HTTP server

You can find all the code for this chapter here

You have been asked to create a web server where users can track how many games players have won.

- `GET /players/{name}` should return a number indicating the total number of wins
- `POST /players/{name}` should record a win for that name, incrementing for every subsequent `POST`

We will follow the TDD approach, getting working software as quickly as we can and then making small iterative improvements until we have the solution. By taking this approach we

- Keep the problem space small at any given time
- Don't go down rabbit holes
- If we ever get stuck/lost, doing a revert wouldn't lose loads of work.

Red, green, refactor

Throughout this book, we have emphasised the TDD process of write a test & watch it fail (red), write the *minimal* amount of code to make it work (green) and then refactor.

This discipline of writing the minimal amount of code is important in terms of the safety TDD gives you. You should be striving to get out of "red" as soon as you can.

Kent Beck describes it as:

> Make the test work quickly, committing whatever sins necessary in process.

You can commit these sins because you will refactor afterwards backed by the safety of the tests.

What if you don't do this?

The more changes you make while in red, the more likely you are to add more problems, not covered by tests.

The idea is to be iteratively writing useful code with small steps, driven by tests so that you don't fall into a rabbit hole for hours.

Chicken and egg

How can we incrementally build this? We can't `GET` a player without having stored something and it seems hard to know if `POST` has worked without the `GET` endpoint already existing.

This is where *mocking* shines.

- `GET` will need a `PlayerStore` *thing* to get scores for a player. This should be an interface so when we test we can create a simple stub to test our code without needing to have implemented any actual storage code.
- For `POST` we can *spy* on its calls to `PlayerStore` to make sure it stores players correctly. Our implementation of saving won't be coupled to retrieval.
- For having some working software quickly we can make a very simple in-memory implementation and then later we can create an implementation backed by whatever storage mechanism we prefer.

Write the test first

We can write a test and make it pass by returning a hard-coded value to get us started. Kent Beck refers this as "Faking it". Once we have a working test we can then write more tests to help us remove that constant.

By doing this very small step, we can make the important start of getting an overall project structure working correctly without having to worry too much about our application logic.

To create a web server in Go you will typically call ListenAndServe.

```
func ListenAndServe(addr string, handler Handler) error
```

This will start a web server listening on a port, creating a goroutine for every request and running it against a `Handler`.

```go
type Handler interface {

        ServeHTTP(ResponseWriter, *Request)
}
```

A type implements the Handler interface by implementing the `ServeHTTP` method which expects two arguments, the first is where we *write our response* and the second is the HTTP request that was sent to the server.

Let's create a file named `server_test.go` and write a test for a function `PlayerServer` that takes in those two arguments. The request sent in will be to get a player's score, which we expect to be `"20"`.

```go
func TestGETPlayers(t *testing.T) {
        t.Run("returns Pepper's score", func(t *testing.T) {
                request, _ := http.NewRequest(http.MethodGet, "/players/Pepper", nil)
                response := httptest.NewRecorder()

                PlayerServer(response, request)

                got := response.Body.String()
                want := "20"

                if got != want {
                        t.Errorf("got %q, want %q", got, want)
                }
        })
}
```

In order to test our server, we will need a `Request` to send in and we'll want to *spy* on what our handler writes to the `ResponseWriter`.

- We use `http.NewRequest` to create a request. The first argument is the request's method and the second is the request's path. The `nil` argument refers to the request's body, which we don't need to set in this case.
- `net/http/httptest` has a spy already made for us called `ResponseRecorder` so we can use that. It has many helpful methods to inspect what has been written as a response.

Try to run the test

`./server_test.go:13:2: undefined: PlayerServer`

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

The compiler is here to help, just listen to it.

