proxy in front of your go application. Nginx for example can handle multiple upstreams - this will make your application easier to scale if needed. It also handles upstream fail-over, if you're upgrading your cluster one app at a time, or if for some reason, one of the back-ends crashes and is unavailable.

Trust me, some level of redundancy is good - even if you're starting development on a single machine, explore what your options will be in the future and start as close to possible to the final structure of your services. Even if you're using nginx to forward requests to a service in go on the same machine, the value becomes apparent later, when you can add another server and just change a few lines of configuration in nginx. Scaling in such a case is almost free.

Using external services (Redis)

Redis is a popular in memory data structure storage. It's common use is to provide a caching mechanism that goes beyond a simple key/value store. If you followed the instructions in the "Setting up the environment" chapter, you already have a Redis node running.

Client library

There are several client libraries available. The library "Redigo" seems to be the most supported and provides all the interfaces we will need for our examples. To install it, run the following command:

```
$ gvt fetch github.com/garyburd/redigo/redis
```

A very simple Redis connection can be made by calling "redis.Dial". Save this file under `foundations/bootstrap` as `redis.go` to provide a simple interface.

redis.go bootstrap package

```
 1  package bootstrap
 2
 3  import "time"
 4  import "github.com/garyburd/redigo/redis"
 5
 6  var (
 7    connectTimeout = redis.DialConnectTimeout(time.Second)
 8    readTimeout    = redis.DialReadTimeout(time.Second)
 9    writeTimeout   = redis.DialWriteTimeout(time.Second)
10  )
11
12  func GetRedis() (redis.Conn, error) {
13    return redis.Dial("tcp", "redis:6379", connectTimeout, readTimeout, writeTimeout)
14  }
```

You now have everything for our first example.

Talking to a Redis instance

Redis provides several commands that implement various data structure storage and retrieval. Most of the commands are very fast, and return data faster than a millisecond.

A simple Redis command - PING

```
1   package main
2
3   import "foundations/bootstrap"
4   import "log"
5   import "fmt"
6
7   func main() {
8     redis, err := bootstrap.GetRedis()
9     if err != nil {
10       log.Fatal("Fatal error: ", err)
11     }
12     fmt.Printf("[%.4f] Starting\n", bootstrap.Now())
13     pong, err := redis.Do("PING")
14     fmt.Printf("[%.4f] Response %s, err %#v\n", bootstrap.Now(), pong, err)
15   }
```

The simplest of all Redis commands is "PING", which just returns "PONG" in less than a millisecond.

Example run of PING

```
== go run redis1.go ==
[0.0000] Starting
[0.4575] Response PONG, err <nil>
```

As we did in the previous chapter using HTTP APIs, we would like to fetch some data from Redis. Redis provides a command DEBUG SLEEP which takes the number of seconds to wait, before returning an OK. We will use it to provide a delay of 100ms and 200ms for two consecutive Redis calls.

Two long-running commands

```
1     fmt.Printf("[%.4f] Starting\n", bootstrap.Now())
2
3     sleep1, err := redis.Do("DEBUG", "SLEEP", "0.1")
4     fmt.Printf("[%.4f] End Sleep 100ms, result %s err %v\n",
5       bootstrap.Now(), sleep1, err)
6
7     sleep2, err := redis.Do("DEBUG", "SLEEP", "0.2")
8     fmt.Printf("[%.4f] End Sleep 200ms, result %s err %v\n",
9       bootstrap.Now(), sleep2, err)
```

As we can see here, the first one sleeps for 100ms, and the second for 200ms.

Output of the example commands

```
== go run redis2.go ==
[0.0000] Starting
[101.0354] End Sleep 100ms, result OK err <nil>
[301.9424] End Sleep 200ms, result OK err <nil>
```

Redis is single threaded, meaning that there will be no advantage in using goroutines - two commands are never run in parallel on one server, and while one command is executing, the next one is waiting for the previous one to finish. This might sound bad, but keep in mind, Redis can handle millions of very small operations very fast. A connection to a Redis server takes several hundred magnitudes more time than a simple GET.

Worst case scenario

If one command will run for a longer amount of time, it would mean that all other commands are waiting for it to finish. If one command runs for a 100ms, you basically stopped all your clients for this amount of time.

Redis provides a `--intrinsic-latency` option to it's redis-cli program, which you can use to analyze live traffic to your Redis instance. For example, a production cluster I'm running, I've ran the following command to get the maximum latency within a 100 second period:

```
$ redis-cli --intrinsic-latency 100
...
Max latency so far: 254 microseconds.

7736079 total runs (avg latency: 12.9264 microseconds / 129264.45 nanoseconds per ru\
n).
Worst run took 20x longer than the average latency.
```

As you can see, the latency is ridiculously low. At the worst case, I'm handling ∼3.800 req/s. At the average case, I'm handling ∼83.000 req/s. But, let's say that it's possible that you're hitting a CPU limit. Looking at Redis SLOWLOG might give us an idea of the true worst case.

```
  3) 1) (integer) 107
     2) (integer) 1459739957
     3) (integer) 21958
     4) 1) "GET"
        2) "api:schedule:list:TVS1:2016-04-07"
```

This is one of the worst-case slow queries on my system, and it's clocking in at 21.958 microseconds (about 22 milliseconds, or 0.022 seconds). If it was common (it's not), it would reduce the request rate to about 45 requests per second. It took 3 full months to get a SLOWLOG with 110 entries, so I have quite some breathing room.

Let's scale it

Scaling Redis is not uncommon. People do it for various reasons - because Redis is single threaded it can't use more than 1 CPU core. Running several redis instances on the same server is very logical in this sense. We also do it to provide fail-over mechanisms in our cache cluster - if one host goes offline, the workload balances out over the remaining cluster.

To scale to many Redis servers, your application will need to maintain a connection pool. Individual queries will run on different servers, so the response of one will not wait for the previous one. We will create and use two redis instances, named `redis1` and `redis2`. Use this bash script to run them with docker:

```bash
#!/bin/bash
NAMES="redis1 redis2"
DOCKERFILE="redis"
for NAME in $NAMES; do
  docker rm -f $NAME
  docker run --restart=always -h $NAME --name $NAME --net=party -d $DOCKERFILE
done
```

Connection pool

Let's make a connection pool with our long-running command example. We want to create two goroutines which use different connections, so the requests can run in parallel. Keep in mind, that the true value of such a connection pool is when you run an API service, and not when you run a simple command line program. In the command line, we will need to warm up the connection pool, so we can skip the latency penalty of establishing a connection.

There exists a battle-tested implementation of a connection pool - as a package of the Vitess project by YouTube. I implemented two functions - one for establishing the pool, and another to run individual Redis commands on a connection from this pool. Install the following packages:

```
$ gvt fetch github.com/youtube/vitess/go/pools
$ gvt fetch golang.org/x/net/context
```

We will create our pool with similar options, to how we create a stand-alone Redis connection. The function getServerName returns "redis1:6379" or "redis2:6379" in sequence.

