Select

You can find all the code for this chapter here

You have been asked to make a function called `WebsiteRacer` which takes two URLs and "races" them by hitting them with an HTTP GET and returning the URL which returned first. If none of them return within 10 seconds then it should return an `error`.

For this, we will be using

- `net/http` to make the HTTP calls.
- `net/http/httptest` to help us test them.
- goroutines.
- `select` to synchronise processes.

Write the test first

Let's start with something naive to get us going.

```go
func TestRacer(t *testing.T) {
	slowURL := "http://www.facebook.com"
	fastURL := "http://www.quii.dev"

	want := fastURL
	got := Racer(slowURL, fastURL)

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

We know this isn't perfect and has problems but it will get us going. It's important not to get too hung-up on getting things perfect first time.

Try to run the test

`./racer_test.go:14:9: undefined: Racer`

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

```go
func Racer(a, b string) (winner string) {
	return
}
```

`racer_test.go:25: got '', want 'http://www.quii.dev'`

Write enough code to make it pass

```go
func Racer(a, b string) (winner string) {
	startA := time.Now()
	http.Get(a)
	aDuration := time.Since(startA)

	startB := time.Now()
	http.Get(b)
	bDuration := time.Since(startB)

	if aDuration < bDuration {
		return a
	}

	return b
}
```

For each URL:

1. We use `time.Now()` to record just before we try and get the `URL`.
2. Then we use `http.Get` to try and get the contents of the `URL`. This function returns an `http.Response` and an `error` but so far we are not interested in these values.
3. `time.Since` takes the start time and returns a `time.Duration` of the difference.

Once we have done this we simply compare the durations to see which is the quickest.

Problems

This may or may not make the test pass for you. The problem is we're reaching out to real websites to test our own logic.

Testing code that uses HTTP is so common that Go has tools in the standard library to help you test it.

In the mocking and dependency injection chapters, we covered how ideally we don't want to be relying on external services to test our code because they can be

- Slow
- Flaky
- Can't test edge cases

In the standard library, there is a package called `net/http/httptest` where you can easily create a mock HTTP server.

Let's change our tests to use mocks so we have reliable servers to test against that we can control.

```go
func TestRacer(t *testing.T) {

	slowServer := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter,
		time.Sleep(20 * time.Millisecond)
		w.WriteHeader(http.StatusOK)
	}))

	fastServer := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter,
		w.WriteHeader(http.StatusOK)
	}))

	slowURL := slowServer.URL
	fastURL := fastServer.URL

	want := fastURL
	got := Racer(slowURL, fastURL)

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

	slowServer.Close()
	fastServer.Close()
}
```

The syntax may look a bit busy but just take your time.

`httptest.NewServer` takes an `http.HandlerFunc` which we are sending in via an *anonymous function*.

`http.HandlerFunc` is a type that looks like this: `type HandlerFunc func(ResponseWriter, *Request)`.

All it's really saying is it needs a function that takes a `ResponseWriter` and a `Request`, which is not too surprising for an HTTP server.

It turns out there's really no extra magic here, **this is also how you would write a *real* HTTP server in Go**. The only difference is we are wrapping it in an `httptest.NewServer` which makes it easier to use with testing, as it finds an open port to listen on and then you can close it when you're done with your test.

Inside our two servers, we make the slow one have a short `time.Sleep` when we get a request to make it slower than the other one. Both servers then write an `OK` response with `w.WriteHeader(http.StatusOK)` back to the caller.

If you re-run the test it will definitely pass now and should be faster. Play with these sleeps to deliberately break the test.

Refactor

We have some duplication in both our production code and test code.

```go
func Racer(a, b string) (winner string) {
	aDuration := measureResponseTime(a)
	bDuration := measureResponseTime(b)

	if aDuration < bDuration {
		return a
	}

	return b
}

func measureResponseTime(url string) time.Duration {
	start := time.Now()
	http.Get(url)
	return time.Since(start)
}
```

This DRY-ing up makes our `Racer` code a lot easier to read.

