Kevin Hoffman
Dan Nemeth

Cloud Native Go

Building Web Applications and Microservices for the Cloud with Go and React

Hong Kong Skyline & Harbor

The cover image, by Lee Yiu Tung, shows a portion of the Hong Kong skyline and harbor. According to The Skyscraper Center, Hong Kong is home to 315 buildings at least 150 meters in height: more than any other city on Earth. Nearly three-fourths of Hong Kong's skyscrapers are residential, helping to explain why more residents live above the 14th floor than in any other city. Hong Kong's tallest building, the International Commerce Centre, is 484 meters high—more than 40 meters taller than the tip of the Empire State Building's spire. At night, during good weather, visitors can experience "A Symphony of Lights," a light and laser show incorporating dozens of buildings on each side of Hong Kong's Victoria Harbor. The Harbor itself—still named after Britain's Queen Victoria nearly 20 years after Hong Kong was restored to China—holds 263 islands, as well as watercraft ranging from cargo freighters to cruise ships, and tourist ferries to traditional Chinese sampans and junks.

Cloud Native Go

Building Web Applications and Microservices for the Cloud with Go and React

Kevin Hoffman
Dan Nemeth

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi
Mexico City • São Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2016956519

ISBN-13: 978-0-672-33779-6

ISBN-10: 0-672-33779-7

First printing: December 2016

❖

This book is dedicated to the A-Team. Four men, sent to Pivotal for crimes they didn't commit, who now roam the countryside in search of developers in need of guidance: innocent people who need help moving their software to the cloud. If you need cloud apps, they will find you.

Without these brave men, the act of writing software would have become so boring and unbearable that this book would never have been written. In fact, the authors may have given up their lives of service to the cloud, only to while away their remaining days as baristas in a smelly hipster coffee shop.

The A-Team is:

Dan "Hannibal" Nemeth

Chris "Murdock" Umbel

Tom "Face" Collings

Kevin "B.A." Hoffman

❖

Contents at a Glance

Contents

Preface

When Dan and I set out to write this book, we didn't want it to be a reference book or "yet another syntax book." Instead, we wanted to put to good use our experience building cloud native solutions for Pivotal customers and nearly a lifetime of combined experience building software for companies of just about every size, shape, and industry.

This book starts off with a philosophical chapter, *The Way of the Cloud*, because we firmly believe that the secret to building good software has more to do with the mindset and discipline of the developers than it does the tooling or language.

From there, we follow The Way of the Cloud in everything we do as we gradually, in a test-driven and highly automated fashion, take you through a series of chapters designed to increase your skills building cloud native services in Go. We cover the fundamentals of building services; middleware; the use of tools like git, Docker, and Wercker; and cloud native fundamentals like environment-based configuration, service discovery, and reactive and push-based applications. We cover patterns like Event Sourcing and CQRS, and combine everything in the book into a final sample that you can use as inspiration for your own projects.

Another of our strongly-held beliefs is that the act of building a piece of software should be as fun (or more!) as using that software. If it's not fun, you're doing it wrong. We wanted the joy we get from building services in Go to infect our readers, and hopefully you will have as much fun reading this book as we did writing it.

About the Authors

Kevin Hoffman helps enterprises bring their legacy applications onto the cloud through modernization and building cloud native services in many different languages. He started programming when he was 10 years old, teaching himself BASIC on a rebuilt Commodore VIC-20. Since then, he has been addicted to the art of building software, and has spent as much time as he can learning languages, frameworks, and patterns. He has built everything from software that remotely controls photography drones to biometric security, ultra-low-latency financial applications, mobile applications, and everything between. He fell in love with the Go language while building custom components for use with Pivotal Cloud Foundry.

Kevin is the author of a popular series of fantasy books (*The Sigilord Chronicles*, http://amzn.to/2fc8iES) and is eagerly awaiting the day when he will finally be able to combine his love for building software with his love for building fictional worlds.

Dan Nemeth currently works at Pivotal as an Advisory Solutions Architect, supporting Pivotal Cloud Foundry. He has been writing software since the days of the Commodore 64. He began coding professionally in 1995 for a local ISP writing CGI scripts in ANSI C. Since then, he has spent the majority of his career as an independent consultant building solutions for industries ranging from finance to pharmaceutical, and using various languages/frameworks that were vogue at the time. Dan has recently embraced Go as a homecoming, of sorts, and is enthusiastically using it for all of his projects.

Should you find Dan away from his computer, he will likely be on the waters near Annapolis either sailing or fly fishing.

