Testing Go Applications

Lecture 10

Learning Outcomes

After completing this lecture you should be able to

- explore the testing feature that Go provides
- create and run table-driven unit tests and subtests in Go
- unit test your HTTP handler
- perform ‘end-to-end’ testing of your web application routes and handlers
- create mocks of your database models and use them in unit tests

Learning Outcomes

After completing this lecture you should be able to

- apply a pattern for testing CSRF-protected HTML form submissions
- use a test instance of Postgres to perform integration tests
- calculate and profile code coverage for your tests

Go's Testing Support

Go provies testing-focused libraries in its standard library

Below are two of the testing packages that are commonly used

```
testing, net/http/httptest
```

The httptest package is a library for testing web applications; it is based on the testing package

Go's Testing Support

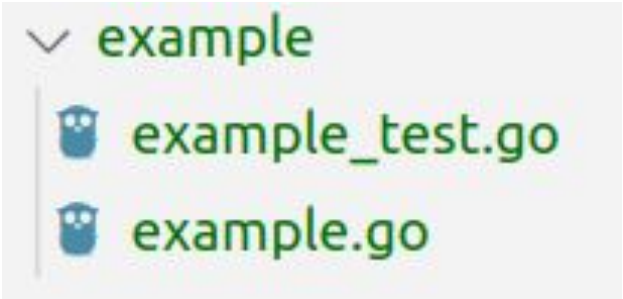
The `testing` package is used with the `go test` command which is used on any Go source files that end with `_test.go`. Usually this corresponds with the name of the source code file you're testing

In go, your test file should have the form `*_test.go` and **must be placed in the same package as the code that you're testing**

Go's Testing Support

Example:

If you have a go file `example.go` which contain the code that you want to test , your test file would be `example_test.go` file that contains all the tests you want to run on the `example.go` file



Go's Testing Support

In the test file you create test functions with the following form:

```
func TestXxx(*testing.T) { ... }

where Xxx is any alphanumeric string in which the first
letter is capitalized
```

When you run the `go test` command in the console, this and other similar functions will be executed

Within these functions you can use `Error` , `Fail` , and other methods to indicate test failure. If there's no failure, the test for the function is considered to have passed

Unit Testing and Sub-Tests

Example: `customDate` function written inside `example.go` file

```go
package example

import (
    "time"
)

func customDate(t time.Time) string {
    return t.UTC().Format("02 Jan 2006 at 15:04")
}
```

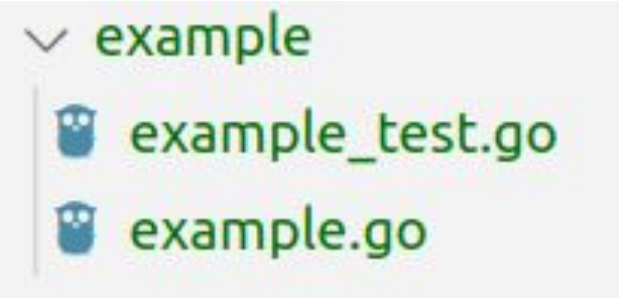
Unit Testing and Sub-Tests

Unit test provides confidence that a unit (a modular part of a program) is correct

> a part of a program is a unit if it can be tested independently

Unit Testing and Sub-Tests

The test file for the example code shown in the previous slide should be written in a file named **`example_test.go`**



Unit Testing and Sub-Tests

`TestCustomDate` function inside `example_test.go`

```
package example

import (
    "testing"
    "time"
)

func TestCustomDate(t *testing.T) {

    actualTime := time.Date(2019, 12, 25, 10, 0, 0, 0, time.UTC)
    wantTime := "25 Dec 2019 at 10:00"
    gotTime := customDate(actualTime)

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

Exercise

Write a simple go function named `sum` which accepts two integers and return the sum of the two integers

Then write a test function that verify if the function returns the expected result

Unit Testing and Sub-Tests

Go testing code patterns

Go tests are just **regular Go code**, which **checks if actual result matches the expected result**

Your Unit Tests are contained in a normal Go function with the signature `func(*testing.T)`

To be a valid unit test **the name of this function must begin with** the word `Test`. Typically this is then followed by the name of the function, method or type that you're testing

Unit Testing and Sub-Tests

Go testing code patterns

You can use the `t.Errorf()` version of the error function to mark a test as failed and log a descriptive message about the failure

Unit Testing and Sub-Tests

Running a test

Type the following command to run the test