```go
func TestRacer(t *testing.T) {

	slowServer := makeDelayedServer(20 * time.Millisecond)
	fastServer := makeDelayedServer(0 * time.Millisecond)

	defer slowServer.Close()
	defer fastServer.Close()

	slowURL := slowServer.URL
	fastURL := fastServer.URL

	want := fastURL
	got := Racer(slowURL, fastURL)

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

func makeDelayedServer(delay time.Duration) *httptest.Server {
	return httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *htt
		time.Sleep(delay)
		w.WriteHeader(http.StatusOK)
	}))
}
```

We've refactored creating our fake servers into a function called `makeDelayedServer` to move some uninteresting code out of the test and reduce repetition.

`defer`

By prefixing a function call with `defer` it will now call that function *at the end of the containing function*.

Sometimes you will need to cleanup resources, such as closing a file or in our case closing a server so that it does not continue to listen to a port.

You want this to execute at the end of the function, but keep the instruction near where you created the server for the benefit of future readers of the code.

Our refactoring is an improvement and is a reasonable solution given the Go features covered so far, but we can make the solution simpler.

Synchronising processes

- Why are we testing the speeds of the websites one after another when Go is great at concurrency? We should be able to check both at the same time.
- We don't really care about *the exact response times* of the requests, we just want to know which one comes back first.

To do this, we're going to introduce a new construct called `select` which helps us synchronise processes really easily and clearly.

```go
func Racer(a, b string) (winner string) {
	select {
	case <-ping(a):
		return a
	case <-ping(b):
		return b
	}
}

func ping(url string) chan struct{} {
	ch := make(chan struct{})
	go func() {
			http.Get(url)
			close(ch)

		}()
		return ch
}
```

`ping`

We have defined a function `ping` which creates a `chan struct{}` and returns it.

In our case, we don't *care* what type is sent to the channel, *we just want to signal we are done* and closing the channel works perfectly!

Why `struct{}` and not another type like a `bool`? Well, a `chan struct{}` is the smallest data type available from a memory perspective so we get no allocation versus a `bool`. Since we are closing and not sending anything on the chan, why allocate anything?

Inside the same function, we start a goroutine which will send a signal into that channel once we have completed `http.Get(url)`.

Always `make` channels

Notice how we have to use `make` when creating a channel; rather than say `var ch chan struct{}`. When you use `var` the variable will be initialised with the "zero" value of the type. So for `string` it is `""`, `int` it is 0, etc.

For channels the zero value is `nil` and if you try and send to it with `<-` it will block forever because you cannot send to `nil` channels

You can see this in action in The Go Playground

`select`

If you recall from the concurrency chapter, you can wait for values to be sent to a channel with `myVar := <-ch`. This is a *blocking* call, as you're waiting for a value.

What `select` lets you do is wait on *multiple* channels. The first one to send a value "wins" and the code underneath the `case` is executed.

We use `ping` in our `select` to set up two channels for each of our `URL`s. Whichever one writes to its channel first will have its code executed in the `select`, which results in its `URL` being returned (and being the winner).

After these changes, the intent behind our code is very clear and the implementation is actually simpler.

Timeouts

Our final requirement was to return an error if `Racer` takes longer than 10 seconds.

Write the test first

```go
t.Run("returns an error if a server doesn't respond within 10s", func(t *testing.T)
	serverA := makeDelayedServer(11 * time.Second)
	serverB := makeDelayedServer(12 * time.Second)

	defer serverA.Close()
	defer serverB.Close()

	_, err := Racer(serverA.URL, serverB.URL)

	if err == nil {
		t.Error("expected an error but didn't get one")
	}
})
```

We've made our test servers take longer than 10s to return to exercise this scenario and we are expecting `Racer` to return two values now, the winning URL (which we ignore in this test with `_`) and an `error`.

Try to run the test

```
./racer_test.go:37:10: assignment mismatch: 2 variables but 1 values
```

Write the minimal amount of code for the test to run and

check the failing test output

```go
func Racer(a, b string) (winner string, error error) {
	select {
	case <-ping(a):
		return a, nil
	case <-ping(b):
		return b, nil
	}
}
```

Change the signature of `Racer` to return the winner and an `error`. Return `nil` for our happy cases.

The compiler will complain about your *first test* only looking for one value so change that line to `got, _ := Racer(slowURL, fastURL)`, knowing that we should check we *don't* get an error in our happy scenario.

If you run it now after 11 seconds it will fail.

