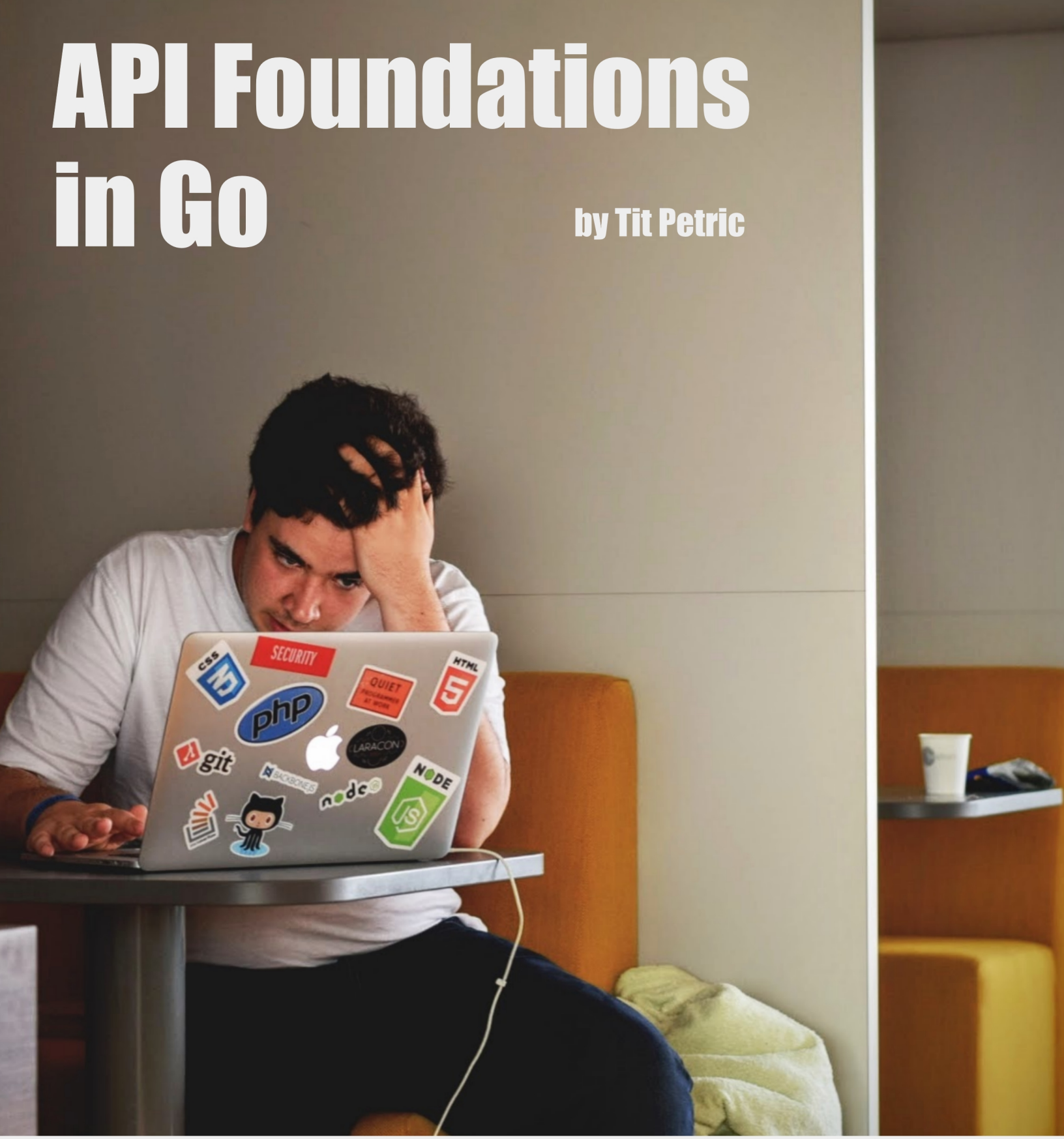
API Foundations in Go

by Tit Petric



API foundations in Go

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

Tit Petric

© 2016 Tit Petric

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:

https://twitter.com/search?q=#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

Introduction

About me

I'm one of those people with about two decades of programming experience under my belt. I've started optimizing code in the 90's, discovered PHP in the 2000's, and build several large-scale projects on to the day, 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 an software development kit for the Slovenian national TV and Radio station website, RTV Slovenia. I'm also the 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, and in the back stores and retrieves all it's data from Redis. Since we have a code base which spans about 13 years for our biggest project, it was more likely that we would keep the working code as-is, and just replace the software around it when needed. Go made this possible.