Create a file named `server.go` and define `PlayerServer`

```
func PlayerServer() {}
```

Try again

```
./server_test.go:13:14: too many arguments in call to PlayerServer
    have (*httptest.ResponseRecorder, *http.Request)
    want ()
```

Add the arguments to our function

```
import "net/http"

func PlayerServer(w http.ResponseWriter, r *http.Request) {

}
```

The code now compiles and the test fails

```
=== RUN   TestGETPlayers/returns_Pepper's_score
    --- FAIL: TestGETPlayers/returns_Pepper's_score (0.00s)
        server_test.go:20: got '', want '20'
```

Write enough code to make it pass

From the DI chapter, we touched on HTTP servers with a `Greet` function. We learned that net/http's `ResponseWriter` also implements io `Writer` so we can use `fmt.Fprint` to send strings as HTTP responses.

```go
func PlayerServer(w http.ResponseWriter, r *http.Request) {
	fmt.Fprint(w, "20")
}
```

The test should now pass.

Complete the scaffolding

We want to wire this up into an application. This is important because

- We'll have *actual working software*, we don't want to write tests for the sake of it, it's good to see the code in action.
- As we refactor our code, it's likely we will change the structure of the program. We want to make sure this is reflected in our application too as part of the incremental approach.

Create a new `main.go` file for our application and put this code in

```go
package main

import (
	"log"
	"net/http"
)

func main() {
	handler := http.HandlerFunc(PlayerServer)
	log.Fatal(http.ListenAndServe(":5000", handler))
}
```

So far all of our application code has been in one file, however, this isn't best practice for larger projects where you'll want to separate things into different files.

To run this, do `go build` which will take all the `.go` files in the directory and build you a program. You can then execute it with `./myprogram`.

`http.HandlerFunc`

Earlier we explored that the `Handler` interface is what we need to implement in order to make a server. *Typically* we do that by creating a `struct` and make it implement the interface by implementing its own ServeHTTP method. However the use-case for structs is for holding data but *currently* we have no state, so it doesn't feel right to be creating one.

HandlerFunc lets us avoid this.

> The HandlerFunc type is an adapter to allow the use of ordinary functions as HTTP handlers. If f is a function with the appropriate signature, HandlerFunc(f) is a Handler that calls f.

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

From the documentation, we see that type `HandlerFunc` has already implemented the `ServeHTTP` method. By type casting our `PlayerServer` function with it, we have now implemented the required `Handler`.

`http.ListenAndServe(":5000"...)`

`ListenAndServe` takes a port to listen on a `Handler`. If there is a problem the web server will return an error, an example of that might be the port already being listened to. For that reason we wrap the call in `log.Fatal` to log the error to the user.

What we're going to do now is write *another* test to force us into making a positive change to try and move away from the hard-coded value.

Write the test first

We'll add another subtest to our suite which tries to get the score of a different player, which will break our hard-coded approach.

```go
t.Run("returns Floyd's score", func(t *testing.T) {
	request, _ := http.NewRequest(http.MethodGet, "/players/Floyd", nil)
	response := httptest.NewRecorder()

	PlayerServer(response, request)

	got := response.Body.String()
	want := "10"

	if got != want {
		t.Errorf("got %q, want %q", got, want)
	}
})
```

You may have been thinking

> Surely we need some kind of concept of storage to control which player gets what score. It's weird that the values seem so arbitrary in our tests.

Remember we are just trying to take as small as steps as reasonably possible, so we're just trying to break the constant for now.

Try to run the test

```
=== RUN   TestGETPlayers/returns_Pepper's_score
    --- PASS: TestGETPlayers/returns_Pepper's_score (0.00s)
=== RUN   TestGETPlayers/returns_Floyd's_score
    --- FAIL: TestGETPlayers/returns_Floyd's_score (0.00s)
        server_test.go:34: got '20', want '10'
```

Write enough code to make it pass

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

	if player == "Pepper" {
		fmt.Fprint(w, "20")
		return
	}

	if player == "Floyd" {
		fmt.Fprint(w, "10")
		return
	}
}
```

This test has forced us to actually look at the request's URL and make a decision. So whilst in our heads, we may have been worrying about player stores and interfaces the next logical step actually seems to be about *routing*.

If we had started with the store code the amount of changes we'd have to do would be very large compared to this. **This is a smaller step towards our final goal and was driven by tests**.

We're resisting the temptation to use any routing libraries right now, just the smallest step to get our test passing.

`r.URL.Path` returns the path of the request which we can then use `strings.TrimPrefix` to trim away `/players/` to get the requested player. It's not very robust but will do the trick for now.

Refactor

We can simplify the `PlayerServer` by separating out the score retrieval into a function

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

	fmt.Fprint(w, GetPlayerScore(player))
```

```go
}
func GetPlayerScore(name string) string {
	if name == "Pepper" {
		return "20"
	}

	if name == "Floyd" {
		return "10"
	}

	return ""
}
```