Acknowledgments

This book would not have been possible without the nearly infinite patience of my family, especially my wife. Despite me having said on numerous occasions in the past that I would never again write a technical book, here I was, writing another tech book. They put up with long nights, me wearing ruts into the floor pacing, and the loss of quality time in order to get this book finished. I am more proud of this book than I am of any other I've worked on in the past, and that is directly related to the invaluable support of family, friends, and a brilliant co-author.

—Kevin Hoffman

5

Building Microservices in Go

"The golden rule: can you make a change to a service and deploy it by itself without changing anything else?"

Sam Newman, *Building Microservices*

Every service you build should be a microservice, and, as we've discussed earlier in the book, we generally disagree with using the prefix *micro* at all. In this chapter we're going to be building a service, but this chapter is as much about the process as it is about the end result.

We'll start by following the practice of **API First**, designing our service's RESTful contract before we write a single line of code. Then, when it does come time to write code, we're going to start by writing *tests* first. By writing small tests that go from failure to passing, we will gradually build out our service.

The sample service we're going to build in this chapter is a server implementation of the game of Go. This service will be designed to enable clients of any kind to participate in matches of Go, from iPhones to browsers to other services.

Most importantly, this service needs a name. A service written in Go that resolves matches of the game of Go can be called nothing less than **GoGo**.

In this chapter, we're going to cover:

- API First development disciplines and practices.
- Creating the scaffolding for a microservice.
- Adding tests to a scaffolded service and iterating through adding code to make tests pass.
- Deploying and running a microservice in the cloud.

Designing Services API First

In this next section we're going to design our microservice. One of the classic problems of software development is that what you design is rarely ever what you end up developing. There is always a gap between documentation, requirements, and implementation.

Thankfully, as you'll see, there are some tools available to use for microservice development that actually allow a situation where *the design is the documentation,* which can then be integrated into the development process.

Designing the Matches API

The first thing that we're going to need if we're creating a service that hosts matches is a resource collection for matches. With this collection, we should be able to create a new match as well as list all of the matches currently being managed by the server shown in Table 5.1.

Table 5.1 **The Matches API**

Resource	Method	Description
`/matches`	GET	Queries a list of all available matches.
`/matches`	POST	Creates and starts a new match.
`/matches/`*`{id}`*	GET	Queries the details for an individual match.

If we were building a game of Go that we were hoping to sell for real money, rather than as a sample, we would also implement methods to allow a UI to query things like **chains** and **liberties**, concepts essential to determining legal moves in Go.

Designing the Moves API

Once the service is set up to handle matches, we need to expose an API to let players make moves. This adds the following HTTP methods to the `moves` sub-resource as shown in Table 5.2.

Table 5.2 **The Moves API**

Resource	Method	Description
`/matches/{id}/moves`	GET	Returns a time-ordered list of all moves taken during the match.
`/matches/{id}/moves`	POST	Make a move. A move without a position is a pass.

Creating an API Blueprint

In our desire to simplify everything we do, some time ago we started to eschew complex or cumbersome forms of documentation. Do we really need to share monstrous document files that carry with them decades of backwards compatibility requirements?

For us, **Markdown**[1] is the preferred form of creating documentation and doing countless other things. It is a simple, plain text format that requires no IDE or bloated editing tool, and it can be converted and processed into countless formats from PDF to web sites. As with so many things, the debate over which format people use for documentation has been known to spark massive, blood-soaked inter-office battles.

As a matter of habit, we typically create Markdown documents that we bundle along with our services. This allows other developers to quickly get a list of all of our service's REST resources, the URI patterns, and request/response payloads. As simple as our Go code is, we still wanted a way to document the service contract without making someone go sifting through our router code.

As it turns out, there is a dialect of Markdown used specifically for documenting RESTful APIs: **API Blueprint**. You can get started reading up on this format at the API Blueprint website https://apiblueprint.org/.

If you check out the GitHub repository for this chapter (https://github.com/cloudnativego/gogo-service), you'll see a file called `apiary.apib`. This file consists of Markdown that represents the documentation and specification of the RESTful contract supported by the GoGo service.

Listing 5.1 below shows a sample of the Markdown content. You can see how it describes REST resources, HTTP methods, and JSON payloads.

