Efficient Integration Testing in Go
A Case Study on Dapr

Josh van Leeuwen
Agenda

- TESTING
- DAPR
- FRAMEWORK
- NAMING (hard)
- PROCESS
  - (wrap)
  - bin
  - p|pe
- Assert eventually
- CLEANUP (really)
- OS
- Being Productive
TESTING - aka why are we here?, 42 etc.
Why do we test software?

1. Prove correctness of software
2. Guardrails when changing implementation code
3. Ensure compatibility with external changing modules/APIs
4. Verify performance
5. Provide a framework for finding bugs and experimenting with features
6. Increase velocity of development
Test Types

- **Unit**
- **Integration**
- **End to End**
- **Performance, Smoke, Manual**
Integration Testing

- Setup System State
- Assert Behavior
- Cleanup System State
Integration Testing

./exec daprd

curl

localhost:9090/healthz

kill 1234
DAPR
Dapr

- **Pod**
  - Actor partition placement
  - Dapr runtime injector
  - Cert authority and identity
  - Update component changes
  - My App
  - State Stores
  - Pub/Sub Brokers
  - Secret Stores
  - Bindings & Triggers
  - Observability

- **Kubernetes**
  - Operator
    - Deploys and manages Dapr
  - Pod
    - Placement
    - Actor partition placement
    - Dapr runtime injector
    - Cert authority and identity
    - Update component changes
    - My App

- **Kubelet**
  - Use components

- **Dapr Components**
  - Any cloud or edge infrastructure
  - Microsoft Azure
  - AWS
  - Alibaba Cloud
  - Google Cloud

- **Observability**
  - Readiness and Liveness probe on healthz API to determine Dapr health state
Dapr Integration Design Decisions

1. Go as sole dependency
2. “Quick” to execute (time.Sleep is banned..ish)
3. Portable
4. Extensible
5. Readable
FRAMEWORK
Integration Testing

Setup System State → Assert Behavior → Cleanup System State
Integration Testing

- ([Process].Run)
- (TestCase).Run
- ([Process].Cleanup)
Test Case

- (TestCase).Setup
  - ([Process]).Run
  - (TestCase).Run
  - ([Process]).Cleanup
Test Case

// Case is a test case for the integration test suite.

type Case interface {
    Setup(*testing.T) []framework.Option
    Run(*testing.T, context.Context)
}
func init() {
    suite.Register(new(base))
}

type base struct {
}

func (b *base) Setup(t *testing.T) []framework.Option {
    return []framework.Option{
        framework.WithProcesses(noop.New())
    }
}

func (b *base) Run(t *testing.T, ctx context.Context) []
    assert.Noop(t)
Framework Process

(TestingCase).Setup

(TestingCase).Run

([]Process).Run

([]Process).Cleanup
Framework Process

```go
defualt Interface interface {
    // Run runs the process.
    Run(*testing.T, context.Context)

    // Cleanup cleans up the process.
    Cleanup(*testing.T)
}
```
type NOOP struct {
    foo bool
}

func New(t *testing.T, fopts ...Option) *HTTP {
    t.Helper()

    var opts options
    for _, fopt := range fopts {
        fopt(&opts)
    }

    return &NOOP{
        foo: opts.foo,
    }
}

func (n *NOOP) Run(t *testing.T, ctx context.Context) {
    require.NoError(t,
        os.WriteFile(
            filepath.Join(t.TempDir(), "test.txt"),
            []byte("hello"),
            0600,
        ),
    )
}

(type Option func(*options))

(type options struct {
    foo bool
})

func WithFoo(foo bool) Option {
    return func(o *options) {
        o.foo = foo
    }
}

([]Process).Run
([]Process).Cleanup
func Run(t *testing.T, ctx context.Context, opts ...Option) {
    t.Helper()

    o := options{}
    for _, opt := range opts {
        opt(&o)
    }

    t.Logf("starting %d processes", len(o.procs))

    for i, proc := range o.procs {
        i := i
        proc.Run(t, ctx)
        t.Cleanup(func() { o.procs[i].Cleanup(t) })
    }
}
Integration Suite

Runner

Test Cases
Integration Suite

```go
func RunIntegrationTests(t *testing.T) {
    for _, tcase := range testSuite {
        t.Run(tcase.Name(), func(t *testing.T) {
            options := tcase.Setup(t)
            framework.Run(t, ctx, options...)
            tcase.Run(t, ctx)
        })
    }
}
```
Case Registry

Integration Suite Runner

Test Cases
import (  
   "github.com/dapr/dapr/tests/integration/suite/daprd/base"
 )

for _, tcase := range testSuite {
   func init() {
      suite.Register(new(base))
   }
}

Case Registry
NAMING (hard)
Test Naming

Follow idiomatic Go-

- Meaning derived through context
- Hierarchical
- Less is more
- lowercase, no underscores etc.
Test Naming

Follow idiomatic Go-
   · Meaning derived through context
   · Hierarchical
   · Less is more
   · lowercase, no underscore etc.

