Testing in Go with Ginkgo and Gomega

Alexander Egurnov
Team Lead Backend, Verve Group Europe

About me

- Alexander Egurnov
- Team Lead Backend in Verve Group Europe
- Coding in Go since 2016

About the company

- Mobile AdTech
- Product
  - API Services
  - Go, MySQL, Kafka, Redis, Docker, K8s, CI/CD

Plan

- Testing in Go
  - What options are there?
  - How to choose?
- Ginkgo & Gomega overview
  - Specifics
  - Typical mistakes
  - Personal experience

For whom is this talk?

- Advanced Go coders
- First experience writing tests
- Growing project
- Eager to try something new

Questions

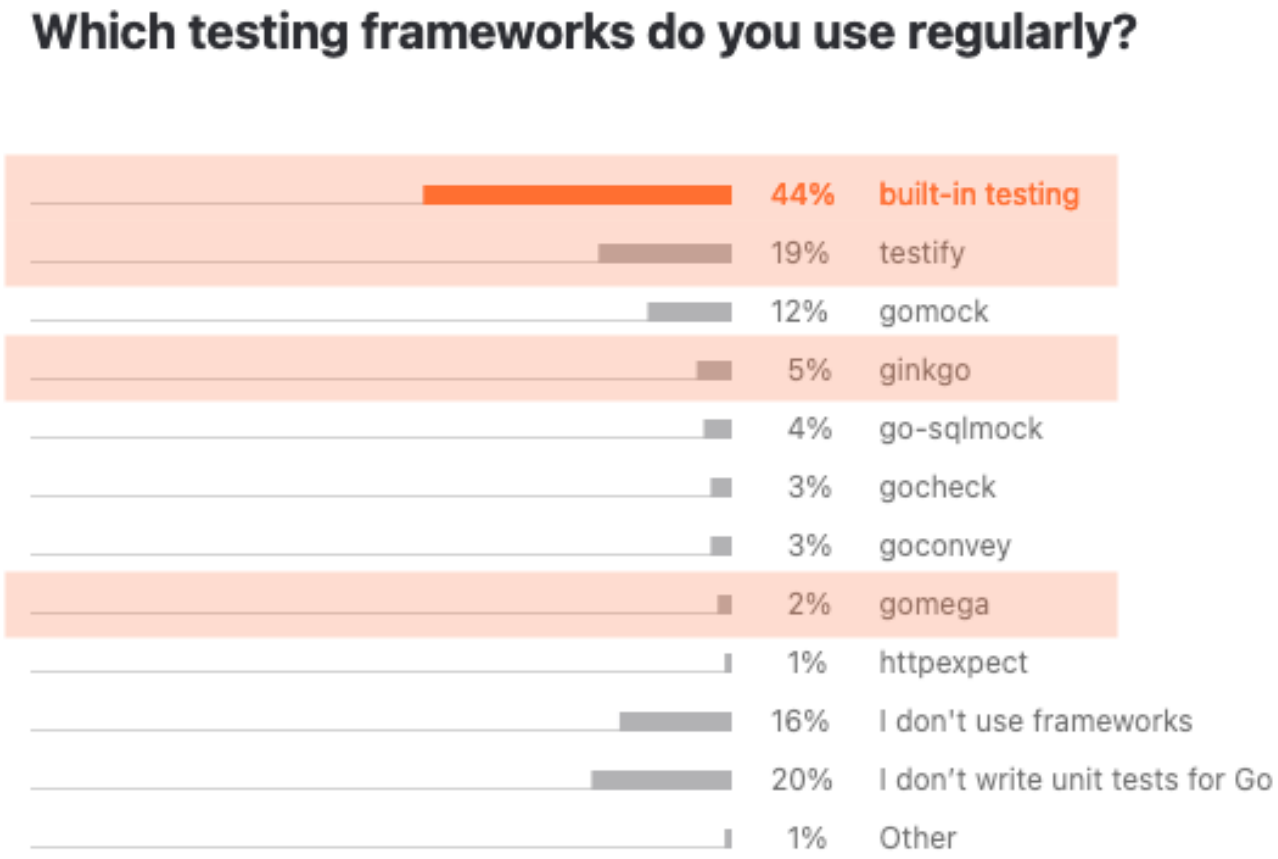
- How often do you write tests?
  - Never
  - Sometimes
  - Every day
- What problems do you encounter while writing tests?
  - None
  - It's not my job to write tests
  - Read-only tests: written one and never changed
  - Brittle tests: any code change breaks lots of tests
  - Robust tests: broken code doesn't break tests
  - slow tests
  - Other: send to chat

What options are there?

