



JAKARTA EE

Jakarta EE Platform, Enterprise Edition 11
Test Compatibility Kit User's Guide, Release
11 for Jakarta EE

Table of Contents

Eclipse Foundation™	1
Preface	2
Who Should Use This Book	3
Before You Read This Book	4
Typographic Conventions	5
Shell Prompts in Command Examples	6
Introduction	7
Compatibility Testing	7
About Jakarta EE 11 Platform TCK	8
Hardware Requirements	13
Software Requirements	13
Additional Jakarta EE 11 Platform TCK Requirements	13
Getting Started With the Jakarta EE 11 Platform TCK Test Suite	14
Procedure for Jakarta Platform, Enterprise Edition 11.0 Certification	15
Certification Overview	15
Compatibility Requirements	15
Rules for Jakarta Platform, Enterprise Edition Version 11 Products	18
Jakarta Platform, Enterprise Edition Version 11 Test Appeals Process	21
Specifications for Jakarta Platform, Enterprise Edition Version 11.0	21
Libraries for Jakarta Platform, Enterprise Edition Version 11.0	21
Procedure for Jakarta Platform, Enterprise Edition 11 Web Profile Certification	25
Certification Overview	25
Compatibility Requirements	25
Jakarta Platform, Enterprise Edition Version 11 Web Profile Test Appeals Process	31
Specifications for Jakarta Platform, Enterprise Edition Version 11, Web Profile	31
Libraries for Jakarta Platform, Enterprise Edition Version 11, Web Profile	31
Installation	34
Installing the Jakarta EE 11 Compatible Implementation	34
Installing the Jakarta EE 11 Platform TCK	34
Verifying Your Installation (Optional)	35
Setup and Configuration	36
Allowed Modifications	36
Configuring the Test Environment	36
Generate a Client Certificate	37
Configuring a Jakarta EE 11 Server	37
Jakarta Platform, Enterprise Edition Server Configuration Scenarios	37
Configuring the Jakarta EE 11 CI as the VI	38
Configuring Your Application Server as the VI	39
Modifying Environment Settings for Specific Technology Tests	40
Backend Database Setup	61
Setup and Configuration for Testing with the Jakarta EE 11 Web Profile	63
Configuring the Jakarta EE 11 Web Profile Test Environment	63
To Run Tests Against a Jakarta EE 11 Web Profile Implementation	63
Executing Tests	64

Jakarta EE 11 Platform TCK Operating Assumptions	64
Running the Tests	64
Rebuilding Tests for Different Databases	78
Test Reports	78
Debugging Test Problems	80
Overview	80
Report Files	80
Debugging Details	80
Troubleshooting	81
Common TCK Problems and Resolutions	81
Support	82
Building the Tests	83
Build the Platform TCK EE Tests	83
Implementing the Porting Package	84
Overview	84
Porting Package APIs	85
Jakarta TCK Test Appeals Process	88
Valid Challenges	88
Invalid Challenges	88
TCK Test Appeals Steps	88
A Common Applications Deployment (Needs Rewrite)	90
Table A-1 Required Common Applications	90
Configuring Your Backend Database	91
Overview	91
The init.<database> Ant Target	91
Database Properties in ts.jte	92
Database DDL and DML Files	93
Testing a Standalone Jakarta Messaging Resource Adapter (Full Platform Only)	95
Setting Up Your Environment	95
Configuring Jakarta EE 11 Platform TCK	95
Configuring a Jakarta EE 11 CI for the Standalone Jakarta Messaging Resource Adapter	96
Modifying the Runtime Deployment Descriptors for the Jakarta Messaging MDB and Resource Adapter Tests	96

Jakarta Platform, Enterprise Edition 11 Test Compatibility Kit User's Guide

Release 11 for Jakarta EE

March 2025

Provides detailed instructions for obtaining, installing, configuring, and using the Eclipse Jakarta, Enterprise Edition 11 Compatibility Test Suite for the Full Profile and the Web Profile.

Jakarta Platform, Enterprise Edition 11 Test Compatibility Kit User's Guide, Release 11 for Jakarta EE

Copyright © 2017, 2025 Oracle and/or its affiliates, and others. All rights reserved.

Emeritus Author: Eric Jendrock

This program and the accompanying materials are made available under the terms of the Eclipse Public License v. 2.0, which is available at <http://www.eclipse.org/legal/epl-2.0>.

SPDX-License-Identifier: EPL-2.0

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

References in this document to Java EE refer to the Jakarta EE unless otherwise noted.

Preface



The Jakarta Enterprise Edition documentation is part of the Jakarta Enterprise Edition contribution to the Eclipse Foundation and is not intended for use in relation to Java Enterprise Edition or Java Licensee requirements. This documentation is in the process of being revised to reflect the new Jakarta EE requirements and branding. Additional changes will be made as requirements and procedures evolve for Jakarta EE. Where applicable, references to Java EE or Java Enterprise Edition should be considered references to Jakarta EE.

Please see the Title page for additional license information.

This book introduces the Test Compatibility Kit (TCK) for the Jakarta Platform, Enterprise Edition 11 (Jakarta EE 11) and Jakarta Platform, Enterprise Edition 11 Web Profile (Jakarta EE 11 Web Profile), and explains how to configure and run the test suite. It also provides information for troubleshooting problems you may encounter as you run the test suite.

The Jakarta Platform, Enterprise Edition 11 Test Compatibility Kit (Jakarta EE 11 TCK) is a portable, configurable automated test suite for verifying the compatibility of an implementer's compliance with the Jakarta EE 11 Specification (hereafter referred to as the implementer's implementation, or VI). The Jakarta EE 11 Platform TCK uses the JavaTest harness version 5.0 to run the test suite.



URLs are provided so you can locate resources quickly. However, these URLs are subject to changes that are beyond the control of the authors of this guide.

Who Should Use This Book

This guide is for developers of the Jakarta EE 11 technology to assist them in running the test suite that verifies compatibility of their implementation of the Jakarta EE 11 Specification.

Before You Read This Book

Before reading this guide, you should familiarize yourself with the Java programming language, the Jakarta Platform, Enterprise Edition 11 Specification, and the JavaTest documentation.

The Jakarta Platform, Enterprise Edition 11 Specification can be downloaded from <https://jakarta.ee/specifications/11>.

For documentation on the test harness used for running the Jakarta EE 11 Platform TCK test suite, see:

- [Arquillian](#)
- [JUnit5](#)

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

Convention	Meaning	Example
Boldface	Boldface type indicates graphical user interface elements associated with an action, terms defined in text, or what you type, contrasted with onscreen computer output.	From the File menu, select Open Project. A cache is a copy that is stored locally. <code>machine_name% su</code> <code>Password:</code>
Monospace	Monospace type indicates the names of files and directories, commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name% you have mail.</code>
Italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.	Read Chapter 6 in the User's Guide. Do not save the file. The command to remove a file is <code>rm filename</code> .

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table 1. Shell Prompts

Shell	Prompt
C shell	machine_name%
C shell for superuser	machine_name#
Bourne shell and Korn shell	\$
Bourne shell and Korn shell for superuser	#
Bash shell	shell_name-shell_version\$
Bash shell for superuser	shell_name-shell_version#

Introduction

This document provides instructions for installing, configuring, and running the Jakarta Platform, Enterprise Edition 11 Test Compatibility Kit (Jakarta EE 11 Platform TCK).

This chapter includes the following topics:

- [Compatibility Testing](#)
- [About Jakarta EE 11 Platform TCK](#)
- [Hardware Requirements](#)
- [Software Requirements](#)
- [Additional Jakarta EE 11 Platform TCK Requirements](#)
- [Getting Started With the Jakarta EE 11 Platform TCK Test Suite](#)

Compatibility Testing

Compatibility testing differs from traditional product testing in a number of ways. The focus of compatibility testing is to test those features and areas of an implementation that are likely to differ across other implementations, such as those features that:

- Rely on hardware or operating system-specific behavior
- Are difficult to port
- Mask or abstract hardware or operating system behavior

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

Why Compatibility Testing is Important

Jakarta Platform compatibility is important to different groups involved with Jakarta technologies for different reasons:

- Compatibility testing ensures that the Jakarta Platform does not become fragmented as it is ported to different operating systems and hardware environments.
- Compatibility testing benefits developers working in the Java programming language, allowing them to write applications once and then to deploy them across heterogeneous computing environments without porting.
- Compatibility testing allows application users to obtain applications from disparate sources and deploy them with confidence.
- Conformance testing benefits Jakarta Platform implementors by ensuring a level playing field for all Jakarta Platform ports.

Compatibility Rules

Compatibility criteria for all technology implementations are embodied in the Compatibility Rules that apply to a specified technology. The Jakarta EE 11 Platform TCK tests for adherence to these Rules as described in [Procedure for Jakarta Platform, Enterprise Edition 11.0 Certification](#) for Jakarta EE 11 and [Procedure for Jakarta Platform, Enterprise Edition 11 Web Profile Certification](#) for Jakarta EE 11 Web Profile.

TCK Overview

A Jakarta EE 11 Platform TCK is a set of tools and tests used to verify that a Implementer's implementation of Jakarta EE 11 technology conforms to the applicable specification. All tests in the TCK are based on the written specifications for the Jakarta Platform. The TCK tests compatibility of a Implementer's implementation of a technology to the applicable specification of the technology. Compatibility testing is a means of ensuring correctness, completeness, and consistency across all implementations developed by technology Implementers.

The set of tests included with the Jakarta EE 11 Platform TCK is called the test suite. All tests in the TCK test suite are self-checking, but some tests may require tester interaction. Most tests return either a Pass or Fail status. For a given platform to be certified, all of the required tests must pass. The definition of required tests may change from platform to platform.

The definition of required tests will change over time. Before your final certification test pass, be sure to download the latest Jakarta EE 11 Platform TCK. The definition of required tests will change over time. See [Exclude Tests](#) for more information.

Jakarta Specification Community Process Program and Compatibility Testing

The Jakarta EE Specification Process (JESP) program is the formalization of the open process that has been used since 2019 to develop and revise Jakarta EE technology specifications in cooperation with the international Jakarta EE community. The JESP program specifies that the following three major components must be included as deliverables in a final Jakarta EE technology release under the direction of the responsible specification project committer group:

- Technology Specification
- A Compatible Implementation
- Technology Compatibility Kit (TCK)

For further information about the JESP program, go to Jakarta EE Specification Process community page (<https://jakarta.ee/specifications>).

About Jakarta EE 11 Platform TCK

Jakarta EE 11 Platform TCK is a portable, configurable, automated test suite for verifying the compliance of an Implementer's implementation of the Jakarta EE 11 technologies.

For documentation on the test harness used for running the Jakarta EE 11 Platform TCK test suite, see [Arquillian](#) and [JUnit5](#) for the underlying test framework documentation.

Jakarta EE 11 Technologies Required for Jakarta EE 11 Platform Compatibility

The Jakarta EE 11 Platform Specification defines the required and optional component specifications. The full list with specification version requirements is defined in the Platform EE Specification document (<https://jakarta.ee/specifications/platform/11/>), see the heading *Full Jakarta™ EE Product Requirements*,

Jakarta EE 11 TCK tests verify partial compatibility for the Jakarta EE Platform. The Platform TCK includes tests for the following components:

The complete list of Jakarta EE 11 technologies for the Platform can be found in section 9.7 of the [Platform Specification](#).

- Jakarta Activation
- Jakarta Authentication

- Jakarta Authorization
- Jakarta Batch
- Jakarta Bean Validation
- Jakarta Common Annotations
- Jakarta Concurrency
- Jakarta Connectors
- Jakarta Contexts and Dependency Injection
- Jakarta Debugging Support for Other Languages
- Jakarta Dependency Injection
- Jakarta Enterprise Beans (also, see optional below)
- Jakarta Enterprise Web Services
- Jakarta Expression Language
- Jakarta Server Faces
- Jakarta Interceptors
- Jakarta JSON Binding
- Jakarta JSON Processing
- Jakarta Mail
- Jakarta Messaging
- Jakarta Persistence
- Jakarta RESTful Web Services
- Jakarta Security
- Jakarta Server Pages
- Jakarta Servlet
- Jakarta Standard Tag Library
- Jakarta Transactions
- Jakarta WebSocket

Jakarta EE 11 Platform TCK provides compatibility certification verification for implementations contained in the Platform for the following component specifications:

- Jakarta Annotations
- Jakarta Authorization
- Jakarta Connectors
- Jakarta Enterprise Beans (including optional elements)
- Jakarta Expression Language
- Jakarta Interceptors
- Jakarta Messaging
- Jakarta Persistence
- Jakarta Server Pages
- Jakarta Servlet
- Jakarta SOAP with Attachments
- Jakarta Standard Tag Library
- Jakarta Transactions
- Jakarta Web Socket
- Jakarta XML Web Services

Jakarta EE 11 Web Profile Technologies Tested With Jakarta EE 11 Platform TCK

The Jakarta EE 11 Web Profile Specification defines the required component specifications. The complete list with specification version requirements is defined in the Web Profile specification document (<https://jakarta.ee/specifications/webprofile/11/>), see heading *Web Profile Definition*, sub-heading *Required Components*.

The Jakarta EE 11 Platform TCK test suite provides partial compatibility verification for the following component technologies:

- Jakarta Annotations
- Jakarta Authentication, Servlet Container Profile
- Jakarta Bean Validation
- Jakarta Common Annotations
- Jakarta Contexts and Dependency Injection
- Jakarta Concurrency
- Jakarta Debugging Support for Other Languages
- Jakarta Dependency Injection
- Jakarta Enterprise Beans, Lite
- Jakarta Expression Language
- Jakarta Faces
- Jakarta Interceptors
- Jakarta JSON Binding
- Jakarta JSON Processing
- Jakarta Persistence
- Jakarta RESTful Web Services
- Jakarta Security
- Jakarta Server Pages
- Jakarta Servlet
- Jakarta Standard Tag Library
- Jakarta Transactions
- Jakarta WebSocket

There are no optional specifications defined in the Web Profile specification.

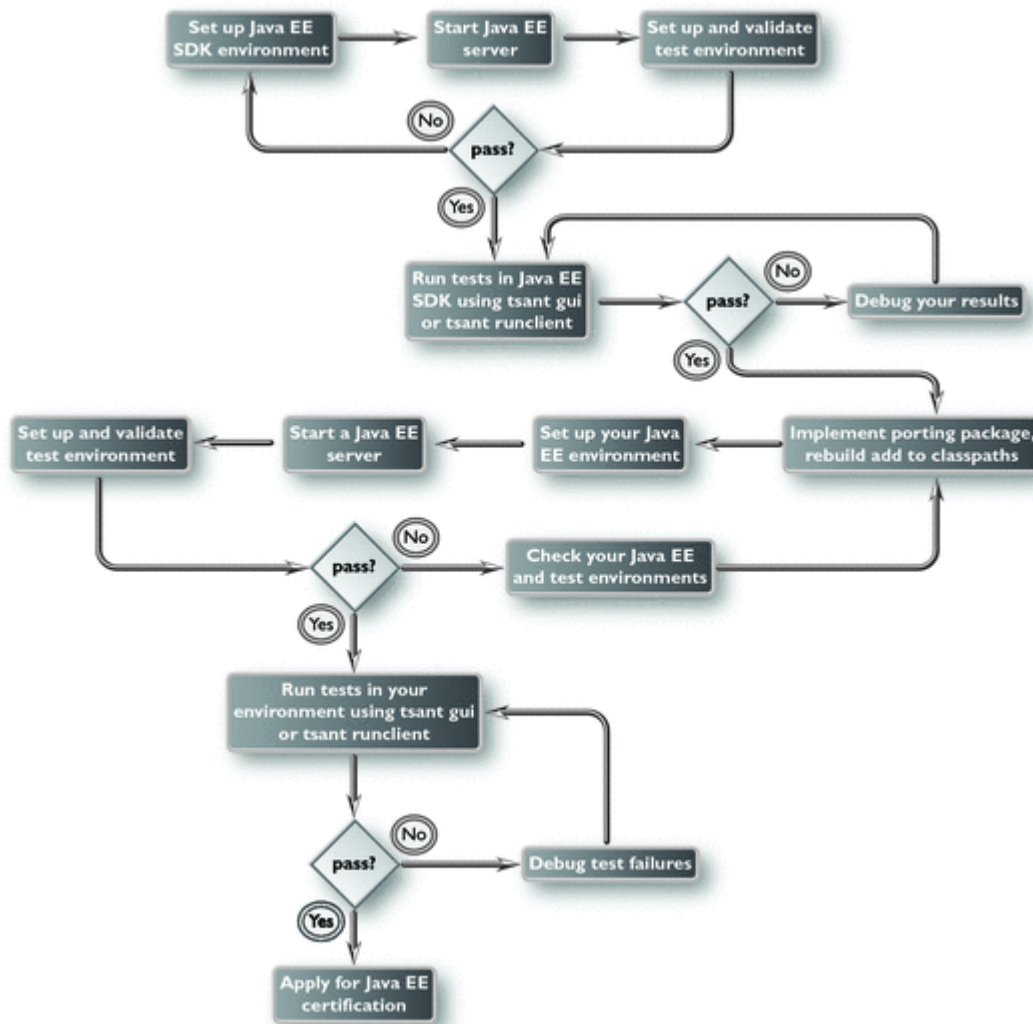
Jakarta EE 11 Platform TCK provides compatibility certification verification for implementations contained in the platform, Web Profile for the following component specifications:

- Jakarta Annotations
- Jakarta Enterprise Beans, Lite
- Jakarta Expression Language
- Jakarta Interceptors
- Jakarta Servlet
- Jakarta Standard Tag Library
- Jakarta Transactions
- Jakarta Web Socket

TCK Tests

The Jakarta EE 11 Platform TCK contains API tests and enterprise edition tests, which are tests that start in the Jakarta EE 11 platform and use the underlying enterprise service or services as specified. For example, a JDBC enterprise edition test connects to a database, uses SQL commands and the JDBC 4.2 API to populate the database tables with data, queries the database, and compares the returned results against the expected results.

