Testing Vapor Apps

Gianluca Tranchedone

@gtranchedone

Agenda

1. How the Swift Package Manager Works
2. Setting up Vapor for Testing
3. Testing Views

How the Swift Package Manager Works

Each subfolder in the `Sources` folder is a package

```
.
├── Sources
|   ├── Package_1
|   ├── Package_2
|   └── Package_3
|
.
```

Similarly, each subfolder of the `Tests` folder is a test target

```
.
├── Tests
|   ├── TestTarget_1
|   ├── TestTarget_2
|   └── TestTarget_3
.
```

Packages containing a `main.swift` file is considered an "Executable" Everything else is a "Library"

Vapor Testing Basics

- XCTest for Swift Packages
- Changing Vapor's default template for testing
- Using Droplets for testing
- Testing with DB transactions

XCTest

- Finds test cases within the test targets automatically on macOS
- Needs a `LinuxMain.swift` file that specifies what tests to run on Linux
- Cannot run SwiftPM packages that are "executables"

DEMO

Changing Vapor's default template for testing

DEMO Recap

1. Move the `main.swift` file in a new package called "Executable"
2. Update the `Package.swift` file with targets (see next slides)
3. Create a `LinuxMain.swift` file to run Tests on Linux - e.g. CI (see next slides)
4. Update your server configuration to run the `Executable` target

DEMO Recap (continued)

1. Run the tests using `vapor test` in Terminal or `CMD-U` using the `App.framework` target in Xcode
2. Run `vapor run serve --exec=Executable` or the `Executable` target in Xcode
3. Use a Droplet created for testing if you need to (see next slides)
4. Update your Config with a test configuration

Updated folder structure

```
.
├── Sources
│   ├── App
│   │   ├── Configuration
│   │   ├── Controllers
│   │   └── Models
│   └── Executable
│       └── main.swift
├── Tests
│   ├── AppTests
│   │   ├── Configuration
│   │   ├── Controllers
│   │   └── Models
│   └── LinuxMain.swift
.
```

Updated Package.swift

```swift
import PackageDescription

let package = Package(
    name: "VaporApp",
    targets: [
        Target(name: "Executable", dependencies: ["App"])
    ],
    dependencies: [
        .Package(url: "https://github.com/vapor/vapor.git", majorVersion: 1, minor: 5)
    ],
    exclude: [
        "Config",
        "Database",
        "Localization",
        "Public",
        "Resources",
    ]
)
```

LinuxMain.Swift

```swift
import XCTest
@testable import AppTests

XCTMain([
    testCase(RoutesTests.allTests),
    testCase(PostTests.allTests),
    testCase(PostsControllerTests.allTests),
])
```

Droplet+Tests.swift

```swift
@testable import Vapor
@testable import Console

extension Droplet {

    static func dropletForTests() throws -> Droplet {
        let droplet = Droplet(arguments: ["dummy/path/", "prepare", "-y"],
                              environment: .test)
        return droplet
    }

    func setUp() throws {
        try runCommands()
    }

    func tearDown() throws {
        let p = Prepare(console: console,
                        preparations: preparations,
                        database: database)
        try p.run(arguments: ["--revert"])
    }

}
```

Testing Views

- Testing static views
- Testing interactive views

You can test static views using the test Droplet

```
func testRouteToHomepage() throws {
    let request = try Request(method: .get, uri: "/")
    let response = try droplet.respond(to: request)
    let expectedResponse = try droplet.view.make("welcome", [
        "message": droplet.localization[request.lang, "welcome", "title"]
    ]).makeResponse()
    XCTAssertEqual("\(response.body)", "\(expectedResponse.body)")
    XCTAssertEqual(response.status, Status.ok)
}
```

Interactive views can be tested using one of the many JavaScript testing frameworks

Best practices

- Modular architecture
- Mock dependencies
- Use a test configuration for your DB! (see Droplet+Tests)

```
// Config with PostgreSQL
VaporApp
  └── Config
      ├── development
      │   └── postgresql.json
      ├── production
      │   └── app.json
      ├── test
      │   └── postgresql.json
      .
```

Notes

1. Vapor 2 will come with improved support for testing
2. You might need to build more than once for the build to succeed in Xcode
3. Exceptions thrown by tests aren't really useful by default, better catch them
4. You still want to mock as much as you can
5. Remember to update your `LinuxMain.swift` file and the `allTests` arrays

Too much effort?
I've got you covered!

1. https://github.com/gtranchedone/vapor-testing-template allows to create new Vapor projects set up for testing
2. https://github.com/gtranchedone/VaporGenerators allows to create classes with tests and updates `LinuxMain.swift` automatically

Create a new app with the testing template

```
vapor new 'MyApp'
  --template=gtranchedone/vapor-testing-template
```

Using VaporGenerators

Create classes, tests, and update LinuxMain.swift with ease

```
vapor run generate model Bar
vapor run generate controller BarsController

or

vapor run generate resource bar
```

Resources

- Vapor documentation, of course!
- https://theswiftwebdeveloper.com/configuring-vapor-apps-for-testing-7b89f1a6e6a
- https://github.com/gtranchedone/vapor-testing-template
- https://github.com/gtranchedone/VaporGenerators
- http://swiftwebweekly.com

Swift Web Weekly

swiftwebweekly.com