Technology Compatibility Kit Reference Guide for Jakarta Contexts and Dependency Injection

Version 4.0

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 the Jakarta Contexts and Dependency Injection.

The CDI TCK is built atop TestNG framework and Arquillian platform. The CDI TCK uses Arquillian version *1.7.0.Alpha10* to execute the test suite.

The CDI TCK is provided under Apache Public License 2.0.

Who Should Use This Book

This guide is for implementors of the Jakarta Context and Dependency Injection 4.0 technology to assist in running the test suite that verifies the compatibility of their implementation.

Before You Read This Book

Before reading this guide, you should familiarize yourself with the Jakarta EE programming model, specifically the Jakarta Enterprise Beans (EJB) 4.0 and the Jakarta Contexts and Dependency Injection 4.0 specifications. A good resource for the Jakarta EE programming model is the Jakarta EE web site.

The CDI TCK is based on the Jakarta Context and Dependency Injection technology specification. Information about the specification, including links to the specification documents, can be found on the CDI page.

Before running the tests in the CDI TCK, read and become familiar with the Arquillian testing platform. A good starting point could be a series of Arquillian Guides.

How This Book Is Organized

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

- Introduction (CDI TCK) gives an overview of the principles that apply generally to all Technology Compatibility Kits (TCKs), outlines the appeals process and describes the CDI 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.
- Appeals Process explains the process to be followed by an implementor, who wish to challenge any test in the TCK.
- Installation explains where to obtain the required software for the CDI TCK and how to install it. It covers both the primary TCK components as well as tools useful for troubleshooting tests.
- 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.

- 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 CDI specification and in understanding how testcases relate to the specification.
- 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.
- Running Tests in Eclipse shows how to run individual tests in Eclipse and advises the best way to setup your Eclipse workspace for running the tests.
- Debugging Tests in Eclipse builds on Running Tests in Eclipse by detailing how to debug individual tests in Eclipse.

Getting Acquainted with the TCK

The CDI TCK must be used to ensure that your implementation conforms to the CDI 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 CDI 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.

Chapter 1. Introduction (CDI TCK)

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

1.1. TCK Primer

A TCK, or Technology Compatibility Kit, is one of the three required pieces for any specification (the other two being the specification document and a compatible 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. Ideally, it should validate assertions by consulting the specification's public API. However, when the information returned by the public API is not low-level enough to validate the assertion, the implementation must be consulted directly. In this case, the TCK provides an independent API as part of a porting package that enables this transparency. The porting package must be implemented for each CDI implementation. CDI TCK Components introduces the porting package and The Porting Package covers the requirements for implementing it.

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 compatible implementation. The compatible 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 Jakarta 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 CDI 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. Compatibility Requirements

The compatibility requirements for Jakarta Contexts and Dependency Injection Version 3.0 consist of meeting the requirements set forth by the rules and associated definitions contained in this section.

1.3.1. Definitions

These definitions are for use only with these compatibility requirements and are not intended for any other purpose.

Table 1. Definitions

Term	Definition
API Definition Product	A Product for which the only Java class files contained in the product are those corresponding to the application programming interfaces defined by the Specifications, and which is intended only as a means for formally specifying the application programming interfaces defined by the Specifications.

Term	Definition
Computational Resource	A piece of hardware or software that may vary in quantity, existence, or version, which may be required to exist in a minimum quantity and/or at a specific or minimum revision level so as to satisfy the requirements of the Test Suite. Examples of computational resources that may vary in quantity are RAM and file descriptors. Examples of computational resources that may vary in existence (that is, may or may not exist) are graphics cards and device drivers. Examples of computational resources that may vary in version are operating systems and device drivers.
Conformance Tests	All tests in the Test Suite for an indicated Technology Under Test, as distributed by the Maintenance Lead, excluding those tests on the Exclude List for the Technology Under Test.
Documented	Made technically accessible and made known to users, typically by means such as marketing materials, product documentation, usage messages, or developer support programs.
Edition	A Version of the Java Platform. Editions include Java Platform Standard Edition and Jakarta Platform Enterprise Edition.
Exclude List	The most current list of tests, distributed by the Maintenance Lead or TCK Lead, that are not required to be passed to certify conformance. The Maintenance Lead or TCK Lead may add to the Exclude List for that Test Suite as needed at any time, in which case the updated Exclude List supplants any previous Exclude Lists for that Test Suite.
Libraries	The class libraries for the Technology Under Test. The Libraries for Jakarta Contexts and Dependency Injection Version 4.0 are listed in Libraries for Jakarta Contexts and Dependency Injection Version 4.0.

Term	Definition
Location Resource	A location of classes or native libraries that are components of the test tools or tests, such that these classes or libraries may be required to exist in a certain location in order to satisfy the requirements of the test suite. For example, classes may be required to exist in directories named in a CLASSPATH variable, or native libraries may be required to exist in directories named in a PATH variable.
Product	A licensee product in which the Technology Under Test is implemented or incorporated, and that is subject to compatibility testing.
Product Configuration	A specific setting or instantiation of an Operating Mode. For example, a Product supporting an Operating Mode that permits user selection of an external encryption package may have a Product Configuration that links the Product to that encryption package.
Compatible Implementation (CI)	The prototype or "proof of concept" implementation of a Specification.
Resource	A Computational Resource, a Location Resource, or a Security Resource.
Rules	These definitions and rules in this Compatibility Requirements section of this User’s Guide.
Security Resource	A security privilege or policy necessary for the proper execution of the Test Suite. For example, the user executing the Test Suite will need the privilege to access the files and network resources necessary for use of the Product.
Specifications	The documents produced through the Jakarta EE Specification Process that define a particular Version of a Technology. The Specifications for the Technology Under Test are referenced later in this chapter.
TCK Lead	Person responsible for maintaining TCK for the Technology. TCK Lead is representative of Red Hat Inc.
Technology	Specifications and a compatible implementation produced through the Jakarta EE Specification Process.

Term	Definition
Technology Under Test	Specifications and the compatible implementation for Jakarta Contexts and Dependency Injection Version 3.0.
Test Suite	The requirements, tests, and testing tools distributed by the Maintenance Lead or TCK Lead as applicable to a given Version of the Technology.
Version	A release of the Technology, as produced through the Jakarta EE Specification Process.

1.3.2. Rules for Jakarta Contexts and Dependency Injection Version 4.0 Products

The following rules apply for each version of an operating system, software component, and hardware platform Documented as supporting the Product:

CDI-1 The Product must be able to satisfy all applicable compatibility requirements, including passing all Conformance Tests, in every Product Configuration and in every combination of Product Configurations, except only as specifically exempted by these Rules.

For example, if a Product provides distinct Operating Modes to optimize performance, then that Product must satisfy all applicable compatibility requirements for a Product in each Product Configuration, and combination of Product Configurations, of those Operating Modes.

CDI-1.1 If an Operating Mode controls a Resource necessary for the basic execution of the Test Suite, testing may always use a Product Configuration of that Operating Mode providing that Resource, even if other Product Configurations do not provide that Resource. Notwithstanding such exceptions, each Product must have at least one set of Product Configurations of such Operating Modes that is able to pass all the Conformance Tests.

For example, a Product with an Operating Mode that controls a security policy which has one or more Product Configurations that cause Conformance Tests to fail may be tested using a Product Configuration that allows all Conformance Tests to pass.

CDI-1.2 A Product Configuration of an Operating Mode that causes the Product to report only version, usage, or diagnostic information is exempted from these compatibility rules.