Typical Jakarta Platform, Enterprise Edition Workflow



Note: References in diagram to Java EE refer to Jakarta EE.

Typical Jakarta Platform, Enterprise Edition Workflow shows how most Implementers will use the test suite. They will set up and run the test suite with the Jakarta Platform, Enterprise Edition 11 Compatible Implementation (Jakarta EE 11 CI) first to become familiar with the testing process. Then they will set up and run the test suite with their own Jakarta EE 11 implementation. This is called the Vendor Implementation, or VI in this document. When they pass all of the tests, they will apply for and be granted certification.

- Before you do anything with the test suite, read the rules in [Procedure for Jakarta Platform, Enterprise Edition 11.0 Certification](#) or [Procedure for Jakarta Platform, Enterprise Edition 11 Web Profile Certification](#). These chapters explain the certification process and provides a definitive list of certification rules for Jakarta EE 11 and Jakarta EE 11 Web Profile implementations.
- Third, install and configure the Jakarta EE 11 Platform TCK software and the Jakarta EE 11 CI or Jakarta EE 11 Web Profile CI and run the tests as described in this guide. This will familiarize you with the testing process.

- Finally, set up and run the test suite with your own Jakarta EE 11 or Jakarta EE 11 Web Profile implementation.



In the instructions in this document, variables in angle brackets need to be expanded for each platform. For example, `<TS_HOME>` becomes `$TS_HOME` on Solaris/Linux and `%TS_HOME%` on Windows. In addition, the forward slashes (/) used in all of the examples need to be replaced with backslashes (\) for Windows.

Arquillian and Junit5

The Arquillian and Junit5 are set of tools designed to run and manage test suites on different Java platforms.

The tests that make up the TCK are precompiled and bundled in the TCK distribution/artifacts directory as test jars. You will need to create a runner that supports executing Junit5 unit tests. The GlassFish 8.0 compatible implementation used maven with the failsafe plugin to run the tests.

Exclude Tests

As of version 11, the Jakarta EE Platform TCK uses Junit5 `org.junit.jupiter.api.Disabled` annotations to exclude tests from the test suite. Test methods or classes that are successfully challenged are annotated with the `@Disabled("https://link-to-challenge-issue")` and released in a new service release.

An implementor is not required to pass or run any tests that are annotated with the `@Disabled("...")` tag. The `@Disabled("...")` annotation is used to indicate that the test is not required for certification. When a service release is made to exclude tests due to a challenge, the tests with any `@Disabled("...")` annotations are removed from the test suite by the Junit5 framework.

A test might be in the Exclude List for reasons such as:

- An error in an underlying implementation API has been discovered which does not allow the test to execute properly.
- An error in the specification that was used as the basis of the test has been discovered.
- An error in the test itself has been discovered.
- The test fails due to a bug in the tools (such as the Arquillian/Junit5 harness, for example).

In addition, all tests are run against the compatible implementations. Any tests that cannot be run on a compatible Jakarta Platform may be put on the Exclude List if the Specification project team agrees the test is invalid. Any test that is not specification-based, or for which the specification is vague, may be excluded. Any test that is found to be implementation dependent (based on a particular thread scheduling model, based on a particular file system behavior, and so on) may be excluded.



Implementers are not permitted to alter or modify which tests are Excluded Tests. Changes the Excluded Tests can only be made by using the procedure described in [Jakarta Platform, Enterprise Edition Version 11 Test Appeals Process](#) and [Jakarta Platform, Enterprise Edition Version 11 Web Profile Test Appeals Process](#)

Apache Ant (Optional)

There are example setup scripts for the TCK test databases that use of Apache Ant 1.9.7 from the Apache Ant Project (<http://ant.apache.org/>). Apache Ant is a free, open-source, Java-based build tool, similar in some ways to the make tool, but more flexible, cross-platform compatible, and centered around XML-based configuration files. You do not need to use these scripts, but the SQL statements in the scripts will have to be loaded into your testing database in order for the TCK tests to pass.

Apache Ant is protected under the Apache Software, License 2.0, which is available on the Apache Ant Project license page at <http://ant.apache.org/license.html>.

Installing Apache Ant

- Download the Apache Ant 1.9.7 binary bundle from the Apache Ant Project.
- Change to the directory in which you want to install Apache Ant and extract the bundle
- Set the ANT_HOME environment variable to point to the apache-ant-<version> directory
- Add <ANT_HOME>/bin directory to the environment variable PATH

Hardware Requirements

The following section lists the hardware requirements for the Jakarta EE 11 TCK software, using the Jakarta EE 11 CI or Jakarta EE 11 Web Profile CI. Hardware requirements for other compatible implementations will vary.

All systems should meet the following recommended hardware requirements:

- CPU running at 2.0 GHz or higher
- 4 GB of RAM or more
- 2 GB of swap space , if required
- 6 GB of free disk space for writing data to log files, the Jakarta EE 11 repository, and the database
- Network access to the Internet

Software Requirements

You can run the Jakarta EE 11 Platform TCK software on platforms running the Linux software that meet the following software requirements:

- Operating Systems:
 - Any operating system that supports the Java SE 17 or 21 platform
- Java SE 17 or 21
- Jakarta EE 11 CI or Jakarta EE 11 Web Profile CI
- Mail server that supports the IMAP and SMTP protocols (Full Platform Only)
- One of the following databases:
 - MySQL
 - Apache Derby

Additional Jakarta EE 11 Platform TCK Requirements

In addition to the instructions and requirements described in this document, all Jakarta EE 11 Platform implementations must also pass the standalone TCKs for the following technologies. See the links for additional details.

- Jakarta Activation — <https://jakarta.ee/specifications/activation/2.1/>
- Jakarta Authentication — <https://jakarta.ee/specifications/authentication/3.1/>
- Jakarta Batch — <https://jakarta.ee/specifications/batch/2.1/>
- Jakarta Bean Validation — <https://jakarta.ee/specifications/bean-validation/3.1/>
- Jakarta Concurrency — <https://jakarta.ee/specifications/concurrency/3.1/>
- Jakarta Contexts and Dependency Injection (including Language Model TCK) — <https://jakarta.ee/specifications/cdi/4.1/>

- Jakarta Data — <https://jakarta.ee/specifications/data/1.0/>
- Jakarta Debugging Support for Other Languages — <https://jakarta.ee/specifications/debugging/2.0/>
- Jakarta Dependency Injection — <https://jakarta.ee/specifications/dependency-injection/2.0/>
- Jakarta Faces — <https://jakarta.ee/specifications/faces/4.1/>
- Jakarta JSON Binding — <https://jakarta.ee/specifications/jsonb/3.0/>
- Jakarta JSON Processing — <https://jakarta.ee/specifications/jsonp/2.1/>
- Jakarta Mail — <https://jakarta.ee/specifications/mail/2.1/>
- Jakarta RESTful Web Services — <https://jakarta.ee/specifications/restful-ws/4.0/>
- Jakarta Security — <https://jakarta.ee/specifications/security/4.0/>
- Jakarta Servlet — <https://jakarta.ee/specifications/servlet/6.1/>
- Jakarta WebSocket — <https://jakarta.ee/specifications/websocket/2.2/>

All Jakarta EE 11 Web Profile implementations must also pass the standalone TCKs for the following technologies:

- Jakarta Authentication — <https://jakarta.ee/specifications/authentication/3.1/>
- Jakarta Bean Validation — <https://jakarta.ee/specifications/bean-validation/3.1/>
- Jakarta Concurrency — <https://jakarta.ee/specifications/concurrency/3.1/>
- Jakarta Contexts and Dependency Injection (including Language Model TCK) — <https://jakarta.ee/specifications/cdi/4.1/>
- Jakarta Data — <https://jakarta.ee/specifications/data/1.0/>
- Jakarta Debugging Support for Other Languages — <https://jakarta.ee/specifications/debugging/>
- Jakarta Dependency Injection — <https://jakarta.ee/specifications/dependency-injection/2.0/>
- Jakarta Faces — <https://jakarta.ee/specifications/faces/4.0/>
- Jakarta JSON Binding — <https://jakarta.ee/specifications/jsonb/3.0/>
- Jakarta JSON Processing — <https://jakarta.ee/specifications/jsonp/2.1/>
- Jakarta RESTful Web Services — <https://jakarta.ee/specifications/restful-ws/4.0/>
- Jakarta Security — <https://jakarta.ee/specifications/security/4.0/>
- Jakarta Servlet — <https://jakarta.ee/specifications/servlet/6.1/>
- Jakarta WebSocket — <https://jakarta.ee/specifications/websocket/2.2/>



Web Profile implementations may ignore sections in this labelled as (Full Platform Only).

Getting Started With the Jakarta EE 11 Platform TCK Test Suite

Installing, configuring, and using the Jakarta EE 11 Platform TCK involves the following general steps:

1. Download, install, and configure a Jakarta EE 11 CI or Jakarta EE 11 Web Profile CI. For example Eclipse GlassFish 8.0.
2. Download and install the Jakarta EE 11 Platform TCK package.
3. Configure your database to work with your CI.
4. Configure the TCK to work with your database and CI.
5. Run the TCK tests.

The remainder of this guide explains these steps in detail. If you just want to get started quickly with the Jakarta EE 11 Platform TCK using the most basic test configuration, refer to [Installation](#).

Procedure for Jakarta Platform, Enterprise Edition 11.0 Certification

This chapter describes the compatibility testing procedure and compatibility requirements for Jakarta Platform, Enterprise Edition Version 11.

This chapter contains the following sections:

- [Certification Overview](#)
- [Compatibility Requirements](#)
- [Jakarta Platform, Enterprise Edition Version 11 Test Appeals Process](#)
- [Specifications for Jakarta Platform, Enterprise Edition Version 11.0](#)
- [Libraries for Jakarta Platform, Enterprise Edition Version 11.0](#)

Certification Overview

The certification process for Jakarta EE 11.0 consists of the following activities:

- Install the appropriate version of the Technology Compatibility Kit (TCK) and execute it in accordance with the instructions in this User's Guide.
- Ensure that you meet the requirements outlined in [Compatibility Requirements](#) below.
- Certify to the Eclipse Foundation that you have finished testing and that you meet all the compatibility requirements, as required by the Eclipse Foundation TCK License.

Compatibility Requirements

The compatibility requirements for Jakarta EE 11.0 consist of meeting the requirements set forth by the rules and associated definitions contained in this section.

Definitions

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

Table 2-1 Definitions

Term	Definition
API Definition Product	A Product for which the only Java class files contained in the product are those corresponding to the application programming interfaces defined by the Specifications, and which is intended only as a means for formally specifying the application programming interfaces defined by the Specifications.
Application	A collection of components contained in a single application package (such as an EAR file or WAR file) and deployed at the same time.

Term	Definition
Computational Resource	<p>A piece of hardware or software that may vary in quantity, existence, or version, which may be required to exist in a minimum quantity and/or at a specific or minimum revision level so as to satisfy the requirements of the Test Suite.</p> <p>Examples of computational resources that may vary in quantity are RAM and file descriptors.</p> <p>Examples of computational resources that may vary in existence (that is, may or may not exist) are graphics cards and device drivers.</p> <p>Examples of computational resources that may vary in version are operating systems and device drivers.</p>
Configuration Descriptor	Any file whose format is well defined by a specification and which contains configuration information for a set of Java classes, archive, or other feature defined in the specification.
Conformance Tests	All tests in the Test Suite for an indicated Technology Under Test, as released and distributed by the Eclipse Foundation, excluding those tests in the Excluded Test set for the Technology Under Test.
Container	An implementation of the associated Libraries, as specified in the Specifications, and a version of a Java Platform, Standard Edition Runtime Product, as specified in the Specifications, or a later version of a Java Platform, Standard Edition Runtime Product that also meets these compatibility requirements.
Deployment Tool	A tool used to deploy applications or components in a Product, as described in the Specifications.
Development Kit	A software product that implements or incorporates a Compiler, a Schema Compiler, a Schema Generator.
Documented	Made technically accessible and made known to users, typically by means such as marketing materials, product documentation, usage messages, or developer support programs.
Edition	A Version of the Java Platform. Editions include Java Platform Standard Edition and Jakarta Platform Enterprise Edition.
Endorsed Standard	A Java API defined through a standards process other than the Jakarta Enterprise Specification Process. The Endorsed Standard packages are listed later in this chapter.
Excluded Tests	Each release of a TCK may include tests that are annotated as excluded. These tests are not required to be run by the TCK user.
Jakarta Server Page	A text-based document that uses Jakarta Pages technology.
Jakarta Server Page Implementation Class	A program constructed by transforming the Jakarta Server Page text into a Java language program using the transformation rules described in the Specifications.
Libraries	<p>The class libraries, as specified through the Jakarta EE Specification Process (JESP), for the Technology Under Test.</p> <p>The Libraries for Jakarta Platform, Enterprise Edition Version 11 are listed at the end of this chapter.</p>

Term	Definition
Location Resource	<p>A location of classes or native libraries that are components of the test tools or tests, such that these classes or libraries may be required to exist in a certain location in order to satisfy the requirements of the test suite.</p> <p>For example, classes may be required to exist in directories named in a CLASSPATH variable, or native libraries may be required to exist in directories named in a PATH variable.</p>
Maintenance Lead	<p>The corresponding Jakarta EE Specification Project is responsible for maintaining the Specification and the TCK for the Technology. The Specification Project Team will propose revisions and updates to the Jakarta EE Specification Committee which will approve and release new versions of the specification and TCK. Eclipse Jakarta EE Specification Committee is the Maintenance Lead for Jakarta Platform, Enterprise Edition Version 11.</p>
Operating Mode	<p>Any Documented option of a Product that can be changed by a user in order to modify the behavior of the Product.</p> <p>For example, an Operating Mode of a Runtime can be binary (enable/disable optimization), an enumeration (select from a list of localizations), or a range (set the initial Runtime heap size).</p> <p>Note that an Operating Mode may be selected by a command line switch, an environment variable, a GUI user interface element, a configuration or control file, etc.</p>
Product	<p>A vendor's product in which the Technology Under Test is implemented or incorporated, and that is subject to compatibility testing.</p>
Product Configuration	<p>A specific setting or instantiation of an Operating Mode.</p> <p>For example, a Product supporting an Operating Mode that permits user selection of an external encryption package may have a Product Configuration that links the Product to that encryption package.</p>
Rebuildable Tests	<p>Tests that must be built using an implementation-specific mechanism. This mechanism must produce specification defined artifacts. Rebuilding and running these tests against a known compatible implementation verifies that the mechanism generates compatible artifacts.</p>
Compatible Implementation (CI)	<p>A ratified compatible implementation of a Specification.</p>
Resource	<p>A Computational Resource, a Location Resource, or a Security Resource.</p>
Rules	<p>These definitions and rules in this Compatibility Requirements section of this User's Guide.</p>
Runtime	<p>The Containers specified in the Specifications.</p>
Security Resource	<p>A security privilege or policy necessary for the proper execution of the Test Suite.</p> <p>For example, the user executing the Test Suite will need the privilege to access the files and network resources necessary for use of the Product.</p>

Term	Definition
Specifications	<p>The documents produced through the Jakarta EE Specification Process (JESP) that define a particular Version of a Technology.</p> <p>The Specifications for the Technology Under Test are referenced later in this chapter.</p>
Technology	Specifications and one or more compatible implementations produced through the Jakarta EE Specification Process (JESP).
Technology Under Test	Specifications and a compatible implementation for Jakarta Platform, Enterprise Edition Version 11.
Test Suite	The requirements, tests, and testing tools distributed by the Maintenance Lead as applicable to a given Version of the Technology.
Version	A release of the Technology, as produced through the Jakarta EE Specification Process (JESP).

