IO and sorting

You can find all the code for this chapter here

In the previous chapter we continued iterating on our application by adding a new endpoint `/league`. Along the way we learned about how to deal with JSON, embedding types and routing.

Our product owner is somewhat perturbed by the software losing the scores when the server was restarted. This is because our implementation of our store is in-memory. She is also not pleased that we didn't interpret the `/league` endpoint should return the players ordered by the number of wins!

The code so far

```
// server.go
package main

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

// PlayerStore stores score information about players
type PlayerStore interface {
        GetPlayerScore(name string) int
        RecordWin(name string)
        GetLeague() []Player
}

// Player stores a name with a number of wins
type Player struct {
        Name string
        Wins int
}

```

```
// PlayerServer is a HTTP interface for player information
type PlayerServer struct {

        store PlayerStore
        http.Handler
}

const jsonContentType = "application/json"

// NewPlayerServer creates a PlayerServer with routing configured
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
}

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

func (p *PlayerServer) playersHandler(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) GetLeague() []Player {
        var league []Player
        for name, wins := range i.store {
                league = append(league, Player{name, wins})
        }
        return league
}

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 := NewPlayerServer(NewInMemoryPlayerStore())
        log.Fatal(http.ListenAndServe(":5000", server))
}
```

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

Store the data

There are dozens of databases we could use for this but we're going to go for a very simple approach. We're going to store the data for this application in a file as JSON.

This keeps the data very portable and is relatively simple to implement.

It won't scale especially well but given this is a prototype it'll be fine for now. If our circumstances change and it's no longer appropriate it'll be simple to swap it out for something different because of the `PlayerStore` abstraction we have used.

We will keep the `InMemoryPlayerStore` for now so that the integration tests keep passing as we develop our new store. Once we are confident our new implementation is sufficient to make the integration test pass we will swap it in and then delete `InMemoryPlayerStore`.

Write the test first

By now you should be familiar with the interfaces around the standard library for reading data (`io.Reader`), writing data (`io.Writer`) and how we can use the standard library to test these functions without having to use real files.

For this work to be complete we'll need to implement `PlayerStore` so we'll write tests for our store calling the methods we need to implement. We'll start with `GetLeague`.

```
//file_system_store_test.go
func TestFileSystemStore(t *testing.T) {
```

```
        t.Run("league from a reader", func(t *testing.T) {

                database := strings.NewReader(`[
            {"Name": "Cleo", "Wins": 10},
            {"Name": "Chris", "Wins": 33}]`)

                store := FileSystemPlayerStore{database}

                got := store.GetLeague()

                want := []Player{
                        {"Cleo", 10},
                        {"Chris", 33},
                }

                assertLeague(t, got, want)
        })
`
```

We're using `strings.NewReader` which will return us a `Reader`, which is what our `FileSystemPlayerStore` will use to read data. In `main` we will open a file, which is also a `Reader`.

Try to run the test

```
# github.com/quii/learn-go-with-tests/io/v1
./file_system_store_test.go:15:12: undefined: FileSystemPlayerStore
```

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

Let's define `FileSystemPlayerStore` in a new file

```
//file_system_store.go
```

```
2 type FileSystemPlayerStore struct{}
```

Try again

```
1 # github.com/quii/learn-go-with-tests/io/v1
2 ./file_system_store_test.go:15:28: too many values in struct initializer
3 ./file_system_store_test.go:17:15: store.GetLeague undefined (type FileSystemPlayerSt
```

It's complaining because we're passing in a `Reader` but not expecting one and it doesn't have `GetLeague` defined yet.

```
1 //file_system_store.go
2 type FileSystemPlayerStore struct {
3         database io.Reader
4 }
5
6 func (f *FileSystemPlayerStore) GetLeague() []Player {
7         return nil
8 }
```

One more try...

```
1 === RUN   TestFileSystemStore//league_from_a_reader
2     --- FAIL: TestFileSystemStore//league_from_a_reader (0.00s)
3         file_system_store_test.go:24: got [] want [{Cleo 10} {Chris 33}]
```

Write enough code to make it pass

We've read JSON from a reader before

```
1 //file_system_store.go
2 func (f *FileSystemPlayerStore) GetLeague() []Player {
3         var league []Player
4         json.NewDecoder(f.database).Decode(&league)
5         return league
```

```
6 }
```

The test should pass.

