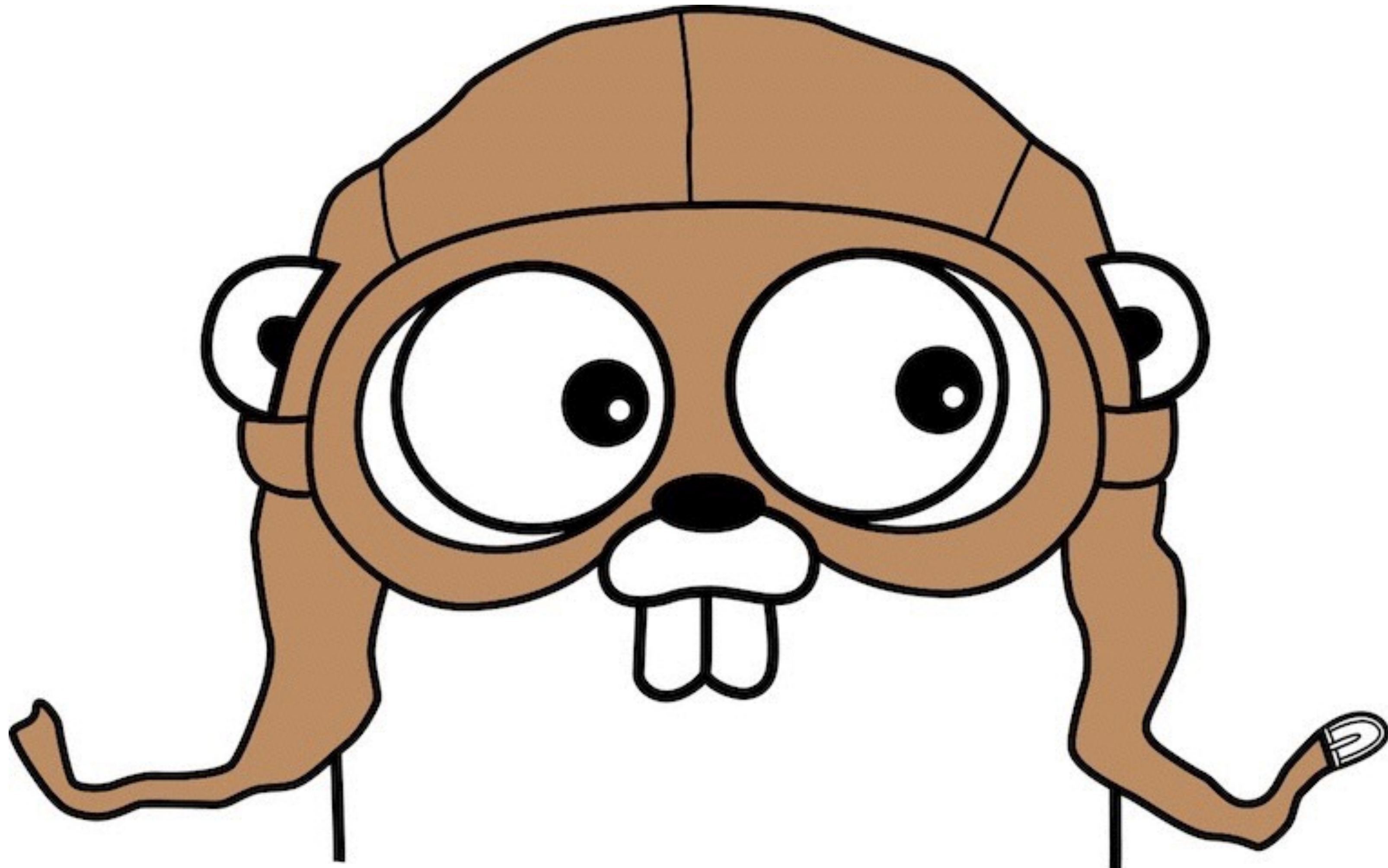


ADVANCED TESTING WITH GO

