# JSON, routing and embedding

**You can find all the code for this chapter here**

In the previous chapter we created a web server to store how many games players have won.

Our product owner has a new requirement; to have a new endpoint called `/league` which returns a list of all players stored. She would like this to be returned as JSON.

## Here is the code we have so far

```go
// server.go
package main

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

type PlayerStore interface {
        GetPlayerScore(name string) int
        RecordWin(name string)
}

type PlayerServer struct {
        store PlayerStore
}

func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
        player := strings.TrimPrefix(r.URL.Path, "/players/")

        switch r.Method {
        case http.MethodPost:
                p.processWin(w, player)
        case http.MethodGet:
                p.showScore(w, player)
```

```
}       }

func (p *PlayerServer) showScore(w http.ResponseWriter, player string) {
        score := p.store.GetPlayerScore(player)

        if score == 0 {
                w.WriteHeader(http.StatusNotFound)
        }

        fmt.Fprint(w, score)
}

func (p *PlayerServer) processWin(w http.ResponseWriter, player string) {
        p.store.RecordWin(player)
        w.WriteHeader(http.StatusAccepted)
}
```

```
// in_memory_player_store.go
package main

func NewInMemoryPlayerStore() *InMemoryPlayerStore {
        return &InMemoryPlayerStore{map[string]int{}}
}

type InMemoryPlayerStore struct {
        store map[string]int
}

func (i *InMemoryPlayerStore) RecordWin(name string) {
        i.store[name]++
}

func (i *InMemoryPlayerStore) GetPlayerScore(name string) int {
        return i.store[name]
}
```

```
// main.go
package main

import (
```

```
        "log"
        "net/http"

)

func main() {
        server := &PlayerServer{NewInMemoryPlayerStore()}
        log.Fatal(http.ListenAndServe(":5000", server))
}
```

You can find the corresponding tests in the link at the top of the chapter.

We'll start by making the league table endpoint.

---

## Write the test first

We'll extend the existing suite as we have some useful test functions and a fake `PlayerStore` to use.

```
//server_test.go
func TestLeague(t *testing.T) {
        store := StubPlayerStore{}
        server := &PlayerServer{&store}

        t.Run("it returns 200 on /league", func(t *testing.T) {
                request, _ := http.NewRequest(http.MethodGet, "/league", nil)
                response := httptest.NewRecorder()

                server.ServeHTTP(response, request)

                assertStatus(t, response.Code, http.StatusOK)
        })
}
```

Before worrying about actual scores and JSON we will try and keep the changes small with the plan to iterate toward our goal. The simplest start is to check we can hit `/league` and get an `OK` back.

---

## Try to run the test

```
    --- FAIL: TestLeague/it_returns_200_on_/league (0.00s)
        server_test.go:101: status code is wrong: got 404, want 200
FAIL
FAIL    playerstore     0.221s
FAIL
```

Our `PlayerServer` returns a `404 Not Found`, as if we were trying to get the wins for an unknown player. Looking at how `server.go` implements `ServeHTTP`, we realize that it always assumes to be called with a URL pointing to a specific player:

```
player := strings.TrimPrefix(r.URL.Path, "/players/")
```

In the previous chapter, we mentioned this was a fairly naive way of doing our routing. Our test informs us correctly that we need a concept how to deal with different request paths.

---

## Write enough code to make it pass

Go has a built-in routing mechanism called `ServeMux` (request multiplexer) which lets you attach `http.Handler`s to particular request paths.

Let's commit some sins and get the tests passing in the quickest way we can, knowing we can refactor it with safety once we know the tests are passing.

```
//server.go
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {

        router := http.NewServeMux()

        router.Handle("/league", http.HandlerFunc(func(w http.ResponseWriter, r *http
                w.WriteHeader(http.StatusOK)
        }))

        router.Handle("/players/", http.HandlerFunc(func(w http.ResponseWriter, r *ht
                player := strings.TrimPrefix(r.URL.Path, "/players/")

```

```
                switch r.Method {
                case http.MethodPost:

                        p.processWin(w, player)
                case http.MethodGet:
                        p.showScore(w, player)
                }
        }))

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

- When the request starts we create a router and then we tell it for `x` path use `y` handler.
- So for our new endpoint, we use `http.HandlerFunc` and an *anonymous function* to `w.WriteHeader(http.StatusOK)` when `/league` is requested to make our new test pass.
- For the `/players/` route we just cut and paste our code into another `http.HandlerFunc`.
- Finally, we handle the request that came in by calling our new router's `ServeHTTP` (notice how `ServeMux` is *also* an `http.Handler`?)

The tests should now pass.

---

## Refactor

`ServeHTTP` is looking quite big, we can separate things out a bit by refactoring our handlers into separate methods.

