JSR107: Java™ Caching API
TCK Users Guide V1.1

Greg Luck
4 November 2017

Comments to: jsr107@googlegroups.com

OR SERVICES, LOSS OF USE, DATA OR PROFITS, OR BUSINESS INTERRUPTION), HOWEVER CAUSED AND UNDER ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, NEGLIGENCE, STRICT LIABILITY, OR TORT, ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. SUN FURTHER DISCLAIMS ANY AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE OR TRADE. NO ADVICE OR INFORMATION, WHETHER ORAL OR WRITTEN, OBTAINED FROM SUN OR ELSEWHERE SHALL CREATE ANY WARRANTY NOT EXPRESSLY STATED IN THIS AGREEMENT. Licensee assumes all risk and liability with respect to use of this document and agrees to indemnify Oracle America, Inc. against any loss, damages, or liability that result from licensee's exercise of these license rights. Oracle’s rights of indemnification shall survive the termination of this license. Oracle may terminate this license at any time if licensee exceeds the scope of the license.

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 JSR107: Java™ Caching API specification.

The TCK is built using JUnit and run from Maven.

1. Who Should Use This Book

This guide is for implementers of the Java™ Caching API specification to assist in running the test suite that verifies the compatibility of their implementation.

2. Before You Read This Book

Before reading this guide, you should familiarize yourself with the Java SE programming model. A good resource for the Java EE programming model is the JCP web site.

Before running the tests in the TCK, you should be familiar with JUnit ane Maven.

3. How This Book Is Organized

This book is organised into the following chapters:

- Chapter 1, Introduction (TCK) gives an overview of the principles that apply generally to all Technology Compatibility Kits (TCKs), outlines the appeals process and describes the TCK architecture and components. It also includes a broad overview of how the TCK is executed and lists the platforms on which the TCK has been tested and verified.

- Chapter 2, Appeals Process explains the process to be followed by an implementer should they wish to challenge any test in the TCK.

- Chapter 3, Installation explains where to obtain the required software for the TCK and how to install it. It covers both the primary TCK components as well as tools useful for troubleshooting tests.

- Chapter 4, Configuration details the configuration of the test harness.

- Chapter 5, Reporting explains the test reports that are generated by the TCK test suite and introduces the TCK audit report as a tool for measuring the completeness of the TCK in testing the JSR107 specification and in understanding how test cases relate to the specification.

- Chapter 7, Executing the Test Suite documents how the TCK test suite is executed.

- Chapter 8, Running Tests in IntelliJ IDEA shows how to run individual tests in IntelliJ IDEA and advises the best way to setup your IntelliJ IDEA environment for running the tests.

- Chapter 9, Debugging Tests in IntelliJ IDEA builds on Chapter 8, Running Tests in IntelliJ IDEA by detailing how to debug individual tests in IntelliJ IDEA.

Chapter 1. Introduction

This chapter explains the purpose of a TCK and identifies the foundation elements of the 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 tests are organised in the TCK by specification package.

For a particular implementation to be certified, all tests must pass.

A TCK is entirely implementation agnostic. Assertions are validated 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 implementer during development for the corresponding assertion.

1.2.1. Why Compatibility Is Important

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

- Compatibility testing is the means by which the JCP ensures that the Java platform does not become fragmented as it's ported to different operating systems and hardware.
- Compatibility testing benefits developers working in the Java programming language, enabling them to write applications once and deploy them across heterogeneous computing environments without porting.

- Compatibility testing enables application users to obtain applications from disparate sources and deploy them with confidence.
- Conformance testing benefits Java platform implementers by ensuring the same extent of reliability for all Java platform ports.

1.2.2. Compatibility Requirements

implementations must:

a) fully implement the Spec(s) including all required interfaces and functionality, except for optional features; and

b) must not modify, subset, superset, or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the JSR107 Specification.

1.3. About the TCK

The TCK is designed as a portable, configurable and automated test suite for verifying the compatibility of an implementation of the JSR107: JCache. The test suite is built on top of JUnit.

1.3.1. TCK Components

The TCK includes the following components:

- Maven modules for running tests.
- JUnit 4.11, the testing framework for selecting and executing the tests in the test suite.
- The test suite, which is a collection of JUnit tests, the JUnit test suite descriptor and supplemental resources that configure other required software components.
- README.md. A short note on how to run the TCK.

1.3.2. Handling Optional Features

A specification does not need to implement optional features but it does need to indicate whether it supports them via `CachingProvider.isSupported(OptionalFeature optionalFeature)`.

The TCK checks whether an optional feature is supported and only runs tests for that feature if it is. This is done automatically and requires no configuration.