Other stack: Java/Python/etc

- Duplication
- Complexity
  - Development
  - CI/CD
- Extra skill requirements for the team

Popular options



Source: JetBrains - The State of Developer Ecosystem 2020 (https://www.jetbrains.com/lp/devecosystem-2020/go/)

testing

+ Standard
+ Simple
- Verbose

testing

```
func TestTesting(t *testing.T) {
        if testing.Short() {
                t.Skip("skipping test in short mode.")
        }

        prepare(t)

        for _, tt := range testCases {
                t.Run(tt.name, func(t *testing.T) {
                        got, err := doSomeStuff(tt.arg)

                        if tt.wantErr != (err != nil) {
                                t.Fatalf("failed: %v", err)
                        }
                        if !reflect.DeepEqual(got.A, tt.want.A) {
                                t.Errorf("expected = %q, want %q", got, tt.want)
                        }
                        validateB(t, tt.want.B)
                })
        }
}
```

testing

```
func validateB(t *testing.T, s string) {
        t.Helper()

        // do some checks
}

func prepare(t *testing.T) {
        // do some setup

        t.Cleanup(func() {
                // do some teardown
        })
}
```

testify

- assert
- require
- mock
- suite

testify

+ Complements testing
+ Rich API
+ Setup/teardown - suite
+ Mocking - mock
? Convenience?
- Limited extendability

testify

```
func TestTestify(t *testing.T) {
        if testing.Short() {
                t.Skip("skipping test in short mode.")
        }

        prepare(t)

        for _, tt := range testCases {
                t.Run(tt.name, func(t *testing.T) {
                        got, err := doSomeStuff(tt.arg)

                        require.NoError(t, err)
                        assert.Equal(t, got.A, tt.want.A)

                        validateB(t, tt.want.B)
                })
        }
}
```

Typical mistake

```
=== RUN   TestTestify
    testify_test.go:10:
                Error Trace:    testify_test.go:10
                Error:          Not equal:
                                expected: "actual"
                                actual  : "expected"
                Test:           TestTestify
--- FAIL: TestTestify (0.00s)
```

Ginkgo & Gomega

+ Structure
+ Setup / teardown
+ Rich DSL
+ Close to natural language
+ Extendable
- Complex

How to choose?

When?

- It becomes hard to
  - read
  - write
  - change
- Testing harness grows too large
  - setup / teardown
  - assertions, helpers, checkers, etc
  - API clients

Harness

Source: https://en.wikipedia.org/wiki/Climbing_harness

(https://en.wikipedia.org/wiki/Climbing_harness)

Source: https://en.wikipedia.org/wiki/Horse_harness

(https://en.wikipedia.org/wiki/Horse_harness)

Why?

- Simplify
  - reading
  - writing
- Remove duplicates
- Write tests, not harness
- Bonus: portability

DRY vs DAMP

- DRY - Don’t Repeat Yourself
- DAMP - Descriptive And Meaningful Phrases

Proverb

Source: Dave Cheney - GopherCon Singapore 2019 (https://dave.cheney.net/paste/clear-is-better-than-clever.pdf)

