Test des API REST en Go

Michel Casabianca
casa@sweetohm.net

De nos jours, aucun développeur sérieux ne songerait à releaser une API non testée. Mais l'écriture de tests pour une API REST est souvent longue et pénible. Nous allons voir dans cet article comment écrire de tels tests sans douleur.

Les sources des exemples de cet article sont disponibles dans ce projet Github.

Tests unitaires

Si l'on cherche sur Google, on trouvera des articles décrivant la manière de tester une API avec des tests unitaires Go. Si nous considérons l'API suivante :

```
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 Engine() *gin.Engine {
        engine := gin.Default()
        engine.GET("/hello/:name", Hello)
        return engine
}

func main() {
        if err := Engine().Run(); err != nil {
                println("ERROR running server:", err.Error())
        }
}
```

Nous pourrions la tester avec un test unitaire comme suit :

```
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)
        }
}
```

Nous y réalisons les tâches suivantes :

- Instanciations du serveur
- Construction d'un recorder pour enregistrer la réponse à la requête
- Exécution de la requête
- Réalisation des assertions suivantes :
    - ♦ Le status code de la réponse doit être *200*
    - ♦ Le contenu est du JSON valide avec le message attendu

C'est assez pénible à coder, peu créatif et sujet à erreurs.

Tests Venom

Il serait beaucoup plus simple et rapide de décrire la requête et le résultat attendu, au format YAML par exemple, comme suit :

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

C'est tout à la fois :

- Bien plus simple à écrire
- Plus rapide
- Moins sujet à erreurs
- Plus explicite

Pour tester ce qu'il se passe lorsque nous appelons une URL inconnue, nous pourrions ajouter ce test :

```
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)
        }
}
```

Encore une fois, ce test pourrait être avantageusement remplacé par :

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

Ces descriptions au format YAML des requêtes et des réponses attendues sont précisément, à peu de choses près, des fichiers de tests d'intégration au format de l'outil de test Venom avec l'executor Tavern.

Nous avons, chez Intercloud, développé cet executor Tavern pour Venom afin de prendre le meilleur des deux mondes de Tavern et de Venom.

Le source du test est le suivant :

```
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"
```

Que nous pouvons exécuter avec la ligne de commande :

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

On parle alors de tests d'intégration car nous testons l'ensemble de notre logiciel, par opposition aux tests unitaires qui n'en testent qu'une petite partie isolément. On pourrait penser qu'ainsi on peut se passer de tests unitaires. Il n'en est rien et les test unitaires et d'intégration sont complémentaires.

Réaliser ainsi ces tests d'intégration permet de couvrir bien plus de cas d'erreurs. Le temps gagné à écrire ces tests permet de couvrir tous les status code attendus. D'autre part, ces tests permettent de documenter l'API de manière simple.

Couverture de test

L'inconvénient des tests d'intégration est qu'il est souvent difficile de mesurer la couverture de test. C'est en Go très facile de le faire pour les tests unitaires, mais dans le cas des tests d'intégration, rien n'est prévu.

Heureusement, il est possible d'utiliser les outils de mesure de couverture du langage Go pour les tests d'intégration. L'astuce consiste à lancer le serveur dans un test unitaire et d'y lancer Venom. Lorsque les tests d'intégration sont passés, Go a mesuré la couverture de test et il est possible de générer un rapport.

