WebSockets

You can find all the code for this chapter here

In this chapter we'll learn how to use WebSockets to improve our application.

Project recap

We have two applications in our poker codebase

- *Command line app*. Prompts the user to enter the number of players in a game. From then on informs the players of what the "blind bet" value is, which increases over time. At any point a user can enter `"{Playername} wins"` to finish the game and record the victor in a store.
- *Web app*. Allows users to record winners of games and displays a league table. Shares the same store as the command line app.

Next steps

The product owner is thrilled with the command line application but would prefer it if we could bring that functionality to the browser. She imagines a web page with a text box that allows the user to enter the number of players and when they submit the form the page displays the blind value and automatically updates it when appropriate. Like the command line application the user can declare the winner and it'll get saved in the database.

On the face of it, it sounds quite simple but as always we must emphasise taking an *iterative* approach to writing software.

First we will need to serve HTML. So far all of our HTTP endpoints have returned either plaintext or JSON. We *could* use the same techniques we know (as they're all ultimately strings) but we can also use the html/template package for a cleaner solution.

We also need to be able to asynchronously send messages to the user saying `The blind is now *y*` without having to refresh the browser. We can use WebSockets to facilitate this.

> WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection

Given we are taking on a number of techniques it's even more important we do the smallest amount of useful work possible first and then iterate.

For that reason the first thing we'll do is create a web page with a form for the user to record a winner. Rather than using a plain form, we will use WebSockets to send that data to our server for it to record.

After that we'll work on the blind alerts by which point we will have a bit of infrastructure code set up.

What about tests for the JavaScript ?

There will be some JavaScript written to do this but I won't go in to writing tests.

It is of course possible but for the sake of brevity I won't be including any explanations for it.

Sorry folks. Lobby O'Reilly to pay me to make a "Learn JavaScript with tests".

Write the test first

First thing we need to do is serve up some HTML to users when they hit `/game`.

Here's a reminder of the pertinent code in our web server

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

const jsonContentType = "application/json"

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
}
```

The *easiest* thing we can do for now is check when we `GET /game` that we get a `200`.

```
func TestGame(t *testing.T) {
        t.Run("GET /game returns 200", func(t *testing.T) {
                server := NewPlayerServer(&StubPlayerStore{})

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

                server.ServeHTTP(response, request)

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

Try to run the test

```
--- FAIL: TestGame (0.00s)
=== RUN   TestGame/GET_/game_returns_200
    --- FAIL: TestGame/GET_/game_returns_200 (0.00s)
        server_test.go:109: did not get correct status, got 404, want 200
```

Write enough code to make it pass

Our server has a router setup so it's relatively easy to fix.

To our router add

```
router.Handle("/game", http.HandlerFunc(p.game))
```

And then write the `game` method

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

Refactor

The server code is already fine due to us slotting in more code into the existing well-factored code very easily.

We can tidy up the test a little by adding a test helper function `newGameRequest` to make the request to `/game`. Try writing this yourself.

```
func TestGame(t *testing.T) {
        t.Run("GET /game returns 200", func(t *testing.T) {
                server := NewPlayerServer(&StubPlayerStore{})

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

                server.ServeHTTP(response, request)

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

You'll also notice I changed `assertStatus` to accept `response` rather than `response.Code` as I feel it reads better.

Now we need to make the endpoint return some HTML, here it is

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Let's play poker</title>
</head>
<body>
<section id="game">
    <div id="declare-winner">
        <label for="winner">Winner</label>
        <input type="text" id="winner"/>
        <button id="winner-button">Declare winner</button>
    </div>
</section>
</body>
<script type="application/javascript">

    const submitWinnerButton = document.getElementById('winner-button')
    const winnerInput = document.getElementById('winner')

    if (window['WebSocket']) {
        const conn = new WebSocket('ws://' + document.location.host + '/ws')

        submitWinnerButton.onclick = event => {
            conn.send(winnerInput.value)
        }
    }
</script>
</html>
```

We have a very simple web page

- A text input for the user to enter the winner into
- A button they can click to declare the winner.
- Some JavaScript to open a WebSocket connection to our server and handle the submit button being pressed

`WebSocket` is built into most modern browsers so we don't need to worry about bringing in any libraries. The web page won't work for older browsers, but we're ok with that for this scenario.

