Unit-Test Pipeline Code

HomeAway

Pipeline-as-code needs tests, too!

Austin Witt
Senior Software Engineer
HomeAway.com, Inc.
awitt@homeaway.com

Unit Tests

HomeAway

What are Unit Tests?

1. Repeatable: You can rerun the same test as many times as you want.

2. Consistent: Every time you run it, you get the same result. (for example: Using threads can produce an inconsistent result)

3. In Memory: It has no "hard" dependencies on anything not in memory (such as file system, databases, network)

4. Fast: It should take less than half a second to run a unit test.

5. Checking one single concern or "use case" in the system: More than one can make it harder to understand what or where the problem is when the problem arises.

6. Implementation-Agnostic: Tests are written to the code's interface and/or contract and assume nothing about the underlying implementation. (Correct code is defined by its contract/behavior and interface; changes in implementation that don't change behavior are not wrong)

From "The Difference Between Unit Testing and Integration Testing"
https://www.typemock.com/unit-tests-integration-tests

Unit Tests

HomeAway

Unit Test Glossary

1. Mock (noun): A “fake” object that implements the interface of a real object but does nothing. Logs all interactions to enable tests to make assertions about what did or should have happened.

2. Mocking (verb): The act of creating a mock of a certain type of object.

3. Stubbing (verb): The act of defining behavior or return value for a specific interaction with a Mock object.

Spock

HomeAway

Spock can unit-test Groovy code:

http://spockframework.org/

Spock Web Console

```
1  import spock.lang.*
2
3  // Hit 'Run Script' below
4  class MyFirstSpec extends Specification {
5    def "let's try this!"() {
6      expect:
7      Math.max(1, 2) == 3
8    }
9  }
```

```
MyFirstSpec
 - let's try this!   FAILED

   Condition not satisfied:

   Math.max(1, 2) == 3
        |          |
        2          false

   at MyFirstSpec.let's try this!(Script1.groovy:7)
```

Pipeline Code

HomeAway

What is pipeline code?

A **D**omain **S**pecific **L**anguage (DSL) on top of Groovy.

1. Jenkinsfile
2. Helper Scripts
3. Pipeline Shared Libraries

HomeAway

Follow Along

github.com/HomeAway/jenkins-spock

“examples” directory

Slides:

bit.ly/2QHzRW9

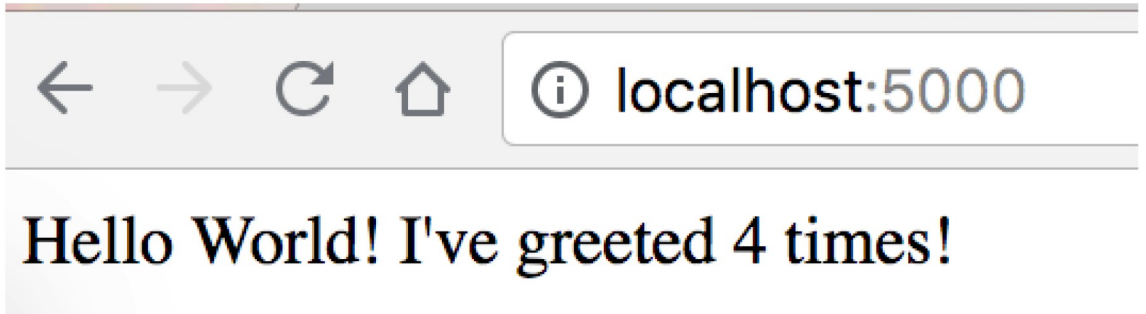
Unit-Test a Jenkinsfile

HomeAway

```
.
├── Dockerfile
├── Jenkinsfile
├── Makefile
├── README.md
└── app
    ├── counter.py
    ├── hello.py
    ├── hello.pyc
    ├── test_counter.py
    └── test_hello.py

1 directory, 9 files
```

```
1  FROM jfloff/alpine-python:3.6
2
3  RUN pip install --upgrade \
4      Flask \
5      pip
6
7  COPY app /app
8
9  WORKDIR app
10 ENV FLASK_APP=hello.py
11 EXPOSE 5000
12
13 ENTRYPOINT ["flask", "run", "--host", "0.0.0.0"]
```



