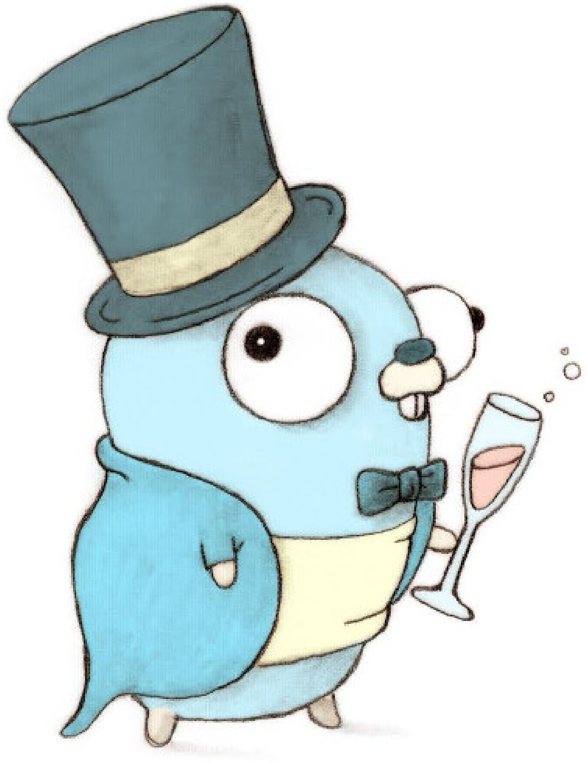
Mocking all the things

Golang Strasbourg - Go SXB Go!

4 December 2016

Leo Unbekandt
CTO at Scalingo

Go SXB Go



Apologies for the delay, but better later than never

Testing with go

Example:

```go
import "errors"

func Foo() error {
    return errors.New("error")
}
```

Run

Can be tested with:

```go
import "testing"

func TestFoo(t *testing.T) {
    err := Foo()
    if err != nil {
        t.Errorf("expecting nil error, got %v", err)
    }
}
```

Run

Testing a third-party client

A sample Github client:

```go
package github

type GithubClient struct {
    Credentials GithubCredentials
}

func (c *GithubClient) GetRepositories() (Repositories, error) {
    // HTTP request using Credentials
    // ...
    return repositories, nil
}
```

Run

Testing a third-party client

How would we test it?

```go
import "testing"

func TestGithubClient_GetRepositories(t *testing.T) {
    setupTestServer()
    defer teardownTestServer()

    c := &GithubClient{}
    if err != nil {
        t.Errorf("expected nil, got %v", err)
    }

    // etc.
}
```

Run

Testing a package which uses a third-party client package

```go
type User struct {
    GithubUsername string
}

func (u *User) RepositoriesCount() (int, error) {
    c := &github.GithubClient{
        Credentials: u.GithubCredentials(),
    }
    rs, err := c.GetRepositories()
    if err != nil {
        return 0, err
    }
    return len(rs), nil
}
```

Run

Testing a package which uses a third-party client package

```go
import "testing"

func TestUser_RepositoriesCount(t *testing.T) {
    u := User{}
    n, err := u.RepositoriesCount()
    // ...
}
```

Run

Problem there, how do you handle the internal GithubClient

- Credentials?
- Automation?
- Repeatability?

Possible answer

A custom HTTP server?

- No easy to configure
- You don't know what is doing the client, it's opaque

Dependency Injection

```go
type User struct {
    GithubUsername string
}

func (u *User) RepositoriesCount() (int, error) {
    c := &github.GithubClient{
        Credentials: u.GithubCredentials(),
    }
    rs, err := c.GetRepositories()
    if err != nil {
        return 0, err
    }
    return len(rs), nil
}
```

Here we can't do anything, everything is frozen,
the GithubClient should be interchangeable.

Dependency Injection

```go
type User struct {
    GithubUsername string
    GithubClient   *github.GithubClient
}

func (u *User) RepositoriesCount() (int, error) {
    c := u.GetGithubClient()
    rs, err := c.GetRepositories()
    if err != nil {
        return 0, err
    }
    return len(rs), nil
}

func (u *User) GetGithubClient() *github.GithubClient {
    if u.GithubClient != nil {
        return u.GithubClient
    } else {
        return &github.GithubClient{
            Credentials: u.GithubCredentials(),
        }
    }
}
```

Run

Dependency Injection - Tests

```go
import "testing"

func TestUser_RepositoriesCount(t *testing.T) {
    c := github.GithubClient{Credentials: testCredentials()}
    u := User{GithubClient: c}
    n, err := u.RepositoriesCount()
    // ...
}
```

Run

Dependency Injection

Better configurability

But we're still using the 3rd party client

Here comes interfaces.