CDI-1.3 A Product may contain an Operating Mode that selects the Edition with which it is compatible. The Product must meet the compatibility requirements for the corresponding Edition for all Product Configurations of this Operating Mode. This Operating Mode must affect no smaller unit of execution than an entire Application.

CDI-1.4 An API Definition Product is exempt from all functional testing requirements defined here, except the signature tests.

CDI-2 Some Conformance Tests may have properties that may be changed. Properties that can be changed are identified in the configuration interview. Properties that can be changed are specified

in TCK Properties. Apart from changing such properties and other allowed modifications described in this User's Guide (if any), no source or binary code for a Conformance Test may be altered in any way without prior written permission.

CDI-3 The testing tools supplied as part of the Test Suite or as updated by the Maintenance Lead or TCK Lead must be used to certify compliance.

CDI-4 The Exclude List associated with the Test Suite cannot be modified.

CDI-5 The Maintenance Lead or TCK Lead can define exceptions to these Rules. Such exceptions would be made available to and apply to all licensees.

CDI-6 All hardware and software component additions, deletions, and modifications to a Documented supporting hardware/software platform, that are not part of the Product but required for the Product to satisfy the compatibility requirements, must be Documented and available to users of the Product. For example, if a patch to a particular version of a supporting operating system is required for the Product to pass the Conformance Tests, that patch must be Documented and available to users of the Product.

CDI-7 The Product must contain the full set of public and protected classes and interfaces for all the Libraries. Those classes and interfaces must contain exactly the set of public and protected methods, constructors, and fields defined by the Specifications for those Libraries. No subsetting, supersetting, or modifications of the public and protected API of the Libraries are allowed except only as specifically exempted by these Rules.

CDI-8 Except for tests specifically required by this TCK to be recompiled (if any), the binary Conformance Tests supplied as part of the Test Suite or as updated by the Maintenance Lead or TCK Lead must be used to certify compliance.

CDI-9 The functional programmatic behavior of any binary class or interface must be that defined by the Specifications.

1.4. About the CDI TCK

The CDI TCK is designed as a portable, configurable and automated test suite for verifying the compatibility of an implementation of the Jakarta CDI specification. The test suite is built atop TestNG framework and Arquillian platform.

Each test class in the suite acts as a deployable unit. The deployable units, or artifacts, can be either a WAR or an EAR.

The test archives are built with ShrinkWrap, a Java API for creating archives. ShrinkWrap is a part of the Arqullian platform ecosystem.

1.4.1. CDI TCK Specifications and Requirements

This section lists the applicable requirements and specifications for the CDI TCK.

- **Specification requirements** - Software requirements for a CDI implementation are itemized in

section 1.2, "Relationship to other specifications" in the CDI specification, with details provided throughout the specification. Generally, the CDI specification targets the Jakarta EE 10 platform and will be aligned with its specifications.

- **Jakarta Contexts and Dependency Injection 4.0 API** - The Java API defined in the CDI specification and provided by the compatible implementation.
- **Testing platform** - The CDI TCK requires version 1.7.0.Alpha10 of the Arquillian (http://arquillian.org). The TCK test suite is based on TestNG 7.4 (http://testng.org). .
- **Porting Package** - An implementation of SPIs that are required for the test suite to run the in-container tests and at times extend the CDI 4.0 API to provide extra information to the TCK.
- **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.
- **Compatible implementation** - A compatible implementation runtime for compatibility testing of the CDI specification is the Jakarta Platform Enterprise Edition 10 compatible implementation.
- **Jarkarta Dependency Injection (DI)** - CDI builds on DI, and as such CDI implementations must additionally pass the Jakarta Dependency Injection TCK.
- **The Jakarta Contexts and Dependency Injection Language Model TCK** - The CDI Language Model TCK included in the CDI TCK distribution must be run and passed by every CDI cmpatible implementation.
- **Jakarta Interceptors** - CDI is an implementation of the Jakarta Interceptors specification. Jakarta Interceptors has no TCK of its own, so the CDI TCK includes an extensive set of tests that validate the Jakarta Interceptors concepts.

The TCK distribution includes weld/porting-package-lib/weld-inject-tck-runner-X.Y.Z-Q-tests.jar which contains two classes showing how the Weld compatible implementation passes the CDI TCK. The source for these classes is available from hhttps://github.com/weld/core/tree/5.0.0.Alpha2/inject-tck-runner/src/test/java/org/jboss/weld/atinject/tck

1.4.2. CDI TCK Components

The CDI TCK includes the following components:

- **Arquillian 1.7.0.Alpha10**
- **TestNG 7.4.0**
- **Porting Package SPIs** - Extensions to the CDI SPIs to allow testing of a container.
- **The test suite**, which is a collection of TestNG tests, the TestNG test suite descriptor and supplemental resources that configure CDI and other software components.
- **The TCK audit** is used to list out the assertions identified in the CDI 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; at least 95% of assertions are tested. 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 CDI TCK has been tested on following platforms:

- WildFly X using Oracle Java SE 11 on Red Hat Enterprise Linux 8.5

CDI supports Jakarta EE 10, Jakarta EE 10 Web Profile, Embeddable Jakarta Enterprise Beans 4.0. The TCK will execute on any of these runtimes, but is only part of the CTS for Jakarta EE 10 and Jakarta EE 10 Web Profile.

1.5. Libraries for Jakarta Contexts and Dependency Injection Version 4.0

The following is the list of packages that constitute the required class libraries for Jakarta Contexts and Dependency Injection Version 4.0:

- jakarta.decorator
- jakarta.enterprise.context
- jakarta.enterprise.context.control
- jakarta.enterprise.context.spi
- jakarta.enterprise.event
- jakarta.enterprise.inject
- jakarta.enterprise.inject.build.compatible.spi;
- jakarta.enterprise.inject.literal
- jakarta.enterprise.inject.se
- jakarta.enterprise.inject.spi
- jakarta.enterprise.inject.spi.configurator
- jakarta.enterprise.lang.model
- jakarta.enterprise.lang.model.declarations
- jakarta.enterprise.lang.model.types
- jakarta.enterprise.util

Chapter 2. Appeals Process

While the CDI TCK is rigorous about enforcing an implementation's conformance to the Jakarta CDI 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 (e.g., test class, @Test method), test case configuration (e.g., beans.xml), test beans, 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 by the Maintenance Lead or by sending an e-mail to link:mailto:cdi-dev@eclipse.org

2.2. How these challenges are submitted?

To submit a challenge, a new issue should be created in the CDI specification project using the label challenge. Any communication regarding the issue should be pursed in the comments of the filed issue for accurate record.

2.3. How and by whom challenges are addressed?

The challenges will be addressed in a timely fashion by the TCK Lead, as designated by Specification Lead, Red Hat Inc. or his/her designate. The appellant can also monitor the process by following the issue report filed in the CDI TCK project issues.

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

2.4. How accepted challenges to the TCK are managed?

The worflow for TCK challenges is outlined in 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 4.0.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 3.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 CDI TCK project from the download page on the CDI specification website. The release stream for Jakarta CDI is named *4.0.x*. The CDI TCK is distributed as a ZIP file, which contains the TCK artifacts (the test suite binary and source, porting package API binary and source, the test suite configuration file, the audit source and report) in /artifacts and documentation in /doc. The TCK library dependencies are not part of the distribution and can be downloaded on demand (see readme.txt file in /lib).

You can also download the current source code from GitHub repository.

