Beyond table tests

Federico Paolinelli - Red Hat

About me

Telco Network Team @ Red Hat 

- Kubernetes
- Networking
- MetalLB maintainer

hachyderm.io/@fedepaol

@fedepaol

fedepaol@gmail.com

Why do we need tests?

We change our code

Reasons for change

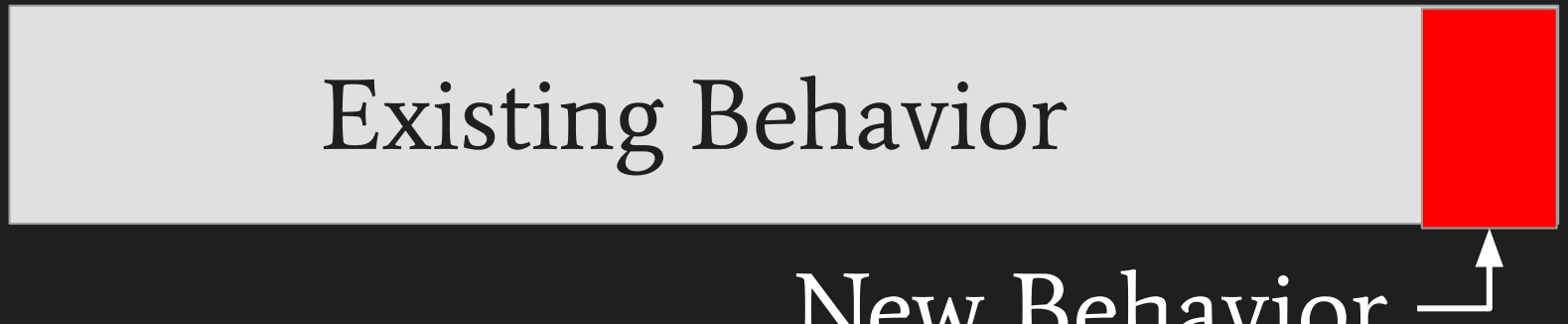
- New features
- Bug fixes
- Refactoring / reducing technical debt
- Optimizations

What do we test?

What do we test?

Existing Behavior

What do we test?



Changes in a system can be made in two primary ways. I like to call them Edit and Pray and Cover and Modify. Unfortunately, Edit and Pray is pretty much the industry standard.

(Micheal Feathers - Working effectively with legacy code)

The cost of fixing a bug

The cost of detecting and fixing defects in software increases exponentially with time in the software development workflow. Fixing bugs in the field is incredibly costly, and risky — often by an order of magnitude or two. The cost is in not just in the form of time and resources wasted in the present, but also in form of lost opportunities of in the future.

from deepsource.io/blog/exponential-cost-of-fixing-bugs/

“In less than an hour, Knight Capital's computers executed a series of automatic orders that were supposed to be spread out over a period of days. Millions of shares changed hands. The resulting loss, which was nearly four times the company's 2011 profit, crippled the firm and brought it to the edge of bankruptcy.”

from money.cnn.com/2012/08/09/technology/knight-expensive-computer-bug

Benefits of having tests

- Better quality
- No regressions
- Document well the instrumented units
- Better design
- Enable CI

Unit tests or integration tests?

Unit tests

- Quick
- Deterministic
- Test a very specific part of our codebase

Integration tests

- Test our system as a whole
- More adherent to reality
- Slow
- May not run on our laptop
- Really hard to master: many moving parts, risk of flakes

What is a unit test?