Listing 5.1 **Sample Blueprint Markdown**

```
### Start a New Match [POST]

You can create a new match with this action. It takes information about the players
 and will set up a new game. The game will start at round 1, and it will be
 **black**'s turn to play. Per standard Go rules, **black** plays first.

+ Request (application/json)

        {
            "gridsize" : 19,
            "players" : [
            {
                "color" : "white",
                "name" : "bob"
            },
            {
                "color" : "black",
                "name" : "alfred"
            }
            ]
        }
```

1 Links to references on Markdown syntax can be found here: https://en.wikipedia.org/wiki/Markdown.

```
+ Response 201 (application/json)

    + Headers

            Location: /matches/5a003b78-409e-4452-b456-a6f0dcee05bd

    + Body

            {
                "id" : "5a003b78-409e-4452-b456-a6f0dcee05bd",
                "started_at": "2015-08-05T08:40:51.620Z",
                "gridsize" : 19,
                "turn" : 0,
                "players" : [
                    {
                        "color" : "white",
                        "name" : "bob",
                        "score" : 10
                    },
                    {
                        "color" : "black",
                        "name" : "alfred",
                        "score" : 22
                    }
                ]

            }
```

Testing and Publishing Documentation with Apiary

In Chapter 1, *The Way of the Cloud*, we cautioned against relying too heavily on tools. Tools should make your life easier, but they should never be mandatory. The API Blueprint Markdown that contains the documentation and specification for our service is just a simple text file, however, there is a tool that can do a *lot* to make our lives both easier and more productive.

Apiary is a website that lets you interactively design your RESTful API. You can think of it as a WYSIWYG editor for API Blueprint Markdown syntax, but that's just the beginning. Apiary will also set up mock server endpoints for you that return sample JSON payloads. This saves you the trouble of having to build your own mock server, and lets you remain in API First mode until after you've gone through the motions of exercising various rough drafts of your API.

In addition to exposing mock server endpoints, you can also see client code in a multitude of languages that exercises your API, further assisting you and your team in validating your API—all before you have to write a single line of server code.

The API Blueprint document for the GoGo service is available in our GitHub repository as well as on Apiary for viewing at http://docs.gogame.apiary.io/. Rather than dump the entire set of documentation into the book, we'll leave most of the details in the blueprint document and on Apiary for you to read on your own.

The purpose of this chapter isn't to teach you how to make a game server, but to teach you the process of building a service in the Go language, so details like the rules of Go and actual game implementation will be secondary to things like Test-Driven Development and setting up a service scaffold, which we'll cover next.

Scaffolding a Microservice

In a perfect world, we would start with a completely blank slate and go directly into testing. The problem with ideal, perfect worlds is they rarely ever exist. In our case, we want to be able to write tests for our RESTful endpoints.

The reality of the situation is we can't really write a test for RESTful endpoints unless we know what kind of functions we're going to be writing per endpoint. To figure this out, and to get a basic scaffolding for our service set up, we're going to create two files.

The first file, `main.go` (Listing 5.2), contains our main function, and creates and runs a new server. We want to keep our main function as small as possible because the main function is usually notoriously hard to test in isolation.

Listing 5.2 **main.go**

```
package main

import (
  "os"
  service "github.com/cloudnativego/gogo-service/service"
)

func main() {
  port := os.Getenv("PORT")
  if len(port) == 0 {
    port = "3000"
  }

  server := service.NewServer()
  server.Run(":" + port)
}
```

The code in Listing 5.2 invokes a function called `NewServer`. This function returns a pointer to a Negroni struct. Negroni is a third-party library for building routed endpoints on top of Go's built-in `net/http` package.

It is also important to note the bolded line of code. External configuration is crucial to your ability to build cloud native applications. By allowing your application to accept its bound port from an environment variable, you're taking the first step toward building a service that will work in the cloud. We also happen to know that a number of cloud providers automatically inject the application port using this exact environment variable.

Listing 5.2 shows our server implementation. In this code we're creating and configuring Negroni in **classic** mode, and we're using Gorilla Mux for our routing library. As a rule, we treat any third party dependency with skepticism, and must justify the inclusion of everything that isn't part of the core Go language.

In the case of Negroni and Mux, these two play very nicely on top of Go's stock `net/http` implementation, and are extensible pieces of middleware that don't interfere with anything we might want to do in the future. Nothing there is mandatory; there is no "magic", just some libraries that make our lives easier so we don't spend so much time writing boilerplate with each service.

For information on Negroni, check out the GitHub repo https://github.com/codegangsta/negroni. And for information on Gorilla Mux, check out that repo at https://github.com/gorilla/mux. Note that these are the same URLs that we import directly in our code, which makes it extremely easy to track down documentation and source code for third-party packages.

