Technology Compatibility Kit Reference Guide for JSR-303: Bean Validation

Specification Lead: Red Hat Inc.

by Emmanuel Bernard Red Hat Inc. and Hardy Ferentschik Red Hat Inc.

Preface

This guide describes how to download, install, configure, and run the Technology Compatibility Kit (TCK) used to verify the compatibility of an implementation of JSR-303: Bean Validation specification.

The Bean Validation TCK is built atop the JBoss Test Harness, a portable and configurable automated test suite for authoring unit and integration tests in a Java EE environment. The TCK uses the JBoss Test Harness version 1.x to execute the test suite.

The Bean Validation TCK is provided under the *Apache Public License 2.0* [http://www.apache.org/licenses/LICENSE-2.0].

1. Who Should Use This Guide

This guide is for implementors of the Bean Validation specification to assist in running the test suite that verifies the compatibility of their implementation.

2. Before You Read This Guide

The Bean Validation TCK is based on the Bean Validation specification 1.0 (JSR-303). Information about the specification, including links to the specification documents, can be found on the *JSR-303 JCP page* [http://jcp.org/en/jsr/detail?id=303].

Before running the tests in the Bean Validation TCK, read and become familiar with the JBoss Test Harness Reference Guide (pending), which describes how the test harness functions.

3. How This Guide Is Organized

If you are running the Bean Validation TCK for the first time, read *Chapter 1, Introduction* and *Chapter 8, Introduction* completely for the necessary background information about the TCK and the JBoss Test Harness, respectively. Once you have reviewed that material, perform the steps outlined in the remaining chapters.

- *Chapter 1, Introduction* gives an overview of the principles that apply generally to all Technology Compatibility Kits (TCKs), outlines the appeals process and describes the Bean Validation TCK architecture and components. It also includes a broad overview of how the TCK is executed and lists the platforms on which the TCK has been tested and verified.
- *Chapter 2, Appeals Process* explains the process to be followed by an implementor should they wish to challenge any test in the TCK.
- *Chapter 3, Installation* explains where to obtain the required software for the Bean Validation TCK and how to install it. It covers both the primary TCK components as well as tools useful for troubleshooting tests.
- *Chapter 4, Configuration* details the configuration of the JBoss Test Harness, how to create a TCK runner for the TCK test suite and the mechanics of how an in-container test is conducted.

- *Chapter 5, Reporting* explains the test reports that are generated by the TCK test suite and introduces the TCK audit report as a tool for measuring the completeness of the TCK in testing the JSR-303 specification and in understanding how testcases relate to the specification.
- *Chapter 7, Executing the Test Suite* documents how the TCK test suite is executed. It covers both modes supported by the TCK, standalone and in-container, and shows how to dump the generated test artifacts to disk.
- *Part III, "JBoss Test Harness"* includes excerpts from the JBoss Test Harness Reference Guide. How to configure the JBoss Test Harness as it relates to the Bean Validation TCK is presented in *Chapter 4, Configuration*. However, to aid in debugging or configuring the TCK in your environment, you may want to read in more detail how to use the JBoss Test Harness.

Part I. Getting Acquainted with the TCK

The Bean Validation TCK must be used to ensure that your implementation conforms to the Bean Validation specification. This part introduces the TCK, gives some background about its purpose, states the requirements for passing the TCK and outlines the appeals process.

In this part you will learn where to obtain the Bean Validation TCK and supporting software. You are then presented with recommendations of how to organize and configure the software so that you are ready to execute the TCK.

Finally, it discusses the reporting provided by the TCK.

Introduction

This chapter explains the purpose of a TCK and identifies the foundation elements of the Bean Validation TCK.

1.1. TCK Primer

A TCK, or Technology Compatibility Kit, is one of the three required pieces for any JSR (the other two being the specification document and the reference implementation). The TCK is a set of tools and tests to verify that an implementation of the technology conforms to the specification. The tests are the primary component, but the tools serve an equally critical role of providing a framework and/or set of SPIs for executing the tests.

The tests in the TCK are derived from assertions in the written specification document. The assertions are itemized in an XML document, where they each get assigned a unique identifier, and materialize as a suite of automated tests that collectively validate whether an implementation complies with the aforementioned assertions, and in turn the specification. For a particular implementation to be certified, all of the required tests must pass (i.e., the provided test suite must be run unmodified).

A TCK is entirely implementation agnostic. It should validate assertions by consulting the specficiation's public API.

1.2. Compatibility Testing

The goal of any specification is to eliminate portability problems so long as the program which uses the implementation also conforms to the rules laid out in the specification.

Executing the TCK is a form of compatibility testing. It's important to understand that compatibility testing is distinctly different from product testing. The TCK is not concerned with robustness, performance or ease of use, and therefore cannot vouch for how well an implementation meets these criteria. What a TCK can do is to ensure the exactness of an implementation as it relates to the specification.

Compatibility testing of any feature relies on both a complete specification and a complete reference implementation. The reference implementation demonstrates how each test can be passed and provides additional context to the implementor during development for the corresponding assertion.

1.2.1. Why Compatibility Is Important

Java platform compatibility is important to different groups involved with Java technologies for different reasons:

- Compatibility testing is the means by which the JCP ensures that the Java platform does not become fragmented as it's ported to different operating systems and hardware.

- Compatibility testing benefits developers working in the Java programming language, enabling them to write applications once and deploy them across heterogeneous computing environments without porting.
- Compatibility testing enables application users to obtain applications from disparate sources and deploy them with confidence.
- Conformance testing benefits Java platform implementors by ensuring the same extent of reliability for all Java platform ports.

The Bean Validation specification goes to great lengths to ensure that programs written for Java EE are compatible and the TCK is rigorous about enforcing the rules the specification lays down.

1.3. About the Bean Validation TCK

The Bean Validation TCK is designed as a portable, configurable and automated test suite for verifying the compatibility of an implementation of JSR-303: Bean Validation specification. The test suite is built atop TestNG and provides a series of extensions that allow runtime packaging and deployment of JEE artifacts for in-container testing (JBoss Test Harness).

Note

The Bean Validation TCK harness is based on the JBoss Test harness, which provides most of the aforementioned functionality.