How do we test we return the correct markup?

There are a few ways. As has been emphasised throughout the book, it is important that the tests you write have sufficient value to justify the cost.

1. Write a browser based test, using something like Selenium. These tests are the most "realistic" of all approaches because they start an actual web browser of some kind and simulates a user interacting with it. These tests can give you a lot of confidence your system works but are more difficult to write than unit tests and much slower to run. For the purposes of our product this is overkill.
2. Do an exact string match. This *can* be ok but these kind of tests end up being very brittle. The moment someone changes the markup you will have a test failing when in practice nothing has *actually broken*.
3. Check we call the correct template. We will be using a templating library from the standard lib to serve the HTML (discussed shortly) and we could inject in the *thing* to generate the HTML and spy on its call to check we're doing it right. This would have an impact on our code's design but doesn't actually test a great deal; other than we're calling it with the correct template file. Given we will only have the one template in our project the chance of failure here seems low.

So in the book "Learn Go with Tests" for the first time, we're not going to write a test.

Put the markup in a file called `game.html`

Next change the endpoint we just wrote to the following

```
1 func (p *PlayerServer) game(w http.ResponseWriter, r *http.Request) {
2         tmpl, err := template.ParseFiles("game.html")
3
4         if err != nil {
5                 http.Error(w, fmt.Sprintf("problem loading template %s", err.Error())
```

```
            return
        }

        tmpl.Execute(w, nil)
}
```

`html/template` is a Go package for creating HTML. In our case we call `template.ParseFiles`, giving the path of our html file. Assuming there is no error you can then `Execute` the template, which writes it to an `io.Writer`. In our case we want it to `Write` to the internet, so we give it our `http.ResponseWriter`.

As we have not written a test, it would be prudent to manually test our web server just to make sure things are working as we'd hope. Go to `cmd/webserver` and run the `main.go` file. Visit `http://localhost:5000/game`.

You *should* have got an error about not being able to find the template. You can either change the path to be relative to your folder, or you can have a copy of the `game.html` in the `cmd/webserver` directory. I chose to create a symlink (`ln -s ../../game.html game.html`) to the file inside the root of the project so if I make changes they are reflected when running the server.

If you make this change and run again you should see our UI.

Now we need to test that when we get a string over a WebSocket connection to our server that we declare it as a winner of a game.

Write the test first

For the first time we are going to use an external library so that we can work with WebSockets.

Run `go get github.com/gorilla/websocket`

This will fetch the code for the excellent Gorilla WebSocket library. Now we can update our tests for our new requirement.

```
t.Run("when we get a message over a websocket it is a winner of a game", func(t *test
        store := &StubPlayerStore{}
        winner := "Ruth"
        server := httptest.NewServer(NewPlayerServer(store))
        defer server.Close()
```

```
6
7         wsURL := "ws" + strings.TrimPrefix(server.URL, "http") + "/ws"
8
9         ws, _, err := websocket.DefaultDialer.Dial(wsURL, nil)
10        if err != nil {
11                t.Fatalf("could not open a ws connection on %s %v", wsURL, err)
12        }
13        defer ws.Close()
14
15        if err := ws.WriteMessage(websocket.TextMessage, []byte(winner)); err != nil
16                t.Fatalf("could not send message over ws connection %v", err)
17        }
18
19        AssertPlayerWin(t, store, winner)
20 })
```

Make sure that you have an import for the `websocket` library. My IDE automatically did it for me, so should yours.

To test what happens from the browser we have to open up our own WebSocket connection and write to it.

Our previous tests around our server just called methods on our server but now we need to have a persistent connection to our server. To do that we use `httptest.NewServer` which takes a `http.Handler` and will spin it up and listen for connections.

Using `websocket.DefaultDialer.Dial` we try to dial in to our server and then we'll try and send a message with our `winner`.

Finally, we assert on the player store to check the winner was recorded.

Try to run the test

```
1 === RUN   TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_game
2     --- FAIL: TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_gam
3         server_test.go:124: could not open a ws connection on ws://127.0.0.1:55838/ws
```

We have not changed our server to accept WebSocket connections on `/ws` so we're not shaking

hands yet.

