TCK Reference Guide for Jakarta Bean Validation

Emmanuel Bernard - Red Hat, Inc., Hardy Ferentschik - Red Hat, Inc., Gunnar Morling - Red Hat, Inc.

Version 2.0.6, 2020-10-06

Table of Contents

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 Jakarta Bean Validation 2.0.

The Jakarta Bean Validation TCK is built atop Arquillian, a portable and configurable automated test suite for authoring unit and integration tests in a Jakarta EE environment.

The Jakarta Bean Validation TCK is provided under the Apache Public License 2.0.

Who Should Use This Guide

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

Before You Read This Guide

The Jakarta Bean Validation TCK is based on the Jakarta Bean Validation specification 2.0. Information about the specification can be found on the Jakarta Bean Validation page.

How This Guide Is Organized

If you are running the Jakarta Bean Validation TCK for the first time, read Introduction completely for the necessary background information about the TCK. Once you have reviewed that material, perform the steps outlined in the remaining chapters.

- Introduction gives an overview of the principles that apply generally to all Technology Compatibility Kits (TCKs), outlines the appeals process and describes the Jakarta Bean Validation TCK architecture and components.
- Appeals Process explains the process to be followed by an implementor should they wish to challenge any test in the TCK.
- Installation explains where to obtain the required software for the Jakarta Bean Validation TCK and how to install it.
- Reports 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 Jakarta Bean Validation specification and in understanding how test cases relate to the specification.
- Running the TCK test suite details the configuration of the test harness and documents how to create a TCK runner for executing the TCK test suite, either in standalone or container mode.

- Running the Signature Test finally documents how to use the SigTest tool for ensuring compatibility of types provided in the package `javax.validation` with the official API signature defined by the specification.

Chapter 1. Introduction

This chapter explains the purpose of a TCK and identifies the foundation elements of the Jakarta 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 (`tck-audit.xml`), 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 (meaning the provided test suite must be run unmodified).

A TCK is entirely implementation agnostic. It should validate assertions by consulting the specification'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 is 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 Jakarta Bean Validation specification goes to great lengths to ensure that programs written for Jakarta EE are compatible and the TCK is rigorous about enforcing the rules the specification lays down.

1.3. About the Jakarta Bean Validation TCK

The Jakarta Bean Validation TCK is designed as a portable, configurable and automated test suite for verifying the compatibility of an implementation of Jakarta Bean Validation. The test suite is built atop TestNG and provides a series of extensions that allow runtime packaging and deployment of Jakarta EE artifacts for in-container testing (Arquillian).

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

The declarative approach allows many of the tests to be executed in a standalone implementation of Jakarta Bean Validation, accounting for a boost 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.

The reason the Jakarta Bean Validation TCK must pass running in a Jakarta EE container is that Jakarta Bean Validation is part of Jakarta EE 8 itself.

1.3.1. TCK Components

The Jakarta Bean Validation TCK includes the following components:

- **The test suite**, which is a collection of TestNG tests, the TestNG test suite descriptor and supplemental resources that configure Jakarta Bean Validation and other software components.
- **The TCK audit** (`tck-audit.xml`) used to list out the assertions identified in the Jakarta Bean Validation specification. It matches the assertions to test cases 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.
- **TCK Container Adapter** provided as a convenience for developers in order to run and debug tests outside of the Jakarta EE container.
- **Setup examples** demonstrating Maven and Ant setups to run the TCK test suite

1.3.2. Passing the Jakarta Bean Validation TCK

In order to pass the Jakarta Bean Validation TCK (which is one requirement for becoming a certified Jakarta Bean Validation provider), you need to:

- Pass the Jakarta Bean Validation signature tests (see Running the Signature Test) asserting the correctness of the Bean Validation API used.
- Run and pass the test suite (see Running the TCK test suite). The test must be run within a Jakarta EE 8 container and pass with an unmodified TestNG suite file.

> The designated reference runtime for compatibility testing of the Jakarta Bean Validation specification is the Jakarta EE 8 reference implementation (RI), aka Eclipse GlassFish 5.1+.

Chapter 2. Appeals Process

While the Jakarta Bean Validation TCK is rigorous about enforcing an implementation's conformance to the Jakarta Bean Validation specification, it's reasonable to assume that an implementor may discover new and/or better ways to validate the assertions. The appeals process is defined by the Jakarta EE Jakarta EE TCK Process 1.0.

