Jakarta Data TCK Reference Guide

Table of Contents

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 Data specification.

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

1.1. Licensing

The Jakarta Data TCK is provided under the **Eclipse Foundation Technology Compatibility Kit License**.

1.2. Who Should Use This Guide

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

- Implementer of Jakarta Data.

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

The term "EE mode" when talking about the Jakarta Data TCK distinguishes the requirements specifically for implementers running the TCK to certify against a Jakarta EE platform. In contrast, the term "SE mode" distinguishes the requirements specifically for implementers running the TCK against a standalone implementation of the Jakarta Data specification.

The Jakarta Data specification has a subset of tests that can run in "SE mode" without the requirement of running against a Jakarta EE Platform.

1.4. Terminology - "Standalone TCK"

The community will sometimes refer to this TCK as the "standalone" Data TCK. This usage comes from the fact that Jakarta Data is part of the Jakarta EE Platform, which has a platform-level TCK from which we are distinguishing this "standalone" TCK.

This terminology is confusing, since the term "standalone" is overloaded to also mean that this TCK can be run in SE Mode which it can. A better term would be `Specification TCK`, but that terminology is not yet being used.

1.5. Terminology - "Test Client" vs "Test Server"

When running in EE mode the Jakarta Data TCK acts as a `Test Client` that will install test applications onto a Jakarta EE Platform Server. The Platform Server will act as a `Test Server` and

run tests based on incoming requests from the `Test Client`. Assertions will occur both on the client and server sides.

1.6. Terminology - "Core Profile" vs "Web Profile" vs "Platform"

The Jakarta EE working group defines sets of specifications that create the core profile, web profile, and platform. The Jakarta Data TCK can run against each of these profiles since Jakarta Data is a Core Profile specification.

References:

- Core profile: https://jakarta.ee/specifications/coreprofile/
- Web profile: https://jakarta.ee/specifications/webprofile/
- Platform: https://jakarta.ee/specifications/platform/

1.7. Before You Read This Guide

Before reading this guide, you should familiarize yourself with Jakarta Data 1.0.0 specification, which can be found at https://jakarta.ee/specifications/data/.

Other useful information and links can be found on the eclipse.org project home page for the Jakarta Data project and also at the GitHub repository home for the specification project.

2. Major TCK Changes

This is the first release of the Jakarta Data TCK.

3. Certify Compatibility

3.1. Runtime Tests and Signature Tests Required

To certify compatibility with the entire Jakarta EE Platform (including Jakarta Data), you will need to run the TCK against your implementation and pass 100% of both the:

- JUnit5 runtime tests
- Signature tests

The two types of tests are encapsulated in a single execution or configuration. This means that the Signature tests will run alongside all other tests and no additional execution or configuration is required.

By "runtime" tests we simply mean tests simulating Jakarta Data applications running against the Data implementation either in SE Mode (for Jakarta Data providers) or EE Mode (For Jakarta EE product providers). These tests verify that the Data applications behave according to the details

defined in the specification, as validated by the TCK test logic.

3.2. Java SE level - Java 17 or Java 21

The Java SE version is important to note, and this version must be used consistently throughout both the JUnit5 runtime and Signature tests for a given certification request.

For the current TCK version, this can be done with either Java SE Version 17 or Version 21.

4. Prerequisites

4.1. Software To Install

1. **Java/JDK** - Install the JDK you intend to use for this certification request (Java SE Version 17 or Version 21).
2. **Maven** - Install Apache Maven 3.6.0 or higher.
3. **Jakarta EE Platform** (if running in EE Mode) - Jakarta EE Application Server or Container [Glassfish, Open Liberty, JBoss, WebLogic, etc.]

4.2. Testing Framework

To better understand how this TCK works, knowing what testing frameworks are being utilized is helpful. Knowledge of how these frameworks operate and interact will help during the project setup.

1. **Arquillian** - Version 1.7.0.Alpha13 or later - The Jakarta Data TCK can run in EE Mode and it uses Arquillian to execute tests within an Arquillian "container" for certifying against an EE Platform. You must configure an Arquillian adapter for your target runtime.
2. **JUnit5** - Version 5.9.0 or later - The Jakarta Data TCK uses JUnit5 as the entry-point for tests and deployments using Arquillian.
3. **Signature Test Plugin** - Version 2.3 exactly - The Jakarta Data TCK uses the Signature Test Plugin to verify API signatures used by an implementation and those release by the specification match.

