Technology Compatibility Kit Reference Guide for JSR-352: Batch Applications for the Java Platform

Specification Lead: IBM

Project Home page:

java.net/projects/jbatch/

NOTE: Please see the change history at the end of this document to understand how these instructions have changed since the initial release of this TCK.

Table of Contents

0. 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 JSR-352: Batch Applications for the Java Platform specification.

The Batch Applications for the Java Platform TCK is built atop TestNG. The Batch Applications for the Java Platform TCK uses TestNG version 6.8 to execute the test suite.

A. Who Should Use This Guide

This guide is for implementers of the Batch Applications for the Java Platform specification, to assist in running the test suite that verifies the compatibility of their implementation.

B. Before You Read This Guide

Before reading this guide, you should familiarize yourself with the Batch Applications for the Java Platform specification.

Information about the specification, including links to the specification documents, can be found on the JSR-352 JCP page [http://jcp.org/en/jsr/detail?id=352].

Before running the tests of the JSR-352 TCK you should become familiar with TestNG. Documentation for TestNG is located at http://testng.org/doc/documentation-main.html

1. Introduction

The JSR-352 TCK tests implementations of the Batch Applications for the Java Platform specification, which describes the job specification language, Java programming model, and runtime environment for batch applications for the Java platform.

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.

For a particular implementation to be certified, all of the required tests must pass (i.e., the provided test suite must be run unmodified).

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

1.2 Compatibility Testing

Compatibility testing is the process of testing a technology implementation to make sure that it operates consistently with each platform, operating system, and other environment. The goal is to ensure portability.

Compatibility test development for a given feature relies on a complete specification and reference implementation for that feature. Compatibility testing is not primarily concerned with robustness, performance, or ease of use.

1.2.1 Why Compatibility Testing is Important

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

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

• Compatibility testing benefits developers working in the Java programming language, allowing them to write applications once and then to deploy them across heterogeneous computing environments without porting.

• Compatibility testing allows application users to obtain applications from disparate sources and deploy them with confidence.

• Compatibility testing benefits Java platform implementers by ensuring a level playing field for all Java platform ports.

1.3 About the JSR 352 TCK

The Batch Applications for the Java platform TCK is designed as a portable, configurable and automated test suite for verifying the compatibility of an implementation of JSR-352: Batch Applications for the Java platform

1.3.1 JSR 352 TCK Specifications and Requirements

This section lists the applicable requirements and specifications for the JSR-352 TCK

- **JSR 352 API** - The Java API defined in the JSR-352 specification and provided by the reference implementation

- **Reference implementation** - The designated SE reference implementation for compatibility testing of the JSR-352 specification can be obtained as a zip download here http://java.net/projects/jbatch/downloads

1.3.2 JSR 352 TCK Components

The JSR 352 TCK includes the following components:

- **TestNG -** The JSR-352 TCK requires version 6.8 of TestNG.
- **Dependency injection implementation -** The JSR-352 TCK requires an implementation of JSR 330 (javax.inject.jar).
- **XMLUnit -** The JSR352 TCK requires version 1.1 of XMLUnit.
- **Test suite -** a collection of TestNG tests, the TestNG test suite descriptor and supplemental resources, such as the test artifacts used.
- **TCK documentation –** README file, and this document.

1.4 What Tests Do I Need To Pass (to pass the TCK)?

To summarize what is spelled out below in more detail, in order to pass the JSR 352 TCK you must run against your implementation, passing 100% of both the:

- Signature Tests
- TestNG Test Suite

The two types of tests are not encapsulated in a single execution task or command; they must be executed separately from each other and each must be executed separately for each version of Java tested (e.g. Java 6 or Java 7).

2. Appeals Process

If a test is determined to be invalid in function or if its basis in the specification is suspect, the test may be challenged by any implementer of the JSR-352 specification. Each test validity issue must be covered by a separate test challenge. Test validity or invalidity will be determined based on its technical correctness, such as:

- The test itself has bugs (i.e., program logic errors)
- Specification item covered by the test is ambiguous
- Test does not match the specification
- Test assumes unreasonable software requirements/configuration
- Test is biased to a particular implementation

Challenges based upon issues unrelated to technical correctness as defined by the specification will normally be rejected, as the specification document is controlled by a separate process.

Tests found to be invalid will either be placed on the Exclude List for that version of the JSR352-TCK or have a corrected or alternate test made available.

