Jakarta Batch TCK Reference Guide

Table of Contents

1. Preface . . . 1
   1.1. Licensing . . . 1
   1.2. Who Should Use This Guide . . . 1
   1.3. Terminology - "SE mode" vs. "EE mode" . . . 1
   1.4. Before You Read This Guide . . . 1
   1.5. Terminology - "Standalone TCK" . . . 1
2. Major TCK Changes: Adding Platform Coverage + Switch to Maven . . . 2
3. What Tests Must I Pass To Certify Compatibility? . . . 2
   3.1. Runtime Tests & Signature Tests Required . . . 2
   3.2. Runtime tests - SE vs. EE . . . 2
   3.3. Java SE level - Java 11 or Java 17 . . . 3
4. Prerequisites . . . 3
   4.1. Software To Install . . . 3
   4.2. Other Dependencies . . . 3
5. A Guide to the TCK Distribution . . . 3
   5.1. Obtaining the Software . . . 3
   5.2. The TCK Environment . . . 3
   5.3. A Quick Tour of the TCK Artifacts . . . 4
6. Quick Start - Run the TCK against jbatch . . . 5
7. TCK Test Requirements . . . 6
   7.1. Runtime tests . . . 6
   7.2. Signature tests . . . 6
8. Requirements As Illustrated by TCK "runners" . . . 6
   8.1. Modifiable runner (with release version) . . . 6
   8.2. Runners use TCK ZIP artifacts for viewing, not for execution . . . 7
   8.3. Runners inherit from Batch TCK modules . . . 7
9. JUnit SE suite - 'se-classpath' runner . . . 7
   9.1. Maven failsafe 'core' execution . . . 7
   9.2. Maven failsafe 'appbean' execution . . . 9
   9.3. Alternative Approaches . . . 10
   9.4. Expected Results . . . 10
   9.5. In case of Failure . . . 11
10. JUnit EE suite - 'platform-arquillian' runner . . . 11
   10.1. Runtime-specific Maven profile . . . 11
   10.2. 'jakarta.batch.arquillian.exec-parent' parent module . . . 11
   10.3. Jakarta Enterprise Beans vs web executions, core vs appbean executions . . . 12
   10.4. One-time setup used for all failsafe executions . . . 12
   10.5. Maven failsafe core execution: 'core-ejb' . . . 14

10.6. Maven failsafe appbean execution: 'appbean-ejb' . . . 16
10.7. Maven failsafe web executions: 'core-web', 'appbean-web' . . . 18
10.8. Alternative Approaches . . . 18
10.9. Expected Output . . . 18
In case of Failure . . . 19
11. Signature Tests . . . 19
11.1. sigtest runner . . . 19
11.2. Java 11 vs Java 17 . . . 19
11.3. Maven setup - OPTIONAL . . . 19
11.4. Maven test execution - REQUIRED . . . 20
11.5. Expected Output . . . 22
11.6. Forcing a Signature Test failure (optional) . . . 23
11.7. Generating the Signature Files (optional) . . . 25
12. JUnit test suite in detail (Optional) . . . 25
12.1. The flow of a typical TCK test . . . 25
12.2. Porting Package SPI . . . 25
12.3. Adjusting the Default Timeout Value . . . 26
12.4. Default Test-Specific Wait Times, and How to Adjust Timeout Values . . . 26
12.5. Working with TCK source (debugging, etc.) . . . 27
12.6. Arquillian / EE Platform Tests . . . 27
12.7. Multiple Maven Executions for Multiple Classpath Configurations . . . 27
13. TCK Challenges/Appeals Process . . . 27
13.1. Filing a Challenge . . . 28
14. Certification of Compatibility . . . 28
14.1. Filing a Certification Request . . . 28
15. TCK SPI "Porting Package" in-depth (optional) . . . 29
16. Links . . . 30
17. Change History . . . 30
17.1. Initial Release - Jakarta Batch 1.0 . . . 30
17.2. Update - Jakarta Batch 2.0 . . . 30
17.3. Update (and major rework moving from Ant→Maven) - Jakarta Batch 2.1 . . . 30

1. 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 Batch specification.

The specification describes the job specification language, Java programming model, and runtime environment for Jakarta Batch applications.

1.1. Licensing

The Jakarta Batch TCK is provided under the **Eclipse Foundation Technology Compatibility Kit License - v 1.0** [https://www.eclipse.org/legal/tck.php].

1.2. Who Should Use This Guide

This guide will assist in running the test suite verifies implementation compatibility for BOTH:

- implementers of the Jakarta Batch specification AND
- implementers of the entire Jakarta EE Platform (of which Jakarta Batch is one component).

1.3. Terminology - "SE mode" vs. "EE mode"

Building on the previous point, it is convenient to use, as shorthand, the term "EE mode" when talking about the TCK constructs and requirements specifically for users running the TCK to certify against the entire EE platform. It is a convenient shorthand term too, then, to use the term "SE mode" for users that are only trying to certify against the Jakarta Batch specification, though this term in some ways might be misleading. Since the tests and the execution requirements are different for each set of users, it is helpful to use this shorthand, and we do at times in this document.

1.4. Before You Read This Guide

Before reading this guide, you should familiarize yourself with the Jakarta Batch Version 2.1 specification, which can be found at https://jakarta.ee/specifications/batch/2.1/.

Other useful information and links can be found on the eclipse.org project home page for the Jakarta Batch project [https://projects.eclipse.org/projects/ee4j.batch] and also at the GitHub repository home for the specification project [https://github.com/eclipse-ee4j/batch-api].

1.5. Terminology - "Standalone TCK"

We sometimes refer to this Batch TCK as the "standalone" TCK. This usage comes from the fact that Jakarta Batch is part of the Jakarta EE Platform, which has a platform-level TCK, which we're distinguishing this "standalone" TCK from.

2. Major TCK Changes: Adding Platform Coverage + Switch to Maven

This version of the Jakarta Batch TCK introduces two major changes to the TCK:

1. We change the official execution of the standalone TCK from Ant to Maven. Though the TCK has long been built with Maven and we even have included execution or "runner" Maven modules, our official documentation described an Ant-based execution. This updated version of the TCK Reference Guide details the requirements and procedures for performing an official Maven-based execution of this TCK.
2. We add coverage for verifying compatibility with the Batch specification when running on the Jakarta EE Platform TCK. With this change we intend to eliminate the need to contribute the same tests packaged here in the "standalone" Jakarta Batch TCK with the Jakarta Platform TCK. Instead someone verifying a compatible implementation of the entire Platform will need to run this Jakarta Batch TCK in "EE mode" (roughly speaking), along with running the remaining Platform tests for verifying the remainder of the Platform.