Each test class in the suite acts as a deployable unit. The deployable units, or artifacts, are defined declaratively using annotations.

The declarative approach allows many of the tests to be executed in a standalone implementation of Bean Validation, accounting for a boast in developer productivity. However, an implementation is only valid if all tests pass using the in-container execution mode. The standalone mode is merely a developer convenience.

1.3.1. Bean Validation TCK Specifications and Requirements

This section lists the applicable requirements and specifications for the Bean Validation TCK.

- **Bean Validation API** - The Java API defined in the Bean Validation specification and provided by the reference implementation.
- **JBoss Test Harness** - The Bean Validation TCK requires version 1.x of the JBoss Test Harness. The Harness is based on *TestNG 5.x* [http://testng.org]. You can read more about the harness in *Part III, "JBoss Test Harness"*.
- **TCK Audit Tool** - An itemization of the assertions in the specification documents which are cross referenced by the individual tests. Describes how well the TCK covers the specification.

- **Reference runtime** - The designated reference runtimes for compatibility testing of the Bean Validation specification is the Sun Java Platform, Enterprise Edition (Java EE) 6 reference implementation (RI). See details at *Java EE 6* [http://java.sun.com/javaee/6/docs/api/]

1.3.2. Bean Validation TCK Components

The Bean Validation TCK includes the following components:

- **JBoss Test Harness 1.x** and related documentation.
- **TestNG 5.9**, the testing framework on which the JBoss Test Harness is based and which provides the extension points for selecting an executing the tests in the test suite.
- **The test suite**, which is a collection of TestNG tests, the TestNG test suite descriptor and supplemental resources that configure Bean Validation and other software components.
- **The TCK audit** is used to list out the assertions identified in the Bean Validation specification. It matches the assertions to testcases in the test suite by unique identifier and produces a coverage report.

  The audit document is provided along with the TCK. Each assertion is defined with a reference to a chapter, section and paragraph from the specification document, making it easy for the implementor to locate the language in the specification document that supports the feature being tested.
- **TCK documentation** accompanied by release notes identifying updates between versions.

The Bean Validation TCK has been tested run on following platforms:

- JBoss AS 5.1.0.GA using Sun Java SE 6 on Red Hat Enterprise Linux 5.2

Appeals Process

While the Bean Validation TCK is rigourous about enforcing an implementation's conformance to the JSR-303 specification, it's reasonable to assume that an implementor may discover new and/or better ways to validate the assertions. This chapter covers the appeals process, defined by the Specification Lead, Red Hat Middleware LLC., which allows implementors of the JSR-303 specification to challenge one or more tests defined by the Bean Validation TCK.

The appeals process identifies who can make challenges to the TCK, what challenges to the TCK may be submitted, how these challenges are submitted, how and by whom challenges are addressed and how accepted challenges to the TCK are managed.

Following the recent adoption of transparency in the JCP, implementors are encouraged to make their appeals public, which this process facilitates. The JCP community should recognize that issue reports are a central aspect of any good software and it's only natural to point out shortcomings and strive to make improvements. Despite this good faith, not all implementors will be comfortable with a public appeals process. Instructions about how to make a private appeal are therefore provided.

2.1. Who can make challenges to the TCK?

Any implementor may submit an appeal to challenge one or more tests in the Bean Validation TCK. In fact, members of the JSR-303 Expert Group (EG) encourage this level of participation.

2.2. What challenges to the TCK may be submitted?

Any test case (e.g., `@Artifact` class, `@Test` method), test case configuration (e.g., validation.xml), test entities, annotations and other resources may be challenged by an appeal.

What is generally not challengable are the assertions made by the specification. The specification document is controlled by a separate process and challenges to it should be handled through the JSR-303 EG by sending an e-mail to *jsr303-comments@jcp.org* [mailto:jsr299-comments@jcp.org].

2.3. How these challenges are submitted?

To submit a challenge, a new issue should be created in the *BVTCK project* [http://opensource.atlassian.com/projects/hibernate/browse/BVTCK] of the Hibernate JIRA using the Issue Type: Bug. The appellant should complete the Summary, Component (TCK Appeal), Environment and Description Field only. Any communication regarding the issue should be pursed in the comments of the filed issue for accurate record.

To submit an issue in the Hibernate JIRA, you must have a (free) Jira member account. You can create a member account using the *on-line registration* [http://opensource.atlassian.com/projects/hibernate/secure/Signup!default.jspa].

If you wish to make a private challenge, you should follow the above procedure, setting the Security Level to Private. Only the issue reporter, TCK Project Lead and designates will be able to view the issue.

2.4. How and by whom challenges are addressed?

The challenges will be addressed in a timely fashion by the Bean Validation TCK Project Lead, as designated by Specification Lead, Red Hat Middleware LLC. or his/her designate. The appellant can also monitor the process by following the issue report filed in the *BVTCK project* [http://opensource.atlassian.com/projects/hibernate/browse/BVTCK] of the Hibernate JIRA.

The current TCK Project Lead is listed on the *BVTCK Project Summary Page* [http://opensource.atlassian.com/projects/hibernate/browse/BVTCK].

2.5. How accepted challenges to the TCK are managed?

Accepted challenges will be acknowledged via the filed issue's comment section. Communication between the Bean Validation TCK Project Lead and the appellant will take place via the issue comments. The issue's status will be set to "Resolved" when the TCK project lead believes the issue to be resolved. The appellant should, within 30 days, either close the issue if they agree, or reopen the issue if they do not believe the issue to be resolved.

Resolved issue not addressed for 30 days will be closed by the TCK Project Lead. If the TCK Project Lead and appellant are unable to agree on the issue resolution, it will be referred to the JSR-303 specification lead or his/her designate.

Installation

This chapter explains how to obtain the TCK and supporting software and provides recommendations for how to install/extract it on your system.

3.1. Obtaining the Software

You can obtain a release of the Bean Validation TCK project via the official *Bean Validation home page* [http://beanvalidation.org/1.0/]. The Bean Validation TCK is distributed as a ZIP file, which contains the TCK artifacts (the test suite binary and source, the test suite descriptor, the audit source and report), the TCK library dependencies in `/lib` and documentation in `/doc`.