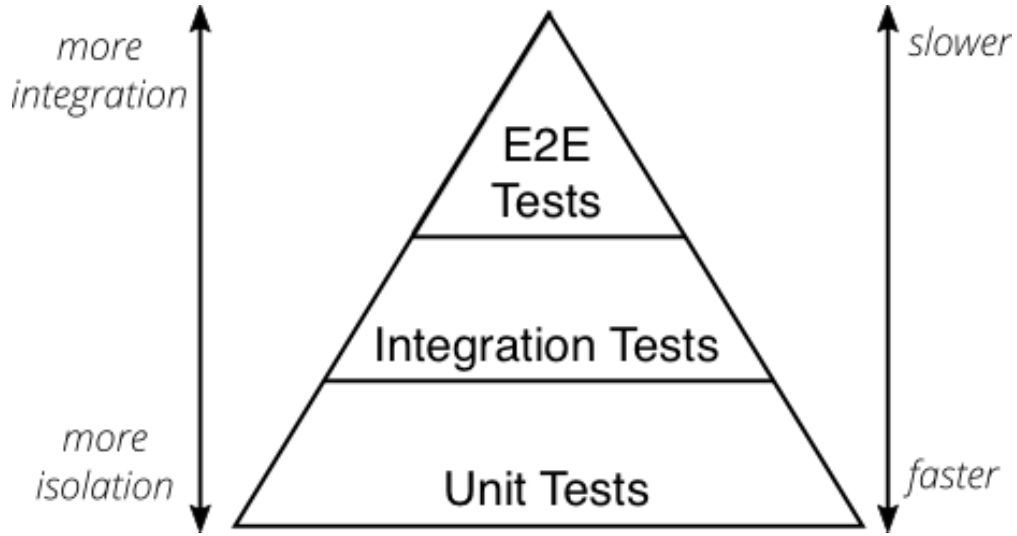
Proverb

Links:

- Clear is better than clever https://dave.cheney.net/2019/07/09/clear-is-better-than-clever (https://dave.cheney.net/2019/07/09/clear-is-better-than-clever)
- Go Proverbs https://go-proverbs.github.io/ (https://go-proverbs.github.io/)

Testing pyramid

- End2End - the whole product, blackbox
- Integration - several components
- Unit - one function or struct



Adapted from martinfowler.com - The Practical Test Pyramid (https://martinfowler.com/articles/practical-test-pyramid.html)

Ginkgo & Gomega

What is it?

- Ginkgo
  - Framework
    - dictates structure
    - controls your code
  - https://onsi.github.io/ginkgo/ (https://onsi.github.io/ginkgo/)
- Gomega
  - Assertion library
    - helps you make assertion
    - is controlled by your code
  - http://onsi.github.io/gomega/ (http://onsi.github.io/gomega/)

Specifics

Ginkgo

Testing support

```
func TestGinkgoTalk2021(t *testing.T) {
        log.SetOutput(GinkgoWriter)

        RegisterFailHandler(Fail)
        RunSpecs(t, "GinkgoTalk2021 Suite")
}
```

Testing tree

```
var _ = Describe("Test tree", func() {
        Context("Some case", func() {
                Specify("some general case", func() {
                        // Test 1
                })
                When("something happens", func() {
                        It("does this", func() {
                                // Test 2
                        })
                        Specify("longer test case", func() {
                                // Test 3
                                By("Step 1")
                                // ...
                                By("Step 2")
                                // ...
                        })
                })
        })
})
```

Testing tree

Two steps

- Build
- Execute

Warning: Closures!

Setup/Teardown

```
Context("Context 1", func() {
        BeforeEach(func() {})
        JustBeforeEach(func() {})
        JustAfterEach(func() {})
        AfterEach(func() {})

        Specify("test 1", func() {})

        Context("Context 2", func() {
                BeforeEach(func() {})
                JustBeforeEach(func() {})
                JustAfterEach(func() {})
                AfterEach(func() {})

                Specify("test 2", func() {})
        })
})
```

Setup/Teardown

```
Before Suite
---------------
BeforeEach 1
JustBeforeEach 1
Test 1
JustAfterEach 1
AfterEach 1
---------------
BeforeEach 1
BeforeEach 2
JustBeforeEach 1
JustBeforeEach 2
Test 2
JustAfterEach 2
JustAfterEach 1
AfterEach 2
AfterEach 1
---------------
After Suite
```