Unit-Test a Jenkinsfile

HomeAway

```
1  def deploy( _env ) {...
19
20 node {
21     stage( "Checkout" ) { checkout scm }
22
23     stage( "Build" ) { sh( "docker build --tag whole-pipeline ." ) }
24
25     stage( "Test" ) {
26         try {
27             sh( "docker run --entrypoint python whole-pipeline -m unittest discover" )
28         } catch( Exception e ) {
29             slackSend(
30                 color: 'error',
31                 message: 'whole-pipeline unit tests failed.' )
32             throw e
33         }
34     }
35
36     stage( "Push" ) { sh( "docker push whole-pipeline" )    }
37
38     stage( "Deploy to TEST" ) { deploy( "test" ) }
39
40     if( BRANCH_NAME == "master" ) {
41         stage( "Deploy to PRODUCTION" ) { deploy( "production" ) }
42     }
43 }
```

Unit-Test a Jenkinsfile

HomeAway

Three Behaviors:

1. Notify Slack when tests fail.

2. Deploy non-master branches to the TEST environment.

3. Deploy *only the master branch* to the PRODUCTION environment.

Three tests to write!

Writing Our First Test

HomeAway

```groovy
import com.homeaway.devtools.jenkins.testing.JenkinsPipelineSpecification

public class JenkinsfileSpec extends JenkinsPipelineSpecification {

    def Jenkinsfile = null

    public static class DummyException extends RuntimeException {
        public DummyException(String _message) { super( _message ); }
    }

    def setup() {
        Jenkinsfile = loadPipelineScriptForTest("/Jenkinsfile")
    }

    def "Slack is notified when tests fail" () {
        setup:
            getPipelineMock("sh")("docker run --entrypoint python whole-pipeline -m unittest discover") >> {
                throw new DummyException("Dummy test failure")
            }
        when:
            try {
                Jenkinsfile.run()
            } catch( DummyException e ) {}
        then:
            1 * getPipelineMock("slackSend")( _ as Map )
    }
}
```

Writing Our First Test

HomeAway

```groovy
import com.homeaway.devtools.jenkins.testing.JenkinsPipelineSpecification

public class JenkinsfileSpec extends JenkinsPipelineSpecification {

    def Jenkinsfile = null

    public static class DummyException extends RuntimeException {
        public DummyException(String _message) { super( _message ); }
    }

    def setup() {
        Jenkinsfile = loadPipelineScriptForTest("/Jenkinsfile")
    }

    def "Slack is notified when tests fail" () {
        setup:
            getPipelineMock("sh")("docker run --entrypoint python whole-pipeline -m unittest discover") >> {
                throw new DummyException("Dummy test failure")
            }
        when:
            try {
                Jenkinsfile.run()
            } catch( DummyException e ) {}
        then:
            1 * getPipelineMock("slackSend")( _ as Map )
    }
}
```

Writing Our First Test

HomeAway

```groovy
import com.homeaway.devtools.jenkins.testing.JenkinsPipelineSpecification

public class JenkinsfileSpec extends JenkinsPipelineSpecification {

    def Jenkinsfile = null

    public static class DummyException extends RuntimeException {
        public DummyException(String _message) { super( _message ); }
    }

    def setup() {
        Jenkinsfile = loadPipelineScriptForTest("/Jenkinsfile")
    }

    def "Slack is notified when tests fail" () {
        setup:
            getPipelineMock("sh")("docker run --entrypoint python whole-pipeline -m unittest discover") >> {
                throw new DummyException("Dummy test failure")
            }
        when:
            try {
                Jenkinsfile.run()
            } catch( DummyException e ) {}
        then:
            1 * getPipelineMock("slackSend")( _ as Map )
    }
}
```