One of the reasons for choosing Go was the constant comparison of Go against Node.js. Node has a much more vocal community which seems to religiously advocate it. We made several tests during the recent months, and Node, while not very hard to start development with it, underperformed 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 better 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, which might not have had a chance to try Go, but are familiar with concepts of API development in languages like PHP or Node.js. 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 discover here.

I'm not going to be explaining what some characteristics of the Go programming language are. I'm going to be diving head first into using it and just making 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

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

A development environment, as well as a production environment, is an important subject 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 our services which we will use can talk to each other. Creating a network is simple, all you should 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 will 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, done in just one line:

```
$ docker run --net=party -p 8080:80 --rm=true -it -v `pwd`:/go 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) to the container
- **"$@"** - passes all arguments to go to the container application

Very simply, what this command will do 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 trough 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

Just as simple as setting up Go, we will use Docker to run Redis. 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

Just like go and redis, other services are 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 flavors)
- 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 testing 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:

```
type Petstore struct {
  Name string
  Location string
  Dogs []*Pet
  Cats []*Pet
}

type Pet struct {
  Name string
  Breed string
}
```

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 uppercase 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 uppercase 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  func main() {
9    dog := Dog{ name: "Rex" };
10   cat := Cat(dog);
11 }
```

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 an 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

```
type Dog struct {
  name string;
  breed string;
}
type Cat struct {
  name string;
  hypoallergenic bool;
}
type Petstore struct {
  name string;
  pets []interface{};
}
```

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

```
func (d Dog) getName() string {
  return d.name;
}
func (c Cat) getName() string {
  return c.name;
}
type Pet interface {
  getName() string;
}
```

Abusing interfaces

Interfaces are a powerful and flexible way to work with objects of variable 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

```
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 that they are not overused. 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, much like encoding/json package which we will use in a following chapter.

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. These packages you can install with the commands "go get [package]". You will use several packages during the book, and the examples will also create some packages, which we will explain in this section.

Suggested package structure

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

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

For whatever API you need, you can create additional packages, from which you can now import `foundations/bootstrap` package as needed.

What to put there?

The `bootstrap` package should create some functions, which you would often need. A very simple function would be this:

Example bootstrap file - now.go

```
1  package bootstrap;
2
3  import "time"
4
5  var StartTime float64;
6  func Now() float64 {
7    myTime := float64(time.Now().UnixNano()) / 1000000.0;
8    if (StartTime < 0.000001) {
9      StartTime = myTime;
10   }
11   return myTime - StartTime;
12 }
```

If you need to time how much time some 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 functions along the way.

The function `New` 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 counting.

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

Example usage of bootstrap package

```
1  package main
2
3  import "foundations/bootstrap"
4  import "fmt"
5
6  func main() {
7    fmt.Printf("Time: %.4f\n", bootstrap.Now());
8    fmt.Printf("Hello world!\n");
9    fmt.Printf("Time: %.4f\n", bootstrap.Now());
10
11   bootstrap.StartTime = 0;
12   fmt.Printf("Time after reset: %.4f\n", bootstrap.Now());
13 }
```

The example aboves produces this output:

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

> Go supports a feature called "import comment", where you can specify the main package location on GitHub, BitBucket or other code hosting services. It gives you a way to structure your packages within the same repository and omit putting them in the `src/[project]/` folder.

Structure

Encoding and decoding JSON

The first thing to specify when encoding such 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.

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

```
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"`
}

type PetStoreList []*Petstore

func main() {
  petstorelist := PetStoreList{};
  petstore := &Petstore{
    Name: "Fuzzy's",
    Location: "New York, 5th and Broadway"
  };
  petstore.Dogs = append(petstore.Dogs,
    &Pet{
      Name: "Whiskers",
      Breed: "Pomeranian"
    }
  );
```

```
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:

```
[
  {
    "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[2] 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.

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[3] to "dump" this structure for inspection, so we can see that all the data was decoded without errors.

[2] https://golang.org/pkg/encoding/json/#Marshal
[3] https://github.com/davecgh/go-spew

Example: Decoding JSON contents into structs

```
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:

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>
 })
}
```

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.

Setting up a simple web server

Most modern languages today have a simple framework which allows you to implement a web server very quickly. True scalability however, is usually a thing which is done by a specialized load balancer or reverse proxy - nginx, haproxy or other.

As an example, PHP provides a FastCGI interface, to implement the communication between the web server, and the PHP interpreter. While you could implement the same kind of interface from a Node.js or Go application, we have better options available - we can implement our own HTTP server. The principles don't change much.

A minimal "Hello World" web server

```
1  package main
2
3  import (
4    "log"
5    "fmt"
6    "net/http"
7  )
8
9  func requestHandler(w http.ResponseWriter, r *http.Request) {
10   fmt.Printf("Request: %s\n", r.URL.Path);
11   fmt.Fprintf(w, "Hello world!");
12 }
13
14 func main() {
15   fmt.Printf("Starting server on port :80\n");
16   http.HandleFunc("/", requestHandler);
17   err := http.ListenAndServe(":80", nil);
18   if (err != nil) {
19     log.Fatal("ListenAndServe: ", err);
20   }
21 }
```

Go ahead and run it, and then open port http://yourmachine:8080/ in your browser.

After running the browser, "Hello World" is displayed

In the console you will see some info about the request. Press CTRL+C to stop the server.

```
$ go run server1.go
Starting server on port :80
Request: /
Request: /favicon.ico
^Csignal: interrupt
```

Routing logic

A big part of having an API server is routing logic, so you know which request needs to run which API logic. For example, the request for `/hello` should respond differently for different parameters. Our goal is to implement the parameter "name", which we can use in the application.

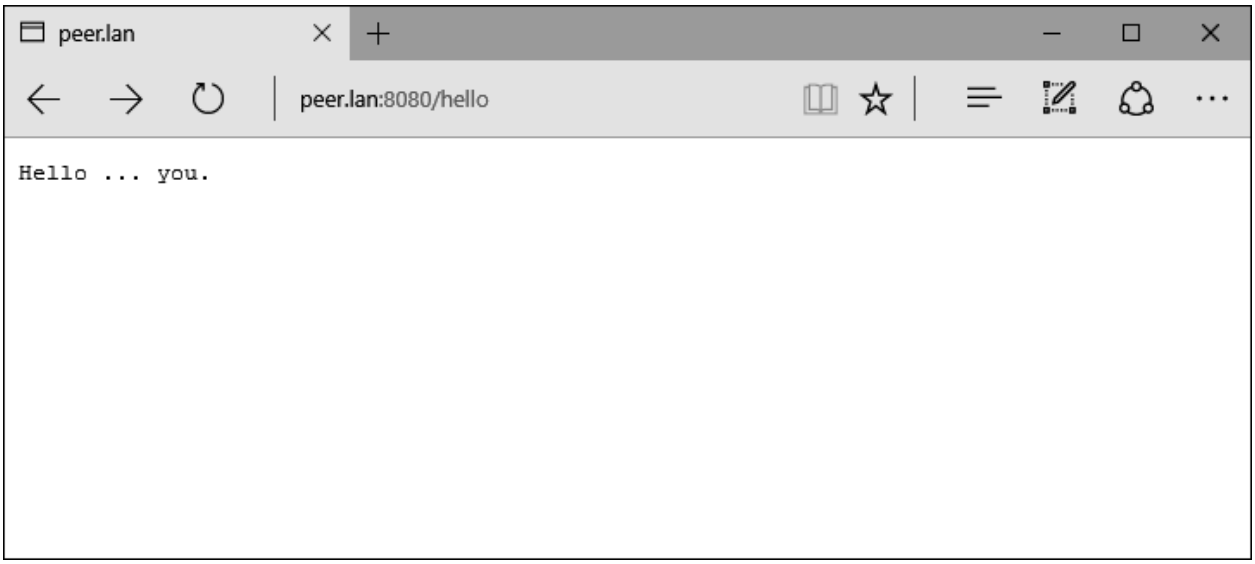
We add a new HandleFunc call with our new handler:

```
1  http.HandleFunc("/hello", requestHelloHandler);
```

And implement our logic. We want to print "Hello … you." if no parameters have been passed to our API call. Let's run the script and test it.

Request handler for our API call