   No action is needed here, but we note that the signature files were built and should be validated with the plugin with group:artifact:version coordinates: **jakarta.tck:sigtest-maven-plugin:2.3**, as used by the sample runner included in the TCK zip. This is a more specific direction than in earlier releases of the platform TCK, 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 an exact version is required.

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 Data TCK is distributed as a zip file, which contains all the files necessary to use, run, and certify your implementation. You can access the current source code from the Git repository.

5.2. The TCK Environment

The Jakarta Data TCK can simply be extracted from the ZIP file. Once the TCK is extracted, you'll see the following structure:

```
data-tck-<version>-dist/
  artifacts/
  docs/
  starter/
  LICENSE
  README.md
```

In more detail:

- `artifacts` contains all the test artifacts pertaining to the TCK
  - The TCK test classes and source
  - The JUnit5 configuration file
  - A copy of the SignatureTest file for reference
  - A script to copy the TCK into local maven repository.
- `docs` contains the documentation for the TCK (i.e. this reference guide)
- `starter` a very basic starter maven project to get you started.

5.3. A Quick Tour of the TCK Artifacts

5.3.1. What is included

The Jakarta Data TCK is a test library that includes four types of packages:

- `ee.jakarta.tck.data.standalone.*` these are basic API tests, or SPI tests that can run in SE Mode or EE mode.
- `ee.jakarta.tck.data.core.*` these are more complex integration tests that must run against at least Core Profile.
- `ee.jakarta.tck.data.web.*` these are more complex integration tests that must run against at least Web Profile.
- `ee.jakarta.tck.data.platform.*` these are more complex integration tests that must run against at least Platform.
- `ee.jakarta.tck.data.framework.*` these are utility packages that help support the development and execution of the TCK.

Tests that exist at a lower level will run on any level above, for example, all core profile tests will run against web profile. Signature tests exist at the standalone level which means they will run in any mode, and any profile.

API Signature Files

One signature file exists per Java version 17 and 21:

1. `artifacts/jakarta.data.sig_17`
2. `artifacts/jakarta.data.sig_21`

This signature file is for reference only. A copy of the signature file is included in the Jakarta Data TCK test jar.

5.3.2. What is not included

The Jakarta Data TCK uses but does not provide the necessary application servers, test frameworks, APIs, SPIs, or implementations required to run. It is up to the tester to include those dependencies and set up a test project to run the TCK.

Here is an essential checklist of what you will need per mode, and links to the section that describe how to satisfy these requirements:

SE Mode:

- The Data API, JUnit5, and Signature Test Plugin libraries available at runtime | Standalone Dependencies
- A dependency injection framework supported by your implementation | Standalone Dependencies
- A logging configuration for TCK logging available at runtime | Standalone Logging

EE Mode:

- An Application Server to test against | Software To Install
- The Jakarta Data API, Arquillian SPI, and JUnit5 libraries available to the `Test Client` | Test Client Dependencies
- An Arquillian SPI implementation for your Application Server | Configure Arquillian
- A logging configuration for TCK logging on `Test Client` and `Test Server` | Configure Client and Server Logging

6. TCK Test Requirements

There is flexibility regarding how a user could use Maven to configure a TCK execution. Therefore, 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.

6.1. Expected Output

For the JUnit5 runtime tests of the TCK, the following table shows the number of tests that should pass based on platform and entity:

entity type	standalone	core	web	platform	skipped
persistence	84	86	99	99	0
nosql	75	77	88	88	0

Note: Counts include signature test, but do not include disabled tests.

7. Set up a TCK runner project

A simple maven project is required to control the lifecycle of the Jakarta Data TCK. Sample pom.xmls has been provided in this distribution under the /starter/ directory.

- se-pom.xml - is used for standalone testing.
- ee-pom.xml - is used for Jakarta Platform testing.