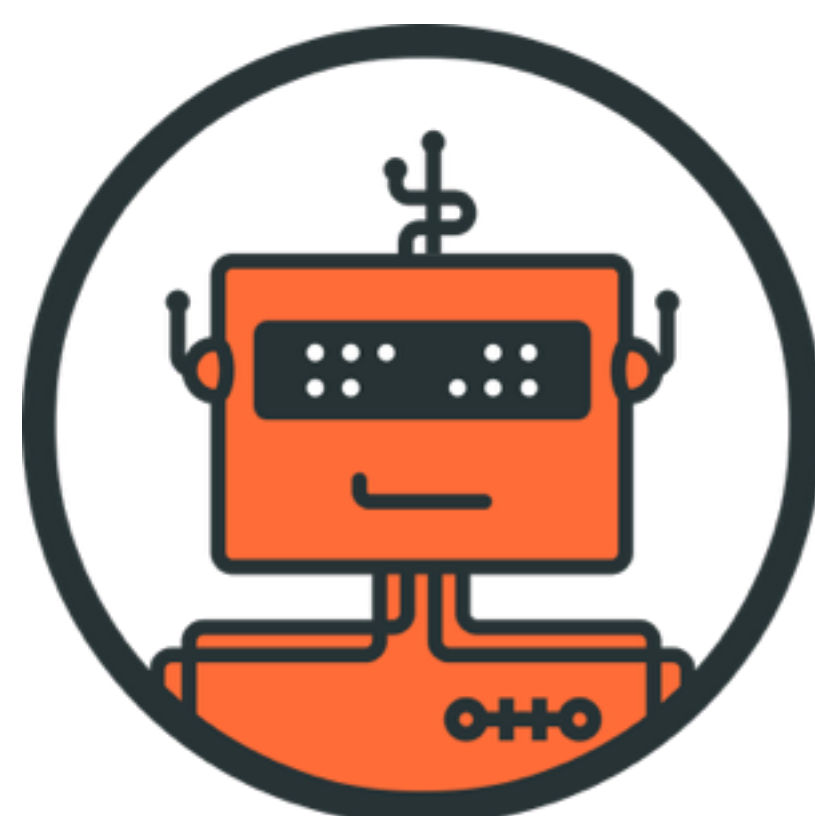
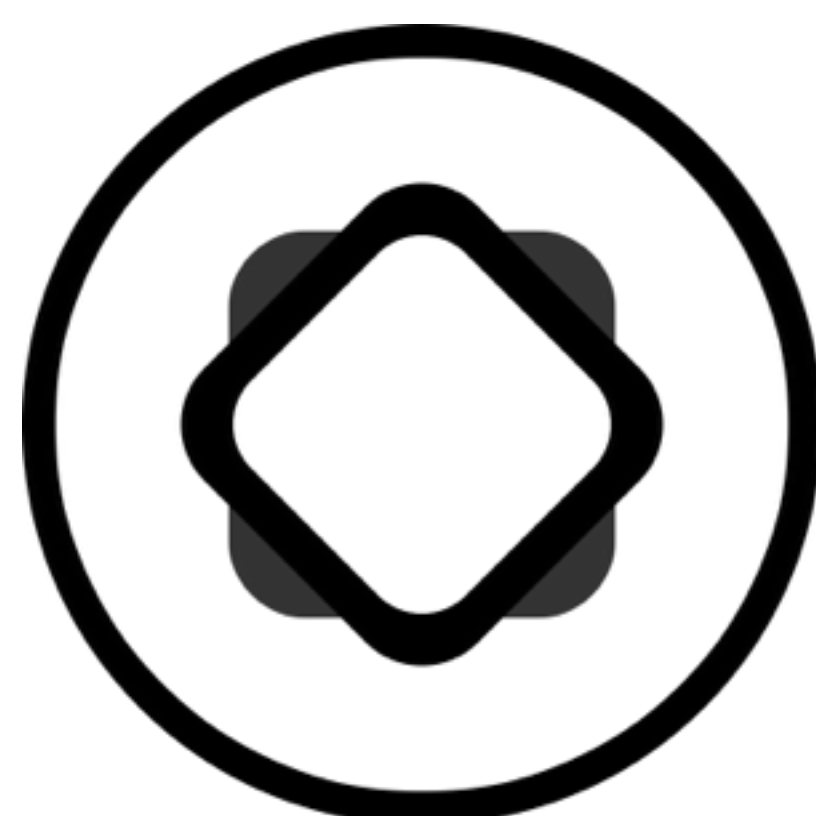