Refactor

We *have* done this before! Our test code for the server had to decode the JSON from the response.

Let's try DRYing this up into a function.

Create a new file called `league.go` and put this inside.

```
1 //league.go
2 func NewLeague(rdr io.Reader) ([]Player, error) {
3         var league []Player
4         err := json.NewDecoder(rdr).Decode(&league)
5         if err != nil {
6                 err = fmt.Errorf("problem parsing league, %v", err)
7         }
8
9         return league, err
10 }
```

Call this in our implementation and in our test helper `getLeagueFromResponse` in `server_test.go`

```
1 //file_system_store.go
2 func (f *FileSystemPlayerStore) GetLeague() []Player {
3         league, _ := NewLeague(f.database)
4         return league
5 }
```

We haven't got a strategy yet for dealing with parsing errors but let's press on.

Seeking problems

There is a flaw in our implementation. First of all, let's remind ourselves how `io.Reader` is defined

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

With our file, you can imagine it reading through byte by byte until the end. What happens if you try to `Read` a second time?

Add the following to the end of our current test.

```
//file_system_store_test.go

// read again
got = store.GetLeague()
assertLeague(t, got, want)
```

We want this to pass, but if you run the test it doesn't.

The problem is our `Reader` has reached the end so there is nothing more to read. We need a way to tell it to go back to the start.

ReadSeeker is another interface in the standard library that can help.

```
type ReadSeeker interface {
        Reader
        Seeker
}
```

Remember embedding? This is an interface comprised of `Reader` and `Seeker`

```
type Seeker interface {
        Seek(offset int64, whence int) (int64, error)
}
```

This sounds good, can we change `FileSystemPlayerStore` to take this interface instead?

```
//file_system_store.go
```

```
type FileSystemPlayerStore struct {
        database io.ReadSeeker
}

func (f *FileSystemPlayerStore) GetLeague() []Player {
        f.database.Seek(0, 0)
        league, _ := NewLeague(f.database)
        return league
}
```

Try running the test, it now passes! Happily for us `string.NewReader` that we used in our test also implements `ReadSeeker` so we didn't have to make any other changes.

Next we'll implement `GetPlayerScore`.

Write the test first

```
//file_system_store_test.go
t.Run("get player score", func(t *testing.T) {
        database := strings.NewReader(`[
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}]`)

        store := FileSystemPlayerStore{database}

        got := store.GetPlayerScore("Chris")

        want := 33

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

Try to run the test

```
./file_system_store_test.go:38:15: store.GetPlayerScore undefined (type FileSystemPla
```

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

We need to add the method to our new type to get the test to compile.

```
//file_system_store.go
func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {
        return 0
}
```

Now it compiles and the test fails

```
=== RUN   TestFileSystemStore/get_player_score
    --- FAIL: TestFileSystemStore//get_player_score (0.00s)
        file_system_store_test.go:43: got 0 want 33
```

Write enough code to make it pass

We can iterate over the league to find the player and return their score

```
//file_system_store.go
func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {

        var wins int

        for _, player := range f.GetLeague() {
                if player.Name == name {
                        wins = player.Wins
                        break
```

```
            }
        }

        return wins
}
```

Refactor

You will have seen dozens of test helper refactorings so I'll leave this to you to make it work

```
//file_system_store_test.go
t.Run("get player score", func(t *testing.T) {
        database := strings.NewReader(`[
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}]`)

        store := FileSystemPlayerStore{database}

        got := store.GetPlayerScore("Chris")
        want := 33
        assertScoreEquals(t, got, want)
})
```

Finally, we need to start recording scores with `RecordWin`.

Write the test first

Our approach is fairly short-sighted for writes. We can't (easily) just update one "row" of JSON in a file. We'll need to store the *whole* new representation of our database on every write.

How do we write? We'd normally use a `Writer` but we already have our `ReadSeeker`. Potentially we could have two dependencies but the standard library already has an interface for us `ReadWriteSeeker` which lets us do all the things we'll need to do with a file.

Let's update our type

```
//file_system_store.go

type FileSystemPlayerStore struct {
        database io.ReadWriteSeeker
}
```

See if it compiles