Creating a Redis connection pool

```go
func RedigoPool() *pools.ResourcePool {
  if !hasPool {
    capacity := 2    // hold two connections
    maxCapacity := 4 // hold up to 4 connections
    idleTimeout := time.Minute
    pool = pools.NewResourcePool(func() (pools.Resource, error) {
      serverName := getServerName()
      c, err := redis.Dial("tcp", serverName, connectTimeout, readTimeout, writeTime\
out)
      fmt.Println("New redis connection: " + serverName)
      return ResourceConn{c}, err
    }, capacity, maxCapacity, idleTimeout)
  }
  return pool
}
```

When you create a pool, be sure to also use `pool.Close()` for when you shut down the application.

```go
redis := bootstrap.RedigoPool()
defer redis.Close()
```

We will create an utility function `RedigoDo` with the same interface as `redis.Do`. I made this choice, because as the pool works, the connection is retrieved from the pool with `pool.Get`, you call your command on the active connection, and then you have to return the connection back to the pool with `pool.Put`, so it's available for further use. If we could have avoided the `pool.Put` command, we wouldn't need to make this utility function.

Executing a Redis command on a pooled connection

```go
func RedigoDo(commandName string, params ...interface{}) (interface{}, error) {
  ctx := context.TODO()
  r, err := pool.Get(ctx)
  if err != nil {
    return "", err
  }
  defer pool.Put(r)

  c := r.(ResourceConn)
  return c.Do(commandName, params...)
}
```

This is all we need to do to create and use many connections from a connection pool. Let's use it in some goroutines and check to see that is works as it should.

Running a Redis command from a goroutine

```
1   go func() {
2       fmt.Printf("[%.4f] Run sleep 100ms\n", bootstrap.Now())
3       sleep1, err := bootstrap.RedigoDo("DEBUG", "SLEEP", "0.1")
4       if err != nil {
5           sleep1 = "ERROR"
6       }
7       sleep1_chan <- sleep1.(string)
8   }()
```

We use channels as we have learned in the previous chapter. Reading from a channel will wait until there is data available. Since we're using two servers, the commands are run in parallel - a sleep of 100ms and a sleep of 200ms will finish just after 200ms.

```
New redis connection: redis1:6379
New redis connection: redis2:6379
[0.0000] Start
[0.1724] Run sleep 200ms
DEBUG: bootstrap.ResourceConn{Conn:(*redis.conn)(0xc82007eb40)}
[0.9338] Run sleep 100ms
DEBUG: bootstrap.ResourceConn{Conn:(*redis.conn)(0xc82007eaa0)}
[102.4917] End Sleep 100ms, result OK
[204.8770] End Sleep 200ms, result OK
```

Great success! As you can see, two redis connections are started in the pool. Two sleep commands are run in goroutines, which are executed on different connections to different redis servers. Individual 100ms and 200ms requests complete in the time of the longest request.

> When you'll be running code in production, you may need to restructure it in a way, that will give you an actual connection, which you can use in your single goroutine. This is very important if you're using Redis in a transactional way with MULTI. If you're using it for simple GET or more complex, but still single-query statements, then you're fine with the example provided.

Using external services (MySQL)

I hope you're familiar with MySQL. We will create a database connection to a MySQL server, and much like in previous examples, try to make a connection pool and run multiple queries in parallel. We will use the available "database/sql" package for access to our database.

Quick start

We will need a MySQL driver. To install it run the following:

```
$ gvt fetch github.com/go-sql-driver/mysql
```

And to use it, add the following import line to your main.go file:

```
import _ "github.com/go-sql-driver/mysql"
```

The underscore after the import line imports the package only for it's side effects. The MySQL driver adds a driver implementation to the base sql package. This way the driver for MySQL gets added, but you don't actually have anything to use from the mysql package.

Opening a database connection is as simple as:

```
1  db, err := sql.Open("mysql", "api:api@tcp(db1:3306)/api")
```

And with that, you can start using the "db" object.

Goodbye simplicity

If you're familiar with PHP, and have a reasonable implementation of a database class, your query might look as simple as this:

```
1  $db->query("select * from table
2      where field1=? or field2=?",
3      $field1, $field2)
```

If you're a bit closer to Ruby DBI:

```
1  sth = dbh.prepare("SELECT * FROM EMPLOYEE
2                          WHERE INCOME > ?")
```

The one thing which these two languages (and other web-targeted languages) have in common is how simple it is to fetch a row from the database and read each rows columns. With PHP you can fetch it into an array (hash or numeric index array) and Ruby also has the same thing with the fetch method, for over a decade, from what I can glance in the documentation.

This is the way you run a Query in go:

```
1  stmt, err := conn.Query("show databases;")
```

And the not-so-simple part:

Fetching a result set from the database

```
1  func showDatabases(conn *sql.DB, sql string) error {
2    stmt, err := conn.Query(sql)
3    if err != nil {
4      return err
5    }
6    defer stmt.Close()
7    for stmt.Next() {
8      var name string
9      if err := stmt.Scan(&name); err != nil {
10       log.Fatal(err)
11     }
12     fmt.Printf("Database: %s\n", name)
13   }
14   return nil
15 }
```

You'll notice in the very specific example, that I'm dealing with only one column, and I'm fetching the data explicitly by columns. The `database/sql` package doesn't provide me with a simple Fetch or FetchAll method, and putting the resulting rows into a map/array comes with some processing as to how many columns your results have, and the obvious caveat - what are the column types.

Simplicity redux

While I struggled if I should write an utility function that uses reflection to put all the returned data from the database into a struct, I realized that I'm probably not the first person to identify the same problem. It was amusing finding reports like these on Stack Overflow:

> # if the easy way is to manually bind columns to struct fields I'm wondering what's the hard way
>
> – Anthony Hunt Sep 6 '15 at 20:36

In the same Stack Overflow thread a person suggested to use `jmoiron/sqlx`. The library provides a much needed abstraction of the "low level" sql package. Let's install it now:

```
$ gvt fetch github.com/jmoiron/sqlx
```

And now let's quickly adapt our example:

Fetching a result set from the database, redux

```go
 1  type Database struct {
 2    Name string `db:"Database"`
 3  }
 4
 5  func main() {
 6    db, err := sqlx.Open("mysql", "api:api@tcp(db1:3306)/api")
 7    if err != nil {
 8      log.Fatal("Error when connecting: ", err)
 9    }
10    databases := []Database{}
11    err = db.Select(&databases, "show databases")
12    if err != nil {
13      log.Fatal("Error in query: ", err)
14    }
15    spew.Dump(databases)
16  }
```

You may notice, how the example includes basically the whole application. The resulting structure returned into the "databases" variable is like follows:

```
([]main.Database) (len=2 cap=2) {
 (main.Database) {
  Name: (string) (len=18) "information_schema"
 },
 (main.Database) {
  Name: (string) (len=3) "api"
 }
}
```

In other words - a traversable, typed result set, created by one line of code. *Simplicity.*

Connection pool

Like in the previous example, we want to create a connection pool that will hold one or many database connections. We will again be using Vitess for this. As MySQL is a threaded server, we can connect to the same server twice, and the queries on each connection will run in parallel.

We will create a similar pool to what we had with Redis, only we will provide two functions in addition to the connection pool one. We will define `SqlxGetConnection` and `SqlxReleaseConnection`. This way we can get only one connection inside a goroutine, and re-use it for many queries (and even transactions!).