Write enough code to make it pass

Add another listing to our router

```
router.Handle("/ws", http.HandlerFunc(p.webSocket))
```

Then add our new `webSocket` handler

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
        upgrader := websocket.Upgrader{
                ReadBufferSize:  1024,
                WriteBufferSize: 1024,
        }
        upgrader.Upgrade(w, r, nil)
}
```

To accept a WebSocket connection we `Upgrade` the request. If you now re-run the test you should move on to the next error.

```
=== RUN   TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_game
    --- FAIL: TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_gam
        server_test.go:132: got 0 calls to RecordWin want 1
```

Now that we have a connection opened, we'll want to listen for a message and then record it as the winner.

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
        upgrader := websocket.Upgrader{
                ReadBufferSize:  1024,
                WriteBufferSize: 1024,
        }
        conn, _ := upgrader.Upgrade(w, r, nil)
        _, winnerMsg, _ := conn.ReadMessage()
```

```
        p.store.RecordWin(string(winnerMsg))
}
```

(Yes, we're ignoring a lot of errors right now!)

`conn.ReadMessage()` blocks on waiting for a message on the connection. Once we get one we use it to `RecordWin`. This would finally close the WebSocket connection.

If you try and run the test, it's still failing.

The issue is timing. There is a delay between our WebSocket connection reading the message and recording the win and our test finishes before it happens. You can test this by putting a short `time.Sleep` before the final assertion.

Let's go with that for now but acknowledge that putting in arbitrary sleeps into tests **is very bad practice**.

```
time.Sleep(10 * time.Millisecond)
AssertPlayerWin(t, store, winner)
```

Refactor

We committed many sins to make this test work both in the server code and the test code but remember this is the easiest way for us to work.

We have nasty, horrible, *working* software backed by a test, so now we are free to make it nice and know we won't break anything accidentally.

Let's start with the server code.

We can move the `upgrader` to a private value inside our package because we don't need to redeclare it on every WebSocket connection request

```
var wsUpgrader = websocket.Upgrader{
        ReadBufferSize:  1024,
        WriteBufferSize: 1024,
}

```

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
        conn, _ := wsUpgrader.Upgrade(w, r, nil)

        _, winnerMsg, _ := conn.ReadMessage()
        p.store.RecordWin(string(winnerMsg))
}
```

Our call to `template.ParseFiles("game.html")` will run on every `GET /game` which means we'll go to the file system on every request even though we have no need to re-parse the template. Let's refactor our code so that we parse the template once in `NewPlayerServer` instead. We'll have to make it so this function can now return an error in case we have problems fetching the template from disk or parsing it.

Here's the relevant changes to `PlayerServer`

```
type PlayerServer struct {
        store PlayerStore
        http.Handler
        template *template.Template
}

const htmlTemplatePath = "game.html"

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

        tmpl, err := template.ParseFiles(htmlTemplatePath)

        if err != nil {
                return nil, fmt.Errorf("problem opening %s %v", htmlTemplatePath, err
        }

        p.template = tmpl
        p.store = store

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

        p.Handler = router

        return p, nil
```

```
}

func (p *PlayerServer) game(w http.ResponseWriter, r *http.Request) {
        p.template.Execute(w, nil)
}
```

By changing the signature of `NewPlayerServer` we now have compilation problems. Try and fix them yourself or refer to the source code if you struggle.

For the test code I made a helper called `mustMakePlayerServer(t *testing.T, store PlayerStore) *PlayerServer` so that I could hide the error noise away from the tests.

```
func mustMakePlayerServer(t *testing.T, store PlayerStore) *PlayerServer {
        server, err := NewPlayerServer(store)
        if err != nil {
                t.Fatal("problem creating player server", err)
        }
        return server
}
```

Similarly, I created another helper `mustDialWS` so that I could hide nasty error noise when creating the WebSocket connection.