Rules for Jakarta Platform, Enterprise Edition Version 11 Products

The following rules apply for each implementation:

EE1 The Product must be able to satisfy all applicable compatibility requirements, including passing all required TCK tests.

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.

EE1.1 Each implementation must have at least one Product Configuration that can be used to pass all required TCK Tests, although such configuration may need adjustment (e.g. whether statically or via administrative tooling).

EE1.2 An implementation may have mode(s) that provide compatibility with previous Jakarta EE versions.

EE1.3 An API Definition Product is exempt from all functional testing requirements defined here, except the signature tests.

EE2 Some Conformance Tests may have properties that may be changed. Properties that can be changed are identified in the configuration interview. Properties that can be changed are identified in the JavaTest Environment (.jte) files in the lib directory of 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 Conformance Test may be altered in any way without prior written permission. Any such allowed alterations to the Conformance Tests will be provided via the Jakarta EE Specification Project website and apply to all vendor compatible implementations.

EE3 The testing tools supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.

EE4 The Excluded Tests associated with the Test Suite cannot be modified.

EE5 The Maintenance Lead may define exceptions to these Rules. Such exceptions would be made available as above, and will apply to all vendor implementations.

EE6 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.

EE7 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.

EE7.1 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.

EE7.2 A Product may provide a newer version of an Endorsed Standard. Upon request, the Maintenance Lead will make available alternate Conformance Tests as necessary to conform with such newer version of an Endorsed Standard. Such alternate tests will be made available to and apply to all implementers. If a Product provides a newer version of an Endorsed Standard, the version of the Endorsed Standard supported by the Product must be Documented.

EE7.3 The Maintenance Lead may authorize the use of newer Versions of a Technology included in the Technology Under Test. A Product that provides a newer Version of a Technology must meet the Compatibility Requirements for that newer Version, and must Document that it supports the newer Version.

For example, the Jakarta Platform, Enterprise Edition Maintenance Lead could authorize use of a newer version of a Java technology such as Jakarta XML Web Services.

EE8 Except for tests specifically required by this TCK to be rebuilt (if any), the binary Conformance Tests supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.

EE9 The functional programmatic behavior of any binary class or interface must be that defined by the Specifications.

EE9.1 A Product may contain Operating Modes that meet all of these requirements, except Rule EE9, provided that:

1. The Operating Modes must not violate the Java Platform, Standard Edition Rules.
2. Some Product Configurations of such Operating Modes may provide only a subset of the functional programmatic behavior required by the Specifications. The behavior of applications that use more than the provided subset, when run in such Product Configurations, is unspecified.
3. The functional programmatic behavior of any binary class or interface in the above defined subset must be that defined by the Specifications.
4. Any Product Configuration that invokes this rule must be clearly Documented as not fully meeting the requirements of the Specifications.

EE10 Each Container must make technically accessible all Java SE Runtime interfaces and functionality, as defined by the Specifications, to programs running in the Container, except only as specifically exempted by these Rules.

EE10.1 Containers may impose security constraints, as defined by the Specifications.

EE11 A web Container must report an error, as defined by the Specifications, when processing a Jakarta Server Page that does not conform to the Specifications.

EE12 The presence of a Java language comment or Java language directive in a Jakarta Server Page that specifies "java" as the scripting language, when processed by a web Container, must not cause the functional programmatic behavior of that Jakarta Server Page to vary from the functional programmatic behavior of that Jakarta Server Page in the absence of that Java language comment or Java language directive.

EE13 The contents of any fixed template data (defined by the Specifications) in a Jakarta Server Page, when processed by a web Container, must not affect the functional programmatic behavior of that Jakarta Server Page, except as defined by the Specifications.

EE14 The functional programmatic behavior of a Jakarta Server Page that specifies "java" as the scripting language

must be equivalent to the functional programmatic behavior of the Jakarta Server Page Implementation Class constructed from that Jakarta Server Page.

EE15 A Deployment Tool must report an error when processing a Configuration Descriptor that does not conform to the Specifications.

EE16 The presence of an XML comment in a Configuration Descriptor, when processed by a Deployment Tool, must not cause the functional programmatic behavior of the Deployment Tool to vary from the functional programmatic behavior of the Deployment Tool in the absence of that comment.

EE17 A Deployment Tool must report an error when processing an Jakarta Enterprise Beans deployment descriptor that includes an Jakarta Enterprise Beans QL expression that does not conform to the Specifications.

EE18 The Runtime must report an error when processing a Configuration Descriptor that does not conform to the Specifications.

EE19 An error must be reported when processing a configuration descriptor that includes a Java Persistence QL expression that does not conform to the Specifications.

EE20 The presence of an XML comment in a Configuration Descriptor, when processed by the Runtime, must not cause the functional programmatic behavior of the Runtime to vary from the functional programmatic behavior of the Runtime in the absence of that comment.

EE21 Compliance testing for Jakarta EE 11.0 consists of running Jakarta EE 11.0 TCK and the following Technology Compatibility Kits (TCKs). Version details are defined in the Platform EE Specification document (<https://jakarta.ee/specifications/platform/11/>), see the heading *Full Jakarta™ EE Product Requirements*:

- Jakarta Activation
- Jakarta Authentication
- Jakarta Batch
- Jakarta Bean Validation
- Jakarta Concurrency
- Jakarta Contexts and Dependency Injection
- Jakarta Data
- Jakarta Debugging Support for Other Languages
- Jakarta Dependency Injection
- Jakarta Faces
- Jakarta JSON Binding
- Jakarta JSON Processing
- Jakarta Mail
- Jakarta RESTful Web Services
- Jakarta Security

In addition to the compatibility rules outlined in this TCK User's Guide, Jakarta EE 11.0 implementations must also adhere to all of the compatibility rules defined in the User's Guides of the aforementioned TCKs.

EE21.1 If the Jakarta EE 11 implementation uses a runtime which has already been validated by the Technology Compatibility Kit, the Jakarta EE 11 implementation may use result of such validation to claim its compliance with the Technology Compatibility Kit.

Jakarta Platform, Enterprise Edition Version 11 Test Appeals Process

See [TCK Test Appeals Steps](#) for the Jakarta Platform, Enterprise Edition Version 11 Test Appeals Process.

Specifications for Jakarta Platform, Enterprise Edition Version 11.0

The Specifications for Jakarta Platform, Enterprise Edition 11.0 are found on the Eclipse Foundation, Jakarta EE Specifications web site at <https://jakarta.ee/specifications/platform/11/>. You may also find information available from the EE4J Jakarta EE Platform project page, at <https://projects.eclipse.org/projects/ee4j.jakartaee-platform>.

Libraries for Jakarta Platform, Enterprise Edition Version 11.0

The following list constitutes the complete list of packages that are required for Jakarta EE 11.0:

- jakarta.annotation
- jakarta.annotation.security
- jakarta.annotation.sql
- jakarta.batch.api
- jakarta.batch.api.chunk
- jakarta.batch.api.chunk.listener
- jakarta.batch.api.listener
- jakarta.batch.api.partition
- jakarta.batch.operations
- jakarta.batch.runtime
- jakarta.batch.runtime.context
- jakarta.decorator
- jakarta.data
- jakarta.data.exceptions
- jakarta.data.metamodel
- jakarta.data.page
- jakarta.data.repository
- jakarta.data.spi
- jakarta.ejb
- jakarta.ejb.embeddable (removed from Jakarta EE 11 Platform but still part of Jakarta Enterprise Beans 4.0)
- jakarta.ejb.spi
- jakarta.el
- jakarta.enterprise.concurrent
- jakarta.enterprise.context
- jakarta.enterprise.context.control
- jakarta.enterprise.context.spi
- jakarta.enterprise.event
- jakarta.enterprise.inject
- jakarta.enterprise.inject.literal
- jakarta.enterprise.inject.se
- jakarta.enterprise.inject.spi

- jakarta.enterprise.inject.spi.configurator
- jakarta.enterprise.util
- jakarta.faces
- jakarta.faces.annotation
- jakarta.faces.application
- jakarta.faces.bean
- jakarta.faces.component
- jakarta.faces.component.behavior
- jakarta.faces.component.html
- jakarta.faces.component.search
- jakarta.faces.component.visit
- jakarta.faces.context
- jakarta.faces.convert
- jakarta.faces.el
- jakarta.faces.event
- jakarta.faces.flow
- jakarta.faces.flow.builder
- jakarta.faces.lifecycle
- jakarta.faces.model
- jakarta.faces.push
- jakarta.faces.render
- jakarta.faces.validator
- jakarta.faces.view
- jakarta.faces.view.facelets
- jakarta.faces.webapp
- jakarta.inject
- jakarta.interceptor
- jakarta.jms
- jakarta.json
- jakarta.json.bind
- jakarta.json.bind.adapter
- jakarta.json.bind.annotation
- jakarta.json.bind.config
- jakarta.json.bind.serializer
- jakarta.json.bind.spi
- jakarta.json.spi
- jakarta.json.stream
- jakarta.mail
- jakarta.mail.event
- jakarta.mail.internet
- jakarta.mail.search
- jakarta.mail.util
- jakarta.persistence

- jakarta.persistence.criteria
- jakarta.persistence.metamodel
- jakarta.persistence.spi
- jakarta.resource
- jakarta.resource.cci
- jakarta.resource.spi
- jakarta.resource.spi.endpoint
- jakarta.resource.spi.security
- jakarta.resource.spi.work
- jakarta.security.auth.message
- jakarta.security.auth.message.callback
- jakarta.security.auth.message.config
- jakarta.security.auth.message.module
- jakarta.security.enterprise
- jakarta.security.enterprise.authentication.mechanism.http
- jakarta.security.enterprise.credential
- jakarta.security.enterprise.identitystore
- jakarta.security.jacc
- jakarta.servlet
- jakarta.servlet.annotation
- jakarta.servlet.descriptor
- jakarta.servlet.http
- jakarta.servlet.jsp
- jakarta.servlet.jsp.el
- jakarta.servlet.jsp.jstl.core
- jakarta.servlet.jsp.jstl.fmt
- jakarta.servlet.jsp.jstl.sql
- jakarta.servlet.jsp.jstl.tlv
- jakarta.servlet.jsp.tagext
- jakarta.transaction
- javax.transaction.xa
- jakarta.validation
- jakarta.validation.bootstrap
- jakarta.validation.constraints
- jakarta.validation.constraintvalidation
- jakarta.validation.executable
- jakarta.validation.groups
- jakarta.validation.metadata
- jakarta.validation.spi
- jakarta.validation.valueextraction
- jakarta.websocket
- jakarta.websocket.server
- jakarta.ws.rs

- `jakarta.ws.rs.client`
- `jakarta.ws.rs.container`
- `jakarta.ws.rs.core`
- `jakarta.ws.rs.ext`
- `jakarta.ws.rs.sse`

Procedure for Jakarta Platform, Enterprise Edition 11 Web Profile Certification

This chapter describes the compatibility testing procedure and compatibility requirements for Jakarta Platform, Enterprise Edition Version 11 Web Profile.

This chapter contains the following sections:

- [Certification Overview](#)
- [Compatibility Requirements](#)
- [Jakarta TCK Test Appeals Process](#)
- [Specifications for Jakarta Platform, Enterprise Edition Version 11, Web Profile](#)
- [Libraries for Jakarta Platform, Enterprise Edition Version 11, Web Profile](#)

Certification Overview

The certification process for Jakarta EE 11, Web Profile consists of the following activities:

- Install the appropriate version of the Technology Compatibility Kit (TCK) and execute it in accordance with the instructions in this User's Guide.
- Ensure that you meet the requirements outlined in "Compatibility Requirements," below.
- Certify to the Eclipse Foundation that you have finished testing and that you meet all of the compatibility requirements, as required by the Eclipse Foundation TCK License.

Compatibility Requirements

The compatibility requirements for Jakarta EE 11, Web Profile consist of meeting the requirements set forth by the rules and associated definitions contained in this section.

Definitions

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

Table 3-1 Definitions

Term	Definition
API Definition Product	A Product for which the only Java class files contained in the product are those corresponding to the application programming interfaces defined by the Specifications, and which is intended only as a means for formally specifying the application programming interfaces defined by the Specifications.
Application	A collection of components contained in a single application package (such as an EAR file or WAR file) and deployed at the same time.

Term	Definition
Computational Resource	<p>A piece of hardware or software that may vary in quantity, existence, or version, which may be required to exist in a minimum quantity and/or at a specific or minimum revision level so as to satisfy the requirements of the Test Suite.</p> <p>Examples of computational resources that may vary in quantity are RAM and file descriptors.</p> <p>Examples of computational resources that may vary in existence (that is, may or may not exist) are graphics cards and device drivers.</p> <p>Examples of computational resources that may vary in version are operating systems and device drivers.</p>
Configuration Descriptor	Any file whose format is well defined by a specification and which contains configuration information for a set of Java classes, archive, or other feature defined in the specification.
Conformance Tests	All tests in the Test Suite for an indicated Technology Under Test, as released and distributed by the Eclipse Foundation, excluding those tests in the Excluded Test set for the Technology Under Test.
Container	An implementation of the associated Libraries, as specified in the Specifications, and a version of a Java Platform, Standard Edition Runtime Product, as specified in the Specifications, or a later version of a Java Platform, Standard Edition Runtime Product that also meets these compatibility requirements.
Deployment Tool	A tool used to deploy applications or components in a Product, as described in the Specifications.
Documented	Made technically accessible and made known to users, typically by means such as marketing materials, product documentation, usage messages, or developer support programs.
Edition	A Version of the Java Platform. Editions include Java Platform Standard Edition and Jakarta Platform Enterprise Edition.
Endorsed Standard	A Java API defined through a standards process other than the Jakarta Enterprise Specification Process. The Endorsed Standard packages are listed later in this chapter.
Excluded Tests	Each release of a TCK may include tests that are annotated as excluded. These tests are not required to be run by the TCK user.
Jakarta Server Page	A text-based document that uses Jakarta Pages technology.
Jakarta Server Page Implementation Class	A program constructed by transforming the Jakarta Server Page text into a Java language program using the transformation rules described in the Specifications.
Libraries	<p>The class libraries, as specified through the Jakarta EE Specification Process (JESP), for the Technology Under Test.</p> <p>The Libraries for Jakarta Platform, Enterprise Edition Version 11 are listed at the end of this chapter.</p>

Term	Definition
Location Resource	<p>A location of classes or native libraries that are components of the test tools or tests, such that these classes or libraries may be required to exist in a certain location in order to satisfy the requirements of the test suite.</p> <p>For example, classes may be required to exist in directories named in a CLASSPATH variable, or native libraries may be required to exist in directories named in a PATH variable.</p>
Maintenance Lead	<p>The corresponding Jakarta EE Specification Project is responsible for maintaining the Specification and the TCK for the Technology. The Specification Project Team will propose revisions and updates to the Jakarta EE Specification Committee which will approve and release new versions of the specification and TCK. Eclipse Jakarta EE Specification Committee is the Maintenance Lead for Jakarta Platform, Enterprise Edition Version 11, Web Profile.</p>
Operating Mode	<p>Any Documented option of a Product that can be changed by a user in order to modify the behavior of the Product.</p> <p>For example, an Operating Mode can be binary (enable/disable optimization), an enumeration (select from a list of protocols), or a range (set the maximum number of active threads).</p> <p>Note that an Operating Mode may be selected by a command line switch, an environment variable, a GUI user interface element, a configuration or control file, etc.</p>
Product	<p>A vendor's product in which the Technology Under Test is implemented or incorporated, and that is subject to compatibility testing.</p>
Product Configuration	<p>A specific setting or instantiation of an Operating Mode.</p> <p>For example, a Product supporting an Operating Mode that permits user selection of an external encryption package may have a Product Configuration that links the Product to that encryption package.</p>
Rebuildable Tests	<p>Tests that must be built using an implementation specific mechanism. This mechanism must produce specification-defined artifacts. Rebuilding and running these tests against a known compatible implementation verifies that the mechanism generates compatible artifacts.</p>
Compatible Implementation (CI)	<p>A verified compatible implementation of a Specification.</p>
Resource	<p>A Computational Resource, a Location Resource, or a Security Resource.</p>
Rules	<p>These definitions and rules in this Compatibility Requirements section of this User's Guide.</p>
Runtime	<p>The Containers specified in the Specifications.</p>
Security Resource	<p>A security privilege or policy necessary for the proper execution of the Test Suite.</p> <p>For example, the user executing the Test Suite will need the privilege to access the files and network resources necessary for use of the Product.</p>

Term	Definition
Specifications	<p>The documents produced through the Jakarta EE Specification Process (JESP) that define a particular Version of a Technology.</p> <p>The Specifications for the Technology Under Test are referenced later in this chapter.</p>
Technology	Specifications and one or more compatible implementations produced through the Jakarta EE Specification Process (JESP).
Technology Under Test	Specifications and a compatible implementation for Jakarta Platform, Enterprise Edition Version 11, Web Profile.
Test Suite	The requirements, tests, and testing tools distributed by the Maintenance Lead as applicable to a given Version of the Technology.
Version	A release of the Technology, as produced through the Jakarta EE Specification Process (JESP).