You can also download the current source code from Github - *https://github.com/beanvalidation/beanvalidation-tck*.

The TCK project is available in the JBoss Maven 2 repository as org.hibernate.jsr303.tck:org.hibernate.jsr303.tck; the POM defines all dependencies required to run the TCK.

Executing the TCK requires a Java EE 5 or better runtime environment (i.e., application server), to which the test artifacts are deployed and the individual tests are invoked. The TCK does not depend on any particular Java EE implementation.

The JSR-303: Bean Validation reference implementation (RI) project is named Hibernate Validator. You can obtain the latest Hibernate Validator release from the *download page* [http://www.hibernate.org/subprojects/validator/download] on Hibernate website.

Note

Hibernate Validator is not required for running the Bean Validation TCK, but it can be used as a reference for familiarizing yourself with the TCK before testing your own Bean Validation implementation.

3.2. The TCK Environment

The TCK requires the following two Java runtime environments:

- Java 5 or better
- Java EE 5 or better (e.g., JBoss AS 5.x or GlassFish V3)

You should refer to vendor instructions for how to install the runtime.

The rest of the TCK software can simply be extracted. It's recommended that you create a folder named jsr303 to hold all of the jsr303-related projects. Then, extract the TCK distribution into a

subfolder named tck. You can also check out the full Hibernate Validator source into a subfolder ri. This will allow you to run the TCK against Hibernate Validator.

```
git clone git://github.com/hibernate/hibernate-validator.git ri
git checkout 4.2.0.Final
```

If you have downloaded the Hibernate Validator distribution, extract it into a sibling folder named hibernate-validator. The resulting folder structure is shown here:

```
jsr303/
   ri/
   tck/
```

Each test class is treated as an individual artifact (hence the `@Artifact(artifactType = ArtifactType.JSR303)` annotation on the class). All test methods (i.e., methods annotated with `@Test`) in the test class are run.

Note

Using `ArtifactType.JSR303` is an additional artifact type to the standard WAR and EAR types offered by the JBoss Test Harness. It is basically a normal WAR file with a `validation.xml` file at the root of the classpath. The `validation.xml` file can be specified with the `@ValidationXml` annotation. Using `ArtifactType.JSR303` together with `@ValidationXml` also allows to have test with custom validation.xml files running in standalone mode. This is possible by modifying the classloader in standalone mode as shown in `org.hibernate.jsr303.tck.util.StandaloneContainersImpl`. This container implementation is also the default for sandalone mode.

Running the TCK against the Bean Validation RI (Hibernate Validator) and JBoss AS 7

- First, you should download the latest JBoss AS 7 release from the JBoss AS *project page* [http://jboss.org/jbossas/downloads].
- Set the `JBOSS_HOME` environment variable to the location of the JBoss AS software.
- Change to the `ri/hibernate-validator-tck-runner` directory.
- You need to install Maven. You can find documention on how to install Maven 2 in the *Maven: The Definitive Guide* [http://www.sonatype.com/books/

maven-book/reference/installation-sect-maven-install.html] book published by Sonatype. Web Beans bundles a copy of Maven in the `lib/maven` directory.

- Next, instruct Maven to run the TCK:

```
mvn test -Dincontainer
```

- TestNG will report, via Maven, the outcome of the run, and report any failures on the console. Details can be found in `target/surefire-reports/TestSuite.txt`.

Configuration

This chapter lays out how to configure the TCK Harness by specifying the SPI implementation classes, defining the target container connection information, and various other switches. You then learn how to setup a TCK runner project that executes the the TCK test suite, putting these settings into practice. Finally, a detailed account of how the JBoss Test Harness negotiates the execution of the tests in the container is given.

This chapter does not discuss in detail how to use the TCK in standalone mode. The JBoss Test Harness guide provides more on running in standalone mode.

4.1. TCK Harness Properties

The JBoss Test Harness allows the test suite to be launched in a pluggable fashion. In order to execute the TCK, the JBoss Test Harness must be configured by specifying implementations of the test launcher and container APIs.

System properties and/or the resource `META-INF/jboss-test-harness.properties`, a Java properties file, are used to configure the JBoss Test Harness. You can read more about configuring the JBoss Test Harness in *Section 9.1, "JBoss Test Harness Properties"*.

You should set the following properties:

Table 4.1. Required JBoss Test Harness Configuration Properties

Property = Required/Example Value	Description
`org.jboss.testharness.libraryDirectory=/path/to/extra/libraries`	Directory containing extra JARs you want placed in artifact library directory.
`org.jboss.testharness.standalone=false`	You must run the tests in-container to pass the TCK
`org.jboss.testharness.runIntegrationTests=true`	You must run the integration tests to pass the TCK
`org.jboss.testharness.spi.Containers=com.acme.AcmeContainer`	The container implementation for deploying and executing in-container tests. See *Note*
`org.jboss.testharness.api.TestLauncher=org.jboss.testharness.impl.runner.servlet.ServletTestLauncher`	You should use the `ServletTestLauncher` for Java EE 6 and Java EE 6 Web Profile.

To run the full TCK you must additionally implement `org.jboss.testharness.spi.Containers`, which handles deploying the test artifact to the container. An implementations of this API is already available for JBoss AS 5.1. Therefore, you only need to implement this if you wish to use another container.

Note

Red Hat Middleware LLC encourages Bean Validation implementators to contribute JBoss Test Harness Deployment API implementations for other containers under the ASL license. Please contact the Bean Validation TCK lead.

4.2. Configuring TestNG to execute the TCK

The JBoss Test Harness is built atop TestNG, and it's TestNG that is responsible for selecting the tests to execute, the order of execution, and reporting the results. Detailed TestNG documentation can be found at *testng.org* [http://testng.org/doc/documentation-main.html].

The `tck-tests.xml` artifact provided in the TCK distribution must be run by TestNG 5.9 (described by the TestNG documenation as "with a `testng.xml` file") unmodified for an implementation to pass the TCK. This file also allows tests to be excluded from a run:

```
<suite name="JSR-303 TCK" verbose="2">
   <test name="JSR-303 TCK">
      ...
      <classes>
         <class name="org.hibernate.jsr303.tck.tests.bootstrap.ValidationProviderTest">
            <methods>
               <exclude name="testFirstMatchingValidationProviderResolverIsReturned"/>
            </methods>
         </class>
      </classes>
      ...
   </test>
</suite>
```

TestNG provides extensive reporting information. Depending on the build tool or IDE you use, the reporting will take a different format. Please consult the TestNG documentation and the tool documentation for more information.

