Go Testing Bootcamp

Introduction

Federico
Paolinelli
fedepaol@gmail.com
fpaoline@redhat.com

- Telco Network Team @ Red Hat
- All things networking and kubernetes
- MetalLB maintainer

Francesco
Romani
fromani@gmail.com
fromani@redhat.com

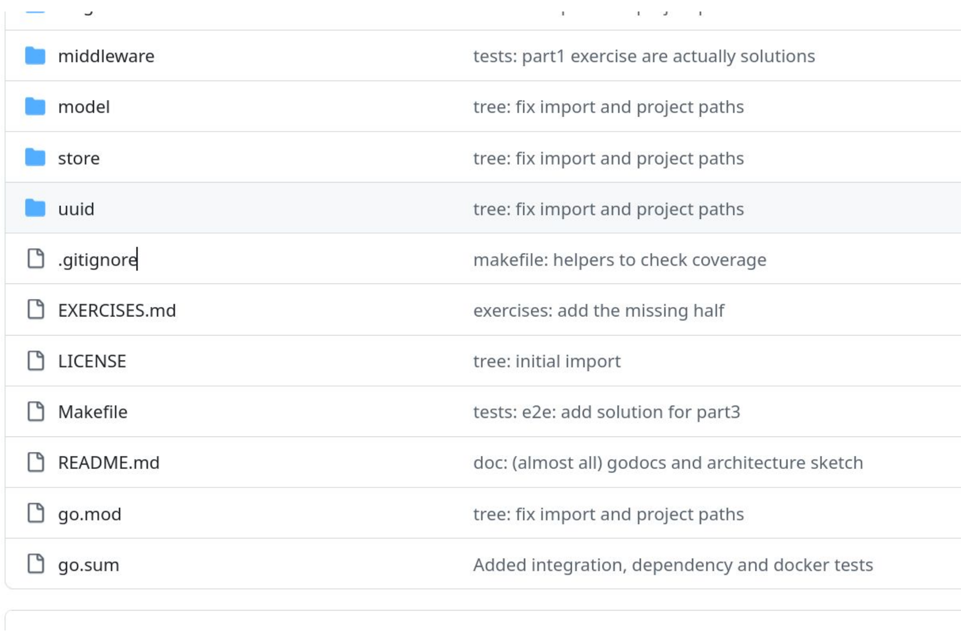
- Telco Compute Team @ Red Hat
- Kubernetes tuning/enhancement for low latency
- Kubernetes SIG-node reviewer

Workshop structure

- Topic introduction
- Practice
- Practice review

Workshop structure

- Topic introduction
- Practice
- Practice review



middleware	tests: part1 exercise are actually solutions
model	tree: fix import and project paths
store	tree: fix import and project paths
uuid	tree: fix import and project paths
.gitignore	makefile: helpers to check coverage
EXERCISES.md	exercises: add the missing half
LICENSE	tree: initial import
Makefile	tests: e2e: add solution for part3
README.md	doc: (almost all) godocs and architecture sketch
go.mod	tree: fix import and project paths
go.sum	Added integration, dependency and docker tests

tinyurl.com/gotodoapp

Tests: ~~Who~~, What, When, ~~Where~~, Why?

Why tests? A recap

Document, record and demonstrate behavior

- Tests against behavior, at all level
- Tests as documentation
- Tests as regression avoidance tool
- Tests enable refactorings
- Test to improve the code

What to test? start here

(Prefer) Test public interfaces

(with RARE exceptions)

Test public interfaces at unit level.

Test public interfaces at integration, system level.

Test public interfaces at end-to-end (e2e) level.

Some tests DON’Ts

As **general** guideline:

- DON’T test internal implementation details
- DON’T add test helpers
- DON’T add test-only mode

Some tests DOs

As general guideline:

- DO (re)organize the code to make it testable
- DO use coverage/usage metrics to spot untested areas
- DO focus on the error paths
- ...
- Do make exceptions **sparingly**

Type of tests: a walkthrough

The taxonomy we will adopt:

unit tests: test a single unit

integration tests: test how some modules of a larger system work together

end-to-end (e2e) tests: test a user flow

Some types of tests also depend on scope

Let’s consider a system composed of (micro)services

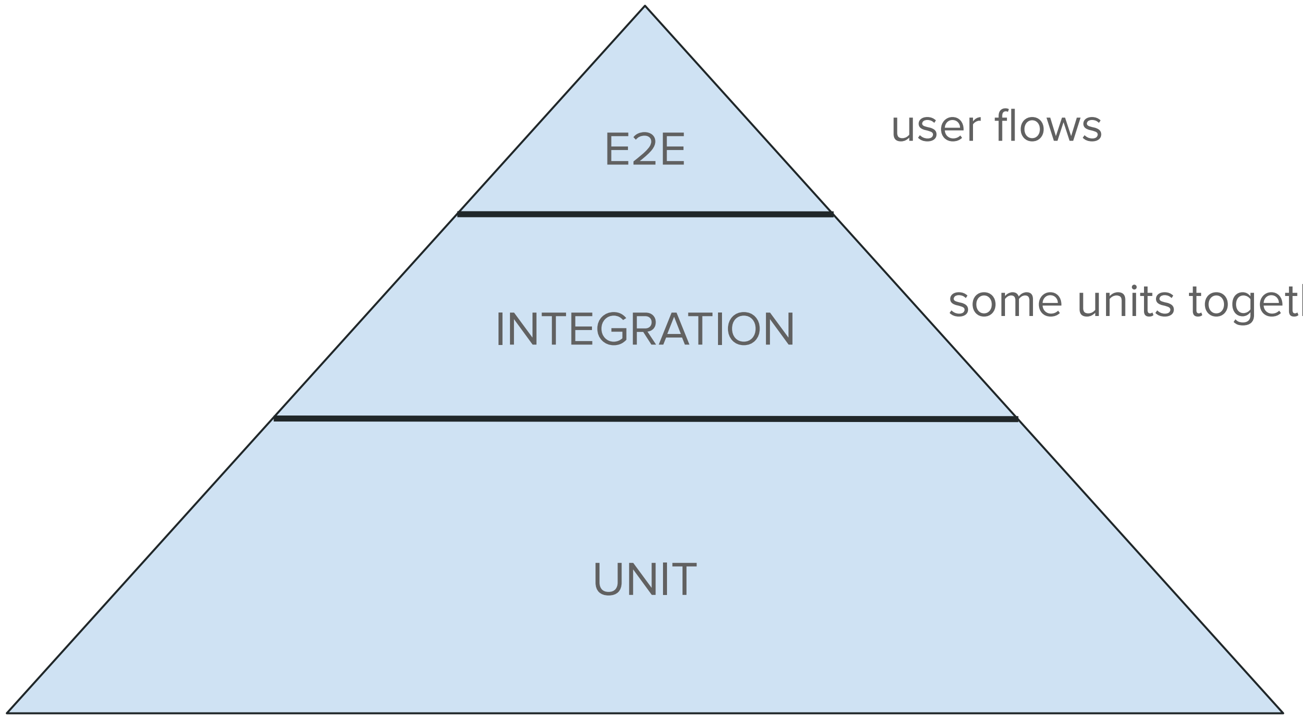
Testing a single service:

- can be seen as integration: service composed of modules
- can be seen as e2e: testing at service boundary

Yet the single service is a part of the larger system:
integration/e2e again

Testing pyramid(s)