7.1. SE Mode

7.1.1. Standalone Dependencies

The runtime will need to be configured with the dependencies necessary to run the TCK.

Example se-pom.xml:

```
<!-- The Junit5 test frameworks -->
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>org.junit</groupId>
   <artifactId>junit-bom</artifactId>
   <version>${junit.version}</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>

<!-- Runtime Dependencies -->
<dependencies>
 <!-- The TCK -->
 <dependency>
  <groupId>jakarta.data</groupId>
  <artifactId>jakarta.data-tck</artifactId>
  <version>${jakarta.data.version}</version>
```

```
  </dependency>
  <!-- The API -->
  <dependency>
   <groupId>jakarta.data</groupId>
   <artifactId>jakarta.data-api</artifactId>
   <version>${jakarta.data.version}</version>
  </dependency>
  <!-- TODO add your implementation of the Jakarta Data API -->
  <!-- Junit5 -->
  <dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter</artifactId>
  </dependency>
  <!-- Signature Test Plugin -->
  <dependency>
   <groupId>jakarta.tck</groupId>
   <artifactId>sigtest-maven-plugin</artifactId>
   <version>${sigtest.version}</version>
  </dependency>
  <!-- APIs referenced by TCK that do not require implementations for standalone tests
-->
  <dependency>
   <groupId>org.jboss.shrinkwrap</groupId>
   <artifactId>shrinkwrap-api</artifactId>
   <version>${shrinkwrap.version}</version>
  </dependency>
  <dependency>
   <groupId>org.jboss.arquillian.junit5</groupId>
   <artifactId>arquillian-junit5-core</artifactId>
   <version>${arquillian.version}</version>
  </dependency>
  <dependency>
   <groupId>jakarta.servlet</groupId>
   <artifactId>jakarta.servlet-api</artifactId>
   <version>${jakarta.servlet.version}</version>
  </dependency>
 <!-- APIs referenced by TCK that do require implementations for standalone tests -->
  <dependency>
   <groupId>jakarta.enterprise</groupId>
   <artifactId>jakarta.enterprise.cdi-api</artifactId>
   <version>${jakarta.enterprise.cdi.version}</version>
  </dependency>
  <!-- TODO add a DI framework implementation that is supported by your Jakarta Data
implementation -->
 </dependencies>
```

7.1.2. Configure JUnit5

JUnit5 needs to be configured to know which packages contain tests to run. This test discovery is done automatically by configuring a `dependenciesToScan` element.

In order for your maven project to execute these tests the surefire plugin needs to be configured.

Example se-pom.xml:

```
<!-- Surefire plugin - Entrypoint for Junit5 -->
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-surefire-plugin</artifactId>
 <version>${maven.surefire.plugin.version}</version>
 <configuration>
  <dependenciesToScan>
   <dependency>jakarta.data:jakarta.data-tck</dependency>
  </dependenciesToScan>
  <systemPropertyVariables>
   <!-- Required, Arquillian will deploy artifacts by default -->
   <jakarta.tck.skip.deployment>true</jakarta.tck.skip.deployment>
  </systemPropertyVariables>
  <!-- Supported tags
      Profiles: [standalone]
      Entities:[nosql|persistence]
      Other:   [signature] -->
  <groups>${includedTests}</groups>
  <!-- Ensure surfire plugin looks under src/main/java
   instead of src/test/java -->
  <testSourceDirectory>${basedir}/src/main/java/</testSourceDirectory>
 </configuration>
</plugin>
```

7.1.3. Standalone Mode

All test classes that support running in standalone mode are annotated with the `@Standalone` annotation.

When running in standalone mode, include the group `<groups>standalone<groups>`. This will ensure that none of the core/web/platform profile tests are run.

When running in standalone mode, include the system property `<jakarta.tck.skip.deployment>true</jakarta.tck.skip.deployment>`. This will ensure that no Arquillian deployments are created and that all tests are run on the client JVM.

7.1.4. Entity Modes

All test classes in the TCK are annotated with `@NoSQL`, `@Persistence`, or `@AnyEntity` annotations.

If your Jakarta Data implementation supports NoSQL entities, include the group `<groups>nosql<groups>`.