2.1. What challenges to the TCK may be submitted?

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

What is generally not challengeable 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 Jakarta Bean Validation EG by sending an e-mail to bean-validation-dev@eclipse.org.

2.2. How these challenges are submitted?

To submit a challenge, a new issue of type Bug should be created against BVTCK in the Hibernate JIRA instance. The appellant should complete the Summary, Component (TCK Appeal), Environment and Description fields only. Any communication regarding the issue should be added in the comments of the 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.

2.3. 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 the Specification Lead, Red Hat Inc., or his/her designate. The appellant can also monitor the process by watching the issue filed against BVTCK.

The current TCK Project Lead is listed on the Bean Validation Project Summary Page on Jakarta EE.

2.4. How accepted challenges to the TCK are managed?

The workflow for TCK challenges is outlined in the Jakarta EE TCK Process 1.0.

Periodically, an updated TCK will be released, containing tests altered due to challenges - no new tests will be added. Implementations are required to pass the updated TCK. This release stream is named 2.0.x, where x will be incremented.

Additionally, new tests will be added to the TCK improving coverage of the specification. We encourage implementations to pass this TCK, however it is not required. This release stream is named 2.y.z where y >= 1.

Chapter 3. 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 Jakarta Bean Validation TCK project via the official Jakarta Bean Validation home page. The Jakarta 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`. The contents should look like:

```
artifacts/
changelog.txt
docs/
lib/
license.txt
setup-examples/
src/
readme.md
```

You can also download the source code from GitHub - https://github.com/eclipse-ee4j/beanvalidation-tck.

The Jakarta Bean Validation reference implementation (RI) project is named Hibernate Validator. You can obtain the Hibernate Validator release used as reference implementation from the Hibernate Validator download page.

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 Jakarta Bean Validation implementation.

3.2. The TCK Environment

The TCK requires the following two Java runtime environments:

- Java 8 (including a JavaFX implementation)
- Jakarta EE 8 or better (e.g. Eclipse GlassFish 5.1+)

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 dedicated folder to hold all of the Jakarta Bean Validation-related artifacts. This guide assumes the folder is called `jakarta-bean-validation`. Extract the `src` folder of the TCK distribution into a sub-folder named `tck` or use the following git commands:

```
git clone git://github.com/eclipse-ee4j/beanvalidation-tck tck
git checkout 2.0.6
```

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 6.2.0.Final
```

The resulting folder structure is shown here:

```
jakarta-bean-validation/
   ri/
   tck/
```

Now lets have a look at one concrete test of the TCK, namely `ConstraintInheritanceTest` (found in `tck/tests/src/main/java/org/hibernate/beanvalidation/tck/tests/constraints/inheritance/ConstraintInheritanceTest.java`):

```
package org.hibernate.beanvalidation.tck.tests.constraints.inheritance;

import static
org.hibernate.beanvalidation.tck.util.ConstraintViolationAssert.assertCorrectC
onstraintTypes;
import static org.testng.Assert.assertEquals;
import static org.testng.Assert.assertTrue;

import java.lang.annotation.Annotation;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.constraints.DecimalMin;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.validation.metadata.BeanDescriptor;
import javax.validation.metadata.ConstraintDescriptor;
import javax.validation.metadata.PropertyDescriptor;

import org.hibernate.beanvalidation.tck.beanvalidation.Sections;
import org.hibernate.beanvalidation.tck.tests.AbstractTCKTest;
import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.shrinkwrap.api.spec.WebArchive;
import org.jboss.test.audit.annotations.SpecAssertion;
```

```java
import org.jboss.test.audit.annotations.SpecVersion;
import org.testng.annotations.Test;

/**
 * @author Hardy Ferentschik
 */
@SpecVersion(spec = "beanvalidation", version = "2.0.0")
public class ConstraintInheritanceTest extends AbstractTCKTest {

    @Deployment
    public static WebArchive createTestArchive() {
        return webArchiveBuilder()
                .withTestClassPackage( ConstraintInheritanceTest.class )
                .build();
    }

