BND Testing Harness

BND Testing Harness

The latest versions of BND have testing harness capabilities for JUnit built in. In fact, the OSGi Alliance uses BND for its own build and testing system. BND already provided lots of support for creating bundles, so testing is perhaps a logical progression of those capabilities. This task was simplified by a proposed standard framework launching and embedding API for the OSGi R4.2 specification, which BND uses to configure and launch OSGi frameworks in a platform-independent way. I should note that this BND functionality is in its early stages, so there are some rough edges.

This document describes an example BND project used to test Felix' Framework and Logger subprojects. This document is not intended as a BND tutorial, since it is beyond my capabilities to describe how BND works. The goal is to document my hacked up example sufficiently so that other people can try it out for themselves. The original build files and configuration for this example are based on an example from Peter Kriens. BND also has an Eclipse plugin which makes it easy to run the test cases, but I have not experimented with that; this document works solely with the command line.

With that in mind, let's get started.

Prerequisites

This document assumes you have Ant, Maven, and Subversion installed. If you don't, please do this first.

Getting Started

The first thing you need to do is check out the example with the following command:

```
svn co http://svn.apache.org/repos/asf/felix/sandbox/rickhall/bnd-test
```

The above command will create a directory called `bnd-test`, so go into that directory. Before we can perform the tests, we need to get some dependent libraries for our setup. Similar to Maven, BND has a repository concept, which in this example will be rooted in `bnd-test/cnf/repo`. However, this example does not use this repository except for the BND library itself; instead, we leverage Maven's existing repositories to get our dependencies. Luckily, BND's repository has a plugin to get it to search the local Maven repository for needed JAR files, all we need to do is populate our local Maven repository with what we need. The example includes a `pom.xml` file for this purpose, so we just need to type:

```
mvn install
```

The sole purpose of this POM file is to install our dependencies in the local Maven repository (i.e., `~/.m2/repository/`) so BND can find them. As a byproduct, it creates a `target/` directory, but this can be safely ignored. Using Maven is not strictly necessary, since we could just use BND's repository directly, but then we would need some way to populate its repository or would have to check the binaries into SVN. So, our Maven hybrid approach is not necessarily optimal, but it does allow us to capture the dependencies in a centralized POM file and easily populate the repository. Now we should be good to go.

The Setup

The important parts of the example are in the following directories:

- `cnf` - This contains the default build setup and the BND repository.
- `org.apache.felix.framework.test` - This contains some test cases for the Felix Framework.
- `org.apache.felix.framework.bootdelegation` - This contains a boot delegation test case for the Felix Framework.
- `org.apache.felix.log` - This contains the Felix Log Service and test cases.

We will discuss the organization of each of these.

`cnf`

The default Ant build file `cnf/build.xml` is sufficiently generic and I didn't need to modify it, although it could probably be further cleaned up. The `cnf/build.bnd` sets up a lot of properties for BND, most of which I didn't touch. The main thing I modified in here was changing the `-runpath` to use the 1.5.0 version of Felix' Framework and configured BND's Maven plugin to use "`org.apache.felix`" as its `groupId`.

`org.apache.felix.framework.test`

This example is somewhat special since it tests the framework itself instead of a bundle, but even then it works basically the same as testing a bundle except that the framework is built separately. The project's `build.xml` file just includes the generic one from `cnf`, while the `bnd.bnd` file contains of the BND commands to create the resulting test bundle. We also use this file to specify the build path for the project and to indicate which classes are the test cases.

The source for the project is contained in the `src/` directory, but this is only source for the test cases. The `recipes/` directory contains additional `.bnd` files, which describe other bundles that get created and embedded into the resulting project bundle. If you look at the `Include-Resource` line in `bnd.bnd`, you can see that it instructs BND to include JAR files with names corresponding to the `.bnd` files in the `recipes/` directory. This is a cool feature, since it allows you to have a bunch of test code in the same project and generate any number of bundles from it, but these bundles don't actually exist in the file system or have to be managed. When embedding JAR files, BND searches for the JAR file or a `.bnd` file with a corresponding name to generate the JAR file. (This rule is actually set in the `cnf/build.bnd` file.)