2.1. Who can make challenges to the TCK?

Any implementer may submit an appeal to challenge one or more tests in the JSR-352 TCK.

2.2. What challenges to the TCK may be submitted?

Any test case (e.g., @Test method or an artifact class), test case configuration, annotations and other resources may be challenged by an appeal. Assertions made by the specification are controlled by a separate process.

2.3. How these challenges are submitted?

A test challenge must be made by creating a new issue in the issue tracker of the java.net/projects/jbatch project. Summary and description fields should be completed.

2.4. How and by whom challenges are addressed?

The challenges will be addressed in a timely fashion by a JSR-352 lead, as designated by Specification Lead, IBM.

2.5. How are challenges managed?

If the test challenge is approved and one or more tests are invalidated, the lead may correct or replace the invalidated tests, or place the invalidated tests on the Exclude List for that version of the TCK.

Accepted challenges will be acknowledged via comments in the issue tracker. The issue status will be set to resolved when the lead believes the issue is resolved.

The implementer should, within 30 days, either close the issue if he/she agrees, or reopen the issue if it is not considered to be resolved. Resolved issues not addressed for 30 days will be closed.

3. Installation

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

3.1 Obtaining the Software

You can obtain a release of the Batch Applications for the Java Platform (JSR-352) TCK as a java.net zip download.

The JSR-352 TCK is distributed as a zip file, which contains the TCK artifacts (the test suite binary and source, porting package SPI binary and source, the test suite descriptor) in

`/artifacts`, the TCK library dependencies in `/lib` and documentation in `/doc`

You can also download the current source code from the Git repository here:
http://java.net/projects/jbatch/sources

The JSR-352: Batch Applications for the Java Platform TCK can be obtained from the java.net download page [**http://java.net/projects/jbatch/downloads]**

Also see the Wiki page [**http://java.net/projects/jbatch/pages/RI_TCK**] for more detailed descriptions of the difference between various downloads, as well as change history.

3.2 The TCK Environment

The software can simply be extracted from the ZIP file.

Once the TCK is extracted, you'll see the following structure:

```
jsr352-tck-1.0/
     artifacts/
     doc/
     lib/
     build.xml
     jsr352-tck.properties
     LICENSE.txt
     NOTICE.txt
     readme.txt
```

`artifacts` contains all the test artifacts pertaining to the TCK: The TCK test classes and source, the TCK SPI classes and source, the TestNG suite.xml file and the SigTest signature files.

`doc` contains the documentation for the TCK (this PDF file)

`lib` contains the necessary prereqs for the TCK

`build.xml` is an ant build file which is used to run (and optionally build from source) the TCK.

`jsr352-tck.properties` is the properties file where required properties for the TCK are specified.

3.3 TCK test classes

The TCK test methods are contained in a number of test classes in the `com.ibm.jbatch.tck.tests` package. Each test method is flagged as a TestNG test using the @org.testng.annotations.Test annotation.

3.4 TCK test artifacts

Besides the test classes themselves, the JSR-352 TCK is comprised of a number of test artifact classes located in the `com.ibm.jbatch.tck.artifacts` package. These are the batch artifacts that have been implemented based on the JSR-352 API, and which are used by the individual test methods. The final set of test artifacts is the set of test JSL (XML) files, which are packaged in the `META-INF/batch-jobs` directory within `artifacts/jsr352-tck-impl.jar`.

The basic test flow simply involves a TestNG test method using the JobOperator API to start (and possibly restart) one or more job instances of jobs defined via one of the test JSLs, making use of some number of `com.ibm.jbatch.tck.artifacts` Java artifacts. The JobOperator is wrapped by a thin layer which blocks waiting for the job to finish executing (more on this in the discussion of the "porting package SPI" later in the document).

4. Configuration

4.1 TCK Properties

In order to run the TCK, you must define a property pointing to the JSR-352 runtime implementation that you are running the TCK against.

You will need to set one required property prior to running the JSR-352 TCK. This property is defined in the `jsr352-tck.properties` as follows:

Property = Required/Example Value	Description
`batch.impl.classes=$HOME/foo/lib/classes:$HOME/foo/lib/foo.jar:$HOME/foo/lib/batch-api.jar`	Path listing of the SE JSR-352 runtime implementation (that you are running the TCK against)

An optional property with name `jvm.options` is provided to specify JVM arguments using the TestNG <jvmarg line=""/> function: This property should list the JVM arguments, separated by spaces.