```
./file_system_store_test.go:15:34: cannot use database (type *strings.Reader) as type
    *strings.Reader does not implement io.ReadWriteSeeker (missing Write method)
./file_system_store_test.go:36:34: cannot use database (type *strings.Reader) as type
    *strings.Reader does not implement io.ReadWriteSeeker (missing Write method)
```

It's not too surprising that `strings.Reader` does not implement `ReadWriteSeeker` so what do we do?

We have two choices

- Create a temporary file for each test. `*os.File` implements `ReadWriteSeeker`. The pro of this is it becomes more of an integration test, we're really reading and writing from the file system so it will give us a very high level of confidence. The cons are we prefer unit tests because they are faster and generally simpler. We will also need to do more work around creating temporary files and then making sure they're removed after the test.
- We could use a third party library. Mattetti has written a library filebuffer which implements the interface we need and doesn't touch the file system.

I don't think there's an especially wrong answer here, but by choosing to use a third party library I would have to explain dependency management! So we will use files instead.

Before adding our test we need to make our other tests compile by replacing the `strings.Reader` with an `os.File`.

Let's create some helper functions which will create a temporary file with some data inside it, and abstract our score tests

```
//file_system_store_test.go
```

```
func createTempFile(t testing.TB, initialData string) (io.ReadWriteSeeker, func()) {
        t.Helper()

        tmpfile, err := ioutil.TempFile("", "db")

        if err != nil {
                t.Fatalf("could not create temp file %v", err)
        }

        tmpfile.Write([]byte(initialData))

        removeFile := func() {
                tmpfile.Close()
                os.Remove(tmpfile.Name())
        }

        return tmpfile, removeFile
}

func assertScoreEquals(t testing.TB, got, want int) {
        t.Helper()
        if got != want {
                t.Errorf("got %d want %d", got, want)
        }
}
```

TempFile creates a temporary file for us to use. The `"db"` value we've passed in is a prefix put on a random file name it will create. This is to ensure it won't clash with other files by accident.

You'll notice we're not only returning our `ReadWriteSeeker` (the file) but also a function. We need to make sure that the file is removed once the test is finished. We don't want to leak details of the files into the test as it's prone to error and uninteresting for the reader. By returning a `removeFile` function, we can take care of the details in our helper and all the caller has to do is run `defer cleanDatabase()`.

```
//file_system_store_test.go
func TestFileSystemStore(t *testing.T) {

        t.Run("league from a reader", func(t *testing.T) {
                database, cleanDatabase := createTempFile(t, `[
            {"Name": "Cleo", "Wins": 10},
            {"Name": "Chris", "Wins": 33}]`)
                defer cleanDatabase()
```

```
9
10                store := FileSystemPlayerStore{database}
11
12                got := store.GetLeague()
13
14                want := []Player{
15                        {"Cleo", 10},
16                        {"Chris", 33},
17                }
18
19                assertLeague(t, got, want)
20
21                // read again
22                got = store.GetLeague()
23                assertLeague(t, got, want)
24        })
25
26        t.Run("get player score", func(t *testing.T) {
27                database, cleanDatabase := createTempFile(t, `[
28            {"Name": "Cleo", "Wins": 10},
29            {"Name": "Chris", "Wins": 33}]`)
30                defer cleanDatabase()
31
32                store := FileSystemPlayerStore{database}
33
34                got := store.GetPlayerScore("Chris")
35                want := 33
36                assertScoreEquals(t, got, want)
37        })
38 }
```

Run the tests and they should be passing! There were a fair amount of changes but now it feels like we have our interface definition complete and it should be very easy to add new tests from now.

Let's get the first iteration of recording a win for an existing player

```
1 //file_system_store_test.go
2 t.Run("store wins for existing players", func(t *testing.T) {
3         database, cleanDatabase := createTempFile(t, `[
4         {"Name": "Cleo", "Wins": 10},
5         {"Name": "Chris", "Wins": 33}]`)
6         defer cleanDatabase()
7
```