```
func mustDialWS(t *testing.T, url string) *websocket.Conn {
        ws, _, err := websocket.DefaultDialer.Dial(url, nil)

        if err != nil {
                t.Fatalf("could not open a ws connection on %s %v", url, err)
        }

        return ws
}
```

Finally, in our test code we can create a helper to tidy up sending messages

```
func writeWSMessage(t testing.TB, conn *websocket.Conn, message string) {
        t.Helper()
        if err := conn.WriteMessage(websocket.TextMessage, []byte(message)); err != r
                t.Fatalf("could not send message over ws connection %v", err)
```

```
        }
}
```

Now the tests are passing try running the server and declare some winners in `/game`. You should see them recorded in `/league`. Remember that every time we get a winner we *close the connection*, you will need to refresh the page to open the connection again.

We've made a trivial web form that lets users record the winner of a game. Let's iterate on it to make it so the user can start a game by providing a number of players and the server will push messages to the client informing them of what the blind value is as time passes.

First update `game.html` to update our client side code for the new requirements

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Lets play poker</title>
</head>
<body>
<section id="game">
    <div id="game-start">
        <label for="player-count">Number of players</label>
        <input type="number" id="player-count"/>
        <button id="start-game">Start</button>
    </div>

    <div id="declare-winner">
        <label for="winner">Winner</label>
        <input type="text" id="winner"/>
        <button id="winner-button">Declare winner</button>
    </div>

    <div id="blind-value"/>
</section>

<section id="game-end">
    <h1>Another great game of poker everyone!</h1>
    <p><a href="/league">Go check the league table</a></p>
</section>

</body>
```

```
<script type="application/javascript">
    const startGame = document.getElementById('game-start')

    const declareWinner = document.getElementById('declare-winner')
    const submitWinnerButton = document.getElementById('winner-button')
    const winnerInput = document.getElementById('winner')

    const blindContainer = document.getElementById('blind-value')

    const gameContainer = document.getElementById('game')
    const gameEndContainer = document.getElementById('game-end')

    declareWinner.hidden = true
    gameEndContainer.hidden = true

    document.getElementById('start-game').addEventListener('click', event => {
        startGame.hidden = true
        declareWinner.hidden = false

        const numberOfPlayers = document.getElementById('player-count').value

        if (window['WebSocket']) {
            const conn = new WebSocket('ws://' + document.location.host + '/ws')

            submitWinnerButton.onclick = event => {
                conn.send(winnerInput.value)
                gameEndContainer.hidden = false
                gameContainer.hidden = true
            }

            conn.onclose = evt => {
                blindContainer.innerText = 'Connection closed'
            }

            conn.onmessage = evt => {
                blindContainer.innerText = evt.data
            }

            conn.onopen = function () {
                conn.send(numberOfPlayers)
            }
        }
    })
</script>
```

```
74
```

The main changes is bringing in a section to enter the number of players and a section to display the blind value. We have a little logic to show/hide the user interface depending on the stage of the game.

Any message we receive via `conn.onmessage` we assume to be blind alerts and so we set the `blindContainer.innerText` accordingly.

How do we go about sending the blind alerts? In the previous chapter we introduced the idea of `Game` so our CLI code could call a `Game` and everything else would be taken care of including scheduling blind alerts. This turned out to be a good separation of concern.

```
type Game interface {
        Start(numberOfPlayers int)
        Finish(winner string)
}
```

When the user was prompted in the CLI for number of players it would `Start` the game which would kick off the blind alerts and when the user declared the winner they would `Finish`. This is the same requirements we have now, just a different way of getting the inputs; so we should look to re-use this concept if we can.

Our "real" implementation of `Game` is `TexasHoldem`

```
type TexasHoldem struct {
        alerter BlindAlerter
        store   PlayerStore
}
```

By sending in a `BlindAlerter` `TexasHoldem` can schedule blind alerts to be sent to *wherever*

```
type BlindAlerter interface {
        ScheduleAlertAt(duration time.Duration, amount int)
}
```

And as a reminder, here is our implementation of the `BlindAlerter` we use in the CLI.

```
func StdOutAlerter(duration time.Duration, amount int) {
        time.AfterFunc(duration, func() {
                fmt.Fprintf(os.Stdout, "Blind is now %d\n", amount)
        })
}
```

This works in CLI because we *always want to send the alerts to* `os.Stdout` but this won't work for our web server. For every request we get a new `http.ResponseWriter` which we then upgrade to `*websocket.Conn`. So we can't know when constructing our dependencies where our alerts need to go.

For that reason we need to change `BlindAlerter.ScheduleAlertAt` so that it takes a destination for the alerts so that we can re-use it in our webserver.