    @Test
    @SpecAssertion(section = Sections
.CONSTRAINTDECLARATIONVALIDATIONPROCESS_INHERITANCE, id = "b")
    public void testConstraintsOnSuperClassAreInherited() {
        BeanDescriptor beanDescriptor = getValidator().getConstraintsForClass(
Bar.class );

        String propertyName = "foo";
        assertTrue( beanDescriptor.getConstraintsForProperty( propertyName )
!= null );
        PropertyDescriptor propDescriptor = beanDescriptor
.getConstraintsForProperty( propertyName );

        Annotation constraintAnnotation = propDescriptor
.getConstraintDescriptors()
                .iterator()
                .next().getAnnotation();
        assertTrue( constraintAnnotation.annotationType() == NotNull.class );
    }

    @Test
    @SpecAssertion(section = Sections
.CONSTRAINTDECLARATIONVALIDATIONPROCESS_INHERITANCE, id = "a")
    @SpecAssertion(section = Sections
.CONSTRAINTDECLARATIONVALIDATIONPROCESS_INHERITANCE, id = "b")
    public void testConstraintsOnInterfaceAreInherited() {
        BeanDescriptor beanDescriptor = getValidator().getConstraintsForClass(
Bar.class );

        String propertyName = "fubar";
        assertTrue( beanDescriptor.getConstraintsForProperty( propertyName )
!= null );
        PropertyDescriptor propDescriptor = beanDescriptor
.getConstraintsForProperty( propertyName );

        Annotation constraintAnnotation = propDescriptor
.getConstraintDescriptors()
                .iterator()
                .next().getAnnotation();
        assertTrue( constraintAnnotation.annotationType() == NotNull.class );
    }

    @Test
    @SpecAssertion(section = Sections
.CONSTRAINTDECLARATIONVALIDATIONPROCESS_INHERITANCE, id = "a")
```

```
    @SpecAssertion(section = Sections
.CONSTRAINTDECLARATIONVALIDATIONPROCESS_INHERITANCE, id = "c")
    public void testConstraintsOnInterfaceAndImplementationAddUp() {
        BeanDescriptor beanDescriptor = getValidator().getConstraintsForClass(
Bar.class );

        String propertyName = "name";
        assertTrue( beanDescriptor.getConstraintsForProperty( propertyName )
!= null );
        PropertyDescriptor propDescriptor = beanDescriptor
.getConstraintsForProperty( propertyName );

        List<Class<? extends Annotation>> constraintTypes =
getConstraintTypes( propDescriptor.getConstraintDescriptors() );

        assertEquals( constraintTypes.size(), 2 );
        assertTrue( constraintTypes.contains( DecimalMin.class ) );
        assertTrue( constraintTypes.contains( Size.class ) );
    }

    @Test
    @SpecAssertion(section = Sections
.CONSTRAINTDECLARATIONVALIDATIONPROCESS_INHERITANCE, id = "a")
    @SpecAssertion(section = Sections
.CONSTRAINTDECLARATIONVALIDATIONPROCESS_INHERITANCE, id = "c")
    public void testConstraintsOnSuperAndSubClassAddUp() {
        BeanDescriptor beanDescriptor = getValidator().getConstraintsForClass(
Bar.class );

        String propertyName = "lastName";
        assertTrue( beanDescriptor.getConstraintsForProperty( propertyName )
!= null );
        PropertyDescriptor propDescriptor = beanDescriptor
.getConstraintsForProperty( propertyName );

        List<Class<? extends Annotation>> constraintTypes =
getConstraintTypes( propDescriptor.getConstraintDescriptors() );

        assertEquals( constraintTypes.size(), 2 );
        assertTrue( constraintTypes.contains( DecimalMin.class ) );
        assertTrue( constraintTypes.contains( Size.class ) );
    }

    @Test
    @SpecAssertion(section = Sections
.CONSTRAINTDECLARATIONVALIDATIONPROCESS_VALIDATIONROUTINE, id = "a")
    public void testValidationConsidersConstraintsFromSuperTypes() {
        Set<ConstraintViolation<Bar>> violations = getValidator().validate(
new Bar() );
        assertCorrectConstraintTypes(
                violations,
                DecimalMin.class, DecimalMin.class, ValidBar.class, //Bar
                NotNull.class, Size.class, ValidFoo.class, //Foo
                NotNull.class, Size.class, ValidFubar.class //Fubar
        );
    }

    private List<Class<? extends Annotation>> getConstraintTypes(Iterable
<ConstraintDescriptor<?>> descriptors) {
        List<Class<? extends Annotation>> constraintTypes = new ArrayList
<Class<? extends Annotation>>();
```