```
go test ./example
```

You should see the following output

```
ok      github.com/betsegawlemma/restaurant/example
```

Unit Testing and Sub-Tests

Running a test

You can add the `-v` flag to see the details of the test

```
go test -v ./example
```

You should see the following output

```
=== RUN   TestCustomDate
--- PASS: TestCustomDate (0.00s)
PASS
ok    github.com/betsegawlemma/restaurant/example  0.002s
```

Table-Driven Tests

In Go, an idiomatic way to run multiple test cases is to use **table-driven** tests

Steps for table-driven tests

- create a **table of test cases** containing the **inputs** and **expected** outputs
- loop over the test cases, running each test case in a sub-test

Table-Driven Tests

Modify the `TestCustomDate` function

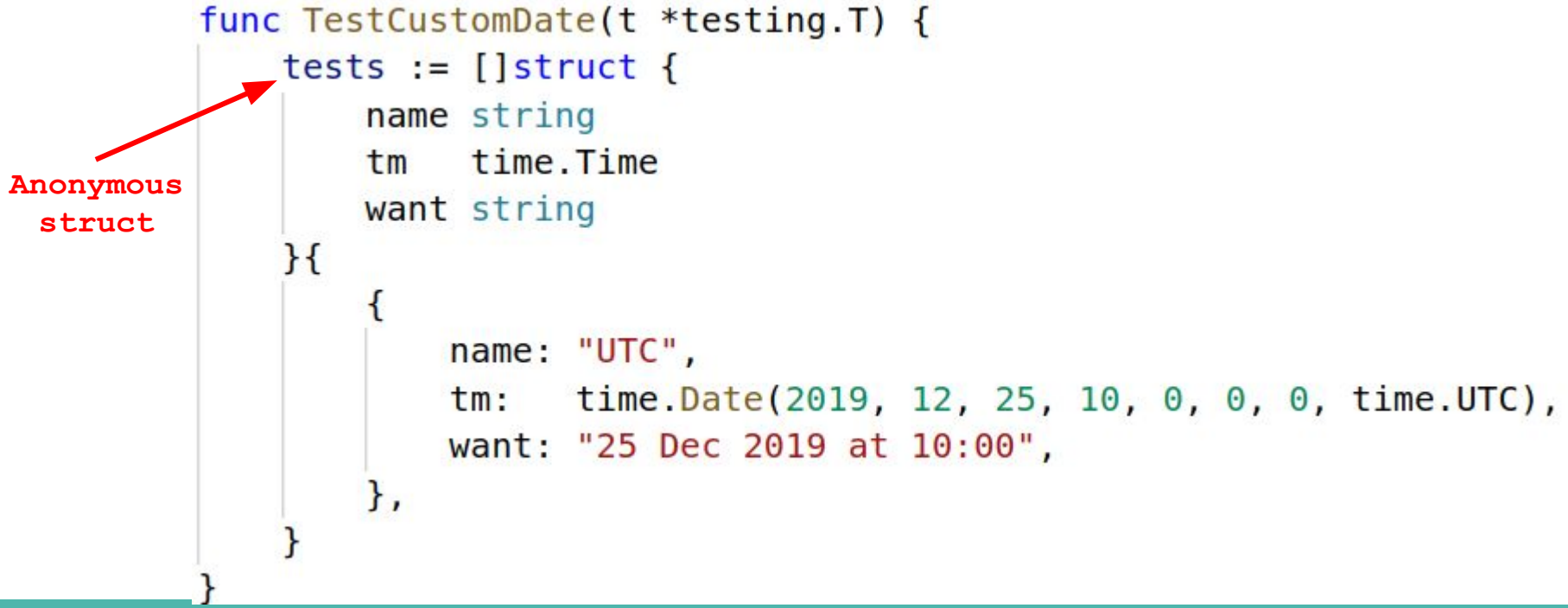


Table-Driven Tests

Modify the `TestCustomDate` function


