GuiceBerry 4.*.* tutorial

GuiceBerry

GuiceBerry brings the joys of dependency injection to your test cases and test infrastructure. It leverages Guice to accomplish this. It allows you to use a composition model for the services your test needs, rather than the traditional "extends MyTestCase" approach.

Installation

The simplest way to bootstrap GuiceBerry is to download GuiceBerry's own full "Source code" zip file from https://github.com/zorzella/guiceberry/releases. Besides the GuiceBerry sources proper and its dependency jars, this zip also has all the sources and dependency jars for this tutorial. It also has an ant build.xml as well as Eclipse and IntelliJ project files.

But if you would rather download the jars yourself, these are the jars you'll need to use GuiceBerry (version or subversion indicated with a "*" -- any recent version should work):

- guiceberry-4.*.*.jar
- guava-*.jar
- guice-4.*.jar -- you also need guice-servlet-4.*.jar and aopalliance.jar
- Either junit-4.*.jar[1] or testng-6.*.jar
- servlet-api-2.5.jar

Note that this list does not include the jars to compile and run the tutorial, so it is highly recommended you do download the full "Source code" zip.

Notes

GuiceBerry does not supplant your JUnit/TestNG testing framework -- it builds on top of it (and works around it, when necessary), so you can run your tests normally, from your favorite command line or IDE environment. Currently JUnit 3, JUnit 4 and TestNG are supported.

Using GuiceBerry in your tests does not require the use of Guice for your production code. However, there are two significant benefits to using the two in tandem: a common mental model, and a few compelling advanced features. So, for purposes of this tutorial, we will assume a working knowledge of Guice.

This tutorial assumes basic proficiency with Guice itself. If that is not the case, you should read some appropriate introduction on Guice.

There is a nice presentation walk-through of this tutorial. Find the link to the video and the

[1] Note: if you want to use JUnit3 with TearDownTestCase, you also need guava-testlib http://search.maven.org/remotecontent?filepath=com/diffplug/guava/guava-testlib/19.0.0/guava-testlib-19.0.0.jar

slides on our site.

What problem does it address?

JUnit and most other test frameworks are well-designed to handle simple tests -- those that exercise one or a small number of classes in isolation, and don't need setup of complex external dependencies. But once you need to write fully automated tests that depend on starting servers, setting up databases, and the like, these frameworks do very little to help you. You're left to build your own framework -- usually in the form of layers upon layers of base classes for test cases to extend.

GuiceBerry addresses this problem.

Tutorials

These tutorials are backed up by the executable source code (test cases etc) that can be downloaded as described in the Installation section. Note that only highlights of the code are repeated here: one is expected to follow along the full code to comprehend this.

The same tutorials are available for JUnit4, vanilla JUnit3, JUnit3 with TearDownTestCase and TestNG. There are are 2 self-contained tutorials, numbered 0 and 1, packages *.tutorial_0_basic and *.tutorial_1_server. For example, junit4.tutorial_0_basic is the basic tutorial for JUnit4.

The rest of this document (except for the Hello World, where I compare the different versions) will refer to the JUnit4 version of the tutorial, though the differences between the tests in the different frameworks is so small that this should not be a problem.

Tutorial 0 introduces the reader to GuiceBerry, its syntax and walks you through its basic features, in complete isolation of any external systems.

Tutorial 1 is a more practical example of how a test that uses GuiceBerry looks like in the real world -- it actually starts a server, and connects to it using WebDriver, etc.

Feel free to skip back and forward between the two tutorials, to balance your needs for the fundamentals vs. just learning by example.

Tutorial 0 -- The basics

In this first set of examples, we'll explore the basic GuiceBerry features.

Example 0 -- Hello World

Let's just start with a "Hello World" test case, as you can see in Example0HelloWorldTest.java. This is the smallest test that can be written with GuiceBerry, and GuiceBerry is, in fact, a no-op here (it gets setup, but does not do anything).

In the JUnit4 version, note these two things:

1. There is an enclosed "Env" class that extends GuiceBerryModule
2. There is a JUnit4 @Rule that points to this enclosed "Env" class (which is what the doc refers to as a "GuiceBerry Env"). This is the only link between JUnit4 and GuiceBerry

As we said, GuiceBerry does no do anything visible in this case, but you should realize (as we'll demonstrate on the next Example), that GuiceBerry is creating an Injector with the "Env" as its only Module.