If your Jakarta Data implementation supports Persistence entities, include the group `<groups>persistence<groups>`.

7.1.5. Filtering Tests

As mentioned in the prior sections tests can be filtered by modifying the `<groups>` element. It is recommended to avoid using `<excludedGroups>` as the surefire plugin has a known issue where excluded groups do not have precedence over included groups.

Therefore, if you wanted to run standalone tests with a NoSQL entities you would configure `<groups><![CDATA[standalone & nosql]]><groups>`.

7.1.6. Standalone Logging

The Jakarta Data TCK uses `java.util.logging` for logging debug messages, and to output test results in some cases. This is done by pointing the JVM to the logging configuration file using the property. An example logging configuration file has been provided under the `/starter` directory.

To enable logging for the Client side of tests, add a system property to the surefire plugin:

Example se-pom.xml:

```
    <systemPropertyVariables>
     <java.util.logging.config.file>${logging.config}</java.util.logging.config.file>
    </systemPropertyVariables>
```

7.2. EE Mode

7.2.1. Test Client Dependencies

The entry point to running the TCK will be on the client-side using JUnit5. The Test Client will need to be configured with the dependencies necessary to run the TCK. Some of these dependencies will depend on the application server you are using, and comments have been added to this sample describing the customization necessary.

Example ee-pom.xml:

```
<!-- The Arquillian and Junit5 test frameworks -->
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>org.jboss.arquillian</groupId>
   <artifactId>arquillian-bom</artifactId>
   <version>${arquillian.version}</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
  <dependency>
   <groupId>org.junit</groupId>
   <artifactId>junit-bom</artifactId>
   <version>${junit.version}</version>
```

```
    <type>pom</type>
    <scope>import</scope>
   </dependency>
  </dependencies>
</dependencyManagement>

<!-- Client Dependencies -->
<dependencies>
 <!-- The TCK -->
 <dependency>
  <groupId>jakarta.data</groupId>
  <artifactId>jakarta.data-tck</artifactId>
  <version>${jakarta.data.version}</version>
 </dependency>
 <!-- The API -->
 <dependency>
  <groupId>jakarta.data</groupId>
  <artifactId>jakarta.data-api</artifactId>
  <version>${jakarta.data.version}</version>
 </dependency>
 <!-- Arquillian Implementation for JUnit5 -->
 <dependency>
  <groupId>org.jboss.arquillian.junit5</groupId>
  <artifactId>arquillian-junit5-container</artifactId>
 </dependency>
 <!-- TODO add Arquillian SPI impl for your Jakarta EE Platform Server -->
 <!-- Junit5 -->
 <dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
 </dependency>
 <!-- Signature Test Plugin -->
 <dependency>
  <groupId>jakarta.tck</groupId>
  <artifactId>sigtest-maven-plugin</artifactId>
  <version>${sigtest.version}</version>
 </dependency>
 <!-- APIs provided by your Jakarta EE Platform server -->
 <dependency>
  <groupId>jakarta.servlet</groupId>
  <artifactId>jakarta.servlet-api</artifactId>
  <version>${jakarta.servlet.version}</version>
 </dependency>
 <dependency>
  <groupId>jakarta.enterprise</groupId>
  <artifactId>jakarta.enterprise.cdi-api</artifactId>
  <version>${jakarta.enterprise.cdi.version}</version>
 </dependency>
</dependencies>
```

Each of these Arquillian tests run within the runtime container, with the help of an Arquillian adapter for that runtime implementation (mentioned as a prerequisite).

7.2.2. Configure JUnit5

JUnit5 needs to be configured to know which packages contain tests to run. This test discovery is done automatically by configuring a `dependenciesToScan` element.

In order for your maven project to execute these tests the surefire plugin needs to be configured.