Writing Our First Test

HomeAway

```groovy
import com.homeaway.devtools.jenkins.testing.JenkinsPipelineSpecification

public class JenkinsfileSpec extends JenkinsPipelineSpecification {

    def Jenkinsfile = null

    public static class DummyException extends RuntimeException {
        public DummyException(String _message) { super( _message ); }
    }

    def setup() {
        Jenkinsfile = loadPipelineScriptForTest("/Jenkinsfile")
    }

    def "Slack is notified when tests fail" () {
        setup:
            getPipelineMock("sh")("docker run --entrypoint python whole-pipeline -m unittest discover") >> {
                throw new DummyException("Dummy test failure")
            }
        when:
            try {
                Jenkinsfile.run()
            } catch( DummyException e ) {}
        then:
            1 * getPipelineMock("slackSend")( _ as Map )
    }
}
```

Running Our First Test

HomeAway

Run that specification!

Wait… how?

Maven

Jenkins' components – core and plugins – are distributed as Maven artifacts. So is this test library. That's *very important!*

Running Our First Test

HomeAway

Running Spock with Maven

```
<plugin>
    <groupId>org.codehaus.gmavenplus</groupId>
    <artifactId>gmavenplus-plugin</artifactId>
    <version>${groovy.gmaven.pluginVersion}</version>
    <executions>
        <execution>
            <id>groovy</id>
            <goals>
                <goal>addTestSources</goal>
                <goal>generateTestStubs</goal>
                <goal>compileTests</goal>
                <goal>removeTestStubs</goal>
            </goals>
            <configuration>
                <testSources>
                    <testSource>
                        <directory>src/test/groovy</directory>
                        <includes>
                            <include>**/*.groovy</include>
                        </includes>
                    </testSource>
                </testSources>
            </configuration>
        </execution>
    </executions>
</plugin>
```

```
<testResources>
    <testResource>
        <includes>
            <include>Jenkinsfile</include>
        </includes>
        <directory>${project.basedir}</directory>
    </testResource>
    <testResource>
        <directory>src/test/resources</directory>
    </testResource>
</testResources>
```

Mock Pipeline Steps

```
[ERROR] Slack is notified when tests fail(JenkinsfileSpec)  Time elapsed: 1.238 s  <<< ERROR!
java.lang.IllegalStateException:
There is no pipeline step mock for [sh].
        1. Is the name correct?
        2. Does the pipeline step have a descriptor with that name?
        3. Does that step come from a plugin? If so, is that plugin listed as a dependency in your pom.xml?
        4. If not, you may need to call explicitlyMockPipelineStep('sh') in your test's setup: block.
        at JenkinsfileSpec.Slack is notified when tests fail(JenkinsfileSpec.groovy:18)
```

Pipeline steps come from JARs.
Those JARs come from Maven dependencies.
Only the steps whose JARs you pull in get mocked!

```
<dependency>
    <!-- provides the sh() pipeline step -->
    <groupId>org.jenkins-ci.plugins.workflow</groupId>
    <artifactId>workflow-durable-task-step</artifactId>
    <version>2.21</version>
    <scope>test</scope>
</dependency>
```

Mock Pipeline Steps

HomeAway

Bonus:

pom.xml now contains an explicit list of every Jenkins Plugin that a Jenkins Master needs to install in order to be able to run this project's pipeline!

Well, kind-of. Groovy is interpreted line-by-line, so if unit tests don't touch some code, they might never try to use a pipeline step and you might never know.

Really, you have a list of every plugin needed to be able to do *everything the tests say you can do*.

All the more reason to write a *full* test-suite!

Mock Pipeline Variables

HomeAway

```
[ERROR] Slack is notified when tests fail(JenkinsfileSpec)  Time elapsed: 1.327 s  <<< ERROR!
java.lang.IllegalStateException:
There is no pipeline variable mock for [scm].
        1. Is the name correct?
        2. Is it a GlobalVariable extension point? If so, does the getName() method return [scm]?
        3. Is that variable normally defined by Jenkins? If so, you may need to define it by hand in your Spec.
        4. Does that variable come from a plugin? If so, is that plugin listed as a dependency in your pom.xml?
        5. If not, you may need to call explicitlyMockPipelineVariable("scm") during your test setup.
        at JenkinsfileSpec.Slack is notified when tests fail(JenkinsfileSpec.groovy:22)
Caused by: groovy.lang.MissingPropertyException: No such property: (intercepted on instance [Jenkinsfile@3ec2ecea]
 during test [JenkinsfileSpec@352e612e]) scm for class: Jenkinsfile
        at JenkinsfileSpec.Slack is notified when tests fail(JenkinsfileSpec.groovy:22)
```