```
func TestCustomDate(t *testing.T) {
    tests := []struct {
        name string
        tm   time.Time
        want string
    }{
        {
            name: "UTC",
            tm:   time.Date(2019, 12, 25, 10, 0, 0, 0, time.UTC),
            want: "25 Dec 2019 at 10:00",
        },
    }
}
```

Exercise

Rewrite the anonymous struct in the form of named struct and create an instance of the struct with the information shown

Add another test case for EAT

Where: **EAT = UTC + 3**

Table-Driven Tests

EAT and CET test cases

Note: CET = UTC + 1

```
{
    name: "EAT",
    tm:   time.Date(2019, 12, 25, 10, 0, 0, 0, time.FixedZone("EAT", 3*60*60)),
    want: "25 Dec 2019 at 07:00",
},
{
    name: "CET",
    tm:   time.Date(2019, 12, 25, 10, 0, 0, 0, time.FixedZone("EAT", 1*60*60)),
    want: "25 Dec 2019 at 09:00",
},
```

Table-Driven Tests

Add a loop inside `TestCustomDate` function to loop over the test cases

```
func TestCustomDate(t *testing.T) {
    tests := []struct {…
    }{…
    }
}
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            got := customDate(tt.tm)
            if got != tt.want {
                t.Errorf("want %q; got %q", tt.want, got)
            }
        })
    }
}
```

Table-Driven Tests

Add a loop inside `TestCustomDate` function to loop over the test cases

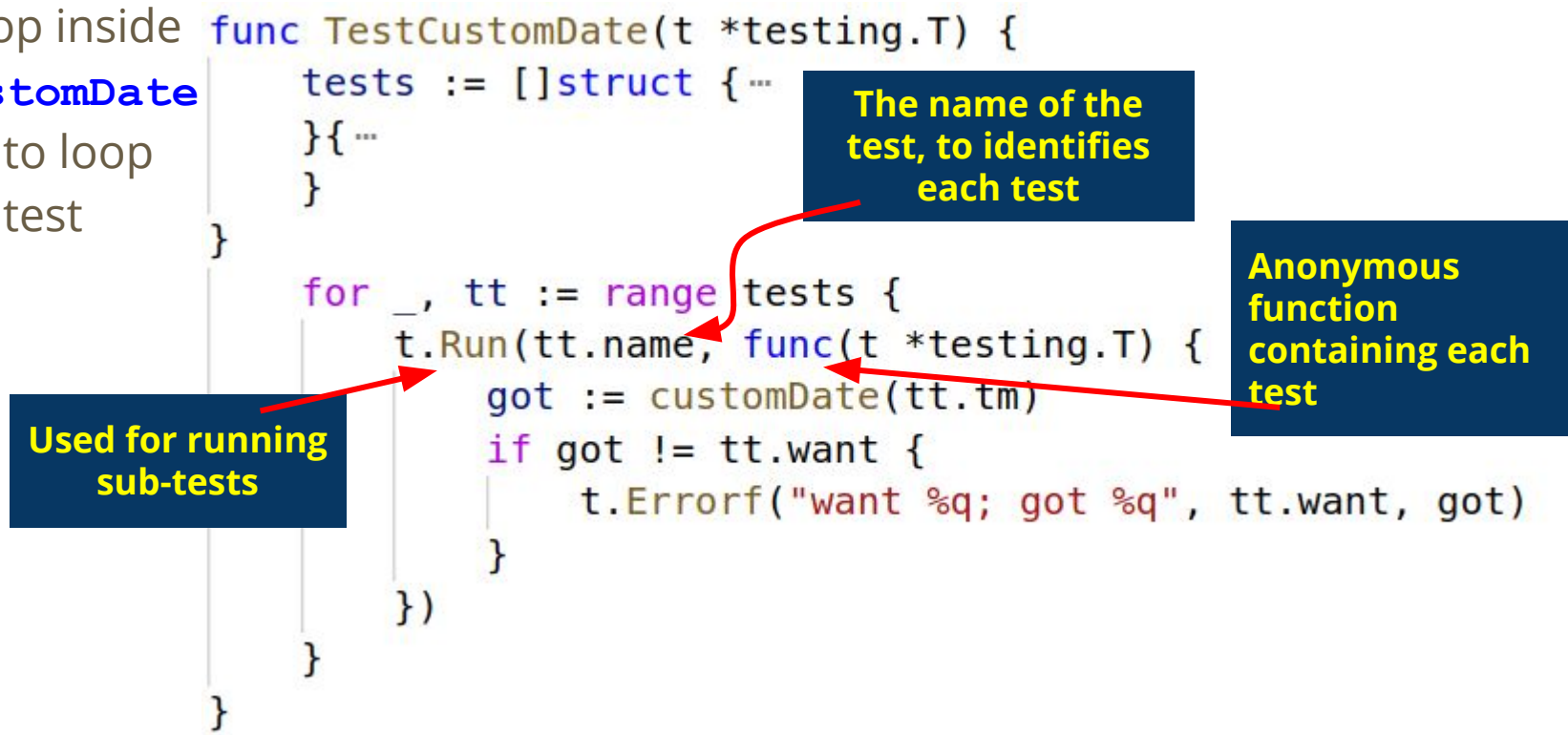


Table-Driven Tests

Output of running `TestCustomDate` with three test cases