```
        store := FileSystemPlayerStore{database}

        store.RecordWin("Chris")

        got := store.GetPlayerScore("Chris")
        want := 34
        assertScoreEquals(t, got, want)
})
```

Try to run the test

`./file_system_store_test.go:67:8: store.RecordWin undefined (type FileSystemPlayerStore has no field or method RecordWin)`

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

Add the new method

```
//file_system_store.go
func (f *FileSystemPlayerStore) RecordWin(name string) {

}
```

```
=== RUN   TestFileSystemStore/store_wins_for_existing_players
    --- FAIL: TestFileSystemStore/store_wins_for_existing_players (0.00s)
        file_system_store_test.go:71: got 33 want 34
```

Our implementation is empty so the old score is getting returned.

Write enough code to make it pass

```
//file_system_store.go
func (f *FileSystemPlayerStore) RecordWin(name string) {
        league := f.GetLeague()

        for i, player := range league {
                if player.Name == name {
                        league[i].Wins++
                }
        }

        f.database.Seek(0, 0)
        json.NewEncoder(f.database).Encode(league)
}
```

You may be asking yourself why I am doing `league[i].Wins++` rather than `player.Wins++`.

When you `range` over a slice you are returned the current index of the loop (in our case `i`) and a *copy* of the element at that index. Changing the `Wins` value of a copy won't have any effect on the `league` slice that we iterate on. For that reason, we need to get the reference to the actual value by doing `league[i]` and then changing that value instead.

If you run the tests, they should now be passing.

Refactor

In `GetPlayerScore` and `RecordWin`, we are iterating over `[]Player` to find a player by name.

We could refactor this common code in the internals of `FileSystemStore` but to me, it feels like this is maybe useful code we can lift into a new type. Working with a "League" so far has always been with `[]Player` but we can create a new type called `League`. This will be easier for other developers to understand and then we can attach useful methods onto that type for us to use.

Inside `league.go` add the following

```
//league.go
type League []Player

```

```
func (l League) Find(name string) *Player {
        for i, p := range l {
                if p.Name == name {
                        return &l[i]
                }
        }
        return nil
}
```

Now if anyone has a `League` they can easily find a given player.

Change our `PlayerStore` interface to return `League` rather than `[]Player`. Try to re-run the tests, you'll get a compilation problem because we've changed the interface but it's very easy to fix; just change the return type from `[]Player` to `League`.

This lets us simplify our methods in `file_system_store`.

```
//file_system_store.go
func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {

        player := f.GetLeague().Find(name)

        if player != nil {
                return player.Wins
        }

        return 0
}

func (f *FileSystemPlayerStore) RecordWin(name string) {
        league := f.GetLeague()
        player := league.Find(name)

        if player != nil {
                player.Wins++
        }

        f.database.Seek(0, 0)
        json.NewEncoder(f.database).Encode(league)
}
```

This is looking much better and we can see how we might be able to find other useful functionality around `League` can be refactored.
We now need to handle the scenario of recording wins of new players.

Write the test first

```
//file_system_store_test.go
t.Run("store wins for new players", func(t *testing.T) {
        database, cleanDatabase := createTempFile(t, `[
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}]`)
        defer cleanDatabase()

        store := FileSystemPlayerStore{database}

        store.RecordWin("Pepper")

        got := store.GetPlayerScore("Pepper")
        want := 1
        assertScoreEquals(t, got, want)
})
```

Try to run the test

```
=== RUN   TestFileSystemStore/store_wins_for_new_players#01
    --- FAIL: TestFileSystemStore/store_wins_for_new_players#01 (0.00s)
        file_system_store_test.go:86: got 0 want 1
```

Write enough code to make it pass

We just need to handle the scenario where `Find` returns `nil` because it couldn't find the player.

```
//file_system_store.go
func (f *FileSystemPlayerStore) RecordWin(name string) {
        league := f.GetLeague()
        player := league.Find(name)

        if player != nil {
                player.Wins++
        } else {
                league = append(league, Player{name, 1})
        }

        f.database.Seek(0, 0)
        json.NewEncoder(f.database).Encode(league)
}
```

The happy path is looking ok so we can now try using our new `Store` in the integration test. This will give us more confidence that the software works and then we can delete the redundant `InMemoryPlayerStore`.

In `TestRecordingWinsAndRetrievingThem` replace the old store.