Example ee-pom.xml:

```
   <!-- Surefire plugin - Entrypoint for Junit5 -->
   <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${maven.surefire.plugin.version}</version>
    <configuration>
     <dependenciesToScan>
      <dependency>jakarta.data:jakarta.data-tck</dependency>
     </dependenciesToScan>
     <systemPropertyVariables>
      <jimage.dir>${jimage.dir}</jimage.dir>
      <signature.sigTestClasspath>[path-to]/jakarta.data-
api.jar:${jimage.dir}/java.base:${jimage.dir}/java.rmi:${jimage.dir}/java.sql:${jimage
.dir}/java.naming</signature.sigTestClasspath>
      <!-- Optional, Arquillian will deploy artifacts by default -->
      <jakarta.tck.skip.deployment>false</jakarta.tck.skip.deployment>
     </systemPropertyVariables>
     <!-- Supported tags
         Profiles:[core|web|platform]
         Entities:[nosql|persistence]
         Other:   [signature] -->
     <groups>${includedTests}</groups>
     <!-- Ensure surfire plugin looks under src/main/java
     instead of src/test/java -->
     <testSourceDirectory>${basedir}/src/main/java/</testSourceDirectory>
    </configuration>
   </plugin>
```

7.2.3. Profile Mode

All test classes that require running on a Jakarta profile are annotated with the `@Core`, `@Web`, or `@Platform` annotation.

When running against a Jakarta profile, include the group that matches your supported profile. For example: `<groups>core<groups>`.

When running against a Jakarta profile, optionally include the system property

`<jakarta.tck.skip.deployment>false</jakarta.tck.skip.deployment>`. By default Arquillian will deploy test artifacts to a container.

7.2.4. Entity Modes

All test classes in the TCK are annotated with `@NoSQL`, `@Persistence`, or `@AnyEntity` annotations.

If your Jakarta Data implementation supports NoSQL entities, include the group `<groups>nosql<groups>`.

If your Jakarta Data implementation supports Persistence entities, include the group `<groups>persistence<groups>`.

7.2.5. Filtering Tests

As mentioned in the prior sections tests can be filtered by modifying the `<groups>` element. It is recommended to avoid using `<excludedGroups>` as the surefire plugin has a known issue where excluded groups do not have precedence over included groups.

Therefore, if you wanted to run core profile tests with a NoSQL entities you would configure `<groups><![CDATA[core & nosql]]><groups>`.

7.2.6. Configure Arquillian

Application Servers that implement the Arquillian SPI use a configuration file to define properties, such as hostname, port, username, password, etc. These properties will allow Arquillian to connect to the application server, install applications, and get test responses. An example Arquillian configuration file has been provided in the `starter/` directory.

It is possible to configure the surefire plugin to past variables to the Arquillian container:

```
<systemPropertyVariables>
 <tck_server>[TODO]</tck_server>
 <tck_hostname>[TODO]</tck_hostname>
 <tck_username>[TODO]</tck_username>
 <tck_password>[TODO]</tck_password>
 <tck_port>[TODO]</tck_port>
 <tck_port>[TODO]</tck_port>
</systemPropertyVariables>
```

7.2.7. Configure Jakarta EE Platform Server

The Jakarta Data TCK requires that your Jakarta EE Platform Server has a valid implementation of the Jakarta Data API.

7.2.8. Configure Client and Server Logging

The Jakarta Data TCK uses `java.util.logging` for logging debug messages, and to output test results in some cases. Registered loggers exist both on the Test Client and Test Server meaning you will

need to configure both sides to enable logging. This is done by pointing the JVM to the logging configuration file using the property. An example logging configuration file has been provided under the `/starter` directory.

To enable logging for the Client side of tests, add a system property to the surefire plugin:

Example ee-pom.xml:

```
     <systemPropertyVariables>
      <java.util.logging.config.file>${logging.config}</java.util.logging.config.file>
     </systemPropertyVariables>
```

To enable logging for the Server side of tests, set the same system property on the JVM running your application server.

7.2.9. Advanced Arquillian Configuration

Some application servers may have custom deployment descriptors that they would like to include as part of the applications that are being deployed to their server. The custom deployment descriptors can be included in a programmatic way using ShrinkWrap and the Arquillian SPI.