Executing the TCK requires a Jakarta EE 10 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 Jakarta EE implementation.

A Jakarta Contexts and Dependency Injection for compatible implementation project is named Weld. The release stream for Jakarta CDI 4.0 is named *{revmajor}.x*. You can obtain the latest release from the download page on the Weld website.

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

Naturally, to execute Java programs, you must have a Java SE runtime environment. The TCK requires Java SE 11 or better, which you can obtain from the Java Software website.

3.2. The TCK Environment

The TCK requires the following two Java runtime environments:

- Java SE 11 or better
- Jakarta EE 10 or better (e.g., WildFly 27.x or GlassFish V7)

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

The rest of the TCK software can simply be extracted. Extract the TCK distribution to create a *core-tck-4.x.y* directory. The resulting folder structure is shown here:

This layout is assumed through all descriptions in this reference guide.

```
core-tck-4.x.y/
   artifacts/
   doc/
   lib/
   weld/
   LICENSE
   README.adoc
```

Each test class is treated as an individual artifact. All test methods (i.e., methods annotated with @Test) in the test class are run in the application, meaning bean discovery occurs exactly once per artifact and the same BeanManager is used by each test method in the class.

Running the TCK against Weld and WildFly

- First, you should download WildFly 27.x from the WildFly project page.
- Set the JBOSS_HOME environment variable to the location of the WildFly software.

The CDI TCK distribution includes a TCK runner that executes the TCK using Weld as the CDI implementation and WildFly as the Jakarta EE runtime. To run the TCK:

- You need to install Maven. You can find documentation on how to install Maven in the Maven: The Definitive Guide book published by Sonatype.
- Next, integrate the Weld jars into WildFly:

```
cd core-tck-4.x.y/weld/jboss-as
mvn -Pupdate-jboss-as package
```

- Next, integrate the TCK ext jar into WildFly:

```
cd core-tck-4.x.y/weld/jboss-as
mvn -Dtck package
```

- Next, instruct Maven to run the TCK:

```
cd core-tck-4.x.y/weld/jboss-tck-runner
mvn test -Dincontainer
```

- Use cdi.tck-4-0.versionn system property to specify particular TCK version:

```
mvn test -Dincontainer -Dcdi.tck-4-0.version=4.0.5
```

- 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.

3.3. Eclipse Plugins

Eclipse, or any other IDE, is not required to execute or pass the TCK. However an implementor may wish to execute tests in an IDE to aid debugging the tests. This section introduces two essential Eclipse plugins, TestNG and Maven, and points you to resources explaining how to install them.

3.3.1. TestNG Plugin

The TCK test suite is built on the TestNG. Therefore, having the TestNG plugin installed in Eclipse is essential. Instructions for using the TestNG update site to add the TestNG plugin to Eclipse are provided on the TestNG download page. You can find a tutorial that explains how to use the TestNG

plugin on the TestNG Eclipse page.

3.3.2. Maven Plugin (m2e)

Another useful plugin is m2e. The TCK project uses Maven. Therefore, to work with TCK in Eclipse, you may wish to have native support for Maven projects, which the m2e plugin provides. Instructions for using the m2e update site to add the m2e plugin to Eclipse are provided on the m2e home page.

You can alternatively use the Eclipse plugin for Maven to generate native Eclipse projects from Maven projects.

If you have Maven installed, you have everything you need. Just execute the following command from any Maven project to produce the Eclipse project files.

```
mvn eclipse:eclipse
```

Again, the Eclipse plugins are not required to execute the TCK, but can be very helpful when validating an implementation against the TCK test suite and especially when using the modules from the project.

Chapter 4. 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 TCK test suite, putting these settings into practice.

4.1. TCK Properties

System properties and/or the resource META-INF/cdi-tck.properties, a Java properties file, are used to configure the TCK.

You should set the following required properties:

Table 2. Required TCK Configuration Properties

Property = Example Value	**Description**
org.jboss.cdi.tck.libraryDirectory=/path/to/extra/libraries	The directory containing extra JARs to be placed in the test archive library directory such as the porting package implementation.
org.jboss.cdi.tck.cdiLiteMode=true	Enable the CDI Lite mode. When enabled, none of the org.jboss.cdi.tck.test* properties below related to Jakarta EE tests are required.
org.jboss.cdi.tck.testDataSource=java:jboss/datasources/ExampleDS	A few TCK tests work with Jakarta Persistence services and require a data source to be provided. This property defines JNDI name of such resource. Required for the tests within the *persistence* test group.
org.jboss.cdi.tck.testJmsConnectionFactory=java:/ConnectionFactory	The JNDI name of the JMS test ConnectionFactory. Required for the tests within the *jms* test group.
org.jboss.cdi.tck.testJmsQueue=java:/queue/test	The JNDI name of the JMS test Queue. Required for the tests within the *jms* test group.
org.jboss.cdi.tck.testJmsTopic=java:/topic/test	The JNDI name of the JMS test Topic. Required for the tests within the *jms* test group.

Table 3. Optional TCK Configuration Properties

Property = Example Value	**Description**
org.jboss.cdi.tck.testTimeoutFactor=200	Tests use this percentage value to adjust the final timeout (e.g. when waiting for some async processing) so that it's possible to configure timeouts according to the testing runtime performance and throughput. The value must be an integer greater than zero. The default value is 100% - i.e. timeouts will remain the same.

4.2. Arquillian settings

The Arquillian testing platform will look for configuration settings in a file named *arquillian.xml* in the root of your classpath. If it exists it will be auto loaded, else default values will be used. This file is not a requirement however it's very useful for container configuration. See an example configuration for JBoss TCK runner:

```
weld/jboss-tck-runner/src/test/wildfly8/arquillian.xml
```

4.3. The Porting Package

The CDI TCK relies on an implementation of the porting package to function. There are times when the tests need to tap directly into the CDI implementation to manipulate behavior or verify results. The porting package is Java package named "org.jboss.cdi.tck.spi" and includes a set of SPIs that provide the TCK with this level of access without tying the tests to a given implementation.

The SPI classes in the CDI TCK are as follows:

- org.jboss.cdi.tck.spi.Beans
- org.jboss.cdi.tck.spi.Contexts
- org.jboss.cdi.tck.spi.EL

Please consult the JavaDoc for these interfaces for the implementation requirements.

4.4. Using the CDI TCK with the Jakarta EE Core Profile

You can configure the CDI TCK to just run tests related to the CDI Lite specification appropriate for the Jakarta EE Core Profile by excluding TestNG groups *javaee-full*, *se*, e.g. via a maven-surefire-plugin configuration like:

```
<plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <excludedGroups>cdi-full,se</excludedGroups>
        <dependenciesToScan>
          <dependency>jakarta.enterprise:cdi-tck-core-impl</dependency>
        </dependenciesToScan>
      </configuration>
 </plugin>
```

4.5. Using the CDI TCK with the Jakarta EE Web Profile

You can configure the CDI TCK to just run tests appropriate to the Jakarta EE Web Profile by excluding TestNG group *javaee-full,se* e.g. via a maven-surefire-plugin configuration like:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <excludedGroups>javaee-full,se</excludedGroups>
        <dependenciesToScan>
          <dependency>jakarta.enterprise:cdi-tck-core-impl</dependency>
          <dependency>jakarta.enterprise:cdi-tck-web-impl</dependency>
        </dependenciesToScan>
      </configuration>
 </plugin>
```

4.6. Using the CDI TCK with the Jakarta EE Full Platform

You can configure the CDI TCK to just run tests appropriate to the Jakarta EE Full Platform by excluding TestNG group *se* e.g. via a maven-surefire-plugin configuration like:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <excludedGroups>se</excludedGroups>
        <dependenciesToScan>
          <dependency>jakarta.enterprise:cdi-tck-core-impl</dependency>
          <dependency>jakarta.enterprise:cdi-tck-web-impl</dependency>
        </dependenciesToScan>
      </configuration>
 </plugin>
```