```
=== RUN   TestCustomDate
=== RUN   TestCustomDate/UTC
=== RUN   TestCustomDate/EAT
=== RUN   TestCustomDate/CET
--- PASS: TestCustomDate (0.00s)
    --- PASS: TestCustomDate/UTC (0.00s)
    --- PASS: TestCustomDate/EAT (0.00s)
    --- PASS: TestCustomDate/CET (0.00s)
PASS
ok    github.com/betsegawlemma/restaurant/example     0.002s
```

Exercise

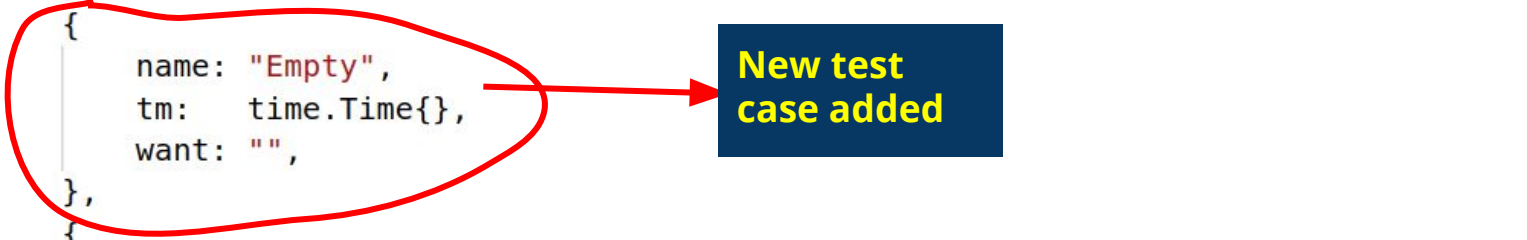
For the **sum** function you wrote write three **table-driven** tests for different input values

Modify your test function to use the table-driven test

Failing Tests

Run your test code with the following test cases

```
{
    name: "UTC",
    tm:   time.Date(2019, 12, 25, 10, 0, 0, 0, time.UTC),
    want: "25 Dec 2019 at 10:00",
},
{
    name: "Empty",
    tm:   time.Time{},
    want: "",
},
{
    name: "EAT",
    tm:   time.Date(2019, 12, 25, 10, 0, 0, 0, time.FixedZone("EAT", 3*60*60)),
    want: "25 Dec 2019 at 07:00",
},
```



Failing Test

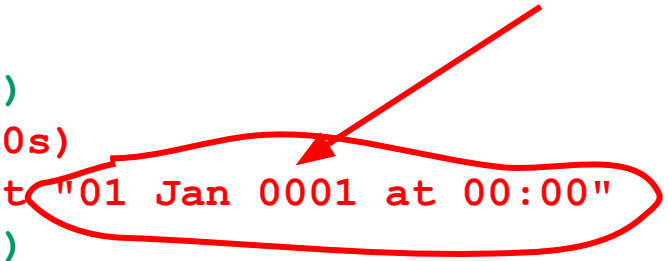
```
=== RUN   TestCustomDate
=== RUN   TestCustomDate/UTC
=== RUN   TestCustomDate/Empty
=== RUN   TestCustomDate/EAT
--- FAIL: TestCustomDate (0.00s)
    --- PASS: TestCustomDate/UTC (0.00s)
    --- FAIL: TestCustomDate/Empty (0.00s)
        example_test.go:35: want ""; got "01 Jan 0001 at 00:00"
    --- PASS: TestCustomDate/EAT (0.00s)
FAIL
FAIL    github.com/betsegawlemma/restaurant/example     0.002s
FAIL
```

What is the reason for the test failure?

Failing Test