“.. We asked him (Kent Beck) for his definition and he replied with something like "in the morning of my training course I cover 24 different definitions of unit test"

(Martin Fowler)

Common traits of unit tests

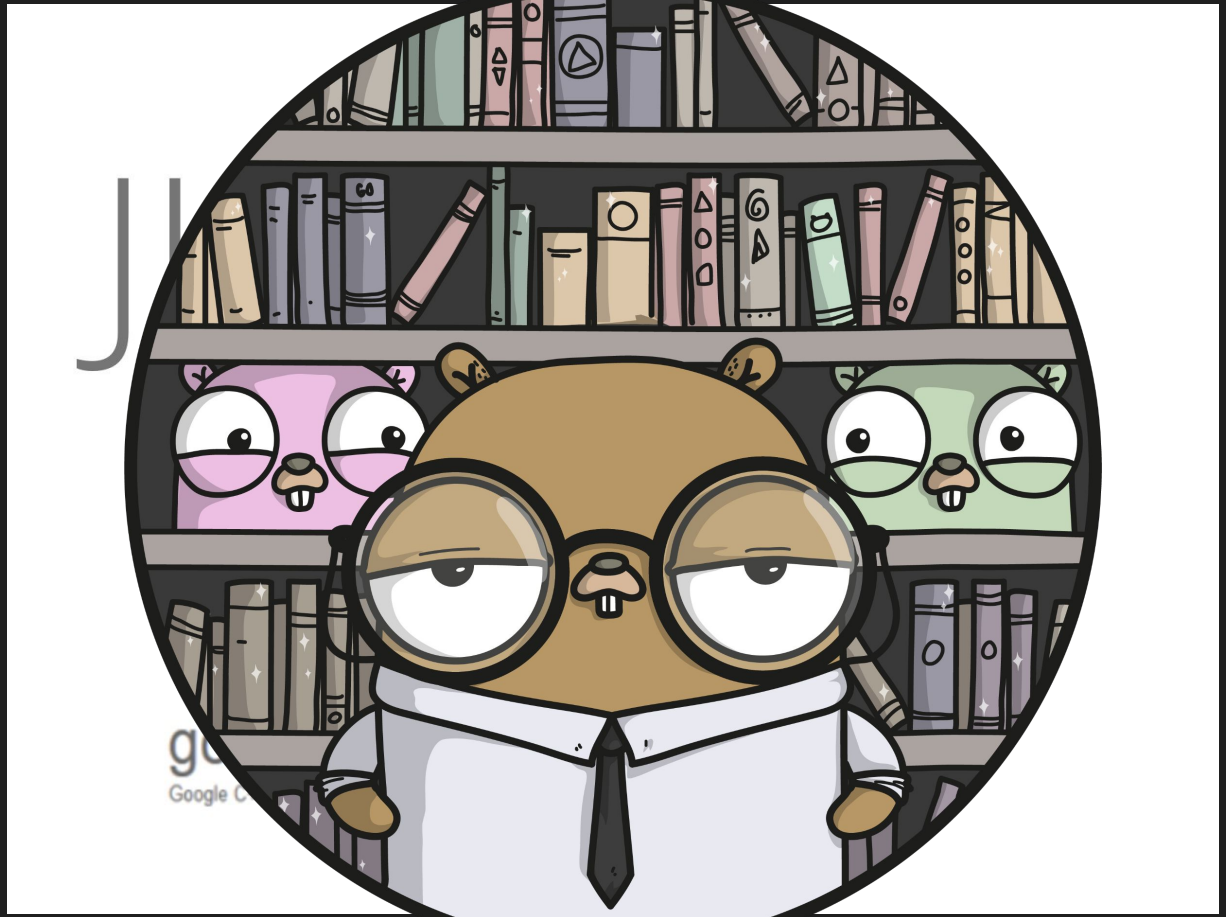
- Low level, focus on a small part of the system
- Written by programmers
- Faster than normal tests
- Not depending on external systems

Unit tests in Go

JUnit 5

googletest
Google C++ Testing Framework

pytest



The basics

```
func Add(x, y int) int {
    return x + y
}
```

from *pkg/math/add.go*

The basics

```go
func Add(x, y int) int {
    return x + y
}
```

from *pkg/math/add.go*

```go
func TestAdd(t *testing.T) {
        res := Add(3, 4)
        if res != 7 {
                t.Fatalf("Expecting 7, got %d", res)
        }
}
```

from *pkg/math/add_test.go*

The basics

```
$ go test
PASS
ok      github.com/fedepaol/gotests/pkg/math    0.002s
```