Back in our `github` package, we create an interface with the methods of our client.

```
type API interface {
    GetRepositories() (Repositories, error)
}

type GithubClient struct {
    Credentials GithubCredentials
}

func (c *GithubClient) GetRepositories() (Repositories, error) {
    // HTTP request using Credentials
    // ...
    return repositories, nil
}
```

Run

New version of User

```go
type User struct {
    GithubUsername string
    GithubClient   github.API
}

func (u *User) RepositoriesCount() (int, error) {
    c := u.GetGithubClient()
    rs, err := c.GetRepositories()
    if err != nil {
        return 0, err
    }
    return len(rs), nil
}

func (u *User) GetGithubClient() github.API {
    if u.GithubClient != nil {
        return u.GithubClient
    } else {
        return &github.GithubClient{
            Credentials: u.GithubCredentials(),
        }
    }
}
```

Run

Let write our mock

It should respect the github.API interface

```
package githubmock

type Client struct{}

func (c *Client) GetRepositories() (github.Repositories, error) {
    return github.Repositories{}, nil
}
```

Run

And rewrite the test

```
import "testing"

func TestUser_RepositoriesCount(t *testing.T) {
    c := githubmock.Client{}
    u := User{GithubClient: c}
    n, err := u.RepositoriesCount()
    // ...
}
```

Run

A few words about what we have

In our tests, we're replacing the standard third-party client by a mock

What's lacking?
- Tracability
- Configurability?

Easy as pie

```go
type GetRepositoriesData struct {
    Repositories github.Repositories
    Err          error
}

type Client struct {
    GetRepositoriesData
}

func (c *Client) GetRepositories() (github.Repositories, error) {
    return c.GetRepositoriesData.Repositories, c.GetRepositoriesData.Err
}
```

Run

Easy as pie

```
import "testing"

func TestUser_RepositoriesCount(t *testing.T) {
    c := githubmock.Client{}
    c.GetRepositoriesData = githubmock.GetRepositoriesData{
        Repositories: github.Repositories{{
            Name: "repo1",
        }, {
            Name: "repo2",
        }},
    }
    u := User{GithubClient: c}
    n, err := u.RepositoriesCount()
    if err != nil {
        t.Errorf("expecting nil err, got %v", err)
    }
    if n != 2 {
        t.Errorf("expecting 2 repositories, got %v", n)
    }
    // ...
}
```

Run

And what about calls?

We want to know how the client is calls, let's change the API

```
type API interface {
    GetRepositories(GetRepositoriesOpts) (Repositories, error)
}

type GetRepositoriesOpts struct {
    StrictOwner bool
}

func (c *GithubClient) GetRepositories(opts GetRepositoriesOpts) (Repositories, error) {
    // ...
    return repositories, nil
}
```

Run

And what about calls?

We have to save how the method is called

```
type Client struct {
    GetRepositoriesData
    GetRepositoriesCalls []github.GetRepositoriesOpts
}

func (c *Client) GetRepositories(opts github.GetRepositoriesOpts) (github.Repositories, error) {
    c.GetRepositoriesCalls = append(c.GetRepositoriesCalls, opts)
    return c.GetRepositoriesData.Repositories, c.GetRepositoriesData.Err
}
```

Run

Check the calls

```go
import "testing"

func TestUser_RepositoriesCount(t *testing.T) {
    c := githubmock.Client{}
    c.GetRepositoriesData = githubmock.GetRepositoriesData{
        Repositories: github.Repositories{{Name: "repo1"}, {Name: "repo2"}},
    }
    u := User{GithubClient: c}
    n, err := u.RepositoriesCount()
    if err != nil {
        t.Errorf("expecting nil err, got %v", err)
    }
    if n != 2 {
        t.Errorf("expecting 2 repositories, got %v", n)
    }
    if len(c.GetRepositoriesCalls) != 1 {
        t.Errorf("expecting github client to be called once, it wasn't")
    }
    if !c.GetRepositoriesCalls[0].StrictOwner {
        t.Errorf("expecting StrictOwner to be false on github client, it was true")
    }
}
```

Run

That's all folks

- Good design of tests is important, you'll regret it laster if you don't take care of that
- Helps you designing your code to be testable

Don't forget, mock everything!

Post Scriptum

My example work if the client is used in a threadsafe manner, otherwise you have to adapt the Mock

```go
type Client struct {
    GetRepositoriesData
    GetRepositoriesCalls []github.GetRepositoriesOpts
}

func (c *Client) GetRepositories(opts github.GetRepositoriesOpts) (github.Repositories, error) {
    c.GetRepositoriesCalls = append(c.GetRepositoriesCalls, opts)
    return c.GetRepositoriesData.Repositories, c.GetRepositoriesData.Err
}
```

Run

Thank you

Leo Unbekandt
CTO at Scalingo
@Soulou (http://twitter.com/Soulou)