`org.apache.felix.framework.bootdelegation`

This example, like the last, tests the framework itself. In theory, this test case could have been combined with the previous test cases, but since it is testing boot delegation, I didn't want it to potentially interfere with the other tests. It actually configures the `org.osgi.framework.bootdelegation` property in its `bnd.bnd`, so when BND launches the framework it uses this configuration property to determine how it performs boot delegation. If we combined this with the previous tests, then all of those tests would be impacted by this boot delegation value, which could cause some of them to behave improperly.

This is the case because all test cases within a given project are executed in one session in the same framework instance, which is a good reason that framework tests should always clean up after themselves (i.e., make sure all installed bundles are removed) because any leftover artifacts could interfere with subsequent tests. Therefore, if you have a test case that may interfere with other tests, you may want to create a separate project for it, like this one, so it runs in its own framework instance.

Like the previous project, the source for the test case is in the `src/` directory. This example does not embed any additional bundles.

`org.apache.felix.log`

This example tests the Felix Log Service bundle and actually demonstrates how you can include your tests cases within your project. The project's `build.xml` file just includes the generic one from `cnf`, while the `bnd.bnd` file contains the BND commands to create the resulting log service bundle. As before, we also specify the build path and the test cases for the project.

The source for the project is contained in the `src/` directory, which contains both the source for the Log Service and the test cases. This example doesn't embed any additional bundles.

Running the Examples

For the most part, all projects work the same way. We build them by typing `ant` and we can build and execute the tests by typing `ant test`. The results of the build are placed in the `tmp/` directory. To clean up, type `ant clean`, that's it. What actually happens is Ant is used to compile the source, while BND is used to create the resulting bundle. If you executed the tests, BND launches the framework, installs the bundle into it and starts it. Any test cases referenced in the `bnd.bnd` file are instantiated and their test methods are invoked. If the test cases have a `setBundleContext()` method, then they are injected with their bundle's context.

That is what is happening generically. Let's look in a little more detail at what each included example is doing.

`org.apache.felix.framework.test`

This example actually creates three test cases to test the framework itself, these test cases are:

- `org.apache.felix.framework.test.TestClassLoading` - This test case verifies various class loading behavior.
- `org.apache.felix.framework.test.TestConcurrency` - This test case tries to deadlock the framework.
- `org.apache.felix.framework.test.TestFilter` - This test case directly tests filter functionality.
- `org.apache.felix.framework.test.TestFragment` - This test case checks host and fragment resolution.
- `org.apache.felix.framework.test.TestResolver` - This test cases verifies resolver behavior.

All of these test cases extend `org.apache.felix.framework.test.FelixTestCase`, which provides some helper functions, such as acquiring the bundle context and using `PackageAdmin` to refresh the framework. All source code for embedded bundles is in the same package tree and the test cases install them into the framework as needed by getting an input stream to the JAR resource.

By typing `ant test`, you should be able to run the test cases and get no errors.

`org.apache.felix.framework.bootdelegation`

This example has a single framework test case, `org.apache.felix.framework.bootdelegation.BootDelegationTest`, which performs various test to make sure the framework properly obeys the boot delegation property. This test is fairly simple and just directly extends the JUnit test case.

By typing `ant test`, you should be able to run the test case and get no errors.

`org.apache.felix.log`

This example builds the Log Service bundle (in package `org.apache.felix.log`) with one included test case (`test.TestLog`). This test case simply tests whether logged messages are added to the log properly and whether log listeners are properly handled. The test case is fairly simple and just directly extends the JUnit test case. If you need to run other bundles in addition to the bundle you are testing, you can specify their bundle symbolic names in `bnd.bnd` with the `-testbundles` option, according to Peter Kriens.

By typing `ant test`, you should be able to run the test case and get no errors.

Conclusion

While not exhaustive, these examples show how BND can be used as a testing harness for the Felix Framework or as part of a complete build system for bundles. For my main use case, which is creating test cases for the framework, it definitely makes my life easier since I can just sit down and whip out some bundles to reproduce issues raised on the mailing lists without actually have to create separate bundle projects per se. Your mileage may vary.