Ceci est mis en pratique dans le test suivant :

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

package main

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

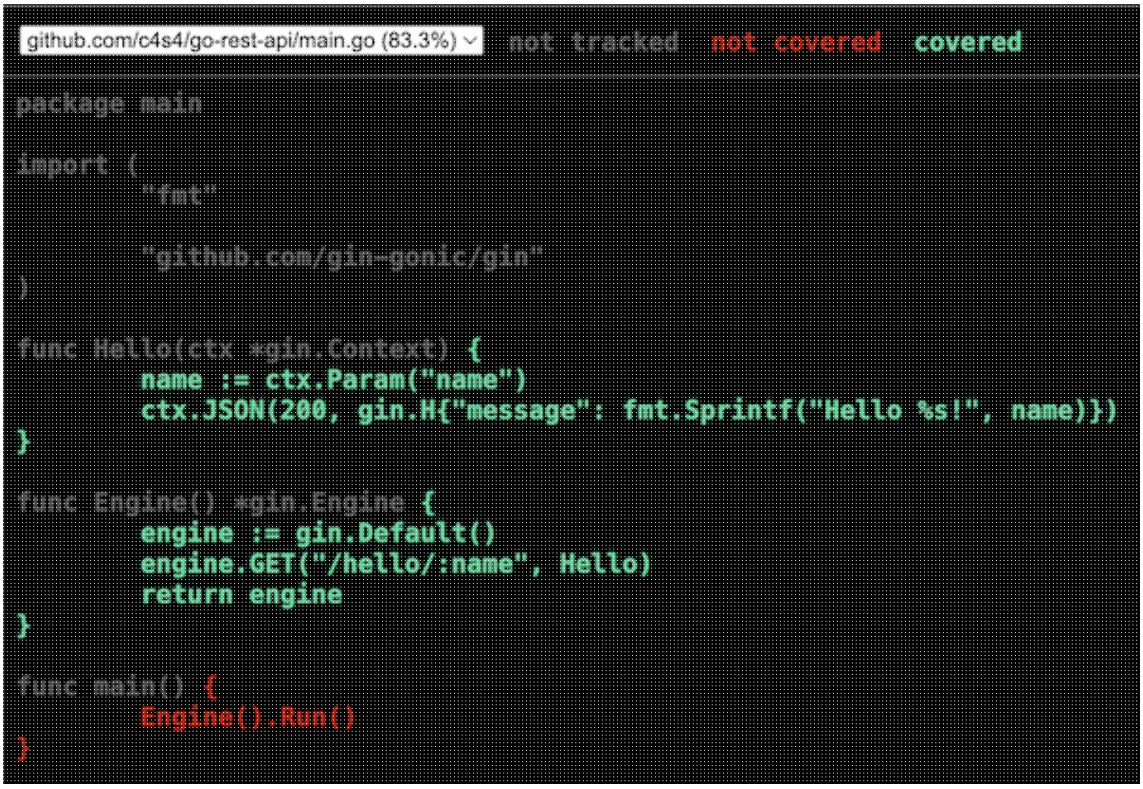
```
func WaitServer() {
        timeout := 10 * time.Millisecond
        for {
                conn, err := net.DialTimeout("tcp", "127.0.0.1:8080", timeout)
                if err == nil {
                        conn.Close()
                        return
                }
                time.Sleep(timeout)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

func TestIntegration(t *testing.T) {
        go Engine().Run()
        WaitServer()
        out, err := exec.Command("venom", "run", "test.yml").Output()
        if err != nil {
                t.Fatalf("running venom: %s", string(out))
        }
}
```

La fonction `WaitServer()` permet d'attendre que le serveur soit démarré sans avoir a appeler une route quelconque. Une autre astuce de ce code consiste a ajouter une tag de compilation *integration* à l'en-tête du source. Ainsi, ce source ne sera compilé que si l'on indique au compilateur Go que nous prenons ce tag en charger, avec l'option en ligne de commande `-tag integration`.

Nous pouvons lancer ce test d'intégration avec génération d'une rapport de couverture avec les commandes suivantes :

```
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
```

Ce qui produira le rapport de couverture de test suivant dans un fichier HTML :



Rapport de Couverture de Test

Bien sûr, un tel exemple ce rapport n'est pas très impressionnant, mais sur de gros projets, il permet de tracer le code qui a été couvert par des tests, les tests à réaliser pour tester les cas d'erreur et ainsi de suite.

Enjoy!