3. What Tests Must I Pass To Certify Compatibility?

3.1. Runtime Tests & Signature Tests Required

Whether you are using this guide and the TCK to certify compatibility of a batch implementation against the Jakarta Batch specification alone, or certifying compatibility with the entire Jakarta EE Platform, you will in both cases need to run the TCK against your implementation and pass 100% of both the:

- JUnit 5 runtime tests
- Signature tests

The two types of tests are not encapsulated in a single execution or configuration; typically they must be executed via multiple executions, as explained in detail later in the guide.

By "runtime" tests we simply mean tests simulating Jakarta Batch applications running against the batch implementation attempting to certify compatibility. These tests verify that the batch applications behave according to the details defined in the specification, as validated by the TCK test logic.

3.2. Runtime tests - SE vs. EE

The set of runtime tests required to use the TCK to certify the Batch portion of Jakarta EE Platform compatibility ("EE mode") is a superset of the set of runtime tests required to satisfy compatibility only with the Jakarta Batch specification ("SE mode").

3.3. Java SE level - Java 11 or Java 17

The JDK used during test execution must be noted and listed as an important component of the certification request. In particular, the Java SE version is important to note, and this version must be used consistently throughout both the JUnit 5 runtime and Signature tests for a given certification request.

For the current TCK version, this can be done with either Java SE Version 11 or Version 17.

4. Prerequisites

4.1. Software To Install

1. **Java/JDK** - Install the JDK you intend to use for this certification request (Java SE Version 11 or Version 17).
2. **Maven** - Install Apache Maven 3.x.

4.2. Other Dependencies

1. **Arquillian** - Since the EE Platform TCK uses Arquillian to execute tests within an Arquillian "container" for certifying against the EE Platform, you must configure an Arquillian adapter for your target runtime.
2. **Signature Test Tool** - No action is needed here, but we note that the signature files were built and should be validated with the Maven plugin with group:artifact:version coordinates: **org.netbeans.tools:sigtest-maven-plugin:1.6**, as used by the sample sigtest runner included in the TCK zip. This is a more specific direction than in earlier releases, in which it was left more open for the user to use a compatible tool. Since there are small differences in the various signature test tools, we standardize on this version.

5. A Guide to the TCK Distribution

This section explains how to obtain the TCK and extract it on your system.

5.1. Obtaining the Software

The Jakarta Batch 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 XML definitions, and signature files) in `/artifacts`, the documentation in `/doc`, and some example Maven modules showing how to run the TCK in `/runners`. You can access the current source code from the Git repository: https://github.com/eclipse-ee4j/batch-tck.

5.2. The TCK Environment

The software can simply be extracted from the ZIP file. Once the TCK is extracted, you'll see the

following structure:

```
jakarta.batch.official.tck-x.y.z/
    artifacts/
    doc/
    runners/
       platform-arquillian/
       se-classpath/
       sigtest/
    LICENSE_EFTL.md
    NOTICE.md
    README.md
```

In more detail:

`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 signature test files.

`doc` contains the documentation for the TCK: this reference guide, plus a script that helps provide an example of how to run the TCK against the 'jbatch' implementation.

`runners` contains three Maven modules that provide samples for executing different portions of the TCK:

- The **se-classpath** runner shows an execution of the JUnit runtime tests against the 'jbatch' implementation (in SE mode, NOT exercising the full Jakarta EE Platform).
- The **platform-arquillian** runner shows an execution of the Platform version of the JUnit runtime test suite, using Arquillian, which must be used when using the Batch TCK as part of certifying compatiblity with the full Jakarta EE Platform).
- The **sigtest** runner shows an execution of the signature tests against the 'jbatch' implementation.

5.3. A Quick Tour of the TCK Artifacts

5.3.1. TCK JUnit test classes/methods

The TCK test methods are contained in a number of test classes in the `com.ibm.jbatch.tck.tests.*` packages. Each test method is annotated as a JUnit 5 test using JUnit 5 annotations such as `org.junit.jupiter.api.Test`, `org.junit.jupiter.params.ParameterizedTest`, etc.

5.3.2. TCK test batch artifacts

Besides the test classes themselves, the Jakarta Batch TCK is comprised of a number of test classes located in the `com.ibm.jbatch.tck.artifacts` package which implement the interfaces defined in the Jakarta Batch API (e.g. ItemReader, ItemProcessor, ItemWriter, the various listeners, etc.). Together these batch artifacts "implement" the jobs run in individual test methods. Another key set of batch artifacts is the set of test Job Specification Language (JSL) XML files, which are packaged in the

`META-INF/batch-jobs` directory within `artifacts/com.ibm.jbatch.tck-x.y.z.jar`.

5.3.3. JUnit 5 "suite" definition XML files

Here we use the term "suite" informally to describe groups of tests required to pass the TCK (and NOT specifically to refer to any particular "suite" construct defined by the JUnit 5 API).

There are three JUnit 5 test "suites" included in the TCK. There is a separate "suite" for each of the "SE mode" TCK and the Batch portion of the "Jakarta EE Platform" TCK and there is a third suite that must be run in both SE and EE "modes".

The SE suites:

1. The `artifacts/batch-tck-impl-SE-core-suite-includes.txt` suite defines the majority of the tests.
2. The `artifacts/batch-tck-impl-appjoboperator-suite-includes.txt` defines a few additional tests.

The reason we need two suites here is that the tests in the second suite require a different classpath configuration than those of the first.

Likewise, for the EE tests we have:

1. `artifacts/batch-tck-impl-EE-platform-core-suite-includes.txt`
2. `artifacts/batch-tck-impl-appjoboperator-suite-includes.txt`

Note: An implementation **MUST** run against each provided suite XML file unmodified for an implementation to pass the TCK.

5.3.4. API Signature Files

The two signature files, for Java 11 and 17, respectively:

1. `artifacts/sigtest-1.6-batch.standalone.tck.sig-2.1-se11-OpenJDK-J9`
2. `artifacts/sigtest-1.6-batch.standalone.tck.sig-2.1-se17-TemurinHotSpot`

6. Quick Start - Run the TCK against jbatch

We stop and take a break from all the description and give the user a concrete way to jump in and get something running.

To run the "SE mode" TCK against the **jbatch** implementation, simply perform the following steps (which should complete without error).

1. Download TCK ZIP
2. jar xf <zip>
3. cd jakarta.batch.official.tck-2.1.0/runners/se-classpath
4. mvn verify
5. cd ../sigtest
6. mvn verify

7. TCK Test Requirements

Because there is flexibility regarding how a user could use Maven to configure a TCK execution, we make a separate, clear note here of the required number of tests needed to be passed in order to claim compliance via this TCK.