Listing 5.3 shows the `NewServer` function referenced by our `main` function and some utility functions. Note that `NewServer` is exported by virtue of its capitalization and functions like `initRoutes` and `testHandler` are not.

Listing 5.3 **server.go**

```
package service

import (
	"net/http"

	"github.com/codegangsta/negroni"
	"github.com/gorilla/mux"
	"github.com/unrolled/render"
)

// NewServer configures and returns a Server.
func NewServer() *negroni.Negroni {

	formatter := render.New(render.Options{
		IndentJSON: true,
	})

	n := negroni.Classic()
	mx := mux.NewRouter()

	initRoutes(mx, formatter)
```

```
        n.UseHandler(mx)
        return n
}

func initRoutes(mx *mux.Router, formatter *render.Render) {
        mx.HandleFunc("/test", testHandler(formatter)).Methods("GET")
}

func testHandler(formatter *render.Render) http.HandlerFunc {

        return func(w http.ResponseWriter, req *http.Request) {
                formatter.JSON(w, http.StatusOK,
                 struct{ Test string }{"This is a test"})
        }
}
```

The most important thing to understand in this scaffolding is the `testHandler` function. Unlike regular functions we've been using up to this point, this function returns an anonymous function.

This anonymous function, in turn, returns a function of type `http.HandlerFunc`, which is defined as follows:

```
type HandlerFunc func(ResponseWriter, *Request)
```

This type definition essentially allows us to treat any function with this signature as an HTTP handler. You'll find this type of pattern used throughout Go's core packages and in many third-party packages.

For our simple scaffolding, we return a function that places an anonymous struct onto the response writer by invoking the `formatter.JSON` method (this is why we pass the formatter to the wrapper function).

The reason this is important is because all of our RESTful endpoints for our service are going to be wrapper functions that return functions of type `http.HandlerFunc`.

Before we get to writing our tests, let's make sure that the scaffolding works and that we can exercise our test resource. To build, we can issue the following command (your mileage may vary with Windows):

```
$ go build
```

This builds all the Go files in the folder. Once you've created an executable file, we can just run the GoGo service:

```
$ ./gogo-service
[negroni] listening on :3000
```

When we hit `http://localhost:3000/test` we get our test JSON in the browser, and we see that because we've enabled the classic configuration in Negroni, we get some nice logging of HTTP request handling:

```
[negroni] Started GET /test
[negroni] Completed 200 OK in 212.121µs
```

Now that we know our scaffolding works, and we have at least a functioning web server capable of handling simple requests, it's time to do some real Test-Driven Development.

Building Services Test First

It's pretty easy to talk about TDD, but, despite countless blogs and books extolling its virtues, it is still pretty rare to find people who practice it regularly. It is even rarer still to find people who practice it without cutting corners. Cutting corners in TDD is the worst of both worlds—you're spending the time and effort on TDD but you're not reaping the benefits of code quality and functional confidence.

In this section of the chapter, we're going to write a method for our service in test-first fashion. If we're doing it right, it should feel like we're spending 95% of our time writing tests, and 5% of our time writing code. The size of our test should be *significantly* larger than the size of the code we're testing. Some of this just comes from the fact that it takes more code to exercise all possible paths through a function under test than it does to write the function itself. For more details on this concept, check out the book *Continuous Delivery* by Jez Humble & David Farley.

Many organizations view the effort to write tests as wasteful, claiming that it does not add value and actually increases time-to-market. There are a number of problems with this myopic claim.

It is true that TDD will, indeed, slow initial development. However, let's consider a new definition of the term development:

> development(n) : The period where the features of the application are being added without the so-called burden of a running version of it in production.
>
> Dan Nemeth

With this definition in mind when we look at the entire life cycle of an application, only for a very small portion of that time is the application ever in this state of "development".

Investment in testing will pay dividends throughout the entire life cycle of the application, but especially in production where:

- Uptime is a must.
- Satisfying change/feature requests is urgent.
- Debugging is costly, difficult, and oftentimes approaching impossible.

To get started on our own TDD journey of service creation, let's create a file called `handlers_test.go` (shown in Listing 5.4). This file is going to test functions written in the `handlers.go` file. If your favorite text editor has a side-by-side or split-screen mode, this would be a great time to use it.

We're going to be writing a test for the HTTP handler invoked when someone POSTs a request to start a new match. If we check back with our Apiary documentation, we'll see that one of the requirements is that this function return an HTTP status code of ***201 (Created)*** when successful.