The basics

```
$ go test
PASS
ok      github.com/fedepaol/gotests/pkg/math    0.002s
```

```
$ go test
--- FAIL: TestAdd (0.00s)
    add_test.go:8: Expecting 7, got 8
FAIL
exit status 1
FAIL    github.com/fedepaol/gotests/pkg/math    0.002s
```

The basics: test only the public API

package math

```
func TestAdd(t *testing.T) {
        res := Add(3, 4)
        if res != 7 {
                t.Fatalf("Expecting 7, got %d", res)
        }
}
```

The basics: test only the public API

package math_test

```
func TestAdd(t *testing.T) {
        res := math.Add(3, 4)
        if res != 7 {
                t.Fatalf("Expecting 7, got %d", res)
        }
}
```

Subtests

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

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

Subtests for

- Better control on what to run
- Share code
- Setup / Tear down pattern
- Enable parallel execution

```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()
        }
    })
}
```

```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()
        }
    })
}
```

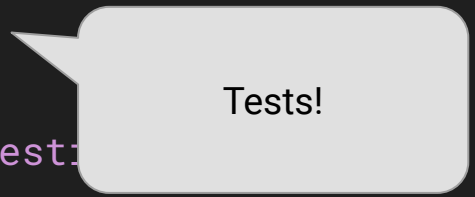
```go
func TestCalculator(t *testing.T) {
    c := NewCalculator()
    t.Cleanup(func() {
        c.Unregister()
    })

    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()
        }
    })
}
```

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 *testi
        if c.Sum(1, 3) != 4 {
            t.Fail()
        }
    })
}
```



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

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

Runs in a separate goroutine

The parent test terminates when all the children are done

Test Main

- One per package
- Lower level
- Useful if a global setup / teardown is needed

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

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()
      }
    })
  }
}
```

```
tests := []struct {
  name     string
  first    int
  second   int
  expected int
}{
  {"1+2", 1, 2, 3},
}
```

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

```
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()
      }
    })
  }
}
```

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

  c := NewCalculator()

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

```
tests := []struct {
  name     string
  first    int
  second   int
  expected int
}{
  {"1+2", 1, 2, 3},
}
```

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

```
  tests := []struct {
    name     string
    first    int
    second   int
    expected int
  }{
    {"1+2", 1, 2, 3},
    {"1+7", 1, 7, 8},
  }
```

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

```
  tests := []struct {
    name     string
    first    int
    second   int
    expected int
  }{
    {"1+2", 1, 2, 3},
    {"1+7", 1, 7, 8},
    {"2+7", 2, 7, 9},
  }
        if c.Sum(tc.first, tc.second) != tc.expected {
          t.Fail()
        }
      })
    }
  }
```

How to control test execution

Filter by name

```
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
```

Slow tests

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

```
go test -short ./...
```

Build tags

```
// +build integration


func TestIntegration(t *testing.T) {
    res, err := Add(2,3)
    ...
}


go test --tags=integration
```

TEST FIXTURES

A test fixture is an environment used to consistently test some item, device, or piece of software.

en.wikipedia.org/wiki/Test_fixture

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

```
$ tree
.
├── add.go
├── add_test.go
└── testdata
    └── sample_config.json
```

Test Fixtures

```
$ tree
.
├── add.go
├── add_test.go
└── testdata
    └── sample_config.json
```