7.1. Runtime tests

For the runtime test (JUnit) component of the TCK:

- 178 tests must be passed to successfully execute the SE TCK suite
- 374 tests must be passed to successfully execute the EE TCK suite

7.2. Signature tests

All signature tests must be passed. The sigtest tool must be pointed to the TCK-provided signature file with an appropriate configuration and get a clean result without error or failures.

8. Requirements As Illustrated by TCK "runners"

Each of the three runners includes a POM and a goal configured to run in the "integration" phase.

In general there are often several ways to accomplish any given task in Maven. We attempt to describe below the essential aspects of the configuration necessary to execute the TCK and claim compliance vs. the parts that may be modified, taking the runner POM only as a sample starting point.

There is a tension here: on the one hand we say that the runners' POMs and configurations plus the descriptions below define requirements for setting up a valid TCK execution to certify compliance. On the other hand we say this is just an example and do not prescribe in exacting detail precisely which Maven constructs can vs. can't be used in writing one's own POM.

We accept this ambiguity and think we can live with this compromise. If it turns out that implementors attempting to certify compliance find these distinctions need clarification, we can take that feedback and improve the documentation in a future release.

8.1. Modifiable runner (with release version)

One unusual aspect of a POM like this is that, one the one hand, it has a non-SNAPSHOT version and is released to Maven Central. On the other hand, it is delivered in the TCK ZIP in a way that the user may simply choose to edit it and run the TCK against their implementation, without changing the Maven GAV coordinates to something they own. We don't consider this a problem but beware, e.g. if you do `mvn install` you'll be overwriting this locally. (This shouldn't affect the TCK execution since nothing else uses any runner as a dependency or parent).

8.2. Runners use TCK ZIP artifacts for viewing, not for execution

Note: the TCK runners mentioned here are configured to run "against" artifacts referenced as Maven dependencies, and so obtained from a Maven repository (e.g Maven Central). They are configured this way even though as explained in the guide to the TCK ZIP contents, the artifacts are all present in the TCK zip.

This is a significant difference with the previous Jakarta Batch Ant-based TCK.

Starting with the Jakarta Batch 2.1 TCK release, the runner configurations are NOT set up referencing relative paths within the TCK zip. The artifacts packaged in the ZIP (detailed below) only help create a self-contained TCK package, providing an easy view of what artifacts are relevant to the TCK in a single place without requiring the user to follow Maven dependency references.

While it would be possible to construct a valid TCK execution configuration running against the artifacts in the TCK ZIP, an example is not provided.

8.3. Runners inherit from Batch TCK modules

The runners each inherit from a module within the Batch TCK, in order to leave the minimum amount of configuration needed to be performed by the implementer. Use standard Maven techniques, e.g. `mvn help:effective-pom` to see the configuration of the module being executed merged with its parent. **NOTE:** Inheriting from these parents is optional, strictly speaking, but it is a good way to ensure a valid configuration for certifying against the TCK is being used.

9. JUnit SE suite - 'se-classpath' runner

We call this suite of tests the "SE" suite just to distinguish from the EE suite. It can be exercised outside of an EE platform.

The 'se-classpath' runner shows an execution of the JUnit runtime tests against the 'jbatch' implementation in SE mode. We walk through its POM configuration below.

9.1. Maven failsafe 'core' execution

First, we look at the 'core' execution, through which the large majority of the runtime tests are executed. It is defined like this in the runner POM:

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-failsafe-plugin</artifactId>
                ...
                <execution>
                    <id>core</id>
                    <goals>
                        <goal>integration-test</goal>
                    </goals>
```

and which includes four aspects, which we use to illustrate required vs. optional aspects of TCK execution.

1. **REQUIRED** - The test includes list must match that defined in `batch-tck-impl-SE-core-suite-includes.txt`, e.g.:

```
    <configuration>
        ...
        <includesFile>${project.build.directory}/test-classes/testprofiles/batch-
tck-impl-SE-core-suite-includes.txt</includesFile>
```

2. **REQUIRED** - The configuration must load the test classes in Maven artifact: `jakarta.batch:com.ibm.jbatch.tck` by a similar, or equivalent mechanism, e.g.:

```
    <configuration>
        ...
        <dependenciesToScan>
            <dependency>jakarta.batch:com.ibm.jbatch.tck</dependency>
        </dependenciesToScan>
```

3. **REQUIRED** - The configuration must NOT include the artifacts packaged in Maven artifact `jakarta.batch:com.ibm.jbatch.tck.appbean` on the test classpath. Since our runner POM includes this as a test-scoped dependency, our execution configuration must exclude this like:

```
    <configuration>
        ...

<classpathDependencyExcludes>jakarta.batch:com.ibm.jbatch.tck.appbean</classpathDep
endencyExcludes>
```

4. **OPTIONAL** - It is up to the user which, if any system properties are passed to the execution, either properties defined by the Batch TCK (e.g. the wait times explained elsewhere in this document) or implementation-specific properties. In the runner we use properties to enable executing with the 'jbatch' implementation:

```
    <configuration>
       ...

<systemPropertiesFile>${project.basedir}/config/tck.exec.properties</systemProperti
esFile>
```

9.2. Maven failsafe 'appbean' execution

Next we look at the 'appbean' execution, through which only a small number of tests are executed. This configuration differs from that of the 'core' execution in that an application JobOperator bean producer is added to the test classpath. It is defined like this in the runner POM:

```
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            ...
            <execution>
                <id>appbean</id>
                <goals>
                    <goal>integration-test</goal>
                </goals>
```

and we step through the same four details:

1. **REQUIRED** - The test includes list must match that defined in `batch-tck-impl-appjoboperator-suite-includes.txt`, e.g.:

```
    <configuration>
        ...
       <includesFile>${project.build.directory}/test-classes/testprofiles/batch-
tck-impl-appjoboperator-suite-includes.txt</includesFile>
```

2. **REQUIRED** - The configuration must load the test classes in Maven artifact: `jakarta.batch:com.ibm.jbatch.tck.appbean` by a similar, or equivalent mechanism, e.g.:

```
    <configuration>
        ...
        <dependenciesToScan>
            <dependency>jakarta.batch:com.ibm.jbatch.tck.appbean</dependency>
        </dependenciesToScan>
```

3. **REQUIRED** - The configuration must include the artifacts packaged in Maven artifact `jakarta.batch:com.ibm.jbatch.tck.appbean` on the test classpath. In the runner we have no special plugin/execution configuration, having used dependency:

```
        <dependency>
            <groupId>jakarta.batch</groupId>
            <artifactId>com.ibm.jbatch.tck.appbean</artifactId>
            <scope>test</scope>
        </dependency>
```

4. **OPTIONAL** - It is up to the user which, if any system properties are passed to the execution, either properties defined by the Batch TCK (e.g. the wait times explained elsewhere in this document) or implementation-specific properties. In the runner we use properties to enable executing with the 'jbatch' implementation:

```
    <configuration>
       ...