Another missing <dependency>?
No – just some setup work that Jenkins does.
In this mocked-up test environment, the test author must do it.

```
11      def setup() {
12          Jenkinsfile = loadPipelineScriptForTest("/Jenkinsfile")
13          Jenkinsfile.getBinding().setVariable( "scm", null )
14      }
```

Our First Test

HomeAway

There are some more missing dependencies, though. Add them and the test will complete!

```
<dependency>
    <!-- provides the slackSend() pipeline step -->
    <groupId>org.jenkins-ci.plugins</groupId>
    <artifactId>slack</artifactId>
    <version>2.3</version>
    <scope>test</scope>
</dependency>

<dependency>
    <!-- provides stage() step -->
    <groupId>org.jenkins-ci.plugins</groupId>
    <artifactId>pipeline-stage-step</artifactId>
    <version>2.3</version>
    <scope>test</scope>
</dependency>
```

```
<dependency>
    <!-- provides sshagent() step -->
    <groupId>org.jenkins-ci.plugins</groupId>
    <artifactId>ssh-agent</artifactId>
    <version>1.16</version>
    <scope>test</scope>
</dependency>
```

```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.923 s - in
 JenkinsfileSpec
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Log Verbosity

HomeAway

There are a lot of control handoffs in this process!

To ease debugging, the library is very talkative!

Don’t worry, though: It’s instrumented with the **SLF4J** logging facade & the **logback** logging framework, so a simple **logback-test.xml** will allow you to quiet it down for “normal” use.

There is nothing original about this use of logback so it is not included in these slides.

2nd Pipeline Test

HomeAway

JenkinsfileSpec.groovy

```
28      def "Attempts to deploy MASTER branch to PRODUCTION" () {
29          setup:
30              Jenkinsfile.getBinding().setVariable( "BRANCH_NAME", "master" )
31          when:
32              Jenkinsfile.run()
33          then:
34              1 * getPipelineMock("sh")({it =~ /ssh deployer@app-prod .*/})
35      }
36
37      def "Does NOT attempt to deploy non-MASTER branch PRODUCTION" () {
38          setup:
39              Jenkinsfile.getBinding().setVariable( "BRANCH_NAME", "develop" )
40          when:
41              Jenkinsfile.run()
42          then:
43              0 * getPipelineMock("sh")({it =~ /ssh deployer@app-prod .*/})
44      }
```

3rd Pipeline Test

HomeAway

Jenkinsfile

```
def deploy( _env ) {

    def DEPLOY_COMMAND="""
    docker-compose pull && \
    docker-compose down && \
    docker-compose rm -f && \
    docker-compose up -d --force-recreate"""

    if( _env == "test" ) {
        sshagent(["test-ssh"]) {
            sh( "ssh deployer@app-test -c '${DEPLOY_COMMAND}'" )
        }
    } else if( _env == "production" ) {
        sshagent(["prod-ssh"]) {
            sh( "ssh deployer@app-prod -c '${DEPLOY_COMMAND}'" )
        }
    }
}
```

3rd Pipeline Test

HomeAway

JenkinsfileSpec.groovy

```
46      def "deploy function deploys to TEST when asked" () {
47          when:
48              Jenkinsfile.deploy( "test" )
49          then:
50              1 * getPipelineMock("sshagent")(["test-ssh"], _ as Closure)
51              1 * getPipelineMock("sh")({it =~ /ssh deployer@app-test .*/})
52      }
53
54      def "deploy function deploys to PRODUCTION when asked" () {
55          when:
56              Jenkinsfile.deploy( "production" )
57          then:
58              1 * getPipelineMock("sshagent")(["prod-ssh"], _ as Closure)
59              1 * getPipelineMock("sh")({it =~ /ssh deployer@app-prod .*/})
60      }
```

Jenkinsfile Test Suite

HomeAway

```
[INFO] --- maven-surefire-plugin:2.22.0:test (default-test) @ jenkinsfile-test-whole-pipeline ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running JenkinsfileSpec
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 6.492 s - in JenkinsfileSpec
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

Unit-Testing Helper Scripts

HomeAway

What if the “deploy” function got very long and complicated?