Let's take a moment to compare here the different the test frameworks. They all have the same "Env" class, but the "link" described in #2 is different:

- on JUnit4, the link, as we said, is an @Rule
- on vanilla JUnit3, the link happens on setUp and tearDown
- on JUnit3 with TearDownTestCase, the link happens on setUp
- on TestNG, the link happens on @BeforeMethod setUp and @AfterMethod tearDown

Aside from this boilerplate, all code in all frameworks is exactly the same.

Example 1 -- @Inject works

This example looks very much like the first one, but it has two new things to call out:

1. The GuiceBerry Env used declares a binding from integer, annotated with NumberOneHundred, to the constant 100
2. A field in the test gets @Injected with a int that's annotated with @NumberOneHundred

This time, we can clearly see that GuiceBerry creates an Injector from the GuiceBerry Env, and then it uses it to inject the test's "number" field. This is asserted in the test.

Example 2 -- Scopes work, plus new TestScope

This example demonstrates the use of Scopes in your test. If you don't use and understand Guice Scopes, you might want to leave this for later.

1. There are three bindings of Integer.class (one for each of of three annotations), bound in different scopes
2. The three scopes used are none ("unscoped"); TestScoped; and Singleton
3. All three of these Integers is bound to the same Provider class (though different instances), this Provider is initialized with a value, and then returns value++ on its "get"
4. Three test fields are being @Injected with a Provider of each of these three bindings

Except for TestScoped, there's nothing new here. Since the `@SingletonScopedIncrementingNumber`-annotated int is bound in `SINGLETON`, we expect to always get the same number out of the Provider (300), and since the `@UnscopedIncrementingNumber`-annotated int is unscoped, we expect to always get an incremented result.

`TestScoped` is a fundamental construct for GuiceBerry. It lasts from the beginning to the end of a test, and, thus, it can be used for anything that we want to use throughout a single test, but not for subsequent tests. We'll see how it is useful later.

The test cases testOne and testTwo simply assert what we expect -- that "getting" the `singletonScopedIncrementingNumber` Provider always returns 300, that "getting" the `testScopedIncrementingNumber` Provider returns the same number throughout a test, but an incremented number in a following test, and that "getting" the `unscopedIncrementingNumber` Provider always returns an incremented number.

Example 3 -- TestWrapper and TestId

This example is a simplified version of Example 2, but it adds an important concept: the TestWrapper. The test's GuiceBerry Env simply binds a certain class to TestWrapper.class. This class gets notified every time a test begins and ends (think @Before and @After in JUnit4 parlance).

It would be not possible to demonstrate this feature through regular asserts. To understand this test you need to look at its output.

This test also makes use of TestId, which we will look at later. For now, just understand that there is a binding to TestId (in TestScoped), and that TestId's toString() method contains the test name. Since TestWrapper is bound as SINGLETON, we inject a Provider<TestId> to it, and "get" it in its two methods.

What you want to see is that the output of this test shows the order of execution of statements, which is, for the first test:

- Beginning: junit3.tutorial_0_basic.Example3TestWrapperTest.testOne:836
- Inside testOne
- Ending: junit3.tutorial_0_basic.Example3TestWrapperTest.testOne:836

In other words, toRunBeforeTest and toRunAfterTest look as though they were @Before and @After methods for this test (or a superclass, as it's common) -- but they are not! This composition (rather than inheritance) will allows us to have cleaner, more versatile tests. (TODO: write a tutorial on this)

Example 4 -- @Injecting TearDownAccepter

No matter which test framework you use GuiceBerry provides you, out of the box, with a useful TearDownAccepter binding.

Example 4 is a showcase for TearDownAccepter. We @Inject it in the test class, and then we can add tear downs to it.

Note a few things:

- this test case will fail!
- testOne is your sunny-case. Everything goes well on the tearDowns -- they both reset

the values of the integers to "0".
- You can see from the console output (and from the assertion on SecondItemResetter) that the second item is reset before the first -- i.e. in reverse order of "addTearDown".
- On testTwoFailsWithException, though, the SecondItemResetter throws an exception. Still, as the Console output shows, FirstItemResetter is still executed. And the test still fails with an exception.

Example 5 -- A custom GuiceBerryEnvSelector

So far, all examples have used a single, static GuiceBerry Env, which is quite sufficient in most cases. But it is easy for you get full control on the GuiceBerryEnv selection process. See in this example how the test name is used to determine which GuiceBerry Env to use.