Finally, some of the TCK tests sleep for a short period of time to allow an operation to complete or to force a timeout. These time values are defaulted via properties that are also specified in `jsr352-tck.properties`. These values can be adjusted if timing issues are seen in the implementation being tested. Refer to the documentation for a specific test (i.e. the comments in the test source) as to how the time value is used for that test.

4.2 Porting Package SPI

The Batch Applications for the Java Platform (JSR-352) TCK relies on an implementation of a "porting package" SPI to function, in order to verify test execution results. The reason is that the JSR 352 specification API alone does not provide a convenient-enough mechanism to check results.

A default, “polling” implementation of this SPI is shipped within the TCK itself. The expectation is that the typical JSR 352 implementation will be content to use the TCK-provided, default implementation of the porting package SPI.

Further detail on the porting package is provided later in this document, in case you wish to provide your own, different implementation.

4.3 Configuring TestNG to run the TCK

TestNG is responsible for selecting the tests to execute, the order of execution, and reporting the results. Detailed TestNG documentation can be found at testng.org [http://testng.org/doc/documentation-main.html].

The `artifacts/jsr352-tck-impl-suite.xml` artifact provided in the TCK distribution must be run by TestNG 6.9 (described by the TestNG documentation as "with a testng.xml file") unmodified for an implementation to pass the TCK. This file also allows tests to be excluded from a run (for, say, debugging purposes).

5. Executing Signature Tests

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

5.1 Obtaining the Signature Test Tool (and prerequisites)

You can obtain the Sigtest tool from the Sigtest home page at http://sigtest.java.net

The other prereq needed for the signature test is an implementation of class **javax.enterprise.util.Nonbinding**. This is, (at the time of this writing anyway), available for download as part of the CDI API, via Maven and other download mechanisms, among other possible options.

(We don't count the javax.inject.* package as a “prereq” here because, although it's a dependency, it is packaged along with our TCK).

5.2 JDK/JRE prerequisite

The signature test files to compare your implementation against were created with Oracle JDKs. The tests will not pass when run with the levels of IBM JDK/JRE available at the time of this writing. An Oracle JDK/JRE should probably be used, then, when running the signature tests.

To summarize: for each of Java 6 and Java 7, both the 'java' executable and the 'lib/rt.jar' referenced in the sample “running the signature tests” commands below should be part of a single Oracle JDK/JRE (a different one for each of Java 6 and Java 7, of course).

5.3 Running the Signature Tests

The TCK package contains the files `jsr352-api-sigtest-java7.sig` and `jsr352-api-sigtest-java6.sig` (in the `artifacts` directory).

Run the signature test by executing a command like the following:

```
java -jar $SIGTEST_DEV_JAR SignatureTest -static -package javax.batch
-filename jsr352-api-sigtest-java7.sig -classpath
$JAVA_HOME/lib/rt.jar:$JAVAX_INJECT_JAR:$JAVAX_ENTERPRISE_UTIL_JAR:
$MY_BATCH_API_JAR
```

Note the four dependencies here (not counting the JDK/JRE itself), the locations of which you may need to modify:

- SIGTEST_DEV_JAR: the location of '`sigtestdev.jar`' from your Sigtest download.
- JAVAX_INJECT_JAR: (for class javax.inject.Qualifier, shipped with TCK)
- JAVAX_ENTERPRISE_UTIL_JAR: (for class javax.enterprise.util.Nonbinding, not shipped with TCK)
- MY_BATCH_API_JAR: Your own API JAR from your own implementation, which you are running the signature test against.

Here is an example showing a sample set of values for the shell variables used in the shorthand above, when running the signature test against the JSR 352 Reference Implementation.

It assumes:

1) You have unzipped both RI and TCK into the same top-level directory.
2) You are executing from working directory *`.../jsr352-tck-1.0`*
3) You have already copied *`sigtestdev.jar`* into this directory
4) You have already copied *`cdi-api-1.0.jar`* into this directory (this JAR contains the **javax.enterprise.util.Nonbinding** class)
5) Your 'java' executable and your 'rt.jar' come from a Java 7 JDK/JRE, since in the example you are running against the Java 7 signature file (based on the ***-filename*** argument)

```
java -jar sigtestdev.jar SignatureTest -static -package javax.batch
-filename artifacts/jsr352-api-sigtest-java7.sig -classpath
$JAVA_HOME/lib/rt.jar:lib/javax.inject.jar:../jsr352-ri-
1.0/javax.batch.api.jar:cdi-api-1.0.jar
```

Again, be sure to choose the correct version of the signature file depending on your the Java version (6 or 7) of your JDK/JRE.