<systemPropertiesFile>${project.basedir}/config/tck.exec.properties</systemProperti
esFile>
```

9.3. Alternative Approaches

Note there is no requirement to configure these two executions from a single POM, as the runner does. E.g. two separate POMs could be used. Hopefully the details above provide clear enough guidance for what is and is not required for each execution. As mentioned above it is not required that the user's "runner" module inherit from the parent like this example one does.

9.4. Expected Results

(Here we abstract out the exact numbers to avoid forgetting to update this count and causing ambiguity with the required test count detailed elsewhere).

The 'core' execution:

```
[INFO]
[INFO] Results:
[INFO]
[WARNING] Tests run: MMM, Failures: 0, Errors: 0, Skipped: NN
```

The 'appbean' execution:

```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: N, Failures: 0, Errors: 0, Skipped: 0
```

9.5. In case of Failure

Note: there are many forced failure scenarios tested by the TCK, so typically the log will show a lot of exception stack traces during a normal, successful execution.

If you experienced a failure, it is possible that you experienced a timing issue. The TCK has several built-in properties allowing for tuning of execution to deal with these, and instructions elsewhere in this guide for doing so.

10. JUnit EE suite - 'platform-arquillian' runner

Each of these Arquillian tests run within the runtime's "container", with the help of an Arquillian adapter for that runtime implementation (mentioned as a prerequisite). This Arquillian-based test suite executes the same set of tests within each of an Jakarta Enterprise Beans container and a Servlet container.

The 'platform-arquillian' runner shows how to configure an execution of the JUnit runtime tests implementation in EE mode against some implementation via an Arquillian adapter. This exercises Batch as part of the full Jakarta EE Platform. We walk through its POM configuration below.

10.1. Runtime-specific Maven profile

Without additional configuration the runner POM is not actually configured to run any tests, because the specific runtime with its specific Arquillian adapter is not configured by default.

Instead the runtime-specific pieces are configured within a set of optional profiles, e.g. `-Pglassfish-remote` or `-Pliberty-managed`, to run against GlassFish or Open Liberty, respectively.

To run against a specific runtime, then, you would need to combine something like this runtime-specific profile with the common configuration defined in the runner module (independent of any profile).

10.2. 'jakarta.batch.arquillian.exec-parent' parent module

The runner provides the common setup and configuration needed for any runtime-specific execution via a parent module that can referenced via:

```
    <parent>
        <groupId>jakarta.batch</groupId>
        <artifactId>jakarta.batch.arquillian.exec-parent</artifactId>
        <version>...</version>
    </parent>
```

While it is not required, strictly speaking, to inherit from this parent, it is a convenient way to add all the required elements of the TCK execution while still providing a space for customization to adapt to your specific runtime.

NOTE: Because the parent module is referenced by other modules we don't include the parent module in the TCK distribution, to avoid confusion. An easy, standard Maven technique for merging a given POM with its parent (and ancestors, more generally) is to run the goal: `mvn help:effective-pom` from the current module directory at the command line.

10.3. Jakarta Enterprise Beans vs web executions, core vs appbean executions

A requirement for certifying in EE mode is to execute each test in each of a servlet ('web') and a Jakarta Enterprise Bean context. So to run from a "Jakarta Enterprise Bean context" is to have the test perform the JobOperator API calls used in each TCK test method from within a Jakarta Enterprise Bean (provided by the TCK). This mechanism is established through the Arquillian adapter plus our Arquillian extension module, configured with the help of the parent POM.

We also need to run a set of tests (the 'appbean' tests) with a different application classpath than the 'core' majority of the tests.

Therefore we set up our parent POM to establish four executions of the 'failsafe' plugin: for each of the two classpath configurations ('core' and 'appbean') we have one execution for each of the Jakarta Enterprise Beans and servlet contexts.

10.4. One-time setup used for all failsafe executions

10.4.1. OPTIONAL - use dependency plugin to conveniently unpack dependencies

The test suite files, the test database DDLs, and reporting XSL can be extracted into place by adding the following plugin to your module build (the execution is configured in the 'jakarta.batch.arquillian.exec-parent' module to run during the `pre-integration-test` phase.

```
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
            </plugin>
```

10.4.2. REQUIRED - test database setup - DDL files

The test suite includes tests that use an application database which must be created separately, prior to TCK execution, potentially in an implementation-specific way. The DDL defining the required table format is specified in a set of *.sql files included in the 'com.ibm.jbatch.tck' module dependency. These can most conveniently be accessed by adding the dependency plugin with

parent configuration, as mentioned in the previous section. They can also be accessed in the TCK zip at file: `artifacts/com.ibm.jbatch.tck-<version>.jar`.

10.4.3. NOTE: database setup process may require extra steps for non-Derby databases

During TCK execution, the test application accesses database tables with the 'app' high-level qualifier (or "schema"), e.g. `insert into app.numbers values(?, ?)`. However, this 'app' high-level qualifier does not appear in the *.sql setup DDL files provided by the TCK. These DDL files reference tables without a high-level qualifier.

Since the Apache Derby database uses 'app' as the default high-level qualifier (schema), the runtime access aligns with the setup DDL when Derby is used as the database product.

However, when setting up other databases not using 'app' as the default schema, additional step(s) may be needed. This could involve modifying the setup DDL to set 'app' as the high-level qualifier and/or setting the current schema way when executing the database setup DDL, perhaps using a database-specific tool.

In issue: https://github.com/eclipse-ee4j/batch-tck/issues/51 a challenge was raised because the TCK seemed too dependent on the use of the Derby database.

The challenge was accepted but it was agreed to resolve this for the current TCK release by documenting the workaround (in the documentation section you are reading now) and opening an issue to enhance the TCK in a future release (that enhancement issue is: https://github.com/eclipse-ee4j/batch-tck/issues/55). This future enhancement could either allow the schema to be parameterized or perhaps removed altogether.

10.4.4. NOTE: any database with JDBC driver may be used

There is no requirement to use one of the databases for which a DDL is included; another database with JDBC-compliant driver could potentially be used.

10.4.5. OPTIONAL test results reporting

Finally, the parent POM configures a couple of plugins which, if added to your build in order (first the xml plugin, then echo plugin), can conveniently collect and report test results across the multiple failsafe executions. This output can then be used to report test results for Jakarta TCK certification.

These plugins will transfer the summary from the failsafe plugin and print it to output, using the XSL (zipped up in the 'com.ibm.jbatch.tck' module dependency which can be unpacked with the dependency plugin as mentioned above).

E.g.:

```
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>xml-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <groupId>com.github.ekryd.echo-maven-plugin</groupId>
            <artifactId>echo-maven-plugin</artifactId>
        </plugin>
```

10.5. Maven failsafe core execution: 'core-ejb'

First, we look at the 'core' executions, through which the large majority of the runtime tests are executed. If we look specifically at the 'core-ejb' execution, is defined like this in the parent POM:

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-failsafe-plugin</artifactId>
                ...
                <execution>
                    <id>core-ejb</id>
                    <goals>
                        <goal>integration-test</goal>
                    </goals>
```

and it includes five aspects, which we use to illustrate required vs. optional aspects of TCK execution.