Let's write a test for this. We'll call the function `TestCreateMatch` and, as with all Go unit tests using the basic unit testing package, it will take as a parameter a pointer to a `testing.T` struct.

Creating a First, Failing Test

In order to test our server's ability to create matches, we need to invoke the HTTP handler. We could invoke this manually by fabricating all of the various components of the HTTP pipeline, including the request and response streams, headers, etc. Thankfully, though, Go provides us with a test HTTP server. This doesn't open up a socket, but it does all the other work we need it to do, which lets us invoke HTTP handlers.

There is a lot going on here, so let's look at the full listing (Listing 5.4) for the test file in our first iteration, which, in keeping with TDD ideology, is a *failing* test.

Listing 5.4 **handlers_test.go**

```
package main

import (
	"bytes"
	"fmt"
	"io/ioutil"
	"net/http"
	"net/http/httptest"
	"testing"

	"github.com/unrolled/render"
)

var (
	formatter = render.New(render.Options{
		IndentJSON: true,
	})
)
```

```
func TestCreateMatch(t *testing.T) {
        client := &http.Client{}
        server := httptest.NewServer(
           http.HandlerFunc(createMatchHandler(formatter)))
        defer server.Close()

        body := []byte("{\n  \"gridsize\": 19,\n  \"players\": [\n    {\n
        \"color\": \"white\",\n      \"name\": \"bob\"\n    },\n    {\n
        \"color\": \"black\",\n      \"name\": \"alfred\"\n    }\n  ]\n}")

        req, err := http.NewRequest("POST",
               server.URL, bytes.NewBuffer(body))
        if err != nil {
                t.Errorf("Error in creating POST request for createMatchHandler: %v",
                err)
        }
        req.Header.Add("Content-Type", "application/json")

        res, err := client.Do(req)
        if err != nil {
                t.Errorf("Error in POST to createMatchHandler: %v", err)
        }

        defer res.Body.Close()

        payload, err := ioutil.ReadAll(res.Body)
        if err != nil {
                t.Errorf("Error reading response body: %v", err)
        }

        if res.StatusCode != http.StatusCreated {
                t.Errorf("Expected response status 201, received %s",
                        res.Status)
        }

        fmt.Printf("Payload: %s", string(payload))
}
```

Here's another reason why we like Apiary so much: if you go to the documentation for the ***create match*** functionality and click on that method, you'll see that it can actually generate sample client code in Go. Much of that generated code is used in the preceding test method in Listing 5.3.

The first thing we do is call `httptest.NewServer`, which creates an HTTP server listening at a custom URL that will serve up the supplied method. After that, we are using most of Apiary's sample client code to invoke this method.

We have two main assertions here:

- We do not receive any errors when executing the request and reading the response bytes
- The response status code is ***201 (Created)***.

If we were to try and run the test above, we would get a compilation failure. This is true TDD, because we haven't even written the method we're testing (`createMatchHandler` doesn't exist yet). To get the test to compile, we can add a copy of our original scaffold test method to our **handlers.go** file as shown in Listing 5.5:

Listing 5.5 **handlers.go**

```
package main

import (
        "net/http"

        "github.com/unrolled/render"
)

func createMatchHandler(formatter *render.Render) http.HandlerFunc {
        return func(w http.ResponseWriter, req *http.Request) {
                formatter.JSON(w,
                 http.StatusOK,
                 struct{ Test string }{"This is a test"})
        }
}
```

Now we can see what happens when we try and test this. First, to test we issue the following command:

```
$ go test -v $(glide novendor)
```

We should see the following output:

```
Expected response status 201, received 200 OK
```

Now we've written our first failing test! At this point, some of you may be starting to doubt these methods. If so, please bear with us; we promise that by the end of the chapter you will have seen the light.

Let's make this failing test a passing one. To make it pass, *all* we do is make the HTTP handler return a status of 201. We don't write the full implementation, we don't add complex logic. The *only* thing we do is make the test pass. It is vitally important to the process that *we only write the minimum code necessary to make the test pass*. If we write code that isn't necessary for the test to pass, we're no longer in *test-first* mode.