```
=== RUN   TestCustomDate
=== RUN   TestCustomDate/UTC
=== RUN   TestCustomDate/Empty
=== RUN   TestCustomDate/EAT
--- FAIL: TestCustomDate (0.00s)
    --- PASS: TestCustomDate/UTC (0.00s)
    --- FAIL: TestCustomDate/Empty (0.00s)
        example_test.go:35: want ""; got "01 Jan 0001 at 00:00"
    --- PASS: TestCustomDate/EAT (0.00s)
FAIL
FAIL    github.com/betsegawlemma/restaurant/example     0.002s
FAIL
```

customDate() function *does not handle the zero time*



Failing Tests

Notes about the failed test

> For failed test cases Go outputs the **failure message** with **filename** and **line number**

```
--- FAIL: TestCustomDate (0.00s)

--- PASS: TestCustomDate/UTC (0.00s)
--- FAIL: TestCustomDate/Empty (0.00s)
    example_test.go:35: want ""; got "01 Jan 0001 at 00:00"
--- PASS: TestCustomDate/EAT (0.00s)
```

Here note also that the test continues after the failed test

Failing Tests

Notes about the failed test

You can use the `-failfast` flag to stop the tests running after the first failure

```
=== RUN   TestCustomDate
=== RUN   TestCustomDate/UTC
=== RUN   TestCustomDate/Empty
--- FAIL: TestCustomDate (0.00s)
    --- PASS: TestCustomDate/UTC (0.00s)
    --- FAIL: TestCustomDate/Empty (0.00s)
        example_test.go:35: want ""; got "01 Jan 0001 at 00:00"
FAIL
FAIL    github.com/betsegawlemma/restaurant/example     0.002s
FAIL
```

Fix the zero-time issue

Modify the `customDate` function to handle the zero time instant: `January 1, year 1, 00:00:00 UTC`

```
func customDate(t time.Time) string {
    if t.IsZero() {
        return ""
    }
    return t.UTC().Format("02 Jan 2006 at 15:04")
}
```

Running All Tests

To run all tests use the following command

```
go test ./...
```

Testing HTTP Handlers

Example: Test the `/about` request handler shown below

```
// About handles requests on route /about
func About(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("About"))
}