And we can DRY up some of the code in the tests by making some helpers

```go
//server_test.go
func TestGETPlayers(t *testing.T) {
	t.Run("returns Pepper's score", func(t *testing.T) {
		request := newGetScoreRequest("Pepper")
		response := httptest.NewRecorder()

		PlayerServer(response, request)

		assertResponseBody(t, response.Body.String(), "20")
	})

	t.Run("returns Floyd's score", func(t *testing.T) {
		request := newGetScoreRequest("Floyd")
		response := httptest.NewRecorder()

		PlayerServer(response, request)

		assertResponseBody(t, response.Body.String(), "10")
	})
}

func newGetScoreRequest(name string) *http.Request {
	req, _ := http.NewRequest(http.MethodGet, fmt.Sprintf("/players/%s", name), r
	return req
}

func assertResponseBody(t testing.TB, got, want string) {
```

```
	t.Helper()
	if got != want {
		t.Errorf("response body is wrong, got %q want %q", got, want)
	}
}
```

However, we still shouldn't be happy. It doesn't feel right that our server knows the scores.

Our refactoring has made it pretty clear what to do.

We moved the score calculation out of the main body of our handler into a function `GetPlayerScore`. This feels like the right place to separate the concerns using interfaces.

Let's move our function we re-factored to be an interface instead

```go
type PlayerStore interface {
	GetPlayerScore(name string) int
}
```

For our `PlayerServer` to be able to use a `PlayerStore`, it will need a reference to one. Now feels like the right time to change our architecture so that our `PlayerServer` is now a `struct`.

```go
type PlayerServer struct {
	store PlayerStore
}
```

Finally, we will now implement the `Handler` interface by adding a method to our new struct and putting in our existing handler code.

```go
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
	player := strings.TrimPrefix(r.URL.Path, "/players/")
	fmt.Fprint(w, p.store.GetPlayerScore(player))
}
```

The only other change is we now call our `store.GetPlayerScore` to get the score, rather than the local function we defined (which we can now delete).

Here is the full code listing of our server

```go
//server.go
type PlayerStore interface {
	GetPlayerScore(name string) int
}

type PlayerServer struct {
	store PlayerStore
}

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

Fix the issues

This was quite a few changes and we know our tests and application will no longer compile, but just relax and let the compiler work through it.

`./main.go:9:58: type PlayerServer is not an expression`

We need to change our tests to instead create a new instance of our `PlayerServer` and then call its method `ServeHTTP`.

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

	t.Run("returns Pepper's score", func(t *testing.T) {
		request := newGetScoreRequest("Pepper")
		response := httptest.NewRecorder()

		server.ServeHTTP(response, request)

		assertResponseBody(t, response.Body.String(), "20")
	})
```

```go
	t.Run("returns Floyd's score", func(t *testing.T) {
		request := newGetScoreRequest("Floyd")
		response := httptest.NewRecorder()

		server.ServeHTTP(response, request)

		assertResponseBody(t, response.Body.String(), "10")
	})
}
```

Notice we're still not worrying about making stores *just yet*, we just want the compiler passing as soon as we can.

You should be in the habit of prioritising having code that compiles and then code that passes the tests.

By adding more functionality (like stub stores) whilst the code isn't compiling, we are opening ourselves up to potentially *more* compilation problems.

Now `main.go` won't compile for the same reason.