Setting up a connection pool is simple:

```
1  pool := bootstrap.SqlxConnectionPool()
2  defer pool.Close()
```

We should warm up our connection pool for our test, just to get the timings right. Connecting to a database takes some time, as you would expect.

```
1  // warm up the connection pool
2  for i:=0; i<5; i++ {
3    db, _ := bootstrap.SqlxGetConnection()
4    db.Ping()
5    bootstrap.SqlxReleaseConnection(db)
6  }
```

Let's test the multi-threaded nature of MySQL by issuing a SLEEP SQL query;

SLEEP(duration)

> Sleeps (pauses) for the number of seconds given by the duration argument, then returns 0. The
> duration may have a fractional part

By issuing SLEEP(0.1) and SLEEP(0.2) we can replicate the same behaviour we used as an example
in previous chapters. In a goroutine we will issue an SQL query like this:

Running an SQL Query from a goroutine

```go
go func() {
    db, err := bootstrap.SqlxGetConnection()
    if err != nil {
        log.Fatal("Error when connecting: ", err)
    }
    defer bootstrap.SqlxReleaseConnection(db)

    fmt.Printf("[%.4f] Run sleep 100ms\n", bootstrap.Now())

    fromSleep := SleepResult{}
    err = db.Get(&fromSleep, "select sleep(0.1) as sleepfor")
    if err != nil {
        sleep1_chan <- "ERROR"
        return
    }
    sleep1_chan <- fromSleep.Result
}()
```

As we're doing everything right, we get this pretty, expected output:

```
New mysql connection: 1
New mysql connection: 2
[0.0000] Start
[0.1714] Run sleep 200ms
[0.3037] Run sleep 100ms
[101.0024] End Sleep 100ms, result 0
[201.1951] End Sleep 200ms, result 0
```

In MySQL, 1 connection represents 1 thread. Depending on the number of CPUs you have
available, only a few threads are needed to completely saturate MySQL with an SQL workload. The
recommended amount of pool connections is about 2-3 times the CPUs available on a MySQL server.
If you have an 8 core machine, 24 pooled connections is more than enough. The more connections
you add, the more RAM you use - but you don't increase performance.

Scaling MySQL beyond MySQL

I'm a big fan of planning for disaster case scenarios - and with MySQL I think I've faced more than many. It made me very proficient at using SQL indexes, as well as pay attention to common issues that forced us to scale to many instances and shard our data. This is what the YouTube project Vitess is trying to solve and it deserves an honourable mention.

Scaling connections

One MySQL connection takes about 2-3 Megabytes RAM. A part of Vitess is a program called `vttablet`, which pools these connections, and can only hold a few connections between it and MySQL. This is a good way to save some memory in MySQL. The application also provides statistics and monitoring to help with operations.

Scaling servers

At one point, you'll be forced to create one or many replicas, so you can scale your read volume, or to split your read volume from your write volume. There are other complexities here, like promoting a slave to a master, sharding your data, and failing over queries in case of failure. Vitess provides a `vtgate` application that handles this logic from configuration, so your application logic can stay simple.

Stopping bad queries

Vitess looks at the queries that are going through it, to find and prevent common problems. A very common example of bad practice are queries that are performing a full table scan - Vitess will add a LIMIT clause to queries that it detects as bad. If some query is causing a lot of problems Vitess will blacklist it, so your site can keep running. Hopefully, the query which was blacklisted wasn't important.

Statistics

Vitess provides detailed statistics that can aid you in pinpointing database performance problems. Statistics are a pain point in MySQL itself - running a query log in production is very expensive, and there are next to no runtime statistics that are provided by MySQL itself.

As we struggled with this problem for over a decade, we switched our servers out for `Percona` fork of MySQL years ago. If you're not prepared to use Vitess yet, the Percona fork is a stable, giving measurable benefits in performance and diagnostics.

In fact, running a Percona build of MySQL is as simple as this:

```bash
1   #!/bin/bash
2   NAME="db1"
3   DOCKERFILE="percona:latest"
4
5   if [ ! -d "/src/$NAME/data" ]; then
6       mkdir -p /src/$NAME/data
7   fi
8
9   docker rm -f $NAME
10  docker run --restart=always \
11      -h $NAME \
12      --name $NAME \
13      --net=party \
14      -v /src/$NAME/conf/conf.d:/etc/mysql/conf.d \
15      -v /src/$NAME/data:/var/lib/mysql \
16      -e MYSQL_ROOT_PASSWORD=$PASSWORD \
17      -d $DOCKERFILE
```

It should work out better than the default MySQL server. The Percona guys write a blog, which is full of nice articles on performance optimization - anything from setting the correct indexes, to performance of UUIDs. You should add it to the list of mandatory reading.

Test driven API development

As I suggested in the previous chapter, the best way to structure your API is to contain it within a package. We will not be very complicated in this, so we will just create a package "api" and create our implementation there. We will nest our functions under the Registry struct, so you will use an api.Registry value to implement your API.

Creating a simple API

Our goal is to create a simple API which will perform the following:

- `/get` with parameter key - will return value from redis
- `/set` with parameters key, value - will set a key/value pair
- `/getAll` retrieve all redis keys

But first, we will make a local implementation of all the required functions. These functions will use our existing Redis interface code in foundations/bootstrap, so they can implement value storage.

Testing an API

When you're developing an API, it is recommended to build tests for the API as you build it. This way, you don't need to implement the complete API client to verify that it works as it should, but can interface with the API package even without a HTTP server/client structure, as you're developing it.

Let's first install the needed packages/libraries for our implementation:

Install all dependencies for our API package

```
gvt fetch "github.com/garyburd/redigo/redis"
gvt fetch "github.com/youtube/vitess/go/pools"
gvt fetch "golang.org/x/net/context"
```

Testing code in Go is done with the `go test` command. When we are creating our API package, we need to create these files: `registry.go` and `registry_test.go`. The last file should contain all the tests you need to perform to see that everything implemented in the first file works as it should.

A partial implementation of our API

```
1  type Registry struct {
2    Name string
3  }
4
5  func (r Registry) GetKey(key string) string {
6    return r.Name + ":" + key
7  }
8  func (r Registry) Get(key string) (string, error) {
9    k := r.GetKey(key)
10   return redis.String(bootstrap.RedigoDo("GET", k))
11 }
12 func (r Registry) Del(key string) (interface{}, error) {
13   k := r.GetKey(key)
14   return bootstrap.RedigoDo("DEL", k)
15 }
16 func (r Registry) Set(key string) (interface{}, error) {
17   k := r.GetKey(key)
18   return bootstrap.RedigoDo("SET", k)
19 }
```

We implement some of our commands, and when we are far enough to test them, we create a function in registry_test.go to call the functions we have implemented.