Open blind_alerter.go and add the parameter `to io.Writer`

```
type BlindAlerter interface {
        ScheduleAlertAt(duration time.Duration, amount int, to io.Writer)
}

type BlindAlerterFunc func(duration time.Duration, amount int, to io.Writer)

func (a BlindAlerterFunc) ScheduleAlertAt(duration time.Duration, amount int, to io.W
        a(duration, amount, to)
}
```

The idea of a `StdoutAlerter` doesn't fit our new model so just rename it to `Alerter`

```
func Alerter(duration time.Duration, amount int, to io.Writer) {
        time.AfterFunc(duration, func() {
                fmt.Fprintf(to, "Blind is now %d\n", amount)
        })
}
```

If you try and compile, it will fail in `TexasHoldem` because it is calling `ScheduleAlertAt` without a destination, to get things compiling again *for now* hard-code it to `os.Stdout`.

Try and run the tests and they will fail because `SpyBlindAlerter` no longer implements `BlindAlerter`, fix this by updating the signature of `ScheduleAlertAt`, run the tests and we

should still be green.

It doesn't make any sense for `TexasHoldem` to know where to send blind alerts. Let's now update `Game` so that when you start a game you declare *where* the alerts should go.

```
type Game interface {
        Start(numberOfPlayers int, alertsDestination io.Writer)
        Finish(winner string)
}
```

Let the compiler tell you what you need to fix. The change isn't so bad:

- Update `TexasHoldem` so it properly implements `Game`
- In `CLI` when we start the game, pass in our `out` property (`cli.game.Start(numberOfPlayers, cli.out)`)
- In `TexasHoldem`'s test i use `game.Start(5, ioutil.Discard)` to fix the compilation problem and configure the alert output to be discarded

If you've got everything right, everything should be green! Now we can try and use `Game` within `Server`.

Write the test first

The requirements of `CLI` and `Server` are the same! It's just the delivery mechanism is different.

Let's take a look at our `CLI` test for inspiration.

```
t.Run("start game with 3 players and finish game with 'Chris' as winner", func(t *tes
        game := &GameSpy{}

        out := &bytes.Buffer{}
        in := userSends("3", "Chris wins")

        poker.NewCLI(in, out, game).PlayPoker()

        assertMessagesSentToUser(t, out, poker.PlayerPrompt)
        assertGameStartedWith(t, game, 3)
```

```
        assertFinishCalledWith(t, game, "Chris")
})
```

It looks like we should be able to test drive out a similar outcome using `GameSpy`

Replace the old websocket test with the following

```
t.Run("start a game with 3 players and declare Ruth the winner", func(t *testing.T) {
        game := &poker.GameSpy{}
        winner := "Ruth"
        server := httptest.NewServer(mustMakePlayerServer(t, dummyPlayerStore, game))
        ws := mustDialWS(t, "ws"+strings.TrimPrefix(server.URL, "http")+"/ws")

        defer server.Close()
        defer ws.Close()

        writeWSMessage(t, ws, "3")
        writeWSMessage(t, ws, winner)

        time.Sleep(10 * time.Millisecond)
        assertGameStartedWith(t, game, 3)
        assertFinishCalledWith(t, game, winner)
})
```

- As discussed we create a spy `Game` and pass it into `mustMakePlayerServer` (be sure to update the helper to support this).
- We then send the web socket messages for a game.
- Finally we assert that the game is started and finished with what we expect.

Try to run the test

You'll have a number of compilation errors around `mustMakePlayerServer` in other tests. Introduce an unexported variable `dummyGame` and use it through all the tests that aren't compiling

```
var (
        dummyGame = &GameSpy{}
```

```
)
```

The final error is where we are trying to pass in `Game` to `NewPlayerServer` but it doesn't support it yet

```
./server_test.go:21:38: too many arguments in call to "github.com/quii/learn-go-with
        have ("github.com/quii/learn-go-with-tests/WebSockets/v2".PlayerStore, "githu
        want ("github.com/quii/learn-go-with-tests/WebSockets/v2".PlayerStore)
```

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

Just add it as an argument for now just to get the test running

```
func NewPlayerServer(store PlayerStore, game Game) (*PlayerServer, error)
```

Finally!

```
=== RUN   TestGame/start_a_game_with_3_players_and_declare_Ruth_the_winner
--- FAIL: TestGame (0.01s)
    --- FAIL: TestGame/start_a_game_with_3_players_and_declare_Ruth_the_winner (0.01s
        server_test.go:146: wanted Start called with 3 but got 0
        server_test.go:147: expected finish called with 'Ruth' but got ''
FAIL
```