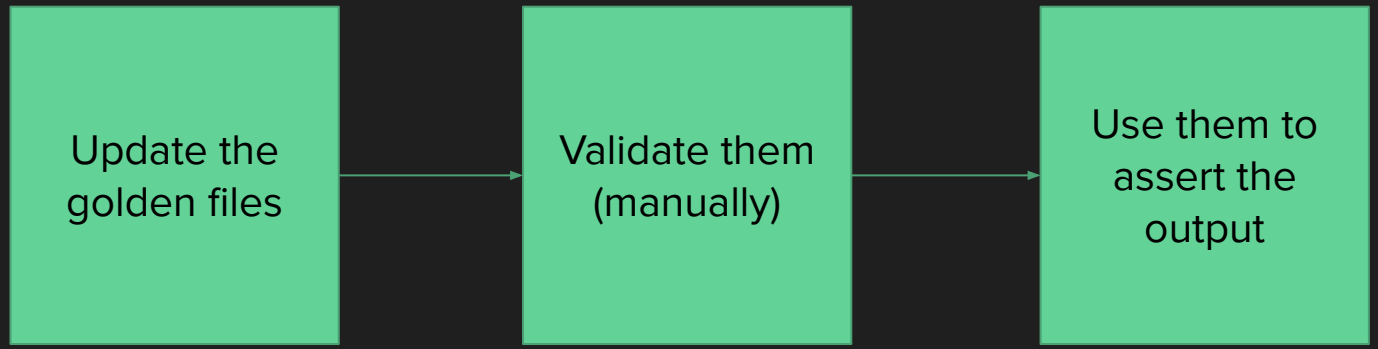
```go
func TestParse(t *testing.T) {
    res, err := parseConfig("testdata/sample_config.json")
    if err != nil {
        t.Fatalf("got error %v but wasn't expecting", err)
    }
    if res != 23 {
        t.Fatalf("got %d but was expecting 23", res)
    }
}
```

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



```
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()
      }
    })
}
```

```
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)
      }
      if err != nil {
        t.Errorf("failed to open golden file %s: %v", goldenFile, err)
      }
      if !bytes.Equal(expected, jsonRes) {
        t.Fail()
      }
    })
}
```

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

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

```
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
```

```
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)
    --- 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
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)
    --- 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
```

EXTRA FEATURES

Coverage

```
go test -cover
PASS
coverage: 50.0% of statements
ok      github.com/fedepaol/gotests/pkg/math    0.002s
```

Benchmarking

```
func BenchmarkAdd(b *testing.B) {
        for i := 0; i < b.N; i++ {
                Add(1, 2)
        }
}
```

Benchmarking

```
$ go test -bench=.
goos: linux
goarch: amd64
pkg: github.com/fedepaol/gotests/pkg/math
cpu: Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz
BenchmarkAdd-4          1000000000               0.2604 ns/op
PASS
ok      github.com/fedepaol/gotests/pkg/math    0.293s
```

Race detector

```
go test -race .
```

Race detector

```
go test -race .
```

Test shuffler

```
go test -shuffle=on .
```

TEST HELPERS

Test Helpers

```
func callAdd(x, y, expected int) error {
  res, err := Add(x, y)
  if err != nil {
      return err
  }

  if res != expected+1 {
      return fmt.Errorf("got %d but was expecting %d", res, expected)
  }
}
```

Test Helpers

```
func callAdd(t *testing.T, x, y, expected int) {
  t.Helper()
  res, err := Add(x, y)
  if err != nil {
      t.Fail()
  }

  if res != expected+1 {
      t.Failf("got %d but was expecting %d", res, expected)
  }
}
```

Test Helpers

```
func TestAdd(t *testing.T, x, y, expected int) error {
  err := callAdd(x, y)
  if err != nil {
      return err
  }
}
```

Test Helpers

```
func TestAdd(t *testing.T, x, y, expected int) error {
  callAdd(t, x, y)
}
```

Test Helpers

```
func TestAdd(t *testing.T, x, y, expected int) error {
  res := callAdd(t, x, y)
}
```

```
FAIL: TestAddWithHelper (0.10s)
  add_test.go:189: got 5 but was expecting 5
```

THE JOURNEY TO (UNIT) TESTABLE CODE

We want our tests

- Not depending from external systems
- Independent of the execution order
- Fast
- Repeatable

Reality check: an instrumented unit can be

- stateful
- interacting with external systems:
  - API
  - os
  - filesystem
  - network

OPTION 1: test pure functions

In computer programming, a pure function is a function that has the following properties: the function return values are identical for identical arguments (no variation with local static variables, non-local variables, mutable reference arguments or input streams), and the function application has no side effects (no mutation of local static variables, non-local variables, mutable reference arguments or input/output streams).

```go
func Parse(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

}
```

```
func Parse(fileName                    r, 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