```
//server.go
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {

        router := http.NewServeMux()
        router.Handle("/league", http.HandlerFunc(p.leagueHandler))
        router.Handle("/players/", http.HandlerFunc(p.playersHandler))

        router.ServeHTTP(w, r)
}

func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
}
```

```
14
15 func (p *PlayerServer) playersHandler(w http.ResponseWriter, r *http.Request) {
16         player := strings.TrimPrefix(r.URL.Path, "/players/")
17
18         switch r.Method {
19         case http.MethodPost:
20                 p.processWin(w, player)
21         case http.MethodGet:
22                 p.showScore(w, player)
23         }
24 }
```

It's quite odd (and inefficient) to be setting up a router as a request comes in and then calling it. What we ideally want to do is have some kind of `NewPlayerServer` function which will take our dependencies and do the one-time setup of creating the router. Each request can then just use that one instance of the router.

```
1  //server.go
2  type PlayerServer struct {
3          store  PlayerStore
4          router *http.ServeMux
5  }
6
7  func NewPlayerServer(store PlayerStore) *PlayerServer {
8          p := &PlayerServer{
9                  store,
10                 http.NewServeMux(),
11         }
12
13         p.router.Handle("/league", http.HandlerFunc(p.leagueHandler))
14         p.router.Handle("/players/", http.HandlerFunc(p.playersHandler))
15
16         return p
17 }
18
19 func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
20         p.router.ServeHTTP(w, r)
21 }
```

- `PlayerServer` now needs to store a router.

- We have moved the routing creation out of `ServeHTTP` and into our `NewPlayerServer` so this only has to be done once, not per request.
- You will need to update all the test and production code where we used to do `PlayerServer{&store}` with `NewPlayerServer(&store)`.

#### One final refactor

Try changing the code to the following.

```
type PlayerServer struct {
        store PlayerStore
        http.Handler
}

func NewPlayerServer(store PlayerStore) *PlayerServer {
        p := new(PlayerServer)

        p.store = store

        router := http.NewServeMux()
        router.Handle("/league", http.HandlerFunc(p.leagueHandler))
        router.Handle("/players/", http.HandlerFunc(p.playersHandler))

        p.Handler = router

        return p
}
```

Then replace `server := &PlayerServer{&store}` with `server := NewPlayerServer(&store)` in `server_test.go`, `server_integration_test.go`, and `main.go`.

Finally make sure you **delete** `func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request)` as it is no longer needed!

---

## Embedding

We changed the second property of `PlayerServer`, removing the named property `router`

`http.ServeMux` and replaced it with `http.Handler`; this is called *embedding*.

> Go does not provide the typical, type-driven notion of subclassing, but it does have the ability to "borrow" pieces of an implementation by embedding types within a struct or interface.

Effective Go - Embedding

What this means is that our `PlayerServer` now has all the methods that `http.Handler` has, which is just `ServeHTTP`.

To "fill in" the `http.Handler` we assign it to the `router` we create in `NewPlayerServer`. We can do this because `http.ServeMux` has the method `ServeHTTP`.

This lets us remove our own `ServeHTTP` method, as we are already exposing one via the embedded type.

Embedding is a very interesting language feature. You can use it with interfaces to compose new interfaces.

```
type Animal interface {
        Eater
        Sleeper
}
```

And you can use it with concrete types too, not just interfaces. As you'd expect if you embed a concrete type you'll have access to all its public methods and fields.

**Any downsides?**

You must be careful with embedding types because you will expose all public methods and fields of the type you embed. In our case, it is ok because we embedded just the *interface* that we wanted to expose ( `http.Handler` ).

If we had been lazy and embedded `http.ServeMux` instead (the concrete type) it would still work *but* users of `PlayerServer` would be able to add new routes to our server because `Handle(path, handler)` would be public.

**When embedding types, really think about what impact that has on your public API.**

It is a *very* common mistake to misuse embedding and end up polluting your APIs and exposing the internals of your type.

Now we've restructured our application we can easily add new routes and have the start of the `/league` endpoint. We now need to make it return some useful information.

We should return some JSON that looks something like this.

```
[
   {
      "Name":"Bill",
      "Wins":10
   },
   {
      "Name":"Alice",
      "Wins":15
   }
]
```

---

## Write the test first

We'll start by trying to parse the response into something meaningful.