Reflect Magic!
// All returns all registered test cases.
// The returned slice is sorted by name.
func All(t *testing.T) []NamedCase {
    all := make([]NamedCase, len(cases))
    for i, tcase := range cases {
        tof := reflect.TypeOf(tcase).Elem()
        _, aft, ok := strings.Cut(tof.PkgPath(), "tests/integration/suite/")
        require.True(t, ok)
        name := aft + "/" + tof.Name()
        all[i] = NamedCase{name, tcase}
    }

    sort.Slice(all, func(i, j int) bool {
        return all[i].name < all[j].name
    })

    return all
}
Test Naming - Reflect Magic!

TestName = packagePath + structName

```go
func init() {
  suite.Register(new(base))
}

type base struct {
}

func (b *base) Setup(t *testing.T) []framework.Option {
  return []framework.Option{
    framework.WithProcesses(noop, New()),
  }
}

func (b *base) Run(t *testing.T, ctx context.Context) {
  assert.Noopt(t)
}
```

```
// All returns all registered test cases.
// The returned slice is sorted by name.
func all(t *testing.T) []NamedCase {
  all := make([]NamedCase, 0, len(all))
  for i, case := range cases {
    if t.Name().Equals(cases[i].Name.Name()) {
      all = append(all, cases[i])
    }
  }
  return all
}

Test_Integration/dapr/dfoo/bas
```
Test Naming – Why?
Test Naming - Focus!

$ go test --focus "actors|placement"
Test Naming - Focus with regexp

$ go test --focus "actors|placement"

```go
focusedTests := make([][]suite.NamedCase, 0)
focus, err := regexp.Compile(*focusF)
require.NoError(t, err)
for _, tcase := range suite.All(t) {
    if !focus.MatchString(tcase.Name()) {
        continue
    }
    focusedTests = append(focusedTests, tcase)
}
```
Test Naming - Focus!

$ go test -v -focus sentry
PROCESS - w(rap)
Process - w(rap)
Process - w(rap) - exec

(exec).Run

(exec).Cleanup

exec.CommandContext() ...

cmd.Process.Signal() && cmd.Wait() ...
Process - w(rap) - Unix Philosophy
return []framework.Option{
    framework.WithProcesses(sentry,
        w.operatorCreate, w.operatorUpdate, w.operatorDelete,
        w.loglineCreate, w.loglineUpdate, w.loglineDelete,
        w.daprdCreate, w.daprdUpdate, w.daprdDelete,
    ),
}

func Run(t *testing.T, ctx context.Context, opts ...Option) {
    t.Helper()
    o := options{}
    for _, opt := range opts {
        opt&o
    }
    t.Logf("starting %d processes", len(o.procs))
    for i, proc := range o.procs {
        i := i
        proc.Run(t, ctx)
        t.cleanupFunc() { o.procs[i].Cleanup(t) }
    }
}
PROCESS - bin
Build from Source
Build from Source

Go’s build cache is very powerful.
func Build(t *testing.T, name string) {
    t.Helper()
    if _, ok := os.LookupEnv(EnvKey(name)); !ok {
        t.Logf("%q not set, building %q binary", EnvKey(name), name)
    }

    _, tfile, _, ok := runtime.Caller(0)
    require.True(t, ok)
    rootDir := filepath.Join(filepath.Dir(tfile), "../..../..")

    // Use a consistent temp dir for the binary so that the binary is cached on
    // subsequent runs.
    binPath := filepath.Join(os.TempDir(), "dapr_integration_tests/"+name)
    if runtime.GOOS == "windows" {
        binPath += ".exe"
    }

    iowut := iowriter.New(t, name)
    iowerr := iowriter.New(t, name)

    t.Logf("Root dir: %q", rootDir)
    t.Logf("Compiling %q binary to: %q", name, binPath)
    cmd := exec.Command("go", "build", "-tags=allcomponents", "-v", "-o", binPath,
    cmd.Dir = rootDir
    cmd.Stdout = iowut
    cmd.Stderr = iowerr
    // Ensure CGO is disabled to avoid linking against system libraries.
    cmd.Env = append(os.Environ(), "CGO_ENABLED=0")
    require.NoError(t, cmd.Run())

    require.NoError(t, iowut.Close())
Build from Source- Go Cache Magic!

```
$ ls -1 /tmp/dapr_integration_tests
daprd
operator
placement
sentry
```
Build from Source- Go Cache Magic!