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

Finally, everything is compiling but the tests are failing

```
=== RUN   TestGETPlayers/returns_the_Pepper's_score
panic: runtime error: invalid memory address or nil pointer dereference [recovered]
    panic: runtime error: invalid memory address or nil pointer dereference
```

This is because we have not passed in a `PlayerStore` in our tests. We'll need to make a stub one up.

```go
//server_test.go
type StubPlayerStore struct {
	scores map[string]int
```

```
}

func (s *StubPlayerStore) GetPlayerScore(name string) int {
        score := s.scores[name]
        return score
}
```

A `map` is a quick and easy way of making a stub key/value store for our tests. Now let's create one of these stores for our tests and send it into our `PlayerServer`.

```
//server_test.go
func TestGETPlayers(t *testing.T) {
        store := StubPlayerStore{
                map[string]int{
                        "Pepper": 20,
                        "Floyd":  10,
                },
        }
        server := &PlayerServer{&store}

        t.Run("returns Pepper's score", func(t *testing.T) {
                request := newGetScoreRequest("Pepper")
                response := httptest.NewRecorder()

                server.ServeHTTP(response, request)

                assertResponseBody(t, response.Body.String(), "20")
        })

        t.Run("returns Floyd's score", func(t *testing.T) {
                request := newGetScoreRequest("Floyd")
                response := httptest.NewRecorder()

                server.ServeHTTP(response, request)

                assertResponseBody(t, response.Body.String(), "10")
        })
}
```

Our tests now pass and are looking better. The *intent* behind our code is clearer now due to the introduction of the store. We're telling the reader that because we have *this data in a `PlayerStore`* that when you use it with a `PlayerServer` you should get the following responses

Run the application

Now our tests are passing the last thing we need to do to complete this refactor is to check if our application is working. The program should start up but you'll get a horrible response if you try and hit the server at `http://localhost:5000/players/Pepper`.

The reason for this is that we have not passed in a `PlayerStore`.

We'll need to make an implementation of one, but that's difficult right now as we're not storing any meaningful data so it'll have to be hard-coded for the time being.

```go
//main.go
type InMemoryPlayerStore struct{}

func (i *InMemoryPlayerStore) GetPlayerScore(name string) int {
	return 123
}

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

If you run `go build` again and hit the same URL you should get `"123"`. Not great, but until we store data that's the best we can do. It also didn't feel great that our main application was starting up but not actually working. We had to manually test to see the problem.

We have a few options as to what to do next

- Handle the scenario where the player doesn't exist
- Handle the `POST /players/{name}` scenario

Whilst the `POST` scenario gets us closer to the "happy path", I feel it'll be easier to tackle the missing player scenario first as we're in that context already. We'll get to the rest later.

Write the test first

Add a missing player scenario to our existing suite

```
//server_test.go
t.Run("returns 404 on missing players", func(t *testing.T) {
	request := newGetScoreRequest("Apollo")
	response := httptest.NewRecorder()

	server.ServeHTTP(response, request)

	got := response.Code
	want := http.StatusNotFound

	if got != want {
		t.Errorf("got status %d want %d", got, want)
	}
})
```

Try to run the test

```
=== RUN   TestGETPlayers/returns_404_on_missing_players
    --- FAIL: TestGETPlayers/returns_404_on_missing_players (0.00s)
        server_test.go:56: got status 200 want 404
```

Write enough code to make it pass

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

	w.WriteHeader(http.StatusNotFound)

	fmt.Fprint(w, p.store.GetPlayerScore(player))
}
```

Sometimes I heavily roll my eyes when TDD advocates say "make sure you just write the minimal amount of code to make it pass" as it can feel very pedantic.

But this scenario illustrates the example well. I have done the bare minimum (knowing it is not correct), which is write a `StatusNotFound` on **all responses** but all our tests are passing!

By doing the bare minimum to make the tests pass it can highlight gaps in your tests. In our case, we are not asserting that we should be getting a `StatusOK` when players *do* exist in the store.

Update the other two tests to assert on the status and fix the code.