We might want to put it in its own file.

Could we still test it?

Yes!

Unit-Testing Helper Scripts

HomeAway

Project Layout

pom.xml

```
<testResources>
    <testResource>
        <includes>
            <include>Jenkinsfile</include>
            <include>*.groovy</include>
        </includes>
        <directory>${project.basedir}</directory>
    </testResource>
    <testResource>
        <directory>src/test/resources</directory>
    </testResource>
</testResources>
```

Unit-Testing Helper Scripts Deployer.groovy

HomeAway

```groovy
def deploy( _env ) {

    def DEPLOY_COMMAND="""
    docker-compose pull && \
    docker-compose down && \
    docker-compose rm -f && \
    docker-compose up -d --force-recreate"""

    if( _env == "test" ) {
        sshagent(["test-ssh"]) {
            sh( "ssh deployer@app-test -c '${DEPLOY_COMMAND}'" )
        }
    } else if( _env == "production" ) {
        sshagent(["prod-ssh"]) {
            sh( "ssh deployer@app-prod -c '${DEPLOY_COMMAND}'" )
        }
    }
}

return this
```

Unit-Testing Helper Scripts DeployerSpec.groovy

HomeAway

```groovy
import com.homeaway.devtools.jenkins.testing.JenkinsPipelineSpecification

public class DeployerSpec extends JenkinsPipelineSpecification {

    def Deployer = null

    def setup() {
        Deployer = loadPipelineScriptForTest("/Deployer.groovy")
    }

    def "deploy function deploys to TEST when asked" () {
        when:
            Deployer.deploy( "test" )
        then:
            1 * getPipelineMock("sshagent")(["test-ssh"], _ as Closure)
            1 * getPipelineMock("sh")({it =~ /ssh deployer@app-test .*/})
    }

    def "deploy function deploys to PRODUCTION when asked" () {
        when:
            Deployer.deploy( "production" )
        then:
            1 * getPipelineMock("sshagent")(["prod-ssh"], _ as Closure)
            1 * getPipelineMock("sh")({it =~ /ssh deployer@app-prod .*/})
    }
}
```

Unit-Testing Helper Scripts Jenkinsfile

HomeAway

```
1   def Deployer = null
2
3   node {
4       stage( "Checkout" ) {
5           checkout scm
6           Deployer = load( "Deployer.groovy" )
7       }
```

```
28      stage( "Deploy to TEST" ) {
29          Deployer.deploy( "test" )
30      }
```

Unit-Testing Helper Scripts JenkinsfileSpec.groovy

HomeAway

```
def setup() {
    Jenkinsfile = loadPipelineScriptForTest("/Jenkinsfile")
    Jenkinsfile.getBinding().setVariable( "scm", null )
    explicitlyMockPipelineVariable("Deployer")
    getPipelineMock("load")("Deployer.groovy") >> {
        getPipelineMock("Deployer")
    }
}
```

Deployer is a pipeline *variable.*

Remember “def Deployer = null” at the top of the Jenkinsfile?

Deployer does **not** come from a plugin or from Jenkins.

We must explicitly mock it.

When a pipeline tries to load the Deployer.groovy file, **stub** that interaction to return our mock.

Unit-Testing Helper Scripts JenkinsfileSpec.groovy

HomeAway

```
def "Attempts to deploy MASTER branch to PRODUCTION" () {
    setup:
        Jenkinsfile.getBinding().setVariable( "BRANCH_NAME", "master" )
    when:
        Jenkinsfile.run()
    then:
        1 * getPipelineMock("Deployer.deploy")({it =~ /production/})
}

def "Does NOT attempt to deploy non-MASTER branch PRODUCTION" () {
    setup:
        Jenkinsfile.getBinding().setVariable( "BRANCH_NAME", "develop" )
    when:
        Jenkinsfile.run()
    then:
        0 * getPipelineMock("Deployer.deploy")({it =~ /production/})
}
```

Method calls on mock Pipeline **variables** are each their own Spock mock.