How to test our API

```
1  func TestRegistryGet(t *testing.T) {
2    redisPool := bootstrap.RedigoPool()
3    defer redisPool.Close()
4
5    reg := Registry{Name: "test"}
6    reg.Del("name")
7    val, err := reg.Get("name")
8    if err == nil || val != "" {
9      t.Errorf("Unexpected result when getting name: %s/%s\n", val, err)
10   }
11 }
```

There can be multiple testing functions in this file, each testing individual aspects or usage patterns in your API. Each testing function should test expected return values from your API, and issue an error if something unexpected occurred. Getting test results is as simple as running:

```
$ go test
PASS
ok      _/go    0.245s
```

This tells us that whatever tests we have implemented performed as expected (no failures). The additional option "-cover" gives us a more complete picture of tests:

```
$ go test -cover
PASS
coverage: 71.4% of statements
ok      _/go    0.224s
```

Code coverage is a number that tells us, how many lines of code out of all the lines of code have been run. In our case, we have a code coverage of 71.4%, which tells us that 28.6% of code was never executed. If you look at our API implementation and at our API tests, you will see the reason - our API function Set is not being tested yet. Let's add a few lines to our test:

How to test our API

```
1   func TestRegistrySet(t *testing.T) {
2     redisPool := bootstrap.RedigoPool()
3     defer redisPool.Close()
4
5     reg := Registry{Name: "test"}
6     status, err := reg.Set("name", "Tit Petric")
7     if status != "OK" || err != nil {
8       t.Errorf("Error when using SET: %s", err)
9     }
10    val, err := reg.Get("name")
11    if err != nil || val != "Tit Petric" {
12      t.Errorf("Got error when getting name: %s/%s\n", val, err)
13    }
14  }
```

And re-run go test to see what the testing result is:

```
$ go test
# _/go
./registry_test.go:26: too many arguments in call to reg.Set
FAIL    _/go [build failed]
```

Ah! We already found our first error when testing. It seems we didn't implement all the arguments for the Set function, which should also take a key value. Let's fix it:

A partial implementation of our API

```
1  func (r Registry) Set(key string, value string) (interface{}, error) {
2    k := r.GetKey(key)
3    return bootstrap.RedigoDo("SET", k, value)
4  }
```

Re-running the test leaves us with another error:

```
--- FAIL: TestRegistrySet (0.22s)
        registry_test.go:32: Got error when getting name: %!s(<nil>)/%!s(<nil>)
FAIL
exit status 1
FAIL    _/go    0.441s
```

Remember how we use a pool of connections? When we use commands like "SET", immediately followed by a command like "GET", they will use two different connections. The value for SET would be written on one connection (own server, redis1), and read from another (redis2) with GET, causing the error above. The test is failing because we're not GETting the value we expected, just after issuing a SET.

I quickly modified the bootstrap package, so getServerName (redigo.go) gives only one server name to connect to. I also decreased the connection pool capacity to 1. The correct way do handle this would be to get a connection from the pool and re-use it for the complete test, as we did with the MySQL connections in the previous chapter.

```
$ go test -cover
PASS
coverage: 100.0% of statements
ok      _/go    0.497s
```

Our tests now cover all of our code. What this means is that each line gets executed at least once, and that whatever tests we made, get expected results from our API implementation.

More detailed testing

When you're developing an API, you're usually interested in many aspects in regards to test coverage. Go provides tooling which allows you to get more details from your tests and your application.

Storing the coverage profile

To run your tests, storing a coverage profile, you can issue the following command:

```
$ go test -coverprofile="coverage.out" -covermode count
PASS
coverage: 100.0% of statements
ok        _/go    0.418s
```

The `covermode` parameter accepts the values: set (default), count and atomic. The value `set` just answers if a statement was run or not (boolean value), while the value `count` answers how many times it was run (number). The `atomic` setting is meant to provide reliable counts with parallel processing.

With the coverage profile, we can use `go tool cover` to get additional reports.

Coverage by function
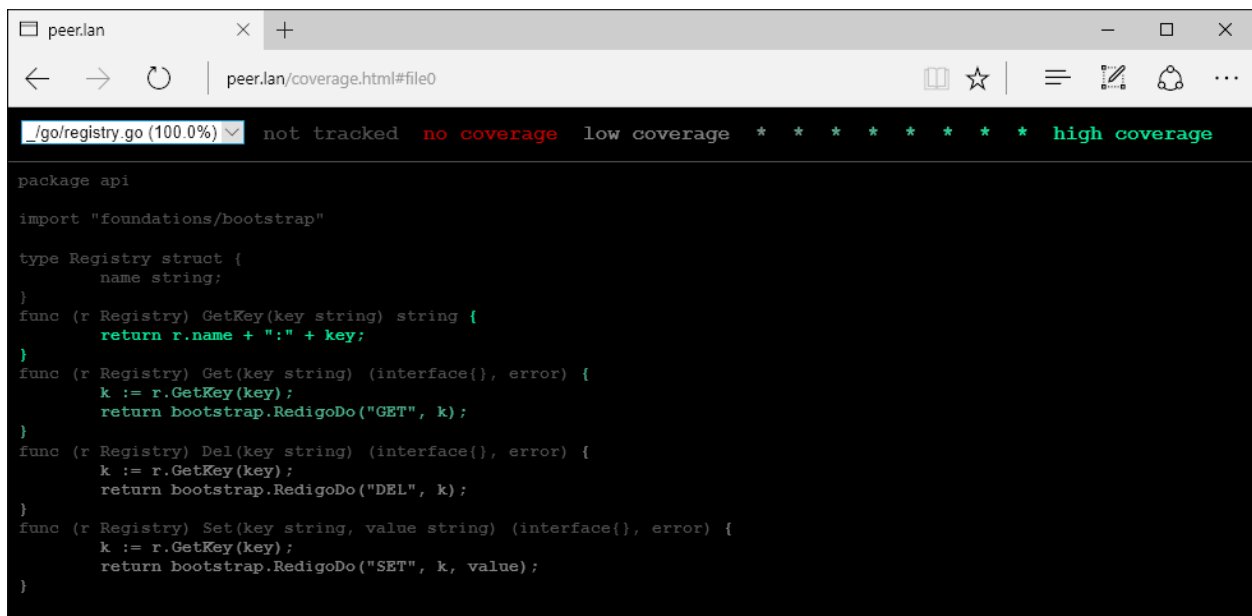
We can display coverage by function, letting us know which functions are not yet tested, or need more complete tests.

```
$ go tool cover --func="coverage.out"
_/go/registry.go:8:     GetKey            100.0%
_/go/registry.go:11:    Get               100.0%
_/go/registry.go:15:    Del               100.0%
_/go/registry.go:19:    Set               0.0%
total:                  (statements)      71.4%
```

Coverage HTML

We can also generate a code coverage report in HTML format, that gives us a better overview of test coverage.

```
$ go tool cover -html=coverage.out -o coverage.html
```

Result without parameter, got expected greeting.

The coverage report makes use of the cover mode displaying lines that have higher coverage in a bright-green colour.

A note on testing

I tend to keep high (as in 100%) code coverage for my core components. Even with such code coverage, issues occasionally happen. Tests don't predict all possible scenarios - you're testing your own code, which uses libraries which may introduce unpredictable behaviour - like calling "panic" in some edge cases, or conditional results which may vary on situation.

Tests tell you exactly what you ask from them. You predict scenarios and behaviour and enforce that your implementation follows them in any cases that you imagine. Even with a 100% code coverage, a new usage pattern might emerge that will result in an issue that needs to be resolved. A great example of this is the SET+GET behaviour at the start of this chapter. Even with 100% code coverage, an issue occurred and needed to be fixed outside of our API implementation - our application and our tests needed no changes.

A tested program works in all the way you imagined, and it fails in all the ways you did not.

Implementing the complete API

As we already created our "Get" and "Set" functions, we will now implement a "GetAll" function and the test for it.