```
//server_integration_test.go
database, cleanDatabase := createTempFile(t, "")
defer cleanDatabase()
store := &FileSystemPlayerStore{database}
```

If you run the test it should pass and now we can delete `InMemoryPlayerStore`. `main.go` will now have compilation problems which will motivate us to now use our new store in the "real" code.

```
//main.go
package main

import (
        "log"
        "net/http"
        "os"
```

```
)

const dbFileName = "game.db.json"

func main() {
        db, err := os.OpenFile(dbFileName, os.O_RDWR|os.O_CREATE, 0666)

        if err != nil {
                log.Fatalf("problem opening %s %v", dbFileName, err)
        }

        store := &FileSystemPlayerStore{db}
        server := NewPlayerServer(store)

        if err := http.ListenAndServe(":5000", server); err != nil {
                log.Fatalf("could not listen on port 5000 %v", err)
        }
}
```

- We create a file for our database.
- The 2nd argument to `os.OpenFile` lets you define the permissions for opening the file, in our case `O_RDWR` means we want to read and write *and* `os.O_CREATE` means create the file if it doesn't exist.
- The 3rd argument means sets permissions for the file, in our case, all users can read and write the file. (See superuser.com for a more detailed explanation).

Running the program now persists the data in a file in between restarts, hooray!

More refactoring and performance concerns

Every time someone calls `GetLeague()` or `GetPlayerScore()` we are reading the entire file and parsing it into JSON. We should not have to do that because `FileSystemStore` is entirely responsible for the state of the league; it should only need to read the file when the program starts up and only need to update the file when data changes.

We can create a constructor which can do some of this initialisation for us and store the league as a value in our `FileSystemStore` to be used on the reads instead.

```
//file_system_store.go
type FileSystemPlayerStore struct {

        database io.ReadWriteSeeker
        league   League
}

func NewFileSystemPlayerStore(database io.ReadWriteSeeker) *FileSystemPlayerStore {
        database.Seek(0, 0)
        league, _ := NewLeague(database)
        return &FileSystemPlayerStore{
                database: database,
                league:   league,
        }
}
```

This way we only have to read from disk once. We can now replace all of our previous calls to getting the league from disk and just use `f.league` instead.

```
//file_system_store.go
func (f *FileSystemPlayerStore) GetLeague() League {
        return f.league
}

func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {

        player := f.league.Find(name)

        if player != nil {
                return player.Wins
        }

        return 0
}

func (f *FileSystemPlayerStore) RecordWin(name string) {
        player := f.league.Find(name)

        if player != nil {
                player.Wins++
        } else {
                f.league = append(f.league, Player{name, 1})
        }

```

```
        f.database.Seek(0, 0)
        json.NewEncoder(f.database).Encode(f.league)
}
```

If you try to run the tests it will now complain about initialising `FileSystemPlayerStore` so just fix them by calling our new constructor.

Another problem

There is some more naivety in the way we are dealing with files which *could* create a very nasty bug down the line.

When we `RecordWin`, we `Seek` back to the start of the file and then write the new data—but what if the new data was smaller than what was there before?

In our current case, this is impossible. We never edit or delete scores so the data can only get bigger. However, it would be irresponsible for us to leave the code like this; it's not unthinkable that a delete scenario could come up.

How will we test for this though? What we need to do is first refactor our code so we separate out the concern of the *kind of data we write, from the writing*. We can then test that separately to check it works how we hope.

We'll create a new type to encapsulate our "when we write we go from the beginning" functionality. I'm going to call it `Tape`. Create a new file with the following:

```
//tape.go
package main

import "io"

type tape struct {
        file io.ReadWriteSeeker
}

func (t *tape) Write(p []byte) (n int, err error) {
        t.file.Seek(0, 0)
        return t.file.Write(p)
}
```

Notice that we're only implementing `Write` now, as it encapsulates the `Seek` part. This means our

`FileSystemStore` can just have a reference to a `Writer` instead.