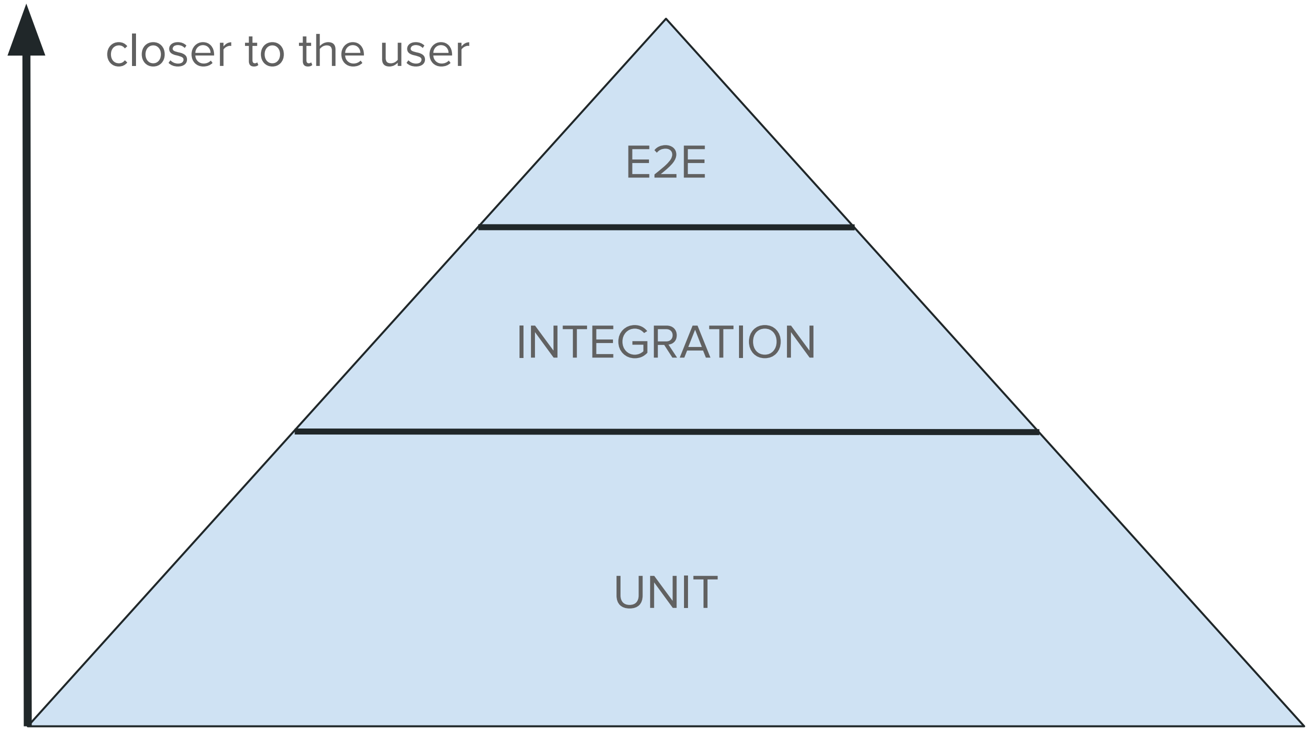
The testing pyramid: explained



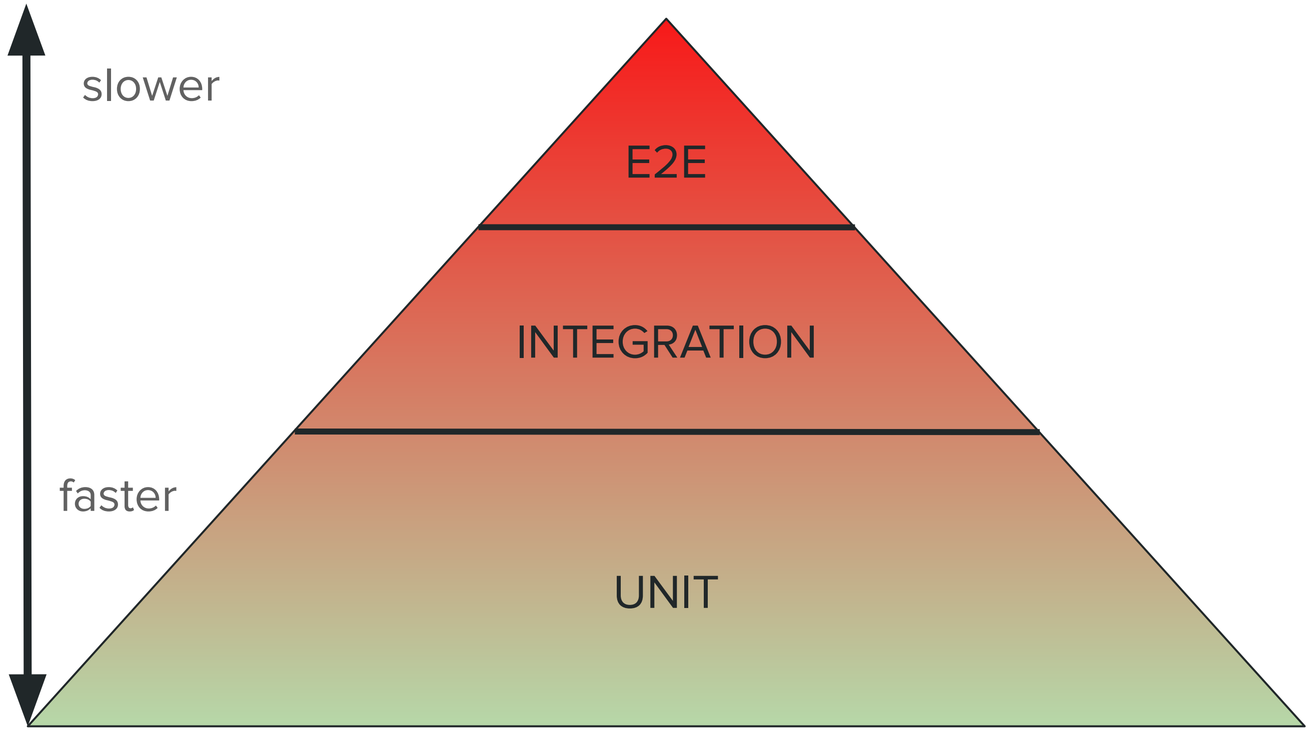
user flows

some units together

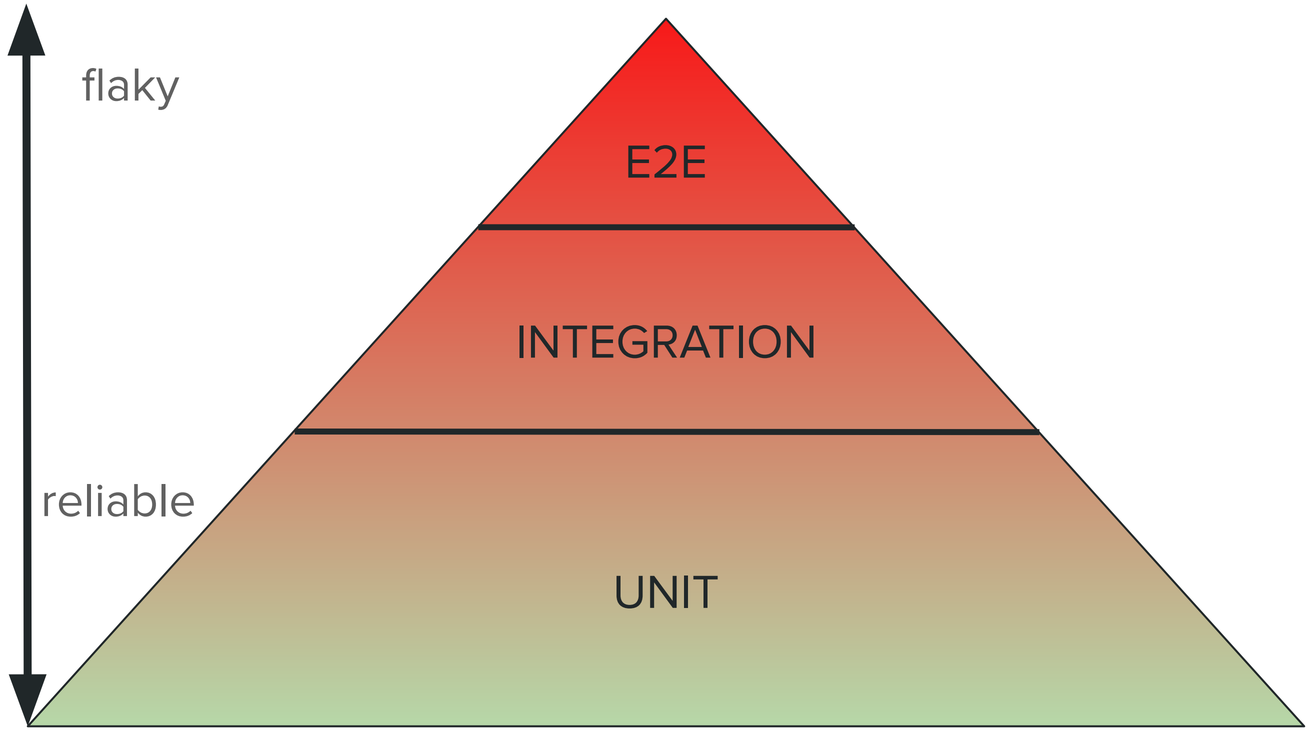
The testing pyramid



The testing pyramid: fast vs slow



The testing pyramid: reliable vs flaky



The testing package

Tests in golang: a recap/primer

```go
// filename: something_test.go

package something

func TestExample(t *testing.T) {
    // your test code here
}
```

- discover the tests
- (re)compiles the package(s)
- builds the test binary
- runs the test binary
- removes the test binary

```
$ go test .
```

A little more from the “go test” tool

```
# run a subset of tests:

$ go test -run ‘regexp’ .




# compile the test binary, don’t run it, save it for later

$ go test -c -o ./path/to/my.test .
$ ./path/to/my.test
$
$ ./path/to/my.test -help
```

Common traits of good tests

- change slower than the implementation
- test behavior (public API)
- isolate failures (failure as close as possible to the bug)
- reliable
    - no false negatives
    - **no false positives**

Test coverage

- helps finding the untested spots
- not a goal per se: 100% coverage doesn’t guarantee anything
    - can actually backfire: more maintenance burden
- how much is enough? YMMV

go testing coverage

```
# run tests with coverage
$ go test -cover .
PASS
coverage: 42.9% of statements
ok      size    0.026s

# coverage broken down by function
$ go tool cover -func=coverage.out .
size.go:    Size          42.9%
total:      (statements)  42.9%
```

Opens the default web browser and gives graphical representation of the coverage

```
$ go tool cover -html=coverage.out
```

Basics and coverage - Practice

Basics and coverage: practice

- write one or more unit test
  - hint: api/v1, config (basic), middleware (advanced)
- learn to check the coverage
- write one or more integration test
  - hint: ledger (basic), controller (advanced)
- verify how coverage changed
- EXTRA:
  - compile the test binary, save it, run it

Basics and coverage - Practice Review

Subtests

FP

Subtests

```
func (t *T) Run(name string, f func(t *T)) bool
```

```
func TestParent(t *testing.T) {
  t.Run("subtest", func(t *testing.T) {
   // subtest body
  })
}
```

Subtests

```
func (t *T) Run(name string, f func(t *T)) bool
```

> *Run runs f as a subtest of t called name. It runs f in a separate goroutine and blocks until f returns or calls t.Parallel to become a parallel test. Run reports whether f succeeded (or at least did not fail before calling t.Parallel).*
>
> *Run may be called simultaneously from multiple goroutines, but all such calls must return before the outer test function for t returns.*

Subtests

Group tests of the same category under the same umbrella for:

- Better control on what to run
- Enabling parallel execution
- Share common code among those tests

Subtests

```go
func TestSum(t *testing.T) {
  for i := 0; i < 5; i++ {
    name := fmt.Sprintf("with %d", i)
    t.Run(name, func(t *testing.T) {
      res := Sum(6, i)
      if 6+i != res {
        t.Errorf("Expected %d from %d, got %d", 6+i, i, res)
      }
    })
  }
}
```

Subtests

```go
func TestSum(t *testing.T) {
  for i := 0; i < 5; i++ {
    name := fmt.Sprintf("with %d", i)
    t.Run(name, func(t *testing.T) {
      res := Sum(6, i)
      if 6+i != res {
        t.Errorf("Expected %d from %d, got %d", 6+i, i, res)
      }
    })
  }
}
```

Subtests

```go
func TestSum(t *testing.T) {
  for i := 0; i < 5; i++ {
    name := fmt.Sprintf("with %d", i)
    t.Run(name, func(t *testing.T) {
      res := Sum(6, i)
      if 6+i != res {
        t.Errorf("Expected %d from %d, got %d", 6+i, i, res)
      }
    })
  }
}
```

Subtests: control over execution

```
go test -v -run TestSum
=== RUN   TestSum
=== RUN   TestSum/with_0
=== RUN   TestSum/with_1
=== RUN   TestSum/with_2
=== RUN   TestSum/with_3
=== RUN   TestSum/with_4
--- PASS: TestSum (2.50s)
    --- PASS: TestSum/with_0 (0.50s)
    --- PASS: TestSum/with_1 (0.50s)
    --- PASS: TestSum/with_2 (0.50s)
    --- PASS: TestSum/with_3 (0.50s)
    --- PASS: TestSum/with_4 (0.50s)
PASS
ok      github.com/fedepaol/section2    2.507s
```

```
go test -v -run TestSum/with_0
=== RUN   TestSum
=== RUN   TestSum/with_0
--- PASS: TestSum (0.50s)
    --- PASS: TestSum/with_0 (0.50s)
PASS
ok      github.com/fedepaol/section2    0.504s
```

Subtests: parallel execution

```go
func TestSum(t *testing.T) {
  for i := 0; i < 5; i++ {
    name := fmt.Sprintf("with %d", i)
    t.Run(name, func(t *testing.T) {
      res := Sum(6, i)
      if 6+i != res {
        t.Errorf("Expected %d from %d, got %d", 6+i, i, res)
      }
    })
  }
}
```

```
go test -run TestSum
PASS
ok      github.com/fedepaol/section2    2.509s
```

Subtests: parallel execution

```go
func TestSum(t *testing.T) {
  for i := 0; i < 5; i++ {
    name := fmt.Sprintf("with %d", i)
    t.Run(name, func(t *testing.T) {
      res := Sum(6, i)
      if 6+i != res {
        t.Errorf("Expected %d from %d, got %d", 6+i, i, res)
      }
    })
  }
}
```

```
go test -run TestSum
PASS
ok      github.com/fedepaol/section2    2.509s
```

Subtests: parallel execution

```go
func TestSum(t *testing.T) {
  for i := 0; i < 5; i++ {
    name := fmt.Sprintf("with %d", i)
    t.Run(name, func(t *testing.T) {
      t.Parallel()
      res := Sum(6, i)
      if 6+i != res {
        t.Errorf("Expected %d from %d, got %d", 6+i, i, res)
      }
    })
  }
}
```

```
go test -run TestSum
PASS
ok      github.com/fedepaol/section2    0.504s
```

Setup & Tear down

Setup & Tear Down

For all the tests of the same category:

- Prepare the scenario
- Run the tests
- Clean the scenario

Setup & Tear Down

```go
func TestCalculator(t *testing.T) {
        c := NewCalculator()
        t.Cleanup(func() {
                c.Unregister()
        })

        t.Run("sum 1+2", func(t *testing.T) {
                if c.Sum(1, 2) != 3 {
                        t.Fail()
                }
        })
        t.Run("sum 1+3", func(t *testing.T) {
                if c.Sum(1, 3) != 4 {
                        t.Fail()
                }
        })
}
```

Setup & Tear Down

```
func TestCalculator(t *testing.T) {
        c := NewCalculator()
        t.Cleanup(func() {
                c.Unregister()
        })

        t.Run("sum 1+2", func(t *testing.T) {
                if c.Sum(1, 2) != 3 {
                        t.Fail()
                }
        })
        t.Run("sum 1+3", func(t *testing.T) {
                if c.Sum(1, 3) != 4 {
                        t.Fail()
                }
        })
}
```

Setup & Tear Down

```go
func TestCalculator(t *testing.T) {
        c := NewCalculator()
        t.Cleanup(func() {
                c.Unregister()
        })

        t.Run("sum 1+2", func(t *testing.T) {
                if c.Sum(1, 2) != 3 {
                        t.Fail()
                }
        })
        t.Run("sum 1+3", func(t *testing.T) {
                if c.Sum(1, 3) != 4 {
                        t.Fail()
                }
        })
}
```

Setup & Tear Down

```
func TestCalculator(t *testing.T) {
        c := NewCalculator()
        t.Cleanup(func() {
                c.Unregister()
        })

        t.Run("sum 1+2", func(t *testing.T) {
                if c.Sum(1, 2) != 3 {
                        t.Fail()
                }
        })
        t.Run("sum 1+3", func(t *testing.T) {
                if c.Sum(1, 3) != 4 {
                        t.Fail()
                }
        })
}
```

Setup / Teardown: TestMain

- lower level
- one per package
- useful when we have a global setup / teardown shared with all the tests

```go
func TestMain(m *testing.M) {
    db.Setup()
    code := m.Run()
    db.Close()
    os.Exit(code)
}
```

Table Tests

Table Tests

```go
func TestCalculator(t *testing.T) {
        c := NewCalculator()
        t.Cleanup(func() {
                c.Unregister()
        })

        t.Run("sum 1+2", func(t *testing.T) {
                if c.Sum(1, 2) != 3 {
                        t.Fail()
                }
        })
        t.Run("sum 1+3", func(t *testing.T) {
                if c.Sum(1, 3) != 4 {
                        t.Fail()
                }
        })
}
```

Table Tests

```go
func TestCalculatorTable(t *testing.T) {
  tests := []struct {
    name     string
    first    int
    second   int
    expected int
  }{
    {"1+2", 1, 2, 3},
  }

  c := NewCalculator()
  t.Cleanup(func() {
    c.Unregister()
  })

  for _, tc := range tests {
    t.Run(tc.name, func(t *testing.T) {
      if c.Sum(tc.first, tc.second) != tc.expected {
        t.Fail()
      }
    })
  }
}
```

Table Tests

```go
func TestCalculatorTable(t *testing.T) {
  tests := []struct {
    name     string
    first    int
    second   int
    expected int
  }{
    {"1+2", 1, 2, 3},
  }

  c := NewCalculator()
  t.Cleanup(func() {
    c.Unregister()
  })

  for _, tc := range tests {
    t.Run(tc.name, func(t *testing.T) {
      if c.Sum(tc.first, tc.second) != tc.expected {
        t.Fail()
      }
    })
  }
}
```

Table Tests

```go
func TestCalculatorTable(t *testing.T) {
  tests := []struct {
    name     string
    first    int
    second   int
    expected int
  }{
    {"1+2", 1, 2, 3},
  }

  c := NewCalculator()
  t.Cleanup(func() {
    c.Unregister()
  })

  for _, tc := range tests {
    t.Run(tc.name, func(t *testing.T) {
      if c.Sum(tc.first, tc.second) != tc.expected {
        t.Fail()
      }
    })
  }
}
```

Table Tests

```go
func TestCalculatorTable(t *testing.T) {
  tests := []struct {
    name     string
    first    int
    second   int
    expected int
  }{
    {"1+2", 1, 2, 3},
    {"1+3", 1, 3, 4},
  }

  c := NewCalculator()
  t.Cleanup(func() {
    c.Unregister()
  })

  for _, tc := range tests {
    t.Run(tc.name, func(t *testing.T) {
      if c.Sum(tc.first, tc.second) != tc.expected {
        t.Fail()
      }
    })
  }
}
```

Subtests / Table tests - Practice

Subtests - Practice Review

Test Fixtures

Test Fixtures

- Sometimes we need some artifact to run our tests against:
  - files to parse
  - images
  - db content
- The content of testdata is ignored at compile time
- when running go test, the current folder matches the test file

Test Fixtures

```go
func Parse(fileName string) (User, error)
```

```
$ tree
.
├── parse.go
├── parse_test.go
└── testdata
    └── basic.json
```

```go
tests := []struct {
        fileName     string
        expected     User
        expectsError bool
}{
        {
                "testdata/basic.json",
                User{"foo", 12},
                false,
        },
```

Test Fixtures

```go
func Parse(fileName string) (User, error)
```

```
$ tree
.
├── parse.go
├── parse_test.go
└── testdata
    └── basic.json
```

```go
tests := []struct {
        fileName      string
        expected      User
        expectsError bool
}{
        {
                "testdata/basic.json",
                User{"foo", 12},
                false,
        },
```

Test Fixtures

```go
func Parse(fileName string) (User, error)
```

```
$ tree

.
├── parse.go
├── parse_test.go
└── testdata
    └── basic.json
```

```go
tests := []struct {
          fileName    string
          expected    User
          expectsError bool
}{
          {
                    "testdata/basic.json",
                    User{"foo", 12},
                    false,
          },
```

Test Fixtures

```go
func Parse(fileName string) (User, error)
```

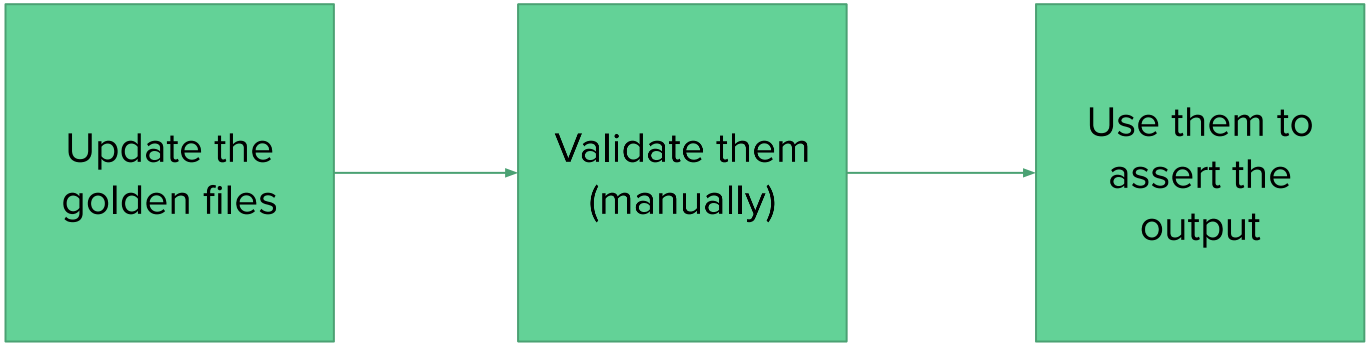
```go
t.Run(tc.fileName, func(t *testing.T) {
      res, err := Parse(tc.fileName)
      if err == nil && tc.expectsError {
        t.Errorf("expecting error, got success")
      }
      if err != nil && !tc.expectsError {
        t.Errorf("not expecting error, got %v", err)
      }
      if !tc.expectsError && res != tc.expected {
        t.Errorf("expecting %v, got %v", tc.expected, res)
      }
    })
```

Golden files

Golden files

- Asserting a generated output is tedious
- Especially in case of generated files / rendered items
    - Template -> configuration file
    - Template -> html page
    - Json output
- A golden file becomes the source of truth for your test result

Golden files



Golden files

```
var update = flag.Bool("update", false, "update .golden.json files")


t.Run(tc.fileName, func(t *testing.T) {
      res, _ := ParseAndIncrementAge(tc.fileName)
      jsonRes, _ := json.Marshal(res)

      goldenFile := tc.fileName + ".golden"
      if *update {
        os.WriteFile(goldenFile, jsonRes, os.ModePerm)
      }
      expected, err := os.ReadFile(goldenFile)
      if err != nil {
        t.Errorf("failed to open golden file %s: %v", goldenFile, err)
      }
      if !bytes.Equal(expected, jsonRes) {
        t.Fail()
      }
    })
```

Golden files

```
var update = flag.Bool("update", false, "update .golden.json files")
```

```
t.Run(tc.fileName, func(t *testing.T) {
      res, _ := ParseAndIncrementAge(tc.fileName)
      jsonRes, _ := json.Marshal(res)

      goldenFile := tc.fileName + ".golden"
      if *update {
        os.WriteFile(goldenFile, jsonRes, os.ModePerm)
      }
      expected, err := os.ReadFile(goldenFile)
      if err != nil {
        t.Errorf("failed to open golden file %s: %v", goldenFile, err)
      }
      if !bytes.Equal(expected, jsonRes) {
        t.Fail()
      }
    })
```

Golden files

```
var update = flag.Bool("update", false, "update .golden.json files")


t.Run(tc.fileName, func(t *testing.T) {
      res, _ := ParseAndIncrementAge(tc.fileName)
      jsonRes, _ := json.Marshal(res)

      goldenFile := tc.fileName + ".golden"
      if *update {
        os.WriteFile(goldenFile, jsonRes, os.ModePerm)
      }
      expected, err := os.ReadFile(goldenFile)
      if err != nil {
        t.Errorf("failed to open golden file %s: %v", goldenFile, err)
      }
      if !bytes.Equal(expected, jsonRes) {
        t.Fail()
      }
    })
```

Golden files

```go
var update = flag.Bool("update", false, "update .golden.json files")


t.Run(tc.fileName, func(t *testing.T) {
      res, _ := ParseAndIncrementAge(tc.fileName)
      jsonRes, _ := json.Marshal(res)

      goldenFile := tc.fileName + ".golden"
      if *update {
        os.WriteFile(goldenFile, jsonRes, os.ModePerm)
      }
      expected, err := os.ReadFile(goldenFile)
      if err != nil {
        t.Errorf("failed to open golden file %s: %v", goldenFile, err)
      }
      if !bytes.Equal(expected, jsonRes) {
        t.Fail()
      }
    })
```

Golden files

```go
var update = flag.Bool("update", false, "update .golden.json files")


t.Run(tc.fileName, func(t *testing.T) {
      res, _ := ParseAndIncrementAge(tc.fileName)
      jsonRes, _ := json.Marshal(res)

      goldenFile := tc.fileName + ".golden"
      if *update {
        os.WriteFile(goldenFile, jsonRes, os.ModePerm)
      }
      expected, err := os.ReadFile(goldenFile)
      if err != nil {
        t.Errorf("failed to open golden file %s: %v", goldenFile, err)
      }
      if !bytes.Equal(expected, jsonRes) {
        t.Fail()
      }
    })
```

Golden files

```
go test
--- FAIL: TestParseAndIncrement (0.00s)
    --- FAIL: TestParseAndIncrement/testdata/basic.json (0.00s)
        parse_test.go:67: failed to open golden file testdata/basic.json.golden: open
testdata/basic.json.golden: no such file or directory
FAIL
```

Golden files

```
go test
--- FAIL: TestParseAndIncrement (0.00s)
    --- FAIL: TestParseAndIncrement/testdata/basic.json (0.00s)
        parse_test.go:67: failed to open golden file testdata/basic.json.golden: open
testdata/basic.json.golden: no such file or directory
FAIL
```

```
go test -update
PASS
ok      github.com/fedepaol/fixturegolden       0.007s
```

Golden files

```
go test
--- FAIL: TestParseAndIncrement (0.00s)
    --- FAIL: TestParseAndIncrement/testdata/basic.json (0.00s)
        parse_test.go:67: failed to open golden file testdata/basic.json.golden: open
testdata/basic.json.golden: no such file or directory
FAIL
```

```
go test -update
PASS
ok      github.com/fedepaol/fixturegolden       0.007s
```

```
go test
PASS
ok      github.com/fedepaol/fixturegolden       0.007s
```

Golden files

```
go test
--- FAIL: TestParseAndIncrement (0.00s)
    --

testda
FAIL



go tes
PASS
ok
```

```
cat testdata/basic.json.golden
{"name":"foo","age":13}
```

```
go test
PASS
ok      github.com/fedepaol/fixturegolden       0.007s
```

Test fixtures + Golden files practice

Test fixtures + Golden files practice Review

Integration and beyond: gingko/gomega

Ginkgo and gomega

Ginkgo is a powerful testing framework for go

Gomega is a library which adds matching capabilities to ginkgo

(BeTrue, IsNil...)