```
//server_test.go
func TestLeague(t *testing.T) {
        store := StubPlayerStore{}
        server := NewPlayerServer(&store)

        t.Run("it returns 200 on /league", func(t *testing.T) {
                request, _ := http.NewRequest(http.MethodGet, "/league", nil)
                response := httptest.NewRecorder()

                server.ServeHTTP(response, request)

                var got []Player

                err := json.NewDecoder(response.Body).Decode(&got)

                if err != nil {
                        t.Fatalf("Unable to parse response from server %q into slice
                }
```

```
                assertStatus(t, response.Code, http.StatusOK)

        })
}
```

### Why not test the JSON string?

You could argue a simpler initial step would be just to assert that the response body has a particular JSON string.

In my experience tests that assert against JSON strings have the following problems.

- *Brittleness*. If you change the data-model your tests will fail.
- *Hard to debug*. It can be tricky to understand what the actual problem is when comparing two JSON strings.
- *Poor intention*. Whilst the output should be JSON, what's really important is exactly what the data is, rather than how it's encoded.
- *Re-testing the standard library*. There is no need to test how the standard library outputs JSON, it is already tested. Don't test other people's code.

Instead, we should look to parse the JSON into data structures that are relevant for us to test with.

### Data modelling

Given the JSON data model, it looks like we need an array of `Player` with some fields so we have created a new type to capture this.

```
//server.go
type Player struct {
        Name string
        Wins int
}
```

### JSON decoding

```
//server_test.go
var got []Player
```

```
3 err := json.NewDecoder(response.Body).Decode(&got)
```

To parse JSON into our data model we create a `Decoder` from `encoding/json` package and then call its `Decode` method. To create a `Decoder` it needs an `io.Reader` to read from which in our case is our response spy's `Body`.

`Decode` takes the address of the thing we are trying to decode into which is why we declare an empty slice of `Player` the line before.

Parsing JSON can fail so `Decode` can return an `error`. There's no point continuing the test if that fails so we check for the error and stop the test with `t.Fatalf` if it happens. Notice that we print the response body along with the error as it's important for someone running the test to see what string cannot be parsed.

---

## Try to run the test

```
1 === RUN   TestLeague/it_returns_200_on_/league
2     --- FAIL: TestLeague/it_returns_200_on_/league (0.00s)
3         server_test.go:107: Unable to parse response from server '' into slice of Pla
```

Our endpoint currently does not return a body so it cannot be parsed into JSON.

---

## Write enough code to make it pass

```
1  //server.go
2  func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
3          leagueTable := []Player{
4                  {"Chris", 20},
5          }
6
7          json.NewEncoder(w).Encode(leagueTable)
8
9          w.WriteHeader(http.StatusOK)
10 }
```

The test now passes.

### Encoding and Decoding

Notice the lovely symmetry in the standard library.

- To create an `Encoder` you need an `io.Writer` which is what `http.ResponseWriter` implements.
- To create a `Decoder` you need an `io.Reader` which the `Body` field of our response spy implements.

Throughout this book, we have used `io.Writer` and this is another demonstration of its prevalence in the standard library and how a lot of libraries easily work with it.

---

## Refactor

It would be nice to introduce a separation of concern between our handler and getting the `leagueTable` as we know we're going to not hard-code that very soon.

```
//server.go
func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
        json.NewEncoder(w).Encode(p.getLeagueTable())
        w.WriteHeader(http.StatusOK)
}

func (p *PlayerServer) getLeagueTable() []Player {
        return []Player{
                {"Chris", 20},
        }
}
```

Next, we'll want to extend our test so that we can control exactly what data we want back.

---

## Write the test first

We can update the test to assert that the league table contains some players that we will stub in our store.

Update `StubPlayerStore` to let it store a league, which is just a slice of `Player`. We'll store our expected data in there.

```
//server_test.go
type StubPlayerStore struct {
        scores   map[string]int
        winCalls []string
        league   []Player
}
```

Next, update our current test by putting some players in the league property of our stub and assert they get returned from our server.