Here are the new tests

```
//server_test.go
func TestGETPlayers(t *testing.T) {
	store := StubPlayerStore{
		map[string]int{
			"Pepper": 20,
			"Floyd":  10,
		},
	}
	server := &PlayerServer{&store}

	t.Run("returns Pepper's score", func(t *testing.T) {
		request := newGetScoreRequest("Pepper")
		response := httptest.NewRecorder()

		server.ServeHTTP(response, request)

		assertStatus(t, response.Code, http.StatusOK)
		assertResponseBody(t, response.Body.String(), "20")
	})

	t.Run("returns Floyd's score", func(t *testing.T) {
		request := newGetScoreRequest("Floyd")
		response := httptest.NewRecorder()

		server.ServeHTTP(response, request)

		assertStatus(t, response.Code, http.StatusOK)
		assertResponseBody(t, response.Body.String(), "10")
	})
```

```
	t.Run("returns 404 on missing players", func(t *testing.T) {
		request := newGetScoreRequest("Apollo")
		response := httptest.NewRecorder()

		server.ServeHTTP(response, request)

		assertStatus(t, response.Code, http.StatusNotFound)
	})
}

func assertStatus(t testing.TB, got, want int) {
	t.Helper()
	if got != want {
		t.Errorf("did not get correct status, got %d, want %d", got, want)
	}
}

func newGetScoreRequest(name string) *http.Request {
	req, _ := http.NewRequest(http.MethodGet, fmt.Sprintf("/players/%s", name), r
	return req
}

func assertResponseBody(t testing.TB, got, want string) {
	t.Helper()
	if got != want {
		t.Errorf("response body is wrong, got %q want %q", got, want)
	}
}
```

We're checking the status in all our tests now so I made a helper `assertStatus` to facilitate that.

Now our first two tests fail because of the 404 instead of 200, so we can fix `PlayerServer` to only return not found if the score is 0.

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

	score := p.store.GetPlayerScore(player)

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

```
        }

        fmt.Fprint(w, score)
}
```

Storing scores

Now that we can retrieve scores from a store it now makes sense to be able to store new scores.

Write the test first

```
//server_test.go
func TestStoreWins(t *testing.T) {
        store := StubPlayerStore{
                map[string]int{},
        }
        server := &PlayerServer{&store}

        t.Run("it returns accepted on POST", func(t *testing.T) {
                request, _ := http.NewRequest(http.MethodPost, "/players/Pepper", nil
                response := httptest.NewRecorder()

                server.ServeHTTP(response, request)

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

For a start let's just check we get the correct status code if we hit the particular route with POST. This lets us drive out the functionality of accepting a different kind of request and handling it differently to `GET /players/{name}`. Once this works we can then start asserting on our handler's interaction with the store.

Try to run the test

```
=== RUN   TestStoreWins/it_returns_accepted_on_POST
    --- FAIL: TestStoreWins/it_returns_accepted_on_POST (0.00s)
        server_test.go:70: did not get correct status, got 404, want 202
```

Write enough code to make it pass

Remember we are deliberately committing sins, so an `if` statement based on the request's method will do the trick.

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

	if r.Method == http.MethodPost {
		w.WriteHeader(http.StatusAccepted)
		return
	}

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

	score := p.store.GetPlayerScore(player)

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

	fmt.Fprint(w, score)
}
```

Refactor

The handler is looking a bit muddled now. Let's break the code up to make it easier to follow and isolate the different functionality into new functions.

```go
//server.go
```

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

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

}

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

	score := p.store.GetPlayerScore(player)

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

	fmt.Fprint(w, score)
}

func (p *PlayerServer) processWin(w http.ResponseWriter) {
	w.WriteHeader(http.StatusAccepted)
}
```

This makes the routing aspect of `ServeHTTP` a bit clearer and means our next iterations on storing can just be inside `processWin`.

Next, we want to check that when we do our `POST /players/{name}` that our `PlayerStore` is told to record the win.

Write the test first

We can accomplish this by extending our `StubPlayerStore` with a new `RecordWin` method and then spy on its invocations.

```go
//server_test.go
type StubPlayerStore struct {
```

```
        scores   map[string]int
        winCalls []string
}

func (s *StubPlayerStore) GetPlayerScore(name string) int {
        score := s.scores[name]
        return score
}

func (s *StubPlayerStore) RecordWin(name string) {
        s.winCalls = append(s.winCalls, name)
}
```

Now extend our test to check the number of invocations for a start