Our final API method: GetAll

```go
func (r Registry) GetAll() (map[string]string, error) {
  k := r.GetKey("*")
  keys, err := redis.Strings(bootstrap.RedigoDo("KEYS", k))
  allkeys := map[string]string{}
  if len(keys) == 0 || err != nil {
    return allkeys, nil
  }
  for _, value := range keys {
    value_redis, err := redis.String(bootstrap.RedigoDo("GET", value))
    if err == nil {
      allkeys[value[len(r.Name+":"):]] = value_redis
    }
  }
  return allkeys, nil
}
```

The function uses the KEYS method of Redis to loop through the namespace and return all the keys with our Registry name (prefix). All the available keys are put in a map[string]string (array). We can check if we have the "name" index set in our test:
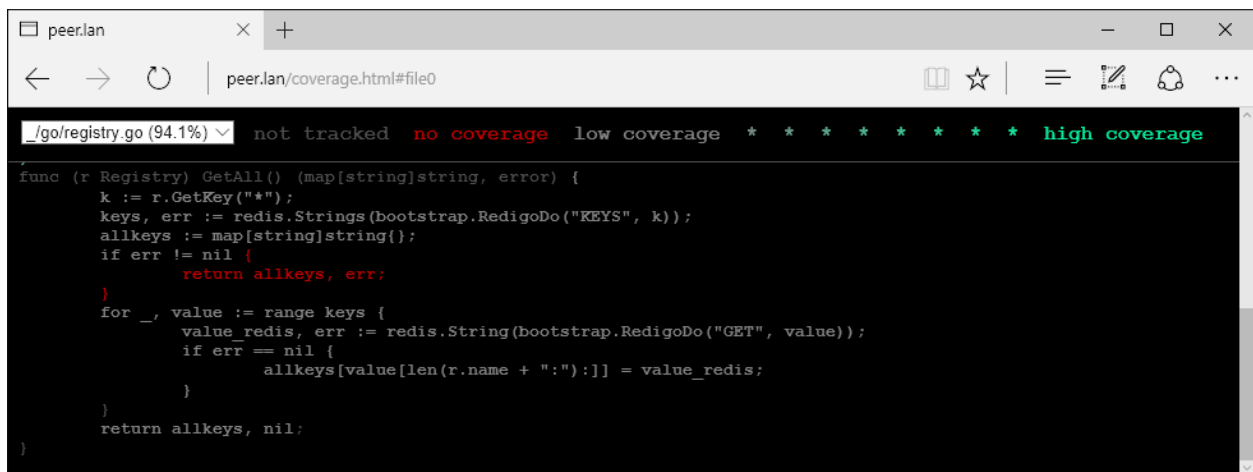
Testing GetAll output

```go
func TestRegistryGetAll(t *testing.T) {
  redisPool := bootstrap.RedigoPool()
  defer redisPool.Close()

  reg := Registry{Name: "test"}
  reg.Set("name", "Tit Petric")
  val, err := reg.GetAll()
  if err != nil || val["name"] != "Tit Petric" {
    t.Errorf("Got error when getting all keys: %#v/%s\n", val, err)
  }
}
```

This brings our test coverage up to 94.1%. There's some lines which we missed, let's look at the HTML report.

Coverage report for GetAll

It seems we missed one line in our tests. Generally, this will be executed when we have an empty result set from KEYS. There are two approaches to increasing code coverage here - provide Mock objects for whatever interface you have, or, provide data and requests that will produce the wanted response. Let's try to use the second method, by writing a new test.

Testing GetAll output

```
1   func TestRegistryGetAllErr(t *testing.T) {
2     redisPool := bootstrap.RedigoPool()
3     defer redisPool.Close()
4
5     reg := Registry{Name: "testerr"}
6     val, err := reg.GetAll()
7     if err != nil || len(val) != 0 {
8       t.Errorf("Expected len=0 and error=nil when getting all keys: %#v/%s\n", val, er\
9   r)
10    }
11  }
```

Since the Registry name 'testerr' doesn't exist in redis, the `redis.Strings` method will return an error. We use the testing function above to expect this error, and increase our code coverage to 100%.

With this, we have a fully featured implementation to use in our HTTP API.

Your first API

We learned enough to implement our first full API. We will use the package we created in the previous chapter, to implement an API that has these specifications:

- /get with parameter key - will return value from redis
- /set with parameters key, value - will set a key/value pair
- /getAll retrieve all redis keys

We will use the net/http package to provide routing to your API.

Putting the API together

First, we create a variable which will hold our Registry type.

```
1   var apiService api.Registry;
```

And we create two utility functions to handle responses - an error response, and an "anything" response (interface{}). To make our endpoint implementation shorter, we even create a generic respond function.

Utility functions to encode API responses

```
1   func respondWithError(w http.ResponseWriter, err error) {
2     response := map[string]string{}
3     response["error"] = fmt.Sprintf("%s", err)
4     response_json, _ := json.MarshalIndent(response, "", "\t")
5     fmt.Fprintf(w, string(response_json[:]))
6   }
7   func respondWith(w http.ResponseWriter, response interface{}) {
8     response_json, _ := json.MarshalIndent(response, "", "\t")
9     fmt.Fprintf(w, string(response_json[:]))
10  }
11  func respond(w http.ResponseWriter, response interface{}, err error) {
12    if err != nil {
13      respondWithError(w, err)
14      return
15    }
16    respondWith(w, response)
17  }
```

With these we can create our http request handlers for individual API calls; Setting up the API is as easy as this:

Implementing the API call for GetAll

```
1    apiService = api.Registry{Name: "api"}
2    http.HandleFunc("/getAll", func(w http.ResponseWriter, r *http.Request) {
3      response, err := apiService.GetAll()
4      respond(w, response, err)
5    })
6    http.HandleFunc("/get", func(w http.ResponseWriter, r *http.Request) {
7      key := r.FormValue("key")
8      response, err := apiService.Get(key)
9      respond(w, response, err)
10   })
11   http.HandleFunc("/set", func(w http.ResponseWriter, r *http.Request) {
12     key := r.FormValue("key")
13     value := r.FormValue("value")
14     response, err := apiService.Set(key, value)
15     respond(w, response, err)
16   })
```

And we have a fully functioning API.

Testing some requests:

```
$ curl http://localhost:8080/set?key=foo&value=bar
"OK"
$ curl http://localhost:8080/set?key=name&value=Tit%20Petric
"OK"
$ curl http://localhost:8080/get?key=name
"Tit Petric"
$ curl http://localhost:8080/getAll
{
        "foo": "bar",
        "name": "Tit Petric"
}
```

Benchmarking it

As a curiosity, let's benchmark our API service. I'm only interested in the "get" endpoint, so I'm running apache bench with some options like -c 4 (concurrency) and -n 50000. I'm doing this from inside a virtual machine on the same host - so take the benchmarks with a pinch of salt.

```
Requests per second:     1908.96 [#/sec] (mean)
```

This is the result I get. And keep in mind, we don't even have goroutines yet! Let's add those.

```
Requests per second:     1947.21 [#/sec] (mean)
```

Uh, this is surprising. It's not exactly the request rate we'd expect. To understand why, we have to know that "net/http" server is already creating goroutines for each connection. That means that unless we're doing some kind of parallel processing, there will be no positive impact from using goroutines. Since goroutines are lightweight compared to threads, you might not even notice any impact.

So, when it comes to using goroutines, we can sum up our experience like this: If you're doing any kind of parallel workload, or I/O operations, using goroutines can speed up the response times of your APIs. If you are doing atomic operations or sequential requests, you will not squeeze a performance benefit out of your code by using goroutines, as the request is already within one.