Rules for Jakarta Platform, Enterprise Edition Version 11 Products

The following rules apply for each implementation:

EE-WP1 The Product must be able to satisfy all applicable compatibility requirements, including passing all required TCK tests.

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.

EE-WP1.1 Each implementation must have at least one Product Configuration that can be used to pass all required TCK Tests, although such configuration may need adjustment (e.g. whether statically or via administrative tooling).

EE-WP1.2 An implementation may have mode(s) that provide compatibility with previous Jakarta EE versions.

EE-WP1.3 An API Definition Product is exempt from all functional testing requirements defined here, except the signature tests.

EE-WP2 Some Conformance Tests may have properties that may be changed. Properties that can be changed are identified in the configuration interview. Properties that can be changed are identified in the `JavaTest Environment (ts.jte)` files in the `bin` directory of 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 Conformance Test may be altered in any way without prior written permission. Any such allowed alterations to the Conformance Tests will be provided via the Jakarta EE Specification Project website and apply to all vendor compatible implementations.

EE-WP3 The testing tools supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.

EE-WP4 The Excluded Tests associated with the Test Suite cannot be modified.

EE-WP5 The Maintenance Lead may define exceptions to these Rules. Such exceptions would be made available as above, and will apply to all vendor implementations.

EE-WP6 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.

EE-WP7 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.

EE-WP7.1 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.

EE-WP7.2 A Product may provide a newer version of an Endorsed Standard. Upon request, the Maintenance Lead will make available alternate Conformance Tests as necessary to conform with such newer version of an Endorsed Standard. Such alternate tests will be made available to and apply to all implementers. If a Product provides a newer version of an Endorsed Standard, the version of the Endorsed Standard supported by the Product must be Documented.

EE-WP7.3 The Maintenance Lead may authorize the use of newer Versions of a Technology included in the Technology Under Test. A Product that provides a newer Version of a Technology must meet the Compatibility Requirements for that newer Version, and must Document that it supports the newer Version.

EE-WP8 Except for tests specifically required by this TCK to be rebuilt (if any), the binary Conformance Tests supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.

EE-WP9 The functional programmatic behavior of any binary class or interface must be that defined by the Specifications.

EE-WP9.1 A Product may contain Operating Modes that meet all of these requirements, except Rule EE-WP9, provided that:

1. The Operating Modes must not violate the Java Platform, Standard Edition Rules.
2. Some Product Configurations of such Operating Modes may provide only a subset of the functional programmatic behavior required by the Specifications. The behavior of applications that use more than the provided subset, when run in such Product Configurations, is unspecified.
3. The functional programmatic behavior of any binary class or interface in the above defined subset must be that defined by the Specifications.
4. Any Product Configuration that invokes this rule must be clearly Documented as not fully meeting the requirements of the Specifications.

EE-WP10 Each Container must make technically accessible all Java SE Runtime interfaces and functionality, as defined by the Specifications, to programs running in the Container, except only as specifically exempted by these Rules.

EE-WP10.1 Containers may impose security constraints, as defined by the Specifications.

EE-WP11 A web Container must report an error, as defined by the Specifications, when processing a Jakarta Server Page that does not conform to the Specifications.

EE-WP12 The presence of a Java language comment or Java language directive in a Jakarta Server Page that specifies "java" as the scripting language, when processed by a web Container, must not cause the functional programmatic behavior of that Jakarta Server Page to vary from the functional programmatic behavior of that Jakarta Server Page in the absence of that Java language comment or Java language directive.

EE-WP13 The contents of any fixed template data (defined by the Specifications) in a Jakarta Server Page, when processed by a web Container, must not affect the functional programmatic behavior of that Jakarta Server Page, except as defined by the Specifications.

EE-WP14 The functional programmatic behavior of a Jakarta Server Page that specifies "java" as the scripting language must be equivalent to the functional programmatic behavior of the Jakarta Server Page Implementation Class

constructed from that Jakarta Server Page.

EE-WP15 A Deployment Tool must report an error when processing a Configuration Descriptor that does not conform to the Specifications.

EE-WP16 The presence of an XML comment in a Configuration Descriptor, when processed by a Deployment Tool, must not cause the functional programmatic behavior of the Deployment Tool to vary from the functional programmatic behavior of the Deployment Tool in the absence of that comment.

EE-WP17 A Deployment Tool must report an error when processing a Jakarta Enterprise Beans deployment descriptor that includes an Jakarta Enterprise Beans QL expression that does not conform to the Specifications.

EE-WP18 The Runtime must report an error when processing a Configuration Descriptor that does not conform to the Specifications.

EE-WP19 The presence of an XML comment in a Configuration Descriptor, when processed by the Runtime, must not cause the functional programmatic behavior of the Runtime to vary from the functional programmatic behavior of the Runtime in the absence of that comment.

EE-WP20 Compatibility testing for the Jakarta EE 11 Web Profile consists of running the tests for the technologies defined in [Jakarta EE 11 Technologies Required for Jakarta EE 11 Platform Compatibility](#) .

EE-WP21 Compliance testing for Jakarta EE 11 Web Profile consists of running the Jakarta EE 11 Web Profile TCK tests and the following component Technology Compatibility Kits (TCKs). Version details are defined in the Platform EE Specification document (<https://jakarta.ee/specifications/webprofile/11/>), see heading *Web Profile Definition*, sub-heading *Required Components*:

- Jakarta Authentication
- Jakarta Concurrency
- Jakarta Contexts and Dependency Injection
- Jakarta Data
- Jakarta Debugging Support for Other Languages
- Jakarta Dependency Injection
- Jakarta Faces
- Jakarta Pages
- Jakarta JSON Binding
- Jakarta JSON Processing
- Jakarta RESTful Web Services
- Jakarta Security
- Jakarta Servlet
- Jakarta Tags (contained within the platform TCK)
- Jakarta Transactions
- Jakarta Validation
- Jakarta WebSocket

See [Additional Jakarta EE 11 Platform TCK Requirements](#) for additional information on the component specification TCKs.

In addition to the compatibility rules outlined in this TCK User's Guide, Jakarta EE 11 implementations must also adhere to all the compatibility rules defined in the User's Guides of the aforementioned TCKs.

EE-WP21.1 If the Jakarta EE 11 Web Profile implementation uses a runtime which has already been validated by the

Technology Compatibility Kit, the Jakarta EE 11 Web Profile implementation may use result of such validation to claim its compliance with the Technology Compatibility Kit.

Jakarta Platform, Enterprise Edition Version 11 Web Profile Test Appeals Process

See [TCK Test Appeals Steps](#) for the Jakarta Platform, Enterprise Edition Version 11 Web Profile Test Appeals Process.

Specifications for Jakarta Platform, Enterprise Edition Version 11, Web Profile

The Specifications for Jakarta Platform, Enterprise Edition 11, Web Profile are found on the Eclipse Foundation, Jakarta EE Specification web site at <https://jakarta.ee/specifications/webprofile/11/>. You may also find information available from the EE4J Jakarta EE Platform project page, at <https://projects.eclipse.org/projects/ee4j.jakartaee-platform>.

Libraries for Jakarta Platform, Enterprise Edition Version 11, Web Profile

The following location provides a list of packages that constitute the required class libraries for the full Java EE 11 platform:

<https://projects.eclipse.org/projects/ee4j.jakartaee-platform>

The following list constitutes the subset of Jakarta EE 11 packages that are required for the Jakarta EE 11 Web Profile:

- jakarta.annotation
- jakarta.annotation.security
- jakarta.annotation.sql
- jakarta.decorator
- jakarta.data
- jakarta.data.exceptions
- jakarta.data.metamodel
- jakarta.data.page
- jakarta.data.repository
- jakarta.data.spi
- jakarta.ejb
- jakarta.ejb.embeddable (removed from Jakarta EE 11 Platform but still part of Jakarta Enterprise Beans 4.0)
- jakarta.ejb.spi
- jakarta.el
- jakarta.enterprise.context
- jakarta.enterprise.context.spi
- jakarta.enterprise.event
- jakarta.enterprise.inject
- jakarta.enterprise.inject.spi
- jakarta.enterprise.util
- jakarta.faces
- jakarta.faces.application
- jakarta.faces.bean
- jakarta.faces.component
- jakarta.faces.component.behavior

- jakarta.faces.component.html
- jakarta.faces.component.visit
- jakarta.faces.context
- jakarta.faces.convert
- jakarta.faces.el
- jakarta.faces.event
- jakarta.faces.flow
- jakarta.faces.flow.builder
- jakarta.faces.lifecycle
- jakarta.faces.model
- jakarta.faces.render
- jakarta.faces.validator
- jakarta.faces.view
- jakarta.faces.view.facelets
- jakarta.faces.webapp
- jakarta.inject
- jakarta.interceptor
- jakarta.json
- jakarta.json.spi
- jakarta.json.stream
- jakarta.persistence
- jakarta.persistence.criteria
- jakarta.persistence.metamodel
- jakarta.persistence.spi
- jakarta.servlet
- jakarta.servlet.annotation
- jakarta.servlet.descriptor
- jakarta.servlet.http
- jakarta.servlet.jsp
- jakarta.servlet.jsp.el
- jakarta.servlet.jsp.jstl.core
- jakarta.servlet.jsp.jstl.fmt
- jakarta.servlet.jsp.jstl.sql
- jakarta.servlet.jsp.jstl.tlv
- jakarta.servlet.jsp.tagext
- jakarta.transaction
- javax.transaction.xa
- jakarta.validation
- jakarta.validation.bootstrap
- jakarta.validation.constraints
- jakarta.validation.constraintvalidation
- jakarta.validation.executable
- jakarta.validation.groups

- jakarta.validation.metadata
- jakarta.validation.spi
- jakarta.websocket
- jakarta.websocket.server
- jakarta.ws.rs
- jakarta.ws.rs.client
- jakarta.ws.rs.container
- jakarta.ws.rs.core
- jakarta.ws.rs.ext
- jakarta.json.bind
- jakarta.json.bind.adapter
- jakarta.json.bind.annotation
- jakarta.json.bind.config
- jakarta.json.bind.serializer
- jakarta.json.bind.spi
- jakarta.security.enterprise
- jakarta.security.enterprise.authentication.mechanism.http
- jakarta.security.enterprise.credential
- jakarta.security.enterprise.identitystore

Installation

This chapter explains how to install the Jakarta EE 11 Platform TCK software and perform a sample test run to verify your installation and familiarize yourself with the TCK. Installation instructions are provided for Eclipse GlassFish 8.0, a Ratifying Compatible Implementation (CI) of Jakarta EE. If you are using another compatible implementation, refer to instructions provided with that implementation.

After installing the software according to the instructions in this chapter, proceed to [Setup and Configuration](#) for instructions on configuring your test environment.

This chapter covers the following topics:

- [Installing the Jakarta EE 11 Compatible Implementation](#)
- [Installing the Jakarta EE 11 Platform TCK](#)
- [Verifying Your Installation \(Optional\)](#)

Installing the Jakarta EE 11 Compatible Implementation

How to install the Jakarta EE 11 CI, Eclipse GlassFish 8.0

Before You Begin

If a Jakarta EE 11 Compatible Implementation (CI) is already installed, it is recommended that you shut it down and start with a new, clean CI installation.

1. Install the Java SE 17 JDK bundle, if it is not already installed.
Refer to the JDK installation instructions for details. The JDK bundle can be downloaded from <https://adoptium.net/temurin/releases/>
2. Create or change to the directory in which you will install the Jakarta EE 11 CI.
3. Copy or download the Jakarta EE 11 CI, for example, Eclipse GlassFish 8.0.
4. Unzip the Jakarta EE 11 CI bundle.
5. For Eclipse GlassFish 8.0, set the following environment variables:
 - JAKARTAE_HOME to the CI directory you just created
 - JAVA_HOME to the JDK you want to use
6. Start the Jakarta EE 11 CI, Eclipse GlassFish 8.0, by executing the following command:

```
<JAKARTAE_HOME>/bin/asadmin start-domain
```

Installing the Jakarta EE 11 Platform TCK

Complete the following procedure to install the Jakarta EE 11 Platform TCK on a system running the Solaris, GNU/Linux, or Windows operating system.



When installing in the Windows environment, the Jakarta EE 11 CI, Java SE JDK, and TCK should be installed on the same drive. If you must install these components on different drives, be sure to change the `ri.applicationRoot` and `slas.applicationRoot` properties as needed in the `<TS_HOME>/bin/ts.jte` TCK configuration file. See [Windows-Specific Properties](#) for more information.

1. Copy or download the Jakarta EE TCK 11 software.
2. Change to the directory in which you want to install the Jakarta EE 11 TCK software and use the `unzip` command to

extract the bundle:

```
cd install_directory
unzip jakartaeeetck-nnn-dist.zip
```

This creates the `jakartaeeetck` directory. The path to the `/jakartaeeetck` directory will be referenced as `TS_HOME`.

3. Set the `TS_HOME` environment variable to point to the `jakartaeeetck` directory.

After you complete the installation, follow the directions in [Setup and Configuration](#) to set up and configure the Jakarta EE 11 Platform TCK test suite.

Verifying Your Installation (Optional)

You can verify your installation by running the `VerifyHashes.java` file found in the artifacts directory. This file validates the MD5 hashes of the artifacts in the distribution. To run the file, execute the following command:

```
cd jakartaeeetck/artifacts
java VerifyHashes.java
```

When run with no arguments this downloads the Jakarta EE staging repository artifacts and validates them against the hashes of the artifacts in the distribution. If you want to validate the artifacts in another repo, pass in the URL to the `jakarta.tck` groupId in the repository. For example:

```
java VerifyHashes.java https://repo1.maven.org/maven2/jakarta/tck/
```

Setup and Configuration

This chapter describes how to set up the Jakarta EE 11 Platform TCK test suite and configure it to work with your test environment. It is recommended that you first set up the testing environment using the Jakarta EE 11 CI and then with your Jakarta EE 11 server.

This chapter includes the following topics:

- [Allowed Modifications](#)
- [Configuring the Test Environment](#)
- [Configuring a Jakarta EE 11 Server](#)
- [Modifying Environment Settings for Specific Technology Tests](#)

Allowed Modifications

You can modify the following test suite components only:

- Your implementation of the porting package
- `ts.jte` environment file
- The vendor-specific SQL files in `<TS_HOME>/bin/sql`
- Any files in `<TS_HOME>/bin` and `<TS_HOME>/bin/xml`

Configuring the Test Environment

The instructions in this section and in [Configuring Your Application Server as the VI](#) step you through the configuration process for the Linux and Microsoft Windows.