```
func requestHelloHandler(w http.ResponseWriter, r *http.Request) {
  fmt.Printf("Request: %s\n", r.URL.Path);
  val := r.FormValue("name");
  if val != "" {
    fmt.Fprintf(w, "Hello %s!", val);
  } else {
    fmt.Fprintf(w, "Hello ... you.");
  }
}
```



Result without parameter, got expected greeting.

Result with parameter, parameter wrapped in brackets?

> *Note*: The HandleFunc is very explicit about the URL. Links like "/hello/", or just starting with the pattern ("/hello_there") will not be caught in our example. They will be caught by the handler for "/" however.

Advanced routing - Pat

There are a few frameworks available for HTTP routing in Go. The Gorilla Mux[4] package seems to be the crowd favorite with most features. Another one recommended for simplicity and speed is Pat[5]. Since the key to APIs is low-latency responses, I did choose Pat for the following examples, but be sure to consider other options if you need more powerful routing.

To install pat, run:

```
$ go get github.com/bmizerany/pat
```

Let's, for the start, just allow passing of the name parameter via URL instead of query arguments. We want to match the pattern "/name/:name".

[4] http://www.gorillatoolkit.org/pkg/mux
[5] https://github.com/bmizerany/pat

Request routing with Pat

```
func requestHelloHandler(w http.ResponseWriter, r *http.Request) {
  fmt.Printf("Request: %s\n", r.URL.Path);
  val := r.URL.Query().Get(":name");
  if val != "" {
    fmt.Fprintf(w, "Hello %s!", val);
  } else {
    fmt.Fprintf(w, "Hello ... you.");
  }
}

func main() {
  fmt.Printf("Starting server on port :80\n");

  m := pat.New();
  m.Get("/hello/:name", http.HandlerFunc(requestHelloHandler));
  m.Get("/", http.HandlerFunc(requestHandler));

  http.Handle("/", m);
  err := http.ListenAndServe(":80", nil);
  if (err != nil) {
    log.Fatal("ListenAndServe: ", err);
  }
}
```

It seems to me, that the request routing is still not where it should be. In fact, the result might be worse than what we had with http.HandleFunc, depending on your requirements.

- “/name/:name” doesn’t catch “/name/” (no parameter) or “/name/a/” (trailing slash)
- ”/” handler doesn’t catch “/anything” (exact match)
- “/name/:name/” catches “/name/:name” only to redirect with trailing slash
- Non-matched requests return 404 (just as a reminder)

The deal-breaker here is the redirection which happens with defining a “/name/:name/” route. Pat will prefer the trailing slash in all cases, redirecting away from “/name/:name” instead of just resolving the call. In an API service - this is a bad thing to do.

```
$ telnet localhost 8080
Trying ::1...
Connected to localhost.
Escape character is '^]'.
GET /hey/ummm HTTP/1.0

HTTP/1.0 301 Moved Permanently
Location: /hey/ummm/
Date: Thu, 31 Mar 2016 14:25:02 GMT
Content-Length: 45
Content-Type: text/html; charset=utf-8

<a href="/hey/ummm/">Moved Permanently</a>.

Connection closed by foreign host.
```

Here's a valid example of how something may be more trouble than it's worth.

Advanced routing - Gorilla Mux

Hello! Let's pretend the previous section / dead end didn't happen. Today we will be trying out Gorilla Mux for our routing needs. Our requirements are quite simple. We want to define our routes in a way where the following URLs will match the same handler:

- /say
- /say/
- /say/{name}
- /say/{name}/

The difference between the links is just that the first two have empty parameters, and the last one has a trailing slash which is ignored. If you add something behind the trailing slash - the route should not be matched.

First, let's install Gorilla Mux:

```
$ go get github.com/gorilla/mux
```

And then let's update our router:

```
m := mux.NewRouter();
hey := m.PathPrefix("/hey").Subrouter();
hey.HandleFunc("/{name}/", requestHelloHandler);
hey.HandleFunc("/{name}", requestHelloHandler);
hey.HandleFunc("/", requestHelloHandler);
http.Handle("/", m);
```

Of course, the requestHelloHandler needed to be updated also. You should use mux.Vars() to create a map of route variables, like "name" in our case.

Request routing with Pat

```
func requestHelloHandler(w http.ResponseWriter, r *http.Request) {
  fmt.Printf("Request: %s\n", r.URL.Path);

  vars := mux.Vars(r);
  name, ok := vars["name"];
  if ok && name != "" {
    fmt.Fprintf(w, "Hello %s!", name);
  } else {
    fmt.Fprintf(w, "Hello ... you.");
  }
}
```

> Note: The main path "/hey" does get redirected to "/hey", but thankfully no redirects happen within individual matching of "/{name}" and "/{name}/". You don't have to enforce a strict client policy and common URL patterns work from the start.

Gorilla Mux provides powerful routing which makes happy even the most demanding developers. In this case the most demanding developer is me.

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 are also using the `foundations/bootstrap` package, with the method `New`, 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