```go
func Parse(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 business logic

```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
}
```

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


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

Our Dependency

OPTION 2: override the status

```
const configPath = "/etc/constant.json"

func AddConstant(x int) (int, error) {
    value, err := parseConfig(configPath)
    if err != nil {
        return 0, err
    }
    return x + value, nil
}
```

```go
var configPath = "/etc/constant.json"

func AddConstant(x int) (int, error) {
    value, err := parseConfig(configPath)
    if err != nil {
        return 0, err
    }
    return x + value, nil
}
```

```go
func TestAddConstant(t *testing.T) {
    oldPath := configPath
    defer func() { configPath = oldPath }()
    configPath = "testdata/sample_config.json"
    res, _ := AddConstant(28)
    if res != 51 {
        t.Fatalf("got %d but was expecting 51", res)
    }
}
```

```go
func TestAddConstant(t *testing.T) {
    oldPath := configPath
    defer func() { configPath = oldPath }()
    configPath = "testdata/sample_config.json"
    res, _ := AddConstant(28)
    if res != 51 {
        t.Fatalf("got %d but was expecting 51", res)
    }
}
```

Sometimes we can’t avoid the interaction

TEST DOUBLES

The generic term he (Gerard Meszaros) uses is a Test Double (think stunt double). Test Double is a generic term for any case where you replace a production object for testing purposes.

Martin Fowler - martinfowler.com/bliki/TestDouble.html

Let’s introduce our dependency

```
func Add(x, y int) (int, error) {
        res, err := cloudmath.add(x, y)
        if err != nil {
                return 0, err
        }
        return res, nil
}
```

Our Dependency

```
func Add(x, y int) (int, error) {
        res, err := cloudmath.add(x, y)
        if err != nil {
                return 0, err
        }
        return res, nil
}
```

```go
var cloudAdd = cloudmath.Add

func Add(x, y int) (int, error) {
        res, err := cloudAdd(x, y)
        if err != nil {
                return res, nil
        }
        return 0, fmt.Errorf("cloudmath error %w", err)
}
```

```go
func TestCloudAdd(t *testing.T) {
  toRestore := cloudAdd
  defer func() { cloudAdd = toRestore }()

  cloudAdd = func(x,y int) (int, error) {
    return 11, nil
  }

  res, err := Add(5, 6)
  if res != 11 {
    t.Fatalf("expecting 11 but got %d", res)
  }
}
```

```go
func TestCloudAdd(t *testing
  toRestore := cloudAdd
  defer func() { cloudAdd = toRestore }()

  cloudAdd = func(x,y int) (int, error) {
    return 11, nil
  }

  res, err := Add(5, 6)
  if res != 11 {
    t.Fatalf("expecting 11 but got %d", res)
  }
}
```

To avoid leaking to other tests

```go
func TestCloudAdd(t *testing.T) {
  toRestore := cloudAdd
  defer func() { cloudAdd = toRestore }()

  cloudAdd = func(x,y int) (int, error) {
    return 11, nil
  }

  res, err := Add(5, 6)
  if res != 11 {
    t.Fatalf("expecting 11 but got %d", res)
  }
}
```

We control the behavior

```
func TestCloudAdd(t *testing.T) {
  toRestore := cloudAdd
  defer func() { cloudAdd = toRestore }()

  called := 0
  cloudAdd = func(x,y int) (int, error) {
    called++
    return 11, nil
  }

  res, err := Add(5, 6)
  if res != 11 {
    t.Fatalf("expecting 11 but got %d", res)
  }
  if called != 1 {
    t.Fatalf("expecting cloudAdd to be called 1 time, got %d",
called)
  }
}
```

```go
func TestCloudAdd(t *testing.T) {
  toRestore := cloudAdd
  defer func() { cloudAdd = toRestore }()

  called := 0
  cloudAdd = func(x,y int) (int, error) {
    called++
    return 11, nil
  }

  res, err := Add(5, 6)
  if res != 11 {
    t.Fatalf("expecting 11 but got %d", res)
  }
  if called != 1 {
    t.Fatalf("expecting cloudAdd to be called 1 time, got %d",
called)
  }
}
```