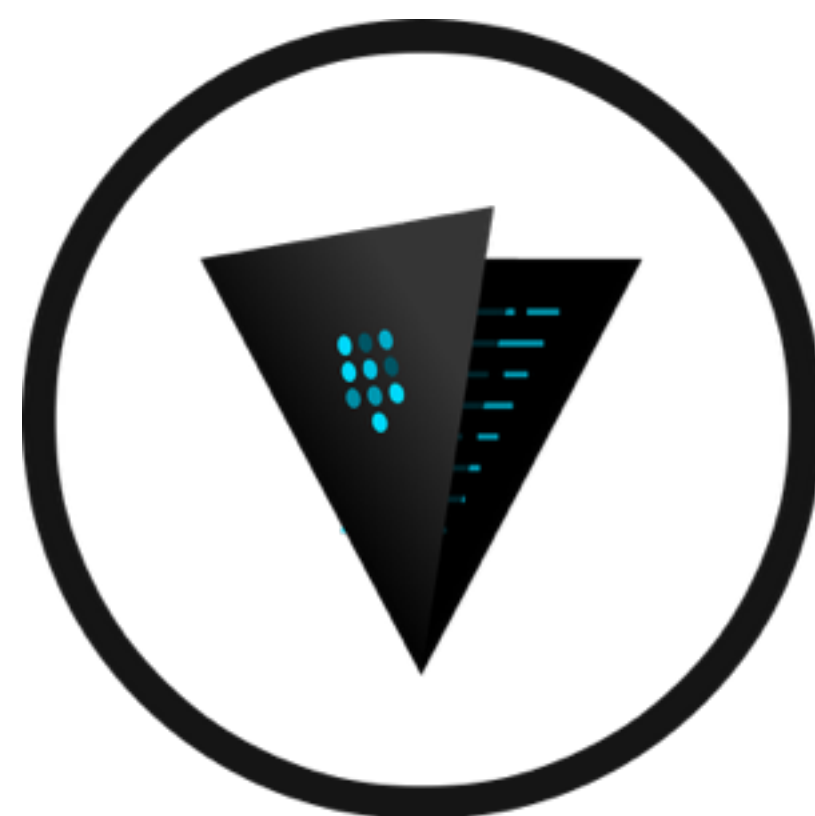
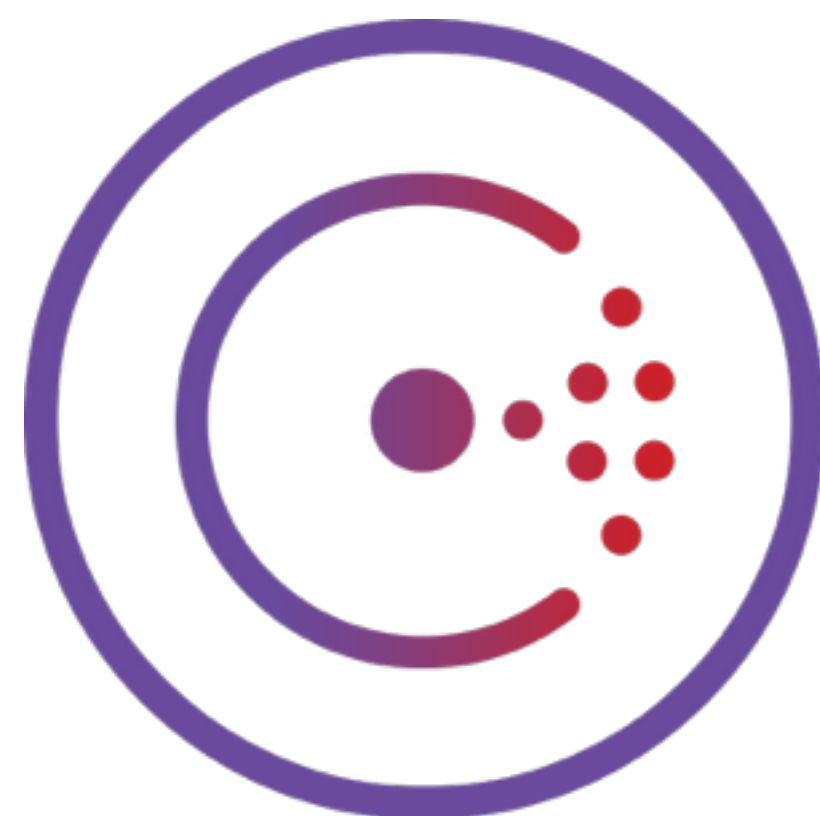


