Behavior Driven Development with Ginkgo and Gomega

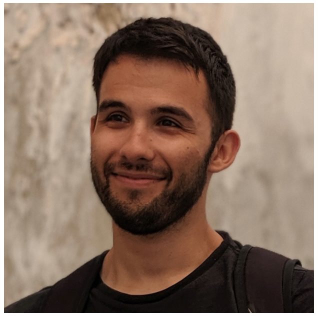
Ginkgo
A Golang BDD Testing Framework

GΩmega
Ginkgo's Preferred Matcher Library

Roberto Jiménez Sánchez

Software Engineer at Delivery Hero

About me



- Backend Engineer in **Ordering Experience** Squad 🛒
- Background: Cloud Foundry/Knative committer and Software Engineer at IBM Cloud
- Gopher since 2014
- **Fun fact**: secretly a bartender 🍻

Agenda

- Test-Driven Development
- Behaviour-Driven Development
- Ginkgo
- Gomega

Test-Driven Development (TDD)

1. 📝 write test
2. 🔴 watch it fail
3. 👩‍💻 add new code
4. ✅ test are green
5. 🔁 Repeat

Make sure that your code is fixing your problem

Behavior-Driven Development (BDD)

- English description of tests
- Derived directly from specifications
- Comprehensive to non-technical readers

```
var _ = Describe("Set", func() {
    Describe("Contains", func() {
        Context("When red has been added", func() {
            It("Should contain red", func() {
            })
        })
    })
})
```

Even if you don't TDD, consider...

how will others use my code?

(e.g. You go read the specs and then see it works as expected)

Ginkgo

A Golang BDD Testing Framework

Ginkgo (⭐3.6k)

- Ginkgo is a BDD(Behavior Driven Development)-style testing framework for Golang, and its preferred matcher library is Gomega.
- Help you efficiently write **descriptive** and **comprehensive** tests
- Support Test Driven Development (TDD)

Ginkgo Docs

How does it achieves that?

By improving the development flow

📝 write test

- find the place quicker (e.g. structure, readability, etc)
- write less code by reusing
  - don't reinvent the wheel (e.g. table tests, etc)

🔴 watch it fail

- run the test or tests you want quickly

💻 add new code

✅ test are green

- run the test or tests you want quickly

🔁 Repeat

Alternative BDD-frameworks

- Convey: 5.3k ⭐

  smartystreets/goconvey

- Godog: 896 ⭐

  DATA-DOG/godog

- Goblin: 652 ⭐

  franela/goblin

Let's get started

1. Install Ginkgo CLI

```
go get github.com/onsi/ginkgo/ginkgo
```

2. Bootstrap tests in a package

```
$ cd path/to/books
$ ls
book.go
$ ginkgo bootstrap
$ ls
book.go
books_test.go # Generated
```

You should see something like this

```
package books_test

import (
    . "github.com/onsi/ginkgo"
    . "github.com/onsi/gomega"
    "testing"
)

func TestBooks(t *testing.T) {
    RegisterFailHandler(Fail)
    RunSpecs(t, "Books Suite")
}
```

3. Generate specs for your code

```
  $ ginkgo generate book
  $ ls
  book.go
  book_test.go #Generated
  books_test.go
```

```
package books_test

import (
    . "/path/to/books"
    . "github.com/onsi/ginkgo"
    . "github.com/onsi/gomega"
)

var _ = Describe("Book", func() {

})
```

4. Write your first spec

```go
package books_test

import (
	. "/path/to/books"
	. "github.com/onsi/ginkgo"
	. "github.com/onsi/gomega"
)

var _ = Describe("Book", func() {
  It("works!", func() {
  })
})
```

5. Run the tests

```
$ ginkgo #or go test

    === RUN TestBootstrap

    Running Suite: Books Suite
    ==========================
    Random Seed: 1378936983

    Will run 1 of 1 specs


    Ran 0 of 0 Specs in 0.000 seconds
    SUCCESS! -- 1 Passed | 0 Failed | 0 Pending | 0 Skipped

    --- PASS: TestBootstrap (0.00 seconds)
    PASS
    ok      books   0.019s
```