We can add probes

Dependency injection to the rescue

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..

```go
type AddFunc func(x, y int) (int, error)

func Add(x, y int, add AddFunc) (int, error) {
    res, err := add(x, y)
    if err != nil {
        return 0, err
    }
    return res, nil
}

Add(3, 4, cloudmath.Add)
```

```go
type AddFunc func(x, y int) (int, error)

func Add(x, y int, add AddFunc) (int, error) {
    res, err := add(x, y)
    if err != nil {
        return 0, err
    }
    return res, nil
}

Add(3, 4, cloudmath.Add)
```

Our dependency comes from outside

```
type AddFunc func(x, y int) (int, error)

func Add(x, y int, add AddFunc) (int, error) {
    res, err := add(x, y)
    if err != nil {
        return 0, err
    }
    return res, nil
}

Add(3, 4, cloudmath.Add)
```

we need to change the calling site

Testing it

```go
func TestAdd(t *testing.T) {
    _, err := Add(2, 3, func(x, y int) (int, error) {
        return 0, fmt.Errorf("boo")
    })

    if err == nil {
        t.Fatal("got no error”)
    }
}
```

we pass the double of our dependency

```
func TestAdd(t *testing.T) {
    _, err := Add(2, 3, func(x, y int) (int, error) {
        return 0, fmt.Errorf("boo")
    })

    if err == nil {
        t.Fatal("got no error”)
    }
}
```

Injecting objects

```go
func AddRemove(x, y int) (int, error) {
      client := cloudmath.NewClient()
      val, err := client.ToAdd()
      if err != nil {
          return 0, err
      }
      val, err = client.Subtract(val, y)
      if err != nil {
          return 0, err
      }
      return x + val + 5, nil
  }
```

```go
func AddRemove(x, y int) (int, error) {
    client := cloudmath.NewClient()
    val, err := client.ToAdd()
    if err != nil {
        return 0, err
    }
    val, err = client.Subtract(val, y)
    if err != nil {
        return 0, err
    }
    return x + val + 5, nil
}
```

Our dependency

```go
func AddRemove(client *cloudmath.Client, x, y int) (int, error) {
    val, err := client.ToAdd()
    if err != nil {
        return 0, err
    }
    val, err = client.Subtract(val, y)
    if err != nil {
        return 0, err
    }
    return x + val + 5, nil
}
```

Our depedency

```
type AddRemover interface {
        ToAdd() (int, error)
        Subtract(x, y int) (int, error)
}

func AddRemove(client AddRemover, x, y int) (int, error) {
    val, err := client.ToAdd()
    if err != nil {
        return 0, err
    }
    val, err = client.Subtract(val, y)
    if err != nil {
        return 0, err
    }
    return x + val + 5, nil
}
```

Now an interface

```
type AddRemover interface {
        ToAdd() (int, error)
        Subtract(x, y int) (int, error)
}

func
    v
    i
    }
    v
    i
        return 0, err
    }
    return x + val + 5, nil
}
```

```
client := cloudmath.NewClient()
res, err := AddRemove(client, x, y)
```

Testing it

```go
type mockClient struct {
    toAddRes    int
    toAddError  error
    toAddCalled bool
}

func (m *mockClient) ToAdd() (int, error) {
    m.toAddCalled = true
    return m.toAddRes, m.toAddError
}

func (m *mockClient) Subtract(x, y int) (int,
error) {
    return y-x, nil
}
```

```
type mockClient struct {
    toAddRes    int
    toAddError  error
    toAddCalled bool
}

func (m *mockClient) ToAdd() (int, error) {
    m.toAddCalled = true
    return m.toAddRes, m.toAddError
}

func (m *mockClient) Subtract(x, y int) (int,
error) {
    return y-x, nil
}
```