```
--- FAIL: TestRacer (12.00s)
    --- FAIL: TestRacer/returns_an_error_if_a_server_doesn't_respond_within_10s (12.0
        racer_test.go:40: expected an error but didn't get one
```

Write enough code to make it pass

```go
func Racer(a, b string) (winner string, error error) {
	select {
	case <-ping(a):
		return a, nil
	case <-ping(b):
		return b, nil
	case <-time.After(10 * time.Second):
		return "", fmt.Errorf("timed out waiting for %s and %s", a, b)
	}
}
```

`time.After` is a very handy function when using `select`. Although it didn't happen in our case you can potentially write code that blocks forever if the channels you're listening on never return a value. `time.After` returns a `chan` (like `ping`) and will send a signal down it after the amount of time you define.

For us this is perfect; if `a` or `b` manage to return they win, but if we get to 10 seconds then our `time.After` will send a signal and we'll return an `error`.

Slow tests

The problem we have is that this test takes 10 seconds to run. For such a simple bit of logic, this doesn't feel great.

What we can do is make the timeout configurable. So in our test, we can have a very short timeout and then when the code is used in the real world it can be set to 10 seconds.

```go
func Racer(a, b string, timeout time.Duration) (winner string, error error) {
	select {
	case <-ping(a):
		return a, nil
	case <-ping(b):
		return b, nil
	case <-time.After(timeout):
		return "", fmt.Errorf("timed out waiting for %s and %s", a, b)
	}
}
```

Our tests now won't compile because we're not supplying a timeout.

Before rushing in to add this default value to both our tests let's *listen to them*.

- Do we care about the timeout in the "happy" test?
- The requirements were explicit about the timeout.

Given this knowledge, let's do a little refactoring to be sympathetic to both our tests and the users of our code.

```go
var tenSecondTimeout = 10 * time.Second

func Racer(a, b string) (winner string, error error) {
	return ConfigurableRacer(a, b, tenSecondTimeout)
}

func ConfigurableRacer(a, b string, timeout time.Duration) (winner string, error errc
	select {
	case <-ping(a):
		return a, nil
	case <-ping(b):
		return b, nil
	case <-time.After(timeout):
		return "", fmt.Errorf("timed out waiting for %s and %s", a, b)
	}
}
```

Our users and our first test can use `Racer` (which uses `ConfigurableRacer` under the hood) and our sad path test can use `ConfigurableRacer`.

```go
func TestRacer(t *testing.T) {

	t.Run("compares speeds of servers, returning the url of the fastest one", fur
		slowServer := makeDelayedServer(20 * time.Millisecond)
		fastServer := makeDelayedServer(0 * time.Millisecond)

		defer slowServer.Close()
		defer fastServer.Close()

		slowURL := slowServer.URL
		fastURL := fastServer.URL

		want := fastURL
		got, err := Racer(slowURL, fastURL)

		if err != nil {
			t.Fatalf("did not expect an error but got one %v", err)
		}

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

	t.Run("returns an error if a server doesn't respond within the specified time
        server := makeDelayedServer(25 * time.Millisecond)

        defer server.Close()

        _, err := ConfigurableRacer(server.URL, server.URL, 20*time.Millisecc

        if err == nil {
            t.Error("expected an error but didn't get one")
        }
    })
}
```

I added one final check on the first test to verify we don't get an `error`.

Wrapping up

`select`

- Helps you wait on multiple channels.
- Sometimes you'll want to include `time.After` in one of your `cases` to prevent your system blocking forever.

`httptest`

- A convenient way of creating test servers so you can have reliable and controllable tests.
- Using the same interfaces as the "real" `net/http` servers which is consistent and less for you to learn.