Or make it fail:

```
$ ginkgo ./.
    Running Suite: Books Suite
    ==========================
    Random Seed: 1580299170
    Will run 1 of 1 specs

    • Failure [0.000 seconds]
    Book
    /Users/r.jimenez/workspace/013-ginkgo-gomega/books/book_test.go:9
      works! [It]
      /Users/r.jimenez/workspace/013-ginkgo-gomega/books/book_test.go:10

      Must fail!

      /Users/r.jimenez/workspace/013-ginkgo-gomega/books/book_test.go:11
    ------------------------------


    Summarizing 1 Failure:

    [Fail] Book [It] works!
    /Users/r.jimenez/workspace/013-ginkgo-gomega/books/book_test.go:11

    Ran 1 of 1 Specs in 0.001 seconds
    FAIL! -- 0 Passed | 1 Failed | 0 Pending | 0 Skipped
    --- FAIL: TestBooks (0.00s)
    FAIL
```

Some useful commands

- `ginkgo watch`: trigger test execution when changes are detected.
- `ginkgo --dryRun`: dry-run your tests.
- `ginkgo --failFast`: make the tests fail as soon as one test fails
- `ginkgo --untilItFails`: run tests until they fail.
- `ginkgo --randomizeAllSpecs`: run specs in a random order.
- `ginkgo --timeout`: set a global timeout for the whole text execution.
- `ginkgo --flakeAttempts 3`: retries in case of tests failure.

Anatomy of a test

WHEN: X happens

GIVEN: Y is true

THEN: Z must be true

WHEN

- `Describe` : individual behaviours of the code.
- `Context`: circumstances of those behaviours

```go
var _ = Describe("Book", func() {
    Describe("loading from JSON", func() {
        Context("when the JSON parses succesfully", func() {
            It("should populate the fields correctly", func() {})

            It("should not error", func() {})
        })

        Context("when the JSON fails to parse", func() {
            It("should return the zero-value for the book", func() {})

            It("should error", func() {})
        })
    })

    Describe("Extracting the author's last name", func() {
        It("should correctly identify and return the last name", func() {})
    })
})
```

GIVEN

- `BeforeSuite`, `AfterSuite` common for all tests, and executed only once (e.g. booting a database)
- `JustBeforeEach`, `BeforeEach` , `JustAfterEach` `AfterEach` are used for common setup.
  - closures are heavily used to share variables across tests.
  - when using nested contexts, they are executed from the outermost to innermost of each type in the following order:

```go
package books_test

import (
    . "github.com/onsi/ginkgo"
    . "github.com/onsi/gomega"

    "your/db"

    "testing"
)

var dbRunner *db.Runner
var dbClient *db.Client

func TestBooks(t *testing.T) {
    RegisterFailHandler(Fail)

    RunSpecs(t, "Books Suite")
}

var _ = BeforeSuite(func() {
    dbRunner = db.NewRunner()
    _ = dbRunner.Start()

    dbClient = db.NewClient()
    _ = dbClient.Connect(dbRunner.Address())
})

var _ = AfterSuite(func() {
    dbClient.Cleanup()
    dbRunner.Stop()
})
```

```go
var _ = Describe("Book", func() {
    var (
        json string
        book Book
        err error
    )

    BeforeEach(func() {
        json = `{
            "title":"Les Miserables",
            "pages":1488
        }`
    })

    JustBeforeEach(func() {
        book, err = NewBookFromJSON(json)
    })

    Describe("loading from JSON", func() {
        Context("when the JSON fails to parse", func() {
            BeforeEach(func() {
                json = `{
                    "title":"Les Miserables",
                    "pages":1488oops
                }`
            })

            It("should return the zero-value for the book", func() {})
        })
    })
})
```

THEN

- `It` or `Specify` for a single spec

```
var _ = Describe("Book", func() {
    It("can be loaded from JSON", func() {
        book := NewBookFromJSON(`{
            "title":"Les Miserables",
            "author":"Victor Hugo",
            "pages":1488
        }`)

        // Do Something
        // Check your expectations
    })
})
```

(This is the actual test code once all the setup and cleanup part has been defined in the context)

Focused Tests

Adding the prefix `F` to any `It`, `Describe` or `Context` allows to run a particular set of tests you are interested at the moment

```
var _ = Describe("Book", func() {
    // Tests within this Describe will run
    FDescribe("loading from JSON", func() {
        Context("when the JSON parses succesfully", func() {
            It("should populate the fields correctly", func() {})

            It("should not error", func() {})
        })

        Context("when the JSON fails to parse", func() {
            It("should return the zero-value for the book", func() {})

            It("should error", func() {})
        })
    })
    // Rest of the tests are ignored
    Describe("Extracting the author's last name", func() {
        It("should correctly identify and return the last name", func() {})
    })
})
```