Ginkgo and Gomega together provide a Domain-Specific Language (DSL) to write tests in go

Ginkgo and gomega vs testing

Ginkgo and gomega augments testing, don’t replace it

functionally ginkgo “specs” == “tests”

The different name is used to distinguish between ginkgo tests and standard go tests

The ginkgo use case: end-to-end tests

ginkgo is **best suited** for integration or end-to-end (e2e) tests

E2E testing is an approach to testing that that simulates real user flows.

Why ginkgo?

- emphasis on behavior
- descriptive tests (specs)
- good support for asynchronous tests: Eventually, Consistently

The ginkgo use case: asynchronous tests

```
Eventually(X).WithTimeout(T).WithPolling(P).WithContext(ctx).Should(MATCHER)
```

Checks an assertion passes *eventually*

- tries polling every **P** time units
- until the timeout **T** expires
- optionally with a context

The ginkgo use case: asynchronous tests /2

```
Eventually(ACTUAL).MustPassRepeatedly(R).Should(MATCHER)
```

Checks an assertion passes *eventually*

And then passes **R consecutive times**

The ginkgo use case: asynchronous tests /3

```
Consistently(ACTUAL).WithTimeout(T).WithPolling(P).WithContext(ctx).Should(MATCHER)
```

Checks that an assertion passes for a period of time

- tries polling every **P** time units
- until the timeout **T** expires
- optionally with a context

Bootstrapping a ginkgo suite

```
# outside GOPATH
go install github.com/onsi/ginkgo/v2/ginkgo@latest

go get github.com/onsi/gomega/...

ginkgo bootstrap
# will create package_suite_test.go
# entry point, scaffolding
```

Gingko suite breakdown for example package foobar

```
package $\boxed{foobar_test}$

import (
  . "github.com/onsi/ginkgo/v2"
  . "github.com/onsi/gomega"
  "testing"
)

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

Gingko suite breakdown for example package foobar

```
package foobar_test

import (
  . "github.com/onsi/ginkgo/v2"
  . "github.com/onsi/gomega"
  "testing"
)

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

Gingko suite breakdown for example package foobar

```
package foobar_test

import (
  . "github.com/onsi/ginkgo/v2"
  . "github.com/onsi/gomega"
  "testing"
)

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

Gingko suite breakdown for example package foobar

```
package foobar_test

import (
  . "github.com/onsi/ginkgo/v2"
  . "github.com/onsi/gomega"
  "testing"
)

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

Gingko suite breakdown for example package foobar

```
package foobar_test

import (
  . "github.com/onsi/ginkgo/v2"
  . "github.com/onsi/gomega"
  "testing"
)

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

Adding specs

alias ginkgo specs == tests

```
ginkgo bootstrap
# will create package_suite_test.go
# entry point, scaffolding

ginkgo generate foo
# will create foo_test.go
# or just create them manually
```

A ginkgo spec breakdown

```
package foobar_test

import (
    . "github.com/onsi/ginkgo/v2"
    . "github.com/onsi/gomega"

   "path/to/foobar"
)

var _ = Describe("Some foo cases", func() {
        ...
})
```

A ginkgo spec breakdown

```
package foobar_test

import (
  . "github.com/onsi/ginkgo/v2"
  . "github.com/onsi/gomega"

  "path/to/foobar"
)

var _ = Describe("Some foo cases", func() {
     ...
})
```

A ginkgo spec breakdown

```
package foobar_test

import (
  . "github.com/onsi/ginkgo/v2"
  . "github.com/onsi/gomega"

  "path/to/foobar"
)

var _ = Describe("Some foo cases", func() {
     ...
})
```

Gingko suite zoom in

```
var _ = Describe("Some foo cases", func() {
     BeforeEach(func() {
          initialize()
     })
     Context("With some conditions", func() {
          It("should behave like this", func() {
               Expect(foo.Something()).To(Equal(somethingElse))
          })
     })
     When("some other conditions apply", func() {
          It("should behave like that", func() {
               Expect(bar).Should(BeTrue())
               Expect(baz).ToNot(BeNil())
          })
     })
})
```

Gingko suite zoom in

```
var _ = Describe("Some foo cases", func() {
        BeforeEach(func() {
                initialize()
        })
        Context("With some conditions", func() {
                It("should behave like this", func() {
                        Expect(foo.Something()).To(Equal(somethingElse))
                })
        })
        When("some other conditions apply", func() {
                It("should behave like that", func() {
                        Expect(bar).Should(BeTrue())
                        Expect(baz).ToNot(BeNil())
                })
        })
})
```

Gingko suite zoom in

```
var _ = Describe("Some foo cases", func() {
      BeforeEach(func() {
            initialize()
      })
      Context("With some conditions", func() {
            It("should behave like this", func() {
                  Expect(foo.Something()).To(Equal(somethingElse))
            })
      })
      When("some other conditions apply", func() {
            It("should behave like that", func() {
                  Expect(bar).Should(BeTrue())
                  Expect(baz).ToNot(BeNil())
            })
      })
})
```

Gingko suite zoom in

```
var _ = Describe("Some foo cases", func() {
        BeforeEach(func() {
                initialize()
        })
        Context("With some conditions", func() {
                It("should behave like this", func() {
                        Expect(foo.Something()).To(Equal(somethingElse))
                })
        })
        When("some other conditions apply", func() {
                It("should behave like that", func() {
                        Expect(bar).Should(BeTrue())
                        Expect(baz).ToNot(BeNil())
                })
        })
})
```

Gingko suite zoom in

```
var _ = Describe("Some foo cases", func() {
      BeforeEach(func() {
            initialize()
      })
      Context("With some conditions", func() {
            It("should behave like this", func() {
                  Expect(foo.Something()).To(Equal(somethingElse))
            })
      })
      When("some other conditions apply", func() {
            It("should behave like that", func() {
                  Expect(bar).Should(BeTrue())
                  Expect(baz).ToNot(BeNil())
            })
      })
})
```

Gingko suite zoom in

```
var _ = Describe("Some foo cases", func() {
      BeforeEach(func() {
            initialize()
      })
      Context("With some conditions", func() {
            It("should behave like this", func() {
                  Expect(foo.Something()).To(Equal(somethingElse))
            })
      })
      When("some other conditions apply", func() {
            It("should behave like that", func() {
                  Expect(bar).Should(BeTrue())
                  Expect(baz).ToNot(BeNil())
            })
      })
})
```

Ginkgo example output

```
Running Suite: E2E Suite - /go/src/github.com/ffromani/go-todo-app/e2e
==================================================================================
Random Seed: 1730484296

Will run 1 of 1 specs
------------------------------
backlog endpoint when todos are added should return them
/go/src/github.com/ffromani/go-todo-app/e2e/backlog_test.go:26
• [0.893 seconds]
------------------------------


Ran 1 of 1 Specs in 0.894 seconds
SUCCESS! -- 1 Passed | 0 Failed | 0 Pending | 0 Skipped
PASS

Ginkgo ran 1 suite in 1.314289024s
Test Suite Passed
```

Ginkgo example output

```
Running Suite: E2E Suite - /go/src/github.com/ffromani/go-todo-app/e2e
====================================================================
Random Seed: 1730484296

Will run 1 of 1 specs
------------------------------
backlog endpoint when todos are added should return them
/go/src/github.com/ffromani/go-todo-app/e2e/backlog_test.go:26
• [0.893 seconds]
------------------------------

Ran 1 of 1 Specs in 0.894 seconds
SUCCESS! -- 1 Passed | 0 Failed | 0 Pending | 0 Skipped
PASS

Ginkgo ran 1 suite in 1.314289024s
Test Suite Passed
```

Ginkgo example output

```
Running Suite: E2E Suite - /go/src/github.com/ffromani/go-todo-app/e2e
==========================================================================
Random Seed: 1730484296

Will run 1 of 1 specs

backlog endpoint when todos are added should return them
/go/src/github.com/ffromani/go-todo-app/e2e/backlog_test.go:26
• [0.893 seconds]
------------------------------

Ran 1 of 1 Specs in 0.894 seconds
SUCCESS! -- 1 Passed | 0 Failed | 0 Pending | 0 Skipped
PASS

Ginkgo ran 1 suite in 1.314289024s
Test Suite Passed
```

Ginkgo example output

```
Running Suite: E2E Suite - /go/src/github.com/ffromani/go-todo-app/e2e
==========================================================================
Random Seed: 1730484296

Will run 1 of 1 specs
------------------------------
backlog endpoint when todos are added should return them
/go/src/github.com/ffromani/go-todo-app/e2e/backlog_test.go:26
• [0.893 seconds]
------------------------------

Ran 1 of 1 Specs in 0.894 seconds
SUCCESS! -- 1 Passed | 0 Failed | 0 Pending | 0 Skipped
PASS

Ginkgo ran 1 suite in 1.314289024s
Test Suite Passed
```

How ginkgo runs

ginkgo specs must be independent

gingko specs runs in random order and by default in parallel

“declare in container nodes, initialize in setup nodes”

How ginkgo runs: walking the tree

Ginkgo runs in two steps: tree construction and run phase

Tree Construction:

- Ginkgo visits all container nodes, invokes their closures and constructs the spec tree.
- Ginkgo captures the relevant setup and subject node closures by visiting the tree, but **does not run them**.

How ginkgo runs: running the tree

Ginkgo runs in two steps: tree construction and run phase

Run phase:

- Ginkgo runs through each spec in the generated spec list sequentially.
- Ginkgo invokes the setup and subject nodes closures in the correct order and tracks any failed assertions, for each spec.
- Container node closures are never invoked.

Common gotchas

- All Ginkgo nodes must only appear at the top-level or within a container node.
- A subject node cannot be top level

Note: you **CAN** nest arbitrarily container nodes though!

Note: you **CAN** have multiple top-level container nodes!

Common gotchas /2

No assertion in container nodes! (ginkgo.Expect() ...)

Note: you **CAN** have any amount of assertions in a subject node!

Common gotchas /3

Do not initialize variables in container nodes

Subject nodes can mutate the values and pollute the state!

Perform initialization in setup nodes: these nodes are guaranteed to be called before every relevant subject node

Note: kinda OK for constants though - but should those be container variables?

Logging: GinkgoWriter

GinkgoWriter is a globally available io.Writer.

Aggregates everything, only emits to stdout if the test fails.

GinkgoWriter.TeeTo(writer): attach additional writers. Any data written to GinkgoWriter will immediately be sent to attached tee writers.

In verbose mode (ginkgo -v) writes to GinkgoWriter are immediately sent to stdout.

Logging: By() clause

By(“my message”)

Display the messages on failure

In verbose mode, displays the steps immediately

Focus