Example ApplicationArchiveProcessor:

```
public class MyApplicationArchiveProcessor implements ApplicationArchiveProcessor {

    //List of test classes that deploy application that you need to customize
    List<String> testClasses;

    @Override
    public void process(Archive<?> archive, TestClass testClass) {
        if(testClasses.contains(testClass.getClass().getCanonicalName())){
            ((WebArchive) archive).addAsWebInfResource("my-custom-sun-web.xml", "sun-
web.xml");
        }
    }
}
```

Example LoadableExtension:

```
public class MyLoadableExtension implements LoadableExtension {
    @Override
    public void register(ExtensionBuilder extensionBuilder) {
        extensionBuilder.service(ApplicationArchiveProcessor.class,
MyApplicationArchiveProcessor.class);
    }
}
```

Example META-INF/services/org.jboss.arquillian.core.spi.LoadableExtension:

```
ee.jakarta.tck.data.example.extension.MyLoadableExtension
```

7.3. Test property reference

In the previous sections, there were sample configurations that contained information about the different system properties that can be set under the `<systemPropertyVariables>` of the Surefire plugin.

This section contains a complete list of the system properties that will be looked up by the TCK and a short description of what data each represents.

Use this reference as a quick guide for customizing how this TCK is run for your implementation.

Key	Required	Description
java.home	true	Path to the java executable used to create the current JVM
java.specification.version	true	Specification version of the java executable
java.io.tmpdir	true	The path to a temporary directory where a copy of the signature file will be created
java.version	true	Full version of the java executable
jakarta.tck.skip.deployment	false	If true, run in SE mode and do not use Arquillian deployment, if false run in EE mode and use Arquillian deployments. Default: false
jakarta.tck.poll.frequency	false	Time in seconds between polls of the repository to verify read-only data was successfully written. Default: 1 second
jakarta.tck.poll.timeout	false	Time in seconds when we will stop polling to verify read-only data was successfully written. Default: 60 seconds
jakarta.tck.consistency.delay	false	Time in seconds after verifying read-only data was successfully written to respository for repository to have consistency. Default: none

Key	Required	Description
jakarta.tck.database.type	false	The type of database being used. Valid values are [OTHER, RELATIONAL, GRAPH, DOCUMENT, TIME_SERIES, COLUMN, KEY_VALUE] (case insensitive). The database type is used to make assertions based on the keywords supported by the underlying database. Default: OTHER
jakarta.tck.database.name	false	The name of database being used. The database name is used to make assertions based on the underlying database. Default: none
signature.sigTestClasspath	false	The path to the Jakarta Data API JAR used by your implementation. Required for standalone testing, but optional when testing on a Jakarta EE profile. Default: none
jimage.dir	true	The path to a directory that is readable and writable that the signature test will cache Java SE modules as classes. Default: none

Note: All non-java properties set on the test client, will be exported to the test server so there is no need to set the same properties on both.

8. Example runners

This section is dedicated to listing example runners for other implementations to use as a reference on how to configure and use the Jakarta Data TCK.

Below are links to projects where the Jakarta Data TCK is being used and run successfully:

Project	Link	Profile(s)
Hibernate	hibernate-data-tck	standalone
Open Liberty	open-liberty/io.openliberty.jakarta.data.1.0_fat_tck	core, web, full

9. Running the TCK

Once the TCK Runner project is created and configured the Jakarta Data TCK is run as part of the maven test lifecycle.

```
$ cd starter
$ mvn clean test
```