```
//server_test.go
func TestLeague(t *testing.T) {

        t.Run("it returns the league table as JSON", func(t *testing.T) {
                wantedLeague := []Player{
                        {"Cleo", 32},
                        {"Chris", 20},
                        {"Tiest", 14},
                }

                store := StubPlayerStore{nil, nil, wantedLeague}
                server := NewPlayerServer(&store)

                request, _ := http.NewRequest(http.MethodGet, "/league", nil)
                response := httptest.NewRecorder()

                server.ServeHTTP(response, request)

                var got []Player

                err := json.NewDecoder(response.Body).Decode(&got)

                if err != nil {
                        t.Fatalf("Unable to parse response from server %q into slice
                }
```

```
27                 assertStatus(t, response.Code, http.StatusOK)

28
29                 if !reflect.DeepEqual(got, wantedLeague) {
30                         t.Errorf("got %v want %v", got, wantedLeague)
31                 }
32         })
33 }
```

## Try to run the test

```
./server_test.go:33:3: too few values in struct initializer
./server_test.go:70:3: too few values in struct initializer
```

## Write the minimal amount of code for the test to run and check the failing test output

You'll need to update the other tests as we have a new field in `StubPlayerStore`; set it to nil for the other tests.

Try running the tests again and you should get

```
=== RUN   TestLeague/it_returns_the_league_table_as_JSON
    --- FAIL: TestLeague/it_returns_the_league_table_as_JSON (0.00s)
        server_test.go:124: got [{Chris 20}] want [{Cleo 32} {Chris 20} {Tiest 14}]
```

## Write enough code to make it pass

We know the data is in our `StubPlayerStore` and we've abstracted that away into an interface `PlayerStore`. We need to update this so anyone passing us in a `PlayerStore` can provide us

with the data for leagues.

```
//server.go
type PlayerStore interface {
        GetPlayerScore(name string) int
        RecordWin(name string)
        GetLeague() []Player
}
```

Now we can update our handler code to call that rather than returning a hard-coded list. Delete our method `getLeagueTable()` and then update `leagueHandler` to call `GetLeague()`.

```
//server.go
func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
        json.NewEncoder(w).Encode(p.store.GetLeague())
        w.WriteHeader(http.StatusOK)
}
```

Try and run the tests.

```
# github.com/quii/learn-go-with-tests/json-and-io/v4
./main.go:9:50: cannot use NewInMemoryPlayerStore() (type *InMemoryPlayerStore) as ty
    *InMemoryPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_integration_test.go:11:27: cannot use store (type *InMemoryPlayerStore) as t
    *InMemoryPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_test.go:36:28: cannot use &store (type *StubPlayerStore) as type PlayerStore
    *StubPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_test.go:74:28: cannot use &store (type *StubPlayerStore) as type PlayerStore
    *StubPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_test.go:106:29: cannot use &store (type *StubPlayerStore) as type PlayerStor
    *StubPlayerStore does not implement PlayerStore (missing GetLeague method)
```

The compiler is complaining because `InMemoryPlayerStore` and `StubPlayerStore` do not have the new method we added to our interface.

For `StubPlayerStore` it's pretty easy, just return the `league` field we added earlier.

```
//server_test.go
```

```
func (s *StubPlayerStore) GetLeague() []Player {
        return s.league
}
```

Here's a reminder of how `InMemoryStore` is implemented.

```
//in_memory_player_store.go
type InMemoryPlayerStore struct {
        store map[string]int
}
```

Whilst it would be pretty straightforward to implement `GetLeague` "properly" by iterating over the map remember we are just trying to *write the minimal amount of code to make the tests pass*.

So let's just get the compiler happy for now and live with the uncomfortable feeling of an incomplete implementation in our `InMemoryStore`.

```
//in_memory_player_store.go
func (i *InMemoryPlayerStore) GetLeague() []Player {
        return nil
}
```

What this is really telling us is that *later* we're going to want to test this but let's park that for now.

Try and run the tests, the compiler should pass and the tests should be passing!

---

## Refactor

The test code does not convey our intent very well and has a lot of boilerplate we can refactor away.

```
//server_test.go
t.Run("it returns the league table as JSON", func(t *testing.T) {
        wantedLeague := []Player{
                {"Cleo", 32},
                {"Chris", 20},
                {"Tiest", 14},
```

```
        }

        store := StubPlayerStore{nil, nil, wantedLeague}
        server := NewPlayerServer(&store)

        request := newLeagueRequest()
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        got := getLeagueFromResponse(t, response.Body)
        assertStatus(t, response.Code, http.StatusOK)
        assertLeague(t, got, wantedLeague)
})
```

Here are the new helpers

```
//server_test.go
func getLeagueFromResponse(t testing.TB, body io.Reader) (league []Player) {
        t.Helper()
        err := json.NewDecoder(body).Decode(&league)

        if err != nil {
                t.Fatalf("Unable to parse response from server %q into slice of Playe
        }

        return
}

func assertLeague(t testing.TB, got, want []Player) {
        t.Helper()
        if !reflect.DeepEqual(got, want) {
                t.Errorf("got %v want %v", got, want)
        }
}

func newLeagueRequest() *http.Request {
        req, _ := http.NewRequest(http.MethodGet, "/league", nil)
        return req
}
```

One final thing we need to do for our server to work is make sure we return a `content-type`

header in the response so machines can recognise we are returning `JSON`.

---

## Write the test first

Add this assertion to the existing test