```
var _ = Describe("Some foo cases", func() {
      It ("should test something", Focus, func() {
      })
})

var _ = Describe("Some bar cases", Focus, func() {
      It ("should test something else", func() {
      })
      It ("should test something else more", func() {
      })
})
```

OR

```
gingko —focus=REGEXP
```

Labels

```
var _ = Describe("Some foo cases", func() {
      It ("should test something", Label("Label_A"), func() {
      })
})

var _ = Describe("Some bar cases", Label("label_B"), func() {
      It ("should test something else", func() {
      })
      It ("should test something else more", func() {
      })
})
```

```
gingko -label-filter=FILTER
```

Ginkgo custom matchers

Custom matchers

Add higher level, domain specific matchers

Make the tests more expressive

```
type GomegaMatcher interface {
    Match(actual interface{}) (success bool, err error)
    FailureMessage(actual interface{}) (message string)
    NegatedFailureMessage(actual interface{}) (message string)
}
```

Gomega matcher interface explained

```
Match(actual interface{}) (success bool, err error)
```

Returns non-nil error is given invalid input.
You can use concrete types! (see examples)
If the actual value matches, returns true; otherwise, returns false.

Gomega matcher interface explained /2

```
FailureMessage(actual interface{}) (message string)
NegatedFailureMessage(actual interface{}) (message string)
```

Only after Match() failed: if Should/To block was called, call FailureMessage() to get the error message; otherwise if a ShouldNot/ToNot block was called, call NegatedFailureMessage

Why custom matchers? we have builtin matchers

```
Expect(Todo.Title).ToNot(BeEmpty())
```

```
Expect(Todo.Assignee).Equal("John Doe")
Expect(Todo.Status).Equal(todov1.Assigned)
```

- straightforward to write
- low level
- complex conditions are lost

A custom matcher

```
import (
    "github.com/onsi/gomega/gcustom"
    "github.com/onsi/gomega/types"
)

func BeValid() types.GomegaMatcher {
    return gcustom.MakeMatcher(func(actual todo.Todo) (bool, error) {
        return actual.Title != “”, nil
    }).WithTemplate("Todo must have a title to be valid")
}
```

A custom matcher explained

```
func BeValid() types.GomegaMatcher {
     return gcustom.MakeMatcher(
          func(actual todo.Todo) (bool, error) {
               return actual.Title != “”, nil
          }
     ).WithTemplate("Todo must have a title to be valid")
}
```

A custom matcher explained

```
func BeValid() types.GomegaMatcher {
     return gcustom.MakeMatcher(
          func(actual todo.Todo) (bool, error) {
               return actual.Title != “”, nil
          }
     ).WithTemplate("Todo must have a title to be valid")
}
```

A custom matcher explained

```
func BeValid() types.GomegaMatcher {
     return gcustom.MakeMatcher(
          func(actual todo.Todo) (bool, error) {
               return actual.Title != “”, nil
          }
     ).WithTemplate("Todo {{.Actual.Title}} must be assigned to {{.Data}}"
}
```

A custom matcher explained

```
func BeValid() types.GomegaMatcher {
    return gcustom.MakeMatcher(
        func(actual todo.Todo) (bool, error) {
            return actual.Title != “”, nil
        }
    ).WithTemplate("Todo must have a title to be valid")
}
```

Another custom matcher

```go
import (
	"github.com/onsi/gomega/gcustom"
	"github.com/onsi/gomega/types"
)

func IsAssignedTo(assignee string) types.GomegaMatcher {
	return gcustom.MakeMatcher(func(actual todo.Todo) (bool, error) {
		return
			actual.Status == apiv1.Assigned &&
			actual.Assignee == assignee
		), nil
	}
	).WithTemplate("Todo {{.Actual.Title}} must be assigned to {{.Data}}"
	).WithTemplateData(assignee)
}
```

Why custom matchers? we have builtin matchers

```
Expect(Todo.Title).ToNot(BeEmpty())

Expect(Todo.Assignee).Equal(“John Doe”)
Expect(Todo.Status).Equal(todov1.Assigned)
```

- straightforward to write
- low level
- complex conditions are lost

```
Expect(Todo).To(BeValid())

Expect(Todo).Should(BeAssignedTo(“John Doe”))
```

- still straightforward to write!
- captures the intent!

Using ginkgo - Practice

Using ginkgo - Practice Review

Testing when we have dependencies

Unit tests must be consistent, reproducible and fast

Dependencies

The behaviour the unit being tested depends on something external:

- The content of a database
- An external service
- Over the network
- That we pay for!

Option 1: Remove the dependency and make our function testable

```go
func ParseWithReader(fileName string) (User, error) {
        res := User{}

        f, err := os.Open(fileName)
        if err != nil {
                return User{}, err
        }
        defer f.Close()

        err = json.NewDecoder(f).Decode(&res)
        if err != nil {
                return res, err
        }
        return res, nil

}
```

Option 1: Remove the dependency and make our function testable

Our
Dependency

```go
func ParseWithReader(                    (User, error) {
        res := User{}

        f, err := os.Open(fileName)
        if err != nil {
                return User{}, err
        }
        defer f.Close()

        err = json.NewDecoder(f).Decode(&res)
        if err != nil {
                return res, err
        }
        return res, nil

}
```

Option 1: Remove the dependency and make our function testable

```go
func ParseWithReader(fileName string) (User, error) {
        res := User{}

        f, err := os.Open(fileName)
        if err != nil {
                return User{}, err
        }
        defer f.Close()

        err = json.NewDecoder(f).Decode(&res)
        if err != nil {
                return res, err
        }
        return res, nil

}
```

Our
Dependency independent
logic

Option 1: Remove the dependency and make our function testable

```go
func Parse(fileName string) (User, error) {
  f, err := os.Open(fileName)
  if err != nil {
    return User{}, err
  }
  defer f.Close()
  return parseReader(f)
}
```

Option 1: Remove the dependency and make our function testable

```go
func Parse(fileName string) (User, error) {
  f, err := os.Open(fileName)
  if err != nil {
    return User{}, err
  }
  defer f.Close()
  return parseReader(f)
}
```

```go
func parseReader(r io.Reader) (User, error) {
  res := User{}
  err := json.NewDecoder(r).Decode(&res)
  if err != nil {
    return res, err
  }
  return res, nil
}
```

Option 1: Remove the dependency and make our function testable

```
func Parse(fileName string) (User, error) {
  f, err := os.Open(fileName)
  if err != nil {
    return User{}, err
  }
  defer f.Close()
  return parseReader(f)
}
```

We test this

```
func parseReader(r io.Reader) (User, error) {
  res := User{}
  err := json.NewDecoder(r).Decode(&res)
  if err != nil {
    return res, err
  }
  return res, nil
}
```

Option 2: Replace the dependency with something we control

Option 2: Replace the dependency with something we control

```
func UsersAverageAge() (int, error) {
  users, err := users.Get()
  if err != nil {
    return 0, err
  }
  return averageAgeForUsers(users), nil
}
```

Option 2: Replace the dependency with something we control

Our Dependency

```go
func UsersAverageAg() (int, error) {
  users, err := users.Get()
  if err != nil {
    return 0, err
  }
  return averageAgeForUsers(users), nil
}
```

Option 2: Replace the dependency with something we control

```go
var usersGet = users.Get

func UsersAverageAgeReplace() (int, error) {
  users, err := usersGet()
  if err != nil {
    return 0, err
  }
  return averageAgeForUsers(users), nil
}
```

On the testing side

```
var returnOneUser = func() ([]users.User, error) {
  return []users.User{{"foo", 12}}, nil
}
var failToGet = func() ([]users.User, error) {
  return nil, errors.New("failed")
}
```

```go
func TestAverageAgeReplace(t *testing.T) {
  old := usersGet

  t.Run("oneUser", func(t *testing.T) {
    usersGet = returnOneUser
    t.Cleanup(func() { usersGet = old })

    avg, _ := UsersAverageAgeReplace()
    if avg != 12 {
      t.Fail()
    }
  })
}
```

```go
func TestAverageAgeReplace(t *testing.T) {
	old := usersGet

	t.Run("oneUser", func(t *testing.T) {
		usersGet = returnOneUser
		t.Cleanup(func() { usersGet = old })

		avg, _ := UsersAverageAgeReplace()
		if avg != 12 {
			t.Fail()
		}
	})
}
```

Dependency Injection

In software engineering, dependency injection is a design pattern in which an object or function receives other objects or functions that it depends on. A form of inversion of control, dependency injection aims to separate the concerns of constructing objects and using them, leading to loosely coupled programs.

Dependency Injection

```
type usersRetriever func() ([]users.User, error)

func UsersAverageAgeInj(findUsers usersRetriever)
(int, error) {
  users, err := findUsers()
  if err != nil {
    return 0, err
  }
  return averageAgeForUsers(users), nil
}
```

Dependency Injection

We inject
the dependency

```
type usersRetriever func()               er, error)

func UsersAverageAgeInj(findUsers usersRetriever)
(int, error) {
  users, err := findUsers()
  if err != nil {
    return 0, err
  }
  return averageAgeForUsers(users), nil
}
```

On the testing side

```go
func TestAverageAge(t *testing.T) {

  t.Run("oneUser", func(t *testing.T) {
    avg, _ := UsersAverageAgeInj(returnOneUser)
    if avg != 12 {
      t.Fail()
    }
  })
  t.Run("with Err", func(t *testing.T) {
    _, err := UsersAverageAgeInj(failToGet)
    if err == nil {
      t.Fail()
    }
  })
}
```

```go
func TestAverageAge(t *testing.T) {

  t.Run("oneUser", func(t *testing.T) {
    avg, _ := UsersAverageAgeInj(returnOneUser)
    if avg != 12 {
      t.Fail()
    }
  })
  t.Run("with Err", func(t *testing.T) {
    _, err := UsersAverageAgeInj(failToGet)
    if err == nil {
      t.Fail()
    }
  })
}
```

With Objects

Dependency Injection

```go
func AppUsersAverageAge() (int, error) {
  app := users.NewApplication()
  users, err := app.Users()
  if err != nil {
    return 0, err
  }
  return averageAgeForUsers(users), nil
}
```

Dependency Injection

Our
Dependency

```
f            AverageAge() (int, error) {
  app := users.NewApplication()
  users, err := app.Users()
  if err != nil {
    return 0, err
  }
  return averageAgeForUsers(users), nil
}
```

Dependency Injection

```go
type UsersGetter interface {
  Users() ([]users.User, error)
}

var _ UsersGetter = users.Application{}

func AppUsersAverageAgeInj(getter UsersGetter) (int, error) {
  users, err := getter.Users()
  if err != nil {
    return 0, err
  }
  return averageAgeForUsers(users), nil
}
```

Dependency Injection