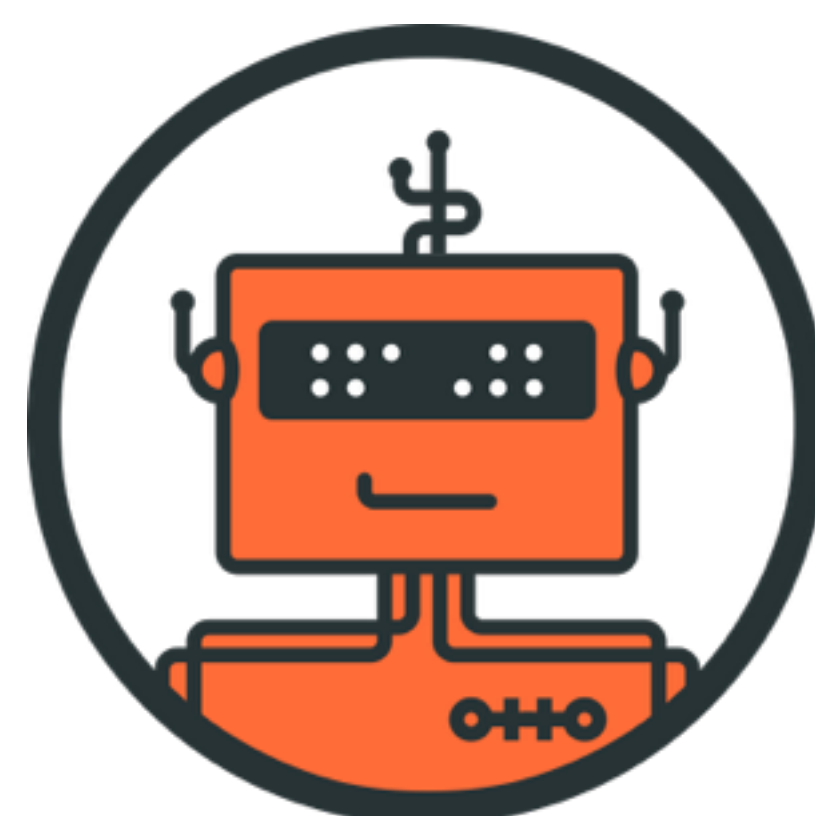
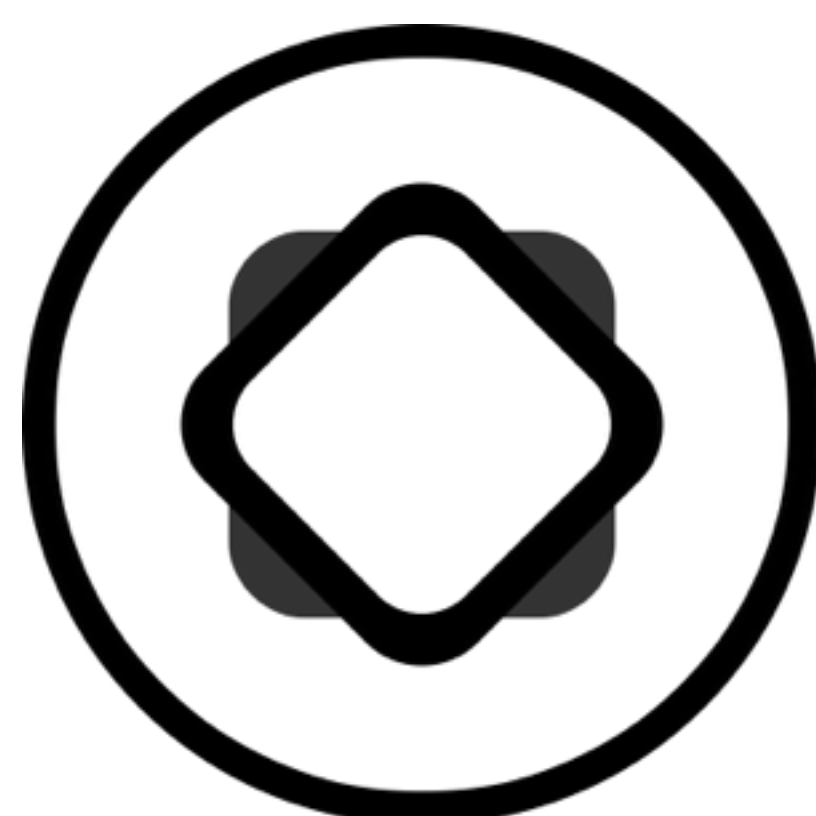