http.HandleFunc("/about", About)
```

Testing HTTP Handlers

Example test cases

Checks that the **response status code** written by the about handler is 200

Checks that the **response body** written by the about handler contains the text “about"

Testing HTTP Handlers

Recording Responses

To test the HTTP handlers you can use the `net/http/httptest` package

`httptest` package contains `httptest.ResponseRecorder` type, which is an implementation of `http.ResponseWriter` and **records** the **response status code**, **headers** and **body** instead of actually writing them to a HTTP connection

Testing HTTP Handlers

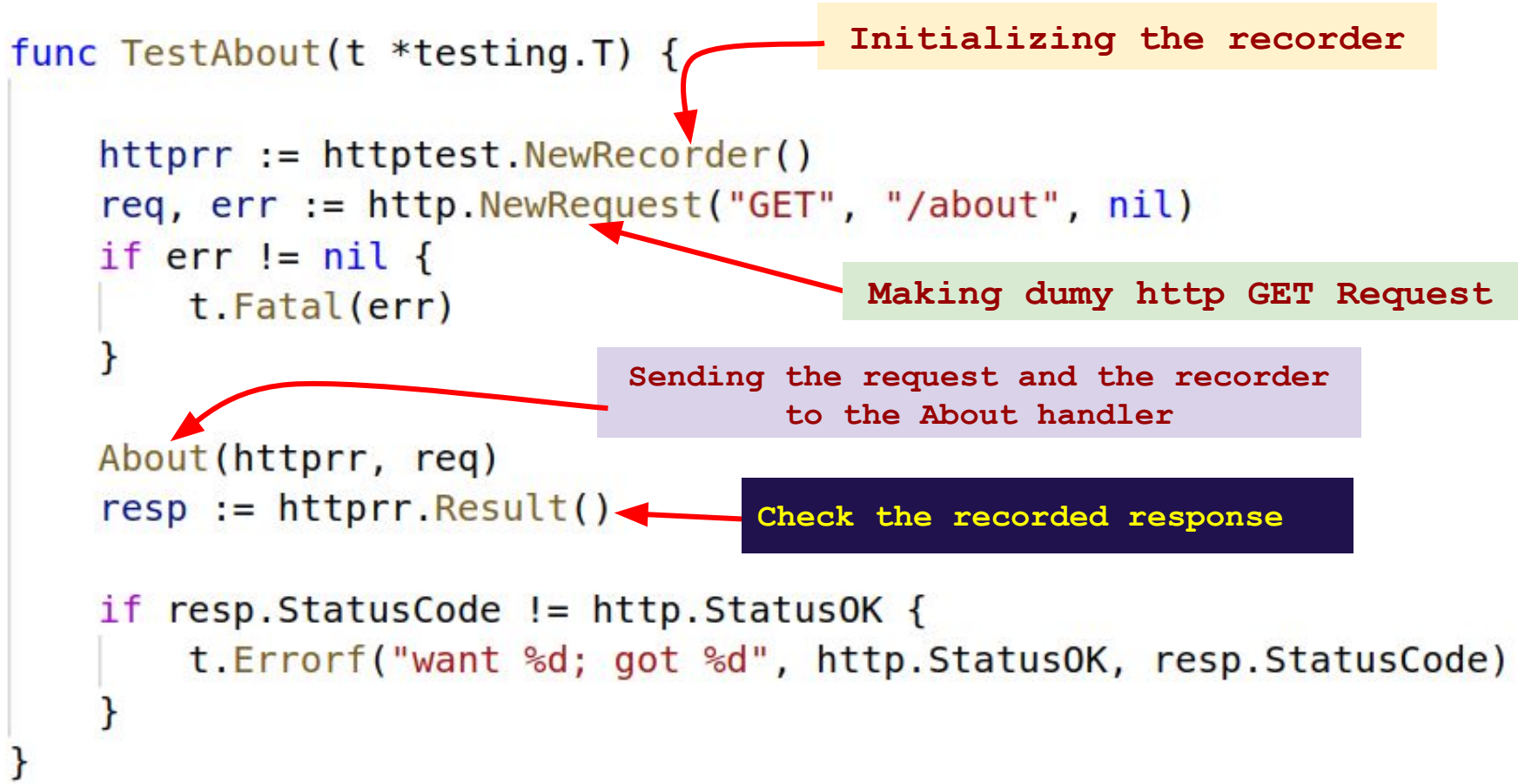
How to unit test your handlers

> create a new `httptest.ResponseRecorder` object, pass it to the handler function, and then examine it again after the handler returns

The next slide shows the `AboutTest` function which tests the `GET /about` request handler function shown below

```
func About(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("About"))
}
```

```
11  func TestAbout(t *testing.T) {
12
13      httprr := httptest.NewRecorder()
14      req, err := http.NewRequest("GET", "/about", nil)
15      if err != nil {
16          t.Fatal(err)
17      }
18
19      About(httprr, req)
20      resp := httprr.Result()
21
22      if resp.StatusCode != http.StatusOK {
23          t.Errorf("want %d; got %d", http.StatusOK, resp.StatusCode)
24      }
25  }
```




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

    httprr := httptest.NewRecorder()
    req, err := http.NewRequest("GET", "/about", nil)
    if err != nil {
        t.Fatal(err)
    }

    About(httprr, req)
    resp := httprr.Result()

    if resp.StatusCode != http.StatusOK {
        t.Errorf("want %d; got %d", http.StatusOK, resp.StatusCode)
    }
}
```