Write enough code to make it pass

We need to add `Game` as a field to `PlayerServer` so that it can use it when it gets requests.

```
type PlayerServer struct {
        store PlayerStore
```

```
        http.Handler
        template *template.Template

        game     Game
}
```

(We already have a method called `game` so rename that to `playGame`)

Next lets assign it in our constructor

```
func NewPlayerServer(store PlayerStore, game Game) (*PlayerServer, error) {
        p := new(PlayerServer)

        tmpl, err := template.ParseFiles(htmlTemplatePath)

        if err != nil {
                return nil, fmt.Errorf("problem opening %s %v", htmlTemplatePath, err
        }

        p.game = game

        // etc
}
```

Now we can use our `Game` within `webSocket`.

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
        conn, _ := wsUpgrader.Upgrade(w, r, nil)

        _, numberOfPlayersMsg, _ := conn.ReadMessage()
        numberOfPlayers, _ := strconv.Atoi(string(numberOfPlayersMsg))
        p.game.Start(numberOfPlayers, ioutil.Discard) //todo: Don't discard the blind

        _, winner, _ := conn.ReadMessage()
        p.game.Finish(string(winner))
}
```

Hooray! The tests pass.

We are not going to send the blind messages anywhere *just yet* as we need to have a think about that. When we call `game.Start` we send in `ioutil.Discard` which will just discard any messages

written to it.

For now start the web server up. You'll need to update the `main.go` to pass a `Game` to the `PlayerServer`

```
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, err := poker.NewFileSystemPlayerStore(db)

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

        game := poker.NewTexasHoldem(poker.BlindAlerterFunc(poker.Alerter), store)

        server, err := poker.NewPlayerServer(store, game)

        if err != nil {
                log.Fatalf("problem creating player server %v", err)
        }

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

Discounting the fact we're not getting blind alerts yet, the app does work! We've managed to re-use `Game` with `PlayerServer` and it has taken care of all the details. Once we figure out how to send our blind alerts through to the web sockets rather than discarding them it *should* all work.

Before that though, let's tidy up some code.

Refactor

The way we're using WebSockets is fairly basic and the error handling is fairly naive, so I wanted to encapsulate that in a type just to remove that messiness from the server code. We may wish to

revisit it later but for now this'll tidy things up a bit

```
type playerServerWS struct {
        *websocket.Conn
}

func newPlayerServerWS(w http.ResponseWriter, r *http.Request) *playerServerWS {
        conn, err := wsUpgrader.Upgrade(w, r, nil)

        if err != nil {
                log.Printf("problem upgrading connection to WebSockets %v\n", err)
        }

        return &playerServerWS{conn}
}

func (w *playerServerWS) WaitForMsg() string {
        _, msg, err := w.ReadMessage()
        if err != nil {
                log.Printf("error reading from websocket %v\n", err)
        }
        return string(msg)
}
```

Now the server code is a bit simplified

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
        ws := newPlayerServerWS(w, r)

        numberOfPlayersMsg := ws.WaitForMsg()
        numberOfPlayers, _ := strconv.Atoi(numberOfPlayersMsg)
        p.game.Start(numberOfPlayers, ioutil.Discard) //todo: Don't discard the blinc

        winner := ws.WaitForMsg()
        p.game.Finish(winner)
}
```

Once we figure out how to not discard the blind messages we're done.

Let's *not* write a test!

Sometimes when we're not sure how to do something, it's best just to play around and try things out! Make sure your work is committed first because once we've figured out a way we should drive it through a test.

The problematic line of code we have is

```
p.game.Start(numberOfPlayers, ioutil.Discard) //todo: Don't discard the blinds messa
```

We need to pass in an `io.Writer` for the game to write the blind alerts to.

Wouldn't it be nice if we could pass in our `playerServerWS` from before? It's our wrapper around our WebSocket so it *feels* like we should be able to send that to our `Game` to send messages to.

Give it a go:

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
        ws := newPlayerServerWS(w, r)

        numberOfPlayersMsg := ws.WaitForMsg()
        numberOfPlayers, _ := strconv.Atoi(numberOfPlayersMsg)
        p.game.Start(numberOfPlayers, ws)
        //etc...
}
```

