Testing and Benchmarking in Go

This lesson shows how to test a program before running it and therefore how to save it from panicking beforehand.

Testing with tool

Every new package should contain sufficient documentation and test code. We already used Go's testing tool `go test` in Chapter 7. Here, we will elaborate on its use with some more examples.

A special package called `testing` provides support for automated testing, logging, and error reporting. It also contains some functionality for benchmarking functions.

> **Remark**: for Windows, every time you see the folder pkg/linux_amd64, replace it with pkg/windows.

To unit-test a package, you write several tests that you can frequently run (after every update) to check the correctness of your code in small units. For that, we will have to write a set of Go source files that will exercise and test our code. This *test-programs* must be *within the same package*, and the files must have names of the form *_test.go. The test code is separated from the actual code of the package.

These _test programs are **NOT** compiled with the normal Go-compiler, so they are not deployed when you put your application into production. Only `go test` compiles all programs: the normal and the test programs.

Those files must import the `testing` package. In them, we write global

Those files must import the `testing` package. In them, we write global functions with names starting with `TestZzz`, where *Zzz* is an alphabetic

description of the function to be tested, like `TestFmtInterface`, `TestPayEmployees`, and so on. Those test functions should have a header of the form: `func TestAbcde(t *testing.T)`, where `T` is a struct type passed to `TestZzz` functions that manages test state and supports formatted test logs, like `t.Log`, `t.Error`, `t.ErrorF`, and so on. At the end of each of these functions, the output is compared with what is expected, and if these are not equal, an error is logged. A successful test function returns. To signal a failure, we have the following functions:

- `func (t *T) Fail()`: marks the test function as having failed, but continues its execution.
- `func (t *T) FailNow()`: marks the test function as having failed and stops its execution. All other tests in this file are also skipped, and execution continues with the next test file.
- `func (t *T) Log(args ...interface{})`: the args are formatted using default formatting and the text is logged in the error-log.
- `func (t *T) Fatal(args ...interface{})`: this has the combined effect of `func (t *T) Log(args ...interface{})` followed by `func (t *T) FailNow()`.

Then, run `go test`. This compiles the test-programs and executes all the `TestZzz-functions`. If all tests are successful, the word **PASS** will be printed. The `go test` can also take one or more test programs as parameters, and some options. With the option `-v`, each test function that is run and its test-status is mentioned. For example:

```
go test -v fmt_test.go
=== RUN fmt.TestFlagParser
--- PASS: fmt.TestFlagParser
=== RUN fmt.TestArrayPrinter
--- PASS: fmt.TestArrayPrinter
...
```

The testing package also contains some types and functions for simple benchmarking; the test code must then contain function(s) starting with `BenchmarkZzz` and take a parameter of type `*testing.B`, e.g.:

```
func BenchmarkReverse(b *testing.B) {
  ...
}
```

The command `go test -test.bench=.*` executes all these functions. This will call the functions in the code **N** number of times (e.g., N = 1000000). Additionally, it shows this **N** and the average execution time of the functions in `ns`(ns/op). If the functions are called with `testing.Benchmark`, you can also run the program.

Now that you know the basics of testing in Go, let's test an application in the next lesson.