Profiling it

The package "net/http" also provides runtime profiling data for your server. This way you can see where your API is really spending CPU cycles and memory. To use pprof, you'll have to add this import line to your application:

```
import _ "net/http/pprof"
```

After you import this package and start your server, you can navigate your browser to http://[server]:8080/debug/. You can also use the go tool pprof to review many aspects of your API service. For example:

```
$ go tool pprof http://[server]:8080/debug/pprof/profile
```

This gives you a console where you can issue commands like topN ('top10 –cum/flat') to find the worst performing functions in your service. There are also commands like "dot > file.dot" which will generate a call graph which you can then render with graphviz.

The tool pprof is very powerful. If you want some information from your API service while it's alive, it's most likely that you will find it by researching run time options and analyzing the output.

Flame graph

With the pprof library enabled, there are also 3rd party tools available to analyze profiler data. One of such tools is go-torch[23] which generates a flame graph in svg format. To install go-torch, along with the FlameGraph dependency, issue the following commands:

[23]https://github.com/uber/go-torch

```
$ gvt fetch github.com/uber/go-torch
$ git clone --depth=1 https://github.com/brendangregg/FlameGraph.git /opt/flamegraph
```

This will download and compile the go-torch binary, which will be placed in the `bin/` folder of your current working directory. I've created a small script to aid me in running it:

Running go-torch to provide 15 seconds of profile data

```bash
1  #!/bin/bash
2  PATH="$PATH:/opt/flamegraph"
3  bin/go-torch --time=15 --file "torch.svg" --url http://10.1.1.2:8080
```

The run, if everything went well, will provide us with a flame graph in the file "torch.svg". Put the file on your web server and open it with a browser and you should get something like this.
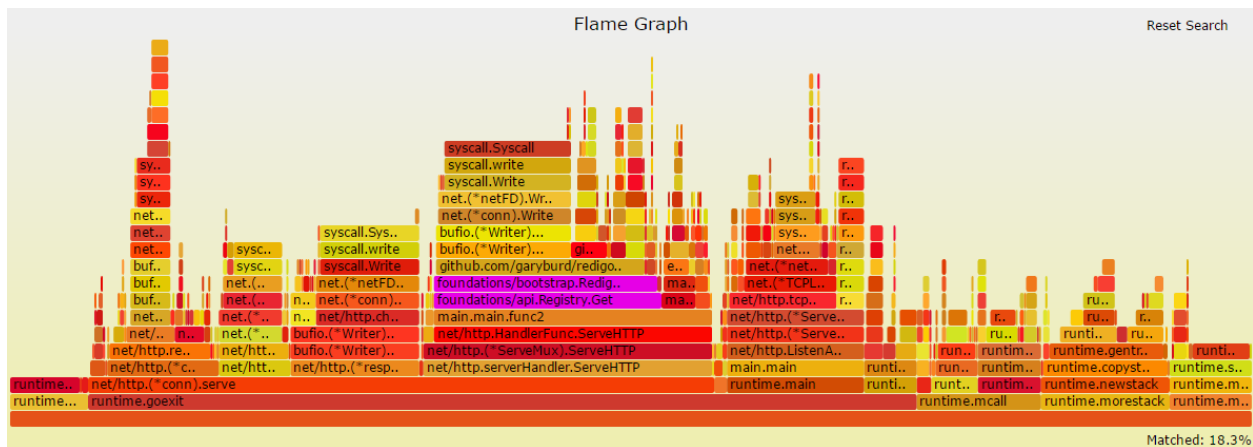
The Flame graph for our API

With the graph it's possible to search for some specific function or package (top right corner). It's also possible to click on an individual block to zoom in on the flame. Clicking our flame (main.main.func2), I can inspect where I'm spending time and with that what can be optimized.
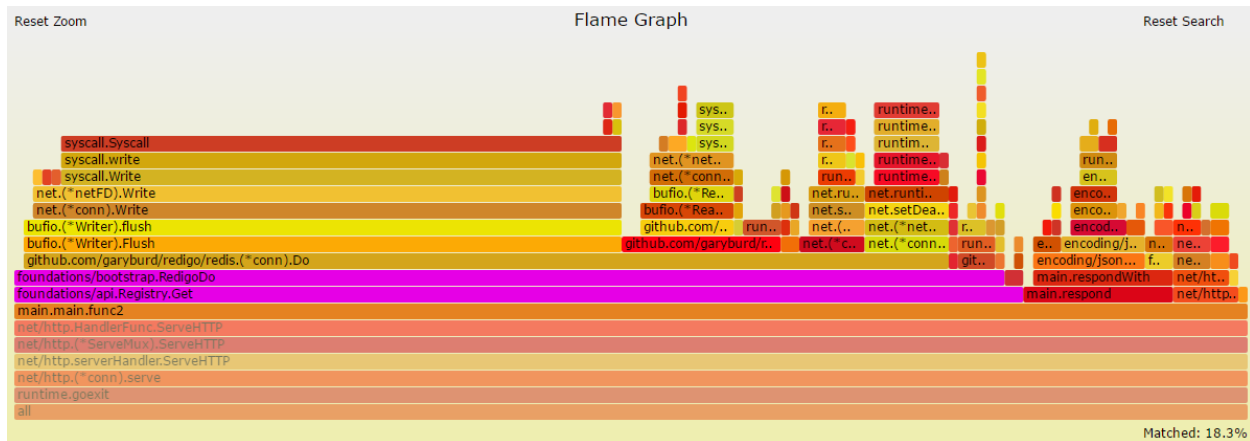
Zooming in on our API

With our simple API, there's no low hanging fruit to optimize away, but you can see that most of the time is taken by fetching data from Redis (about 50%), parsing the data (30%) and the remaining time is taken by encoding this data to JSON and FormValue calls.

Running your API in production

There are a couple of considerations to make when running Go in production. These considerations range from providing configuration, deployment and having insight into the operation of your application. We'll try to cover some established practice to give you a sense of what it takes for frictionless operations.

Configuration

Running applications in production requires a way to pass configuration. A common way to pass configuration via command line is the flag package[24]. Because we follow the Twelve-Factor app paradigm, we want the configuration to come from the environment. The package namsral/flag[25] provides a drop in replacement for the Go flag package.

You should use this package in a way that causes minimal friction towards your application. It is recommended to define flags inside your `func main` and not as global variables. This forces you to be strict about dependency injection patterns, which also makes testing easier.

Example application using namsral/flag

```
1  package main
2
3  import "github.com/namsral/flag"
4  import "fmt"
5  import "os"
6
7  func main() {
8    fs := flag.NewFlagSetWithEnvPrefix(os.Args[0], "GO", 0)
9    var (
10     port = fs.Int("port", 8080, "Port number of service")
11   )
12   fs.Parse(os.Args[1:])
13
14   fmt.Printf("Server port: %d\n", *port)
15 }
```

A good practice shown in the example is to prefix your environment variables, so you don't create any clashes with the default linux environment (usual culprits: MAIL, HOSTNAME, USER). In our

---

[24]https://golang.org/pkg/flag/
[25]https://github.com/namsral/flag

case, we will use a GO prefix, meaning we can safely define GO_MAIL, GO_HOSTNAME and GO_USER.