The compiler complains

```
./server.go:71:14: cannot use ws (type *playerServerWS) as type io.Writer in argument
        *playerServerWS does not implement io.Writer (missing Write method)
```

It seems the obvious thing to do, would be to make it so `playerServerWS` *does* implement `io.Writer`. To do so we use the underlying `*websocket.Conn` to use `WriteMessage` to send the message down the websocket

```
func (w *playerServerWS) Write(p []byte) (n int, err error) {
        err = w.WriteMessage(websocket.TextMessage, p)

        if err != nil {
```

```
        }       return 0, err

        return len(p), nil
}
```

This seems too easy! Try and run the application and see if it works.

Beforehand edit `TexasHoldem` so that the blind increment time is shorter so you can see it in action

```
blindIncrement := time.Duration(5+numberOfPlayers) * time.Second // (rather than a mi
```

You should see it working! The blind amount increments in the browser as if by magic.

Now let's revert the code and think how to test it. In order to *implement* it all we did was pass through to `StartGame` was `playerServerWS` rather than `ioutil.Discard` so that might make you think we should perhaps spy on the call to verify it works.

Spying is great and helps us check implementation details but we should always try and favour testing the *real* behaviour if we can because when you decide to refactor it's often spy tests that start failing because they are usually checking implementation details that you're trying to change.

Our test currently opens a websocket connection to our running server and sends messages to make it do things. Equally we should be able to test the messages our server sends back over the websocket connection.

Write the test first

We'll edit our existing test.

Currently, our `GameSpy` does not send any data to `out` when you call `Start`. We should change it so we can configure it to send a canned message and then we can check that message gets sent to the websocket. This should give us confidence that we have configured things correctly whilst still exercising the real behaviour we want.

```
type GameSpy struct {
        StartCalled     bool
```

```
        StartCalledWith int
        BlindAlert      []byte

        FinishedCalled   bool
        FinishCalledWith string
}
```

Add `BlindAlert` field.

Update `GameSpy` `Start` to send the canned message to `out`.

```
func (g *GameSpy) Start(numberOfPlayers int, out io.Writer) {
        g.StartCalled = true
        g.StartCalledWith = numberOfPlayers
        out.Write(g.BlindAlert)
}
```

This now means when we exercise `PlayerServer` when it tries to `Start` the game it should end up sending messages through the websocket if things are working right.

Finally, we can update the test

```
t.Run("start a game with 3 players, send some blind alerts down WS and declare Ruth
        wantedBlindAlert := "Blind is 100"
        winner := "Ruth"

        game := &GameSpy{BlindAlert: []byte(wantedBlindAlert)}
        server := httptest.NewServer(mustMakePlayerServer(t, dummyPlayerStore, game))
        ws := mustDialWS(t, "ws"+strings.TrimPrefix(server.URL, "http")+"/ws")

        defer server.Close()
        defer ws.Close()

        writeWSMessage(t, ws, "3")
        writeWSMessage(t, ws, winner)

        time.Sleep(10 * time.Millisecond)
        assertGameStartedWith(t, game, 3)
        assertFinishCalledWith(t, game, winner)

        _, gotBlindAlert, _ := ws.ReadMessage()

```

```
21          if string(gotBlindAlert) != wantedBlindAlert {
22                  t.Errorf("got blind alert %q, want %q", string(gotBlindAlert), wantec
23          }
24  })
```

- We've added a `wantedBlindAlert` and configured our `GameSpy` to send it to `out` if `Start` is called.
- We hope it gets sent in the websocket connection so we've added a call to `ws.ReadMessage()` to wait for a message to be sent and then check it's the one we expected.

Try to run the test

You should find the test hangs forever. This is because `ws.ReadMessage()` will block until it gets a message, which it never will.

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

We should never have tests that hang so let's introduce a way of handling code that we want to timeout.

```
1  func within(t testing.TB, d time.Duration, assert func()) {
2          t.Helper()
3
4          done := make(chan struct{}, 1)
5
6          go func() {
7                  assert()
8                  done <- struct{}{}
9          }()
10
11         select {
12         case <-time.After(d):
13                 t.Error("timed out")
```

```
        case <-done:
}
```

What `within` does is take a function `assert` as an argument and then runs it in a go routine. If/When the function finishes it will signal it is done via the `done` channel.