Tutorial 1 -- Starting an actual HTTP server

In this tutorial, contained in the "tutorial_1_server" package, we'll put GuiceBerry to use. We'll be black-box testing a minimalistic skeleton of a Pet Store: our test will bring up a HTTP server, and then connect to it through HTTP, so it can assert things about the returned pages.

To accomplish this, we'll leverage Jetty for our HTTP Server, and WebDriver for our tests.

The prod package

The ".prod" subpackage contains the "production" code. I.e. it has a server and its dependencies that would be ultimately deployed. The same "prod" code is shared by all examples in the tutorial -- it is, in fact, shared by the tutorials in all the different test frameworks! Therefore, even though it is not really related to GuiceBerry, it is important for us to understand the code there before we move on to the examples. Likewise, the "testing" directory, where the bulk of the GuiceBerry code exists, is also shared by all test frameworks. Only the tests themselves are in the test-framework specific package.

PetStoreServer is a simple wrapper around Jetty that takes a port number in its constructor, and builds a server that registers a single Servlet (WelcomePageServlet), and registers the "GuiceFilter" filter. It also has a "start" method that creates an Injector with a single installed Module (PetStoreModule), which is used to inject the WelcomePageServlet's members, and then Jetty's Server.start method is called.

WelcomePageServlet simply spits out a few words of welcome, including the "featured pet", which is a random entry of the Pet enum.

A "main" method is provided in case you want to take it for a spin. It starts the server on

port 8080, leaves the server running for 20 seconds, then stops it. Connect to http://localhost:8888 while the server is still running to see the page.

Example 0 --Hello Server

In this example, our test gets injected a "WebDriver" and a port number for us to connect to (which is a random integer between 8000 and 8099).

The first test simply reads the main page and asserts that the "Welcome" message is there. The second asserts the title of the page.

This is, in some respects, the example you should try to understand the best. However simple-looking, it is a microcosmos of a canonical GuiceBerry test, so I'll spend a few words more here.

First, note that there are two Injectors -- one created by GuiceBerry, which is used to @Inject the test; and the other created in the constructor of your server. This is, as we point out elsewhere, the canonical way to use GuiceBerry, and is the most important thing I need you to understand in this example. Do not get confused by them. Note that you can't "@Inject @PortNumber int" on the Servlet, just like you can't "@Inject @Featured Pet" in the test.

Second, note that the server is being started by means of a GuiceBerryEnvMain called PetStoreServerStarter. A GuiceBerryEnvMain, if bound, causes GuiceBerry to call that class right after the Injector is created.

Finally, note we never shut down the server explicitly -- this is done by the JVM upon exit. There are ways to shutdown your server explicitly, in case you need to, but we won't cover this here, as it's usually not necessary or desired.

Example 1 -- Page Objects

Now that we have a fully functional Injector powering our test, we can start to take advatage of this. In this example, we extract some common functionality using the "Page Object" paradigm. It's the same Example 0 test rewritten with Page Objects.

Note how natural it is to not use inheritance to get access to common functions (such as the

one used to navigate to the welcome page).

Also note our test does not have a direct reference to WebDriver. Which means that, if we decide to replace WebDriver by some other technology, we need not change the tests, just the test Page Objects. Of course, this will only pay itself off if we create Page Objects and methods that get used by many tests.

Controllable Injections -- the issue at hand

We'll now cover one compelling but advanced feature. Make sure you have a good understanding of GuiceBerry basics before proceeding.

Note how the Welcome page presents a changing message about the Featured Pet. In the example code, this is being given by the following factory method:

```
  @Provides
    @Featured Pet getFeaturedPet() {
    return calculateFeaturedPet();
  }

  private final Random random = new Random();

  /** Let's simulate a call to a non-deterministic service -- e.g. an
      * external server, or a DB call to a volatile entry, etc.
      */
  private Pet calculateFeaturedPet() {
    Pet[] allPets = Pet.values();
    return allPets[random.nextInt(allPets.length)];
  }
```

As the comment says, it simply mimics the sort of thing that would happen if that method were to read from an external source. How can we test this message?

You could just have the test read from the same source as the server code does to figure out what's the current pet of the month, like:

```
public void testFeaturedPet() {
  welcomeTestPage.goTo();
  welcomeTestPage.assertFeaturedPetIs(readFromExternalService());
}
```

But there are a few problems with that:

1. If this is a slow service, it will make your test even slower (both server and test client will make independent calls to it)
2. It's brittle -- if your external service returns a value for the server different from the one it returns for the test client, your test breaks
3. It's incomplete -- you couldn't possibly have tests for more than one message. You are held hostage to the current pet of the month. Wouldn't it be ideal to make sure

that **next** month's pet works before next month arrives, for example?
4. It may be worse than that. Say, instead of Featured Pet your server returns your current country based on your IP address. To test other countries, you'd have to deploy your tests across the globe...