Practical examples of passing arguments/config

```
$ go run flags.go
Server port: 8080
$ go run flags.go -port=80
Server port: 80
$ PORT=80 go run flags.go
Server port: 8080
$ GO_PORT=80 go run flags.go
Server port: 80
```

When running "go" via docker, the environment variables are passed via additional configuration parameters. Since we declared a GO prefix for the environment variables used, we can extract only these and pass them to a docker run command. You should pass them explicitly, with the `--env-file` option or individual `-e` options.

A Docker example with flags.sh

```bash
1  #!/bin/bash
2  source ../shell/common.sh
3  gvt fetch "github.com/namsral/flag"
4  printenv | grep GO_ > /tmp/docker.env
5  docker run --rm --env-file /tmp/docker.env -i -v `pwd`:/go/src/app -w /go/src/app go\
6  lang go run flags.go
```

You can run this script with `GO_PORT=80 ./flags.sh`.

> The chosen library also supports parsing configuration from a configuration file, which might be something you prefer over passing environment variables. We usually keep settings like this in some sort of central registry (configuration database) which we extend based on our needs.

Building an application

Building an application is as easy as running:

```
$ go build flags.go
```

This will create a `flags` binary in your current folder, which you can run just like any other program. Since we defined a configuration flag, we can quickly test it to see if it works:

Practical examples of passing arguments/config

```
$ ./flags
Server port: 8080
$ ./flags -port=80
Server port: 80
$ PORT=80 ./flags
Server port: 8080
$ GO_PORT=80 ./flags
Server port: 80
```

The binary itself behaves just like `go run`, except that the compilation was already done beforehand. It's also a static binary, so you can copy it to another server, and run it there - without needing the Go runtime, or some libraries that might be required (Go doesn't rely on shared libraries).

Deploying an app somewhere can be simple

```
$ scp flags luxor:/root
flags                      100% 2627KB   2.6MB/s   00:01
$ ssh luxor "/root/flags"
Server port: 8080
$ ssh luxor "/root/flags -port 80"
Server port: 80
$ ssh luxor "GO_PORT=80 /root/flags"
Server port: 80
```

> If you needed to build against a different operating system or architecture, it's possible to pass environment variables to `go build`: GOOS and GOARCH specifically. With these you can control what kind of destination system you're building for. More information is available in Go build documentation[a].
>
> [a]https://golang.org/pkg/go/build/

Embedding binary assets into an application

For reasons of convenience, you might want to embed the data which you need for your application into your binary directly. This might be anything from static javascript, css and image assets, that

are used to provide some kind of web interface for your API. With the popularity of front-end frameworks like VueJS[26], you can package the complete front-end application together with your API service.

Embedding binary assets into your API is convenient for your users. You provide a single executable which will contain everything, so they don't need to download an installer or extract zip files. We will use go-bindata[27] and resort to code generation to add all the data into the executable with go build.

I wrote a simple static file editor called Pendulum[28] where I implemented all the below steps to provide a binary download with all the needed data embedded into the binary.

Code generation?

Well, sure. It's simple enough. The go-bindata tool already enables us to create a .go file from a `public_html` folder, for example. This is perfect for our use case. But why resort to a bash script or Makefile to produce it? We can use Go's code generation tool and just call `go generate` before we call `go build`.

If you want to get started with code generation, all you need to do is include a single comment somewhere in your source code, `main.go` for example:

```
1  package main
2
3  //go:generate echo "Hello world"
4
5  func main() {
6  }
```

When you run `go generate`, you will find out that it prints "Hello world". It's not really a requirement that you would generate any code with go generate. Whatever you put after the `//go:generate` text will be executed. You can even run `go build` if you wanted.

---

[26]https://vuejs.org/
[27]https://github.com/jteeuwen/go-bindata
[28]https://github.com/titpetric/pendulum

```
1  package main
2
3  //go:generate echo "Hello world"
4  //go:generate go run main.go
5
6  func main() {
7      println("Hello world from Go")
8  }
```

Running it will produce the expected result:

```
1  # go generate
2  Hello world
3  Hello world from Go
```

Inception! Well, go generate is... *interesting*. Node programs are using `babel` to provide ES6/ES7 syntax capabilities to Node ES5 runtime, and people are attempting to use the same approach to provide Go with functionality beyond the current scope of the language.

For example: The genny[29] project is an example more directed at generating typed-code so you don't really have to copy paste aggressively, but projects like Have[30] went closer to what Babel is doing with Node - providing a language which transpiles to Go. I don't know of any other attempts that got more traction, yet. The discussions about Go2 and generics seem to suggest however, that there's some interest for this.

Well, for our case, we're slightly more boring, we're just trying to package some data in our application, so let's interrupt this social commentary and continue:

```
1  //go:generate go-bindata -prefix front/src -o assets/bindata.go -pkg assets -nomemco\
2  py front/src/dist/...
```

That's a bit of a long line, let's break it down just for visual inspection:

- `//go:generate` - hint for `go generate`,
- `go-bindata` - main command that executes,
- `-prefix front/src` - exclude "front/src" from package,
- `-o assets/bindata.go` - generated output file location,
- `-pkg assets` - name of the package we're generating,
- `-nomemcopy` - an optimization for a lower memory footprint[31],
- `front/src/dist/...` - the location we're packing.

This creates an `assets` package in your application folder, which you can import with a short import, i.e. `app/assets`, where `app` matches your application folder.

---

[29]https://github.com/cheekybits/genny
[30]http://havelang.org/
[31]https://github.com/jteeuwen/go-bindata#lower-memory-footprint

Serving embedded files via HTTP

This is where things get a little bit complicated. Or simple, after you read a bit of documentation. If you wanted to serve local files, you would use something like the following line of code:

```
folder := http.Dir("/")
server := http.FileServer(folder)
http.Handle("/", server)
```

In fact, the package go-bindata-assetfs[32] provides an implementation for http.FileServer. Using it is simple enough:

```
import "github.com/elazarl/go-bindata-assetfs"
import "app/assets"
// ...
func main() {
    // ...
    files := assetfs.AssetFS{
        Asset:     assets.Asset,
        AssetDir:  assets.AssetDir,
        AssetInfo: assets.AssetInfo,
        Prefix:    "dist",
    }
    server := http.FileServer(&files)
    // ...
}
```

There is only a slight hiccup. I am using a VueJS app with pushHistory enabled. This means, that when the user navigates the app, they will see links without a shebang (hash, #), but plain ordinary absolute links like /blog/about.md. Which don't exist in this asset filesystem, but are handled with the application.

Well, turns out the solution is simple enough. The assetfs.AssetFS structure has functions AssetInfo (which is the equivalent of os.Stat), and the function Asset (sort of like ioutil.ReadFile). With this it's possible to check if a file exists in the asset filesystem, and to output a different file if it doesn't:

```
1   // Serves index.html in case the requested file isn't found
2   // (or some other os.Stat error)
3   func serveIndex(serve http.Handler, fs assetfs.AssetFS) http.HandlerFunc {
4       return func(w http.ResponseWriter, r *http.Request) {
5           _, err := fs.AssetInfo(path.Join(fs.Prefix, r.URL.Path))
6           if err != nil {
7               contents, err := fs.Asset(path.Join(fs.Prefix, "index.html"))
8               if err != nil {
9                   http.Error(w, err.Error(), http.StatusNotFound)
10                  return
11              }
12              w.Header().Set("Content-Type", "text/html")
13              w.Write(contents)
14              return
15          }
16          serve.ServeHTTP(w, r)
17      }
18  }
```

In case the file is found, I use the provided ServeHTTP method, instead of providing my own implementation. All it takes to use this is a bit of a change in the handler which we defined before:

```
1   http.HandleFunc("/", serveIndex(server, assets))
```

The function `serveIndex` returns a `http.HandlerFunc`, and this line was changed accordingly. This provides a full implementation of how to serve your data which you add to your application with go generate and go-bindata. And if you want to skip the `//go:generate` part and just put it in your CI scripts, that's fine too!