To make the test pass, change the formatter line in `handlers.go` to as follows:

```
formatter.JSON(w, http.StatusCreated, struct{ Test string }{"This is a test"})
```

We just changed the second parameter to `http.StatusCreated`. Now when we run our test, we should see something similar to the following output:

```
$ go test -v $(glide novendor)
=== RUN   TestCreateMatch
--- PASS: TestCreateMatch (0.00s)
PASS
ok      github.com/cloudnativego/gogo-service     0.011s
```

Testing the Location Header

The next thing that we know our service needs to do in response to a ***create match*** request (as stated in our Apiary documentation) is to set the ***Location*** header in the HTTP response. By convention, when a RESTful service creates something, the ***Location*** header should be set to the URL of the newly created thing.

As usual, we start with a failing test condition and then we make it pass.

Let's add the following assertion to our test:

```
if _, ok := res.Header["Location"]; !ok {
  t.Error("Location header is not set")
}
```

Now if we run our test again, we will fail with the above error message. To make the test pass, modify the `createMatchHandler` method in `handlers.go` to look like this:

```
func createMatchHandler(formatter *render.Render) http.HandlerFunc {
        return func(w http.ResponseWriter, req *http.Request) {
                w.Header().Add("Location", "some value")
                formatter.JSON(w, http.StatusCreated,
                        struct{ Test string }{"This is a test"})
        }
}
```

Note that we didn't add a *real* value to that location. Instead, we just added *some* value. Next, we'll add a failing condition that tests that we get a valid location header that contains the `matches` resource and is long enough so that we know it also includes the GUID for the newly created match. We'll modify our previous test for the location header so the code looks like this:

```
        loc, headerOk := res.Header["Location"]
        if !headerOk {
                t.Error("Location header is not set")
        } else {
                if !strings.Contains(loc[0], "/matches/") {
                        t.Errorf("Location header should contain '/matches/'")
                }
                if len(loc[0]) != len(fakeMatchLocationResult) {
                        t.Errorf("Location value does not contain guid of new match")
                }
        }
}
```

We've also added a constant to the test called `fakeMatchLocationResult`, which is just a string that we also pulled off of Apiary representing a test value for the location header. We'll use this for test assertions and fakes. This is defined as follows:

```
const (
   fakeMatchLocationResult = "/matches/5a003b78-409e-4452-b456-a6f0dcee05bd"
)
```

Epic Montage—Test Iterations

Since we have limited space in this book, we don't want to dump the code for every single change we made during every iteration where we went from red (failing) to green (passing) light in our testing.

Instead, we'll describe what we did in each TDD pass we made:

- Wrote a failing test.
- Made the failing test pass.
- Checked in the results.

If you want to examine the history so you can sift through the changes we made line-by-line, check out the commit history in GitHub. Look for commits labelled "TDD GoGo service Pass *n*" where *n* is the testing iteration number.

We've summarized the approaches we took for each failed test and what the resolution was to make the test pass in the following list of steps, so cue up your favorite Hollywood hacker movie montage background music and read on:

1. **TDD Pass 1.** We created the initial setup required to host a test HTTP server that invokes our HTTP handler method (the method under test). This test initially failed because of compilation failure—the method being tested did not yet exist. We got the test to pass by dumping the test resource code into the `createMatchHandler` method.
2. **TDD Pass 2.** Added an assertion that the result included a ***Location*** header in the HTTP response. This test initially failed, so we added a placeholder value in the location header.
3. **TDD Pass 3.** Added an assertion that the ***Location*** header was actually a properly formatted URL pointing at a match identified by a GUID. The test initially failed, so we made it pass by generating a new GUID and setting a proper location header.
4. **TDD Pass 4.** Added an assertion that the ***ID*** of the match in the response payload matched the GUID in the location header. This test initially failed and, to make it pass, we had to add code that un-marshaled the response payload in the test. This meant we actually had to create a struct that represented the response payload on the server. We stopped returning "this is a test" in the handler and now actually return a real response object.

5. **TDD Pass 5.** Added an assertion that the repository used by the handler function has been updated to include the newly created match. To do this, we had to create a repository interface and an in-memory repository implementation.
6. **TDD Pass 6.** Added an assertion that the grid size in the service response was the same as the grid size in the match added to the repository. This forced us to create a new struct for the response, and to make several updates. We also updated another library, `gogo-engine`, which contains minimal Go game resolution logic that should remain mostly isolated from the service.
7. **TDD Pass 7.** Added assertions to test that the players we submitted in the new match request are the ones we got back in the service JSON reply and they are also reflected accordingly in the repository.
8. **TDD Pass 8.** Added assertions to test that if we send something other than JSON, or we fail to send reasonable values for a new match request, the server responds with a ***Bad Request*** code. These assertions fail, so we went into the handler and added tests for JSON un-marshaling failures as well as invalid request objects. Go is pretty carefree about JSON de-serialization, so we catch most of our "bad request" inputs by checking for omitted or default values in the de-serialized struct.

