TESTING

Arquillian: Easy Jakarta EE Testing

How to use the Arquillian framework to test Jakarta EE applications

by Josh Juneau

August 19, 2019

Testing web and enterprise applications can be much more tedious than testing Java SE projects because there are many different types of files, varying scopes and, in most cases, different phases to test. Over the years, Arquillian has become a powerful, robust testing framework for use with Java EE and Jakarta EE. It was developed by the JBoss project (now part of Red Hat), and there are several add-on extensions that can enhance the tests and their output.

The Arquillian framework is very powerful, and with that power there comes a certain amount of complexity. Configuration of test classes can be very easy or fairly complex depending upon the level of testing that needs to be done.

In this article, I walk through configuration and testing with the Arquillian framework. I cover the configuration and build of a basic Jakarta EE 8 application (on par with Java EE 8) along with some tests that use JUnit 4 and Arquillian. After I provide some basic examples, I examine a few of the more important features and extensions that you can use to enhance verification of your Java EE and Jakarta EE projects. To follow along, you should be familiar with Java EE or Jakarta EE projects.

The example application is built using Maven, but any IDE that works with Maven projects should work fine. However, my example uses Apache NetBeans IDE. I recommended that you obtain from GitHub the Maven web application project example entitled ArquillianExample.

Configuring the Environment

One of the most difficult areas of working with the Arquillian framework is configuration. Only a few Maven dependencies are required for basic testing, but you must use the correct versions of these dependencies or you can run into incompatibilities that lead to errors. To begin from scratch, create a Maven web application project, either using a Maven archetype from the command line or an IDE such as Apache NetBeans. If you are using Apache NetBeans, choose Java EE 8 as the platform.

Once you create the project, right-click the project and open the POM file. Add the dependencies that are shown in **Listing 1**, which cover the Arquillian requirements. (Note: If you use the example source code for this article, there are additional dependencies that pertain to the Jakarta EE platform and the Apache Derby database.)

Walking through the dependencies, the `arquillian-bom` artifact is the "bill of materials" for the Arquillian framework, and it is used for any transitive dependencies. Next, in the standard `<dependencies>` section of the POM file, the JUnit framework is added. In this case, JUnit 4 is being used, because JUnit 5 was not yet supported at the time this article was written. However, some work has been done for getting Arquillian to work with JUnit 5. The Arquillian JUnit container is required for the API to work with JUnit. There is also a container available for use with TestNG if that is your preferred testing framework. The JBoss ShrinkWrap Resolver is used to provide support for ShrinkWrap, which enables you to declaratively create a deployment archive via code. The remaining dependencies are required for deploying the test archive to an embedded CDI container named `arquillian-weld-ee-embedded`. There are several deployment options, depending upon the level of testing you are trying to achieve. The Weld embedded container is useful for testing CDI without the need for making database connections or other features that are part of an application server container. Utilization of the embedded container makes startup time and testing a breeze. If you need to perform database connection testing or require the use of other application server features, deployment to a remote container is possible. Later in the article, I'll cover connecting to a GlassFish server.

Writing a Small Application

The application used in this example interacts with a small set of database tables, allowing you to Create, Read, Update, and Delete (CRUD) records pertaining to a swimming pool company and its customers. Apache NetBeans and most other IDEs can be used to generate the application in a very quick manner using wizards. So, I will skip over the initial generation of the application and focus on testing the application logic.

A `Pool` can be entered into the database and associated with a `Customer` object. In this article, I create unit tests of the `Pool` database operations using the Java Persistence API (JPA) and an EJB bean. The article also covers the testing of a CDI bean. This bean performs the business logic that binds the front end to the EJB bean that interacts with the database. Note that the EJB bean could be replaced with RESTful service calls, and testing would occur in a similar manner.

Writing and Executing Basic CDI Unit Tests

Let's begin with a very simple CDI test. The CDI bean for this basic test simply uses CDI injection and tests to ensure that `Pool` objects are being created properly. Before you can write a test, you must set up the testing class properly.

Typically, each class that contains business logic for an application should have a corresponding test class, and in this case, the test class is named `PoolControllerTest`. For the test class to harness Arquillian, you must place the `@RunWith(Arquillian.class)` annotation before the class declaration. This annotation tells the testing framework to run the tests using the Arquillian engine.

Each class must also have a deployment method, which packages all of the specified classes, libraries, and configuration files into a testing package (WAR, EAR, or JAR). The testing package is then automatically deployed to the selected test container and executed by the Arquillian engine after it is built, and the resulting output is displayed to the command line or in the IDE.