4.3. Configuring your build environment to execute the TCK

It's beyond the scope of this guide to describe in how to set up your build environment to run the TCK. The JBoss Test Harness guide describes how Bean Validation uses Maven 2 to execute the Bean Validation TCK. See *Section 10.1, "Building a test suite runner using Maven 2"*. The

TestNG documentation provides extensive information on launching TestNG using the Java, Ant, Eclipse or IntellJ IDEA.

Reporting

This chapter covers the two types of reports that can be generated from the TCK, an assertion coverage report and the test execution results. The chapter also justifies why the TCK is good indicator of how accurately an implementation conforms to the JSR-303 specification.

5.1. Bean Validation TCK Coverage Metrics

The Bean Validation TCK coverage has been measured as follows:

- **Assertion Breadth Coverage**

  The Bean Validation TCK provides at least 100% coverage of identified assertions with test cases.

- **Assertion Breadth Coverage Variance**

  The coverage of specification sub-sections shows at least a normal distribution (centered around 75%).

- **Assertion Depth Coverage**

  The assertion depth coverage has not been measured, as, when an assertion requires more than one testcase, these have been enumerated in an assertion group and so are adequately described by the assertion breadth coverage.

- **API Signature Coverage**

  The Bean Validation TCK covers 100% of all API public methods using the Java CTT Sig Test tool.

5.2. Bean Validation TCK Coverage Report

A specification can be distilled into a collection of assertions that define the behavior of the software. This section introduces the Bean Validation TCK coverage report, which documents the relationship between the assertions that have been identified in the JSR-303 specification document and the tests in the TCK test suite.

The structure of this report is controlled by the assertion document, so we'll start there.

5.2.1. Bean Validation TCK Assertions

The Bean Validation TCK developers have analyzed the JSR-303 specification document and identified the assertions that are present in each chapter. Here's an example of one such assertion found in section 2.1:

The assertions are listed in the XML file `tck-audit.xml` in the Bean Validation TCK distribution. Each assertion is identified by the section of the specification document in which it resides and assigned a unique paragraph identifier to narrow down the location of the assertion further. To continue with the example, the assertion shown above is listed in the tck-audit.xml file using this XML fragment:

```
<section id="2.1.1" title="Constraint definition properties">
  ...
   <assertion id="c">
      <text>Every constraint annotation must define a message element of type String</text>
   </assertion>
  ...
</section>
```

The strategy of the Bean Validation TCK is to write a test which validates this assertion when run against an implementation. A test case (a method annotated with `@Test` in an `@Artifact` class) is correlated with an assertion using the `@org.jboss.test.audit.annotations.SpecAssertion` annotation as follows:

```
@Test
@SpecAssertion(section = "2.1.1", id = "c")
public void testConstraintDefinitionWithoutMessageParameter() {
   try {
       Validator validator = TestUtil.getValidatorUnderTest();
       validator.validate( new DummyEntityNoMessage() );
        fail( "The used constraint does not define a message parameter. The validation should
 have failed." );
   }
   catch ( ConstraintDefinitionException e ) {
       // success
}
```

To help evaluate the distribution of coverage for these assertions, the TCK provides a detailed coverage report. This report is also useful to help implementors match tests with the language in the specification that supports the behavior being tested.

5.2.2. Producing the Coverage Report

The coverage report is an HTML report generated as part of the TCK project build. Specifically, it is generated by an annotation processor that attaches to the compilation of the classes in the TCK test suite, another tool from the JBoss Test Utils project. You can enable this report by setting the commandline property tck-audit to true when running the Maven 2 build in the tck directory.

```
mvn clean install -Dtck-audit=true
```

Note

You must run clean first because the annotation processor performs it's work when the test class is being compiled. If compilation is unnecessary, then the assertions referenced in that class will not be discovered.

The report is written to the file target/coverage.html in the same project. The report has five sections:

1. **Chapter Summary** - List the chapters (that contain assertions) in the specification document along with total assertions, tests and coverage percentage.
2. **Section Summary** - Lists the sections (that contain assertions) in the specification document along with total assertions, tests and coverage percentage.
3. **Coverage Detail** - Each assertion and the test that covers it, if any.
4. **Unmatched Tests** - A list of tests for which there is no matching assertion (useful during TCK development).
5. **Unversioned Tests** - A list of tests for which there is no `@SpecVersion` annotation on the test class (useful during TCK development).

The coverage report is color coded to indicate the status of an assertion, or group of assertions. The status codes are as follows:

- **Covered** - a test exists for this assertion
- **Not covered** - no test exists for this assertion
- **Problematic** - a test exists, but is currently disabled. For example, this may be because the test is under development
- **Untestable** - the assertion has been deemed untestable, a note, explaining why, is normally provided

For reasons provided in the `tck-audit.xml` document and presented in the coverage report, some assertions are not testable.

The coverage report does not give any indication as to whether the tests are passing. That's where the TestNG reports come in.

5.2.3. TestNG Reports

As you by now, the Bean Validation TCK test suite is really just a TestNG test suite. That means an execution of the Bean Validation TCK test suite produces all the same reports that TestNG produces. This section will go over those reports and show you were to go to find each of them.

5.2.3.1. Maven 2, Surefire and TestNG

When the Bean Validation TCK test suite is executed during the Maven 2 test phase of the TCK runner project, TestNG is invoked indirectly through the Maven Surefire plugin. Surefire is a test execution abstraction layer capable of executing a mix of tests written for JUnit, TestNG, and other supported test frameworks.