While that happens we use a `select` statement which lets us wait for a channel to send a message. From here it is a race between the `assert` function and `time.After` which will send a signal when the duration has occurred.

Finally, I made a helper function for our assertion just to make things a bit neater

```
func assertWebsocketGotMsg(t *testing.T, ws *websocket.Conn, want string) {
        _, msg, _ := ws.ReadMessage()
        if string(msg) != want {
                t.Errorf(`got "%s", want "%s"`, string(msg), want)
        }
}
```

Here's how the test reads now

```
t.Run("start a game with 3 players, send some blind alerts down WS and declare Ruth t
        wantedBlindAlert := "Blind is 100"
        winner := "Ruth"

        game := &GameSpy{BlindAlert: []byte(wantedBlindAlert)}
        server := httptest.NewServer(mustMakePlayerServer(t, dummyPlayerStore, game))
        ws := mustDialWS(t, "ws"+strings.TrimPrefix(server.URL, "http")+"/ws")

        defer server.Close()
        defer ws.Close()

        writeWSMessage(t, ws, "3")
        writeWSMessage(t, ws, winner)

        time.Sleep(tenMS)

        assertGameStartedWith(t, game, 3)
        assertFinishCalledWith(t, game, winner)
        within(t, tenMS, func() { assertWebsocketGotMsg(t, ws, wantedBlindAlert) })
```

```
20 }\)
```

Now if you run the test...

```
=== RUN   TestGame
=== RUN   TestGame/start_a_game_with_3_players,_send_some_blind_alerts_down_WS_and_de
--- FAIL: TestGame (0.02s)
    --- FAIL: TestGame/start_a_game_with_3_players,_send_some_blind_alerts_down_WS_an
        server_test.go:143: timed out
        server_test.go:150: got "", want "Blind is 100"
```

Write enough code to make it pass

Finally, we can now change our server code, so it sends our WebSocket connection to the game when it starts

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
        ws := newPlayerServerWS(w, r)

        numberOfPlayersMsg := ws.WaitForMsg()
        numberOfPlayers, _ := strconv.Atoi(numberOfPlayersMsg)
        p.game.Start(numberOfPlayers, ws)

        winner := ws.WaitForMsg()
        p.game.Finish(winner)
}
```

Refactor

The server code was a very small change so there's not a lot to change here but the test code still has a `time.Sleep` call because we have to wait for our server to do its work asynchronously.

We can refactor our helpers `assertGameStartedWith` and `assertFinishCalledWith` so that they can retry their assertions for a short period before failing.

Here's how you can do it for `assertFinishCalledWith` and you can use the same approach for the other helper.

```
func assertFinishCalledWith(t testing.TB, game *GameSpy, winner string) {
        t.Helper()

        passed := retryUntil(500*time.Millisecond, func() bool {
                return game.FinishCalledWith == winner
        })

        if !passed {
                t.Errorf("expected finish called with %q but got %q", winner, game.Fi
        }
}
```

Here is how `retryUntil` is defined

```
func retryUntil(d time.Duration, f func() bool) bool {
        deadline := time.Now().Add(d)
        for time.Now().Before(deadline) {
                if f() {
                        return true
                }
        }
        return false
}
```

Wrapping up

Our application is now complete. A game of poker can be started via a web browser and the users are informed of the blind bet value as time goes by via WebSockets. When the game finishes they can record the winner which is persisted using code we wrote a few chapters ago. The players can find out who is the best (or luckiest) poker player using the website's `/league` endpoint.

Through the journey we have made mistakes but with the TDD flow we have never been very far away from working software. We were free to keep iterating and experimenting.

The final chapter will retrospect on the approach, the design we've arrived at and tie up some loose ends.

We covered a few things in this chapter

WebSockets

- Convenient way of sending messages between clients and servers that does not require the client to keep polling the server. Both the client and server code we have is very simple.
- Trivial to test, but you have to be wary of the asynchronous nature of the tests

Handling code in tests that can be delayed or never finish

- Create helper functions to retry assertions and add timeouts.
- We can use go routines to ensure the assertions don't block anything and then use channels to let them signal that they have finished, or not.
- The `time` package has some helpful functions which also send signals via channels about events in time so we can set timeouts