The primary location of any configuration settings you are likely to make is the `<TS_HOME>/bin/ts.jte` environment file. You may also want to modify the `initdb.xml` Ant configuration files and the vendor-specific SQL files. These two files contain configuration for the Jakarta EE 11 CI and its associated database in order to pass the TCK. An Implementer may choose to implement these targets to work with their server environment to perform the steps described in [Configuring Your Application Server as the VI](#).



The `<TS_HOME>/bin/ts.jte` environment file contains many properties that are no longer in use in the TCK. While we have transitioned from the JavaTest framework used in Jakarta EE 10 and earlier in Jakarta EE 11, there are still remnants in the EE 11 TCK, and these do use some properties in the `ts.jte` file.

The full list of `ts.jte` file properties that need a value is given in [Complete List of 'ts.jte' Properties](#).

Before You Begin

In these instructions, variables in angle brackets need to be expanded for each platform. For example, `<TS_HOME>` becomes `$TS_HOME` on Solaris/Linux and `%TS_HOME%` on Windows. In addition, the forward slashes (/) used in all of the examples need to be replaced with backslashes (\) for Windows.

1. Identify the software pieces and assemble them into the Jakarta EE 11 platform to be tested for certification.
2. Implement the porting package APIs.

Some functionality in the Jakarta EE 11 platform is not completely specified by an API. To handle this situation, the Jakarta EE 11 Platform TCK test suite defines a set of interfaces which serve to abstract any implementation-specific code. You must create your own implementations of the porting package interfaces to work with your particular

- Jakarta EE 11 server environment. See [Implementing the Porting Package](#) for additional information about the porting APIs. API documentation for the porting package interfaces is available in the <TS_HOME>/docs/api directory.
3. Set up the Jakarta Platform, Enterprise Edition Compatible Implementation (CI) server. See [Configuring the Jakarta EE 11 CI as the VI](#) for a list of the modifications that must be made to run CTS against the Jakarta EE 11 CI.
 4. Set up the vendor's Jakarta EE 11 server implementation (VI). See [Configuring Your Application Server as the VI](#) for a list of the modifications that must be made to run CTS against the vendor's Jakarta EE 11 server.
 5. Validate your configuration. Run the sample tests provided. If the tests pass, your basic configuration is valid. See <<[validating-your-test-configuration]>> for information about using Maven/Arquillian to run the sample tests.
 6. Run the TCK tests. See [Executing Tests](#) for information about using Maven to start running tests.
 7. Generate

Generate a Client Certificate

Some tests require a client certificate to be installed on the Jakarta EE 11 server. The following steps describe how to generate a client certificate:

```
keytool -genkeypair -alias cts -keyalg RSA -keysize 2048 -validity 365 -keystore clientcert.p12 -dname "CN=CTS, OU=Eclipse Foundation, O=Jakarta EE" -keypass changeit -storepass changeit
```

```
Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 365 days  
for: CN=CTS, OU=Eclipse Foundation, O=Jakarta EE
```

```
keytool -exportcert -keystore clientcert.p12 -alias cts -keypass changeit -storepass changeit -file cts_cert.cer
```

The values used here match defaults used in the `ts.jte` file and example GlassFish 8.0 runners. Examples and default configurations expect that the keystore and certificate are located in your <TS_HOME>/bin/certificates directory.

```
-Djava.protocol.handler.pkgs=javax.net.ssl  
-Djavax.net.ssl.keyStore=${ts.home}/certificates/clientcert.p12  
-Djavax.net.ssl.keyStorePassword=changeit  
-Djavax.net.ssl.trustStore=<path_to_server_trustStore>
```

Configuring a Jakarta EE 11 Server

This section describes how to configure the Jakarta EE 11 server under test. You can run the TCK tests against the Jakarta EE 11 CI or your own Jakarta Platform, Enterprise Edition server. When performing interoperability (interop) tests or web service-based tests, you will be running two Jakarta EE 11 CI servers, one of which must be a Jakarta EE 11 CI using, or configured to use a database. For example, Eclipse GlassFish 8.0 is bundled and configured to use the Apache Derby database.

For the purposes of this section, it is useful to clarify three terms as they are used here:

- Compatible Implementation (CI): Jakarta EE 11 CI, for example, GlassFish 8.0
- Vendor Implementation (VI): Jakarta EE 11 implementation from a vendor wanting to certify; typically, the goal of running the TCK is to certify a Jakarta EE 11 VI; in some cases, for purposes of familiarizing yourself with TCK, you may choose to run the Jakarta EE 11 CI as the VI
- Bundled Derby: Apache Derby database bundled with the Jakarta EE 11 CI, Eclipse GlassFish 8.0

Jakarta Platform, Enterprise Edition Server Configuration Scenarios

There are three general scenarios for configuring Jakarta EE 11 servers for Jakarta EE 11 Platform TCK testing (Note: in the following images, Java EE refers to Jakarta EE. RI should be replaced with CI for Compatible Implementation):

- Configure the Jakarta EE 11 CI as the server under test



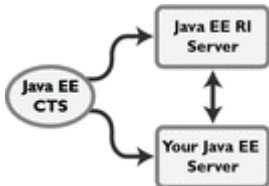
Use the Jakarta EE 11 CI as the Jakarta EE 11 VI; you may want to do this as a sanity check to make sure you are comfortable using the Jakarta EE 11 TCK against a known standard CI with certified sample applications before proceeding with tests against your Jakarta EE 11 VI. See [Configuring the Jakarta EE 11 CI as the VI](#) for instructions.

- Configure your Jakarta EE 11 VI as Server Under Test



This is the primary goal of using the Jakarta EE 11 Platform TCK; you will eventually need to configure the Jakarta EE 11 implementation you want to certify. See [Configuring Your Application Server as the VI](#) for instructions.

- Configure two Jakarta EE 11 servers for the purpose of interop testing



In terms of the Jakarta EE 11 Platform TCK, all TCK configuration settings are made in the `<TS_HOME>/bin/ts.jte` file. When configuring a Jakarta EE 11 server, the important thing is to make sure that the settings you use for your server match those in the `ts.jte` file.

These configuration scenarios are described in the following sections.

Configuring the Jakarta EE 11 CI as the VI

To configure the Jakarta EE 11 CI as the server under test (that is, to use the Jakarta EE 11 CI as the VI) follow the steps listed below. In this scenario, the goal is simply to test the Jakarta EE 11 CI against the CTS for the purposes of familiarizing yourself with TCK test procedures. You may also want to refer to the Quick Start guides included with the Jakarta EE 11 TCK for similar instructions.

1. Set server properties in your `<TS_HOME>/bin/ts.jte` file to suit your test environment. Be sure to set the following properties:
 - a. Set the `webServerHost` property to the name of the host on which your Web server is running that is configured with the CI. The default setting is `localhost`.
 - b. Set the `webServerPort` property to the port number of the host on which the Web server is running and configured with the CI. The default setting is `8001`.
 - c. Set the database-related properties in the `<TS_HOME>/bin/ts.jte` file. [Database Properties in ts.jte](#) lists the names and descriptions for the database properties you need to set.
2. Install the Jakarta EE 11 CI and configure basic settings, as described in [Installation](#)
3. Start the Jakarta EE 11 CI application server. Refer to the application server documentation for complete instructions.
4. Change to the `<TS_HOME>/bin` directory.
5. Start your backend database.
6. Initialize your backend database. Refer to [Configuring Your Backend Database](#)
7. Continue on to [Executing Tests](#) for instructions on running tests.



If you are using MySQL or MS SQL Server as your backend database, see [Backend Database Setup](#) for additional database setup instructions.

Configuring Your Application Server as the VI

To use a Jakarta EE 11 server other than the Jakarta EE 11 CI, follow the steps below.

1. Set server properties in your `<TS_HOME>/bin/ts.jte` file to suit your test environment. Be sure to set the following properties:
 - a. Set the `webServerHost` property to the name of the host on which your Web server is running that is configured with the CI.
The default setting is `localhost`.
 - b. Set the `webServerPort` property to the port number of the host on which the Web server is running and configured with the CI.
The default setting is `8001`.
 - c. Set the `porting.ts.url.class` property to your porting implementation class that is used for obtaining URLs. See [Porting Package APIs](#) for more information.
 - d. Set the database-related properties in the `<TS_HOME>/bin/ts.jte` file. [Database Properties in ts.jte](#) lists the names and descriptions for the database properties you need to set.
2. Install the Jakarta Platform, Enterprise Edition VI and configure basic settings.

Whichever configuration method you choose, make sure that all configuration steps in this procedure are completed as shown. . Install and configure a database for the server under test. . Start your database. . Initialize your database for TCK tests.

Refer to [Configuring Your Backend Database](#) for detailed database configuration and initialization instructions and a list of database-specific initialization targets.

1. Start your Jakarta EE 11 server.
2. Set up required users and passwords.
 - a. Set up database users and passwords that are used for JDBC connections.
The Jakarta EE 11 Platform TCK requires several user names, passwords, and user-to-role mappings. These need to match those set in your `ts.jte` file. By default, `user1`, `user2`, `user3`, `password1`, `password2`, and `password3` are set to `cts1`.
 - b. Set up users and passwords for your Jakarta Platform, Enterprise Edition server.
For the purpose of running the TCK test suite, these should be set as follows:

Table 2. User Password Groups

User	Password	Groups
j2ee_vi	j2ee_vi	staff
javajoe	javajoe	guest
j2ee	j2ee	staff, mgr, asadmin
j2ee_ri	j2ee_ri	staff

1. Make sure that the appropriate JDBC 4.1-compliant database driver class, any associated database driver native libraries, and the correct database driver URL are available.

2. Configure your Jakarta Platform, Enterprise Edition server to use the appropriate JDBC logical name (jdbc/DB1) when accessing your database server.
3. Configure your Jakarta EE 11 server to use the appropriate logical name (jdbc/DBTimer) when accessing your Jakarta Enterprise Beans timer.
4. Provide access to a JNDI lookup service.
5. Provide access to a Web server.
6. Provide access to a Jakarta Mail server that supports the SMTP protocol. (Full Platform only)
7. Install server certificates



This step needs to install server side certificates for interoperability testing; that is, it installs the CI's server certificate to VI and VI's server certificate into the CI. This step is necessary for mutual authentication tests in which both the server and client authenticate to each other.

8. Install the client-side certificate in the trustStore on the Jakarta EE 11 server.
Certificates are located <TS_HOME>/bin/certificates. Use the certificate that suits your environment.
 - a. `cts_cert.cer`: For importing the TCK client certificate into a truststore
 - b. `clientcert.p12`: Contains TCK client certificate in pkcs12 format
9. Make the appropriate transaction interoperability setting on the Jakarta EE 11 server and the server that is running the Jakarta EE 11 CI.
10. If necessary, refer to the sections later in this chapter for additional configuration information you may require for your particular test goals.
11. Restart your Jakarta EE 11 server.
12. Install the Jakarta EE 11 CI.
13. Set the following properties in your <TS_HOME>/bin/ts.jte file.
The current values should be saved since they will be needed later in this step.
 - Set the `javaee.home.uri` property to the location where the Jakarta EE 11 CI is installed.
14. Continue on to <<executing-tests>.

Modifying Environment Settings for Specific Technology Tests

Before you can run any of the technology-specific Jakarta EE 11 Platform TCK tests, you must supply certain information that JavaTest needs to run the tests in your particular environment. This information exists in the <TS_HOME>/bin/ts.jte environment file. This file contains sets of name/value pairs that are used by the tests. You need to assign a valid value for your environment for all of the properties listed in the sections that follow.



This section only discusses a small subset of the properties you can modify. Refer to the [Complete List of 'ts.jte' Properties](#) for what other properties in the `ts.jte` file may be relevant for your particular test environment.

This section includes the following topics:

- [Test Harness Setup](#)
- [Windows-Specific Properties](#)
- [Jakarta WebSocket Test Setup](#)
- [JDBC Test Setup \(Full Platform Only\)](#)
- [Jakarta Mail Test Setup \(Full Platform Only\)](#)
- [Jakarta Connector Test Setup \(Full Platform Only\)](#)

- [XA Test Setup](#)
- [Jakarta Enterprise Beans 4.0 Test Setup](#)
- [Jakarta Enterprise Beans Timer Test Setup](#)
- [Jakarta Persistence API Test Setup](#)
- [Jakarta Messaging Test Setup \(Full Platform Only\)](#)
- [Signature Test Setup](#)
- [Backend Database Setup](#)

Test Harness Setup

Verify that the following properties, which are used by the test harness, have been set in the `<TS_HOME>/bin/ts.jte` file:

```
harness.temp.directory=<TS_HOME>/tmp①
harness.log.port=2000②
harness.log.traceflag=[true | false]③
porting.ts.login.class.1=<vendor-login-class>④
porting.ts.url.class.1=<vendor-url-class>
porting.ts.jms.class.1=<vendor-jms-class>
porting.ts.tsURLConnection.class.1=<vendor-URLConnection-class>
```

- ① The `harness.temp.directory` property specifies a temporary directory that the harness creates and to which the TCK harness and tests write temporary files. The default setting should not need to be changed.
- ② The `harness.log.port` property specifies the port that server components of the tests use to send logging output back to the test harness. If the default port is not available on the machine running the test harness, you must edit this property and set it to an available port. The default setting is `2000`.
- ③ The `harness.log.traceflag` property is used to turn on or turn off verbose debugging output for the tests. The value of the property is set to `false` by default. Set the property to `true` to turn debugging on.
- ④ See [\[tsjte-porting-classes\]](#) for more information about these classes.

Complete List of ‘ts.jte’ Properties

These are the properties that need to have a value in the `ts.jte` file provided to the test runner:

- `s1as`
- `s1as.modules`
- `Driver`
- `authpassword`
- `authuser`
- `binarySize`
- `bin.dir`
- `cofSize`
- `cofTypeSize`
- `db.dml.file`
- `db.supports.sequence`
- `db1`
- `db2`
- `DriverManager`
- `ftable`
- `generateSQL`

- harness.log.port
- harness.log.traceflag
- harness.socket.retry.count
- harness.temp.directory
- imap.port
- iofile
- java.naming.factory.initial
- javaee.level
- javamail.mailbox
- javamail.password
- javamail.protocol
- javamail.root.path
- javamail.server
- javamail.username
- jdbc.db
- jms_timeout
- jstl.db.user
- jstl.db.password
- log.file.location
- logical.hostname.servlet
- longvarbinarySize
- mailuser1
- optional.tech.packages.to.ignore
- org.omg.CORBA.ORBClass
- password
- password1
- platform.mode
- porting.ts.HttpURLConnection.class.1
- porting.ts.HttpURLConnection.class.2
- porting.ts.login.class.1
- porting.ts.login.class.2
- porting.ts.url.class.1
- porting.ts.url.class.2
- porting.ts.jms.class.1
- porting.ts.jms.class.2
- ptable
- rapassword1
- rapassword2
- rauser1
- rauser2
- securedWebServicePort
- sigTestClasspath
- smtp.port

- transport_protocol
- ts_home
- user
- user1
- varbinarySize
- variable.mapper
- vehicle_ear_name
- webServerHost
- webServerPort
- whitebox-anno_no_md
- whitebox-mdcomplete
- whitebox-mixedmode
- whitebox-multianno
- whitebox-notx
- whitebox-notx-param
- whitebox-permissiondd
- whitebox-tx
- whitebox-tx-param
- whitebox-xa
- whitebox-xa-param
- work.dir
- ws_wait

Many of these properties can simply be left to their default values. Those that need specific values are described in the relevant sections of the configuration chapter.

Windows-Specific Properties

When configuring the Jakarta EE 11 Platform TCK for the Windows environment, set the following properties in `<TS_HOME>/bin/ts.jte`:

- pathsep to semicolon (pathsep=;)
 - `s1as.applicationRoot` to the drive on which you have installed CTS (for example, `s1as.applicationRoot=C:`)
- When installing in the Windows environment, the Jakarta Platform, Enterprise Edition CI, JDK, and TCK should all be installed on the same drive. If you must install these components on different drives, also change the `ri.applicationRoot` property in addition to the `pathsep` and `s1as.applicationRoot` properties; for example:

```
ri.applicationRoot=C:
```



When configuring the CI and TCK for the Windows environment, never specify drive letters in any path properties in `ts.jte`.