5.4 Determining success

The output of your execution should include, at the very end:

```
STATUS:Passed
```

Again, in order to pass the JSR352 TCK you have to make sure that your API passes the signature tests.

5.5 Forcing a Signature Test failure (optional)

For additional confirmation that the signature test is working correctly, a failure can be forced by doing the following:

- Edit `jsr352-api-sigtest-java7.sig`
- Modify one of the class signatures. For example, change this:

```
CLSS public abstract interface
javax.batch.api.chunk.CheckpointAlgorithm
meth public abstract boolean isReadyToCheckpoint() throws
java.lang.Exception
meth public abstract int checkpointTimeout() throws
java.lang.Exception
meth public abstract void beginCheckpoint() throws
java.lang.Exception
meth public abstract void endCheckpoint() throws java.lang.Exception
```

to the following:

(changing the `isReadyToCheckpoint` method to accept a `java.lang.String` parameter)

```
CLSS public abstract interface
javax.batch.api.chunk.CheckpointAlgorithm
meth public abstract boolean isReadyToCheckpoint(java.lang.String)
throws java.lang.Exception
meth public abstract int checkpointTimeout() throws
java.lang.Exception
meth public abstract void beginCheckpoint() throws
java.lang.Exception
meth public abstract void endCheckpoint() throws java.lang.Exception
```

When the signature test is then run, it will fail with the following error:

```
Missing Methods
---------------
javax.batch.api.chunk.CheckpointAlgorithm:                    method
```

```
public abstract boolean
javax.batch.api.chunk.CheckpointAlgorithm.isReadyToCheckpoint(java.la
ng.String) throws java.lang.Exception

Added Methods
-------------
javax.batch.api.chunk.CheckpointAlgorithm:                    method
public abstract boolean
javax.batch.api.chunk.CheckpointAlgorithm.isReadyToCheckpoint() throw
s java.lang.Exception

duplicate messages suppressed: 1

STATUS:Failed.3 errors
```

5.6 Creating the Signature File (optional, for reference)

Though the requirement for passing the TCK is to run the signature test against the exact copies of the ".sig" files shipped within the TCK, it can be helpful for debugging to understand how those files were generated.

The ".sig" files were created using a command like the following:

```
java -jar sigtestdev.jar Setup -static -package javax.batch -filename
   jsr352-api-sigtest-java7.sig -classpath
$JAVA_HOME/lib/rt.jar:lib/javax.inject.jar:../jsr352-ri-
1.0/javax.batch.api.jar:cdi-api-1.0.jar
```

This example assumes you are executing from the *`jsr352-tck-1.0`* directory, similar to the "run" command example in Section 5.3. Modify the path as needed in a similar manner.

Also note that the Java version of the 'java' executable in the above command determines what Java version the signature file will correspond to. In this example, we use a filename of `jsr352-api-sigtest-java7.sig'` to show that this a Java 7 executable (assumed for this example).

6. Executing TestNG Test Suite

The `build.xml` file is used for running the test suite in standalone mode with ant.
The default target, `run`, will invoke TestNG, running the tests specified in the suite xml file at `artifacts/jsr352-tck-impl-suite.xml` (described
by the TestNG documentation as "with a testng.xml file"). A report will be generated by TestNG in the `results` directory.

The list of test cases to run can be customized by modifying the the TestNG suite xml file at `artifacts/jsr352-tck-impl-suite.xml`. (Note that an implementation must run against that provided suite.xml file as-is, to pass the TCK.

6.1 Timeouts

The JobOperatorBridge makes use of the following system property:

```
tck.execution.waiter.timeout
```

with a default value of 900000 (900 seconds). The intention here is that the test should not wait forever if something catastrophic occurs causing the job to never complete (or if the porting package SPI "waiter" is never notified for some reason). The test also can't end too soon, causing a test failure because the wait was not long enough.

This timeout value can be customized (say, to increase when debugging or decrease to force a faster failure in some cases).