And with this I implemented a single-executable release for Pendulum[33]. Grab it from the GitHub releases page[34] to check it out.

Deploying an application

Deploying an application can be as simple as copying the binary produced with `docker build`, and managing scripts and configuration around it. But deploying the application with docker itself has some benefits. Creating an image containing the docker binary might seem like an unusual or wasteful practice, but gives many benefits.

  1. The application can be downloaded and run with a simple `docker run` command (pull mode),

---

[33]https://github.com/titpetric/pendulum
[34]https://github.com/titpetric/pendulum/releases

2. Docker images can be transferred between hosts (push mode),
3. Docker is also a simple process manager (`--restart=always`)

The other option is to create `init.d` scripts, or run an instance of `supervisord` to manage execution and restarting of your service. There are also other benefits from docker - isolation from a security standpoint for one.

Even if your application uses many hosts because of scale, running it from Docker should be considered - the Docker images can be versioned, and if you're not exactly doing upgrades of the data model, this means that you can also safely roll-back if you deployed a bug to production.

Automating a deployment of an application can be a dirty business, but when you're deploying the complete environment for the application along with it, you're saving a lot of time by avoiding certain discussions like "Did you copy all the files?", or "I don't see the changes, when will the deploy finish?", "This one file is outdated." and so on.

In a sense, deployment with docker is "atomic". In another sense - it's progressive. You can spawn new container instances of the updated application, and when you verify it works, you can spin down the old, out of date containers. You can use clustering tools like Docker Swarm[35] to achieve this from the start.

Creating a Docker image

Docker images are created from a Dockerfile. Since Go binaries are 'portable', they can run on the smallest Docker image around, Alpine Linux. Keep in mind, that sometimes, just having the binary is not enough, as some of the libraries used need external data files. For example, a web server might need `/etc/mime.types` to return correct Content-type headers for files, an SSL library would need a list of SSL root certificates, usually found in `/etc/ssl/certs`. This is why it's better to use Alpine linux, instead of 'scratch' as the base image - it provides a package manager, allowing you to install some things, without having to COPY them in.

Example Dockerfile for our app

```
1   FROM alpine:latest
2
3   ADD flags /flags
4   RUN chmod +x /flags ; sync; sleep 1
5
6   WORKDIR /
7
8   ENV GO_PORT 8080
9   EXPOSE $GO_PORT
10
11  ENTRYPOINT ["/flags"]
```

[35]https://docs.docker.com/swarm/

Building the Docker image is a one liner -

Build Go application and create docker image

```bash
#!/bin/bash
docker build --rm --no-cache=true -t go-flags .
```

We can run our application with full control of the network port on which it runs. You can expose this port on the host using `-p` or `-P` docker options. If you're using `--net=host` mode, you can prevent some conflicts here with existing services by running multiple containers on different ports.

```
$ docker run -e GO_PORT=81 --rm -i go-flags
Server port: 81
```

The images you create can be used on multiple hosts, by using a Docker registry like Docker Hub or Quay.io. You can also set up your own docker registry. The most basic way to transfer the images between hosts is to save and load them.

```
$ docker save -o go-flags.tar go-flags
$ scp go-flags.tar luxor:/root
go-flags.tar                                                          1\
00% 7565KB    2.5MB/s    00:03
$ ssh luxor "docker load -i go-flags.tar"
$ ssh luxor "docker run -e GO_PORT=1234 --rm go-flags"
Server port: 1234
```

Depending on what you need, pick a deployment strategy which works for you.

> I'm of the opinion that each container should be disposable. If you provide any kind of data storage, it should be done with volume mounts, or some external API service (Amazon S3 for example). You know exactly which *data* needs to be backed up, and it doesn't include the application itself. The app can be rebuilt at any time, and the container re-created from scratch. I'm aiming to reproduce these processes with automation, not by following a setup checklist. My longest running practice is, that I can remove *any* running container with "docker rm -f [name]". Starting them again or creating a new environment should be just as trivial.

Exposing run-time information

When you deploy an application, you need to have some metrics about how it performs. While you can rely on `pprof`, and that access log from your load balancer, it might be neat to export some

additional information about your app. Maybe you want to track sessions, sign-ups, or some variable which you can't get by processing the access log. You can use expvar to log these values to a public interface, available over HTTP on /debug/vars in JSON format. To use it, just import it like this:

```
import "expvar"
```

You should use the expvar package to register some public variables, which will show up in /debug/vars output. The recommended way to do this is in the package init function. Let's extend our registry API with expvar:

Add expvar capability to your Registry API

```
1  import "expvar"
2
3  var (
4    countGet       *expvar.Int
5    countSet       *expvar.Int
6    countDel       *expvar.Int
7    countGetAll    *expvar.Int
8    countGetAllGet *expvar.Int
9  )
10
11 func init() {
12   countGet = expvar.NewInt("registry.get")
13   countSet = expvar.NewInt("registry.set")
14   countDel = expvar.NewInt("registry.del")
15   countGetAll = expvar.NewInt("registry.getAll")
16   countGetAllGet = expvar.NewInt("registry.getAll.get")
17 }
```

Changing a counter value is just as easy as calling value.Add(1). With the counters for GetAll, I also defined a GetAllGet counter, which I increment by a larger value: int64(len(keys)). In practice this tells me how many keys on average I have in the registry (three in my case).

Looking at the output of /debug/vars, after some manual requests, I can see all the counters which we defined:

```
"registry.del": 0,
"registry.get": 5,
"registry.getAll": 18,
"registry.getAll.get": 54,
"registry.set": 2
```

The metrics you define can provide invaluable information about your application and how it performs. Monitoring the metrics values you set is an important part of any application, making sure that you're on the right track towards performance and scalability.

To graph the metrics you can use a variety of tools like Grafana[36] or OpenTSDB[37]. We use also a few others like Ganglia[38], Munin[39] and most recently, netdata[40]. Depending on how long you'd like to keep your metrics, and how much detail you want to store your metrics history, you have quite the choice ahead of you.

---

[36]http://grafana.org/
[37]http://opentsdb.net/
[38]http://ganglia.info/
[39]http://munin-monitoring.org/
[40]https://github.com/firehol/netdata

Resources

You can find all the examples published on GitHub in the repository [titpetric/books](https://github.com/titpetric/books)[41] in the folder `api-foundations`.

As you will check out the repository, code samples are divided into chapters, and individual chapters have `run` scripts, which you can use to run all the samples from the repositories. External dependencies have been vendored into the repository.

All examples are meant to be run with `docker`, but a working go environment should be enough. If you for some reason can't use docker, just empty the contents of the `common.sh` file under `shell` folder.

---

[41]https://github.com/titpetric/books