```
        for ( ConstraintDescriptor<?> constraintDescriptor : descriptors ) {
            constraintTypes.add( constraintDescriptor.getAnnotation()
.annotationType() );
        }

        return constraintTypes;
    }
}
```

Each test class is treated as an individual artifact (hence the `@Deployment` annotation on the class). In most tests the created artifact is a standard Web application Archive build via `WebArchiveBuilder` which in turn is a helper class of the TCK itself alleviating the creation of of the artifact. All methods annotated with `@Test` are actual tests which are getting run. Last but not least we see the use of the `@SpecAssertion` annotation which creates the link between the tck-audit.xml document and the actual test (see TCK Primer).

Example 1. Running the TCK against the Jakarta Bean Validation RI (Hibernate Validator) and WildFly 10.1

- Install Maven. You can find documentation on how to install Maven 3 on the Maven official website.
- Change to the `ri/tck-runner` 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`.

Chapter 4. Reports

This chapter covers the two types of reports that can be generated from the TCK, an assertion coverage report and the test execution results.

4.1. Jakarta 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 Jakarta Bean Validation 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.

4.1.1. Jakarta Bean Validation TCK Assertions

The Jakarta Bean Validation TCK developers have analyzed the Jakarta Bean Validation 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: "Every constraint annotation must define a message element of type String"

The assertions are listed in the XML file `tck-audit.xml` in the Jakarta 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="constraintsdefinitionimplementation-constraintdefinition-
properties-message" title="message" level="4">
    ...
    <!-- 3.1.1.1 -
CONSTRAINTSDEFINITIONIMPLEMENTATION_CONSTRAINTDEFINITION_PROPERTIES_MESSAGE
-->
    <assertion id="a">
        <text>Every constraint annotation must define a message element of
type String.</text>
    </assertion>
    ...
</section>
```

The strategy of the Jakarta 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`) is correlated with an assertion using the `@org.jboss.test.audit.annotations.SpecAssertion` annotation as follows:

```
@Test(expectedExceptions = ConstraintDefinitionException.class)
...
@SpecAssertion(section = Sections
.CONSTRAINTSDEFINITIONIMPLEMENTATION_CONSTRAINTDEFINITION_PROPERTIES_MESSAGE,
id = "a")
...
public void testConstraintDefinitionWithoutMessageParameter() {
    getValidator().validate( new DummyEntityNoMessage() );
    fail( "The used constraint does not define a message parameter. The
validation should have failed." );
}
```

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.

4.1.2. 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. You can find the source code for this processor in the GitHub repository https://github.com/jboss/jboss-test-audit The report is written to the file `target/coverage-report/coverage-beanvalidation.html`. The report itself 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
- **Unimplemented** - a test exists, but is unimplemented
- **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.

4.2. The TestNG Report

As you by now know, the Jakarta Bean Validation TCK test suite is really just a TestNG test suite. That means an execution of the Jakarta 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.

4.2.1. Maven, Surefire and TestNG

When the Jakarta Bean Validation TCK test suite is executed during the Maven 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: 976, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 35.288 sec -
in TestSuite

Results :

