4 - Testing and Profiling

Why Are Tests Important?

Types of Testing

- End-to-end tests:
  - Test the program as a whole
  - Treat the program as a black box
  - Good for QA, but late
- Unit test:
  - Test smallest elements of code (usually functions)
  - Check output and state changes on a granular level
  - Useful for early bug fixing and during code maintenance

Which Tests Are Good?

- Meaningful:
  - Test actual use cases of the functions
  - Test against valid and invalid input
  - Examine ouptput and expected state changes
- Comprehensive:
  - Explore as many code pathways as possible
- Metric:
  - Code coverage - the percentage of code traversed during tests
  - Strive for 100%, but only 80-90% is usually attainbale

Unite Tests in Go

- Test code is contained in xxxx_test.go
- Test code is not included in the build
- Test code cannot be sahred between packages
- Implementation:

```
1 func TestXxxx (t *testing.T) {}
```

An Ideal Unit Test

src/handlers/userHandlers_test.go

```
1 package handlers
```

```
2
3  import (
4      "net/http"
5      "github.com/build-restful-apis/src/user"
6      "testing"
7      "reflect"
8      "io/ioutil"
9      "bytes"
10     "gopkg.in/mgo.v2/bson"
11     "encoding/json"
12 )
13
14 func TestBodyToUser(t *testing.T) {
15     valid := &user.User {
16         ID: bson.NewObjectId(),
17         Name: "John",
18         Role: "Tester",
19     }
20
21     js, err := json.Marshal(valid)
22
23     if err != nil {
24         t.Errorf("Error marshalling a valid users: %s", err)
25         t.FailNow()
26
27     }
28
29     ts := []struct {
30         txt string
31         r *http.Request
32         u *user.User
33         err bool
34         exp *user.User
35     } {
36         // request is nil
37         {
38             txt : "nil request",
39             err: true,
40         },
41         // request is empty
42         {
43             txt: "empty request body",
44             r: &http.Request{},
45             err: true,
46         },
47         // empty user
48         {
49             txt: "emtpy user",
50             r: &http.Request {
51                 Body: ioutil.NopCloser(bytes.NewBufferString("{}")),
52             },
53             err: true,
54         },
55         // malformed data
56         {
57             txt: "malformed data",
```

```
            r: &http.Request {
                Body: ioutil.NopCloser(bytes.NewBufferString(`{"id":12}`)),
            },
            u: &user.User{},
            err: true,
        },
        // valid data
        {
            txt: "valid request",
            r: &http.Request {
                Body: ioutil.NopCloser(bytes.NewBuffer(js)),
            },
            u: &user.User{},
            exp: valid,
        },
    }

    for _, tc := range ts {
        t.Log(tc.txt)
        err := bodyToUser(tc.r, tc.u)
        if tc.err {
            if err == nil {
                t.Error("Expected error, got none.")
            }
            continue
        }

        if err != nil {
            t.Errorf("Unexpected error: %s", err)
            continue
        }

        if !reflect.DeepEqual(tc.u, tc.exp) {
            t.Error("Unmarsshaled data is different:")
            t.Error(tc.u)
            t.Error(tc.exp)
        }
    }

}
```

A Minimum Viable Unit Test

Simple Tests

- Focus on valid, working code and usage
- Do not cover edge cases
- Better than no tests

Benchmarking

Benchmarks in Go

- Benchmarks are specialized tests
- Benchmarks focus on performance
- Run on demand
- Implementation:
    - func BenchmarkXxxx (b *testing.B) {}
    - Loop -for i := 0; i < b.N; i++ { //code }
    - Run -go test -bench
    - TestMain is also executed