```go
type UsersGetter interface {
  Users() ([]users.User, error)
}

var _ UsersGetter = users.Appli

func AppUsersAverageAgeInj(getter UsersGetter) (int, error) {
  users, err := getter.Users()
  if err != nil {
    return 0, err
  }
  return averageAgeForUsers(users), nil
}
```

We inject
the dependency

On the testing side

Dependency Injection

```go
type mockApp struct {
  called    int
  usersRes  []users.User
  shouldErr bool
}
```

Dependency Injection

```go
type mockApp struct {
  called    int
  usersRes  []users.User
  shouldErr bool
}
```

```go
var _ UsersGetter = &mockApp{}

func (m *mockApp) Users() ([]users.User, error) {
  m.called++

  if m.shouldErr {
    return nil, errors.New("failed")
  }
  return m.usersRes, nil
}
```

Dependency Injection

```go
type mockApp struct {
  called    int
  usersRes  []users.User
  shouldErr bool
```

satisfies the interface

```go
var _ UsersGetter = &mockApp{}

func (m *mockApp) Users() ([]users.User, error) {
  m.called++

  if m.shouldErr {
    return nil, errors.New("failed")
  }
  return m.usersRes, nil
}
```

Dependency Injection

```go
type mockApp struct {
  called    int
  usersRes  []users.User
  shouldErr bool
}
```

```go
var _ UsersGetter = &mockApp{}

func (m *mockApp) Users() ([]users.User, error) {
  m.called++

  if m.shouldErr {
    return nil, errors.New("failed")
  }
  return m.usersRes, nil
}
```

control the behaviour

Dependency Injection

```go
type mockApp struct {
	called    int
	usersRes  []users.User
	shouldErr bool
}
```

```go
var _            = &mockApp{}

func             Users() ([]users.User, error) {
	m.called++

	if m.shouldErr {
		return nil, errors.New("failed")
	}
	return m.usersRes, nil
}
```

probe

control the behaviour

Dependency Injection

```
func TestSuccess(t *testing.T) {
  m := &mockApp{
    called:    0,
    usersRes:  []users.User{{"foo", 12}, {"bar", 14}},
    shouldErr: false}

  res, err := AppUsersAverageAgeInj(m)
  if m.called != 1 {
    t.Fail()
  }
  if res != 13 {
    t.Fail()
  }
  // check error is nil
}
```

Dependency Injection

```go
func TestSuccess(t *testing.T) {
  m := &mockApp{
    called:    0,
    usersRes:  []users.User{{"foo", 12}, {"bar", 14}},
    shouldErr: false}

  res, err := AppUsersAverageAgeInj(m)
  if m.called != 1 {
    t.Fail()
  }
  if res != 13 {
    t.Fail()
  }
  // check error is nil
}
```

we verify the probe

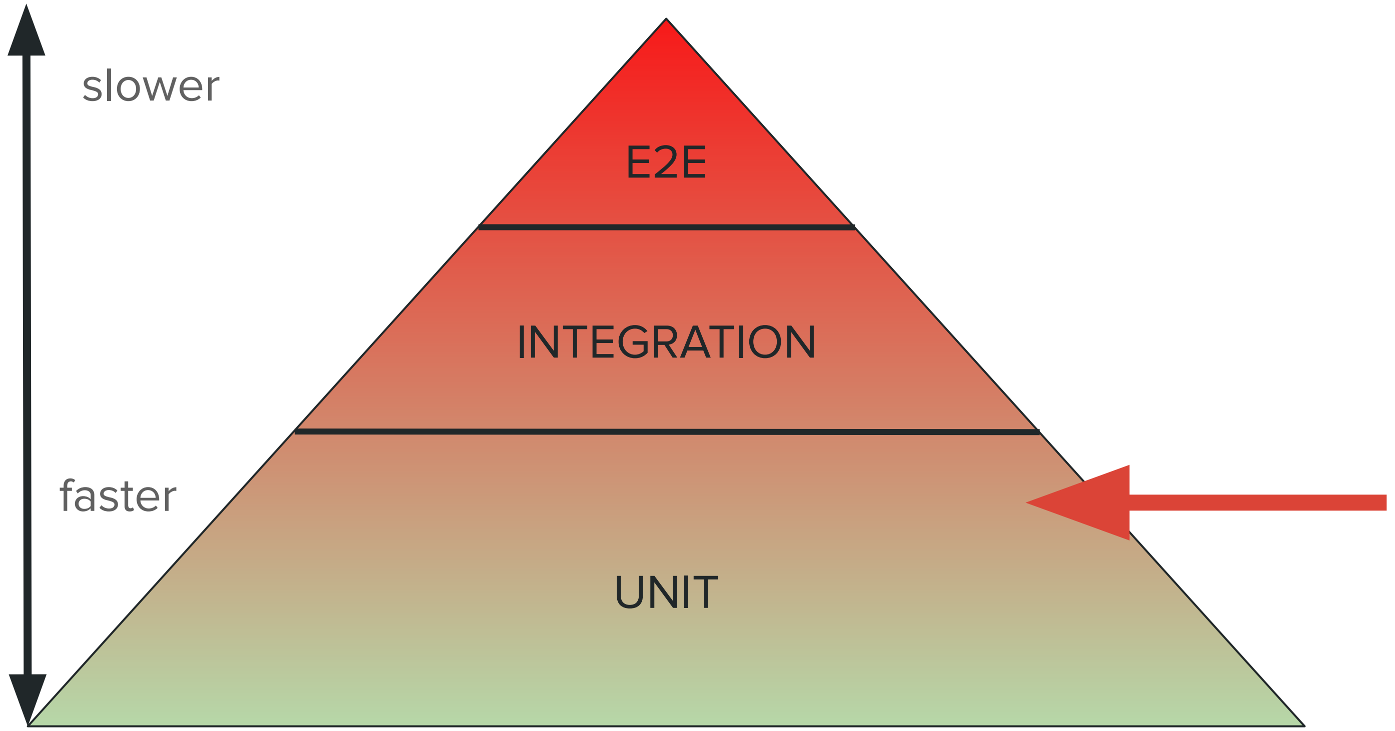
we verify the behaviour

Dependency injection - practice

Dependency injection - practice review

Integration tests - but not really

The testing pyramid: fast vs slow



Instead of checking the server business logic, run it and use a proper client

- Http
- Grpc
- Custom protocols
- Kubernetes API

Grpc Example

Let’s implement a grpc server

```
service UserGet {
  rpc Users (EmptyParams) returns (UsersReply);
}

message UsersReply {
  repeated User users = 1;
}
```

Testing a grpc server

```go
func (s *server) Users(context.Context, *grpcusers.EmptyParams)
(*grpcusers.UsersReply, error) {
  uu, err := s.fetcher.Users()
  if err != nil {
    return nil, err
  }
  res := &grpcusers.UsersReply{
    Users: localUsersToGrpc(uu),
  }
  return res, nil
}
```

Testing a grpc server

```
func (s *server) Users(context.Context, *grpcusers.EmptyParams)
(*grpcusers.UsersReply, error) {
  uu, err := s.fetcher.Users()
  if err != nil {
    return nil, err
  }
  res := &grpcusers.UsersReply{
    Users: localUsersToGrpc(uu),
  }
  return res, nil
}
```

test only the business logic

Testing the business logic

```
func TestBusinessLogic(t *testing.T) {
  grpcUsers := localUsersToGrpc([]users.User{{"foo", 12}, {"bar",
13}})
  if len(grpcUsers) != 2 {
    t.Fail()
  }
}
```

Testing a grpc server

invoke the stub implementation manually

```
func (s *server) Users(context.Context, *grpcusers.EmptyParams)
(*grpcusers.UsersReply, error) {
  uu, err := s.fetcher.Users()
  if err != nil {
    return nil, err
  }
  res := &grpcusers.UsersReply{
    Users: localUsersToGrpc(uu),
  }
  return res, nil
}
```

Testing the stub implementation

```
func TestImplementation(t *testing.T) {
  s := &server{}
  r, err := s.Users(context.Background(), &grpcusers.EmptyParams{})
  if err != nil {
    t.Fail()
  }
  if len(r.Users) != 2 {
    t.Fail()
  }
}
```

Run the server and use a client against it

```go
func TestServer(t *testing.T) {
  s := setupServer()
  clientConn := setupClient()
  client := grpcusers.NewUserGetClient(clientConn)

  t.Cleanup(func() {
    clientConn.Close()
    s.Stop()
  })

  t.Run("simple call", func(t *testing.T) {
    reply, err := client.Users(context.Background(), &grpcusers.EmptyParams{})
    if err != nil {
      t.Fail()
    }
    if len(reply.Users) != 2 {
      t.Fail()
    }
  })
}
```

Run the server and use a c st it

run the server

```
func TestServer(t *testing ) {
  s := setupServer()
  clientConn := setupClient()
  client := grpcusers.NewUserGetClient(clientConn)

  t.Cleanup(func() {
    clientConn.Close()
    s.Stop()
  })

  t.Run("simple call", func(t *testing.T) {
    reply, err := client.Users(context.Background(), &grpcusers.EmptyParams{})
    if err != nil {
      t.Fail()
    }
    if len(reply.Users) != 2 {
      t.Fail()
    }
  })
}
```

Run the server and use a client against it

```go
func TestServer(t *testing.T) {
  s := setupServer()
  clientConn := setupClient()
  client := grpcusers.NewUserGetClient(clientConn)

  t.Cleanup(func() {
    clientConn.Close()
    s.Stop()
  })

  t.Run("simple call", func(t *testing.T) {
    reply, err := client.Users(context.Background(), &grpcusers.EmptyParams{})
    if err != nil {
      t.Fail()
    }
    if len(reply.Users) != 2 {
      t.Fail()
    }
  })
}
```

connect
the client

Run the server and use a client against it

```
func TestServer(t *testing.T) {
  s := setupServer()
  clientConn := setupClient()
  client := grpcusers.NewUserGetClient(clientConn)

  t.Cleanup(func() {
    clientConn.Close()
    s.Stop()
  })

  t.Run("simple call", func(t *testing.T) {
    reply, err := client.Users(context.Background(), &grpcusers.EmptyParams{})
    if err != nil {
      t.Fail()
    }
    if len(reply.Users) != 2 {
      t.Fail()
    }
  })
}
```

use the client

Run the server and use a client against it

```
func TestServer(t *testing.T) {
  s := setupServer()
  clientConn := setupClient()
  client := grpcusers.NewUserGet(                n)

  t.Cleanup(func() {
    clientConn.Close()
    s.Stop()
  })

  t.Run("simple call", func(t *testing.T) {
    reply, err := client.Users(context.Background(), &grpcusers.EmptyParams{})
    if err != nil {
      t.Fail()
    }
    if len(reply.Users) != 2 {
      t.Fail()
    }
  })
}
```

tear down

We can use dependency injection to mock our dependencies

```
type server struct {
  fetcher users.Application
  grpcusers.UnimplementedUserGetServer
}

func (s *server) Users(context.Context, *grpcusers.EmptyParams)
(*grpcusers.UsersReply, error) {
  uu, err := s.fetcher.Users()
  // use users
}
```

We can use dependency injection to mock our dependencies

our dependency

```
type server struct {
  fetcher users.Application
  grpcusers.UnimplementedUserGetServer
}

func (s *server) Users(context.Context, *grpcusers.EmptyParams)
(*grpcusers.UsersReply, error) {
  uu, err := s.fetcher.Users()
  // use users
}
```

Same with Http

Testing an http client