Tests run: 976, Failures: 0, Errors: 0, Skipped: 0
```

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.

Chapter 5. Running the TCK test suite

This chapter lays out how to run and configure the TCK harness against a given Jakarta Bean Validation provider in a given Jakarta EE container. If you have not by now made yourself familiar with the Arquillian documentation, this is a good time to do it. It will give you a deeper understanding of the different parts described in the following sections.

5.1. Setup examples

The TCK distribution comes with a directory `setup-examples` which contains two example projects for running the TCK. If you followed the instructions in Installation you find the directory under `jakarta-bean-validation/tck/setup-examples`. Both setups are using Hibernate Validator as Jakarta Bean Validation Provider and Eclipse GlassFish 5.1+ as Jakarta EE constainer. However, one is using Maven as build tool to run the TCK, the other Ant. Depending which of the examples you want to use, you need to install the corresponding build tool.

Each example comes with a `readme.md` containing the prerequisites for using this setup, how to run the TCK against Hibernate Validator and Eclipse GlassFish. The readme in `setup-examples` itself contains information about what needs to be changed to use a different Jakarta Bean Validation provider and Jakarta EE container.

The following chapters contain some more information about the general structure of the TCK which will give you a deeper understanding above the simple readme files.

5.2. Configuring TestNG to execute the TCK

The Jakarta Bean Validation test harness is built atop TestNG, and it is 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.

The `tck-tests.xml` artifact provided in the TCK distribution must be run by TestNG (described by the TestNG documentation as "with a `testng.xml` file") unmodified for an implementation to pass the TCK. For testing purposes it is of course ok to modify the file (see also the TestNG documentation)

```
<suite name="Jakarta-Bean-Validation-TCK" verbose="1">
    <test name="Jakarta-Bean-Validation-TCK">

        <method-selectors>
            <method-selector>
                <selector-class name=
"org.hibernate.beanvalidation.tck.util.IntegrationTestsMethodSelector"/>
            </method-selector>
        </method-selectors>

        <packages>
            <package name="org.hibernate.beanvalidation.tck.tests"/>
        </packages>
    </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.

5.3. Selecting the `ValidationProvider`

The most important configuration you have make in order to run the Jakarta Bean Validation TCK is to specify your `ValidationProvider` you want to run your tests against. To do so you need to set the Java system property `validation.provider` to the fully specified class name of your `ValidationProvider`. In Maven this is done via the `systemProperties` configuration option of the maven-surefire-plugin, whereas `sysproperty` is used in an Ant testng task. This system property will be picked up by `org.hibernate.beanvalidation.tck.util.TestUtil` which will instantiate the `Validator` under test. This means the test harness does not rely on the service provider mechanism to instantiate the Jakarta Bean Validation provider under test, partly because this selection mechanism is under test as well.

5.4. Selecting the `DeployableContainer`

After setting the `ValidationProvider` you have to make a choice on the right `DeployableContainer`. Arquillian picks which container it is going to use to deploy the test archive and negotiate test execution using Java's service provider mechanism. Concretely Arquillian is looking for an implementation of the `DeployableContainer` SPI on the classpath. The setup examples use a remote Eclipse GlassFish container adapter, which means that Arquillian tries to deploy the test artifacts onto a specified remote Eclipse GlassFish instance, run the tests remotely and report the results back to the current JVM. The installation directory of the remote container is specified via the `container.home` property in the example build files. To make it easier to develop, debug or test the TCK, an in JVM adapter is provided as part of the distribution (`beanvalidation-standalone-container-adapter-2.0.6.jar`). Using this adapter the tests are not executed in a remote Jakarta EE container, but in the current JVM. This allows for easy and fast debugging. Some tests, however, are only runnable in a Jakarta EE container and will fail in this in JVM execution mode. By setting the property `excludeIntegrationTests` to true these tests can be excluded.

The adapter is also available as Maven artifact under the GAV `org.hibernate.beanvalidation.tck:beanvalidation-standalone-container-adapter:2.0.6`. You can refer to `pom.xml` in the tck-runner module of Hibernate Validator (in the directory `jakarta-bean-validation/ri/tck-runner`, if you followed the instruction in Installation) to see how it is used.

5.5. arquillian.xml

The next piece in the configuration puzzle is `arquillian.xml`. This xml file needs to be in the root of the classpath and is used to pass additional options to the selected container. Let's look at an example:

```
<arquillian xmlns="http://jboss.org/schema/arquillian" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://jboss.org/schema/arquillian
        http://jboss.org/schema/arquillian/arquillian_1_0.xsd">
    <defaultProtocol type="Servlet 3.0"/>

    <engine>
        <property name="deploymentExportPath">target/artifacts</property>
    </engine>

    <container qualifier="incontainer" default="true">
        <configuration>
            <property name="glassFishHome">@CONTAINER.HOME@</property>
            <property name="adminHost">localhost</property>
            <property name="adminPort">4848</property>
            <property name="debug">true</property>
        </configuration>
    </container>

</arquillian>
```

The most important container configuration option is the protocol type which determines how Arquillian communicates with the selected container. The most popular types are `Servlet 3.0` and `Local`. The former is used when connecting to a remote container whereas the latter is used for the in JVM mode.