Let's take a breather and look at where things stand after this set of iterations. Listing 5.6 shows the one handler that we have been developing using TDD, iterating through successive test failures which are then made to pass by writing code. To clarify, *we never write code unless it is in service of making a test pass*. This essentially guarantees us the maximum amount of test coverage and confidence possible.

This is a really hard line for many developers and organizations to take, but we think it's worth it and have seen the benefits exhibited by real applications deployed in the cloud.

Listing 5.6 **handlers.go (after 8 TDD iterations)**

```
package service

import (
        "encoding/json"
        "io/ioutil"
        "net/http"

        "github.com/cloudnativego/gogo-engine"
        "github.com/unrolled/render"
)

func createMatchHandler(formatter *render.Render, repo matchRepository)
     http.HandlerFunc {
        return func(w http.ResponseWriter, req *http.Request) {
          payload, _ := ioutil.ReadAll(req.Body)
          var newMatchRequest newMatchRequest
          err := json.Unmarshal(payload, &newMatchRequest)
          if err != nil {
            formatter.Text(w, http.StatusBadRequest,
              "Failed to parse create match request")
            return
           }
           if !newMatchRequest.isValid() {
             formatter.Text(w, http.StatusBadRequest,
               "Invalid new match request")
             return
           }

           newMatch := gogo.NewMatch(newMatchRequest.GridSize,
             newMatchRequest.PlayerBlack, newMatchRequest.PlayerWhite)
           repo.addMatch(newMatch)
           w.Header().Add("Location", "/matches/"+newMatch.ID)
           formatter.JSON(w, http.StatusCreated,
              &newMatchResponse{ID: newMatch.ID,
                         GridSize: newMatch.GridSize,
                          PlayerBlack: newMatchRequest.PlayerBlack,
                         PlayerWhite: newMatchRequest.PlayerWhite})
        }
}
```

While Go's formatting guidelines generally call for an 8-character tab, we've condensed some of that to make the listing a little more readable here.

We have about 20 lines of code in a single function, and we have about 120 lines of code in the two test methods that exercise that code. This is exactly the type of ratio we want. Before we even open a single HTTP test tool to play with our service, we want to have 100% confidence and know exactly how our service should behave.

Based on the tests that we've written thus far, and the code in Listing 5.6, can you spot any testing gaps? Can you see any scenarios or edge cases that might trip up our code that we have not yet accounted for in testing?

There are two glaring gaps that we see:

1. This service is not stateless. If it goes down, we lose all of our in-progress games. This is a known issue, and we're willing to let it slide because we have a crystal ball, and we know that Chapter 7 will address data persistence.
2. There are a number of abuse scenarios against which we are not guarding. Most notably, there is nothing to stop someone from rapidly creating game after game until we exceed our memory capacity and the service crashes. This particular abuse vector is a side-effect of us storing games in memory and us violating a cardinal rule of cloud native: statelessness. We're not going to write tests for this either because, as mentioned in #1, these conditions are temporary and writing DDoS-guarding code is a rabbit hole we want to avoid in this book.

We'll correct some of these as we progress throughout the book, but others, like guarding against all of the edge cases, are really going to be your responsibility as you build production-grade services.

Deploying and Running in the Cloud

Now that we've used Go to build a microservice while following *the way of the cloud*, we can put that effort to good use and deploy our work to the cloud. The first thing we're going to need is *a cloud*. While there are a number of options available to us, in this book we favor Cloud Foundry's PCF Dev and Pivotal Web Services (PWS) as deployment targets because they're both extremely easy to get started with and PWS has a free trial that *does not* require a credit card to get started.

Creating a PWS Account

Head over to http://run.pivotal.io/ to create an account with Pivotal Web Services. Pivotal Web Services is platform powered by Cloud Foundry that lets you deploy your applications in their cloud and take advantage of a number of free and paid services in their marketplace.

Once you've created an account and logged in, you will see the dashboard for your organization. An organization is a logical unit of security and deployment. You can invite other people to join your organization so you can collaborate on cloud projects, or you can keep all that cloudy goodness to yourself.

On the home page or dashboard for your organization, you will see a box giving you some helpful information, including links pointing you to the *Cloud Foundry CLI*. This is a command-line interface that you can use to push and configure your applications in *any* cloud foundry (not just PWS).