```
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

```
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);

  _ = json.Unmarshal(fn_contents, &firstname);
  fullname.Firstname = firstname.Firstname;

  // fetch lastname
```

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

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

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

When we write to a chan from a goroutine, we must also read from this chan. 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

```
  fn_contents := <-fn_chan;
  _ = json.Unmarshal(fn_contents, &firstname);
  fullname.Firstname = firstname.Firstname;

  ln_contents := <-ln_chan;
  _ = json.Unmarshal(ln_contents, &lastname);
  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 trough every item in the twitter feed, detecting links and embedding images. Some program is going trough 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 `src/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 "apiservice"
4
5  func main() {
6    apiservice.HelloWorld();
7  }
```

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:

```
$ go get 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, writeTimeo\
14 ut);
15 }
```

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

```
package main

import "foundations/bootstrap"
import "log"
import "fmt"

func main() {
  redis, err := bootstrap.GetRedis();
  if err != nil {
    log.Fatal("Fatal error: ", err);
  }
  fmt.Printf("[%.4f] Starting\n", bootstrap.Now());
  pong, err := redis.Do("PING");
  fmt.Printf("[%.4f] Response %s, err %#v\n", bootstrap.Now(), pong, err);
}
```

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

Example run of PING

```
[0.0000] Starting
[0.4600] 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

```
  fmt.Printf("[%.4f] Starting\n", bootstrap.Now());

  sleep1, err := redis.Do("DEBUG", "SLEEP", "0.1");
  fmt.Printf("[%.4f] End Sleep 100ms, result %s err %v\n",
    bootstrap.Now(), sleep1, err);

  sleep2, err := redis.Do("DEBUG", "SLEEP", "0.2");
  fmt.Printf("[%.4f] End Sleep 200ms, result %s err %v\n",
    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

```
[0.0000] Starting
[100.7908] End Sleep 100ms, result OK err <nil>
[302.0737] 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 pe\
r run).
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:

```
#!/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:

```
$ go get github.com/youtube/vitess/go/pools
$ go get 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

```
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, write\
Timeout);
      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.

```
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

```
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:

```
$ go get 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:

```
sth = dbh.prepare("SELECT * FROM EMPLOYEE
                          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:

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

And the not-so-simple part:

Fetching a result set from the database

```
func showDatabases(conn *sql.DB, sql string) error {
  stmt, err := conn.Query(sql);
  if err != nil {
    return err;
  }
  defer stmt.Close();
  for stmt.Next() {
    var name string;
    if err := stmt.Scan(&name); err != nil {
      log.Fatal(err);
    }
    fmt.Printf("Database: %s\n", name);
  }
  return nil;
}
```

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:

```
$ go get github.com/jmoiron/sqlx
```

And now let's quickly adapt our example:

Fetching a result set from the database, redux

```
type Database struct {
  Name string `db:"Database"`;
}

func main() {
  db, err := sqlx.Open("mysql", "api:api@tcp(db1:3306)/api");
  if err != nil {
    log.Fatal("Error when connecting: ", err);
  }
  databases := []Database{};
  err = db.Select(&databases, "show databases");
  if err != nil {
    log.Fatal("Error in query: ", err);
  }
  spew.Dump(databases);
}
```

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:

```
pool := bootstrap.SqlxConnectionPool();
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.

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

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 behavior 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

```
1   go func() {
2     db, err := bootstrap.SqlxGetConnection();
3     if err != nil {
4       log.Fatal("Error when connecting: ", err);
5     }
6     defer bootstrap.SqlxReleaseConnection(db);
7
8     fmt.Printf("[%.4f] Run sleep 100ms\n", bootstrap.Now());
9
10    fromSleep := SleepResult{};
11    err = db.Get(&fromSleep, "select sleep(0.1) as sleepfor");
12    if err != nil {
13      sleep1_chan <- "ERROR";
14      return;
15    }
16    sleep1_chan <- fromSleep.Result;
17  }();
```

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 honorable 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 trough 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:

```
#!/bin/bash
NAME="db1"
DOCKERFILE="percona:latest"

if [ ! -d "/src/$NAME/data" ]; then
    mkdir -p /src/$NAME/data
fi

docker rm -f $NAME
docker run --restart=always \
    -h $NAME \
    --name $NAME \
    --net=party \
    -v /src/$NAME/conf/conf.d:/etc/mysql/conf.d \
    -v /src/$NAME/data:/var/lib/mysql \
    -e MYSQL_ROOT_PASSWORD=$PASSWORD \
    -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

```
go get github.com/garyburd/redigo/redis
go get github.com/youtube/vitess/go/pools
go get 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

```
type Registry struct {
  Name string;
}
func (r Registry) GetKey(key string) string {
  return r.Name + ":" + key;
}
func (r Registry) Get(key string) (string, error) {
  k := r.GetKey(key);
  return redis.String(bootstrap.RedigoDo("GET", k));
}
func (r Registry) Del(key string) (interface{}, error) {
  k := r.GetKey(key);
  return bootstrap.RedigoDo("DEL", k);
}
func (r Registry) Set(key string) (interface{}, error) {
  k := r.GetKey(key);
  return bootstrap.RedigoDo("SET", k);
}
```

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

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

  reg := Registry{ Name: "test" };
  reg.Del("name");
  val, err := reg.Get("name");
  if err == nil || val != "" {
    t.Errorf("Unexpected result when getting name: %s/%s\n", val, err);
  }
}
```

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 occured. 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

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

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 GET-ing 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

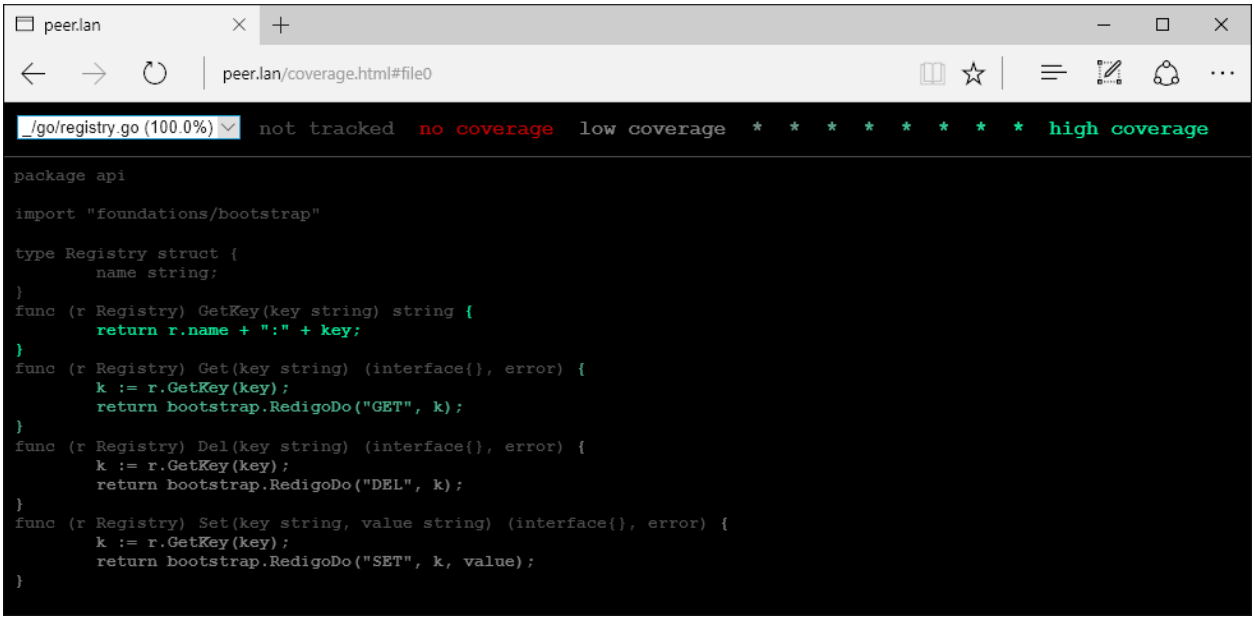
We can display coverage by function, letting us know which functions are not yet tested, or need more complete tests.

```
$ 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 color.

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 intruduce 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 behavior 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 occured 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