To create the deployment package, annotate a public static method with `@Deployment`, and return a `ShrinkWrap` archive. (**Listing 2** illustrates this.) Inside the deployment method, use the builder pattern to construct the deployable archive with `ShrinkWrap`. Call the `create()` method and pass the type of archive that is desired, which in this case is `WebArchive.class`. Other types of archives include `JavaArchive` to create a standard JAR and `EnterpriseArchive` to create an EAR file. You must add to the `CLASSPATH` each class that will be used by the test, and this is achieved by calling the `addClass()`, `addClasses()`, `addPackage()`, or `addPackages()` method accordingly. Note that the `addAsManifest()` method is also called, adding a `beans.xml` file to invoke CDI. Refer to the ShrinkWrap tutorial for more details regarding the various methods that are part of the `create()` builder for the deployment.

Listing 3 contains the source code of the PoolController. In the constructor, two Pool objects are constructed and added to an ArrayList. The ArrayList is tested to ensure that it is not empty. The CDI controller can be injected into the test class by using the standard @Inject annotation. The injection should occur just prior to the test method. Because this test uses JUnit, annotate the test method with @Test, and within the test method, call one of the org.junit.Assert assertion methods to indicate whether the test was successful. See Listing 4 for the complete source code of the test class. As with any other JUnit test, there are several Assertion methods that can be used to test an expression, returning a boolean value. See the JUnit documentation for a list of the Assertion method options.

To execute the test, simply run the Maven build from within an IDE or from the command line; the tests will be run each time the project is built. To perform the tests only, use the `mvn test` command from the command line. When the project is built using Maven or the tests are executed from the command line, the embedded Weld container (which is the selected deployment container for this project) executes the tests. After executing the tests, the output should look as follows:

```
Tests run: 1, Failures: 0,
 Errors: 0, Skipped: 0
 org.javamagazine.arquillianexample.cdi.PoolControll
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Configuring for JPA and GlassFish

To perform testing against an actual database, you need to make a few decisions. It is possible to inject a PersistenceContext directly into a test class, and in such cases, a UserTransaction should also be injected to manage the transactions. In such cases, the test class itself is being treated as a CDI controller. Such a test might look similar to the following:

```
@Test
public void insertData(){
    utx.begin();
    em.joinTransaction();
    Pool pool = new Pool();
    // set values
    em.persist(pool);
    utx.commit();
    em.clear();
}
```

In this article, instead of using a PersistenceContext, I use a remote GlassFish server for deployment and testing. It is also possible to use Payara as an alternative to GlassFish, because it will work with the same dependencies. There are several other servers you can choose as well. Testing against a remote server has a number of benefits, including the ability to test in a real environment as opposed to a test environment, not to mention the ability to configure database connection pools and server environmental settings to match production use cases.

To test against a remote server, the server must be up and running before you attempt to execute the tests or they will fail. The server is outside the Arquillian engine, so it must be configured as a “remote” server even though it might be installed on the same machine that is running the tests. As an alternative, you can configure a truly remote server to run the tests.

Additional configuration is required within the `src/test/java/resources` area for setting up the remote server environment and the additional dependency for `arquillian-glassfish-remote-3.1` in the POM file, as shown in

Listing 5. The credentials for logging in to the GlassFish server must be placed into an `arquillian.xml` file, as shown in **Listing 6**.

Note that one of the attributes for configuring the container is `adminHost`, which can point to a truly remote server, if desired. It is a good idea to set up a `test-persistence.xml` file in this resources area, so testing can occur using a test database. Create a persistence context named `test-persistence.xml`, and place it in the project's `src/test/java/resources` folder. The file should contain a Java Transaction API (JTA) connection configuration for a data source that is configured within the remote application server container.

Now that the configuration for a remote GlassFish server is in place, it is time to construct tests for the EJB bean `CustomerFacade`. The EJB tests should be placed inside the `src/test/java` folder and named `org.javamagazine.arquillianexample.session.CustomerFacadeTest`. The source code for `CustomerFacadeTest` is shown in **Listing 7**. The test class should include the standard `@RunWith(Arquillian.class)` annotation before the declaration, and it should contain a static method annotated with `@Deployment` for returning a ShrinkWrap archive.

Here, the deployment method is named `createDeployment()` and contains references to each of the classes that must be in the `CLASSPATH` for running the EJB test. The deployment uses the builder pattern to call

```
addAsManifestResource(EmptyAsset.INSTANCE,"beans.xml
```

for configuring CDI within the test deployment artifact. It also contains a call to