Download and install the CF CLI and make sure it works by running a few test commands such as `cf apps` or `cf spaces` to verify that you're connected and working. Remember that you have 60 days to play in the PWS sandbox without ever having to supply a credit card, so make sure you take full advantage of it.

For information on what you can do with the CF CLI, check out the documentation here http://docs.run.pivotal.io/devguide/cf-cli/.

Setting up PCF Dev

If you're more adventurous, or you simply like to tinker, then **PCF Dev** is the tool for you. Essentially, **PCF Dev** is a stripped-down version of Cloud Foundry that provides application developers all of the infrastructure necessary to deploy an application into a CF deployment, but without all of the production-level stuff that would normally prevent you from running a cloud on your laptop.

PCF Dev utilizes a virtual machine infrastructure (you can choose between VMware or VirtualBox) and a tool called ***vagrant*** to spin up a single, self-contained virtual machine that will play host to PCF Dev and your applications.

You can use PCF Dev to test how well your application behaves in the cloud without having to push to PWS. We've found it invaluable for testing things like service bindings and doing testing that falls somewhere between automated integration testing and full acceptance testing.

At the time this book is being written, PCF Dev is still in its early stages and, as a result, the instructions for installing and configuring the various releases are likely to change.

To get set up with PCF Dev, go to https://docs.pivotal.io/pcf-dev/.

The beauty of PCF Dev is that once you have the pre-requisites installed, you can simply issue the `start` command and everything you need will be brought up for you on your local virtualization infrastructure. For example, on OS X, you start your foundation with the `./start-osx` script.

Using the exact same Cloud Foundry CLI that you used to communicate with your PWS cloud, you can retarget that CLI to your new MicroPCF installation:

```
$ cf api api.local.pcfdev.io --skip-ssl-validation
Setting api endpoint to api.local.pcfdev.io...
OK

API endpoint:   https://api.local.pcfdev.io (API version: 2.44.0)
Not logged in. Use 'cf login' to log in.
```

Make sure you login as the instructions indicate (the default username and password are ***admin*** and ***admin***), and you can then issue standard Cloud Foundry CLI commands to communicate with your newly started local, private CF deployment:

```
$ cf apps
Getting apps in org local.pcfdev.io-org / space kev as admin...
OK
```

Pushing to Cloud Foundry

Now that you've got the CF CLI installed and you can choose whether your CLI is targeting the PWS cloud or your local PCF Dev installation, you can push your application and run it in the cloud.

While you can manually supply all of the various options that you need to push your application to the cloud, it's easier (and more compatible with the CD pipeline work we'll be doing later in the book) to create a **manifest** file, like the one in Listing 5.7.

Listing 5.7 **manifest.yml**

```
applications:
- path: .
  memory: 512MB
  instances: 1
  name: your-app-name
  disk_quota: 1024M
  command: your-app-binary-name
  buildpack: https://github.com/cloudfoundry/go-buildpack.git
```

With this manifest file in the main directory of your application, you can simply type the following command and your application will be deployed in the cloud.

```
$ cf push
```

As we'll also illustrate later in the book, you can even configure your Wercker pipeline to automatically deploy your application to the Cloud Foundry of your choice at the end of a successful build for continuous delivery.

A Note on the Go Buildpack

Buildpacks are designed to merge your application code with the underlying requirements necessary to run your app. The Java buildpack contains the JDK and the JRE, the Node buildpack contains `node`, etc. While the Go buildpack might suffice for tinkering, it is far too easy to violate the "single immutable artifact" rule with it. It's also possible that someone will commit a change to the buildpack that breaks your code or pipeline. As you'll see later in the book, when we deploy real apps, we are going to favor deploying our Docker images to the cloud directly from Docker Hub. The choice of buildback vs. Docker is entirely up to you and your organization and often boils down to simple personal preference.

Summary

In this chapter we illustrated the basics of building microservices in Go. We took a look at the code you need in order to set up basic routes and handlers, but more importantly, we showed you how to build this code *test-first*.

Further, we walked you through getting your code deployed into the cloud. The rest of the book is going to get more technical and explore more in-depth topics, so you may want to take a moment to review any of the content of this chapter you didn't quite understand before continuing on.

This would also be a great time to tinker a bit and create your own ***hello world*** services, deploy them to PWS and play with starting, stopping, and scaling your applications. You may also want to browse the *marketplace* in PWS to get an idea of the types of incredibly powerful services, including databases, message queues, and monitoring, that are available to the applications you deploy there.

Index