Focus/Skip/Pending

```
FIt("Focus", func() {
        // ...
})

PIt("Pending", func() {
        Fail("never happens")
})

XIt("Also pending", func() {
        Fail("never happens")
})

It("Skip", func() {
        if someCondition {
                Skip("nope")
        }
})
```

GinkgoWriter

```
func TestGinkgoTalk2021(t *testing.T) {
        log.SetOutput(GinkgoWriter)

        RegisterFailHandler(Fail)
        RunSpecs(t, "GinkgoTalk2021 Suite")
}
```

Gomega

Independent use

```
func TestGomegaStandalone(t *testing.T) {
        g := NewWithT(t)
        g.Expect(5).To(Equal(5))
}
```

Matchers

```
Expect(5).To(Equal(5))
Expect(5.0).To(BeEquivalentTo(5))
Expect(p1).To(BeIdenticalTo(p2))

Expect(err).ToNot(HaveOccurred())
Expect(someTask()).To(Succeed())
Expect(f).To(Panic())

Expect(5).To(BeNumerically("<", 5.1))
Expect(d1).To(BeTemporally("~", d2, 5*time.Minute))

Expect("Abracadabra").To(ContainSubstring("cad"))
Expect("x-y=z").To(ContainSubstring("%v-%v", "x", "y"))
Expect("me@example.com").To(MatchRegexp("[a-z]+@[a-z]+\\.[a-z]{2,}"))
Expect("{\"a\": 1, \"b\": 2}").To(MatchJSON("{\"b\": 2, \"a\": 1}"))
```

Matchers

```
Expect(ch).To(BeSent(7))
Expect(ch).To(Receive(&v))
Expect(ch).ToNot(BeClosed())

theSequence := []int{4, 8, 15, 16, 23, 42}
Expect(theSequence).ToNot(BeEmpty())
Expect(theSequence).To(HaveLen(6))
Expect(theSequence).To(ContainElement(23))
Expect(15).To(BeElementOf(theSequence))
Expect(theSequence).To(ConsistOf(8, 16, 42, 23, 15, 4))

shoppingList := map[string]int{"apples": 4, "tomatoes": 10, "milk": 1}
Expect(shoppingList).To(HaveKey("apples"))
Expect(shoppingList).To(HaveKeyWithValue("tomatoes", 10))
Expect(shoppingList).To(ConsistOf(1, 4, 10))
```

Matchers

```
Eventually(someCheck).Should(Succeed())
Consistently(someFunc).Should(BeNumerically(">", 3))
```

Matchers - Combining

```
Expect(5).To(
        And(
                BeNumerically(">", 4),
                BeNumerically("<", 6),
        ),
)

BeInRange := func(from, to interface{}) types.GomegaMatcher {
        return SatisfyAll(
                BeNumerically(">", from),
                BeNumerically("<", to),
        )
}
Expect(5).To(BeInRange(3, 6))

Expect([]int{4, 8, 15, 16, 23, 42}).To(
        ContainElements(
                BeInRange(10, 20),
                BeInRange(40, 50),
        ),
)
```

Matchers - Transforming

```
type T struct {
        name, id string
}

getName := func(t T) string {
        return t.name
}

withName := func(name string) types.GomegaMatcher {
        return WithTransform(getName, Equal(name))
}

arr := []T{{"a", "1"}, {"b", "2"}, {"c", "3"}}
Expect(arr).To(ContainElement(withName("a")))
```

Typical mistakes

Comparing heterogeneous values

```
var f float64 = 5
Expect(f).ToNot(Equal(5))

Expect(5.1).To(BeEquivalentTo(5))
Expect(5).ToNot(BeEquivalentTo(5.1))

Expect(f).To(BeNumerically("==", 5))
```

ContainElement / ContainSubstring

```
s := []byte("abracadabra")
Expect(s).ToNot(ContainElement("cad"))

Expect(s).To(ContainSubstring("cad"))
```