Another interesting property is `deploymentExportPath` which is optional and instructs Arquillian to dump the test artifacts to the specified directory on disk. Inspection of the deployed artifacts can be very useful when debugging test failures.

The Jakarta Bean Validation specification mandates a support of JavaFX if JavaFX is available in the classpath.

While JavaFX is included in the Oracle JDK and some other JDKs include OpenJFX, it might not be included in all JDKs.

Having JavaFX available in a container environment might also not be straightforward.

For these reasons, the JavaFX tests are disabled when running the TCK with the default options.

It is highly recommended to run the TCK in at least one configuration that allows to test the support of JavaFX.

Using this configuration, the JavaFX tests can be enabled by passing the `-DincludeJavaFXTests=true` option to the TCK.

Chapter 6. Running the Signature Test

One of the requirements of an implementation passing the TCK is for it to pass the Jakarta Bean Validation signature test. This section describes how to run it against your implementation as a part of a Maven build.

6.1. Executing the signature check

The signature file bundled inside this TCK is created using the SigTest Maven plugin. The same plugin can be used to run a signature test to check for any incompatibilities. Let's take a look how it can be done as a part of a Maven build. Note that there must be no dependency declared for this project besides the API artifact you wish to test.

Before running an actual test you need to obtain the signature file first. It is packaged inside the *beanvalidation-tck-tests* artifact, so we can get it using the `unpack` goal of the `maven-dependency-plugin` as shown below:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>3.0.0</version>
    <executions>
        <execution>
            <id>copy-tck-bv-api-signature-file</id>
            <phase>generate-test-sources</phase>
            <goals>
                <goal>unpack</goal>
            </goals>
            <configuration>
                <artifactItems>
                    <artifactItem>
                        <groupId>org.hibernate.beanvalidation.tck</groupId>
                        <artifactId>beanvalidation-tck-tests</artifactId>
                        <version>${beanvalidation-tck-tests.version}</version>
                        <type>jar</type>
                        <overWrite>false</overWrite>
                    </artifactItem>
                </artifactItems>
                <!-- We just need the signature file and nothing else -->
                <includes>**/*.sig</includes>
                <outputDirectory>${project.build.directory}/api-
signature</outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

To actually run a signature test, the `check` goal of the `sigtest-maven-plugin` can be used. The plugin configuration above puts the signature file to the *api-signature* subdirectory of your project's build directory. Having the file there, it can be referenced via the `sigfile` parameter of the `sigtest-maven-plugin` plugin like this:

```
<plugin>
    <groupId>org.netbeans.tools</groupId>
    <artifactId>sigtest-maven-plugin</artifactId>
    <version>1.0</version>
    <executions>
        <execution>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
    <configuration>

<packages>javax.validation,javax.validation.bootstrap,javax.validation.constra
ints,

javax.validation.constraintvalidation,javax.validation.executable,javax.valida
tion.groups,

javax.validation.metadata,javax.validation.spi,javax.validation.valueextractio
n
        </packages>
        <sigfile>${project.build.directory}/api-signature/validation-api-
java8.sig</sigfile>
    </configuration>
</plugin>
```

6.2. Forcing a signature test failure

If you would like to verify that the signature test is running correctly, make a copy of the signature file somewhere on your local file system and modify it. For example let us change the `value()` of `javax.validation.constraints.Max` to `val()` which should make SigTest fail.

After modifying the signature file, update the `sigfile` parameter of the `sigtest-maven-plugin` to point to the modified file:

```
<sigfile>${path_to_folder_containing_your_modified_signature_file}/validation-
api-java8.sig</sigfile>
```

If all is done correctly, while running `mvn sigtest:check` on your project, you should see an error similar to next:

```
[INFO] SignatureTest report
Base version: 2.0.0-SNAPSHOT
Tested version: 2.0.0-SNAPSHOT
Check mode: bin [throws removed]
Constant checking: on


Class javax.validation.constraints.Max
  "E2.7 - Removing member from annotation type" : method public abstract long
javax.validation.constraints.Max.val()
```