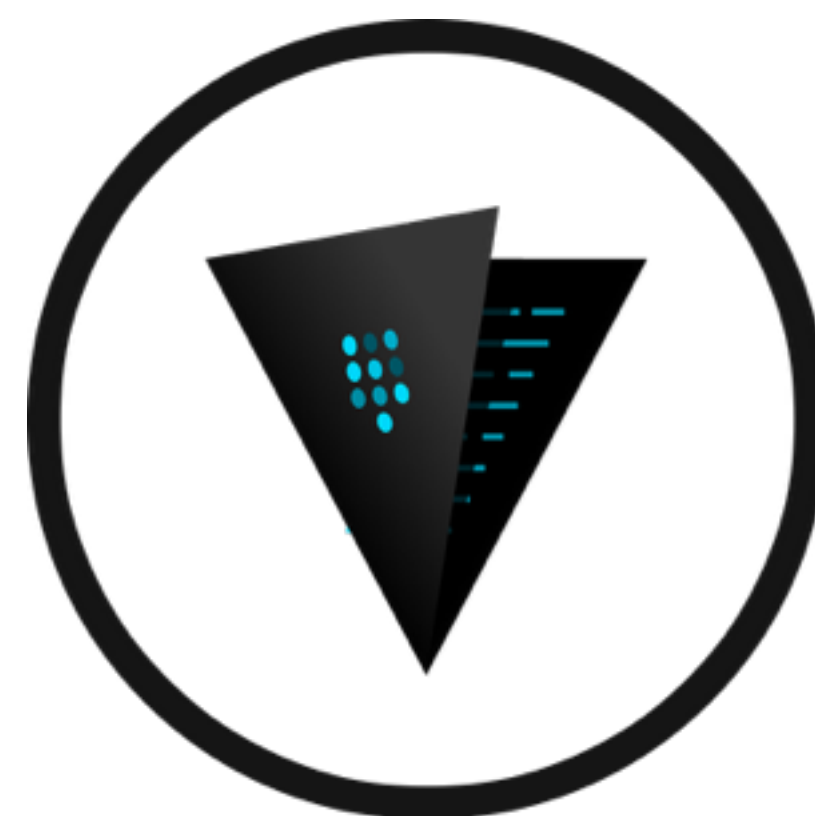
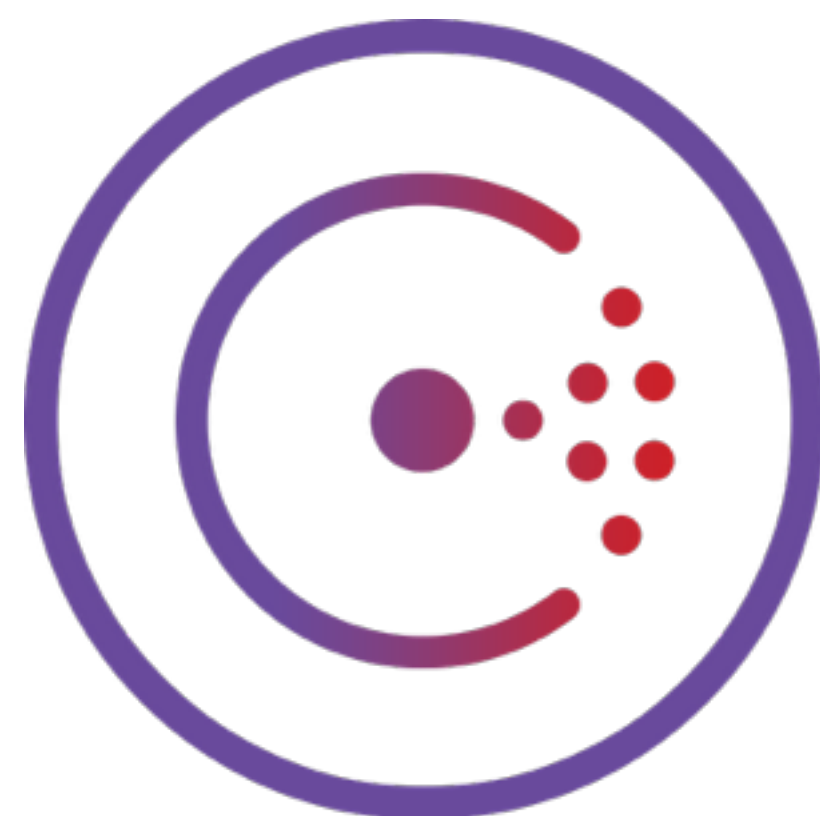