Eventually

```
start := time.Now()
isReady := func() bool {
        return time.Since(start) > 800*time.Millisecond
}

Eventually(isReady()).ShouldNot(BeTrue())

Eventually(isReady).Should(BeTrue())
```

Closures

```
var v = 5

It("uses v", func() {
        Expect(v).To(Equal(5))
})

It("uses v too", func() {
        v = 6
        Expect(v).To(Equal(6))
})
```

Closures

```
var v int

BeforeEach(func() {
        v = 5
})

It("uses v", func() {
        Expect(v).To(Equal(5))
})

It("uses v too", func() {
        v = 6
        Expect(v).To(Equal(6))
})
```

Loop variable

```
for i := 0; i < 5; i++ {
        Specify("test #"+strconv.Itoa(i), func() {
                log.Println("Running test #", i)
        })
}
```

```
2021/04/01 15:23:11 Running test # 5
2021/04/01 15:23:11 Running test # 5
2021/04/01 15:23:11 Running test # 5
2021/04/01 15:23:11 Running test # 5
2021/04/01 15:23:11 Running test # 5
```

Loop variable

```
for i := 0; i < 5; i++ {
        i := i
        Specify("test #"+strconv.Itoa(i), func() {
                log.Println("Running test #", i)
        })
}
```

```
2021/04/01 15:23:11 Running test # 0
2021/04/01 15:23:11 Running test # 1
2021/04/01 15:23:11 Running test # 2
2021/04/01 15:23:11 Running test # 3
2021/04/01 15:23:11 Running test # 4
```

GinkgoRecover

```
It("panics in a goroutine", func(done Done) {
        go func() {
                Fail("Oh noes!")
                close(done)
        }()
})
```

```
panic:
Your test failed.
Ginkgo panics to prevent subsequent assertions from running.
Normally Ginkgo rescues this panic so you shouldn't see it.

But, if you make an assertion in a goroutine, Ginkgo can't capture the panic.
To circumvent this, you should call

        defer GinkgoRecover()

at the top of the goroutine that caused this panic.
```

GinkgoRecover

```
It("fails in a goroutine", func(done Done) {
        go func() {
                defer GinkgoRecover()
                Fail("Oh noes!")
                close(done)
        }()
})
```

```
[Fail] FootShots Async functions Recover [It] fails in a goroutine too
```

Asynchronous tests

```
It("fails in a goroutine", func() {
        go func() {
                defer GinkgoRecover()
                time.Sleep(500 * time.Millisecond)
                Fail("Oh noes!")
        }()
})

It("doesn't do anything bad", func() {
        time.Sleep(800 * time.Millisecond)
})
```

```
[Fail] FootShots Async functions in a bad case [It] doesn't do anything bad
```

Asynchronous tests

```
It("fails in a goroutine", func(done Done) {
        go func() {
                defer GinkgoRecover()
                time.Sleep(500 * time.Millisecond)
                Fail("Oh noes!")
                close(done)
        }()
})

It("doesn't do anything bad", func() {
        time.Sleep(800 * time.Millisecond)
})
```

```
[Fail] FootShots Async functions when done properly [It] fails in a goroutine
```

Personal experience

Positive

- Automation
- Readability
- Simplicity
- Code reuse
- Portability

Negative

- Gotchas
- Complexity
- Readability >_<

Conclusion

- Write tests!
- Write different tests
- Pick you tools
- Start with a broken test
- Don't overcomplicate, seek balance

Links

- This talk
  - Code https://github.com/egurnov/ginkgo-talk-2021-pub/tree/english (https://github.com/egurnov/ginkgo-talk-2021-pub/tree/english)
- Previous talk
  - Code https://github.com/egurnov/ginkgo-talk-2020 (https://github.com/egurnov/ginkgo-talk-2020)
  - Video https://youtu.be/FaDx5GTlXNE (https://youtu.be/FaDx5GTlXNE)

Thank you

Alexander Egurnov
Team Lead Backend, Verve Group Europe
alexander.egurnov@gmail.com (mailto:alexander.egurnov@gmail.com)