4.7. Using the CDI TCK with the Java SE

You can configure the CDI TCK to just run tests appropriate to the Java SE runtime by including the TestNG group *se* and *arquillian*, e.g. via a maven-surefire-plugin configuration like:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <groups>se,arquillian</groups>
        <dependenciesToScan>
          <dependency>jakarta.enterprise:cdi-tck-core-impl</dependency>
        </dependenciesToScan>
      </configuration>
 </plugin>
```

The *arquillian* group specification is needed due to an issue open issue with how the Arquillian TestNG integration behaves: ARQ-395

4.8. Configuring TestNG to execute the TCK

The CDI TCK is built atop Arquillian and 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.

Certain TestNG configuration file must be run by TestNG (described by the TestNG documentation as "with a testng.xml file") unmodified for an implementation to pass the TCK. The TCK distribution contains the configuration file accurate at the date of the release - artifacts/cdi-tck-impl-suite.xml. However this artifact may not be up to date due to unresolved challenges or pending releases. Therefore a canonical configuration file exists. This file is located in the CDI TCK source code repository at *${CORRESPONDING_BRANCH_ROOT}/impl/src/main/resources/tck-tests.xml*.

The canonical configuration file for CDI TCK is located at https://github.com/eclipse-ee4j/cdi-tck/blob/master/impl/src/main/resources/tck-tests.xml.

This file also allows tests to be excluded from a run:

```
<suite name="CDI TCK" verbose="0" configfailurepolicy="continue">
   <test name="CDI TCK">
      ...
      <classes>
         <class name=
"org.jboss.cdi.tck.tests.context.application.ApplicationContextTest">
            <methods>
               <exclude name="testApplicationScopeActiveDuringServiceMethod"/>
            </methods>
         </class>
      </classes>
      ...
   </test>
</suite>
```

Additionally there is available canonical configuration file at https://github.com/eclipse-ee4j/cdi-tck/blob/master/impl/src/main/resources/tck-tests-previous.xml. Please note that this exclude list serves only for the previous micro version of TCK release! This means that if the latest version of TCK is e.g. 4.0.1 then this exclude list is valid only for the version 4.0.0 and invalid for any other version!

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.

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

4.10. Configuring your application server to execute the TCK

The TCK makes use of the Java 1.4 keyword assert; you must ensure that the JVM used to run the application server is started with assertions enabled. See Programming With Assertions for more information on how to enable assertions.

Tests within the *jms* test group require some basic Java Message Service configuration. A connection factory, a queue destination for PTP messaging domain and a topic destination for pub/sub messaging domain must be available via JNDI lookup. The corresponding JNDI names are specified with configuration properties - see TCK Properties.

Tests within the *persistence* test group require basic data source configuration. The data source has to be valid and JTA-based. The JNDI name of the DataSource is specified with configuration property - see TCK Properties.

Tests within the *installedLib* test group require the CDI TCK cdi-tck-ext-lib artifact to be installed as a library (see also Jakarta EE 10 specification, section EE.10.2.2 "Installed Libraries").

Tests within the *systemProperties* test group require the following system properties to be set:

Name	**Value**
cdiTckExcludeDummy	true

Tests within the *security* test group require the following mapping of roles to principals:

Principal	**Group**
student	student
alarm	alarm, student
printer	printer

Tests within *SE* test groups require execution in a separate JVM instance with isolated classpath including all required dependencies.

Chapter 5. 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 CDI specification.

5.1. CDI TCK Coverage Metrics

The CDI TCK coverage has been measured as follows:

- **Assertion Breadth Coverage**

The CDI TCK provides at least 95% coverage of identified assertions with test cases.

- **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 CDI TCK covers 100% of all API public methods using the Java CTT Sig Test tool.

5.2. CDI TCK Coverage Report

A specification can be distilled into a collection of assertions that define the behavior of the software. This section introduces the CDI TCK coverage report, which documents the relationship between the assertions that have been identified in the Jakarta CDI 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. CDK TCK Assertions

The CDI TCK developers have analyzed the Jakarta CDI specification document and identified the assertions that are present in each chapter. Here's an example of one such assertion found in section 2.3.3:

> Any bean may declare multiple qualifier types.

The assertions are listed in the XML file impl/src/main/resources/tck-audit.xml in the CDI TCK distribution. Each assertion is identified by the section identifier 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="declaring_bean_qualifiers" title="Declaring the qualifiers of a
bean">
          ...
          <assertion id="d">
             <text>Any bean may declare multiple qualifier types.</type>
          </assertion>
          ...
            </section>
```

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

```
@Test
@SpecAssertion(section = DECLARING_BEAN_QUALIFIERS, id = "d")
public void testMultipleQualifiers()
{
  Bean<?> model = getBeans(Cod.class, new ChunkyBinding(true), new WhitefishBinding()
).iterator().next();
  assert model.getBindings().size() == 3;
}
```

Section identifiers are not used directly. Instead automatically generated constants are applied.

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. The report is only generated when using Java 6 or above, as it requires the annotation processor.

```
mvn clean install
```

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** - Lists 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

The CDI TCK test suite is really just a TestNG test suite. That means an execution of the CDI TCK test suite produces the same reports as TestNG does. This section will go over those reports and show you where to find each of them.

Maven, Surefire and TestNG

When the CDI 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
[XmlMethodSelector]
CLASSNAME:org.jboss.testharness.impl.testng.DisableIntegrationTestsMethodSelector
[XmlMethodSelector] SETTING PRIORITY:0
[XmlMethodSelector]
CLASSNAME:org.jboss.testharness.impl.testng.ExcludeIncontainerUnderInvestigationMethod
Selector
[XmlMethodSelector] SETTING PRIORITY:0
Tests run: 441, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 22.816 sec

Results :

Tests run: 441, 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 CDI TCK requires.

If the Maven reporting plugin that complements 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.

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, shown below, is written to the file index.html. It produces

the same information as the generic Surefire report.

Test results

Suite	Passed	Failed	Skipped	testng.xml
Total	*441*	*1*	*0*	
JSR-299 TCK	441	1	0	Link

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.

JSR-299 TCK

Tests passed/Failed/Skipped:	441/1/0
Started on:	Wed Jul 29 12:53:39 EDT 2009
Total time:	12 seconds (12169 ms)
Included groups:	
Excluded groups:	broken rewrite stub deployment underInvestigation ri-broken

(Hover the method name to see the test class name)

FAILED TESTS		
Test method	**Time (seconds)**	**Exception**
testDecoratorNotResolved	0	java.lang.AssertionError at org.jboss.jsr299.tck.tests.lookup.typesafe.resolution at org.jboss.testharness.AbstractTest.run(AbstractTest.j at org.apache.maven.surefire.testng.TestNGExecutor.run(T at org.apache.maven.surefire.testng.TestNGXmlTestSuite.e at org.apache.maven.surefire.Surefire.run(Surefire.java: at org.apache.maven.surefire.booter.SurefireBooter.runSu at org.apache.maven.surefire.booter.SurefireBooter.main(... Removed 29 stack frames Click to show all stack frames

Test method
testAbstractApiType
testAbstractClassDeclaredInJavaNotDiscovered
testAllBindingTypesSpecifiedForResolutionMustAppearOnBean
testAmbiguousDependency

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. A partial view of the emailable report is shown below.