1.3.2 Out-Of-Process Feature Implementations

The specification often states that it is not assumed that the implementation of a feature is in-process. Examples include `CacheWriter` and `CacheListener`. Accordingly the TCK provides network connectivity to instances of features which may be out-of-process to ascertain

state for the purposes of test assertion. Once again this is automatic and requires no configuration.

Chapter 2. Appeals Process

While the TCK is rigorous about enforcing an implementation's conformance to the JSR107 specification, it's reasonable to assume that an implementer may discover new and/or better ways to validate the assertions. This chapter covers the appeals process, defined by the Specification Leads, which allows implementers of the JSR107 specification to challenge one or more tests defined by the TCK.

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

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

2.1. Who can make challenges to the TCK?

Any implementer may submit an appeal to challenge one or more tests in the TCK.

2.2. What challenges to the TCK may be submitted?

Any test case 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 JSR107 EG by sending an e-mail to jsr-107-eg@jcp.org.

2.3. How these challenges are submitted?

To submit a challenge, a new issue should be created at https://github.com/jsr107/jsr107tck/issues. The appellant should describe the issue clearly. Any communication regarding the issue should be added to the comments on the issue at GitHub.

To submit an issue on GitHub, you must have a free GitHub account.

If you wish to make a private challenge, you should email to jsr-107-eg@jcp.org.

2.4. How and by whom challenges are addressed?

The challenges will be addressed in a timely fashion by the TCK Project Leads, as designated by the Specification Leads. The appellant can also monitor the process by following the issue online at GitHub.

2.5. How accepted challenges to the TCK are managed?

Accepted challenges will be acknowledged via the filed issue's comment section.

Communication between the TCK Project Lead and the appellant will take place via the issue comments. The issue's status will be set to "Closed" when the TCK project lead believes the issue to be resolved.

The appellant should, within 30 days, either close the issue if they agree, or reopen the issue if they do not believe the issue to be resolved. If the TCK Project Lead and appellant are unable to agree on the issue resolution, it will be referred to the JSR107 specification lead or his/her designate.

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 1.0.z where z 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 1.y.z where y > 0.

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 TCK project from https://github.com/jsr107/jsr107tck. The TCK is distributed as a Maven project with source code. Maven will resolve all dependencies automatically.

3.2. The TCK Environment

The TCK requires the following three software executables:

1. Maven 3.2.5 or higher. Available from http://maven.apache.org
2. Java 1.7 or higher. Available from http://www.oracle.com/technetwork/java/index.html
3. Git 1.8.4 or higher. Available from http://git-scm.com/downloads

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

3.3. Checking out the TCK

https://github.com/jsr107/jsr107tck contains instructions on how to “clone” the repository to your local environment.

Ensure you check out from https://github.com/jsr107/jsr107tck/tree/v1.0.z where z matches the latest appeal release of the TCK.

When cloned Git will created a directory called jsr107tck with sub-directories under.

Test Cases are annotated with @Test.

3.4. Verifying the Installation

Before testing your own implementation, gain familiarity with the tools by running the TCK against the RI. As the TCK only uses artifacts from Maven central, including the RI and API, there is no need to configure the Maven settings.xml file.

From the jsr107tck directory, run:

```
mvn clean install
```

JUnit will report, via Maven, the outcome of the run, and report any failures on the console.

Details can be found in:

```
jsr107tck/implementation-tester/specific-implementation-tester/target/surefire-reports
```

Chapter 4. Running the TCK Against Your Own Implementation

4.1. Configuring your Implementation as a Dependency

The jsr107tck/pom.xml specifies coordinate properties for the implementation to be tested.

In order to test your implementation, it must be packaged as a Maven artifact, and available to Maven in a repository.

Then change the following Maven properties to the coordinates for your implementation.

```
<implementation-groupId>org.jsr107.ri</implementation-groupId>
<implementation-artifactId>cache-ri-impl</implementation-artifactId>
<implementation-version>${project.parent.version}</implementation-version>
```

4.2. Providing an MBeanFactoryBuilder

For Version 1.0.0

The 1.0.0 specification requires that `MBeans` be made available in an `MBeanServer`. So that the TCK can resolve the actual MBeanServer that the implementation has registered MBeans to, an implementation must provide an `javax.management.MBeanServerBuilder` and set the following Maven properties in the jsr107tck/pom.xml to resolve it:

```
<javax.management.builder.initial>org.jsr107.ri.management.RITCKMBeanServerBuilder
      </javax.management.builder.initial>
<org.jsr107.tck.management.agentId>RIMBeanServer</org.jsr107.tck.management.agentId>
```

For Version 1.1.0

The 1.1.0 specification requires that `MBeans` be made available in the platform `MBeanServer` provided by `ManagementFactory#getPlatformMBeanServer()`.