```go
func TestFetchUsers(t *testing.T) {

  svr := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    uu := []users.User{{"foo", 12}, {"bar", 13}}
    json.NewEncoder(w).Encode(uu)
  }))


  t.Cleanup(svr.Close)


  toCheck, err := FetchUsers(svr.URL)
  if err != nil {
    t.Error("received error", err)
  }
  if len(toCheck) != 2 {
    t.Fail()
  }
}
```

Testing an http client

```go
func TestFetchUsers(t *testing.T) {

  svr := httptest.NewServer(http.HandlerFunc(func(w http.Res                  http.Request) {
    uu := []users.User{{"foo", 12}, {"bar", 13}}
    json.NewEncoder(w).Encode(uu)
  }))


  t.Cleanup(svr.Close)


  toCheck, err := FetchUsers(svr.URL)
  if err != nil {
    t.Error("received error", err)
  }
  if len(toCheck) != 2 {
    t.Fail()
  }
}
```

we control the behaviour

Testing an http client

```go
func TestFetchUsers(t *testing.T) {

  svr := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    uu := []users.User{{"foo", 12}, {"bar", 13}}
    json.NewEncoder(w).Encode(uu)
  }))


  t.Cleanup(svr.Close)


  toCheck, err := FetchUsers(svr.URL)
  if err != nil {
    t.Error("received error", err)
  }
  if len(toCheck) != 2 {
    t.Fail()
  }
}
```

our dependency

Integration tests - practice

Integration tests - practice review

Integration tests - a bit more!

Problem statement:

- We interact with an external component
- The interaction is low level
- Spinning up the external component is relatively simple

Problem statement:

- We interact with an external component
- The interaction is low level
- Spinning up the external component is relatively simple

- SQL queries
- Prometheus
- Generated configuration files

Testing the interaction with Redis

```go
func (s *Storage) AddUser(ctx context.Context, user users.User) error {
  jsonUser, err := json.Marshal(user)
  if err != nil {
    return err
  }
  _, err = s.client.Do(ctx, "sel", user.Name, string(jsonUser)).Result()
  if err != nil {
    return err
  }
  return nil
}
```

Testing the interaction with Redis

```
func (s *Storage) AddUser(ctx                t, user users.User) error {
  jsonUser, err := json.Marsha
  if err != nil {
    return err
  }
  _, err = s.client.Do(ctx, "sel", user.Name, string(jsonUser)).Result()
  if err != nil {
    return err
  }
  return nil
}
```

our dependency

Let’s use a mock!

Testing with a mock

```
func (s *Storage) AddUser(ctx context.Context, user users.User) error {
  jsonUser, err := json.Marshal(user)
  if err != nil {
    return err
  }
  _, err = s.client.Do(ctx, "sel", user.Name, string(jsonUser)).Result()
  if err != nil {
    return err
  }
  return nil
}
```

Testing with a mock

```
func (f *fakeRedisClient) Do(ctx context.Context, cmd
...interface{}) *redis.Cmd {
  f.lastCall = cmd
  return &redis.Cmd{}
}
```

Testing with a mock

```go
func (f *fakeRedisClient) Do(ctx context.Context, cmd
...interface{}) *redis.Cmd {
  f.lastCall = cmd
  return &redis.Cmd{}
}


func TestWithMock(t *testing.T) {
  f := &fakeRedisClient{}
  s := Storage{client: f}

  s.AddUser(context.TODO(), users.User{

  if f.lastCall[0].(string) != "sel" {
    t.Fatal()
  }
  // assert lastCall[1] == json(user)
}
```

We check the interaction with the dependency

Testing with a mock

```
func (f *fakeRedisClient) Do(ctx context.Context, cmd
```

Problem: the right command is **set**, not sel!

No one warned us. When we pass the parameters to a mock object, we program it to behave the way **we think** it's going to have

This includes assuming that we are passing the right parameters

```
    // assert lastCall[1] == json(user)
}
```

Let’s test against the real thing

Testing with test containers

```
func TestWithRedis(t *testing.T) {
  req := testcontainers.ContainerRequest{
    Image:        "redis:latest",
    ExposedPorts: []string{"6379/tcp"},
    WaitingFor:   wait.ForLog("Ready to accept connections"),
  }
  redisC, _ := testcontainers.GenericContainer(context.Background(),
  testcontainers.GenericContainerRequest{
    ContainerRequest: req,
    Started:          true,
  })

  mapped, _ := redisC.MappedPort(context.Background(), "6379/tcp")

  t.Cleanup(func() { redisC.Terminate(context.Background()) })

  storage := NewStorage("127.0.0.1:" + mapped.Port())

  // test
}
```

Testing with test containers

```go
func TestWithRedis(t *testing.T) {
  req := testcontainers.ContainerRequest{
    Image:        "redis:latest",
    ExposedPorts: []string{"6379/tcp"},
    WaitingFor:   wait.ForLog("Ready to accept connections"),
  }
  redisC, _ := testcontainers.GenericContainer(context.Background(),
  testcontainers.GenericContainerRequest{
    ContainerRequest: req,
    Started:          true,
  })

  mapped, _ := redisC.MappedPort(context.Background(), "6379/tcp")

  t.Cleanup(func() { redisC.Terminate(context.Background()) })

  storage := NewStorage("127.0.0.1:" + mapped.Port())

  // test
}
```

Testing with test containers

```go
func TestWithRedis(t *testing.T) {
  req := testcontainers.ContainerRequest{
    Image:        "redis:latest",
    ExposedPorts: []string{"6379/tcp"},
    WaitingFor:   wait.ForLog("Ready to accept connections"),
  }
  redisC, _ := testcontainers.GenericContainer(context.Background(),
  testcontainers.GenericContainerRequest{
    ContainerRequest: req,
    Started:          true,
  })

  mapped, _ := redisC.MappedPort(context.Background(), "6379/tcp")

  t.Cleanup(func() { redisC.Terminate(context.Background()) })

  storage := NewStorage("127.0.0.1:" + mapped.Port())

  // test
}
```

Testing with test containers

```
  err := storage.AddUser(context.Background(), users.User{"foo", 12})
  if err != nil {
    t.Fatal("add user failed", err)
  }
  user, err := storage.GetUser(context.Background(), "foo")
  if err != nil {
    t.Fatal("get user failed", err)
  }
  if user.Age != 12 {
    t.Fatal("age is not 12")
  }
}
```

Testing with test containers

```
  err := storage.AddUser(context.Background(), users.User{"foo", 12})
  if err != nil {
    t.Fatal("add user failed", err)
  }
  user, err := storage.GetUser(context.Background(), "foo")
  if err != nil {
    t.Fatal("get user failed", err)
  }
  if user.Age != 12 {
    t.Fatal("age is not 12")
  }
}
```

```
--- FAIL: TestWithRedis (2.20s)
    redis_client_test.go:35: add user failed ERR unknown command 'sel',
       with args beginning with: 'foo' '{"name":"foo","age":12}'
```

Interacting with the real object advantages

- No behavior discrepancy between the mocked and the real dependency
- Early validation of configurations
- Easier to setup than real end to end tests

Interacting with the real object disadvantages

- Slower than mock objects
- More moving parts (even if just a little), more subject to flakes
    - For example, calling get right after an add might not succeed

Use test.Short()

```
func TestWithRedis(t *testing.T) {
  if testing.Short() {
    t.Skip("container test, skipping with -short")
  }

  //
```

```
go test –short ./…
```

Container tests practice

Making our tests stable

Flaky tests

are bad

If the test sometimes fail, the responsibility can be:

- of our code
- of the testing environment

Flaky tests

are bad

If the test sometimes fail, the responsibility can be:

- of our code
- of the testing environment
- getting a green run after retrying is not a valid excuse

Ignoring a flaky test could mean ignoring a real bug

If the number of flaky tests grows, we’ll lose confidence in their value

Fixing flaky tests is a thankless job

Debuggability difficulty (in growing order)

- Consistent failure happening on our laptop
- Consistent failure happening in CI
- A flaky test happening often and locally too
- A flaky test happening only in CI
- A bug happening in production

What to log in our tests

The more moving parts, the more info we need to collect

- Our test is (hopefully) going to run with other 1000s of tests in CI
- When we have a failure, we must have all the information to understand what happened

Unit Tests

with Mocks

- Easy to reproduce
- Less likely to be influenced by the state
- No need to be verbose: just rerun the test!

Unit tests can just be “verbose enough”

```
func reply(path string, t *testing.T) []byte {
  expected, err := os.ReadFile(fmt.Sprintf("testdata/%s.json", path))
  if err != nil {
    t.Fatalf(“path not found”)
  }
  return expected
}


func TestFetchUsers(t *testing.T) {
  svr := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter,
                                                 r *http.Request) {
    res := reply(r.URL.Path, t)
    w.Write(res)
  }))
```

Unit tests can just be “verbose enough”

```
func reply(path string, t *testing.T) []byte {
  expected, err := os.ReadFile(fmt.Sprintf("tes                    path))
  if err != nil {
    t.Fatalf(“path not found”)
  }
  return expected
}


func TestFetchUsers(t *testing.T) {
  svr := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter,
                                                r *http.Request) {
    res := reply(r.URL.Path, t)
    w.Write(res)
  }))
```

it will always fail!

Using test helpers

- we want to maintain readability
- it’s not one of those “should never fail” scenarios because it depends on the instrumented code
- reports the failure as happening in the caller

Using test helpers

```go
func reply(path string, t *testing.T) []byte {
  t.Helper()
  expected, err := os.ReadFile(fmt.Sprintf("testdata/%s.json", path))
  if err != nil {
    t.Fatalf("path not found")
  }
  return expected
}


func TestFetchUsers(t *testing.T) {
  svr := httptest.NewServer(http.Handl          http.ResponseWriter,
                                               equest) {

    res := reply(r.URL.Path, t)
    w.Write(res)
  }))
```

no need to check and handle the error

End to end / integration tests

- We run our tests against an external component
- Non trivial risk to leak state across the tests
- Risk to have timing issues
- Network!
- We need to collect what we need to understand what happened

Examples

- Using container tests
- End to end against a running system / set of microservices
- Kubernetes!

Example: testing against an external storage

```
err := storage.AddUser(context.Background(), users.User{"foo", 12})
 if err != nil {
   t.Fatal("add user failed", err)
 }
 user, err := storage.GetUser(context.Background(), "foo")
 if err != nil {
   t.Fatal("get user failed", err)
 }
 if user.Age != 12 {
   t.Fatal("age is not 12")
 }
}
```

One day this test will fail in CI and you’ll scratch your head

Dump the status of the system being instrumented after a failure

- The external system is not reset across tests
- We must ensure that the status is what we were expecting
- Maybe we made the wrong assumptions about the system!
- Maybe our tests are not resilient enough

Example: testing against an external storage

```go
t.Run("add and get", func(t *testing.T) {
  err := storage.AddUser(context.Background(), users.User{"foo", 12})
  if err != nil {
    dumpRedisContent(url, t)
    t.Fatal("add user failed", err)
  }
  user, err := storage.GetUser(context.Background(), "foo")
  if err != nil {
    dumpRedisContent(url, t)
    t.Fatal("get user failed", err)
  }
  //
})
```

Example: testing against an external storage

