Testing Go REST API

Michel Casabianca
casa@sweetohm.net

Nowadays, no developer would dare to release an API without tests. But writing API tests is time consuming and painful. We will see in this article how we can do so without pain.

Source code for examples of this article are available in this Github project.

Unit Testing

If you Google *golang api testing*, you will find articles to unit test your API with Go. Let's consider following API:

```
package main

import (
        "fmt"
        "github.com/gin-gonic/gin"
)

func Hello(ctx *gin.Context) {
        name := ctx.Param("name")
        ctx.JSON(200, gin.H{"message": fmt.Sprintf("Hello %s!", name)})
}

func main() {
        engine := gin.Default()
        engine.GET("/hello/:name", Hello)
        engine.Run()
}
```

We could test it with a unit test as follows:

```
func TestHello(t *testing.T) {
        engine := Engine()
        recorder := httptest.NewRecorder()
        request, err := http.NewRequest("GET", "/hello/World", nil)
        if err != nil {
                t.Fatalf("building request: %v", err)
        }
        engine.ServeHTTP(recorder, request)
        if recorder.Code != 200 {
                t.Fatalf("bad status code: %d", recorder.Code)
        }
        var response Response
        body := recorder.Body.String()
        if err != nil {
```

```
                t.Fatalf("reading response body: %v", err)
        }
        if err := json.Unmarshal([]byte(body), &response); err != nil {
                t.Fatalf("parsing json response: %v", err)
        }
        if response.Message != "Hello World!" {
                t.Fatalf("bad response message: %s", response.Message)
        }
}
```

In this code we perform following tasks:

- Instantiate server
- Build a recorder to record request response
- Execute request
- Make following assertions:
    - Response status code must be *200*
    - Response body contents must be valid JSON with expected message

This is quite a tedious code to write, uncreative and error prone.

Venom Tests

It would be much simpler and faster to describe request and response, in YAML format for instance, as follows:

```
request:
    url: "http://127.0.0.1:8080/hello/World"
    method: GET
response:
    statusCode: 200
    json:
        message: "Hello World!"
```

This is:

- Far simpler to write
- Faster
- Less error prone
- More explicit

To test what happens when calling an unknown URL, we could write:

```
func TestHelloNotFound(t *testing.T) {
        engine := Engine()
        recorder := httptest.NewRecorder()
        request, err := http.NewRequest("GET", "/hello", nil)
        if err != nil {
                t.Fatalf("building request: %v", err)
```

```
        }
        engine.ServeHTTP(recorder, request)
        if recorder.Code != 404 {
                t.Fatalf("bad status code: %d", recorder.Code)
        }
}
```

Once more, this could be replaced to your benefit, with:

```
request:
    url: "http://127.0.0.1:8080/hello"
    method: GET
response:
    statusCode: 404
    body: "404 page not found"
```

These request and response descriptions in YAML format are, pretty much, integration test files in Venom test format for Tavern executor.

We, at Intercloud, have developed this Tavern executor for Venom to have the best of both worlds of Tavern and Venom.

Source for these tests are:

```
name: Hello
vars:
  URL: "http://127.0.0.1:8080"

testcases:

- name: Test hello
  steps:
  - type: tavern
    request:
      url: "{{.URL}}/hello/World"
      method: GET
    response:
      statusCode: 200
      json:
        message: "Hello World!"

- name: Test not found
  steps:
  - type: tavern
    request:
      url: "{{.URL}}/hello"
      method: GET
    response:
      statusCode: 404
      body: "404 page not found"
```

We can run these tests with following command line:

```
$ venom run test.yml
 • Hello (test.yml)
        • Test-hello SUCCESS
        • Test-not-found SUCCESS
```

These are integration tests as we test the whole application, as opposed to unit tests that test an isolated piece of code. One could think that we could thus do without unit tests. This is plain wrong, unit and integration tests are complementary.

Achieving tests this way allows you to cover far more error cases. Time you save testing this way may be invested in covering all expected status codes. Furthermore, these tests are a way to document your API in a legible way.

Code Coverage

Downside of integration tests is that it is often quite difficult to measure code coverage. This is easy to measure code coverage for Go unit tests, but nothing is provided for integration tests.

Fortunately, it is possible to use tools that measure unit tests coverage for integration tests. The trick is to run server and Venom tests in a unit test. When tests are finished, Go has measured code coverage and it is possible to generate a report.