Jakarta WebSocket Test Setup

Make sure that the following WebSocket property has been set in the `ts.jte` file:

```
ws_wait=[number_of_seconds]
```

The `ws_wait` property configures the wait time, in seconds, for the socket to send or receive a message. A multiple of 5 of this time is also used to test socket timeouts.

The Jakarta WebSocket tests also use the following properties: `webServerHost` and `webServerPort`. See [Configuring the Jakarta EE 11 CI as the VI](#) for more information about setting these properties.



The SSL related tests under `/ts/javaeetck/src/com/sun/ts/tests/websocket/platform/jakarta/websocket/server/handshakerequest/authenticatcdssl/` use self signed certificate bundled with the TCK bundle. These certificates are generated with `localhost` as the hostname and would work only when `orb.host` value is set to `localhost` in `ts.jte`. If the server's hostname is used instead of the `localhost`, the tests in this suite might fail with the below exception - `jakarta.websocket.DeploymentException: SSL handshake has failed`.

JDBC Test Setup (Full Platform Only)

The JDBC tests require you to set the timezone by modifying the `tz` property in the `ts.jte` file. On Solaris systems, you can check the timezone setting by looking in the file `/etc/default/init`. Valid values for the `tz` property are in the directory `/usr/share/lib/zoneinfo`. The default setting is `US/Eastern`. This setting is in `/usr/share/lib/zoneinfo/US`.



The `tz` property is only used for Linux configurations; it does not apply to Windows.

Jakarta Mail Test Setup (Full Platform Only)

Complete the following tasks before you run the Jakarta Mail tests:

1. Set the following properties in the `ts.jte` file:

```
mailuser1=[user@domain]
mailFrom=[user@domain]
mailHost=mailserver
javamail.password=password
```

- Set the `mailuser1` property to a valid mail address. Mail messages generated by the Jakarta Mail tests are sent to the specified address. This user must be created in the IMAP server.
- Set the `mailFrom` property to a mail address from which mail messages that the Jakarta Mail tests generate will be sent.
- Set the `mailHost` property to the address of a valid mail server where the mail will be sent.
- Set the `javamail.password` property to the password for `mailuser1`.

2. Populate your IMAP server with sample messages.

The `com.sun.ts.tests.javamail.ee.util.fpopulate` class included in the `javamail` test artifact can be run with a configuration like the following to accomplish this:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.2.1</version>
  <configuration>
    <mainClass>com.sun.ts.tests.javamail.ee.util.fpopulate</mainClass>
    <arguments>
      <argument>-s</argument>
      <argument>${mailboxFolder1}</argument>
      <argument>-d</argument>
      <argument>${destinationURL}</argument>
      <argument>-user</argument>
    </arguments>
  </configuration>
</plugin>
```

```

        <argument>${javamail.username}</argument>
        <argument>-password</argument>
        <argument>${javamail.password}</argument>
        <argument>-host</argument>
        <argument>${javamail.server}</argument>
        <argument>-port</argument>
        <argument>${imap.port}</argument>
        <argument>-protocol</argument>
        <argument>${javamail.protocol}</argument>
    </arguments>
</configuration>
<executions>
    <execution>
        <id>1-populate-mailbox</id>
        <goals>
            <goal>java</goal>
        </goals>
        <phase>pre-integration-test</phase>
    </execution>
</executions>
</plugin>

```

Jakarta RESTful Web Services Test Setup

Edit your <TS_HOME>/bin/ts.jte file and set the following environment variables:

1. Set the `servlet_adaptor` property to point to the Servlet adaptor class for the Jakarta RESTful Web Services implementation.
The default setting for this property is `org/glassfish/jersey/servlet/ServletContainer.class`, the servlet adaptor supplied in Jersey.
2. Set the `webServerHost` property to the name of the host on which your Web server is running that is configured with the CI.
3. Set the `webServerPort` property to the port number of the host on which the Web server is running and configured with the CI.

Jakarta Connector Test Setup (Full Platform Only)

The Jakarta Connector tests verify that a Jakarta EE 11 server correctly implements the Jakarta Connector V1.7 specification. The Connector compatibility tests ensure that your Jakarta EE 11 server still supports the Connector V1.0 functionality.

Your runner for the Jakarta Connector TCK tests needs to deploy the RAR files listed in [Extension Libraries](#) and create the required connection resources and connection pools used for the Connector tests. The `glassfish-runner/connector-platform-tck` example runner also performs several other tasks, such as creating required users and security mappings, setting appropriate JVM options, etc. that also are needed to run the Connector tests.

Extension Libraries

The following Connector files need to be deployed as part of VI server setup for the Jakarta Connector TCK platform tests:

- `whitebox-mixedmode.rar`
- `whitebox-tx-param.rar`
- `whitebox-multianno.rar`
- `whitebox-tx.rar`

- `whitebox-anno_no_md.rar`
- `whitebox-notx-param.rar`
- `whitebox-xa-param.rar`
- `whitebox-mdcomplete.rar`
- `whitebox-notx.rar`
- `whitebox-xa.rar`

These RAR artifacts are available in the platform TCK distribution artifacts directory. Each RAR file has a dependency on the `whitebox` extension library. The `jakarta.tck:whitebox` `whitebox.jar` file is a Shared Library that must be deployed as a separate entity that all the Jakarta Connector RAR files access. This extension library is needed to address classloading issues.

The RAR files that are used with Jakarta EE 11 Platform TCK test suite differ from those that were used in other test suites. Jakarta EE 11 Platform TCK no longer bundles the same common classes into every RAR file. Duplicate common classes have been removed and now exist in the `whitebox.jar` file, an Installed Library that is deployed and is made available before any other RAR files are deployed.

This was done to address the following compatibility issues:

- Portable use of Installed Libraries for specifying a resource adapter's shared libraries
See section EE.8.2.2 of the Jakarta EE 11 platform specification and section 20.2.0.1 in the Jakarta Connectors (formerly JCA) 1.7 specification, which explicitly state that the resource adapter server may employ the library mechanisms in Jakarta EE 11.
- Support application-based standalone connector accessibility
Section 20.2.0.4 of the Jakarta Connectors (formerly JCA) 1.7 Specification uses the classloading requirements that are listed in section 20.3 in the specification.

Connector Resource Adapters and Classloading

Jakarta EE 11 Platform TCK has scenarios in which multiple standalone RAR files that use the same shared library (for example, `whitebox.jar`) are referenced from an application component.

Each standalone RAR file gets loaded in its own classloader. Since the application component refers to more than one standalone RAR file, all the referenced standalone RAR files need to be made available in the classpath of the application component. In versions of the TCK prior to Java EE 5, since each standalone RAR file contained a copy of the `whitebox.jar` file, every time there was a reference to a class in the `whitebox.jar` file from a standalone RAR, the reference was resolved by using the private version of `whitebox.jar` (the `whitebox.jar` file was bundled in each standalone RAR file). This approach can lead to class type inconsistency issues.

Use Case Problem Scenario

Assume that RAR1 and RAR2 are standalone RAR files that are referred to by an application, where:

- RAR1's classloader has access to RAR1's classes and its copy of `whitebox.jar`. (RAR1's classloader contains RAR1's classes and `whitebox.jar`)
- RAR2's classloader has access to RAR2's classes and its copy of `whitebox.jar`. (RAR2's classloader contains RAR2's classes and `whitebox.jar`)

When the application refers to both of these RAR files, a classloader that encompasses both of these classloaders (thereby creating a classloader search order) is provided to the application. The classloader search order could have the following sequence: ([RAR1's Classloader: RAR1's classes and `whitebox.jar`], [RAR2's Classloader: RAR2's classes and `whitebox.jar`]).

In this scenario, when an application loads a class (for example, class `Foo`) in `whitebox.jar`, the application gets class `Foo` from RAR1's classloader because that is first in the classloader search order. However, when this is cast to a class (for example, `Foo` or a subclass of `Foo` or even a class that references `Foo`) that is obtained from RAR2's classloader (a sequence that is typically realized in a `ConnectionFactory` lookup), this would result in a class-cast exception.

The portable way of solving the issues raised by this use case problem scenario is to use installed libraries, as was described in section EE.8.2.2 in the Jakarta EE 10 platform specification. If both RAR files (RAR1 and RAR2) reference `whitebox.jar` as an installed library and the application server can use a single classloader to load this common dependency, there will be no type-related issues.

For Jakarta EE 11 and beyond, VI servers will need to configure the common `whitebox.jar` in whatever manner is supported by the server.

In the CI Eclipse GlassFish 8.0, `domain-dir/lib/applibs` is used as the Installed Library directory and is the location to which the `whitebox.jar` file gets copied.

Required Porting Package

The Jakarta EE 11 Platform TCK test suite treats the `whitebox.jar` dependency as an Installed Library dependency instead of bundling the dependency (or dependencies) with every RAR file.

It is necessary to identify the `whitebox.jar` to the connector server as an installed library. Because the `whitebox.jar` file depends on Jakarta EE APIs, one cannot simply put the `whitebox.jar` file into a `java.ext.dir` directory, which gets loaded by the VM extension classloader, because that mechanism does not allow the `whitebox.jar` file to support its dependencies on the Jakarta EE APIs. For this reason, the common `whitebox.jar` must support access to the Jakarta EE APIs.

Creating Security Mappings for the Connector RAR Files

The VI server needs to map Resource Adapter user information to existing user information in the CI.

For the Eclipse GlassFish 8.0 CI, the required mappings add a line to the `domain.xml` file, similar to the one shown below, and should include 6 of these mappings:

```
<jvm-options>-Dwhitebox-tx-map=cts1=j2ee</jvm-options>
<jvm-options>-Dwhitebox-tx-param-map=cts1=j2ee</jvm-options>
<jvm-options>-Dwhitebox-notx-map=cts1=j2ee</jvm-options>
<jvm-options>-Dwhitebox-notx-param-map=cts1=j2ee</jvm-options>
<jvm-options>-Dwhitebox-xa-map=cts1=j2ee</jvm-options>
<jvm-options>-Dwhitebox-xa-param-map=cts1=j2ee</jvm-options>
```

If the `rauser1` property has been set to `cts1` and the `user` property has been set to `j2ee` in the `ts.jte` file, the following mappings would be required in the connector runtime:

- For RA `whitebox-tx`, map `cts1` to `j2ee`
- For RA `whitebox-tx-param`, map `cts1` to `j2ee`
- For RA `whitebox-notx`, map `cts1` to `j2ee`
- For RA `whitebox-notx-param`, map `cts1` to `j2ee`
- For RA `whitebox-xa`, map `cts1` to `j2ee`
- For RA `whitebox-xa-param`, map `cts1` to `j2ee`

Creating Required Server-Side JVM Options

Create the required JVM options that enable user information to be set and/or passed from the `ts.jte` file to the server. The RAR files use some of the property settings in the `ts.jte` file.

To see some of the required JVM options for the server under test, see the `slas.jvm.options` property in the `ts.jte` file. The connector tests require that the following subset of JVM options be set in the server under test:

```
-Dj2eelogin.name=j2ee
-Dj2eelogin.password=j2ee
-Deislogin.name=cts1
-Deislogin.password=cts1
```

XA Test Setup

The XA tests reference some JDBCWhitebox name bindings that are created as part of the `config.vi` target but those name bindings are not tied to any JDBC RAR files. Instead, the following XA-specific connection pool JNDI names are referenced by the XA tests:

- `eis/JDBCwhitebox-xa`
- `eis/JDBCwhitebox-tx`
- `eis/JDBCwhitebox-notx`

These JNDI names need to point to a JDBC connection pool. Complete the following steps (create JDBC connection pools and JDBC resource elements, deploy the RAR files) to set up your environment to run the XA tests:

1. Create a JDBC connection pool with the following attributes:

- Set the resource type to `javax.sql.XADataSource`
- Set the `datasourceclassname` to `org.apache.derby.jdbc.EmbeddedXADataSource`
- Set the property to `DatabaseName=<Derby-location>:user=cts1:password=cts1`
- Set the connection pool name to `cts-derby-XA-pool`

For example, you could use the `asadmin` command line utility in the Jakarta EE 11 CI, Eclipse GlassFish 6.1 to create this connection pool:

```
asadmin create-jdbc-connection-pool --restype javax.sql.XADataSource \
--datasourceclassname org.apache.derby.jdbc.EmbeddedXADataSource \
--property 'DatabaseName=/tmp/DerbyDB:user=cts1:password=cts1' \
cts-derby-XA-pool
```

2. Create three JDBC connection pool elements (more specifically, the JDBC connection pool elements) with the following JNDI names:

- For the first connection pool element, set the connection pool id to `cts-derby-XA-pool` and the JNDI name to `eis/JDBCwhitebox-xa`
- For the second connection pool element, set the connection pool id to `cts-derby-XA-pool` and the JNDI name to `eis/JDBCwhitebox-tx`
- For the third connection pool element, set the connection pool id to `cts-derby-XA-pool` and the JNDI name to `eis/JDBCwhitebox-notx`

For example, you could use the `asadmin` command line utility in the Jakarta EE 11 CI to create the three connection pool elements:

```
asadmin asadmin create-jdbc-resource --connectionpoolid cts-derby-XA-pool \
eis/JDBCwhitebox-xa
asadmin create-jdbc-resource --connectionpoolid cts-derby-XA-pool \
```

```
eis/JDBCwhitebox-tx
asadmin create-jdbc-resource --connectionpoolid cts-derby-XA-pool \
eis/JDBCwhitebox-notx
```

If two or more JDBC resource elements point to the same connection pool element, they use the same pool connection at runtime. Jakarta EE 11 Platform TCK does reuse the same connection pool ID for testing the Jakarta EE 11 CI Eclipse GlassFish 8.0.

3. Make sure that the following EIS and RAR files have been deployed into your environment before you run the XA tests:
 - For the EIS resource adapter, deploy the standalone RAR files found in the TCK distribution artifacts directory. With the CI runner these files are deployed as part of pre-integration-test phase. The following `ts.jte` properties refer to the required common RAR files and define the JNDI bindings for the RAR connection factory as deployed by the VI server deployments.