There's a very elegant way of solving this problem using a "Controllable Injection". Controllable Injection is not exactly a tool, but rather a well-supported set of techniques (and tools) that, put together, address this kind of issue. The fully realized Controllable Injection prescription can be daunting to understand all at once, so we'll solve it the naive way, and build upon it.

Example 2 -- Static override -- the naivest solution

To start with, let's imagine both the PetStoreServer and the test live in the same JVM, just like in the example. Here's what we can do.

First change the factory method code to read:

```
    public static Pet override = null;

    @Provides
      @Featured Pet getFeaturedPet() {
      if (override != null) {
        // This should never happen in production
        return override;
      }
      return calculateFeaturedPet();
    }
```

Then have your test install an override and then clear it on tearDown:

```
public class Example2StaticOverrideTest {
  // [...]

  @Test
  public void testWhenDogIsFeatured() {
    Pet expected = Pet.DOG;
    PetStoreModule.override = expected;
    // register a tearDown, so that at the end of the test,
    // the override is set to null again
      tearDownAccepter.addTearDown(new TearDown() {
      public void tearDown() {
        PetStoreModule.override = null;
      }
    });
    welcomeTestPage.goTo();
    welcomeTestPage.assertFeaturedPetIs(expected);
  }
```

The code for that is in the Example 2. Let's review this approach:

- I changed my factory method to honor a static override (that, in prod, should always be unset)
- My test sets this static override (and then resets it on tearDown)
- My tests are deterministic, and I can test any Pet I want

We say in this case that the test can control the @Featured Pet injection (ergo the term "Controllable Injection"). This, as stressed in the last point above, is perfectly functional, but it has a few caveats:

- It only works for "same JVM" between server and test
- It makes use of a global static variable, which is greatly frowned upon (note the tearDown is important, otherwise the following tests, until this is cleared, will also get DOG as their @Featured Pet)
- It pollutes production code with for-testing code
- It introduces boilerplate. For every injection you want to be able to control, you need to write a boilerplate public static override variable and the "if" statement
- It prevents tests from running in parallel. This point is, often, moot, since few people actually run their tests in parallel, but, if you do, this issue is very important to you. This stems directly from the use of a global static variable

How can we make this better?

Example 3 -- No single global static override by leveraging TestId

Note, in the previous example, that if you were to put a breakpoint in the first test after you set the override (say, in the "goTo" line), run it, and open a browser against http://localhost:8888, you would always get DOG as your @Featured Pet. This is, as we point out, due to the global static override. Let's say we want a given overridden injection to apply to a single test, rather than globally. We would conceptually have to change:

```
if (override != null) {
```

to something like

```
if (getOverrideForTest(currentlyRunningTest) != null) {
```

The problem is, the server runs in a separate thread stack from the test, and can't immediately know what is the currently running test.

But note that they communicate through HTTP, and the HTTP protocol allows for Cookies. Also note that, as seen in Tutorial 0's Example 3, GuiceBerry provides a "Test Scoped" "TestId" instance for you out the box.

So, first, change your WebDriver factory to set a cookie named TestId.COOKIE_NAME with the value testId.toString(), like here. Note that we also had to add a visit to the domain before we add the Cookie. This is because Cookies are tied to the domain, and things would not work otherwise. What's in **bold** is what is being added here. Everything else we already had before:

```
public class MyGuiceBerryEnv {

  @Provides
  public WebDriver getWebDriver(TestId testId) {
    WebDriver driver = new HtmlUnitDriver();
    driver.get("http://localhost:" + portNumber);
    driver.manage().addCookie(
        new Cookie(TestId.COOKIE_NAME, testId.toString()));
    return driver;
  }
}
```

Second, your server also needs to be able to @Inject a TestId, which it reads from the Cookie. There is a TestIdServerModule provided by GuiceBerry that reads the TestId from that Cookie. We install this module in our **server**.