Unit-Testing Helper Scripts

HomeAway

```
[INFO] --- maven-surefire-plugin:2.22.0:test (default-test) @ jenkinsfile-test-helper-script ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running DeployerSpec
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.967 s - in DeployerSpec
[INFO] Running JenkinsfileSpec
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.794 s - in JenkinsfileSpec
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

Unit-Testing Shared Libraries

HomeAway

Helper Scripts are Pipeline Variables? You know what else has Pipeline Variables? Shared Libraries!

Can we unit-test those?

Yes!

Unit-Testing Shared Libraries

HomeAway

For large-scale Jenkins installations, common pieces of pipeline code may be delivered through the **Pipeline Shared Groovy Libraries** plugin, from different git repositories than the project source.

This allows one central location to define the common parts of many projects' pipelines.

Simple Shared Library

HomeAway

Jenkinsfile

```
1  @Library( "python-webapp-library" ) _
2
3  DefaultPipeline()
```

Simple Shared Library

HomeAway

Project Layout

```
.
├── Makefile
├── README.md
├── resources
│   └── com
│       └── example
│           └── SlackMessageTemplate.txt
├── src
│   └── com
│       └── example
│           └── SharedLibraryConstants.groovy
└── vars
    ├── DefaultPipeline.groovy
    └── Deployer.groovy
```

Simple Shared Library

HomeAway

SlackMessageTemplate.txt

```
1  Unit Tests for build ${BUILD_TAG} failed!
2
3  ${BUILD_URL}
```

SharedLibraryConstants.groovy

```
1  package com.example
2
3  public class SharedLibraryConstants {
4      public static final String DEPLOY_COMMAND = """
5      docker-compose pull && \
6      docker-compose down && \
7      docker-compose rm -f && \
8      docker-compose up -d --force-recreate"""
9  }
```

Simple Shared Library vars/Deployer.groovy

HomeAway

```groovy
import com.example.SharedLibraryConstants

def call( _env ) {

    if( _env == "test" ) {
        sshagent(["test-ssh"]) {
            sh( "ssh deployer@app-test -c '${SharedLibraryConstants.DEPLOY_COMMAND}'" )
        }
    } else if( _env == "production" ) {
        sshagent(["prod-ssh"]) {
            sh( "ssh deployer@app-prod -c '${SharedLibraryConstants.DEPLOY_COMMAND}'" )
        }
    }
}

```

Simple Shared Library vars/DefaultPipeline.groovy

HomeAway

```groovy
def call( Map _args ) {

    node {
        stage( "Checkout" ) { checkout scm }

        stage( "Build" ) { sh( "docker build --tag whole-pipeline ." ) }

        stage( "Test" ) {
            try {
                sh( "docker run --entrypoint python whole-pipeline -m unittest discover" )
            } catch( Exception e ) {
                def message = evaluate( '"""' +
                    libraryResource( "com/example/SlackMessageTemplate.txt" ) + '"""' )
                slackSend( color: 'error', message: message )
                throw e
            }
        }

        stage( "Push" ) { sh( "docker push whole-pipeline" ) }

        stage( "Deploy to TEST" ) { Deployer( "test" ) }

        if( BRANCH_NAME == "master" ) {
            stage( "Deploy to PRODUCTION" ) {
                Deployer( "production" )
            }
        }
    }
}
```

Simple Shared Library pom.xml

HomeAway

Must include vars/* Groovy scripts as resources
    This is how Jenkins will see them (initially).
Must include the “resources” directory as resources.
    This is *definitely* how Jenkins will see *them.*

```
<testResources>
    <testResource>
        <includes>
            <include>vars/**/*.groovy</include>
        </includes>
        <directory>${project.basedir}</directory>
    </testResource>
    <testResource>
        <directory>test/resources</directory>
    </testResource>
    <testResource>
        <directory>resources</directory>
    </testResource>
</testResources>
```

Simple Shared Library pom.xml

HomeAway

The pom must also *compile* groovy sources!