```
//file_system_store.go
type FileSystemPlayerStore struct {
        database io.Writer
        league   League
}
```

Update the constructor to use `Tape`

```
//file_system_store.go
func NewFileSystemPlayerStore(database io.ReadWriteSeeker) *FileSystemPlayerStore {
        database.Seek(0, 0)
        league, _ := NewLeague(database)

        return &FileSystemPlayerStore{
                database: &tape{database},
                league:   league,
        }
}
```

Finally, we can get the amazing payoff we wanted by removing the `Seek` call from `RecordWin`. Yes, it doesn't feel much, but at least it means if we do any other kind of writes we can rely on our `Write` to behave how we need it to. Plus it will now let us test the potentially problematic code separately and fix it.

Let's write the test where we want to update the entire contents of a file with something that is smaller than the original contents.

Write the test first

Our test will create a file with some content, try to write to it using the `tape`, and read it all again to see what's in the file. In `tape_test.go`:

```
//tape_test.go
func TestTape_Write(t *testing.T) {
```

```
        file, clean := createTempFile(t, "12345")
        defer clean()

        tape := &tape{file}

        tape.Write([]byte("abc"))

        file.Seek(0, 0)
        newFileContents, _ := ioutil.ReadAll(file)

        got := string(newFileContents)
        want := "abc"

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

Try to run the test

```
=== RUN   TestTape_Write
--- FAIL: TestTape_Write (0.00s)
    tape_test.go:23: got 'abc45' want 'abc'
```

As we thought! It writes the data we want, but leaves the rest of the original data remaining.

Write enough code to make it pass

`os.File` has a truncate function that will let us effectively empty the file. We should be able to just call this to get what we want.

Change `tape` to the following:

```
//tape.go
type tape struct {
```

```
}       file *os.File


func (t *tape) Write(p []byte) (n int, err error) {
        t.file.Truncate(0)
        t.file.Seek(0, 0)
        return t.file.Write(p)
}
```

The compiler will fail in a number of places where we are expecting an `io.ReadWriteSeeker` but we are sending in `*os.File`. You should be able to fix these problems yourself by now but if you get stuck just check the source code.

Once you get it refactoring our `TestTape_Write` test should be passing!

One other small refactor

In `RecordWin` we have the line `json.NewEncoder(f.database).Encode(f.league)`.

We don't need to create a new encoder every time we write, we can initialise one in our constructor and use that instead.

Store a reference to an `Encoder` in our type and initialise it in the constructor:

```
//file_system_store.go
type FileSystemPlayerStore struct {
        database *json.Encoder
        league   League
}

func NewFileSystemPlayerStore(file *os.File) *FileSystemPlayerStore {
        file.Seek(0, 0)
        league, _ := NewLeague(file)

        return &FileSystemPlayerStore{
                database: json.NewEncoder(&tape{file}),
                league:   league,
        }
}
```

Use it in `RecordWin`.

```go
func (f *FileSystemPlayerStore) RecordWin(name string) {

        player := f.league.Find(name)

        if player != nil {
                player.Wins++
        } else {
                f.league = append(f.league, Player{name, 1})
        }

        f.database.Encode(f.league)
}
```

Didn't we just break some rules there? Testing private things? No interfaces?

On testing private types

It's true that *in general* you should favour not testing private things as that can sometimes lead to your tests being too tightly coupled to the implementation, which can hinder refactoring in future.

However, we must not forget that tests should give us *confidence*.

We were not confident that our implementation would work if we added any kind of edit or delete functionality. We did not want to leave the code like that, especially if this was being worked on by more than one person who may not be aware of the shortcomings of our initial approach.

Finally, it's just one test! If we decide to change the way it works it won't be a disaster to just delete the test but we have at the very least captured the requirement for future maintainers.

Interfaces

We started off the code by using `io.Reader` as that was the easiest path for us to unit test our new `PlayerStore`. As we developed the code we moved on to `io.ReadWriter` and then `io.ReadWriteSeeker`. We then found out there was nothing in the standard library that actually implemented that apart from `*os.File`. We could've taken the decision to write our own or use an open source one but it felt pragmatic just to make temporary files for the tests.
Finally, we needed `Truncate` which is also on `*os.File`. It would've been an option to create our own interface capturing these requirements.

```
type ReadWriteSeekTruncate interface {
        io.ReadWriteSeeker
        Truncate(size int64) error
}
```

But what is this really giving us? Bear in mind we are *not mocking* and it is unrealistic for a **file system** store to take any type other than an `*os.File` so we don't need the polymorphism that interfaces give us.