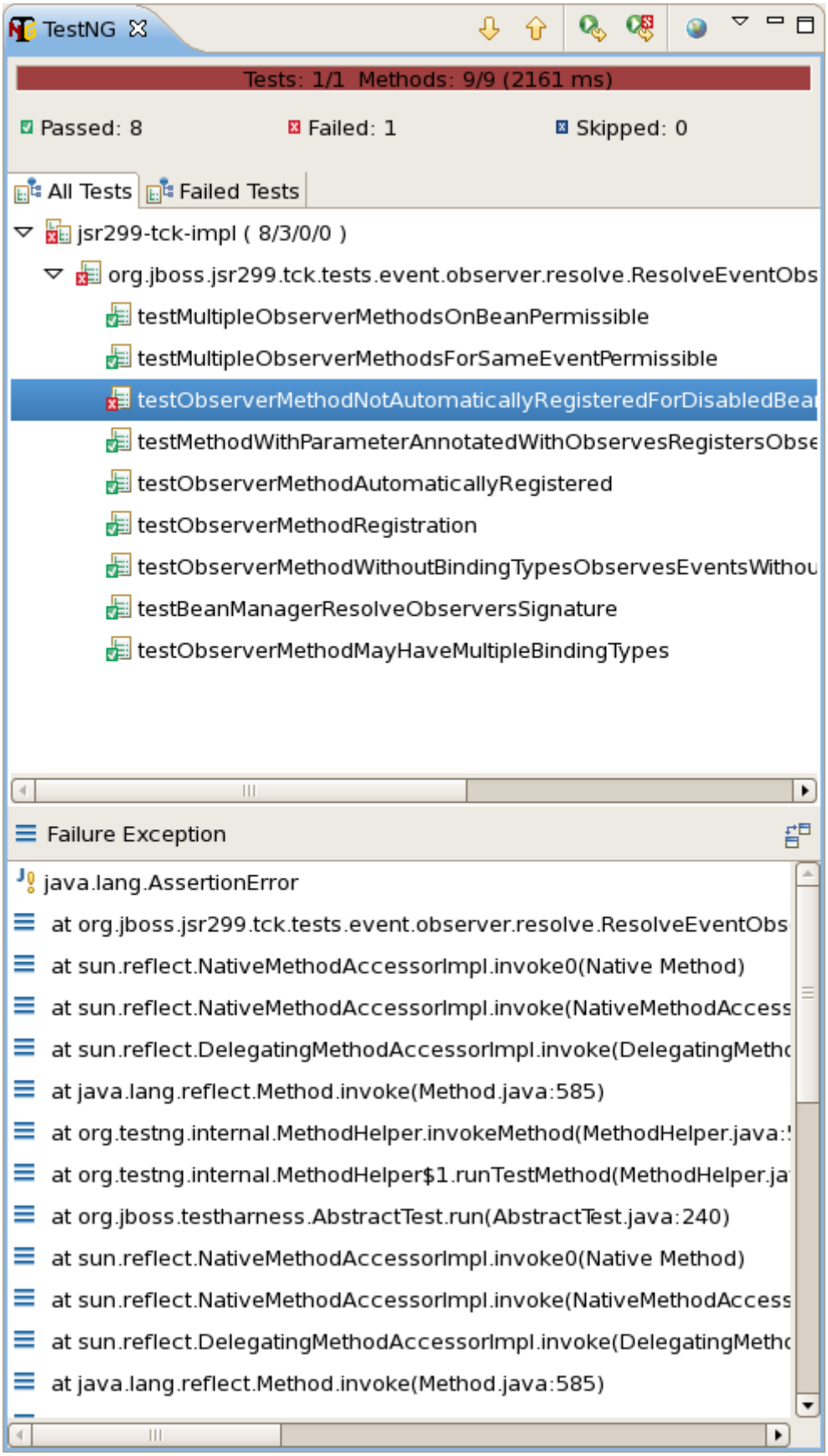
Test	Methods Passed	Scenarios Passed	# skipped	# failed	Total Time	Included Groups	Excluded Groups
JSR-299 TCK	441	441	0	1	12.2 seconds		broken rewrite stub deployment underInvestigation ri-broken

Class
org.jboss.jsr299.tck.tests.lookup.typesafe.resolution.decorator.DecoratorNotResolvedTest
org.jboss.jsr299.tck.tests.context.ContextTest
org.jboss.jsr299.tck.tests.context.DestroyedInstanceReturnedByGetTest
org.jboss.jsr299.tck.tests.context.GetFromContextualTest
org.jboss.jsr299.tck.tests.context.GetOnInactiveContextTest
org.jboss.jsr299.tck.tests.context.GetWithNoCreationalContextTest
org.jboss.jsr299.tck.tests.context.NormalContextTest

Now that you have seen two ways to get test results from the Maven test execution, let's switch over to the IDE, specifically Eclipse, and see how it presents TestNG test results.

Test Results in the TestNG Plugin View

After running a test in Eclipse, the test results are displayed in the TestNG plugin view, as shown below.



The view offers two lists. The first is a list of all methods (tests) in the class flagged as either passed or failed. The second is a list of methods (tests) in the class that failed. If there is a test failure, you can click on the method name to get the stacktrace leading up to the failure to display in the lower frame.

You can also find the raw output of the TestNG execution in the IDE console view. In that view, you

can click on a test in the stacktrace to open it in the editor pane.

One of the nice features of TestNG is that it can keep track of which tests failed and offer to run only those tests again. You can also rerun the entire class. Buttons are available for both functions at the top of the view.

Executing and Debugging Tests

In this part you learn how to execute the CDI TCK on the Weld compatible implementation. First, you are walked through the steps necessary to execute the test suite on Weld. Then you discover how to modify the TCK runner to execute the test suite on your own implementation. Finally, you learn how to debug tests from the test suite in Eclipse.

Chapter 6. Running the Signature Test

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

6.1. Obtaining the sigtest plugin

The sigtest plugin is availble from Maven Central using a dependency like:

```
...
    <plugin>
    <groupId>org.netbeans.tools</groupId>
    <artifactId>sigtest-maven-plugin</artifactId>
    <version>1.5</version>
    </plugin>
```

The source for the sigtest plugin can be found here: https://github.com/jtulach/netbeans-apitest