```
addAsResource("test-persistence.xml", "META-INF/pers
```

for configuring the persistence context.

In this case, the `test-persistence.xml` file is loaded as a standard `persistence.xml` file for the test deployment. The test method within `CustomerFacadeTest` is very simple, because it calls `Assert.assertTrue` to determine whether the `Customers` entity is queried and it returns results from using the `findAllCustomers()` method.

In the actual application, the EJB bean is not called directly, but rather the CDI controller `CustomerController` is invoked from the front end. Therefore, the `CustomerController` CDI bean should be tested to ensure that it is invoking the application logic properly to return the results to the screen. The `org.javamagazine.arquillianexample.cdi.CustomerControllerTest` is used to perform testing against the CDI controller, as shown in **Listing 8**. Very similar to the EJB test, this test calls the `CustomerController getCustomerList()` method to ensure it is populated. The `CustomerController` class populates the `customerList` upon instantiation, so the test should return a `true` result.

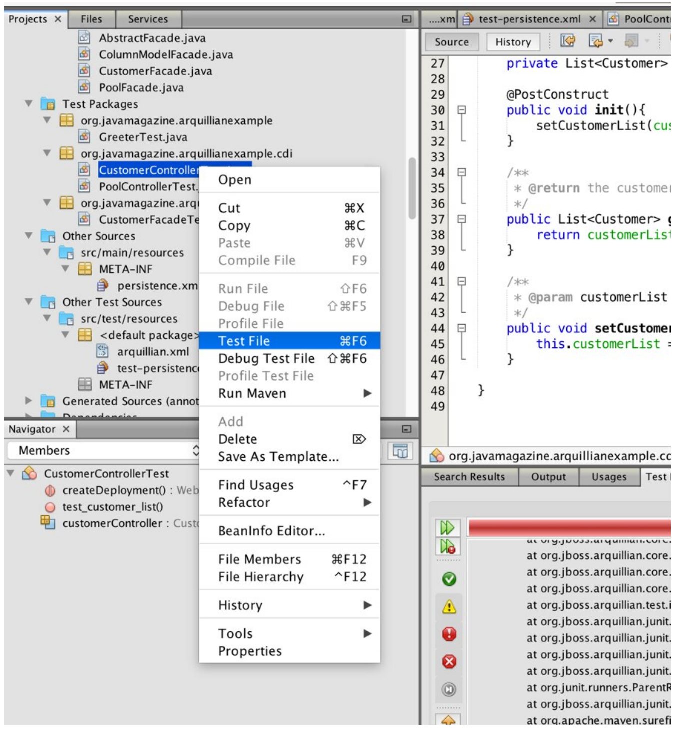
Options for Test Execution

The tests can be executed from the command line or within an IDE. Tests can be run separately for each file to be tested, or all tests for a project can be harnessed into the build process. To execute a single file test from the command line, open a command line and navigate to the project directory; then issue the following command:

```
mvn test
```

The results of the test will be displayed immediately. In the case where X number of tests are configured, there should be X number of successes. If you are running the tests as part of the Maven build, the build will fail if any of the tests fail. This prevents the packaging of an application when failing tests are encountered.

Testing within an IDE can be beneficial for receiving visual feedback. The NetBeans IDE allows a test to be executed by right-clicking the test file or the project itself and choosing Test File, as shown in **Figure 1**.

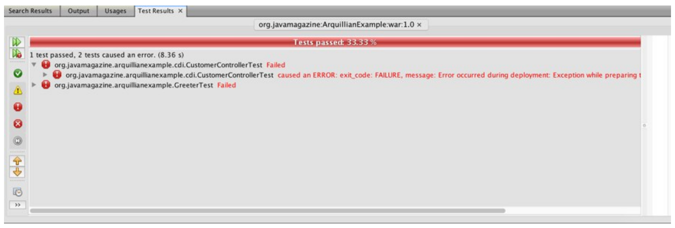


```
    private List<Customer>

    @PostConstruct
    public void init(){
        setCustomerList(cu
    }

    /**
     * @return the custome
     */
    public List<Customer> g
        return customerList
    }

    /**
     * @param customerList
     */
    public void setCustome
        this.customerList =
    }

}
```

```
at org.jboss.arquillian.core.
at org.jboss.arquillian.core.
at org.jboss.arquillian.core.
at org.jboss.arquillian.test.i
at org.jboss.arquillian.junit.
at org.jboss.arquillian.junit.
at org.jboss.arquillian.junit.
at org.jboss.arquillian.junit.
at org.junit.runners.ParentR
at org.jboss.arquillian.junit.
at org.apache.maven.surefi
```

Figure 1. Running a single test within the NetBeans IDE

When you execute a test within NetBeans, a Test Results panel will be displayed to show the outcome of the tests. If there are several tests, each of the failed tests is listed within the panel so the results can be parsed and corrections can be made, as shown in **Figure 2**. The panel also contains buttons on the left for easily navigating the test results or re-executing the tests.