Pending tests

In the same way, you to mark one or multiple tests as `Pending` with the prefix `P` to ignore them:

```
var _ = Describe("Book", func() {
    Describe("loading from JSON", func() {
        Context("when the JSON parses succesfully", func() {
            It("should populate the fields correctly", func() {})

            It("should not error", func() {})
        })

        Context("when the JSON fails to parse", func() {
            It("should return the zero-value for the book", func() {})

            It("should error", func() {})
        })
    })
    // Ignore all the tests inside this Describe and run the rest
    PDescribe("Extracting the author's last name", func() {
        It("should correctly identify and return the last name", func() {})
    })
})
```

Convert standard tests to Ginkgo tests

```
$ ginkgo convert path/to/mypackage
```

GΩmega

Ginkgo's Preferred Matcher Library

Gomega (⭐1k)

- Gomega is a **matcher/assertion** library. It is best paired with the Ginkgo BDD test framework, but can be adapted for use in other contexts too.
- Focused on **readability** and **modularity**.
- Alternative to Testify (⭐9.5k) stretchr/testify

Gomega

Matchers

- Matchers for anything you can expect as you would expect like `Equal` , `BeNil`, `BeEmpty` , `ContainElement`, `BeTrue`, `BeFalse` , `MatchJSON`, etc.
- Matchers can be combined as well.

```
MatchError(ContainSubstring("beginning of my error"))
```

- You can define **custom GomegaMatchers** by implement `GomegaMatcher` from [`github.com/onsi/gomega/types`]
- More here in [Gomega Godoc]

Synchronous assertions

Assertions start with `Expect` and follow the following syntax:

```
Expect(foo).To(Equal("foo"))
// For the opposite
Expect(foo).ToNot(Equal("bar"))
```

Check errors

```go
err := DoSomething()
Expect(err).ToNot(HaveOcurred())

// or alternatively

Expect(DoSomething()).To(Succeed())

// Or check a concrete error

Expect(err).To(MatchError("expected error"))

// Check if a concrete error contains some substring
err := errors.New("didn't work: because you weren't lucky")

Expect(err).To(MatchError(ContainSubstring("didn't work")))
```

Notice that we don't have to pass the value to the matchers.

Checking maps

```go
Expect(myMap).To(HaveKey("foo"))
Expect(myMap).To(HaveKeyWithValue("foo", "bar"))


// Checking multiple things at once
Expect(myMap).To(
  SatisfyAll(
    HaveKey("foo"),
    HaveKey("bar"),
  ),
)
```

Asynchronous assertions

`Eventually`

Checks if the assertion eventually passes.

```
Eventually(func() []int {
    return thing.SliceImMonitoring
}).Should(HaveLen(2))

Eventually(func() string {
    return thing.Status
}).ShouldNot(Equal("Stuck Waiting"))
```

`Consistently`

checks that an assertion passes for a period of time

```
Consistently(func() []int {
    return thing.MemoryUsage()
}).Should(BeNumerically("<", 10))
```

Timeout and Polling interval

- You can configure a polling internal and a timeout in both `Eventually` and `Consistently`

```
duration := time.Second()
pollingInterval := 100 * time.Millisecond()

Consistently(func() []int {
    return thing.MemoryUsage()
}, timeout, pollingInterval).Should(BeNumerically("<", 10))
```

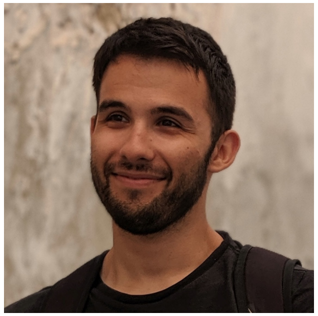
Other utilities

- **Gexec**: testing external processes https://onsi.github.io/gomega/#gexec-testing-external-processes
- **Ghttp**: testing HTTP clients https://onsi.github.io/gomega/#ghttp-testing-http-clients
- **Gbytes**: testing streaming buffers https://onsi.github.io/gomega/#gbytes-testing-streaming-buffers

To wrap up

- Ginkgo helps you structure your tests in a more descriptive way and run tests in a more convenient way.
- Gomega provides a set of matchers that can be combined.
- Together they can improve your development flow by making you more efficient.

Questions?



Twitter @jszroberto

Medium @totemteleko

More detailed notes in Notion.so