10.5.1. REQUIRED POM elements

1. **REQUIRED** - The test includes list must match that defined in `batch-tck-impl-EE-platform-core-suite-includes.txt`, e.g.:

   ```
       <configuration>
           ...
           <includesFile>${project.build.directory}/test-classes/testprofiles/batch-tck-impl-EE-platform-core-suite-includes.txt</includesFile>
   ```

2. **REQUIRED** - The configuration must load the test classes in Maven artifact: `jakarta.batch:com.ibm.jbatch.tck` by a similar, or equivalent mechanism, e.g.:

```
<configuration>
    ...
    <dependenciesToScan>
        <dependency>jakarta.batch:com.ibm.jbatch.tck</dependency>
    </dependenciesToScan>
```

3. **REQUIRED** - By default there's nothing to do here. But, for the sake of completeness, we note the configuration must NOT include the artifacts packaged in Maven artifact `jakarta.batch:com.ibm.jbatch.tck.appbean` on the test classpath. For the analogous case the corresponding SE test configuration used the classpathDependencyExcludes parameter to ensure the 'appbean' artifact did not appear on the test classpath. Our Arquillian-based test uses custom ShrinkWrap logic to package a test application. By default, we will not package 'appbean'. The exact requirement using this ShrinkWrap logic is that the **arquillian.extensions.jakarta.batch.appbean** system property must NOT equal to `true` (via a case-insensitive match).
4. **REQUIRED** - The system property:
    - The **junit.jupiter.extensions.autodetection.enabled** system property must be set to `true` to allow the TCK's JUnit extension to plugin to JUnit 5.

```
    <configuration>
        ...
        <systemPropertyVariables>

<junit.jupiter.extensions.autodetection.enabled>true</junit.jupiter.extensions.a
utodetection.enabled>
        </systemPropertyVariables>
```

5. **REQUIRED** - The system property:
    - The **jakarta.batch.tck.vehicles.vehicleName** system property must be set to `ejb` to execute within the Jakarta Enterprise Beans context, e.g.:

```
    <configuration>
        ...
        <systemPropertyVariables>

<jakarta.batch.tck.vehicles.vehicleName>ejb</jakarta.batch.tck.vehicles.vehicleN
ame>
        </systemPropertyVariables>
```

10.5.2. OPTIONAL POM properties

As long as the above requirements are met, it is up to the user which, if any, other system properties are passed to the execution, either properties defined by the Batch TCK (e.g. the wait times explained elsewhere in this document) or implementation-specific properties.

Some special, optional properties we mention which are defined by the TCK itself:

1. Specify to exclude some artifacts on the maven test classpath from the Arquillian test deployment with the `arquillian.extensions.jakarta.batch.groupPrefixesToIgnore` system property if they cause problems. Specify prefixes of group names, separated by a column.
2. If the JNDI name of the `EJBVehicleRemote` Jakarta Enterprise Bean is different from the default name, specify the correct name using the `jakarta.batch.tck.vehicles.ejb.jndiName` system property, either in the failsafe maven plugin, or inside the implementation container.

TCK wait times

The TCK wait times for the various tests can be configured via system properties, however note that configuring the failsafe execution which these system properties does not itself guarantee that these properties will be propagated to the underlying Arquillian container. If you want to apply them within the implementation container, you need to apply them to the implementation in a vendor-specific way, before you execute the TCK.

E.g. to apply a custom set of wait time properties in Glassfish you could execute `mvn pre-integration-test`, then take the generated `test.properties` and apply using `asadmin create-system-properties` against a running GlassFish server.

10.6. Maven failsafe appbean execution: 'appbean-ejb'

Next we look at the 'appbean' execution, through which only a small number of tests are executed. This configuration differs from that of the 'core-ejb' execution in that an application JobOperator bean producer is added to the test classpath. For the EE/platform suite, this is accomplished via a custom ShrinkWrap routine defined in our Arquillian extension module.

10.6.1. REQUIRED POM elements

1. **REQUIRED** - The test includes list must match that defined in `batch-tck-impl-appjoboperator-suite-includes.txt`, e.g.:

```
    <configuration>
        ...
       <includesFile>${project.build.directory}/test-classes/testprofiles/batch-
tck-impl-appjoboperator-suite-includes.txt</includesFile>
```

2. **REQUIRED** - The configuration must load the test classes in Maven artifact: `jakarta.batch:com.ibm.jbatch.tck.appbean` by a similar, or equivalent mechanism, e.g.:

```
    <configuration>
        ...
        <dependenciesToScan>
            <dependency>jakarta.batch:com.ibm.jbatch.tck.appbean</dependency>
        </dependenciesToScan>
```

3. **REQUIRED** - The configuration must include the artifacts packaged in Maven artifact `jakarta.batch:com.ibm.jbatch.tck.appbean` on the test classpath. This is accomplished by configured the custom ShrinkWrap logic via setting the **arquillian.extensions.jakarta.batch.appbean** to `true`, e.g.:

```
    <configuration>
        ...
        <systemPropertyVariables>
            ...

<arquillian.extensions.jakarta.batch.appbean>true</arquillian.extensions.jakarta.ba
tch.appbean>
        </systemPropertyVariables>
```

4. **REQUIRED** - The **junit.jupiter.extensions.autodetection.enabled** system property must be set to `true` to allow the TCK's JUnit extension to plugin to JUnit 5, e.g.:

```
    <configuration>
        ...
        <systemPropertyVariables>
            ...

<junit.jupiter.extensions.autodetection.enabled>true</junit.jupiter.extensions.auto
detection.enabled>
        </systemPropertyVariables>
```

5. **REQUIRED** - The system property:
    - The **jakarta.batch.tck.vehicles.vehicleName** system property must be set to `ejb` to execute within the Jakarta Enterprise Bean context, e.g.:

```
    <configuration>
        ...
        <systemPropertyVariables>

<jakarta.batch.tck.vehicles.vehicleName>ejb</jakarta.batch.tck.vehicles.vehicleN
ame>
        </systemPropertyVariables>
```

10.6.2. OPTIONAL POM properties

See the discussion in the earlier section regarding the 'core-ejb' execution, as the same details apply to this execution as well.