```
1 test passed, 2 tests caused an error. (8.36 s)
  org.javamagazine.arquillianexample.cdi.CustomerControllerTest Failed
    org.javamagazine.arquillianexample.cdi.CustomerControllerTest caused an ERROR: exit_code: FAILURE, message: Error occurred during deployment: Exception while preparing t
  org.javamagazine.arquillianexample.GreeterTest Failed
```

Figure 2. Apache NetBeans Test Results panel

Testing with Multiple Containers

To configure a project with the ability to run tests in different containers, consider declaring Maven profiles within the POM file. Maven profiles enable dependencies to be separated from each other for each container; and when the project is built, the desired profile can be selected. It is also possible to designate a default profile in the POM file by setting the `activeByDefault` element equal to `true` as shown in **Listing 9**.

Each profile contains the dependencies for the container, and the active container can be switched by changing the `activeByDefault` element. Note that some IDEs also have the option to allow different Maven profiles to be chosen from a user interface.

Going Further

The Arquillian framework includes a number of test enrichers that can be used within test classes to further coordinate the tests and control execution. For instance, as demonstrated in previous examples, injection points can be made in fields by using the `@Inject` annotation, or they can be made directly in a test method by declaring them as arguments.

Also shown in the examples, the `@EJB` enricher is used to inject session beans into a test class. Other enrichers include `@PersistenceContext`, `@PersistenceUnit`, and `@Resource`. The first two are self-explanatory; the `@Resource` annotation is used to inject any object that is available to the Java Naming and Directory Interface (JNDI). This option is very useful for performing remote container tests against other available JNDI resources.

Tests can be run in two modes: container mode and client mode. Container mode is the default, and it provides the ability to repackage the `@Deployment` by adding Arquillian support classes, so it can be tested remotely in a container. The client mode option does not repackage the `@Deployment` or deploy to a remote server. Rather, the test cases run in the JVM so clients can test the container from outside. To run in client mode, annotate the static deployment method as follows:

```
@Deployment(testable=false)
```

To run both container and client tests, declare the deployment with `testable=true`, and annotate any test methods to be run in client mode with `@RunAsClient`. One reason to run as a client would be so the front end can be tested with the help of the Warp extension. The Warp extension enables you to write client-side tests and assert server-side logic, allowing you to cover integration tests across both the client and server.

Extensions

Several extensions can be paired with Arquillian to perform different types of testing. As mentioned earlier, the Warp extension can be used to write client-side tests asserting server-side logic. Another popular extension is Graphene, which can be especially useful for testing front ends built with JavaServer Faces (JSF) or Spring MVC. It uses Selenium WebDriver to provide a programmatic way to communicate with the browser, so you can use Java code to fill in forms or navigate web pages through business logic and testing behavior via assertions. Along the way, content can be validated so tests can ensure the proper functionality is being achieved within the UI. Drone is yet another extension that works with Selenium WebDriver to help manage the life cycle of the browser and inject the browser into the testing class. The Graphene extension depends upon Drone and Selenium, so you'll need to add some dependencies to the project in order to utilize these extensions. The `@Deployment` for a Graphene test should be executed in "client" mode, so it should contain the `testable=false` attribute. The deployment package for testing a user interface must contain each of the views that will be tested. Therefore, if the application contains a view named `customer.xhtml`, then the `@Deployment` method should add this resource, as follows:

```
.addAsWebResource("/customer.xhtml");
```

Graphene enables you to develop "page objects" and "page fragments," which makes possible the testing of separate pages and functionality contained within those pages by using Java, rather than markup. I won't dive into detail on these concepts in this article, but I strongly recommend looking at these features for your testing needs.

Conclusion

The Arquillian framework is an essential tool for any Java EE/Jakarta EE developer, because it enables the testing of components that are primarily utilized by enterprise applications. The framework supports fine-tuned and focused testing, allowing you to include only selected classes and features in test deployments. It is also a mature testing framework for which many extensions have been created, which enable testing of technologies used in enterprise development, such as JSF, REST, Servlet, and Spring MVC.

Also in This Issue

Josh Juneau

Josh Juneau (@javajuneau) works as an application developer, system analyst, and database administrator. He primarily develops using Java and other JVM languages. He is a frequent contributor to Oracle Technology Network and *Java Magazine* and has written several books for Apress about Java and Java EE. Juneau was a JCP Expert Group member for JSR 372 and JSR 378. He is a member of the NetBeans Dream Team, a Java Champion, leader for the CJUG OSS Initiative, and a regular voice on the *JavaPubHouse Off* Heap podcast.