Why is this relevant? It means two things. First, it means that you are going to get a summary of the test run on the commandline. Here's the output generated when the tests are run using standalone mode.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running TestSuite
Tests run: 237, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 11.062 sec

Results :

Tests run: 237, Failures: 0, Errors: 0, Skipped: 0
```

Note

The number of tests executed, the execution time, and the output will differ when you run the tests using in-container mode as the Bean Validation TCK requires.

If the Maven reporting plugin that compliments Surefire is configured properly, Maven will also generate a generic HTML test result report. That report is written to the file test-report.html in the target/surefire-reports directory of the TCK runner project. It shows how many tests were run, how many failed and the success rate of the test run.

The one drawback of the Maven Surefire report plugin is that it buffers the test failures and puts them in the HTML report rather than outputting them to the commandline. If you are running the test suite to determine if there are any failures, it may be more useful to get this information in the foreground. You can prevent the failures from being redirected to the report using the following commandline switch:

```
mvn test -Dsurefire.useFile=false
```

The information that the Surefire provides is fairly basic and the detail pales in comparison to what the native TestNG reports provide.

5.2.3.2. TestNG HTML Reports

TestNG produces several HTML reports for a given test run. All the reports can be found in the target/surefire-reports directory in the TCK runner project. Below is a list of the three types of reports:

- Test Summary Report
- Test Suite Detail Report
- Emailable Report

The first report, the test summary report is written to the file index.html. It produces the same information as the generic Surefire report.The summary report links to the test suite detail report, which has a wealth of information. It shows a complete list of test groups along with the classes in each group, which groups were included and excluded, and any exceptions that were raised, whether from a passed or failed test. A partial view of the test suite detail report is shown below.

The test suite detail report is very useful, but it borderlines on complex. As an alternative, you can have a look at the emailable report, which is a single HTML document that shows much of the same information as the test suite detail report in a more compact layout.

Part II. Executing and Debugging Tests

In this part you learn how to execute the Bean Validation TCK on the Bean validation reference implementation (Hibernate Validator). You are walked through the steps necessary to execute the test suite against Hibernate Validator and you discover how to modify the TCK runner to execute the test suite against your own implementation. First, however, you learn how to pass the Signature Test.

Running the Signature Test

One of the requirements of an implementation passing the TCK is for it to pass the Bean Validation signature test. This section describes how the signature file is generated and how to run it against your implementation.

6.1. Obtaining the sigtest tool

You can obtain the Sigtest tool from the *Sigtest home page* [https://wiki.openjdk.java.net/display/CodeTools/SigTest]. The TCK uses version 3_0-dev-bin-b09-24_apr_2013 which can be obtained from the SigTest *download page* [http://download.java.net/sigtest/download.html]. The user guide can be found *here* [http://download.oracle.com/javame/test-tools/sigtest/2_2/sigtest2_2_usersguide.pdf] (the latest published documentation version is 2.2 but this documentation still applies to SigTest 3.0 in general). The downloadable package contains the jar files used in the commands below.

6.2. Creating the signature file

The TCK package contains the files `validation-api-java5.sig`, `validation-api-java6.sig`, `validation-api-java7.sig` and `validation-api-java8.sig` (in the `artifacts` directory) which were created using the following command (using the corresponding Java version):

```
java -jar sigtestdev.jar Setup -classpath $JAVA_HOME/jre/lib/rt.jar:lib/validation-api-1.0.0.GA.jar -package javax.validation -filename validation-api-java6.sig
```

In order to pass the Bean Validation TCK you have to make sure that your API passes the signature tests against `validation-api.sig`.

6.3. Running the signature test

To run the signature test use:

```
java -jar sigtest.jar Test -classpath $JAVA_HOME/jre/lib/rt.jar:lib/validation-api-1.0.0.GA.jar
 -static -package javax.validation -filename validation-api-java6.sig
```

You have to chose the right version of the signature file depending on your Java version. In order to run against your own Bean Validation API replace validation-api-1.0.0.GA.jar with your own API jar. You should get the message `"STATUS:Passed."`.

6.4. Forcing a signature test failure

Just for fun (and to confirm that the signature test is working correctly), you can try the following:

1) Edit validation-api.sig

2) Modify one of the class signatures - in the following example we change one of the constructors for `ValidationException` - here's the original:

```
CLSS public javax.validation.ValidationException
cons public ValidationException()
cons public ValidationException(java.lang.String)
cons public ValidationException(java.lang.String,java.lang.Throwable)
cons public ValidationException(java.lang.Throwable)
supr java.lang.RuntimeException
```

Let's change the default (empty) constructor parameter to one with a `java.lang.Integer` parameter instead:

```
CLSS public javax.validation.ValidationException
cons public ValidationException(java.lang.Integer)
cons public ValidationException(java.lang.String)
cons public ValidationException(java.lang.String,java.lang.Throwable)
cons public ValidationException(java.lang.Throwable)
supr java.lang.RuntimeException
```

3) Now when we run the signature test using the above command, we should get the following errors:

```
Missing Constructors
--------------------

javax.validation.ValidationException:                              constructor          public
 javax.validation.ValidationException.ValidationException(java.lang.Integer)

Added Constructors
------------------

javax.validation.ValidationException:                              constructor          public
 javax.validation.ValidationException.ValidationException()


STATUS:Failed.2 errors
```

Executing the Test Suite

This chapter explains how to run the TCK on Hibernate Validator as well as your own implementation. The Bean Validation TCK uses the Maven 2 TestNG plugin and the JBoss Test Harness to execute the test suite. Learning to execute the test suite from Maven 2 is prerequisite knowlege for running the tests in an IDE, such as Eclipse.

7.1. The Test Suite Runner

The test suite is executed by the Maven 2 TestNG plugin during the test phase of the Maven 2 life cycle. The execution happens within a TCK runner project (as opposed to the TCK project itself). Hibernate Validator includes a TCK runner project (hibernate-validator-tck-runner) that executes the Bean Validation TCK on Hibernate Validator running inside JBoss AS 5.1. To execute the Bean Validation TCK on your own Bean Validation implementation, you could modify the TCK runner project included with Hibernate Validator to use your Bean Validation implementation as described in *Chapter 4, Configuration*.

Note

For the Bean Validation TCK to run the system property validation.provider has to be specified as system property. The value has to be the fully specified class name of the `ValidationProvider` of your Bean Validation Implementation. In the case of Hibernate Validator this is `org.hibernate.validator.HibernateValidator`. This applies for standalone as well as in-container mode.

7.2. Running the Tests In Standalone Mode

To execute the TCK test suite against Bean Validation, first switch to the `hibernate-validator-tck-runner` directory in the checked out Hibernate Validator distribution:

```
cd ri/hibernate-validator-tck-runner
```

Note

These instructions assume you have extracted the Bean Validation related software according to the recommendation given in *Section 3.2, "The TCK Environment"*.

Then execute the Maven 2 life cycle through the test phase:

```
mvn test
```

Without any command-line flags, the test suite is run in standalone mode, which means that any test class with the `@org.jboss.testharness.impl.packaging.IntegrationTest` annotation is skipped. This mode uses the `StandaloneContainers` SPI to invoke the test artifact within a mock Java EE life cycle and capture the results of the test. However, passing the suite in this mode is not sufficient to pass the TCK as a whole. The suite must be passed while executing using the in-container mode.

7.3. Running the Tests In the Container

To execute the test suite using in-container mode with the JBoss TCK runner, you first have to setup JBoss AS as described in the *Running the TCK against the Bean Validation RI (Hibernate Validator) and JBoss AS 7* callout.

Then, execute the TCK runner with Maven 2 as follows:

```
mvn test -Dincontainer
```

The presence of the `incontainer` property activates a Maven 2 profile that assigns the `org.jboss.testharness.standalone` system property to `false` and the `org.jboss.testharness.runIntegrationTests` system property to `true`, hence activating the in-container test mode. This time, all the test artifacts in the test suite are executed.

The in-container profile will also start and stop the application server automatically, a feature which the profile activates by setting the `org.jboss.testharness.container.forceRestart` to `true`.

The in-container mode uses the `Containers` SPI to deploy the test artifact to the container and execute the test in a true Java EE life cycle. The JBoss TCK runner has a dependency on the library that provides an implementation of this interface for JBoss AS 5.1.

Since in-container tests are executed in a remote JVM, the results of the test must be communicated back to the runner over a container-supported protocol. The JBoss Test Harness provides servlet-based communication over HTTP as described in *Section 8.1, “Negotiating the execution of an in-container test”*.

7.4. Dumping the Test Artifacts

As you have learned, when the test suite is executing using in-container mode, each test class is packaged as a deployable artifact and deployed to the container. The test is then executed within the context of the deployed application. This leaves room for errors in packaging. When investigating a test failure, you may find it helpful to inspect the artifact after it's generated. The TCK can accommodate this type of inspection by "dumping" the generated artifact to disk.

The feature just described is activated in the jboss-tck-runner project by appending the `dumpArtifacts` command line property to the end of the command that invokes the Maven 2 test phase.