Third, your static constant needs to be tuned into a Map, and your factory needs to use the TestId to find the right value:

```
    public static final Map<TestId, Pet> override = Maps.newHashMap();

    @Provides
      @Featured Pet getFeaturedPet(TestId testId) {
      Pet pet = override.get(testId);
      if (pet != null) {
        return pet;
      }
        return calculateFeaturedPet();
    }
```

Finally, we change our test:

```
  @Inject
  TestId testId;

  public void testDogAsPotm() {
      Pet expected = Pet.DOG;
    PetStoreModule.override.put(testId, expected);
    // register a tearDown, so that at the end of the test,
    // the override is set to null again
    addTearDown(new TearDown() {
      public void tearDown() {
        PetStoreModule.override.remove(testId);
      }
    });
    welcomeTestPage.goTo();
    welcomeTestPage.assertFeaturedPet(expected);
 }
```

All this code is captured in Example3CookiesOverrideTest. Let's recap:

- GuiceBerry gives you an @Injectable TestId for free (in the test Injector, **not** the

server)
- We set this TestId as a Cookie when we create an HTTP session to the server
- The server installs TestIdServerModule that provides a TestId to the server by reading the Cookie
- The server's factory to @Featured Pet now uses an @Injected TestId as a key to a Map of overrides
- The test sets an override in the map by using the TestId as a key

Now let's see where we are with regards to the caveats before:

- It **still** only works for "same JVM" between server and test
- It, **technically**, **still** makes use of a global static variable, but **not really**. The impact of the override is on a single test, since it's keyed by TestId. The "tearDown" is a sane thing to do, but it's not strictly necessary -- future tests (or other tests that run in parallel) won't be affected. In fact, if you run the **same** test twice (even in parallel), each one will have a different TestId, and, therefore, won't step on each other's toes. You may do the breakpoint experiment we talked about in the beginning of this example and you'll see that the browser does not get affected by the override.
- It pollutes production code with for-testing code **even more**. We took a step back here, but we'll see how to fix that later, so bear with me
- It introduces **more** boilerplate. Likewise, this requires even (a little bit) more boilerplate. Again, we'll solve this issue later.
- It **allows** tests to run in parallel. This problem is 100% solved

Example 4 -- Rewriting your bindings through IcMaster

GuiceBerry provides with some pre-canned code that allows you perform overriding of injections without having to write boilerplate code or polluting your production code with testing code.

There are two parts to the problem: the client Injector and the server Injector. Let's get back to Example 1, and examine the code flow at the moment Example1PageObjectsTest is executed as a test for the first time.

1. JUnit creates an instance of Example1PageObjectsTest, GuiceBerry is set up
2. GuiceBerry is told to use the PetStoreEnv0Simple class, so that is instantiated
3. PetStoreEnv0Simple's configure method is called. It configures PetStoreServerStarter as its main method
4. GuiceBerry creates the Test Injector from PetStoreEnv0Simple
5. GuiceBerry uses this Test Injector to create an instance of PetStoreServerStarter, which requires a PetStoreServer, which is @ProvidedBy buildPetStoreServer
6. buildPetStoreServer simply creates an instance of the server, at a random port
7. GuiceBerry calls PetStoreServerStarter's run method, which calls MyPetStoreServer's start method
8. The Server Injector gets created
9. The server is started

Now let's look at Example4InjectionControllerTest, which uses

PetStoreEnv4InjectionController.

During step "3", i.e. **before the Test Injector is created**, we instantiate an IcMaster, and we tell it all the classes we want to be able to control, and what is the strategy it should use to control them. Note it says it only wants to control Pet.class (using the StaticMapInjectionController.strategy(), which we'll talk about later).

Then it asks this IcMaster to install some generated Module in the soon-to-be-created Test Injector. What this generated module does is, for each class you want to control the injection (in the server), it creates a binding InjectionController<ThatClass> to some black box code. In our case, a binding to @Featured InjectionController<Pet> is created. Note how we can @Inject that into the test.

Then, it stores the created IcMaster to be used later.

Now, during step "6", i.e. right **before the Server Injector is created**, it changes the Module to be used to create the Injector. Two things are done: first, it installs TestIdServerModule (which allows the server to @Inject a TestId, just like in Example 3); second, it passes all the Modules to the IcMaster.buildServerModule we saved from before, and gets a new Module back, that it then returns.

This is the second bit of magic. This returned module is functionally equivalent to the sent modules, except that the bindings for each thing we said we want to control (@Featured Pet in our case) are **rewritten** to do something like what we did manually in Example 2 -- honor some overridden variable whenever we're controlling an injection.