$ go test -v -focus sentry

$ ls -l /tmp/dapr_measurement_tests
  daprd
  operator
  placement
  sentry
Version Skew Tests
Process – Pipe

- Software writes messages to “logs”
- These can be noisy
- This fills disk space
- This makes it impossible to read test output
Process - Pipe

Capture exec pipes to in-memory buffers

```go
stdout: iowriter.New(t, filepath.Base(binPath)),
stderr: iowriter.New(t, filepath.Base(binPath)),
```
Process - P|pe

Only write logs to test output when it matters- when the test fails!

```go
// flush writes the buffer to the test logger. Expects the lock to be held
// before calling.
func (w *stowriter) flush() {
    w.lock.Lock()
    defer w.lock.Unlock()
    defer w.buf.Reset()

    // Don't log if the test hasn't failed and the user hasn't requested logs to
    // always be printed.
    if !w.t.Failed() &&
        !utils.IsTruthy(os.Getenv("DAPR_INTEGRATION_LOGS")) {
        return
    }

    for {
        line, err := w.buf.ReadByteSlice('
')
        if len(line) > 0 {
            w.t.Log(w.t.Name() + "" + w.procName + "" + strings.TrimSuffix(string(line), "\n")
            if err != nil {
                if !errors.Is(err, io.EOF) {
                    w.t.Log(w.t.Name() + "" + w.procName + "" + err.Error())
                } else {
                    break
                }
            }
        }
    }
```
Process - P|pe

You can even test against the output!
(logline is yet another process)

```go
i.logline = logline.New(t,
    logline.WithStdoutLineContains(
        "Blocking graceful shutdown for 2s or until app reports unhealthy...",
        "Block shutdown period expired, entering shutdown...",
        "Daprd shutdown gracefully",
    ),
)

daprd.WithAppHealthThreshold(1),
daprd.WithExecOptions(exec.WithStdout(i.logline.Stdout())),
daprd.WithResourceFiles()
```
Assert eventually
Assert eventually

- All software is eventually consistent
- Asserting behaviour requires waiting for it to complete
- *We have* to wait for that process to complete to observe it, however
- *Never* use `time.Sleep` ...
Assert eventually

time.Sleep is nuclear

· If a single test sleeps for 5 seconds
· CI runs 4 times a day
· This equates to **2 hours** of idle CPU a year...

· Dapr (currently) has 133 integration tests
· If just 10% of those tests Sleep for 5 seconds
· This equates to **more than an entire day** of idle CPU a year...

· Think of the polar bears (and developers...)
Assert eventually

- Use polling (eventually) with short intervals to assert behaviour
- testify is your friend

```
"github.com/stretchr/testify/assert"
"github.com/stretchr/testify/require"
```

```go
assert.EventuallyWithT(t, func(c *assert.CollectT) {
    resp := util.GetMetaComponents(c, ctx, client, u.daprd1.HTTPPort())
    assert.ElementsMatch(c, []*rtv1.RegisteredComponents{
        {Name: "uppercase", Type: "middleware.http.routeralias", Version: "v1"},
        {Name: "uppercase2", Type: "middleware.http.routeralias", Version: "v1"},
    }, resp)
}, time.Second*5, time.Millisecond*100, "expected components are loaded")
```
CLEANUP (really)
CLEANUP (really)

Tests should never leak

- As test case number increases, so too could resource consumption
- Every test should be given a clean “sandbox” to run in

- Leaking state between tests gives breaks
CLEANUP (really)

Some of your friends include:

- t.TempDir()
- t.Cleanup()
- Port 0
- In-Memory state stores/sqlite
- No dependency on the Internet
- cmd.Wait()
- Functions should not receive stop channels
- ❤️ context.Context ❤️
OS

(i use nixos btw)
OS - oh cool, I use NixOS actually

Some operating systems are **weird**
Use build flags where you can, and work through the pain.

```go
//go:build !windows
// +build !windows
```

```go
if runtime.GOOS == "windows" {
  binPath += ".exe"
}
```

```go
if runtime.GOOS == "windows" {
  return !strings.Contains(err.Error(), "An existing connection was forcibly closed by the remote host.")
}
```
Being Productive
Being Productive

- Building a culture of integration testing in a distributed team is always a WIP
- A good testing framework should be usable as a feature development sandbox
- The more higher-order your Processes are the more productive writing tests (and features/experiments) your team will be

```go
func init() {
    suite.Register(new(gpt))
}

type gpt struct {
    daprd *daprd.Daprd
}

func (g *gpt) Setup(t *testing.T) []framework.Option {
    g.daprd = daprd.New(t,
        daprd.WithGPT(daprd.LMAO),
    )
    return []framework.Option{
        framework.WithProcesses(g daprd),
    }
}

func (g *gpt) Run(t *testing.T, ctx context.Context) {
    g.daprd.GRPCClient(t), GPTAsAService(t, ctx)
}
```
Efficient Integration Testing in Go
A Case Study on Dapr
Josh van Leeuwen