```
mvn test-compile -DdumpArtifacts
```

The directory where the artifacts get written is configured using the `org.jboss.testharness.outputDirectory` property. The `dumpArtifacts` profile in the jboss-tck-runner project sets this value to the relative directory path `target/jsr303-artifacts`.

You can read more about this feature in *Section 10.2, "Dumping the Test Artifacts to Disk"*.

Part III. JBoss Test Harness

In this part you learn about the JBoss Test Harness through selected chapters from the JBoss Test Harness Reference Guide.

Introduction

This chapter explains the purpose of the test harness and describes its key features.

The JBoss Test Harness is a testing framework based on TestNG that provides a series of extensions that allow runtime packaging and deployment of Java EE artifacts (EAR or WAR) for in-container testing. It's important to note that the JBoss Test Harness has no relation with, or dependency on, the JBoss Application Server (JBoss AS).

Note

You'll often see the term *in-container* used in this reference guide. This term refers to running the test suite in any of the aforementioned environments, whilst *standalone* refers to running the tests outside the container via an implementation-specific standalone bootstrap. The standalone mode only runs those tests which the Bean Validation RI can run without deployment in a Java EE container.

The last thing Java developers want is yet another testing framework to make their life more complicated. That's why the JBoss Test Harness is built entirely upon TestNG. TestNG is one of two prominent test frameworks for Java (the other being JUnit). Furthermore, what developers want is a good integration with their Integrated Development Environment (IDE). These days, if a tool doesn't have an IDE plugin, then it won't get the attention it deserves. TestNG plugins are available for all major IDEs and build tools (Ant and Maven 2). Again, a motivating factor for extending TestNG.

Because it leverages the existing TestNG ecosystem, there is no need for a special test launcher for the JBoss Test Harness. You simply use the IDE or build tool of your choice (so long as it has TestNG support). You also get reporting and debugging for free (various reporting plugins are provided for TestNG).

You can read more about TestNG at *testng.org* [http://testng.org/doc/documentation-main.html].

The JBoss Test Harness supports the following features:

- Test activation via any method supported by the TestNG configuration descriptor (package, group, class)
- Exclusion of in-container tests in standalone mode
- Exclusion of individual tests labeled as under investigation
- Integration with any TestNG plugin (Eclipse, IntelliJ, Ant, Maven)
- Automated reporting capability as provided by TestNG

- Standalone and in-container test mode
- Container pluggability
- Declarative packaging of additional resources and classes in artifact
- Declarative deployment exception trapping
- Artifact dumping for failure and packaging analysis

A test is designated by a method annotated with `@org.testng.annotations.Test` in a class which extends `org.jboss.testharness.AbstractTest` and is annotated with `@org.jboss.testharness.impl.packaging.Artifact`.

Note

Test suites may often choose to extend `AbstractTest` and require tests to extend that base class. In fact, both the CDI TCK and the Bean Validation TCK provide base classes that extend `AbstractTest` to provide functionality specific to the needs of the TCK.

The `@Test` annotation is provided by TestNG, the `@Artifact` annotation is provided by the JBoss Test Harness and the `AbstractTest` is part of the JBoss Test Harness. There is a one-to-one mapping between a TestNG test class and an artifact. The packaging type is defined by the `@org.jboss.testharness.impl.packaging.Packaging` annotation on the test class, defaulting to a WAR if not specified.

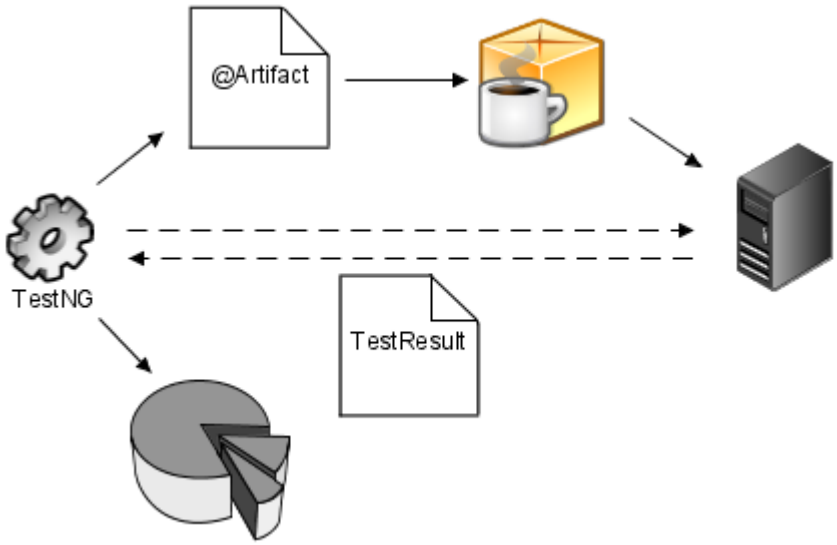
Prior to executing the tests for a given class, the JBoss Test Harness packages the class as a deployable artifact (EAR or WAR), along with any extra resources specified, and deploys the artifact to the container. The harness provides test execution and result reporting via HTTP communication to a simple Servlet using a thin layer over the TestNG test launcher. The test harness can also catch and enforce expected deployment exceptions. This setup and tear down activity is provided by the super class `org.jboss.testharness.AbstractTest`, which all test classes must extend (directly or indirectly).

If the annotation `@org.jboss.testharness.impl.packaging.IntegrationTest` is not present on the test class, then it means the test class can be executed in standalone mode. In standalone mode, the deployable artifact is assembled on the local classpath and the tests execute in the same JVM as the launcher, just as though it were a regular TestNG test case. The standalone mode is provided for convenience and efficiency, allowing you the speed of mock-based testing and the confidence of an in-container test, using the same test objects and tests.

8.1. Negotiating the execution of an in-container test

The basic procedure of an in-container test is as follows. The JBoss Test Harness produces a deployable artifact from an `@Artifact` test class and any declared dependent classes, descriptors or other resources. Then it deploys the artifact to the container using the `Containers` SPI,

negotiates with the container to execute the test and return the result and, finally, undeploys the artifact. TestNG collects the results of all the tests run in the typical way and produces a report.



The question is, how does the JBoss Test Harness negotiate with the container to execute the test when TestNG is being invoked locally? Technially the mechanism is pluggable, but JBoss Test Harness provides a default implementation that uses HTTP communication that you will likely use. Here's how the default implementation works.

The artifact generator bundles and registers (in the web.xml descriptor) an `HttpServlet`, `org.jboss.testharness.impl.runner.servlet.ServletTestRunner`, that responds to test execution GET requests. TestNG running on the client side delegates to a test launcher (more on that in a moment) which originates these text execution requests to transfer control to the container JVM. The name of the test method to be executed is specified in a request query parameter named `methodName`.

When the test execution request is received, the servlet delegates to an instance of `org.jboss.testharness.impl.runner.TestRunner`, passing it the name of the test method. `TestRunner` reads the name of the test class from the resource META-INF/jboss-test-harness.properties, which is bundled in the artifact by the artifact generator. It then combines the class name and the method name to produce a TestNG test suite and runs the suite (within the context of the container).

TestNG returns the results of the run as an `ITestResult` object. `ServletTestRunner` translates this object into a `org.jboss.testharness.api.TestResult` and passes it back to the test launcher on the client side by encoding the translated object into the response. The object gets encoded as either html or a serialized object, depending on the value of the `outputMode` request parameter that was passed to the servlet. Once the result has been transfered to the client-side TestNG, TestNG wraps up the run of the test as though it had been executed in the same JVM.

There's one piece missing. How does TestNG on the client side know to submit a request to the `ServletTestRunner` servlet to get TestNG to execute the test in the container JVM? That's the role of the test launcher.

The test launcher is the API that allows test suite to launch the test in a pluggable fashion. `AbstractTest` implements `IHookable`, a TestNG interface which allows the execution of the test method to be intercepted. Using that mechanism, `AbstractTest` delegates execution of the test method (a method annotated with `@Test` in an `@Artifact` class) to an implementation of `org.jboss.testharness.api.TestLauncher` if the tests are being executed in-container. As you might anticipate, the implementation is specified using a property with the same name as the interface in a META-INF/jboss-test-launcher.properties resource. The JBoss Test Harness provides a default implementation, `org.jboss.testharness.impl.runner.servlet.ServletTestLauncher`, that hooks into the HTTP communication infrastructure described above. It invokes the `ServletTestRunner` servlet for each method annotated with `@Test` in the `@Artifact` that is not otherwise disabled.