10.7. Maven failsafe web executions: 'core-web', 'appbean-web'

The requirements for each of the 'core-web', 'appbean-web' executions are identical to those of the 'core-ejb', 'appbean-ejb' executions, respectively, with a single change:

1. **REQUIRED** - The system property:
    - The **jakarta.batch.tck.vehicles.vehicleName** system property must be set to `web` to execute within a Servlet (web) context, e.g.:

```
    <configuration>
        ...
        <systemPropertyVariables>

<jakarta.batch.tck.vehicles.vehicleName>web</jakarta.batch.tck.vehicles.vehicleN
ame>
        </systemPropertyVariables>
```

So each of these two "web" executions must be run, and all tests contained must pass.

10.8. Alternative Approaches

Note there is no requirement to configure these four executions from a single POM, as the runner does. E.g. one conceivable approach would be to use four separate POMs even, if that were preferred for some reason. Hopefully the details above provide clear enough guidance for what is and is not required for each execution, and if not an issue should be raised with the Jakarta Batch TCK project. As mentioned above it is not required that the user's "runner" module inherit from the parent like this example one does.

10.9. Expected Output

Here is example output when we, as in the sample runner, configure the 'xml' and 'echo' Maven plugins to collect and display the test results.

(Here we abstract out the exact numbers to avoid forgetting to update this count and causing ambiguity with the required test count detailed elsewhere).

```
Jakarta Batch TCK completed running NNN tests.
Number of Tests Passed      = NNN
Number of Tests with Errors = 0
Number of Tests Failed      = 0
Number of Tests Skipped     = XX
```

In case of Failure

Note: there are many forced failure scenarios tested by the TCK, so typically the log will show a lot of exception stack traces during a normal, successful execution.

If you experienced a failure, it is possible that you experienced a timing issue. The TCK has several built-in properties allowing for tuning of execution to deal with these, and instructions elsewhere in this guide for doing so.

11. Signature Tests

The signature tests validate the integrity of the `jakarta.batch` Java "namespace" (or "package prefix") of the batch implementation. This would be especially important for an implementation packaging its own API JAR in which the API must be validated in its entirety. For implementations expecting their users to rely on the API released by the Jakarta Batch specification project (e.g. to Maven Central) the signature tests are also important to validate that improper (non-spec-defined) extensions have not been added to `jakarta.batch.*` packages/classes/etc.

As mentioned in the prerequisite section the signature file formats across the various signature test tools have diverged and this test suite uses the Maven plugin with group:artifact:version coordinates: **org.netbeans.tools:sigtest-maven-plugin:1.6**.

11.1. sigtest runner

The 'sigtest' runner shows an execution of the signature tests against the 'jbatch' implementation, while pulling in its dependencies:

- Jakarta Batch 2.1 API
- Jakarta Inject 2.0 API
- Jakarta CDI 4.0 API

11.2. Java 11 vs Java 17

The TCK provides a distinct signature file depending on whether a Java 11 or a Java 17 JDK is used to certify.

The runner provides a profile-based POM config which is automatically activated based on the Java SE level of the JDK used to run Maven.

11.3. Maven setup - OPTIONAL

In calling the setup "OPTIONAL" we are referring to the fact that an implementer does not need to use any of these exact goals or configuration, though it is probably necessary that some analogous setup work will need to be done before executing the signature tests.

In our example runner, the 'maven-dependency-plugin' is used in the 'pre-integration-test' phase to

setup the later sigtest execution.

The first execution unpacks (copies) the signature files themselves into place in the **target/sigtest-copy** location:

```
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
            <execution>
                <id>unpack-sigfiles</id>
                ...
                <configuration>
                   <outputDirectory>${project.build.directory}/sigtest-
copy</outputDirectory>
```

The second execution unpacks the jbatch impl, including its API dependencies (it uses the default output directory of **target/dependency**):

```
    <execution>
        <id>unpack-classes</id>
```

The 'sigtest-maven-plugin' can also be configured to pick up dependencies as Maven dependencies, building the "test classpath" via Maven. However, we chose to show a more explicit approach, "flattening" the dependency tree by copying everything into a single directory we will execute the tests against

11.4. Maven test execution - REQUIRED

11.4.1. Required JDK level

It is REQUIRED that you run with a JDK of the Java version that you are certifying against. We did not develop the current instructions and requirements with support for the ability to somehow "target" a different Java version (e.g. Java 11) than the version of the JDK using to run the tests via Maven (e.g. Java 17).

In particular note this implies you can NOT use the `<release/>` parameter with the sigtest plugin configuration, e.g. `<release>17</release>`, though it might seem that it is designed for exactly this purpose. (Though this could possibly be an area for future enhancement for someone interested in improving the Batch TCK for future releases).

11.4.2. Required Maven configuration

The execution of the signature tests is accomplished by running the sigtest plugin's `check` goal.

In the runner, this is done during the integration-test phase (after the pre-integration-test setup). The "core" configuration (the part that's independent of profile) in the runner looks like:

```
<plugin>
    <groupId>org.netbeans.tools</groupId>
    <artifactId>sigtest-maven-plugin</artifactId>
    ...
   <executions>
       <execution>
       <id>default-cli</id>
       <phase>integration-test</phase>
       <goals>
           <goal>check</goal>
    ...
    <configuration>
        <action>strictcheck</action>
        <failOnError>true</failOnError>
        <packages>jakarta.batch.**</packages>
        <classes>${project.build.directory}/dependency</classes>
    </configuration>
```

For Java 11, the corresponding Java 11 signature file must be configured, e.g. in the Java 11-activated profile:

```
<plugin>
    <groupId>org.netbeans.tools</groupId>
    <artifactId>sigtest-maven-plugin</artifactId>
    <configuration>
       <sigfile>${project.build.directory}/sigtest-copy/sigtest/sigtest-1.6-
batch.standalone.tck.sig-2.1-se11-OpenJDK-J9</sigfile>
    </configuration>
</plugin>
```

Likewise for Java 17, the corresponding Java 17 signature file must be configured, e.g. in the Java 17-activated profile:

```
<plugin>
    <groupId>org.netbeans.tools</groupId>
    <artifactId>sigtest-maven-plugin</artifactId>
    <configuration>
        <sigfile>${project.build.directory}/sigtest-copy/sigtest/sigtest-1.6-
batch.standalone.tck.sig-2.1-se17-TemurinHotSpot</sigfile>
    </configuration>
</plugin>
```

Note we construct our runner POM like this leveraging the standard Maven behavior for merging profile plugin configuration with non-profile plugin configuration.