```
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;
}
```

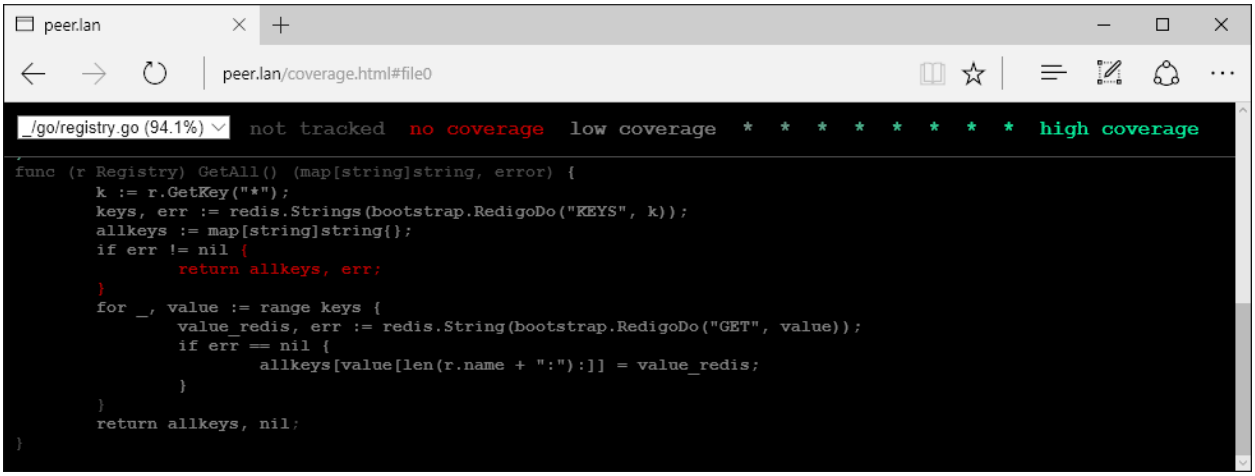
The function uses the KEYS method of Redis to loop trough the namespace and return all the keys with our Registry name (prefix). All the available keys are put in a map[string]string (array). We can check if we have the "name" index set in our test:

Testing GetAll output

```
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\
9  , err);
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.

```
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

```
func respondWithError(w http.ResponseWriter, err error) {
  response := map[string]string{};
  response["error"] = fmt.Sprintf("%s", err);
  response_json, _ := json.MarshalIndent(response, "", "\t");
  fmt.Fprintf(w, string(response_json[:]));
}
func respondWith(w http.ResponseWriter, response interface{}) {
  response_json, _ := json.MarshalIndent(response, "", "\t");
  fmt.Fprintf(w, string(response_json[:]));
}
func respond(w http.ResponseWriter, response interface{}, err error) {
  if err != nil {
    respondWithError(w, err);
    return;
  }
  respondWith(w, response);
}
```

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