Mitchell Hashimoto

@mitchellh







HASHICORP GO

- ◆ Primary language for ~4 years
- ◆ Deployed by millions, plus significantly in enterprise
- ◆ Distributed systems (Consul, Serf, Nomad, etc.)
- ◆ Extreme performance (Consul, Nomad)
- ◆ Security (Vault)
- ◆ Correctness (Terraform, but also Consul, Nomad, Vault)

TWO PARTS OF TESTING

TEST METHODOLOGY

WRITING TESTABLE CODE

TEST METHODOLOGY

SLIDE STYLE!

WRITING TESTABLE CODE

TEST METHODOLOGY

- ✦ Methods to test specific cases
- ✦ Techniques to write better tests
- ✦ A lot more to testing than “assert(func() == expected)”

TESTABLE CODE

- ◆ How to write code that can be tested well and easily
- ◆ Just as important as writing good tests is writing code that can be tested well
- ◆ Many developers that tell me “this can’t be tested” aren’t wrong, they just wrote the code in a way that made it so. We very rarely see cases at HashiCorp that truly can’t be tested [well].
- ◆ Rewriting existing code to be testable is a pain, but worth it

TABLE DRIVEN TESTS

TABLE DRIVEN TESTS

```
func TestAdd(t *testing.T) {  
    cases := []struct{ A, B, Expected int }{  
        { 1, 1, 2 },  
        { 1, -1, 0 },  
        { 1, 0, 1 },  
        { 0, 0, 0 },  
    }  
  
    for _, tc := range cases {  
        actual := tc.A + tc.B  
        if actual != tc.Expected {  
            t.Errorf(  
                "%d + %d = %d, expected %d",  
                tc.A, tc.B, actual, tc.Expected)  
        }  
    }  
}
```


TABLE DRIVEN TESTS

- ✦ Low overhead to add new test cases
- ✦ Makes testing exhaustive scenarios simple
- ✦ Makes reproducing reported issues simple
- ✦ Do this pattern *a lot*
- ✦ Follow pattern even for single cases, if its possible to grow

TABLE DRIVEN TESTS

(CONSIDER NAMING CASES)

```
func TestAdd(t *testing.T) {
    cases := map[string]struct{ A, B, Expected int }{
        "foo": { 1, 1, 2 },
        "bar": { 1, -1, 0 },
    }

    for k, tc := range cases {
        actual := tc.A + tc.B
        if actual != tc.Expected {
            t.Errorf(
                "%s: %d + %d = %d, expected %d",
                k, tc.A, tc.B, actual, tc.Expected)
        }
    }
}
```

TEST FIXTURES

TEST FIXTURES

```
func TestAdd(t *testing.T) {  
    data := filepath.Join("test-fixtures", "add_data.json")  
  
    // ... Do something with data  
}
```

TEST FIXTURES

- ✦ “go test” sets pwd as package directory
- ✦ Use relative path “test-fixtures” directory as a place to store test data
- ✦ Very useful for loading config, model data, binary data, etc.

GOLDEN FILES
(ALSO TEST FLAGS)

GOLDEN FILES

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

func TestAdd(t *testing.T) {
    // ... table (probably!)

    for _, tc := range cases {
        actual := doSomething(tc)
        golden := filepath.Join("test-fixtures", tc.Name+".golden")
        if *update {
            ioutil.WriteFile(golden, actual, 0644)
        }

        expected, _ := ioutil.ReadFile(golden)
        if !bytes.Equal(actual, expected) {
            // FAIL!
        }
    }
}
```

GOLDEN FILES

```
$ go test
```

```
...
```

```
$ go test -update
```

```
...
```

GOLDEN FILES

- ✦ Test complex output without manually hardcoding it
- ✦ Human eyeball the generated golden data. If it is correct, commit it.
- ✦ Very scalable way to test complex structures (write a `String()` method)

GLOBAL STATE

GLOBAL STATE

- ◆ Avoid it as much as possible.
- ◆ Instead of global state, try to make whatever is global a configuration option using global state as the *default*, allowing tests to modify it.
- ◆ If necessary, make global state a var so it can be modified. This is a last case scenario, though.

GLOBAL STATE

```
// Not good on its own  
const port = 1000
```

```
// Better  
var port = 1000
```

```
// Best  
const defaultPort = 1000
```

```
type ServerOpts {  
    Port int // default it to defaultPort somewhere  
}
```

TEST HELPERS

TEST HELPERS

```
func testTempFile(t *testing.T) string {  
    tf, err := ioutil.TempFile("", "test")  
    if err != nil {  
        t.Fatalf("err: %s", err)  
    }  
    tf.Close()  
  
    return tf.Name()  
}
```

TEST HELPERS

- ✦ *Never* return errors. Pass in `*testing.T` and fail.
- ✦ By not returning errors, usage is much prettier since error checking is gone.
- ✦ Used to make tests clear on what they're testing vs what is boilerplate

TEST HELPERS

```
func testTempFile(t *testing.T) (string, func()) {  
    tf, err := ioutil.TempFile("", "test")  
    if err != nil {  
        t.Fatalf("err: %s", err)  
    }  
    tf.Close()  
  
    return tf.Name(), func() { os.Remove(tf.Name()) }  
}  
  
func TestThing(t *testing.T) {  
    tf, tfclose := testTempFile(t)  
    defer tfclose()  
}
```

TEST HELPERS