4.3 Providing your concrete implementations to the TCK.

The unwrap tests need to know the concrete implementations. Change the following properties in jsr107tck/pom.xml:

```
<CacheManagerImpl>org.jsr107.ri.RICacheManager</CacheManagerImpl>
<CacheImpl>org.jsr107.ri.RICache</CacheImpl>
<CacheEntryImpl>org.jsr107.ri.RIEntry</CacheEntryImpl>
```

4.3. Running the Tests

From the jsr107tck directory, run:

```
mvn clean install
```

JUnit will report, via Maven, the outcome of the run, and report any failures on the console.

Details can be found in:
`jsr107tck/implementation-tester/specific-implementation-tester/target/surefire-reports`

Chapter 5. Running Tests from IntelliJ IDEA

IntelliJ IDEA, or any other IDE, is not required to execute or pass the TCK. However an implementer may wish to execute tests in an IDE to aid debugging the tests. This section explains how to do this from IntelliJ IDEA. An implementer can then adjust the process to their own IDE which should be similar.

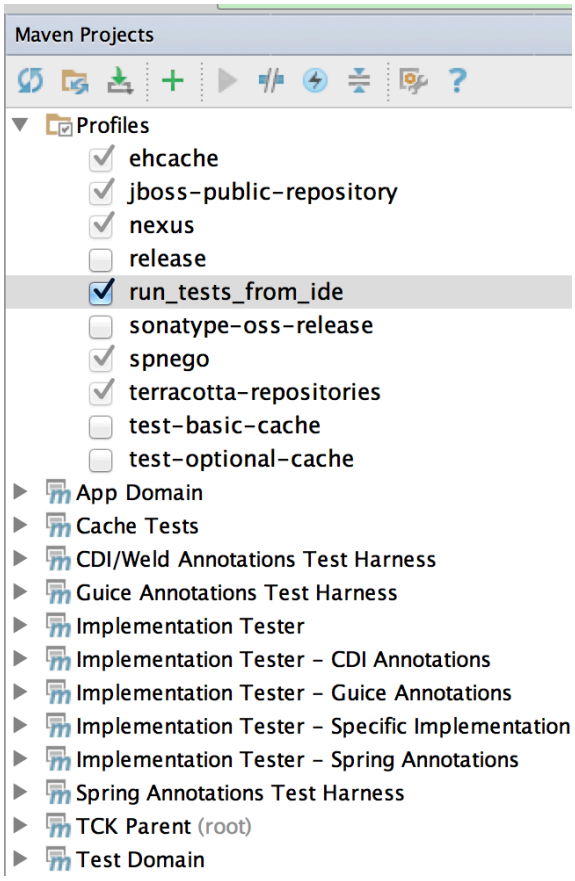
These instructions initially used IDEA 12.1.6, but were verified with IDEA Community 2017.2.

5.1. Setting Up an IDEA project

From IDEA File -> Open and select the pom.xml in the Git clone directory i.e. jsr107tck/pom.xml. IDEA will automatically create a multi-module project from the pom.xml.

5.2. Setting the Profile

Select the Maven Projects tab and tick the run_tests_from_ide profile as shown below.



The profile adds your implementation to the project's classpath.

5.3 Running Individual Tests

IDEA is Maven and JUnit aware. To run an individual test, right click on any method bearing the @Test annotation in the jsr107tck/cache-tests/src/test/java directory or subdirectories and click Run '_test_name_' from the context sensitive menu. It will then run the test.

5.4 Running Test Suites

To run a suite of tests, right click on any directory under the jsr107tck/cache-tests/src/test/java directory and click Run 'All Tests' from the context sensitive menu. It will run all tests, indicated with the @Test annotation, in classes below that point.

If you select the jsr107tck/cache-tests/src/test/java directory itself, all tests will be run.

Note that there is a test that is meant to fail when run from the IDE. It is the Caching#dummyTest which is defined as follows:

```
@Test
  public void dummyTest() {
    fail();
  }
```

This test exists as a test of the JUnit test suite exclusion filter.

5.5 Debugging

Debugging is as per running a test, except that ‘Debug’ is selected from the context sensitive menu rather than ‘Run’.

To debug, you must have your implementation’s source code and JCache’s source code attached to your project.

Chapter 6. Reporting

This chapter covers the three types of reports that can be generated from the TCK:

1. JUnit Test Results
2. Clover Coverage Report
3. Findbugs report.

The chapter also justifies why the TCK is good indicator of how accurately an implementation conforms to the JSR107 specification.

6.1 JUnit Test Results

The TCK test suite is really just a JUnit test suite. That means an execution of the TCK test suite produces all the same reports that JUnit produces. This section will go over those reports and show you were to go to find each of them.

