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.

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.