```
func testChdir(t *testing.T, dir string) func() {  
    old, err := os.Getwd()  
    if err != nil {  
        t.Fatalf("err: %s", err)  
    }  
  
    if err := os.Chdir(dir); err != nil {  
        t.Fatalf("err: %s", err)  
    }  
  
    return func() { os.Chdir(old) }  
}  
  
func TestThing(t *testing.T) {  
    defer testChdir(t, "/other")()  
  
    // ...  
}
```

TEST HELPERS

- ✦ Returning a func() for cleanup is an elegant way to hide that
- ✦ The func() is a closure that can have access to *testing.T to also fail
- ✦ Example: testChdir proper setup/cleanup would be at least 10 lines without the helper. Now avoids that in all our tests.

PACKAGE/FUNCTIONS

PACKAGE/FUNCTIONS

- ◆ Break down functionality into packages/functions judiciously
- ◆ NOTE: Don't overdo it. Do it where it makes sense.
- ◆ Doing this correctly will aid testing while also improving organization. Over-doing it will complicate testing and readability.
- ◆ Qualitative, but practice will make perfect.

PACKAGE/FUNCTIONS

- ◆ Unless the function is *extremely* complex, we try to test only the exported functions, the exported API.
- ◆ We treat unexported functions/structs as implementation details: they are a means to an end. As long as we test the end and it behaves within spec, the means don't matter.
- ◆ Some people take this too far and choose to *only* integration/acceptance test, the ultimate “test the end, ignore the means.” We disagree with this approach.

NETWORKING

NETWORKING

- ✦ Testing networking? Make a real network connection.
- ✦ Don't mock `net.Conn`, no point.

NETWORKING

```
// Error checking omitted for brevity
func TestConn(t *testing.T) (client, server net.Conn) {
    ln, err := net.Listen("tcp", "127.0.0.1:0")

    var server net.Conn
    go func() {
        defer ln.Close()
        server, err = ln.Accept()
    }()

    client, err := net.Dial("tcp", ln.Addr().String())
    return client, server
}
```

NETWORKING

- ✦ That was a one-connection example. Easy to make an N-connection.
- ✦ Easy to test any protocol.
- ✦ Easy to return the listener as well.
- ✦ Easy to test IPv6 if needed.
- ✦ Why ever mock `net.Conn`? (Rhetorical, for readers)

CONFIGURABILITY

CONFIGURABILITY

- ◆ Unconfigurable behavior is often a point of difficulty for tests.
 - ◆ Example: ports, timeouts, paths
- ◆ Over-parameterize structs to allow tests to fine-tune their behavior
- ◆ It is okay to make these configurations unexported so only tests can set them.

CONFIGURABILITY

```
// Do this, even if cache path and port are always the same
// in practice. For testing, it lets us be more careful.
type ServerOpts struct {
    CachePath string
    Port      int
}
```

SUBPROCESSING

SUBPROCESSING

- ✦ Subprocessing is typical a point of difficult-to-test behavior.
- ✦ Two options:
 1. Actually do the subprocess
 2. Mock the output or behavior

SUBPROCESSING: REAL

- ✦ Actually executing the subprocess is nice
- ✦ Guard the test for the existence of the binary
- ✦ Make sure side effects don't affect any other test

SUBPROCESSING: REAL

```
var testHasGit bool

func init() {
    if _, err := exec.LookPath("git"); err == nil {
        testHasGit = true
    }
}

func TestGitGetter(t *testing.T) {
    if !testHasGit {
        t.Log("git not found, skipping")
        t.Skip()
    }

    // ...
}
```

SUBPROCESSING: MOCK

- ✦ You *still actually execute*, but you're executing a *mock*!
- ✦ Make the `*exec.Cmd` configurable, pass in a custom one
- ✦ Found this in the stdlib, it is how they test os/exec!
- ✦ How HashiCorp tests go-plugin and more

SUBPROCESSING: MOCK

GET THE *EXEC.CMD

```
func helperProcess(s ...string) *exec.Cmd {
    cs := []string{"-test.run=TestHelperProcess", "--"}
    cs = append(cs, s...)
    env := []string{
        "GO_WANT_HELPER_PROCESS=1",
    }

    cmd := exec.Command(os.Args[0], cs...)
    cmd.Env = append(env, os.Environ()...)
    return cmd
}
```

SUBPROCESSING: MOCK

WHAT IT EXECUTES

```
func TestHelperProcess(*testing.T) {  
    if os.Getenv("GO_WANT_HELPER_PROCESS") != "1" {  
        return  
    }  
    defer os.Exit(0)  
  
    args := os.Args  
    for len(args) > 0 {  
        if args[0] == "--" {  
            args = args[1:]  
            break  
        }  
  
        args = args[1:]  
    }  
}
```

SUBPROCESSING: MOCK

WHAT IT EXECUTES

...