6.1.1. Maven 2, Surefire and JUnit

When the TCK test suite is executed during the Maven test phase, JUnit 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, and other supported test frameworks.

6.1.1.1. Command Line Reporting

Surefire will print out its test results to the command line, including a summary line which looks like the following:

Results :

Tests run: 479, Failures: 0, Errors: 0, Skipped: 0

6.1.1.2. JUnit HTML Reports

JUnit 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

The first report, the test summary report, show below, is written to the file index.html. It produces the same information as the generic Surefire report.

To run these reports, from the jsr107tck directory, type:

```
mvn clean install site
```

An example surefire report is shown in JUnit Suite report in
https://www.dropbox.com/sh/94yggkttm2z3q9l/YvXAml1ir8.

6.2 Clover Coverage Report

Clover is a standard commercial tool for measuring code coverage. An open source license was provided for use with the RI.

The Clover tool measures coverage via classes. It does not cover interfaces. Classes in the API are measured directly. As the API is mostly interfaces, these have been measured indirectly by measuring statement execution in the RI.

Based on this, the TCK coverage is as follows:
javax.cache - 97.1% based on statement coverage. 96% based on method coverage.
RI - 90% based on statement coverage. 85.6% based on method coverage.

Clover is not available to be run against implementations.

A complete clover report against the final draft can be found at
https://www.dropbox.com/sh/94yggkttm2z3q9l/YvXAml1ir8.

6.3 Findbugs Report

The findbugs report may be generated by cloning the jsr107spec module to your local machine.
See https://github.com/jsr107/jsr107spec

Then from the directory run `mvn clean compile site`.

The results may be found in target/findbugs.

The run on the final draft showed no errors.

The FindBugs report against the final draft can be found at
https://www.dropbox.com/sh/94yggkttm2z3q9l/YvXAml1ir8.

6.4 TCK Coverage Metrics

The TCK coverage has been measured as follows:

- Assertion Breadth Coverage
- The TCK provides at least 75% coverage of identified assertions with test cases.

- Assertion Breadth Coverage Variance
- The coverage of specification sub-sections shows at 75% of sections with greater than 75% coverage.
- 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 TCK covers 100% of all API public methods using the Java CTT Sig Test tool.

6.5 TCK Coverage Report

A specification can be distilled into a collection of assertions that define the behavior of the software. This section introduces the TCK coverage report, which documents the relationship between the assertions that have been identified in the JSR107 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. TCK Assertions

The TCK developers have analyzed the JSR107 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:

A bean may declare multiple binding types.

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

```
<section id="2.3.3" title="Declare the bindings of a bean">

  ...

  <assertion id="d">

    <text>A bean may declare multiple binding types.</type>

  </assertion>

  ...
```

</section>

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

```
@Test

@SpecAssertion(section = "2.3.3", id = "d")

public void testMultipleBindings()

{

  Bean<?> model = getBeans(Cod.class, new ChunkyBinding(true), new
WhitefishBinding()).iterator().next();

  assert model.getBindings().size() == 3;

}
```

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

6.2.2. 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
```

Note

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

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

1. Chapter Summary - List the chapters (that contain assertions) in the specification document along with total assertions, tests and coverage percentage.
2. Section Summary - Lists the sections (that contain assertions) in the specification document along with total assertions, tests and coverage percentage.

3. Coverage Detail - Each assertion and the test that covers it, if any.
4. Unmatched Tests - A list of tests for which there is no matching assertion (useful during TCK development).
5. Unversioned Tests - A list of tests for which there is no @SpecVersion annotation on the test class (useful during TCK development).

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

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

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

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

Chapter 7. Running the Signature Test

One of the requirements of an implementation passing the TCK is for it to pass the JSR107 signature test. This section describes how the signature file is generated and how to run it against your implementation.

7.1. The sigtest tool

You can obtain the Sigtest tool (the TCK uses version 3.0) from the Sigtest home page at https://wiki.openjdk.java.net/display/CodeTools/sigtest. The user guide can be found at https://docs.oracle.com/javacomponents/sigtest-3-1/user-guide/toc.htm.

7.2. Running the signature test

The jsr107tck/sigtest directory contains everything you need to run the signature test.

Edit the `test.sh` script to point to your JCache API.

Then run `test.sh`. You should see output like the following:

```
vb➜jsr107/jsr107tck/sigtest(master)» ./test.sh
SignatureTest report
Base version: 1.0.0
Tested version: 1.0.0
Check mode: bin [throws removed]
Constant checking: on

Warning: Not found annotation type javax.enterprise.util.Nonbinding

STATUS:Passed.
```