11.4.3. Required Maven configuration - summarized

To summarize and say it another way, the **REQUIRED** config consists of these configuration parameter elements with values exactly as shown above in the previous section:

- <action/>
- <failOnError/>
- <packages/>
- <sigfile/> (with value depending on Java 11 vs 17)

and additionally the `<classes/>` parameter **REQUIRED** to be set in such a way that the implementation is tested, (including any packaging of the Jakarta Batch API by the implementation).

11.5. Expected Output

```
$ mvn verify
[INFO] Scanning for projects...
[INFO]
[INFO] -----------< jakarta.batch:com.ibm.jbatch.tck.sigtest.exec >------------

 ...
 ...

[INFO] --- sigtest-maven-plugin:1.6:check (default-cli) @
com.ibm.jbatch.tck.sigtest.exec ---

[INFO] Packages: jakarta.batch.**
[INFO] SignatureTest report
Base version: 2.1.0
Tested version: 2.1.0
Check mode: src [throws normalized]
Constant checking: on

Warning: incorrect classpath parameter: C:\git\jakarta\batch-
tck\com.ibm.jbatch.tck.sigtest.exec\target\classes (C:\git\jakarta\batch-
tck\com.ibm.jbatch.tck.sigtest.exec\target\classes). This directory or jar file will
be ignored!


[INFO] /home/java_user/jkbatch/tckdir/jakarta.batch.official.tck-
2.1.0/runners/sigtest/target/surefire-reports/sigtest/TEST-
com.ibm.jbatch.tck.sigtest.exec-2.1.0.xml: 0 failures in
/home/java_user/jkbatch/tckdir/jakarta.batch.official.tck-
2.1.0/runners/sigtest/target/sigtest-copy/sigtest/sigtest-1.6-
batch.standalone.tck.sig-2.1-se11-OpenJDK-J9
```

NOTE: Also a '0' exit status should be returned from the `mvn verify` command line (shown by `echo`

`$?`).

NOTE: Included in the output above is an expected warning about the "incorrect classpath parameter" **target/classes**. This occurs because our execution project only tests content pulled in as dependencies. It is unfortunate the plugin is this noisy here, and this can be ignored.

NOTE: The `Base version:` in the above output comes from the signature file under test, so should appear exactly as shown. The `Test version:`, on the other hand, comes from the version of the module executing the tests, and so a user not using the runner module could see their own value here.

11.6. Forcing a Signature Test failure (optional)

Though it is not required it can be useful to validate the setup by forcing a failure in the signature test.

E.g. one approach would be to to swap the profile activation definition in the runner POM so that the the 'jdk11' profile gets activated in the presence of Java 17 (instead of Java 11), and vice versa, so that when running with each of Java 11 or 17 the opposite, incorrect signature file will be used.

E.g.:

```
    <profiles>
        <profile>
            <id>jdk11</id>
            <activation>
                <!-- Force failure
                 <jdk>11</jdk>
                 -->
                <jdk>17</jdk>
                ...
        <profile>
            <id>jdk17</id>
            <activation>
                <!-- Force failure
                 <jdk>17</jdk>
                 -->
                <jdk>11</jdk>
```

This should produce output like:

```
$ mvn verify
[INFO] Scanning for projects...
[INFO]
[INFO] -----------< jakarta.batch:com.ibm.jbatch.tck.sigtest.exec >------------
 ...
 ...
[INFO] --- sigtest-maven-plugin:1.6:check (default-cli) @
com.ibm.jbatch.tck.sigtest.exec ---
```

```
[INFO] Packages: jakarta.batch.**
[ERROR] SignatureTest report
Base version: 2.1.0
Tested version: 2.1.0
Check mode: src [throws normalized]
Constant checking: on

Warning: incorrect classpath parameter: C:\git\jakarta\batch-
tck\com.ibm.jbatch.tck.sigtest.exec\target\classes (C:\git\jakarta\batch-
tck\com.ibm.jbatch.tck.sigtest.exec\target\classes). This directory or jar file will
be ignored!

Missing Nested Classes
----------------------

jakarta.batch.api.partition.PartitionReducer$PartitionStatus:                    nested
public final static java.lang.Enum$EnumDesc
jakarta.batch.runtime.BatchStatus:      nested public final static
java.lang.Enum$EnumDesc
jakarta.batch.runtime.Metric$MetricType:nested public final static
java.lang.Enum$EnumDesc

Missing Superclasses or Superinterfaces
---------------------------------------

jakarta.batch.api.partition.PartitionReducer$PartitionStatus:
interface java.lang.constant.Constable
jakarta.batch.runtime.BatchStatus:      interface java.lang.constant.Constable
jakarta.batch.runtime.Metric$MetricType:interface java.lang.constant.Constable

Missing Methods
---------------

jakarta.batch.api.partition.PartitionReducer$PartitionStatus:                    method
public final
java.util.Optional<java.lang.Enum$EnumDesc<jakarta.batch.api.partition.PartitionReduce
r$PartitionStatus>> java.lang.Enum.describeConstable()
jakarta.batch.runtime.BatchStatus:      method public final
java.util.Optional<java.lang.Enum$EnumDesc<jakarta.batch.runtime.BatchStatus>>
java.lang.Enum.describeConstable()
jakarta.batch.runtime.Metric$MetricType:method public final
java.util.Optional<java.lang.Enum$EnumDesc<jakarta.batch.runtime.Metric$MetricType>>
java.lang.Enum.describeConstable()



[INFO] C:\git\jakarta\batch-tck\com.ibm.jbatch.tck.sigtest.exec\target\surefire-
reports\sigtest\TEST-com.ibm.jbatch.tck.sigtest.exec-2.1.0.xml: 1 failures in
C:\git\jakarta\batch-tck\com.ibm.jbatch.tck.sigtest.exec\target\sigtest-
copy\sigtest\sigtest-1.6-batch.standalone.tck.sig-2.1-se17-TemurinHotSpot
[INFO] ------------------------------------------------------------------------
```

```
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
```

and a non-zero exit status should be returned by `mvn verify` (shown by `echo $?`).

11.7. Generating the Signature Files (optional)

Though it is not required for running the TCK, it could perhaps be useful debugging to know how to generate the signature files.

Of course, this is also necessary for producing the TCK in the first place, and updating it after making new API changes or supporting new Java versions.

Simply run:

```
$ mvn sigtest:generate
```

Since the main plugin configuration as well as the profile-based configuration (activated by current JDK level) is added at the plugin-level (rather than the execution-level), no additional configuration is required. The plugin will generate to the same output locations used during the `check` goal, e.g. **target/sigtest-copy/sigtest/sigtest-1.6-batch.standalone.tck.sig-2.1-se11-OpenJDK-J9** (possibly overwriting whatever was there).

12. JUnit test suite in detail (Optional)

12.1. The flow of a typical TCK test

The basic test flow simply involves a JUnit 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).