```
cmd, args := args[0], args[1:]  
switch cmd {  
case "foo":  
    // ...
```

INTERFACES

INTERFACES

- ◆ Interfaces are mocking points.
- ◆ Behavior can be defined regardless of implementation and exposed via custom framework or testing.go (covered elsewhere)
- ◆ Similar to package/functions: do this judiciously, but overdoing it will complicate readability.

TESTING AS A PUBLIC API

TESTING AS A PUBLIC API

- ◆ Newer HashiCorp projects have adopted the practice of making a “testing.go” or “testing_*.go” files.
- ◆ These are exported APIs for the sole purpose of providing mocks, test harnesses, helpers, etc.
- ◆ Allows other packages to test using our package without reinventing the components needed to meaningful use our package in a test.

TESTING AS A PUBLIC API

- ◆ Example: config file parser
 - ◆ TestConfig(t) => Returns a valid, complete configuration for tests
 - ◆ TestConfigInvalid(t) => Returns an invalid configuration

TESTING AS A PUBLIC API

- ◆ Example: API server
 - ◆ TestServer(t) (net.Addr, io.Closer) => Returns a fully started in-memory server (address to connect to) and a closer to close it.

TESTING AS A PUBLIC API

- ◆ Example: interface for downloading files
- ◆ `TestDownloader(t, Downloader)` => Tests all the properties a downloader should have.
- ◆ `struct DownloaderMock{}` => Implements Downloader as a mock, allowing recording and replaying of calls.

CUSTOM FRAMEWORKS

CUSTOM FRAMEWORKS

- ✦ ``go test`` is an incredible workflow tool
- ✦ Complex, pluggable systems? Write a custom framework *within* ``go test``, rather than a separate test harness.
- ✦ Example: Terraform providers, Vault backends, Nomad schedulers

CUSTOM FRAMEWORKS

```
// Example from Vault
func TestBackend_basic(t *testing.T) {
    b, _ := Factory(logical.TestBackendConfig())

    logicaltest.Test(t, logicaltest.TestCase{
        PreCheck: func() { testAccPreCheck(t) },
        Backend:  b,
        Steps: []logicaltest.TestStep{
            testAccStepConfig(t, false),
            testAccStepRole(t),
            testAccStepReadCreds(t, b, "web"),
            testAccStepConfig(t, false),
            testAccStepRole(t),
            testAccStepReadCreds(t, b, "web"),
        },
    })
}
```


CUSTOM FRAMEWORKS

- ✦ “logicaltest.Test” is just a custom harness doing repeated setup/teardown, assertions, etc.
- ✦ Other examples: Terraform provider acceptance tests
- ✦ We can still use `go test` to run them

TIMING-DEPENDENT TESTS

TIMING-DEPENDENT TESTS

```
func TestThing(t *testing.T) {  
    // ...  
  
    select {  
    case <-thingHappened:  
    case <-time.After(timeout):  
        t.Fatal("timeout")  
    }  
}
```

TIMING-DEPENDENT TESTS

- ✦ We don't use "fake time"
- ✦ We just have a multiplier available that we can set to increase timeouts
- ✦ Not perfect, but not as intrusive as fake time. Still, fake time could be better, but we haven't found an effective way to use it yet.

TIMING-DEPENDENT TESTS

```
func TestThing(t *testing.T) {  
    // ...  
  
    timeout := 3 * time.Minute * timeMultiplier  
  
    select {  
    case <-thingHappened:  
    case <-time.After(timeout):  
        t.Fatal("timeout")  
    }  
}
```

PARALLELIZATION

TEST HELPERS

```
func TestThing(t *testing.T) {  
    t.Parallel()  
}
```

PARALLELIZATION

- ✦ Don't do it. Run multiple processes.
- ✦ Makes test failures uncertain: is it due to pure logic but, or race?
- ✦ *OR*: Run tests both with ``-parallel=1`` and ``-parallel=N``
- ✦ We've preferred to just not use parallelization. We use multiple processes and unit tests specifically written to test for races.

THANK YOU!
QUESTIONS?