9.1. Expected Output

Here is example output when successfully running (platform + persistence entity) the starter project:

```
$ mvn clean test
...
[INFO] --- maven-surefire-plugin:3.0.0-M7:test (default-test) @ tck.runner ---
[INFO] Using auto detected provider
org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running ee.jakarta.tck.data.core.cdi.CDITests
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0,Time elapsed: y.yy s - in
ee.jakarta.tck.data.core.cdi.CDITests
[INFO]
[INFO] Running ee.jakarta.tck.data.standalone.entity.EntityTests
[INFO] Tests run: 73, Failures: 0, Errors: 0, Skipped: 0,Time elapsed: y.yy s - in
ee.jakarta.tck.data.standalone.entity.EntityTests
[INFO]
[INFO] Running ee.jakarta.tck.data.standalone.persistence.PersistenceEntityTests
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0,Time elapsed: y.yy s - in
ee.jakarta.tck.data.standalone.persistence.PersistenceEntityTests
[INFO]
[INFO] Running ee.jakarta.tck.data.standalone.signature.SignatureTests
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,Time elapsed: y.yy s - in
ee.jakarta.tck.data.standalone.signature.SignatureTests
[INFO]
[INFO] Running ee.jakarta.tck.data.web.example.ComplexServletTests
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0,Time elapsed: y.yy s - in
ee.jakarta.tck.data.web.example.ComplexServletTests
[INFO]
[INFO] Running ee.jakarta.tck.data.web.transaction.PersistenceTests
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0,Time elapsed: y.yy s - in
ee.jakarta.tck.data.web.transaction.PersistenceTests
[INFO]
[INFO] Running ee.jakarta.tck.data.web.validation.ValidationTests
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0,Time elapsed: y.yy s - in
```

```
ee.jakarta.tck.data.web.validation.ValidationTests
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 99, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------
[INFO] Total time:  xx.xxx s
[INFO] Finished at: yyyy-mm-ddThh:mm:ss.mmmm
[INFO] ------------------------------------------------------
```

10. Signature Tests

The signature tests validate the integrity of the `jakarta.data` Java "namespace" (or "package prefix") of the Data 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 Data 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.data.\*` packages/classes/etc.

For more information about generating the signature test file, and how the signature test runs read: ee.jakarta.tck.concurrent.framework.signaturetest/README.md

10.1. Running signature tests

The Jakarta Data TCK will run signature tests within a standalone or deployed application, and not as part of a separate plugin / execution. This means that the signature tests will run during the maven `test` phase.

Your test client is configured with the property `jimage.dir=${project.build.directory}/jimage/`. This is because, when running the signature tests on JDK 9+ we need to convert the JDK modules back into class files for signature testing.

The signature test plugin we use will also attempt to perform reflective access of classes, methods, and fields. Due to the new module system in JDK 9+ special permissions need to be added in order for these tests to run:

10.1.1. with Modules

By default the java.base module only exposes certain classes for reflective access. Therefore, the Data TCK test will need access to the `jdk.internal.vm.annotation` class. To give the `sigtest-maven-plugin` access to this class set the following JVM properties:

```
--add-exports java.base/jdk.internal.vm.annotation=ALL-UNNAMED
--add-opens java.base/jdk.internal.vm.annotation=ALL-UNNAMED
```

Some JDKs will mistake the space in the prior JVM properties as delimiters between properties In this case use:

```
--add-exports=java.base/jdk.internal.vm.annotation=ALL-UNNAMED
--add-opens=java.base/jdk.internal.vm.annotation=ALL-UNNAMED
```

10.2. Expected output

Here is example output when successfully running the signature tests:

```
**************************************************
All package signatures passed.
    Passed packages listed below:
        jakarta.data(static mode)
        jakarta.data(reflection mode)
        jakarta.data.exceptions(static mode)
        jakarta.data.exceptions(reflection mode)
        jakarta.data.metamodel(static mode)
        jakarta.data.metamodel(reflection mode)
        jakarta.data.metamodel.impl(static mode)
        jakarta.data.metamodel.impl(reflection mode)
        jakarta.data.page(static mode)
        jakarta.data.page(reflection mode)
        jakarta.data.page.impl(static mode)
        jakarta.data.page.impl(reflection mode)
        jakarta.data.repository(static mode)
        jakarta.data.repository(reflection mode)
        jakarta.data.spi(static mode)
        jakarta.data.spi(reflection mode)

**************************************************
```

11. TCK Challenges/Appeals Process

The Jakarta EE TCK Process 1.4.1 will govern all process details used for challenges to the Jakarta Data TCK.

Except from the **Jakarta EE TCK Process 1.4.1**:

> 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.

11.1. Filing a Challenge

The certification of compatibility process is defined within the Challenges section within the **Jakarta EE TCK Process 1.4.1**.

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

As a shortcut through the challenge process mentioned in the **Jakarta EE TCK Process 1.4.1** you can click here, though it is recommended that you read through the challenge process to understand it in detail.

11.2. Successful Challenges

The following tests are exempt from TCK testing due to challenges:

Class	Method	Reason

12. Certification of Compatibility

The Jakarta EE TCK Process 1.4.1 will define the core process details used to certify compatibility with the Jakarta Data specification, through execution of the Jakarta Data TCK.