Our test double. It implements addRemover

```
type mockClient struct {
    toAddRes    int
    toAddError  error
    toAddCalled bool
}

func (m *mockClient) ToAdd() (int, error) {
    m.toAddCalled = true
    return m.toAddRes, m.toAddError
}

func (m *mockClient) Subtract(x, y
error) {
    return y-x, nil
}
```

We control the behavior

```
type mockClient struct {
    toAddRes    int
    toAddError  error
    toAddCalled bool
}

func (m *mockClient) ToAdd() (int, error) {
    m.toAddCalled = true
    return m.toAddRes, m.toA
}

func (m *mockClient) Subtract(x, y int) (int,
error) {
    return y-x, nil
}
```

```go
func TestAddConst(t *testing.T) {
    client := mockClient{toAddRes: 25, toAddError: nil}
    res, err := AddRemove(&client, 5, 4)
    if err != nil {
        t.Fatalf("got error %v but wasn't expecting", err)
    }
    if res != 30 {
        t.Fatalf("got %d but was expecting 30", res)
    }
    if !client.toAddCalled {
        t.Fatalf("client.ToAdd() was not called")
    }
}
```

```go
func TestAddConst(t *testing.T) {
    client := mockClient{toAddRes: 25, toAddError: nil}
    res, err := AddRemove(&client, 5, 4)
    if err != nil {
        t.Fatalf("got error %v but                    err)
    }
    if res != 30 {
        t.Fatalf("got %d but was expecting 30", res)
    }
    if !client.toAddCalled {
        t.Fatalf("client.ToAdd() was not called")
    }
}
```

The double of our dependency is injected

```go
func TestAddConst(t *testing.T) {
    client := mockClient{toAddRes: 25, toAddError: nil}
    res, err := AddRemove(&client, 5, 4)
    if err != nil {
        t.Fatalf("got error %v but wasn't expecting", err)
    }
    if res != 30 {
        t.Fatalf("got %d but was expecting 30", res)
    }
    if !client.toAddCalled {
        t.Fatalf("client.ToAdd(
    }
}
```

We can assert the probes

Dependency injection

is about injecting dependencies

With dependency injection

- the injected object can be an object or a function
- the dependecy can be injected to an object or a function
- the dependency can be hierarchical

```go
package localmath

type Math struct {
  client cloudmath.Client
}

func New() *Math {
  return &Math{client: cloudmath.NewClient()}
}
```

```go
package localmath

type Math struct {
  client cloudmath.Client
}

func New() *Math {
  return &Math{client: cloudmath.NewClient()}
}
```

We need to inject it

```
package localmath

type Math struct {
  client AddRemover
}

func New(c AddRemover) *Math {
    return &Math{client: c}
}
```

TOO MUCH WORK?

github.com/stretchr/testify
github.com/vektra/mockery

DO WE ALWAYS NEED MOCKS?

MOCKS BUT NOT REALLY

Working with the filesystem

- use real files under testdata
- use afero

Afero is a filesystem framework providing a simple, uniform and universal API interacting with any filesystem, as an abstraction layer providing interfaces, types and methods.

Working with the filesystem

- use real files under testdata
- use afero

```
var AppFs = afero.NewMemMapFs()
var AppFs = afero.NewOsFs()

AppFs.Open("/tmp/foo")
```

Afe form and abstraction layer providing interfaces, types and methods.

Networking

- Consider spinning up the server as part of the test
  - Custom protocol
  - Http test
  - Grpc
  - K8s tests

```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()
  }
}
```

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

  svr := httptest.NewServer(http.Handl                    onseWriter,
                                                          r *http.Request) {
    uu := []users.User{{"foo", 12}, {"bar", 13}}

  svr := httptest.NewServer(http.HandlerFunc(func(w
http.ResponseWriter, r *http.Request) {

    uu := []users.User{{"foo", 12}, {"bar", 13}}
    json.NewEncoder(w).Encode(uu)

  }))


  }
}
```

Runs a real server!

ALMOST-INTEGRATION TESTS

Makes sense when

- We interact with a single external component
- It's relatively easy to start it up
- When the interaction is low level enough that writing a mock is risky

```
type Database interface {
    Query(query string, args ...interface{}) (int, error)
}

func getQuantity(db Database, id int) (int, error) {
    // Query for a value based on a single row.
    res, err := db.Query("SELECT quontity from
                bucket where id = ?", id)
    if err != nil {
        return 0, fmt.Errorf("getQuantity %d: %v", id,
err)
    }
    return res, nil
}
```

LET'S WRITE A MOCK!