Testing HTTP Handlers

Running the test with the following command displays the output shown

```
go test -v ./delivery/http/handler/
```

```
=== RUN   TestAbout
--- PASS: TestAbout (0.00s)
PASS
ok
github.com/betsegawlemma/restaurant/delivery/http/handler
0.004s
```

Testing HTTP Handlers

Add the following code inside `TestAbout` function to check for the body content

```
defer resp.Body.Close()
body, err := ioutil.ReadAll(resp.Body)
if err != nil {
    t.Fatal(err)
}

if string(body) != "About" {
    t.Errorf("want the body to contain the word %q", "about")
}
```

Testing HTTP Handlers

Running the test fails as the handler contains typo (**“abut” instead of “about”**)

```
=== RUN   TestAbout
--- FAIL: TestAbout (0.00s)
      handler_test.go:33: want the body to contain the word "about"
FAIL
FAIL    github.com/betsegawlemma/restaurant/delivery/http/handler  0.004s
FAIL
```

```
func About(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Abut"))
}
```

Testing HTTP Handlers

In the previous tests `t.Fatal()` function is used to handle situations where there is an unexpected error in the test code

When called, `t.Fatal()` will **mark the test as failed**, **log the error**, and then **stops execution of any further tests**

Running Specific Tests

If you have multiple test functions, you can only run specific tests by using the `-run` flag

This allows you to pass in a regular expression — and only tests with a name that matches the regular expression will be run

Example

```
go test -v -run="^AboutTest$" ./delivery/http/handler/
```

Running Specific Tests

You can also use the **`-run`** flag to limit testing to some specific subtests

Example

```
go test -v -run="^TestCustomDate$/^UTC|EAT$" ./example/
```

```
=== RUN   TestCustomDate
=== RUN   TestCustomDate/UTC
=== RUN   TestCustomDate/EAT
--- PASS: TestCustomDate (0.00s)
    --- PASS: TestCustomDate/UTC (0.00s)
    --- PASS: TestCustomDate/EAT (0.00s)
PASS
ok      github.com/betsegawlemma/restaurant/example     0.003s
```

Parallel Testing

By default, the go test command executes all tests in a serial manner

You can indicate that it's OK for a test to be run in concurrently alongside other tests by calling the t.Parallel() function at the start of the test

```
func TestCustomDate(t *testing.T) {
    t.Parallel()
        ...
}
```

Parallel Testing

By default, the maximum number of tests that will be run simultaneously is the current value of **`GOMAXPROCS`**. You can override this by setting a specific value via the **`-parallel`** flag when you run your tests

```
go test -parallel 4 ./...
```

End-To-End Testing

To perform `end-to-end` testing you can use `httptest.NewTLSServer()` or `httptest.NewServer()` function, which spins up a `httptest.Server` instance

You can use the `httptest.Server` instance to make `http` or `https` requests

An example function `TestContact` is shown in the next slide

The table shows the test plan

Method	Path	Handler	Expected result
GET	/contact	Contact	200 Status Code
GET	/contact	Contact	"About" text on response body

```
37  func TestContact(t *testing.T) {
38      mux := http.NewServeMux()
39      mux.HandleFunc("/contact", Contact)
40      testServ := httptest.NewTLSServer(mux)
41      defer testServ.Close()
42
43      testClient := testServ.Client()
44      url := testServ.URL
45
46      resp, err := testClient.Get(url + "/contact")
47      if err != nil {
48          t.Fatal(err)
49      }
50
51      if resp.StatusCode != http.StatusOK {
52          t.Errorf("want %q got %q", http.StatusOK, resp.StatusCode)
53      }
54  }
```

```go
func TestContact(t *testing.T) {
    mux := http.NewServeMux()
    mux.HandleFunc("/contact", Contact)
    testServ := httptest.NewTLSServer(mux)
    defer testServ.Close()

    testClient := testServ.Client()
    url := testServ.URL

    resp, err := testClient.Get(url + "/contact")
    if err != nil {
        t.Fatal(err)
    }

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

Creates an instance of **httptest.Server**

Returns http client configured for making requests to the server

Returns base URL of the server of form http://ipaddr:port

Client making GET /contact request to the server

End-To-End Testing

Add the following code at the end of **`TestContact`** function to check if the body contains the expected text

Note that the handler code writes the text "**`Contact`**" to the **`http.ResponseWriter`**