```
whitebox-tx=java:comp/env/eis/whitebox-tx
whitebox-notx=java:comp/env/eis/whitebox-notx
whitebox-xa=java:comp/env/eis/whitebox-xa
whitebox-tx-param=java:comp/env/eis/whitebox-tx-param
whitebox-notx-param=java:comp/env/eis/whitebox-notx-param
whitebox-xa-param=java:comp/env/eis/whitebox-xa-param
whitebox-embed-xa=
"__SYSTEM/resource/ejb_Tsr#whitebox-xa#com.sun.ts.tests.common.connector.whitebox.TSConnectionFactory"
```

- The EIS RAR files are located TCK distribution artifacts directory. Deployment can either be done ahead of time or at runtime, as long as connection pools and resources are established prior to test execution. The XA tests make use of existing connector RAR files. Note that there are currently no `JDBCwhitebox` source files and no `JDBCwhitebox` RAR files.

Jakarta Enterprise Beans 4.0 Test Setup

This section explains special configuration that needs to be completed before running the Jakarta Enterprise Beans 4.0 DataSource and Stateful Timeout tests.

The Jakarta Enterprise Beans 4.0 DataSource tests do not test XA capability and XA support in a database product is typically not required for these tests. However, some Jakarta EE products could be implemented in such a way that XA must be supported by the database. For example, when processing the `@DataSourceDefinition` annotation or `<data-source>` descriptor elements in tests, a Jakarta EE product infers the datasource type from the interface implemented by the driver class. When the driver class implements multiple interfaces, such as `javax.sql.DataSource`, `javax.sql.ConnectionPoolDataSource`, or `javax.sql.XADataSource`, the vendor must choose which datasource type to use. If `javax.sql.XADataSource` is chosen, the target datasource system must be configured to support XA. Consult the documentation for your database system and JDBC driver for information that explains how to enable XA support.

To Configure the Test Environment to Run the Jakarta Enterprise Beans 4.0 Resource Tests (Full Platform Only)

Several `jakarta.tck:ejb30` test classes require bindings for `@Resource` injection of type `java.net.URL` as well as a custom JNDI resource.

The `com.sun.ts.tests.ejb30.bb.mdb.dest.jarwar.ClientTest` test has an `com.sun.ts.tests.ejb30.bb.mdb.dest.jarwar.Client` application client that expects a `Resource("url")` to be injected with a URL pointing to a servlet that is deployed in the web container. With the default `webServerHost` of `localhost` and `webServerPort` of `8080`, the URL should be set to `http://localhost:8080/mdbdest/TestServlet`.

Other tests using a `java.net.URL` simply expect a valid `http://localhost:8080`, or whatever your test environment

webServerHost/webServerPort map to.

The `com.sun.ts.tests.ejb30.bb.session.stateless.annotation.resource` package has an application client with EJBs that expect a `Resource("custom/dog")` to be injected with a custom resource of type `com.sun.ts.lib.deliverable.cts.resource.Dog`. The JNDI `javax.naming.spi.ObjectFactory` for the custom type is `com.sun.ts.lib.deliverable.cts.resource.DogFactory`.

With GlassFish 8.0 the runner for the `jakarta.tck:ejb30` tests uses these custom resource bindings to satisfy these requirements:

```
create-custom-resource --restype com.sun.ts.lib.deliverable.cts.resource.Dog --factoryclass
com.sun.ts.lib.deliverable.cts.resource.DogFactory custom/dog
create-custom-resource --restype java.net.URL --factoryclass
org.glassfish.resources.custom.factory.URLObjectFactory --property spec=http://localhost:8080 webServerURL
create-custom-resource --restype java.net.URL --factoryclass
org.glassfish.resources.custom.factory.URLObjectFactory --property spec=http://localhost:8080/mdbdest/TestServlet
webServerMdbtestURL
```

A GlassFish 8.0 specific descriptor is used to configure the custom resource bindings, e.g., this is the `sun-application-client.xml` for the `com.sun.ts.tests.ejb30.bb.mdb.dest.jarwar.ClientTest` test:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-application-client PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0 Application Client
5.0//EN"
"http://www.sun.com/software/appserver/dtds/sun-application-client_5_0-0.dtd">

<sun-application-client>
  <resource-ref>
    <res-ref-name>url</res-ref-name>
    <jndi-name>webServerMdbtestURL</jndi-name>
  </resource-ref>
  ...
</sun-application-client>
```

To Configure the Test Environment to Run the Jakarta Enterprise Beans 4.0 DataSource Tests

The EJB 3.2 DataSource tests under the following `jakarta.tck:ejb30` artifact packages may require you to update the `@DataSourceDefinition` used in the test class to match your database environment. You are allowed to recompile these tests with those changes before running them.

- `com/sun/ts/tests/ejb30/lite/packaging/war/datasource`
- `com/sun/ts/tests/ejb30/misc/datasource`
- `com/sun/ts/tests/ejb30/assembly/appres`

If your database vendor requires you to set any vendor-specific or less common DataSource properties, complete step [\[jdbc.datasource.props\]](#) and then complete step [\[configure_datasource_tests\]](#), as explained below.

1. Set any vendor-specific or less common datasource properties with the `jdbc.datasource.props` property in the `ts.jte` file.

The value of the property is a comma-separated array of name-value pairs, in which each property pair uses a "name=value" format, including the surrounding double quotes.

The value of the property must not contain any extra spaces.

For example:

```
jdbc.datasource.props="driverType=thin","name2=vale2"
```

2. Run the `configure.datasource.tests` Ant target to rebuild the Jakarta Enterprise Beans 4.0 DataSource Definition tests using the new database settings specified in the `ts.jte` file.

This step must be completed for Jakarta EE 11 and Jakarta EE 11 Web Profile testing.

To Configure the Test Environment to Run the Jakarta Enterprise Beans 4.0 Stateful Timeout Tests

The Jakarta Enterprise Beans 4.0 Stateful Timeout Tests in the following `jakarta.tck:ejb30` artifact test directories require special setup:

- `com/sun/ts/tests/ejb30/lite/stateful/timeout`
- `com/sun/ts/tests/ejb30/bb/session/stateful/timeout`
 1. Set the `javatest.timeout.factor` property in the `ts.jte` file to a value such that the JavaTest harness does not time out before the test completes.
A value of 2.0 or greater should be sufficient.
 2. Set the `test.ejb.stateful.timeout.wait.seconds` property, which specifies the minimum amount of time, in seconds, that the test client waits before verifying the status of the target stateful bean, to a value that is appropriate for your server.
The value of this property must be an integer number. The default value is 480 seconds. This value can be set to a smaller number (for example, 240 seconds) to speed up testing, depending on the stateful timeout implementation strategy in the target server.

Jakarta Enterprise Beans Timer Test Setup

Set the following properties in the `ts.jte` file to configure the Jakarta Enterprise Beans timer tests:

```
ejb_timeout=[interval_in_milliseconds]
ejb_wait=[interval_in_milliseconds]
```

- The `ejb_timeout` property sets the duration of single-event and interval timers. The default setting and recommended minimum value is 30000 milliseconds.
- The `ejb_wait` property sets the period for the test client to wait for results from the `ejbTimeout()` method. The default setting and recommended minimum value is 60000 milliseconds.

Jakarta EE 11 Platform TCK does not have a property that you can set to configure the date for date timers.

The timer tests use the specific `jndi-name jdbc`/DBTimer`` for the datasource used for container-managed persistence to support the use of an XA datasource in the Jakarta EE 11 timer implementation. For example:

```
<jdbc-resource enabled="true" jndi-name="jdbc/DBTimer"
  object-type="user" pool-name="cts-javadb-XA-pool" />
```

The test directories that use this datasource are:

```
ejb/ee/timer
ejb/ee/bb/entity/bmp/allowedmethodtest
ejb/ee/bb/entity/cmp20/allowedmethodtest
```

When testing against the Jakarta Platform, Enterprise Edition CI, Eclipse GlassFish 8.0, you must first start the Derby DB and initialize it in addition to any other database you may be using, as explained in [Configuring the Jakarta EE 11 CI as the VI](#)

Jakarta Persistence API Test Setup

The Jakarta Persistence API tests exercise the requirements as defined in the Jakarta Persistence API Specification. This specification defines a persistence context to be a set of managed entity instances, in which for any persistent identity

there is a unique entity instance. Within the persistence context, the entity instances and their life cycles are managed by the entity manager.

Within a Jakarta Platform, Enterprise Edition environment, support for both container-managed and application-managed entity managers is required. Application-managed entity managers can be Jakarta Transactions or resource-local. Refer to Chapter 7 of the Jakarta Persistence API Specification (<https://jakarta.ee/specifications/persistence/3.2>) for additional information regarding entity managers.



There are tests that install a custom Jakarta Persistence provider in the Jakarta Persistence API tests. The tests expect that the `log.file.location` from the `ts.jte` file has been propagated to a system property in the server environment. Normally this is automatically done by the TCK harness, but if your Jakarta Persistence integration causes the custom `jakarta.persistence.spi.PersistenceProvider` or `jakarta.persistence.spi.ProviderUtil` to initialize before the TCK harness, you may need to set the system property manually in the container configuration. If this is not set, it will default to a temporary file in the `'java.io.tmpdir'` and any setting for `log.file.location` in the `ts.jte` will be ignored in favor of the temporary file.

To Configure the Test Environment to Run the Jakarta Persistence Pluggability Tests

The Jakarta Persistence Pluggability tests under the `src/tcks/apis/persistence/persistence-inside-container/platform-tests/src/main/java/ee/jakarta/tck/persistence/ee/pluggability` directory ensure that a third-party persistence provider is pluggable, in nature.

After Java EE 7 TCK, the pluggability tests were rewritten to use a stubbed-out legacy JPA 2.1 implementation, which is located in the `src/tcks/apis/persistence/persistence-inside-container/common/src/main/java/ee/jakarta/tck/persistence/common/pluggability/altprovider` directory.

In Java EE 7 TCK, the Persistence API pluggability tests required special setup to run. This is no longer the case, since Jakarta EE 11 Platform TCK now enables the pluggability tests to be executed automatically along with all the other Persistence tests.

Enabling Second Level Caching Support

Jakarta Persistence supports the use of a second-level cache by the persistence provider. The `ts.jte` file provides a property that controls the TCK test suite's use of the second-level cache.

The `persistence.second.level.caching.supported` property is used to determine if the persistence provider supports the use of a second-level cache. The default value is `true`. If your persistence provider does not support second level caching, set the value to `false`.

Persistence Test Vehicles

The persistence tests are run in a variety of "vehicles" from which the entity manager is obtained and the transaction type is defined for use. There are six vehicles used for these tests:

- `stateless3`: Bean-managed stateless session bean using a Jakarta Transactions `EntityManager`; uses `UserTransaction` methods for transaction demarcation (Full Platform Only)
- `stateful3`: Container-managed stateful session bean using `@PersistenceContext` annotation to inject Jakarta Transactions `EntityManager`; uses container-managed transaction demarcation with a transaction attribute (required) (Full Platform Only)

- `appmanaged`: Container-managed stateful session bean using `@PersistenceUnit` annotation to inject an `EntityManagerFactory`; the `EntityManagerFactory` API is used to create an Application-Managed Jakarta Transactions `EntityManager`, and uses the container to demarcate transactions (Full Platform Only)
- `appmanagedNoTx`: Container-managed stateful session bean using `@PersistenceUnit` annotation to inject an `EntityManagerFactory`; the `EntityManagerFactory` API is used to create an Application-Managed Resource Local `EntityManager`, and uses the `EntityTransaction` APIs to control transactions (Full Platform Only)
- `pmservlet`: Servlet that uses the `@PersistenceContext` annotation at the class level and then uses JNDI lookup to obtain the `EntityManager`; alternative to declaring the persistence context dependency via a `persistence-context-ref` in `web.xml` and uses `UserTransaction` methods for transaction demarcation
- `puservlet`: Servlet that injects an `EntityManagerFactory` using the `@PersistenceUnit` annotation to create a `ResourceLocal` `EntityManager`, and uses `EntityTransaction` APIs for transaction demarcation
- `none`: There are a new tests that do not use the EE10 vehicle concept. Rather they make use of existing Arquillian protocols based on servlets or rest endpoints.



For vehicles using a `RESOURCE_LOCAL` transaction type, be sure to configure a non-transactional resource with the logical name `jdbc/DB_no_tx`. Refer to the `ts.jte` file for information about the `jdbc.db` property.

GeneratedValue Annotation

The Jakarta Persistence API Specification also defines the requirements for the `GeneratedValue` annotation. The default for this annotation is `GenerationType.AUTO`. Per the specification, `AUTO` indicates that the persistence provider should pick an appropriate strategy for the particular database. The `AUTO` generation strategy may expect a database resource to exist, or it may attempt to create one.

The `db.supports.sequence` property is used to determine if a database supports the use of `SEQUENCE`. If it does not, this property should be set to `false` so the test is not run. The default value is `true`.

If the database under test is not one of the databases defined and supported by TCK, the user will need to create an entry similar to the one listed in [Example 5-1 GeneratedValue Annotation Test Table](#).

Example 5-1 GeneratedValue Annotation Test Table

```
DROP TABLE SEQUENCE;
CREATE TABLE SEQUENCE (SEQ_NAME VARCHAR(10), SEQ_COUNT INT, CONSTRAINT SEQUENCE_PK /
PRIMARY KEY (SEQ_NAME) );
INSERT into SEQUENCE(SEQ_NAME, SEQ_COUNT) values ('SEQ_GEN', 0) ;
```

You should add your own table to your chosen database DDL file provided prior to running these tests.

The `persistence.xml` file, which defines a persistence unit, contains the `unitName` `CTS-EM` for Jakarta Transactions entity managers. This corresponds to `jta-data-source`, `jdbc/DB1`, and to `CTS-EM-NOTX` for `RESOURCE_LOCAL` entity managers, which correspond to a non-`jta-data-source` `jdbc/DB_no_tx`.

Jakarta Messaging Test Setup (Full Platform Only)

This section explains how to set up and configure the Jakarta EE 11 Platform TCK test suite before running the Jakarta Messaging tests.



The client-specified values for `JMSDeliveryMode`, `JMSExpiration`, and `JMSPriority` must not be overridden when running the TCK Jakarta Messaging tests.

To Configure a Slow Running System

Make sure that the following property has been set in the `ts.jte` file:

```
jms_timeout=10000
```

This property specifies the length of time, in milliseconds, that a synchronous receipt operation will wait for a message. The default value of the property should be sufficient for most environments. If, however, your system is running slowly, and you are not receiving the messages that you should be, you need to increase the value of this parameter.

To Test Your Jakarta Messaging Resource Adapter

If your implementation supports Jakarta Messaging as a Resource Adapter, you must set the name of the `jmsra.name` property in the `ts.jte` file to the name of your Jakarta Messaging Resource Adapter. The default value for the property is the name of the Jakarta Messaging Resource Adapter in the Jakarta EE 11 CI.

If you modify the `jmsra.name` property, you must rebuild the Jakarta Messaging tests that use this property. You rebuild the tests by doing the following:

1. Change to the `TS_HOME/bin` directory.
2. Invoke the following Ant task:

```
ant rebuild.jms.rebuildable.tests
```

This rebuilds the tests under `TS_HOME/src/com/sun/ts/tests/jms/ee20/resourcedefs`.

To Create Jakarta Messaging Administered Objects

If you do not have an API to create Jakarta Messaging Administered objects, you can use the list that follows and manually create the objects. If you decide to create these objects manually, you need to provide a dummy implementation of the Jakarta Messaging porting interface, `TSJMSAdminInterface`.

The list of objects you need to manually create includes the following factories, queues, and topics.

- Factories:

```
jms/TopicConnectionFactory
jms/DURABLE_SUB_CONNECTION_FACTORY, clientId=cts
jms/MDBTACCESSTEST_FACTORY, clientId=cts1
jms/DURABLE_BMT_CONNECTION_FACTORY, clientId=cts2
jms/DURABLE_CMT_CONNECTION_FACTORY, clientId=cts3
jms/DURABLE_BMT_XCONNECTION_FACTORY, clientId=cts4
jms/DURABLE_CMT_XCONNECTION_FACTORY, clientId=cts5
jms/DURABLE_CMT_TXNS_XCONNECTION_FACTORY, clientId=cts6
jms/QueueConnectionFactory
jms/ConnectionFactory
```