Don't be afraid to chop and change types and experiment like we have here. The great thing about using a statically typed language is the compiler will help you with every change.

Error handling

Before we start working on sorting we should make sure we're happy with our current code and remove any technical debt we may have. It's an important principle to get to working software as quickly as possible (stay out of the red state) but that doesn't mean we should ignore error cases!

If we go back to `FileSystemStore.go` we have `league, _ := NewLeague(f.database)` in our constructor.

`NewLeague` can return an error if it is unable to parse the league from the `io.Reader` that we provide.

It was pragmatic to ignore that at the time as we already had failing tests. If we had tried to tackle it at the same time, we would have been juggling two things at once.

Let's make it so our constructor is capable of returning an error.

```
//file_system_store.go
```

```
func NewFileSystemPlayerStore(file *os.File) (*FileSystemPlayerStore, error) {
        file.Seek(0, 0)

        league, err := NewLeague(file)

        if err != nil {
                return nil, fmt.Errorf("problem loading player store from file %s, %v
        }

        return &FileSystemPlayerStore{
                database: json.NewEncoder(&tape{file}),
                league:   league,
        }, nil
}
```

Remember it is very important to give helpful error messages (just like your tests). People on the internet jokingly say that most Go code is:

```
if err != nil {
        return err
}
```

That is 100% not idiomatic. Adding contextual information (i.e what you were doing to cause the error) to your error messages makes operating your software far easier.

If you try to compile you'll get some errors.

```
./main.go:18:35: multiple-value NewFileSystemPlayerStore() in single-value context
./file_system_store_test.go:35:36: multiple-value NewFileSystemPlayerStore() in singl
./file_system_store_test.go:57:36: multiple-value NewFileSystemPlayerStore() in singl
./file_system_store_test.go:70:36: multiple-value NewFileSystemPlayerStore() in singl
./file_system_store_test.go:85:36: multiple-value NewFileSystemPlayerStore() in singl
./server_integration_test.go:12:35: multiple-value NewFileSystemPlayerStore() in sing
```

In main we'll want to exit the program, printing the error.

```
//main.go
store, err := NewFileSystemPlayerStore(db)

if err != nil {
        log.Fatalf("problem creating file system player store, %v ", err)
```

```
}
```

In the tests we should assert there is no error. We can make a helper to help with this.

```
//file_system_store_test.go
func assertNoError(t testing.TB, err error) {
        t.Helper()
        if err != nil {
                t.Fatalf("didn't expect an error but got one, %v", err)
        }
}
```

Work through the other compilation problems using this helper. Finally, you should have a failing test:

```
=== RUN   TestRecordingWinsAndRetrievingThem
--- FAIL: TestRecordingWinsAndRetrievingThem (0.00s)
    server_integration_test.go:14: didn't expect an error but got one, problem loadin
```

We cannot parse the league because the file is empty. We weren't getting errors before because we always just ignored them.

Let's fix our big integration test by putting some valid JSON in it:

```
//server_integration_test.go
func TestRecordingWinsAndRetrievingThem(t *testing.T) {
        database, cleanDatabase := createTempFile(t, `[]`)
        //etc...
}
```

Now that all the tests are passing, we need to handle the scenario where the file is empty.

Write the test first

```
```

```
//file_system_store_test.go
t.Run("works with an empty file", func(t *testing.T) {

        database, cleanDatabase := createTempFile(t, "")
        defer cleanDatabase()

        _, err := NewFileSystemPlayerStore(database)

        assertNoError(t, err)
})
```

Try to run the test

```
=== RUN   TestFileSystemStore/works_with_an_empty_file
    --- FAIL: TestFileSystemStore/works_with_an_empty_file (0.00s)
        file_system_store_test.go:108: didn't expect an error but got one, problem lo
```

Write enough code to make it pass

Change our constructor to the following

```
//file_system_store.go
func NewFileSystemPlayerStore(file *os.File) (*FileSystemPlayerStore, error) {

        file.Seek(0, 0)

        info, err := file.Stat()

        if err != nil {
                return nil, fmt.Errorf("problem getting file info from file %s, %v",
        }

        if info.Size() == 0 {
                file.Write([]byte("[]"))
                file.Seek(0, 0)
        }
```

```
        league, err := NewLeague(file)

        if err != nil {
                return nil, fmt.Errorf("problem loading player store from file %s, %v
        }

        return &FileSystemPlayerStore{
                database: json.NewEncoder(&tape{file}),
                league:   league,
        }, nil
}
```

`file.Stat` returns stats on our file, which lets us check the size of the file. If it's empty, we `Write` an empty JSON array and `Seek` back to the start, ready for the rest of the code.