Except from the **Jakarta EE TCK Process 1.4.1**:

> 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.

12.1. Filing a Certification Request

The certification of compatibility process is defined within the Certification of Compatibility section within the **Jakarta EE TCK Process 1.4.1**.

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

As a shortcut through the certification of compatibility process mentioned in the **Jakarta EE TCK Process 1.4.1** you can click here, though it is recommended that you read through the certification process to understand it in detail.

13. Rules for Jakarta Data Products

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

- **Data1** The Product must be able to satisfy all applicable compatibility requirements, including passing all Compatibility Tests, in every Product Configuration and in every combination of Product Configurations, except only as specifically exempted by these Rules. For example, if a Product provides distinct Operating Modes to optimize performance, then that Product must satisfy all applicable compatibility requirements for a Product in each Product Configuration, and combination of Product Configurations, of those Operating Modes.
- **Data1.1** If an Operating Mode controls a Resource necessary for the basic execution of the Test Suite, testing may always use a Product Configuration of that Operating Mode providing that Resource, even if other Product Configurations do not provide that Resource. Notwithstanding such exceptions, each Product must have at least one set of Product Configurations of such Operating Modes that is able to pass all the Compatibility Tests. For example, a Product with an Operating Mode that controls a security policy (i.e., Security Resource) which has one or more Product Configurations that cause Compatibility Tests to fail may be tested using a Product Configuration that allows all Compatibility Tests to pass.
- **Data1.2** A Product Configuration of an Operating Mode that causes the Product to report only version, usage, or diagnostic information is exempted from these compatibility rules.
- **Data1.3** An API Definition Product is exempt from all functional testing requirements defined here, except the signature tests.
- **Data2** Some Compatibility Tests may have properties that may be changed. Properties that can be changed are identified in the configuration interview. Properties that can be changed are identified in the JavaTest Environment (.jte) files in the Test Suite installation. Apart from changing such properties and other allowed modifications described in this User's Guide (if any), no source or binary code for a Compatibility Test may be altered in any way without prior written permission. Any such allowed alterations to the Compatibility Tests will be provided via the Jakarta EE Specification Project website and apply to all vendor compatible implementations.
- **Data3** The testing tools supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.
- **Data4** The Exclude List associated with the Test Suite cannot be modified.
- **Data5** The Maintenance Lead can define exceptions to these Rules. Such exceptions would be made available as above, and will apply to all vendor implementations.
- **Data6** All hardware and software component additions, deletions, and modifications to a Documented supporting hardware/software platform, that are not part of the Product but required for the Product to satisfy the compatibility requirements, must be Documented and available to users of the Product. For example, if a patch to a particular version of a supporting

operating system is required for the Product to pass the Compatibility Tests, that patch must be Documented and available to users of the Product.

- **Data7** The Product must contain the full set of public and protected classes and interfaces for all the Libraries. Those classes and interfaces must contain exactly the set of public and protected methods, constructors, and fields defined by the Specifications for those Libraries. No subsetting, supersetting, or modifications of the public and protected API of the Libraries are allowed except only as specifically exempted by these Rules.
- **Data7.1** If a Product includes Technologies in addition to the Technology Under Test, then it must contain the full set of combined public and protected classes and interfaces. The API of the Product must contain the union of the included Technologies. No further modifications to the APIs of the included Technologies are allowed.
- **Data8** Except for tests specifically required by this TCK to be rebuilt (if any), the binary Compatibility Tests supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.
- **Data9** The functional programmatic behavior of any binary class or interface must be that defined by the Specifications.

14. Links

- Jakarta Data TCK repository - https://github.com/jakartaee/data/tree/main/tck
- Jakarta Data API repository - https://github.com/jakartaee/data/tree/main/api
- Jakarta Data specification repository - https://github.com/jakartaee/data/tree/main/spec
- Jakarta Data project home page - https://projects.eclipse.org/projects/ee4j.data
- Arquillian and ShrinkWrap doc: https://arquillian.org/guides/shrinkwrap_introduction/