If you wish to implement the runner yourself, you must return a `TestResult` as a result of executing the method in the container. You must also ensure that any exception which occurs during deployment is wrapped as a `org.jboss.testharness.api.DeploymentException`, and that any communication problem is rethrown as an `IOException`. The deployment exception may be transformed by an implementation of the `org.jboss.testharness.api.DeploymentExceptionTransformer` interface, which is specified using the `org.jboss.testharness.container.deploymentExceptionTransformer` property. The default implementation passes on the original exception unchanged.

So in short, JBoss Test Harness takes care of all the interfaces you need to execute tests in-container except for the implementation of the `Containers` SPI. That is, unless you are deploying to one of the containers supported by the JBoss Test Harness.

Configuration

This chapter lays out how to configure the JBoss Test Harness by specifying the API implementation classes, defining the target container connection information, and various other switches. Finally, a detailed account of how the JBoss Test Harness negotiates the execution of the tests in the container is given.

9.1. JBoss Test Harness Properties

The JBoss Test Harness allows the test suite to be launched in a pluggable fashion. In order to execute a test suite, the JBoss Test Harness must be configured by specifying implementations of the test launcher and container APIs.

System properties and/or the resource `META-INF/jboss-test-harness.properties`, a Java properties file, are used to configure the JBoss Test Harness. The bootstrap configuration builder looks to the property `org.jboss.testharness.api.ConfigurationBuilder`, the first property listed in table 3.1, for the fully qualified class name (FQCN) of a concrete configuration builder implementation to get started. This implementation loads the remaining configuration settings and produces a JBoss Test Harness configuration.

For you convenience, the default configuration builder implementation `org.jboss.testharness.impl.PropertiesBasedConfigurationBuilder` is provided, which collates all the JBoss Test Harness configuration settings from system and Java properties. It does so by aggregating the system properties with the properties defined in the META-INF/jboss-test-harness.properties resource in any classpath entry under a single properties map, allowing you to partition the configuration settings as needed.

A complete list of configuration properties for the JBoss Test Harness has been itemized in *Table 9.1, "JBoss Test Harness Configuration Properties"*, accompanied by the default value (if any) and a description for each property.

Table 9.1. JBoss Test Harness Configuration Properties

Property = Default Value	Description
`org.jboss.testharness.api.ConfigurationBuilder= org.jboss.testharness.impl.PropertiesBasedConfigurationBuilder`	The configuration bootstrap class for the JBoss Test Harness.
`org.jboss.testharness.testPackage=`	The top-level Java package containing classes to be tested. Used to determine which artifacts to dump to disk only; not used during running of a suite.

Property = Default Value	Description
`org.jboss.testharness.libraryDirectory=`	Directory containing extra JARs which should be deployed in the artifact (for example in `WEB-INF/lib`.
`org.jboss.testharness.standalone=true`	Tests are run using standalone mode if true or using in-container mode if false.
`org.jboss.testharness.runIntegrationTests=false`	If true, integration tests are run. In-container mode must be activated.
`org.jboss.testharness.spi.Containers=`	The deployment implementation for setting up and tearing down the container and deploying and undeploying in-container tests.
`org.jboss.testharness.host=localhost:8080`	The host and port on which the container is running.
`org.jboss.testharness.connectDelay=5000`	The timeout (ms) when attempting to connect to the container (e.g. via http).
`org.jboss.testharness.api.TestLauncher=`	The in-container test launcher, the built in `org.jboss.testharness.impl.runner.serv` is provided and suitable for any Servlet environment.
`org.jboss.testharness.container.\ deploymentExceptionTransformer=`	Provides an interception feature for deployment exceptions, allowing them to be inspected and altered before reporting to the test harness for validation by the test case.
`org.jboss.testharness.container.forceRestart=false`	Whether the container should be restarted

Property = Default Value	Description
	before the tests are executed.
`org.jboss.testharness.container.extraConfigurationDir=`	A directory containing a `build.properties` or `local.build.properties` files that define additional properties. Can be used to provide runtime specific properties.
`org.jboss.testharness.spi.StandaloneContainers=`	The container implementation for executing standalone tests.
`dumpArtifacts=false`	Whether the test artifacts should be written to disk for inspection.
`org.jboss.testharness.outputDirectory= %java.io.tmpdir %/jsr-299-tck/`	Directory where test artifacts will be written to disk, if `dumpArtifacts` is true.

Executing a Test Suite

This chapter explains how to execute and debug a test suite built using the JBoss Test Harness.

10.1. Building a test suite runner using Maven 2

The test suite runner project is the magic that makes everything come together and allows you to execute the test suite. If you fully understand how the JBoss Test Harness functions, and have a good grasp on Maven 2, then it's not to difficult to understand how the test suite runner project works. Regardless of your background, this guide covers what you need to know to get up and running by studying the test suite runner used to run the Bean Validation TCK against the Bean Validation RI, Hibernate Validator.

The TCK runner for the Hibernate Validator can be found in the `hibernate-validator-tck-runner` directory in the Hibernate Validator checkout. The dependencies of the TCK runner project for Hibernate Validator are listed in *Table 10.1, "Bean Validation TCK Runner Dependencies"*.

Table 10.1. Bean Validation TCK Runner Dependencies

Group ID	Artifact ID	Version
javax.validation	validation-api	1.0.0.GA
org.hibernate	hibernate-validator	4.2.0.Final
org.hibernate.jsr303.tck	jsr303-tck	1.0.8.GA
org.hibernate.javax.persistence	hibernate-jpa-2.0-api	1.0.1.Final
org.sl4j	slf4j-api, slf4j-simple	1.6.1
org.testng	testng (classifier: jdk15)	5.8
org.jboss.jbossas.as7-cdi-tck	jbossas-container	1.0.0.Alpha1

You can find all of these artifacts in the *JBoss Maven repository* [http://repository.jboss.org/maven2].

You should substituate the hibernate-validator artifact from *Table 10.1, "Bean Validation TCK Runner Dependencies"* with your own artifact. You'll also need to replace the jbossas-container artifact if you are not testing your implementation on JBoss AS. The jbossas-container artifact contains implementations of the `Containers` SPI for the JBoss Test Harness for JBoss AS 7.

Note

When running the test suite in the in-container mode, the tests will run against libraries installed into the container.

The TCK is executed using the Maven TestNG plugin. Maven 2 profiles are used to control the properties that are set at the time of the execution. For instance, the `incontainer` profile enables integration tests and disables standalone mode, changing the default settings.

10.2. Dumping the Test Artifacts to Disk

As you have learned, when the test suite is executing using in-container mode, each test class is packaged as a deployable artifact and deployed to the container. The test is then executed within the context of the deployed application. This leaves room for errors in packaging. When investigating a test failure, it's helpful to be able to inspect the artifact after it is generated. The JBoss Test Harness can accommodate this type of inspection by "dumping" the generated artifact to disk.

If you want to write the artifacts to disk, and avoid executing the test suite, you can simply execute the main method of the class `org.jboss.testharness.api.TCK`. For example you could use a Maven profile that is activated when the `dumpArtifacts` command line property is defined:

```
mvn test-compile -DdumpArtifacts
```

The output directory where the artifacts are written is defined by the property `org.jboss.testharness.outputDirectory`.

Once the artifact is written to disk, you have an option of manually deploying it to the container. You can execute the tests in the artfact by requesting the context path of the application in the browser. If you want to execute an individual test method, specify the method name in the `methodName` request parameter (e.g., ?methodName=testMethodName).