Golang Tooling

Unit testing

Check out the repo

A bunch of demos and examples we’ll go through:

https://github.com/ClearPointNZ/go-crufting

What you’ll need:

- git
- VSCode (or some other IDE)
- Go installed
- GNU “make”

Make targets

Have a look at “make test”:

```
go test ./… --cover
```

- It automatically runs all of the unit tests for every package it finds under every subdirectory (./…)
- It gives you a brief summary of test coverage for each package
- Any failures result in a non-zero return code (useful for CI/CD)
- Makefiles seem to be a popular way of semi-automating Go’s tooling

Basic (“testing” package)

Go’s basic test tooling does work, but is a bit clunky, and requires the user to be responsible for checking results and managing success/failure.

Have a look at the /internal/basic package:

- *basic.go* contains a simple public function which scrambles a string
- *basic_test.go* contains a test which checks that a given string doesn’t remain the same
- Note the 100% coverage (woohoo)

Assertions (“testify/assert”) to check rest results

Writing a whole bunch of “if / then / else / t.Fail()” boilerplate quickly becomes annoying. The testify/assert package (https://github.com/stretchr/testify/assert) makes all of this a bit easier. Tests written with assert are easier to read because they are more succinct, and the assertions are more obvious.

Have a look at the internal/cruft package:

- It contains an interface (Crufter) and two implementations which both need to be tested
- The test files are split up - one for each implementation
- Note the various assertions used (there are many more available)

Suites (“testify/suite”) for more complex testing setups

Sometimes your tests require quite a bit of complicated setup (injecting dependencies, configuring mocks, logging to buffers etc). Having to do this repeatedly becomes tedious. Again testify offers a solution: “suites”, which allow you to define test dependencies once, and re-use for all your test cases:

Have a look at the internal/yaica package:

- yaica_test.go sets up the suite with dependencies
- encruft_eggs_test.go and replace_eggs_test.go use the suite’s dependencies to test the two methods
- Note the use of a programmable mock, and byte buffer for logging tests

Test “runs” (the native version of suites?)

The “testing” package does provide a basic facility to run multiple test cases, but in practise it is fairly clunky.

Take a look at the internal/reruns package:

- Test cases are defined as a slice of whatever data you need
- t.Run() runs the test cases for you
- Asserting the results is still quite manual
- I think it is quite ugly

Code coverage report: find out what you’ve missed

By now you’ve probably realised that the /internal/cruft package only has 66.7% coverage. Some of you may already know why, but let’s do this the proper way.

- `cd /internal/cruft`
- `go test -coverprofile=coverage.out`
- `go tool cover -html=coverage.out`

Code coverage report

It turns out that we’ve only tested the “happy” path (where the things being tested didn’t return any errors).

Fortunately both of the things being tested in the /internal/cruft package can pretty easily be made to return errors (by providing them with empty strings).

Do you feel like going for 100%? Add some more tests!