6.2. Running the signature test

To run the signature test, use a pom file like that found in https://github.com/eclipse-ee4j/cdi-tck/blob/master/impl/src/main/resources/sigtest-pom.xml and shown here:

```
<?xml version="1.0"?>
<!-- Sample maven pom to verify signatures -->
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <!-- For access to staging repos, add -Pstaging -->
  <parent>
    <groupId>org.eclipse.ee4j</groupId>
    <artifactId>project</artifactId>
    <version>1.0.6</version>
  </parent>

  <groupId>jakarta.enterprise</groupId>
  <artifactId>cdi-tck-sigtest</artifactId>
  <version>4.0</version>
  <name>CDI TCK Signature Tests</name>
  <description>CDI TCK Signature test validation of CDI dependent API
jars</description>
  <properties>
    <!-- Set the api jar artifact versions here -->
    <annotation.api.version>2.1.0</annotation.api.version>
    <atinject.api.version>2.0.1</atinject.api.version>
```

```
    <interceptor.api.version>2.1.0</interceptor.api.version>
    <el.api.version>5.0.0</el.api.version>
    <cdi.api.version>4.0.1</cdi.api.version>
  </properties>

  <!-- Set the api jar artifact dependencies here -->
  <dependencies>
    <dependency>
      <groupId>jakarta.annotation</groupId>
      <artifactId>jakarta.annotation-api</artifactId>
      <version>${annotation.api.version}</version>
    </dependency>
    <dependency>
      <groupId>jakarta.el</groupId>
      <artifactId>jakarta.el-api</artifactId>
      <version>${el.api.version}</version>
    </dependency>
    <dependency>
      <groupId>jakarta.interceptor</groupId>
      <artifactId>jakarta.interceptor-api</artifactId>
      <version>${interceptor.api.version}</version>
    </dependency>
    <dependency>
      <groupId>jakarta.inject</groupId>
      <artifactId>jakarta.inject-api</artifactId>
      <version>${atinject.api.version}</version>
    </dependency>
    <dependency>
      <groupId>jakarta.enterprise</groupId>
      <artifactId>jakarta.enterprise.lang-model</artifactId>
      <version>${cdi.api.version}</version>
    </dependency>
    <dependency>
      <groupId>jakarta.enterprise</groupId>
      <artifactId>jakarta.enterprise.cdi-api</artifactId>
      <version>${cdi.api.version}</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>unpack-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>unpack-dependencies</goal>
            </goals>
```

```
            <configuration>
              <stripVersion>true</stripVersion>
              <overWriteReleases>true</overWriteReleases>
              <outputDirectory>target/classes</outputDirectory>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.netbeans.tools</groupId>
        <artifactId>sigtest-maven-plugin</artifactId>
        <version>1.5</version>
        <executions>
          <execution>
            <id>sigtest</id>
            <phase>verify</phase>
            <goals>
              <goal>check</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <sigfile>cdi-api-jdk11.sig</sigfile>
          <packages>
jakarta.decorator,jakarta.enterprise,jakarta.interceptor</packages>
          <classes>target/classes</classes>
          <report>cdi-sig-report.txt</report>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Your version should specify the dependencies on these jars as used in your compatible implementation.

```
Scotts-iMacPro:resources starksm$ mvn -f sigtest-pom.xml verify
[INFO] Scanning for projects...
[INFO]
[INFO] -----------------< jakarta.enterprise:cdi-tck-sigtest >-----------------
[INFO] Building CDI TCK Signature Tests 4.0
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
...
[INFO] --- sigtest-maven-plugin:1.5:check (sigtest) @ cdi-tck-sigtest ---
[INFO] Packages: jakarta.decorator,jakarta.enterprise
[INFO] SignatureTest report
Base version: 4.0.0-SNAPSHOT
Tested version: 4.0
Check mode: bin [throws removed]
Constant checking: on

Warning: The return type java.lang.reflect.Member can't be resolved
Warning: The return type java.lang.reflect.Member can't be resolved
Warning: The return type java.lang.reflect.Member can't be resolved


[INFO] /Users/starksm/Dev/JBoss/Jakarta/cdi-tck/impl/src/main/resources/cdi-sig-
report.txt: 0 failures in /Users/starksm/Dev/JBoss/Jakarta/cdi-
tck/impl/src/main/resources/cdi-api-jdk11.sig
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.941 s
[INFO] Finished at: 2021-12-13T11:38:06-06:00
[INFO] ------------------------------------------------------------------------
```

You can ignore the following warnings: "The return type java.lang.reflect.Member can't be resolved"

The important thing is that the mvn version shows "BUILD SUCCESS".

Another example that just specifies a compatible implementation test as the dependency to validate the API artifact signatures from the transitive dependencies is pom file like that found in https://github.com/eclipse-ee4j/cdi-tck/blob/master/impl/src/main/resources/sigtest-weld-pom.xml and shown here:

```
<?xml version="1.0"?>
<!-- Sample maven pom to verify signatures using only the weld-core-imp artifiact and
its dependencies -->
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <!-- For access to staging repos, add -Pstaging -->
```

```
  <parent>
    <groupId>org.eclipse.ee4j</groupId>
    <artifactId>project</artifactId>
    <version>1.0.6</version>
  </parent>

  <groupId>jakarta.enterprise</groupId>
  <artifactId>cdi-tck-weld-sigtest</artifactId>
  <version>4.0</version>
  <name>CDI TCK Signature Tests</name>
  <description>CDI TCK Signature test validation of CDI dependent API
jars</description>
  <properties>
    <!-- Set the Weld version to test -->
    <weld.version>5.0.0.CR2</weld.version>
  </properties>

  <!-- Set the api jar artifact dependencies here -->
  <dependencies>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-core-impl</artifactId>
      <version>${weld.version}</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>unpack-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>unpack-dependencies</goal>
            </goals>
            <configuration>
              <outputDirectory>target/classes</outputDirectory>
              <overWriteReleases>false</overWriteReleases>
              <overWriteSnapshots>false</overWriteSnapshots>
              <overWriteIfNewer>true</overWriteIfNewer>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.netbeans.tools</groupId>
        <artifactId>sigtest-maven-plugin</artifactId>
        <version>1.5</version>
```

```
        <executions>
          <execution>
            <id>sigtest</id>
            <phase>verify</phase>
            <goals>
              <goal>check</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <sigfile>cdi-api-jdk11.sig</sigfile>
          <packages>
jakarta.decorator,jakarta.enterprise,jakarta.interceptor</packages>
          <classes>target/classes</classes>
          <report>cdi-sig-report.txt</report>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

6.3. CDI Lite Signature Tests

CDI Lite requires the same signature tests as Full. Even though CDI Lite does not require some of the Jakarta Interceptors behaviors, we did not want to restrict what CDI Lite implementations might provide in the way of interceptors, for example, an implementation that supports both Lite and Full. An implementation of CDI Lite can simply depend on the Jakarta Interceptors API artifact to meet the signature test requirements.

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 cdi-api-jdk11.sig

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

```
CLSS public jakarta.enterprise.context.BusyConversationException
cons public BusyConversationException()
cons public BusyConversationException(java.lang.String)
cons public BusyConversationException(java.lang.String,java.lang.Throwable)
cons public BusyConversationException(java.lang.Throwable)
supr jakarta.enterprise.context.ContextException
hfds serialVersionUID
```

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

```
CLSS public jakarta.enterprise.context.BusyConversationException
cons public BusyConversationException(java.lang.Integer)
cons public BusyConversationException(java.lang.String)
cons public BusyConversationException(java.lang.String,java.lang.Throwable)
cons public BusyConversationException(java.lang.Throwable)
supr jakarta.enterprise.context.ContextException
hfds serialVersionUID
```

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

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

jakarta.enterprise.context.BusyConversationException:         constructor public
jakarta.enterprise.context.BusyConversationException.BusyConversationException(java.la
ng.Integer)

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

jakarta.enterprise.context.BusyConversationException:         constructor public
jakarta.enterprise.context.BusyConversationException.BusyConversationException()