Refactor

Our constructor is a bit messy now, so let's extract the initialise code into a function:

```
//file_system_store.go
func initialisePlayerDBFile(file *os.File) error {
        file.Seek(0, 0)

        info, err := file.Stat()

        if err != nil {
                return fmt.Errorf("problem getting file info from file %s, %v", file.
        }

        if info.Size() == 0 {
                file.Write([]byte("[]"))
                file.Seek(0, 0)
        }

        return nil
}
```

```
//file_system_store.go
func NewFileSystemPlayerStore(file *os.File) (*FileSystemPlayerStore, error) {

        err := initialisePlayerDBFile(file)

        if err != nil {
                return nil, fmt.Errorf("problem initialising player db file, %v", err
        }

        league, err := NewLeague(file)

        if err != nil {
                return nil, fmt.Errorf("problem loading player store from file %s, %v
        }

        return &FileSystemPlayerStore{
                database: json.NewEncoder(&tape{file}),
                league:   league,
        }, nil
}
```

Sorting

Our product owner wants `/league` to return the players sorted by their scores, from highest to lowest.

The main decision to make here is where in the software should this happen. If we were using a "real" database we would use things like `ORDER BY` so the sorting is super fast. For that reason, it feels like implementations of `PlayerStore` should be responsible.

Write the test first

We can update the assertion on our first test in `TestFileSystemStore`:

```
//file_system_store_test.go
t.Run("league sorted", func(t *testing.T) {
        database, cleanDatabase := createTempFile(t, `[
```

```
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}]`)
        defer cleanDatabase()

        store, err := NewFileSystemPlayerStore(database)

        assertNoError(t, err)

        got := store.GetLeague()

        want := []Player{
                {"Chris", 33},
                {"Cleo", 10},
        }

        assertLeague(t, got, want)

        // read again
        got = store.GetLeague()
        assertLeague(t, got, want)
})
```

The order of the JSON coming in is in the wrong order and our `want` will check that it is returned to the caller in the correct order.

Try to run the test

```
=== RUN   TestFileSystemStore/league_from_a_reader,_sorted
    --- FAIL: TestFileSystemStore/league_from_a_reader,_sorted (0.00s)
        file_system_store_test.go:46: got [{Cleo 10} {Chris 33}] want [{Chris 33} {Cl
        file_system_store_test.go:51: got [{Cleo 10} {Chris 33}] want [{Chris 33} {Cl
```

Write enough code to make it pass

```
func (f *FileSystemPlayerStore) GetLeague() League {
    sort.Slice(f.league, func(i, j int) bool {
        return f.league[i].Wins > f.league[j].Wins
    })
    return f.league
}
```

`sort.Slice`

> Slice sorts the provided slice given the provided less function.

Easy!

Wrapping up

What we've covered

- The `Seeker` interface and its relation to `Reader` and `Writer`.
- Working with files.
- Creating an easy to use helper for testing with files that hides all the messy stuff.
- `sort.Slice` for sorting slices.
- Using the compiler to help us safely make structural changes to the application.

Breaking rules

- Most rules in software engineering aren't really rules, just best practices that work 80% of the time.
- We discovered a scenario where one of our previous "rules" of not testing internal functions was not helpful for us so we broke the rule.
- It's important when breaking rules to understand the trade-off you are making. In our case, we were ok with it because it was just one test and would've been very difficult to exercise the scenario otherwise.
- In order to be able to break the rules **you must understand them first**. An analogy is with learning guitar. It doesn't matter how creative you think you are, you must understand and

practice the fundamentals.

Where our software is at

- We have an HTTP API where you can create players and increment their score.
- We can return a league of everyone's scores as JSON.
- The data is persisted as a JSON file.