Note that some of the tests (e.g. the chunk tests involving time-based checkpointing) will take at least 15-25 seconds to run on any hardware, so any value less than that for the whole TCK will cause some test failures simply due to timing (and not because of any failure in the underlying JSR 352 implementation).

The 900 seconds value, then, was chosen to avoid falsely reporting an error because of timing out too soon, allowing plenty of leeway. It also facilitates debugging. It does not, however, provide "fast failure" in case of a hang or runaway thread.

6.2 Building the TCK (optional, for reference):

The TCK tests can be optionally built from source. However, note that for an implementation to pass the TCK, it must run against the shipped TCK test suite binary as-is (and not against a modified TCK). Still it may be convenient to be able to build the TCK from source for debugging purposes.

The TCK source is included with the TCK zip, and can be located in `jsr352-tck-impl-src.jar`. Extract this archive to a directory, and note that location. Modify the "tck-src" property to point to the directory to which you've extracted the source. The "compile" target can then be used to build the TCK from source, with the resulting class files being located in the "build" directory.

7. Additional Information

7.1 Coverage

A challenge when writing software tests is to assess the quality of a test suite. A practical way to answer that question is to analyze what code is executed during a test run, i.e., to measure the code coverage of the test suite. The coverage analysis tool EMMA was used to transparently measure coverage while executing the test suite.

API Signature Coverage statement: The JSR352 TCK covers 100% of all API public methods using EMMA Coverage analysis tool.

The tests themselves are commented to indicate test assertion and test strategy.

7.2 Porting Package in-depth (optional)

The two porting package SPI classes in the JSR-352 TCK are:

`com.ibm.jbatch.tck.spi.JobExecutionWaiter`
`com.ibm.jbatch.tck.spi.JobExecutionWaiterFactory`

The default implementations of these provided by the JSR-352 TCK are:

```
com.ibm.jbatch.tck.polling.TCKPollingExecutionWaiterFactory$TCKPollingExecutionWaiter
com.ibm.jbatch.tck.polling.TCKPollingExecutionWaiterFactory
```

The interface definitions are simply:

```
public interface JobExecutionWaiterFactory {
       public JobExecutionWaiter createWaiter(long executionId, JobOperator jobOp, long sleepTime);
}

public interface JobExecutionWaiter {
       JobExecution awaitTermination() throws JobExecutionTimeoutException;
}
```

This SPI can be understood with a simple example showing how it used by the TCK (this sample code is extracted from class **com.ibm.jbatch.tck.utils.JobOperatorBridge**)

```
  long executionId = jobOp.start(jobName, jobParameters);
  JobExecutionWaiter waiter = waiterFactory.createWaiter(executionId, jobOp, sleepTime);
  try {
       terminatedJobExecution = waiter.awaitTermination();
  } catch (JobExecutionTimeoutException e) {
     …
```

So all that's happening here is that we're “waiting” for the asynchronous job execution to complete with a blocking method that will either return when execution is complete or throw an exception if we reach the specified “`sleepTime`”.

And the provided, `com.ibm.jbatch.tck.polling.TCKPollingExecutionWaiterFactory` implementation simply polls repeatedly until the timeout.

Finally, note that the **java.util.ServiceLoader** mechanism is used to reference and load the particular SPI implementation. This implies that you need to update file **META-INF/services/com.ibm.jbatch.tck.spi.JobExecutionWaiterFactory** and update the contents with your factory classname to replace the default implementation.

8. Change History

8.1 Updated – Jun 26, 2013

Major Changes (impacts how/whether tests succeed vs. fail)

- The signature tests were re-generated with class **javax.enterprise.util.Nonbinding** on the classpath (at generation time). The bottom line here: you need to run the signature test with **javax.enterprise.util.Nonbinding** on the classpath (e.g. from the CDI API JAR) when running the test as well.

  The reason for the change, in short, is that SigTest assumes annotations of API classes which are not themselves on the classpath behave as @Documented annotations and are therefore part of the public API, when in this case the @Nonbinding class is not actually annotated with @Documented. So it was a mistake to generate the original .sig files without @Nonbinding on the classpath.

  Be sure to download the latest version of the TCK, as running with the new instructions will result in signature test failure when run against the .sig files in the older version of the TCK download.

Minor Changes (better clarification, organization, etc.)

- Added sections 1.4, 5.2 for additional clarification
- Reworked signature test section to be clear that generating your own signature file was not required
- Spelled out details of porting package SPI
- Added table of contents
- Some other sections were re-organized