Testing Controls

Table of contents

1. Overview

A Beehive Control can be tested either inside of an application container or outside in a standalone Java environment. The latter can be particularly useful running unit tests or during test driven development (TDD). This document describes how to unit test a Beehive Control using JUnit.

2. The JUnit Controls Container

The Controls JAR file `beehive-controls.jar` provides build time, run time, and test time support for developing Controls. This makes it very easy to begin unit testing Controls that are built as part of an application. Out of the box, Controls provides integration into the JUnit test framework via the ControlTestCase base class. This base class provides a Control *container* that hosts a Control for the duration of a Control test. It also provides help in instantiating a Control declaratively via the @Control annotation.

To author a JUnit Controls test using the base class, the test case should be declared as:

```
public class FooTest
    extends ControlTestCase {
...
}
```

For each test case with a name method matching the JUnit naming convention *test**, the JUnit container will start and stop the ControlTestContainerContext. The `beginContext` method will be called at the beginning of each test in the `setUp()` method, and the `endContext` method will be called at the end of each test in the `tearDown()` method. This will simulate a *interaction lifetime* with the control where multiple Control instances can be invoked multiple times. The Control will hold any resources it acquires for the duration of the test method. As an example, this begin / end Context lifetime represents the same lifetime as that for a single HttpServletRequest in the web tier. Any resources loaded from the `ControlTestContainerContext` are loaded from the current thread's context class loader.

For a single test, once the `ControlTestContainerContext` has been initialized, the controls in the JUnit test class are declaratively instantiated via the `ClientInitializer` that was generated for the test case.

Note:

In order to use a `ClientInitializer`, the JUnit test cases must have been processed with the Controls annotation processor via the `<build-controls>` Ant macro.

In cases where a test needs to provide a custom implementation of a Controls container, a new container implementation will be created by overriding the initializeControlContainerContext() method.

In cases where a test needs to override the base `setUp` and `tearDown` JUnit lifecycle methods, the test author should remember to call `super.setUp()` and `super.tearDown()` from the overridden methods.

3. Control Instantiation

Controls declared with the `@Control` annotation will be declaratively instantiated by the JUnit container. These references will be valid for the duration of the JUnit test.

4. Using another Base Class

In cases where tests are unable to extend the `ControlTestCase` base class, the Control container and its lifecycle can be implemented using utilities available in the class ControlContainerContextManager. This class provides methods to begin and end a Context, to instantiate controls, and to get the Context object itself. To implement a `ControlContainerContext` for a single test case, the following code can be added to a test case method:

```
public void testFoo() {
    ControlContainerContext ccc = new ControlTestContainerContext();
    ControlContainerContextManager cccManager =
ControlContainerContextManagerFactory.getInstance(ccc);
    cccManager.beginContext();
    cccManager.instantiateControls(this);

   ... test code ...

   cccManager.endContext();
}
```

The same `ControlContainerContext` methods could be added to the JUnit test lifecycle methods `setUp()` and `tearDown()`.

5. Running the JUnit Tests

The JUnit tests for a Control can be executed in a variety of ways including via Ant or from and IDE like IntelliJ or Eclipse. Ant can run these JUnit tests in the usual means by executing them directly or by using the optional Ant tasks to support running and reporting results for JUnit tests.

To run Controls JUnit tests from an IDE, the command line build to code generate the Controls support classes often needs to be run so that the Control support classes are available in classpath. Once these classes have been generated, an IDE's JUnit integration should successfully. While this is inconvenient, as support for annotations and APT improves in IDEs, this process should become easier.