Several tests intentionally produce failures to test relevant portions of the specification, so a normal execution may cause a number of stack traces, error messages, etc. to stdout.

12.2. Porting Package SPI

The Jakarta Batch TCK relies on an implementation of a "porting package" SPI to function, in order to verify test execution results. The reason is that the Jakarta Batch 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 Jakarta Batch 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.

12.3. Adjusting the Default Timeout Value

The JobOperatorBridge is a utility/helper class in the Jakarta Batch TCK which makes use of the following system property:

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

using a default value of `900000` (900 seconds).

This prevents tests from "hanging" indefinitely if something catastrophic occurs causing the job to never complete (or if the porting package SPI "waiter" is never notified for some reason).

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 default value less than that applied to all tests would cause failures simply due to timing (and not because of any failure in the underlying Jakarta Batch implementation).

The value of 900 seconds was chosen, then, to avoid falsely reporting an error because of timing out too soon, allowing plenty of time for a test to finish executing, even on slower hardware, and leaves some time to attach a debugger.

It does not, however, provide "fast failure" in case of a hang or runaway thread.

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

12.4. Default Test-Specific Wait Times, and How to Adjust Timeout Values

Some of the TCK tests sleep for a short period of time to allow an operation to complete or to force a timeout.

These wait times are defaulted via properties that are also specified in the TCK source repo at path: `com.ibm.jbatch.tck/src/main/resources/tck.default.sleep.time.properties`. (Note this exact file may not be used in the sample 'runner' modules provided with the TCK).

As with many typical decisions regarding timeout values, we attempt to strike a good balance between failing quickly when appropriate but allowing legitimate work to complete.

These values can be adjusted if timing issues are seen in the implementation being tested. Refer to the comments in the test source for a specific test to better understand how the time value is used for that test.

12.5. Working with TCK source (debugging, etc.)

For most development/debug use cases it is recommended to refer to the source in the Jakarta Batch TCK] GitHub repository [https://github.com/eclipse-ee2j/batch-tck], using this documentation, and GitHub tags/releases, etc. to match the official level tested in the TCK distribution.

Note too 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).

12.6. Arquillian / EE Platform Tests

The EE Platform version of the TCK uses Arquillian to run the JUnit 5 tests. It works by:

1. Using the standard Arquillian dependencies as for JUnit4, except the JUnit4 runner
2. Using the official Arquillian JUnit5 extension and enables it globally via a service loader file
3. Enabling the Arquillian JUnit5 extension by setting `junit.jupiter.extensions.autodetection.enabled` system property to true in pom.xml
4. Using an Arquillian extension specific for Batch TCK to create a deployment for each test

The Batch TCK Arquillian extension is implemented in its own module within the Batch TCK. It contains :

- The Arquillian extension class
- A service loader file to register the extension with Arquillian
- A service loader file to register the Arquillian JUnit 5 extension with JUnit 5 (because it's not included in the extension module)

12.7. Multiple Maven Executions for Multiple Classpath Configurations

One specification detail in particular significantly complicates the TCK. To validate the rules described in **Section 10.4 JobOperator** and subsections, the TCK needs to test both the cases in which the application does and does not provide a JobOperator CDI Bean. This requires multiple "classpath" variations (with the TCK itself playing the role of "application" here). Because of this we require multiple test executions to be configured in Maven, and the full JUnit portion of the TCK will consist of both of these executions, both of which must be executed successfully in order to pass the TCK and claim compliance.

13. TCK Challenges/Appeals Process

The Jakarta EE TCK Process 1.1 will govern all process details used for challenges to the Jakarta Batch TCK.

Except from the **Jakarta EE TCK Process 1.1**:

> Specifications are the sole source of truth and considered overruling to the TCK in all senses. In the course of implementing a specification and attempting to pass the TCK, implementations may come to the conclusion that one or more tests or assertions do not conform to the specification, and therefore MUST be excluded from the certification requirements.

> Requests for tests to be excluded are referred to as Challenges. This section identifies who can make challenges to the TCK, what challenges to the TCK may be submitted, how these challenges are submitted, how and to whom challenges are addressed.

13.1. Filing a Challenge

The challenge process is defined within the Challenges section within the **Jakarta EE TCK Process 1.1**.

Challenges will be tracked via the issues of the Jakarta Batch Specification repository.

As a shortcut through the challenge process mentioned in the **Jakarta EE TCK Process 1.1** you can click here, though it is recommended that you read through the challenge process to understand it in detail.

14. Certification of Compatibility

The Jakarta EE TCK Process 1.1 will define the core process details used to certify compatibility with the Jakarta Batch specification, through execution of the Jakarta Batch TCK.

Except from the **Jakarta EE TCK Process 1.1**:

> Jakarta EE is a self-certification ecosystem. If you wish to have your implementation listed on the official https://jakarta.ee implementations page for the given specification, a certification request as defined in this section is required.

14.1. Filing a Certification Request

The certification of compatibility process is defined within the Certification of Compatibility section within the **Jakarta EE TCK Process 1.1**.

Certifications will be tracked via the issues of the Jakarta Batch Specification repository.

As a shortcut through the certification of compatibility process mentioned in the **Jakarta EE TCK Process 1.1** you can click here, though it is recommended that you read through the certification process to understand it in detail.

15. TCK SPI "Porting Package" in-depth (optional)

Most users should be able to skip this section. They will be able to rely on having the TCK do "polling" for job completion. In case an alternate solution is required, the following details are included.

The two porting package SPI classes in the Jakarta Batch TCK are:

- **com.ibm.jbatch.tck.spi.JobExecutionWaiter**
- **com.ibm.jbatch.tck.spi.JobExecutionWaiterFactory**

The default implementations of these provided by the Jakarta Batch TCK are, respectively:

- **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, using 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, in order to replace the default implementation.

16. Links

- Jakarta Batch TCK repository - https://github.com/eclipse-ee4j/batch-tck
- Jakarta Batch specification/API repository - https://github.com/eclipse-ee4j/batch-api
- Jakarta Batch project home page - https://projects.eclipse.org/projects/ee4j.jakartabatch
- In case there is some detail in the previous JSR 352 TCK reference guide not ported that could possibly be helpful, here is the former JSR 352 TCK reference guide.
- The original JSR 352 page: JSR 352: Batch Applications for the Java Platform).
- Arquillian and ShrinkWrap doc: https://arquillian.org/guides/shrinkwrap_introduction/

17. Change History

17.1. Initial Release - Jakarta Batch 1.0

- July 17, 2019

17.2. Update - Jakarta Batch 2.0

- July 30, 2020

17.3. Update (and major rework moving from Ant→Maven) - Jakarta Batch 2.1

- January 30, 2022