```
//server_test.go
if response.Result().Header.Get("content-type") != "application/json" {
        t.Errorf("response did not have content-type of application/json, got %v", re
}
```

---

## Try to run the test

```
=== RUN   TestLeague/it_returns_the_league_table_as_JSON
    --- FAIL: TestLeague/it_returns_the_league_table_as_JSON (0.00s)
        server_test.go:124: response did not have content-type of application/json, g
```

---

## Write enough code to make it pass

Update `leagueHandler`

```
//server.go
func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("content-type", "application/json")
        json.NewEncoder(w).Encode(p.store.GetLeague())
}
```

The test should pass.

---

## Refactor

Create a constant for "application/json" and use it in `leagueHandler`

```
//server.go
const jsonContentType = "application/json"

func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("content-type", jsonContentType)
        json.NewEncoder(w).Encode(p.store.GetLeague())
}
```

Then add a helper for `assertContentType`.

```
//server_test.go
func assertContentType(t testing.TB, response *httptest.ResponseRecorder, want string
        t.Helper()
        if response.Result().Header.Get("content-type") != want {
                t.Errorf("response did not have content-type of %s, got %v", want, re
        }
}
```

Use it in the test.

```
//server_test.go
assertContentType(t, response, jsonContentType)
```

Now that we have sorted out `PlayerServer` for now we can turn our attention to `InMemoryPlayerStore` because right now if we tried to demo this to the product owner `/league` will not work.

The quickest way for us to get some confidence is to add to our integration test, we can hit the new endpoint and check we get back the correct response from `/league`.

---

## Write the test first

We can use `t.Run` to break up this test a bit and we can reuse the helpers from our server tests - again showing the importance of refactoring tests.

```
//server_integration_test.go
func TestRecordingWinsAndRetrievingThem(t *testing.T) {
        store := NewInMemoryPlayerStore()
        server := NewPlayerServer(store)
        player := "Pepper"

        server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))
        server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))
        server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))

        t.Run("get score", func(t *testing.T) {
                response := httptest.NewRecorder()
                server.ServeHTTP(response, newGetScoreRequest(player))
                assertStatus(t, response.Code, http.StatusOK)

                assertResponseBody(t, response.Body.String(), "3")
        })

        t.Run("get league", func(t *testing.T) {
                response := httptest.NewRecorder()
                server.ServeHTTP(response, newLeagueRequest())
                assertStatus(t, response.Code, http.StatusOK)

                got := getLeagueFromResponse(t, response.Body)
                want := []Player{
                        {"Pepper", 3},
                }
                assertLeague(t, got, want)
        })
}
```

## Try to run the test

```
```

```
=== RUN   TestRecordingWinsAndRetrievingThem/get_league (0.00s)
    server_integration_test.go:35: got [] want [{Pepper 3}]
```

## Write enough code to make it pass

`InMemoryPlayerStore` is returning `nil` when you call `GetLeague()` so we'll need to fix that.

```
//in_memory_player_store.go
func (i *InMemoryPlayerStore) GetLeague() []Player {
        var league []Player
        for name, wins := range i.store {
                league = append(league, Player{name, wins})
        }
        return league
}
```

All we need to do is iterate over the map and convert each key/value to a `Player`.

The test should now pass.

## Wrapping up

We've continued to safely iterate on our program using TDD, making it support new endpoints in a maintainable way with a router and it can now return JSON for our consumers. In the next chapter, we will cover persisting the data and sorting our league.

What we've covered:

- **Routing**. The standard library offers you an easy to use type to do routing. It fully embraces the `http.Handler` interface in that you assign routes to `Handler`s and the router itself is also a `Handler`. It does not have some features you might expect though such as path variables (e.g `/users/{id}`). You can easily parse this information yourself but you might want to consider looking at other routing libraries if it becomes a burden. Most of the popular ones stick to the standard library's philosophy of also implementing `http.Handler`.
-

- **Type embedding**. We touched a little on this technique but you can learn more about it from Effective Go. If there is one thing you should take away from this is that it can be extremely useful but *always thinking about your public API, only expose what's appropriate*.
- **JSON deserializing and serializing**. The standard library makes it very trivial to serialise and deserialise your data. It is also open to configuration and you can customise how these data transformations work if necessary