```go
t.Run("add and get", func(t *testing.T) {
  err := storage.AddUser(context.Background(), users.User{"foo", 12})
  if err != nil {
    dumpRedisContent(url, t)
    t.Fatal("add user failed", err)
  }
  user, err := storage.GetUser(context.Background(), "foo")
  if err != nil {
    dumpRedisContent(url, t)
    t.Fatal("get user failed", err)
  }
  //
})
```

Dump the status and add it to your CI artifacts

- use t.Name() to get the name of the test, and use it to name the file containing the dump
- multiple files with the test name as root for different sections for better navigability
- dumps as part of the CI artifacts

Example: dump the content of redis

```go
func dumpRedisContent(url string, rdb *redis.Client, t *testing.T) string {
  t.Helper()
  ctx := context.Background()

  iter := rdb.Scan(ctx, 0, "", 0).Iterator()
  res := ""
  for iter.Next(ctx) {
    key := iter.Val()
    val, _ := rdb.Get(ctx, key).Result()
    res = res + fmt.Sprintf("%s: %s\n", key, val)
  }
  filename := strings.Replace(t.Name(), "/", "-", -1) + ".dump"
  if err := os.WriteFile(filename, []byte(res), 0666); err != nil {
    t.Fatal(err)
  }
  return res
}
```

Example: dump the content of redis

```
func dumpRedisContent(url string, rdb *redis.Client, t *testing.T) string {
  t.Helper()
  ctx := context.Background()

  iter := rdb.Scan(ctx, 0, "", 0).Iterator()
  res := ""
  for iter.Next(ctx) {
    key := iter.Val()
    val, _ := rdb.Get(ctx, key).Result()
    res = res + fmt.Sprintf("%s: %s\n", key, val)
  }
  filename := strings.Replace(t.Name(), "/", "-", -1) + ".dump"
  if err := os.WriteFile(filename, []byte(res), 0666); err != nil {
    t.Fatal(err)
  }
  return res
}
```

dump the status in res

Example: dump the content of redis

```go
func dumpRedisContent(url string, rdb *redis.Client, t *testing.T) string {
  t.Helper()
  ctx := context.Background()

  iter := rdb.Scan(ctx, 0, "", 0).Iterator()
  res := ""
  for iter.Next(ctx) {
    key := iter.Val()
    val, _ := rdb.Get(ctx, key).Result()
    res = res + fmt.Sprintf("%s: %s\n", key, val)
  }
  filename := strings.Replace(t.Name(), "/", "-", -1) + ".dump"
  if err := os.WriteFile(filename, []byte(res), 0666); err != nil {
    t.Fatal(err)
  }
  return res
}
```

write it to a file named after the test

Example: dump the content of redis

```go
func dumpRedisContent(url string, rdb *redis.Client, t *testing.T) string {
  t.Helper()
  ctx := context.Background()

  iter := rdb.Scan(ctx, 0, "", 0).Iterator()
  res := ""
  for iter.Next(ctx) {
    key := iter.Val()
    val, _ := rdb.Get(ctx, key).Result()
    res = res + fmt.Sprintf("%s: %s\n", key, val)
  }
  filename := strings.Replace(t.Name(), "/", "-", -1) + ".dump"
  if err := os.WriteFile(filename, []byte(res), 0666); err != nil {
    t.Fatal(err)
  }
  return res
}
```

$\searrow$

`TestWithRedis-add_and_get.dump`

It’s an iterative process

Wrap up and takeaways

Test to ensure and document behavior

A good test suite pays back dividends in the long run

“Just enough” e2e tests

Keep the testsuite reliable: chase flakes and false positive/negatives

Avoid “retry” more than once

Good e2e tests require continued effort

Thank you!

Backup and Extras

Benchmarking

Benchmarking

- A function is in the critical path and we want to optimize it
- We already found that a function is a performance bottleneck (possibly with pprof)
- You want to reduce the memory footprint of a given function

Benchmarking

```go
func BenchmarkParse(b *testing.B) {
  for i := 0; i < b.N; i++ {
    ParseWithReader("testdata/basic.json")
  }
}
```

Benchmarking

```go
func BenchmarkParse(b *testing.B) {
  for i := 0; i < b.N; i++ {
    ParseWithReader("testdata/basic.json")
  }
}
```

Benchmarking

```
func BenchmarkParse(b *testing.B) {
  for i := 0; i < b.N; i++ {
    ParseWithReader("testdata/basic.json")
  }
}
```

Benchmarking

```go
func BenchmarkParse(b *testing.B) {
  for i := 0; i < b.N; i++ {
    ParseWithReader("testdata/basic.json")
  }
}
```

Benchmarking

```go
func BenchmarkParse(b *testing.B) {
  for i := 0; i < b.N; i++ {
    ParseWithReader("testdata/basic.json")
  }
}
```

```
go test -bench . -benchmem
goos: linux
goarch: amd64
pkg: benchmarking
cpu: Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
BenchmarkParse-8           61327            16744 ns/op            1072 B/op           11 allocs/op
PASS
ok      benchmarking    1.231s
```

Benchmarking

```go
func BenchmarkParse(b *testing.B) {
  for i := 0; i < b.N; i++ {
    ParseWithReader("testdata/basic.json")
  }
}
```

Nano seconds / Operation

```
go test -bench . -benchmem
goos: linux
goarch: amd64
pkg: benchmarking
cpu: Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
BenchmarkParse-8           61327          16744 ns/op         1072 B/op          11 allocs/op
PASS
ok      benchmarking    1.231s
```

Benchmarking

```go
func BenchmarkParse(b *testing.B) {
  for i := 0; i < b.N; i++ {
    ParseWithReader("testdata/basic.json")
  }
}
```

Allocated bytes / Operation

```
go test -bench . -benchmem
goos: linux
goarch: amd64
pkg: benchmarking
cpu: Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
BenchmarkParse-8           61327           16744 ns/op            1072 B/op          11 allocs/op
PASS
ok      benchmarking    1.231s
```

Benchmarking

```go
func BenchmarkParse(b *testing.B) {
  for i := 0; i < b.N; i++ {
    ParseWithReader("testdata/basic.json")
  }
}
```

```
go test -bench . -benchmem
goos: linux
goarch: amd64
pkg: benchmarking
cpu: Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
BenchmarkParse-8           61327           16744 ns/op             1072 B/op           11 allocs/op
PASS
ok      benchmarking    1.231s
```

Allocations / Operation

Benchmarking

- Same knobs as tests:
    - sub benchmarks (b.Run)
    - setup / teardown (b.Cleanup)

Benchstat

go install golang.org/x/perf/cmd/benchstat@latest

```
benchstat benchmarshal.txt benchdecode.txt
goos: linux
goarch: amd64
pkg: benchmarking
cpu: Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
        │ benchmarshal.txt │           benchdecode.txt            │
        │      sec/op      │    sec/op     vs base                │
Parse-8       25.73µ ± 23%   19.05µ ± 11%  -25.98% (p=0.003 n=10)

        │ benchmarshal.txt │           benchdecode.txt           │
        │       B/op       │     B/op      vs base               │
Parse-8       1.070Ki ± 0%   1.047Ki ± 0%  -2.19% (p=0.000 n=10)

        │ benchmarshal.txt │        benchdecode.txt         │
        │    allocs/op     │ allocs/op   vs base            │
Parse-8         11.00 ± 0%   11.00 ± 0%  ~ (p=1.000 n=10) ¹
¹ all samples are equal
```

go install golang.org/x/perf/cmd/benchstat@latest

Benchstat

```
benchstat benchmarshal.txt benchdecode.txt
goos: linux
goarch: amd64
pkg: benchmarking
cpu: Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
        │ benchmarshal.txt │           benchdecode.txt           │
        │      sec/op      │    sec/op     vs base               │
Parse-8        25.73µ ± 23%   19.05µ ± 11%  -25.98% (p=0.003 n=10)

        │ benchmarshal.txt │           benchdecode.txt          │
        │       B/op       │     B/op      vs base              │
Parse-8        1.070Ki ± 0%   1.047Ki ± 0%  -2.19% (p=0.000 n=10)

        │ benchmarshal.txt │        benchdecode.txt        │
        │    allocs/op     │ allocs/op   vs base           │
Parse-8          11.00 ± 0%   11.00 ± 0%  ~ (p=1.000 n=10) ¹
¹ all samples are equal
```

Enhancing go testing

go-cmp: richer comparison

go get github.com/google/go-cmp

[...] A more powerful and safer alternative to reflect.DeepEqual for comparing whether two values are semantically equal.

[...]Equality is determined [by default] by recursively comparing the primitive kinds on both values, much like reflect.DeepEqual. Unlike reflect.DeepEqual, unexported fields are not compared by default.

Easy to spot differences: cmp.Diff()

```
package foobar

import “github.com/google/go-cmp/cmp"

func TestFoo(t *testing.T) {
     got, want := Foo()
     if diff := cmp.Diff(want, got); diff != "" {
          t.Errorf("Foo() mismatch (-want +got):\n%s", diff)
     }
}
```

Easy to spot differences: cmp.Diff(): example

```
--- FAIL: TestAssign (0.00s)
     todo_test.go:116:
          Error Trace:    /github.com/ffromani/go-todo-app/model/todo_test.go:116
          Error:          Not equal:
                          expected: ""
                          actual  : "John Doe"

                          Diff:
                          --- Expected
                          +++ Actual
                          @@ -1 +1 @@
                          -
                          +John Doe
          Test:           TestAssign
          Messages:       Assigned local todo has unexpected assignee
```

Test for equality: cmp.Equal

```
package foobar
import "github.com/google/go-cmp/cmp"
func TestBar(t *testing.T) {
    trans := cmp.Transformer("Sort", func(in []int) []int {
        out := append([]int(nil), in...); sort.Ints(out); return out
    })
    want := []int{0, 1, 2, 3}
    got := Bar() // []int{3, 0, 1, 2}
    if !cmp.Equal(x, y, trans) {
        t.Errorf("Bar() mismatch: want: %v got: %v", want, got)
    }
}
```

testify: augmenting the go testing

go get github.com/stretchr/testify

Enhances go testing with:

- Test assertions
- Mocking
- Testing suites (setup/teardown…)

testify: assertions

```go
package foobar

import "github.com/stretchr/testify/assert"

func TestFoo(t *testing.T) {
	got, want := Foo()
	assert.Equal(t, got, want, "Foo() result doesn’t match expectation")
}
```

testify: assertions /2

```
package foobar

import "github.com/stretchr/testify/assert"

func TestFoo(t *testing.T) {
     want := MyObject{value: 42}
     got := FooObject()
    as := assert.New(t)
     if as.NotNil(object) {
       as.Equal(got, want, "FooObject() result doesn’t match expectation")
    }
}
```

testify: assertions example

```
=== RUN   TestString
     types_test.go:22:
          Error Trace:
/home/france/go/src/github.com/ffromani/go-todo-app/config/types_test.go:22
          Error:          Should be empty, but was - address: localhost:8181
          Test:           TestString
          Messages:       String result is empty
--- FAIL: TestString (0.00s)
```

testify: mocks

```
package foobar

import "github.com/stretchr/testify/mock"

type MObj struct {
     mock.Mock
}

func (m *MObj) Stuff(num int) error {
     args := m.Called(num)
     return args.Error(0)
}
```

```
func TestSomething(t *testing.T) {
     testObj := new(MObj)
     testObj.On("Stuff", 42).Return(nil)
     useObjectSomehow(testObj)
     testObj.AssertExpectations(t)
}
```