```
//server_test.go
func TestStoreWins(t *testing.T) {
        store := StubPlayerStore{
                map[string]int{},
        }
        server := &PlayerServer{&store}

        t.Run("it records wins when POST", func(t *testing.T) {
                request := newPostWinRequest("Pepper")
                response := httptest.NewRecorder()

                server.ServeHTTP(response, request)

                assertStatus(t, response.Code, http.StatusAccepted)

                if len(store.winCalls) != 1 {
                        t.Errorf("got %d calls to RecordWin want %d", len(store.winCa
                }
        })
}

func newPostWinRequest(name string) *http.Request {
        req, _ := http.NewRequest(http.MethodPost, fmt.Sprintf("/players/%s", name),
        return req
}
```

Try to run the test

```
./server_test.go:26:20: too few values in struct initializer
./server_test.go:65:20: too few values in struct initializer
```

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

We need to update our code where we create a `StubPlayerStore` as we've added a new field

```
//server_test.go
store := StubPlayerStore{
        map[string]int{},
        nil,
}
```

```
--- FAIL: TestStoreWins (0.00s)
    --- FAIL: TestStoreWins/it_records_wins_when_POST (0.00s)
        server_test.go:80: got 0 calls to RecordWin want 1
```

Write enough code to make it pass

As we're only asserting the number of calls rather than the specific values it makes our initial iteration a little smaller.

We need to update `PlayerServer`'s idea of what a `PlayerStore` is by changing the interface if we're going to be able to call `RecordWin`.

```
//server.go
```

```go
type PlayerStore interface {
	GetPlayerScore(name string) int

	RecordWin(name string)
}
```

By doing this `main` no longer compiles

```
./main.go:17:46: cannot use InMemoryPlayerStore literal (type *InMemoryPlayerStore)
    *InMemoryPlayerStore does not implement PlayerStore (missing RecordWin method)
```

The compiler tells us what's wrong. Let's update `InMemoryPlayerStore` to have that method.

```go
//main.go
type InMemoryPlayerStore struct{}

func (i *InMemoryPlayerStore) RecordWin(name string) {}
```

Try and run the tests and we should be back to compiling code - but the test is still failing.

Now that `PlayerStore` has `RecordWin` we can call it within our `PlayerServer`

```go
//server.go
func (p *PlayerServer) processWin(w http.ResponseWriter) {
	p.store.RecordWin("Bob")
	w.WriteHeader(http.StatusAccepted)
}
```

Run the tests and it should be passing! Obviously `"Bob"` isn't exactly what we want to send to `RecordWin`, so let's further refine the test.

Write the test first

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

```
	store := StubPlayerStore{
		map[string]int{},
		nil,
	}
	server := &PlayerServer{&store}

	t.Run("it records wins on POST", func(t *testing.T) {
		player := "Pepper"

		request, _ := http.NewRequest(http.MethodPost, fmt.Sprintf("/players/
		response := httptest.NewRecorder()

		server.ServeHTTP(response, request)

		assertStatus(t, response.Code, http.StatusAccepted)

		if len(store.winCalls) != 1 {
			t.Fatalf("got %d calls to RecordWin want %d", len(store.winCa
		}

		if store.winCalls[0] != player {
			t.Errorf("did not store correct winner got %q want %q", store
		}
	})
}
```

Now that we know there is one element in our `winCalls` slice we can safely reference the first one and check it is equal to `player`.

Try to run the test

```
=== RUN   TestStoreWins/it_records_wins_on_POST
    --- FAIL: TestStoreWins/it_records_wins_on_POST (0.00s)
        server_test.go:86: did not store correct winner got 'Bob' want 'Pepper'
```

```go
//server.go
func (p *PlayerServer) processWin(w http.ResponseWriter, r *http.Request) {
	player := strings.TrimPrefix(r.URL.Path, "/players/")
	p.store.RecordWin(player)
	w.WriteHeader(http.StatusAccepted)
}
```

We changed `processWin` to take `http.Request` so we can look at the URL to extract the player's name. Once we have that we can call our `store` with the correct value to make the test pass.

Refactor

We can DRY up this code a bit as we're extracting the player name the same way in two places