STATUS:Failed.2 errors
```

Chapter 7. Executing the Test Suite

This chapter explains how to run the TCK on Weld as well as your own implementation. The CDI TCK uses the Maven Surefire plugin and the Arquillian test platform to execute the test suite. Learning to execute the test suite from Maven is prerequisite knowledge for running the tests in an IDE, such as Eclipse.

7.1. The Test Suite Runner

The test suite is executed by the Maven Surefire plugin during the test phase of the Maven life cycle. The execution happens within a TCK runner project (as opposed to the TCK project itself). Weld includes a TCK runner project that executes the CDI TCK on Weld running inside WildFly 27.x. To execute the CDI TCK on your own CDI implementation, you could modify the TCK runner project included with Weld to use your CDI implementation.

7.2. Running the Tests In Standalone Mode

To execute the TCK test suite against Weld, first switch to the jboss-tck-runner directory in the extracted TCK distribution:

```
cd core-tck-4.x.y/weld/jboss-tck-runner
```

These instructions assume you have extracted the Jakarta CDI TCK software according to the recommendation given in The TCK Environment.

Then execute the Maven life cycle through the test phase:

```
mvn test
```

Without any command-line flags, the test suite is run in standalone mode (activating weld-embedded Maven profile), which means that any test within the *integration, javaee-full* and SE TestNG group is excluded. This mode uses the *Weld EE Embedded Arquillian container adapter* to invoke the test within a mock Jakarta 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 - Core and EE parts

To execute tests within Core and EE parts of the specification you need to use in-container mode with the JBoss TCK runner, you first have to setup WildFly as described in the Running the TCK against Weld and WildFly callout.

Then, execute the TCK runner with Maven as follows:

```
mvn test -Dincontainer
```

The presence of the incontainer property activates an incontainer Maven profile. This time, all the tests except the tests within SE TestNG group are executed.

In order to run tests appropriate to the Jakarta EE Web Profile execute:

```
mvn test -Dincontainer=webprofile
```

To specify particular TCK version:

```
mvn test -Dincontainer -Dcdi.tck-4-0.version=4.0.4
```

In order to run the TCK Test Suite in the container an Arquillian container adapter is required. See also Arqullian reference guide.

The Arquillian will also start and stop the application server automatically (provided a managed Arqullian container adapter is used).

Since Arquillian 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 TCK utilizes servlet-based protocol (communication over HTTP).

Some implementations of Jakarta Faces use modern Javascript that is incompatible with the TCK's default test tooling. In order to test these implementations, specify the property -Drun.selenium=true. An installation of Chrome is required.

7.4. Running the Tests In the Container - SE part

To execute full TCK testsuite you have to run tests within SE group as well. SE tests make use of Arquillian container SE. This way the tests are executed in a separate JVM instance with isolated and configurable classpath. The Arquillian container does not start CDI container in any way - this is still done directly in the tests using CDI SE bootstrap API and `jakarta.enterprise.inject.se.SeContainerInitializer`. In order to run SE TCK tests in Weld, you need to execute "weld-se" Maven profile from the JBoss TCK runner POM file as follows:

```
---
mvn test -Dincontainer=se
---
```

The profile needs to provide RI dependencies as well as Arquillian settings (`arquillian.xml`). These two need to be stored into a directory so that Arquillian container can then be instructed to pick them up. In Weld, this orchestration is done using `maven-dependency-plugin` along with `maven-surefire-plugin`.

7.5. Dumping the Test Archives

As you have learned, when the test suite is executing using in-container mode, each test class is packaged as a deployable archive 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 archive after it's generated. The TCK (or Arquillian respectively) can accommodate this type of inspection by "dumping" the generated archive to disk.

The feature just described is activated in the Arquillian configuration file (Arquillian settings). In order to export the test archive you'll have to add the deploymentExportPath property element inside engine element and assign a relative or absolute directory where the test archive should be exported, e.g.:

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

Arquillian will export the archive to that location for any test you run.

To enable the export for just a single test, use the VM argument arquillian.deploymentExportPath:

```
-Darquillian.deploymentExportPath=target/deployments/
```

Chapter 8. Executing the Lang Model Test Suite

The Language Model TCK does not depend on any test framework or test runner. Assertions are made using Java `assert`. The tests are executed in an implementation-defined manner.

To run the Language Model TCK, implementations must call the `org.jboss.cdi.lang.model.tck.LangModelVerifier#verify()` static method and pass it a `ClassInfo` object for the `LangModelVerifier` class. The way how this method is called and how the `ClassInfo` object is obtained are not specified, so that implementations are free to use whatever works best for them. Two conditions must be satisfied:

- assertions are enabled;
- the language model under test is configured to only return runtime-retained annotations.

If the `verify` method returns successfully, the TCK passed. If it throws an exception, the TCK failed.

To aid with debugging, the `verify` method prints a message to the JVM standard output in case of a success.

8.1. Recommendation

For CDI implementations, it is easiest to run the Language Model TCK using a build compatible extension. For example:

```
public class LangModelVerifierExtension implements BuildCompatibleExtension {
    @Enhancement(types = LangModelVerifier.class, withAnnotations = Annotation.class)
    public void run(ClassInfo clazz) {
        LangModelVerifier.verify(clazz);
    }
}
```

8.2. Example Weld Test Suite Runner

To execute the TCK test suite against Weld, first switch to the lang-model-tck-runner directory in the extracted TCK distribution:

```
cd core-tck-4.x.y/weld/lang-model-tck-runner
```

These instructions assume you have extracted the Jakarta CDI TCK software according to the recommendation given in The TCK Environment.

Then, execute the TCK runner with Maven as follows:

```
mvn test
```

Chapter 9. Running Tests in Eclipse

This chapter explains how to run individual tests using the Eclipse TestNG plugin. It covers running non-integration tests in standalone mode and integration tests (as well as non-integration tests) in in-container mode. You should be able to use the lessons learned here to debug tests in an alternate IDE as well.

9.1. Leveraging Eclipse's plugin ecosystem

Using an existing test harness (TestNG) allows the tests to be executed and debugged in an Integrated Development Environment (IDE) using available plugins. Using an IDE is also the easiest way to execute a test class in isolation.

The TCK can be executed in any IDE for which there is a TestNG plugin available. Running a test from the CDI TCK test suite using the Eclipse TestNG plugin is almost as simple as running any other TestNG test. You can also use the plugin to debug a test, which is described in the next chapter.

Before running a test from the TCK test suite in Eclipse, you must have the Eclipse TestNG plugin and either the m2e plugin or an Eclipse project generated using the Maven 2 Eclipse plugin (maven-eclipse-plugin). Refer to Eclipse Plugins for more information on these plugins.

In order to run the TCK tests in Eclipse you must have CDI TCK and Weld JBoss TCK runner projects imported. Get the source from GitHub repositories https://github.com/eclipse-ee4j/cdi-tck and https://github.com/weld/core.

With the m2e plugin installed, Eclipse should recognize the CDI TCK projects as valid Eclipse projects (or any Weld project for that matter). Import them into the Eclipse workspace at this time. You should also import the Weld projects if you want to debug into that code, which is covered later.

If you choose to use the Maven 2 Eclipse plugin (maven-eclipse-plugin), you should execute the plugin in both the tck and weld projects:

```
cd tck
mvn clean eclipse:clean eclipse:eclipse -DdownloadSources
-DdownloadJavadocs
cd ../weld
mvn clean eclipse:clean eclipse:eclipse -DdownloadSources
-DdownloadJavadocs
```

9.2. Readying the Eclipse workspace

When setting up your Eclipse workspace, we recommended creating three workings sets:

Weld - Groups the CDI API and the Weld projects CDI TCK - Groups the CDI TCK API and the test suite projects Weld JBoss TCK Runner - Groups the porting package implementation and TCK

runner projects The dependencies between the projects will either be established automatically by the m2e plugin, based on the dependency information in the pom.xml files, or as generated by the mvn eclipse:eclipse command.

Your workspace should appear as follows:

```
Weld
  cdi-api
  weld-core
  ...
CDI TCK
  cdi-tck-api
  cdi-tck-impl
  cdi-tck-parent
Weld JBoss TCK Runner
  weld-jboss-runner-tck
  weld-porting-package-tck
```

The tests in the TCK test suite are located in the cdi-tck-impl project. You'll be working within this project in Eclipse when you are developing tests. However, as you learned earlier, there are no references to a CDI implementation in the TCK. So how can you execute an individual test in Eclipse? The secret is that you need to establish a link in Eclipse (not in Maven) between the cdi-tck-impl project and your TCK runner project, which in this case is weld-jboss-runner-tck (the project in the jboss-tck-runner directory).

Here are the steps to establish the link:

1. Right click on the cdi-tck-impl project
2. Select Build Path > Configure Build Path…
3. Click on the Projects tab
4. Click the Add… button on the right
5. Check the TCK runner project (e.g., weld-jboss-runner-tck)
6. Click the OK button on the Required Project Selection dialog window
7. Click the OK button on the Java Build Path window

Of course, the weld-jboss-runner-tck also depends on the cdi-tck-impl at runtime (so it can actually find the tests to execute). But m2e plugin doesn't distinguish between build-time and runtime dependencies. As a result, we've created a circular dependency between the projects. In all likelihood, Eclipse will struggle (if not fail) to compile one or more projects. How can we break this cycle?

As it turns out, the TCK runner doesn't need to access the tests to build. It only needs its classes, configurations and other dependencies at runtime (when the TestNG plugin executes). Therefore, we can disable Resolve dependencies from workspace projects setting on weld-jboss-runner-tck project:

1. Right click on the weld-jboss-runner-tck project

2. Select Maven
3. Uncheck Resolve dependencies from workspace projects option
4. Click the OK button on the Properties window

As you have learned, the TCK determines how to behave based on the values of system properties or properties defined in META-INF/cdi-tck.properties classpath resources. In order to run the tests, you need to add a properties file to the classpath or define corresponding system properties.

The CDI TCK project conveniently provides the properties file src/test/resources/META-INF/cdi-tck.properties that contains all of the necessary properties for testing in Eclipse. You have to tune the org.jboss.cdi.tck.libraryDirectory and org.jboss.cdi.tck.testDataSource properties to point to the relative location of the related projects and specify the name of test datasource. The properties should be defined as follows:

org.jboss.cdi.tck.libraryDirectory - the path to the target/dependency/lib directory in the TCK runner project org.jboss.cdi.tck.testDataSource - the JNDI name of the test datasource, e.g. WildFly 22:

```
org.jboss.cdi.tck.testDataSource=java:jboss/datasources/ExampleDS
```

You are now ready to execute an individual test class (or artifact). Let's start with a test artifact capable of running in standalone mode.

9.3. Running a test in standalone mode

Use weld-embedded Maven profile (active by default) in order to run a test in standalone mode.

If using m2e Eclipse plugin, you can activate/deactivate the profile in Maven section of project properties.

Note that all TestNG tests that are not included in integration and javaee-full test groups are considered to be standalone artifacts.

Select a test class containing standalone tests and open it in the Eclipse editor. Now right click in the editor view and select Run As > TestNG Test. The TestNG view should pop out and you should see all the tests in that artifact pass (if all goes well).

If the TCK complains that there is a property missing, close all the projects, open them again, and rebuild. The m2e plugin can be finicky getting everything built correctly the first time.

So far you have executed a test in standalone mode. That's not sufficient to pass the TCK. The test must be executed using in-container mode.

Let's see what has to be done to execute an integration test. This will result in the artifact being deployed to the container, which is WildFly if you are using the JBoss TCK runner.

9.4. Running integration tests

In order to run a test in the container you must explicitly specify following active Maven profiles in JBoss TCK runner Eclipse project properties: incontainer,!weld-embedded.

Note that all TestNG tests that are included in integration and javaee-full test groups are considered to be integration tests and must be run in in-container mode. javaee-full TestNG test group contains tests that require full Jakarta EE platform (EAR packaging, JAX-WS, EJB timers, etc.).

Select an integration test (a class that extends org.jboss.cdi.tck.AbstractTest and open it in your Eclipse editor. Right click in the editor view and select Run As > TestNG Test.

You have now mastered running the CDI TCK against Weld using both Maven and within Eclipse. Now you're likely interested in how to debug a test so that you can efficiently investigate test failures.

Chapter 10. Debugging Tests in Eclipse

This chapter explains how to debug standalone and integration tests from the TCK test suite in Eclipse. You should be able to use the lessons learned here to debug tests in an alternate IDE as well.

10.1. Debugging a standalone test

There is almost no difference in how you debug a standalone test from how you run it. With the test class open in the Eclipse editor, simply right click in the editor view and select Debug As > TestNG Test. Eclipse will stop at any breakpoints you set just like it would with any other local debug process.

If you plan to step into a class in the Weld implementation (or any other dependent library), you must ensure that the source is properly associated with the library. Below are the steps to follow to associate the source of Weld with the TestNG debug configuration:

1. Select the Run > Debug Configurations… menu from the main menubar
2. Select the name of the test class in the TestNG category
3. Select the Source tab
4. Click the Add… button on the right
5. Select Java Project
6. Check the project that contains the class you want to debug (e.g., weld-core)
7. Click OK on the Project Selection window
8. Click Close on the Debug Configurations window

You'll have to complete those steps for any test class you are debugging, though you only have to do it once (the debug configuration hangs around indefinitely).

Again, running a test in standalone mode isn't enough to pass the TCK and cannot be used to run or debug an integration test. Let's look at how to debug a test running in the context of the container.

10.2. Debugging an integration test

In order to debug an integration test, or any test run using in-container mode, the test must be configured to run in-container, as described in Running integration tests, and you must attach the IDE debugger to the container. That puts the debugger on both sides of the fence, so to speak.

Since setting up a test to run in-container has already been covered, we'll look at how to attach the IDE debugger to the container, and then move on launching the test in debug mode.

10.2.1. Attaching the IDE debugger to the container

There are two ways to attach the IDE debugger to the container. You can either start the container in debug mode from within the IDE, or you can attach the debugger over a socket connection to a standalone container running with JPDA enabled.

The Eclipse Server Tools, a subproject of the Eclipse Web Tools Project (WTP), has support for launching most major application servers, including WildFly 22. However, if you are using WildFly, you should consider using JBoss Tools instead, which offers tighter integration with JBoss technologies. See either the Server Tools documentation or the JBoss Tools documentation for how to setup a container and start it in debug mode.

See this blog entry to learn how to start WildFly with JPDA enabled and how to get the Eclipse debugger to connect to the remote process.

10.2.2. Launching the test in the debugger

Once Eclipse is debugging the container, you can set a breakpoint in the test and debug it just like a standalone test. Let's give it a try.

Open a test in the Eclipse editor, right click in the editor view, and select Debug As > TestNG Test (this works for the container started in debug mode from within the IDE) or run the TestNG test and debug Remote Java Application (remote debug configuration) in the same time (when attaching the debugger over a socket connection to a container). This time when the IDE hits the breakpoint, it halts the JVM thread of the container rather than the thread that launched the test.

Remember that if you need to debug into dependent libraries, the source code for those libraries will need to be registered with the TestNG debug configuration as described in the first section in this chapter.