This is what we do in following test:

```
//go:build integration
// +build integration

package main

import (
        "net"
        "os/exec"
        "testing"
        "time"
)

func WaitServer() {
        timeout := 10 * time.Millisecond
        for {
                conn, err := net.DialTimeout("tcp", "127.0.0.1:8080", timeout)
                if err == nil {
                        conn.Close()
                        return
                }
                time.Sleep(timeout)
        }
}

func TestIntegration(t *testing.T) {
        go Engine().Run()
        WaitServer()
        out, err := exec.Command("venom", "run", "*.yml").Output()
```

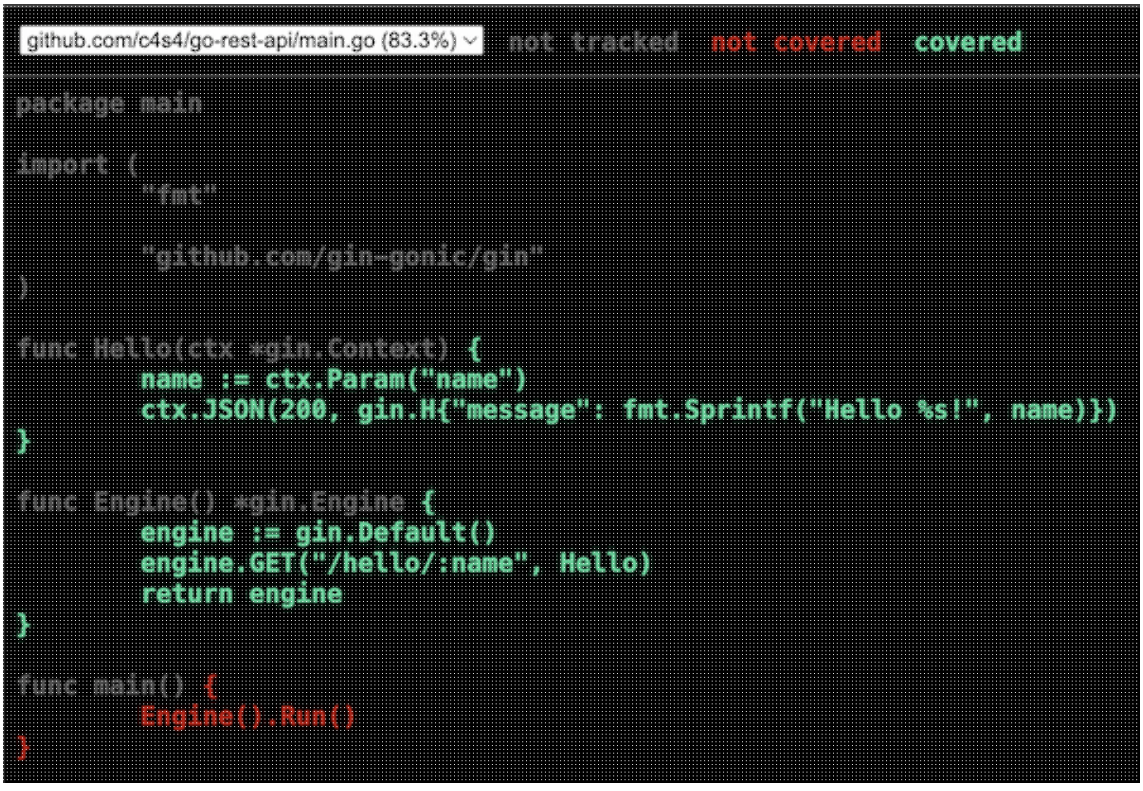
```
		if err != nil {
			t.Fatalf("running venom: %s", string(out))
		}
	}
```

Function `WaitServer()` waits for server to start without calling any specific route. Another trick in this code is to add *integration* compilation tag in source header. This way, this source will be compiled only if indicating compiler to process this tag, with `-tag integration` option.

We can run this test and generate coverage report with following command lines:

```
mkdir -p build
go test -c -o build/go-rest-api-integ -covermode=set -coverpkg=./... -tags integrat
build/go-rest-api-integ -test.coverprofile=build/coverage-integ.out
go tool cover -html=build/coverage-integ.out -o build/coverage-integ.html
```

This will produce an HTML report as follows:



Rapport de Couverture de Test

Of course, this sample report is not so impressive, but it is very useful for large projects to track code that is not covered with integration tests and thus tells tests you must write to check uncovered error cases.

Enjoy!