```go
//server.go
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)
}
```

Even though our tests are passing we don't really have working software. If you try and run `main` and use the software as intended it doesn't work because we haven't got round to implementing `PlayerStore` correctly. This is fine though; by focusing on our handler we have identified the interface that we need, rather than trying to design it up-front.

We *could* start writing some tests around our `InMemoryPlayerStore` but it's only here temporarily until we implement a more robust way of persisting player scores (i.e. a database).

What we'll do for now is write an *integration test* between our `PlayerServer` and `InMemoryPlayerStore` to finish off the functionality. This will let us get to our goal of being confident our application is working, without having to directly test `InMemoryPlayerStore`. Not only that, but when we get around to implementing `PlayerStore` with a database, we can test that implementation with the same integration test.

Integration tests

Integration tests can be useful for testing that larger areas of your system work but you must bear in mind:

- They are harder to write
- When they fail, it can be difficult to know why (usually it's a bug within a component of the integration test) and so can be harder to fix
- They are sometimes slower to run (as they often are used with "real" components, like a database)

For that reason, it is recommended that you research *The Test Pyramid*.

Write the test first

In the interest of brevity, I am going to show you the final refactored integration test.

```go
//server_integration_test.go
package main

```

```
import ("net/http"

	"net/http/httptest"
	"testing"
)

func TestRecordingWinsAndRetrievingThem(t *testing.T) {
	store := InMemoryPlayerStore{}
	server := PlayerServer{&store}
	player := "Pepper"

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

	response := httptest.NewRecorder()
	server.ServeHTTP(response, newGetScoreRequest(player))
	assertStatus(t, response.Code, http.StatusOK)

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

- We are creating our two components we are trying to integrate with: `InMemoryPlayerStore` and `PlayerServer`.
- We then fire off 3 requests to record 3 wins for `player`. We're not too concerned about the status codes in this test as it's not relevant to whether they are integrating well.
- The next response we do care about (so we store a variable `response`) because we are going to try and get the `player`'s score.

Try to run the test

```
--- FAIL: TestRecordingWinsAndRetrievingThem (0.00s)
    server_integration_test.go:24: response body is wrong, got '123' want '3'
```

Write enough code to make it pass

I am going to take some liberties here and write more code than you may be comfortable with without writing a test.

This is allowed! We still have a test checking things should be working correctly but it is not around the specific unit we're working with (`InMemoryPlayerStore`).

If I were to get stuck in this scenario, I would revert my changes back to the failing test and then write more specific unit tests around `InMemoryPlayerStore` to help me drive out a solution.

```go
//in_memory_player_store.go
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]
}
```

- We need to store the data so I've added a `map[string]int` to the `InMemoryPlayerStore` struct
- For convenience I've made `NewInMemoryPlayerStore` to initialise the store, and updated the integration test to use it:

  ```go
  //server_integration_test.go
  store := NewInMemoryPlayerStore()
  server := PlayerServer{store}
  ```

- The rest of the code is just wrapping around the `map`

The integration test passes, now we just need to change `main` to use `NewInMemoryPlayerStore()`

```go
//main.go
package main

import (
	"log"
	"net/http"
)

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

Build it, run it and then use `curl` to test it out.

- Run this a few times, change the player names if you like `curl -X POST http://localhost:5000/players/Pepper`
- Check scores with `curl http://localhost:5000/players/Pepper`

Great! You've made a REST-ish service. To take this forward you'd want to pick a data store to persist the scores longer than the length of time the program runs.

- Pick a store (Bolt? Mongo? Postgres? File system?)
- Make `PostgresPlayerStore` implement `PlayerStore`
- TDD the functionality so you're sure it works
- Plug it into the integration test, check it's still ok
- Finally plug it into `main`

Refactor

We are almost there! Lets take some effort to prevent concurrency errors like these

```
fatal error: concurrent map read and map write
```

By adding mutexes, we enforce concurrency safety especially for the counter in our `RecordWin` function. Read more about mutexes in the sync chapter.

Wrapping up

`http.Handler`

- Implement this interface to create web servers
- Use `http.HandlerFunc` to turn ordinary functions into `http.Handler`s
- Use `httptest.NewRecorder` to pass in as a `ResponseWriter` to let you spy on the responses your handler sends
- Use `http.NewRequest` to construct the requests you expect to come in to your system

Interfaces, Mocking and DI

- Lets you iteratively build the system up in smaller chunks
- Allows you to develop a handler that needs a storage without needing actual storage
- TDD to drive out the interfaces you need