```go
type MockDB struct {
    result int
    err    error
}

func (m *MockDB) Query(query string,
                       args ...interface{}) (int, error) {
    return m.result, m.err
}
```

```go
func TestGetQuantity(t *testing.T) {
	m := &MockDB{result: 10, err: nil}
	q, _ := GetQuantity(m, 47)
	if q != 10 {
		t.Errorf("Expected 10, got %d", q)
	}
}
```

```go
func TestGetQuantity(t *testing.T) {
      m := &MockDB{result: 10, err: nil}
      q, _ := GetQuantity(m, 47)
      if q != 10 {
          t.Errorf("Expected 10, got %d", q)
      }
  }
```

```
$ go test
PASS
ok      github.com/fedepaol/gotests/pkg/db    0.002s
```

```
type Database interface {
    Query(query string, args ...interfac
}

func getQuantity(db Database, id int) (ir , error) {
    // Query for a value based on a single row.
    res, err := db.Query("SELECT quontity from
                bucket where id = ?", id)
    if err != nil {
        return 0, fmt.Errorf("getQuantity %d: %v", id,
err)
    }
    return res, nil
}
```

There was a syntax error!

Use testcontainers or dockertest

- Run the dependency (redis, mysql) in a container
- Hooks to wait for the process to be ready
- Validate the syntax against the real thing

github.com/ory/dockertest

github.com/testcontainers/testcontainers-go

```go
func TestMain(m *testing.M) {
    pool, err := dockertest.NewPool("")
    err = pool.Client.Ping()

    resource, err := pool.Run("mysql", "5.7",
            []string{"MYSQL_ROOT_PASSWORD=secret"})

    if err := pool.Retry(func() error {
        var err error
        db, err = sql.Open("mysql", ...)
        if err != nil {
            return err
        }
        return db.Ping()
    }); err != nil {
        log.Fatalf("Could not connect to database: %s", err)
    }
    m.Run()
    //
}
```

```
func TestMain(m *testing.M) {
    pool, err := dockertest.NewPool("")
    err = pool.Client.Ping()

    pool, err := dockertest.NewPool("")
    err = pool.Client.Ping()

    resource, err := pool.Run("mysql", "5.7",
            []string{"MYSQL_ROOT_PASSWORD=secret"})

            return err
        }
        return db.Ping()
    }); err != nil {
        log.Fatalf("Could not connect to database: %s", err)
    }
    m.Run()
    //
}
```

```
func TestMain(m *testing.M) {
    pool, err := dockertest.NewPool("")

    if err := pool.Retry(func() error {
        var err error
        db, err = sql.Open("mysql", ...)
        if err != nil {
            return err
        }
        return db.Ping()
    }); err != nil {
        log.Fatalf("Could not connect to database: %s", err)
    }

    //
}
```

Using docker tests / container tests

- Quicker than waiting for integration to happen
- Depends on external state - risk of flakes / test status contamination
- Useful in very specific use cases

WRAPPING UP

Wrapping up

- Go has testing superpowers
- We learned a few go tests tricks
- We learned how to isolate our dependency to make our tests unit tests
- We learned how to cross the “unit” boundaries in some cases

Unfortunately, Edit and Pray is pretty much the industry standard.

Thanks!

Any questions?

@fedepaol

hachyderm.io/@fedepaol

fedepaol@gmail.com

fpaoline@redhat.com

Slides at: speakerdeck.com/fedepaol