```
<plugin>
    <groupId>org.codehaus.gmavenplus</groupId>
    <artifactId>gmavenplus-plugin</artifactId>
    <version>${groovy.gmaven.pluginVersion}</version>
    <executions>
        <execution>
            <id>groovy</id>
            <goals>
                <goal>addSources</goal>
                <goal>addTestSources</goal>
                <goal>generateStubs</goal>
                <goal>generateTestStubs</goal>
                <goal>compile</goal>
                <goal>compileTests</goal>
                <goal>removeStubs</goal>
                <goal>removeTestStubs</goal>
            </goals>
            <configuration>
                <sources>
                    <source>
                        <directory>src</directory>
                        <includes>
                            <include>**/*.groovy</include>
                        </includes>
                    </source>
                </sources>
                <testSources>
                    <testSource>
                        <directory>test</directory>
                        <includes>
                            <include>**/*.groovy</include>
                        </includes>
                    </testSource>
                </testSources>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Testing Shared Global Vars

HomeAway

Two minor changes: Path & invocation. Done!

```groovy
import com.homeaway.devtools.jenkins.testing.JenkinsPipelineSpecification

public class DeployerSpec extends JenkinsPipelineSpecification {

    def Deployer = null

    def setup() {
        Deployer = loadPipelineScriptForTest("vars/Deployer.groovy")
    }

    def "deploy function deploys to TEST when asked" () {
        when:
            Deployer( "test" )
        then:
            1 * getPipelineMock("sshagent")(["test-ssh"], _ as Closure)
            1 * getPipelineMock("sh")({it =~ /ssh deployer@app-test .*/})
    }

    def "deploy function deploys to PRODUCTION when asked" () {
        when:
            Deployer( "production" )
        then:
            1 * getPipelineMock("sshagent")(["prod-ssh"], _ as Closure)
            1 * getPipelineMock("sh")({it =~ /ssh deployer@app-prod .*/})
    }
}
```

Testing Shared Pipelines

HomeAway

Setup: Change the path. Also, stub **libraryResource** since this pipeline uses it.

```
def setup() {
    DefaultPipeline = loadPipelineScriptForTest("vars/DefaultPipeline.groovy")
    DefaultPipeline.getBinding().setVariable( "scm", null )
    getPipelineMock("libraryResource")(_) >> {
        return "Dummy Message"
    }
}
```

It comes from a plugin, so we'll need a new <dependency> too:

```
<dependency>
    <!-- provides libraryResource() step -->
    <groupId>org.jenkins-ci.plugins.workflow</groupId>
    <artifactId>workflow-cps-global-lib</artifactId>
    <version>2.10</version>
    <scope>test</scope>
</dependency>
```

Testing Shared Pipelines

HomeAway

Global Library variables are .call()'d, so expect *that* instead of .deploy()

```
def "Attempts to deploy MASTER branch to PRODUCTION" () {
    setup:
        DefaultPipeline.getBinding().setVariable( "BRANCH_NAME", "master" )
    when:
        DefaultPipeline()
    then:
        1 * getPipelineMock("Deployer.call")("production")
}
```

Unit-Testing Shared Libraries

HomeAway

```
[INFO] --- maven-surefire-plugin:2.22.0:test (default-test) @ jenkinsfile-test-helper-script ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running DefaultPipelineSpec
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 8.07 s - in DefaultPipelineSpec
[INFO] Running DeployerSpec
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.93 s - in DeployerSpec
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

Unit-Testing Pipeline Code

HomeAway

Able to write unit-tests

1. In Groovy
2. With the Spock framework
3. With automatic mocks for pipeline step dependencies

For code in

1. Jenkinsfiles
2. Shared Libraries

Unit-Testing Pipeline Code

HomeAway

1. Unit-test your Jenkinsfile
2. Unit-test your shared libraries

Source: **github.com/HomeAway/jenkins-spock**

Slides: **bit.ly/2QHzRW9**

(photograph this slide)

Austin Witt
Senior Software Engineer
HomeAway.com, Inc.
awitt@homeaway.com