```
  apiService = api.Registry{ Name: "api" };
  http.HandleFunc("/getAll", func (w http.ResponseWriter, r *http.Request) {
    response, err := apiService.GetAll();
    respond(w, response, err);
  });
  http.HandleFunc("/get", func (w http.ResponseWriter, r *http.Request) {
    key := r.FormValue("key");
    response, err := apiService.Get(key);
    respond(w, response, err);
  });
  http.HandleFunc("/set", func (w http.ResponseWriter, r *http.Request) {
    key := r.FormValue("key");
    value := r.FormValue("value");
    response, err := apiService.Set(key, value);
    respond(w, response, err);
  });
```

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 reponse 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[6] which generates a flame graph in svg format. To install go-torch, along with the FlameGraph dependecy, issue the following commands:

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

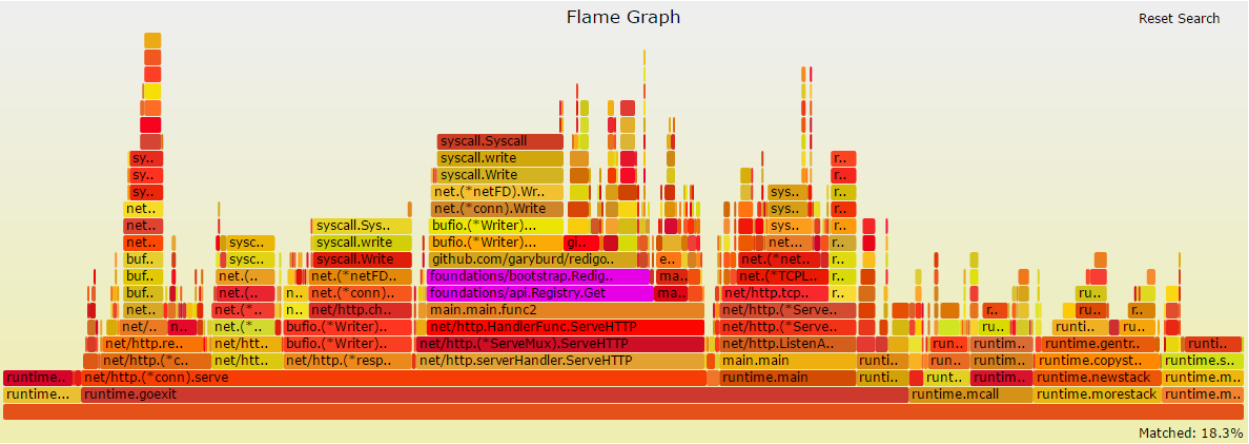
```
$ go get github.com/uber/go-torch
$ git clone --depth=1 https://github.com/brendangregg/FlameGraph.git /opt/flameg\
raph
```

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

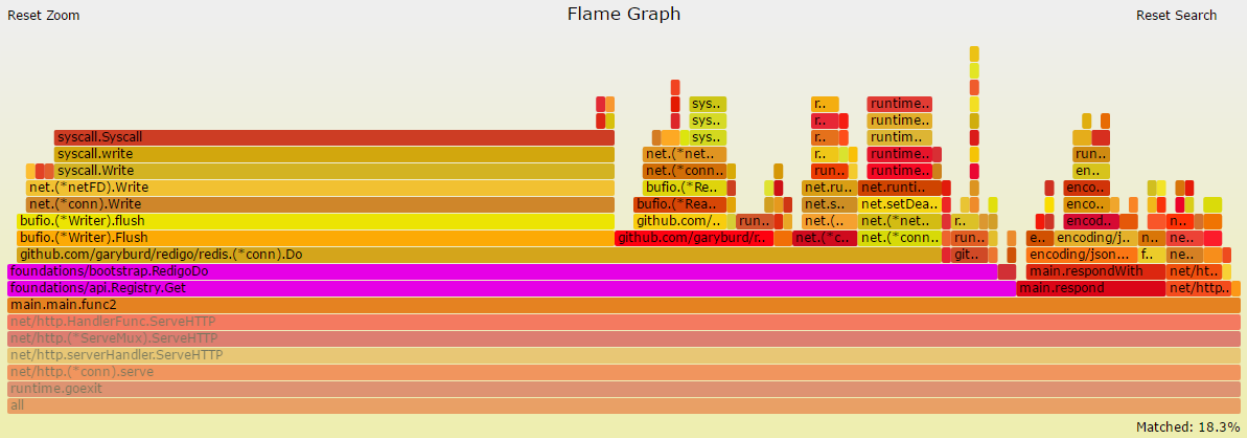
```
1  #!/bin/bash
2  PATH="$PATH:/opt/flamegraph"
3  bin/go-torch --time=15 --file "torch.svg" --url http://10.1.1.2:8080
```

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[7]. Because we follow the Twelve-Factor app paradigm, we want the configuration to come from the environment. The package namsral/flag[8] provides a drop in replacement for the Go flag package.

Installing configuration dependencies

```
go get github.com/namsral/flag
```

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:])
```

[7] https://golang.org/pkg/flag/
[8] https://github.com/namsral/flag

```
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 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

```
1  #!/bin/bash
2  set | grep GO_ > /tmp/docker.env
3  docker run --rm --env-file /tmp/docker.env -i -v `pwd`:/go golang 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/

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),
2. Docker images can be transfered 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[9] 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.

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

Example Dockerfile for our app

```
FROM alpine:latest

ADD flags /flags
RUN chmod +x /flags ; sync; sleep 1

WORKDIR /

ENV GO_PORT 8080
EXPOSE $GO_PORT

ENTRYPOINT ["/flags"]
```

Building the Docker image is a one liner -

Build Go application and create docker image

```
#!/bin/bash
go build flags.go
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                                                                  \
   100% 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[10] or OpenTSDB[11]. We use also a few others like Ganglia[12], Munin[13] and most recently, netdata[14]. 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.

[10] http://grafana.org/
[11] http://opentsdb.net/
[12] http://ganglia.info/
[13] http://munin-monitoring.org/
[14] https://github.com/firehol/netdata