```
body, err := ioutil.ReadAll(resp.Body)
if err != nil {
    t.Fatal(err)
}
if string(body) != "Contact" {
    t.Errorf("want the body to contain the word %q", "Contact")
}
```

Mocking Dependencies

Mocks, **fakes**, **test doubles** or **stubs** commonly refer to same concept

They expose the same interface as the object they mock for the purpose of testing

They are simulations of objects, structures, or functions that are used during testing when it's inconvenient to use the actual object, structure, or function

Dependency Injection for Testing

Dependency injection is a software design pattern that allows you to decouple the dependencies between two or more layers of software

Dependency injection is done through passing a dependency to the called object, structure, or function

Your software layers should be decoupled enough for mocking to work

In go you can employ interfaces to achieve this

Dependency Injection Example

Suppose you want to test the `CategoryService` type shown below

```
// CategoryService implements menu.CategoryService interface
type CategoryService struct {
    categoryRepo menu.CategoryRepository
}
```

Note: `CategoryService` depends on **`CategoryRepository`**

Dependency Injection Example

The following code shows how to inject dependency to the `CategoryService`

```
categoryRepo := repository.NewCategoryGormRepo(dbconn)
categoryServ := service.NewCategoryService(categoryRepo)
```

Dependency Injection Example

The following code shows how to inject dependency to the `CategoryService`

```
categoryRepo := repository.NewCategoryGormRepo(dbconn)

categoryServ := service.NewCategoryService(categoryRepo)
```

Here, actual implementation of **`menu.CategoryRepository`** interface is being injected

This is not desirable from testing perspective, what are the disadvantages of doing this?

Dependency Injection Example

The following code shows how to inject dependency to the `CategoryService`

```
categoryRepo := repository.NewCategoryGormRepo(dbconn)
categoryServ := service.NewCategoryService(categoryRepo)
```

Instead of passing actual implementation of **`menu.CategoryRepository`** interface we can pass fake implementation of **`menu.CategoryRepository`**

What are the advantages of these approach?

Dependency Injection Example

The `CategoryRepository` interface

```
// CategoryRepository specifies food menu category database operations
type CategoryRepository interface {
    Categories() ([]entity.Category, []error)
    Category(id uint) (*entity.Category, []error)
    UpdateCategory(category *entity.Category) (*entity.Category, []error)
    DeleteCategory(id uint) (*entity.Category, []error)
    StoreCategory(category *entity.Category) (*entity.Category, []error)
    ItemsInCategory(category *entity.Category) ([]entity.Item, []error)
}
```

Dependency Injection Example

Actual implementation of one of the method

```
Category(id uint) (*entity.Category, []error)
```

```go
// Category retrieve a category from the database by its id
func (cRepo *CategoryGormRepo) Category(id uint) (*entity.Category, []error) {
    ctg := entity.Category{}
    errs := cRepo.conn.First(&ctg, id).GetErrors()
    if len(errs) > 0 {
        return nil, errs
    }
    return &ctg, errs
}
```

How do you fake/mock this?

Dependency Injection Example

Fake/Mock implementation

```
// Category retrieve a category with id 1
func (fCatRepo *FakeCategoryRepo) Category(id uint) (*entity.Category, []error) {
    ctg := entity.Category{
        ID:          1,
        Name:        "Fake Cat 01",
        Description: "Fake Category 01",
        Image:       "fake_cat01.png",
    }

    if id == 1 {
        return &ctg, nil
    }
    return nil, nil
}
```

Integration Testing

Test the integration of two or more components

Example

> Verify that the production Postgres database models are working as expected