- Queues:

```
MDB_QUEUE
```

MDB_QUEUE_REPLY
 MY_QUEUE
 MY_QUEUE2
 Q2
 QUEUE_BMT
 ejb_ee_bb_localaccess_mdbqaccesstest_MDB_QUEUE
 ejb_ee_deploy_mdb_ejblink_casesensT_ReplyQueue
 ejb_ee_deploy_mdb_ejblink_casesens_ReplyQueue
 ejb_ee_deploy_mdb_ejblink_casesens_TestBean
 ejb_ee_deploy_mdb_ejblink_scopeT_ReplyQueue
 ejb_ee_deploy_mdb_ejblink_scope_ReplyQueue
 ejb_ee_deploy_mdb_ejblink_scope_TestBean
 ejb_ee_deploy_mdb_ejblink_singleT_ReplyQueue
 ejb_ee_deploy_mdb_ejblink_single_ReplyQueue
 ejb_ee_deploy_mdb_ejblink_single_TestBean
 ejb_ee_deploy_mdb_ejblink_single_TestBeanBMT
 ejb_ee_deploy_mdb_ejbref_casesensT_ReplyQueue
 ejb_ee_deploy_mdb_ejbref_casesens_ReplyQueue
 ejb_ee_deploy_mdb_ejbref_casesens_TestBean
 ejb_ee_deploy_mdb_ejbref_scopeT_ReplyQueue
 ejb_ee_deploy_mdb_ejbref_scope_Cyrano
 ejb_ee_deploy_mdb_ejbref_scope_ReplyQueue
 ejb_ee_deploy_mdb_ejbref_scope_Romeo
 ejb_ee_deploy_mdb_ejbref_scope_Tristan
 ejb_ee_deploy_mdb_ejbref_singleT_ReplyQueue
 ejb_ee_deploy_mdb_ejbref_single_ReplyQueue
 ejb_ee_deploy_mdb_ejbref_single_TestBean
 ejb_ee_deploy_mdb_ejbref_single_TestBeanBMT
 ejb_ee_deploy_mdb_enventry_casesensT_ReplyQueue
 ejb_ee_deploy_mdb_enventry_casesens_CaseBean
 ejb_ee_deploy_mdb_enventry_casesens_CaseBeanBMT
 ejb_ee_deploy_mdb_enventry_casesens_ReplyQueue
 ejb_ee_deploy_mdb_enventry_scopeT_ReplyQueue
 ejb_ee_deploy_mdb_enventry_scope_Bean1_MultiJar
 ejb_ee_deploy_mdb_enventry_scope_Bean1_SameJar
 ejb_ee_deploy_mdb_enventry_scope_Bean2_MultiJar
 ejb_ee_deploy_mdb_enventry_scope_Bean2_SameJar
 ejb_ee_deploy_mdb_enventry_scope_ReplyQueue
 ejb_ee_deploy_mdb_enventry_singleT_ReplyQueue
 ejb_ee_deploy_mdb_enventry_single_AllBean
 ejb_ee_deploy_mdb_enventry_single_AllBeanBMT
 ejb_ee_deploy_mdb_enventry_single_BooleanBean
 ejb_ee_deploy_mdb_enventry_single_ByteBean
 ejb_ee_deploy_mdb_enventry_single_DoubleBean
 ejb_ee_deploy_mdb_enventry_single_FloatBean
 ejb_ee_deploy_mdb_enventry_single_IntegerBean
 ejb_ee_deploy_mdb_enventry_single_LongBean
 ejb_ee_deploy_mdb_enventry_single_ReplyQueue
 ejb_ee_deploy_mdb_enventry_single_ShortBean
 ejb_ee_deploy_mdb_enventry_single_StringBean
 ejb_ee_deploy_mdb_resref_singleT_ReplyQueue
 ejb_ee_deploy_mdb_resref_single_ReplyQueue
 ejb_ee_deploy_mdb_resref_single_TestBean
 ejb_ee_sec_stateful_mdb_MDB_QUEUE
 ejb_sec_mdb_MDB_QUEUE_BMT
 ejb_sec_mdb_MDB_QUEUE_CMT
 jms_ee_mdb_mdb_exceptQ_MDB_QUEUE_TXNS_CMT
 jms_ee_mdb_mdb_exceptQ_MDB_QUEUE_BMT
 jms_ee_mdb_mdb_exceptQ_MDB_QUEUE_CMT
 jms_ee_mdb_mdb_exceptT_MDB_QUEUE_TXNS_CMT
 jms_ee_mdb_mdb_exceptT_MDB_QUEUE_BMT
 jms_ee_mdb_mdb_exceptT_MDB_QUEUE_CMT
 jms_ee_mdb_mdb_msgHdrQ_MDB_QUEUE
 jms_ee_mdb_mdb_msgPropsQ_MDB_QUEUE
 jms_ee_mdb_mdb_msgTypesQ1_MDB_QUEUE
 jms_ee_mdb_mdb_msgTypesQ2_MDB_QUEUE
 jms_ee_mdb_mdb_msgTypesQ3_MDB_QUEUE

```
jms_ee_mdb_mdb_rec_MDB_QUEUE
jms_ee_mdb_sndQ_MDB_QUEUE
jms_ee_mdb_sndToQueue_MDB_QUEUE
jms_ee_mdb_mdb_synchrec_MDB_QUEUE
jms_ee_mdb_xa_MDB_QUEUE_BMT
jms_ee_mdb_xa_MDB_QUEUE_CMT
testQ0
testQ1
testQ2
testQueue2
fooQ
```

- Topics:

```
MY_TOPIC
MY_TOPIC2
TOPIC_BMT
ejb_ee_bb_localaccess_mdbtaccessstest_MDB_TOPIC
ejb_ee_deploy_mdb_ejblink_casesensT_TestBean
ejb_ee_deploy_mdb_ejblink_scopeT_TestBean
ejb_ee_deploy_mdb_ejblink_singleT_TestBean
ejb_ee_deploy_mdb_ejblink_singleT_TestBeanBMT
ejb_ee_deploy_mdb_ejbref_casesensT_TestBean
ejb_ee_deploy_mdb_ejbref_scopeT_Cyrano
ejb_ee_deploy_mdb_ejbref_scopeT_Romeo
ejb_ee_deploy_mdb_ejbref_scopeT_Tristan
ejb_ee_deploy_mdb_ejbref_singleT_TestBean
ejb_ee_deploy_mdb_ejbref_singleT_TestBeanBMT
ejb_ee_deploy_mdb_enventry_casesensT_CaseBean
ejb_ee_deploy_mdb_enventry_casesensT_CaseBeanBMT
ejb_ee_deploy_mdb_enventry_scopeT_Bean1_MultiJar
ejb_ee_deploy_mdb_enventry_scopeT_Bean1_SameJar
ejb_ee_deploy_mdb_enventry_scopeT_Bean2_MultiJar
ejb_ee_deploy_mdb_enventry_scopeT_Bean2_SameJar
ejb_ee_deploy_mdb_enventry_singleT_AllBean
ejb_ee_deploy_mdb_enventry_singleT_AllBeanBMT
ejb_ee_deploy_mdb_enventry_singleT_BooleanBean
ejb_ee_deploy_mdb_enventry_singleT_ByteBean
ejb_ee_deploy_mdb_enventry_singleT_DoubleBean
ejb_ee_deploy_mdb_enventry_singleT_FloatBean
ejb_ee_deploy_mdb_enventry_singleT_IntegerBean
ejb_ee_deploy_mdb_enventry_singleT_LongBean
ejb_ee_deploy_mdb_enventry_singleT_ShortBean
ejb_ee_deploy_mdb_enventry_singleT_StringBean
ejb_ee_deploy_mdb_resref_singleT_TestBean
jms_ee_mdb_mdb_exceptT_MDB_DURABLETXNS_CMT
jms_ee_mdb_mdb_exceptT_MDB_DURABLE_BMT
jms_ee_mdb_mdb_exceptT_MDB_DURABLE_CMT
jms_ee_mdb_mdb_msgHdrT_MDB_TOPIC
jms_ee_mdb_mdb_msgPropsT_MDB_TOPIC
jms_ee_mdb_mdb_msgTypesT1_MDB_TOPIC
jms_ee_mdb_mdb_msgTypesT2_MDB_TOPIC
jms_ee_mdb_mdb_msgTypesT3_MDB_TOPIC
jms_ee_mdb_mdb_rec_MDB_TOPIC
jms_ee_mdb_mdb_sndToTopic_MDB_TOPIC
jms_ee_mdb_mdb_sndToTopic_MDB_TOPIC_REPLY
jms_ee_mdb_xa_MDB_DURABLE_BMT
jms_ee_mdb_xa_MDB_DURABLE_CMT
testT0
testT1
testT2
```



Implementations of `TSJMSAdminInterface` are called inside the JavaTest VM. (Obsolete, need to define if this is used and how)

Jakarta Enterprise Beans Endpoint Security

element : login-config

This only applies to Jakarta Enterprise Beans endpoints and is optional. It is used to specify how authentication is performed for Jakarta Enterprise Beans endpoint invocations. It consists of a single subelement named `auth-method`. `auth-method` is set to `BASIC` or `CLIENT_CERT`. The equivalent security for servlet endpoints is set through the standard web-application security elements. For example:

```
<ejb>
  <ejb-name>GoogleEjb</ejb-name>
  <webservice-endpoint>
    <port-component-name>GoogleSearchPort</port-component-name>
    <endpoint-address-uri>google/GoogleSearch</endpoint-address-uri>

    <login-config>
      <auth-method>BASIC</auth-method>
    </login-config>
  </webservice-endpoint>
</ejb>
```

Transport Guarantee

element : transport-guarantee

This is an optional setting on `webservice-endpoint`. The allowable values are `NONE`, `INTEGRAL`, and `CONFIDENTIAL`. If not specified, the behavior is equivalent to `NONE`. The meaning of each option is the same as is defined in the Security chapter of the Jakarta Servlet 6.0 Specification. This setting will determine the scheme and port used to generate the final endpoint address for a web service endpoint. For `NONE`, the scheme will be `HTTP` and port will be the default `HTTP` port. For `INTEGRAL/CONFIDENTIAL`, the scheme will be `HTTPS` and the port will be the default `HTTPS` port.

Signature Test Setup

The signature test setup includes the following:

Signature Test JVM Arguments

The following JVM arguments are required for the signature tests:

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

sigTestClasspath Property

Set the `sigTestClasspath` property in the `<TS_HOME>/bin/ts.jte` file to include a `CLASSPATH` containing the following:

```
sigTestClasspath=jar_to_test:jars_used_by_yours
```

where:

- ```jar_to_test```: The JAR file you are validating when running the signature tests; when running against the Jakarta Platform, Enterprise Edition CI, Eclipse GlassFish 6.1, set to `javaee.jar`
- ```jars_used_by_yours```: The JAR file or files that are used or referenced by your JAR file; must include any classes that might be extended or implemented by the classes in your `jar_to_test`; include `rt.jar` when running against the Jakarta Platform, Enterprise Edition CI

javaee.level Property

Set the `javaee.level` property in the `<TS_HOME/bin/ts.jte` file to set the profile to run. For example `platform` for the full platform or `web` for the web profile.

optional.tech.packages.to.ignore Property

Set the `optional.tech.packages.to.ignore` property in the `<TS_HOME/bin/ts.jte` file to set the a comma delimited list of packages to ignore that are optionally available in the distribution. For example, for the web profile, you may need to exclude the `jakarta.mail` packages.

bin.dir Property

Set the `bin.dir` property in the `<TS_HOME>/bin/ts.jte` file to set the path where the signature mapping and package list files are located. Typically you'd extract the files from the test archive.

Additional Signature Test Information

The Jakarta EE 11 Platform TCK signature tests perform verifications in two different modes: static and reflection. The test results list which SPEC API signature tests pass or fail, and the mode (static or reflection) for that test.

Any signature test failure means one of two things, either you have not yet corrected the `sigTestClasspath` or the respective SPEC API jar in your Jakarta EE implementation needs a modification to exactly match the Jakarta EE 11 Platform SPEC API. Your implementation SPEC API jars cannot contain additional public methods/fields, nor can it be missing any expected public methods/fields.

As a troubleshooting aid when failures occur, consider the following:

- All static mode tests fail:
Verify that the `sigTestClasspath` is using correct SPEC API file names. When running on Windows, be sure to use semicolons (;) for CLASSPATH separators.
- For all other signature test failures:
Check the report output from the test to determine which tests failed and why.

For example, some failures from an actual failure:

```
SVR: ***** Status Report 'jakarta.servlet.jsp.jstl.core' *****
```

```
SVR: SignatureTest report
Base version: 2.0_se11
Tested version: 2.0_se11
Check mode: src [throws normalized]
Constant checking: on
```

```
Missing Fields :
```

```
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.FMT_FALLBACK_LOCALE = "jakarta.servlet.jsp.jstl.fmt.fallbackLocale"
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.FMT_LOCALE = "jakarta.servlet.jsp.jstl.fmt.locale"
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.FMT_LOCALIZATION_CONTEXT = "jakarta.servlet.jsp.jstl.fmt.localizationContext"
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.FMT_TIME_ZONE = "jakarta.servlet.jsp.jstl.fmt.timeZone"
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.SQL_DATA_SOURCE = "jakarta.servlet.jsp.jstl.sql.dataSource"
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.SQL_MAX_ROWS = "jakarta.servlet.jsp.jstl.sql.maxRows"
```

Added Fields :

```
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.FMT_FALLBACK_LOCALE = "javax.servlet.jsp.jstl.fmt.fallbackLocale"
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.FMT_LOCALE = "javax.servlet.jsp.jstl.fmt.locale"
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.FMT_LOCALIZATION_CONTEXT = "javax.servlet.jsp.jstl.fmt.localizationContext"
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.FMT_TIME_ZONE = "javax.servlet.jsp.jstl.fmt.timeZone"
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.SQL_DATA_SOURCE = "javax.servlet.jsp.jstl.sql.dataSource"
jakarta.servlet.jsp.jstl.core.Config:  field public final static java.lang.String
jakarta.servlet.jsp.jstl.core.Config.SQL_MAX_ROWS = "javax.servlet.jsp.jstl.sql.maxRows"
```

SVR: ***** Package 'jakarta.servlet.jsp.jstl.core' - FAILED (STATIC MODE) *****

The failure above is a little strange, isn't it? Why are there missing fields? Why are there added fields? The failure means that the `jakarta.servlet.jsp.jstl.core.Config` class needs to be updated to assign the correct values to the indicated constant fields. Basically, instead of setting `Config.FMT_FALLBACK_LOCALE = "javax.servlet.jsp.jstl.fmt.fallbackLocale"`, you should set `Config.FMT_FALLBACK_LOCALE = "jakarta.servlet.jsp.jstl.fmt.fallbackLocale"`. The same correction is needed for the other identified fields as well.

Another example only with methods is:

SVR: ***** Status Report 'jakarta.el' *****

```
SVR: SignatureTest report
Base version: 4.0_se11
Tested version: 4.0_se11
Check mode: src [throws normalized]
Constant checking: on
```

Missing Methods :

```
jakarta.el.ELContext:          method public java.lang.Object
jakarta.el.ELContext.getContext(java.lang.Class<?>)
jakarta.el.ELContext:          method public void
jakarta.el.ELContext.putContext(java.lang.Class<?>, java.lang.Object)
jakarta.el.StandardELContext:  method public java.lang.Object
jakarta.el.StandardELContext.getContext(java.lang.Class<?>)
jakarta.el.StandardELContext:  method public void
jakarta.el.StandardELContext.putContext(java.lang.Class<?>, java.lang.Object)
```

Added Methods :

```
jakarta.el.ELContext:          method public java.lang.Object
jakarta.el.ELContext.getContext(java.lang.Class)
jakarta.el.ELContext:          method public void
jakarta.el.ELContext.putContext(java.lang.Class, java.lang.Object)
jakarta.el.StandardELContext:  method public java.lang.Object
jakarta.el.StandardELContext.getContext(java.lang.Class)
jakarta.el.StandardELContext:  method public void
jakarta.el.StandardELContext.putContext(java.lang.Class, java.lang.Object)
```

The failure above is a little strange, isn't it? Why are there missing methods? Why are there added methods? The failure means that the `java.lang.Object jakarta.el.ELContext.getContext(java.lang.Class)` method needs a signature change from `getContext(Class key)` to `getContext(Class<?> key)`. The same correction is needed for the other identified methods as well.



Refer to [Debugging Test Problems](#) for additional debugging information.

Signature Test Configuration Examples

An example `ts.jte` file will look something like:

```
webServerHost=localhost
webServerPort=8080
ts_home=./target/jakartaee

bin.dir=./target/jakartaee/com/sun/ts/tests/signaturetest/signature-repository
sigTestClasspath=./target/wildfly/modules/system/layers/base/jakarta/activation/api/main/jakarta.activation-api-
2.1.3.jar:./target/wildfly/modules/system/layers/base/jakarta/annotation/api/main/jakarta.annotation-api-
3.0.0.jar:./target/wildfly/modules/system/layers/base/jakarta/batch/api/main/jakarta.batch-api-
2.1.1.jar:./target/wildfly/modules/system/layers/base/jakarta/data/api/main/jakarta.data-api-
1.0.1.jar:./target/wildfly/modules/system/layers/base/jakarta/ejb/api/main/jakarta.ejb-api-
4.0.1.jar:./target/wildfly/modules/system/layers/base/jakarta/el/api/main/jakarta.el-api-
6.0.1.jar:./target/wildfly/modules/system/layers/base/jakarta/enterprise/api/main/jakarta.enterprise.cdi-api-
4.1.0.jar:./target/wildfly/modules/system/layers/base/jakarta/enterprise/api/main/jakarta.enterprise.cdi-el-api-
4.1.0.jar:./target/wildfly/modules/system/layers/base/jakarta/enterprise/api/main/jakarta.enterprise.lang-model-
4.1.0.jar:./target/wildfly/modules/system/layers/base/jakarta/enterprise/concurrent/api/main/jakarta.enterprise.con-
current-api-3.1.0.jar:./target/wildfly/modules/system/layers/base/jakarta/faces/impl/main/jakarta.faces-
4.1.0.jar:./target/wildfly/modules/system/layers/base/jakarta/inject/api/main/jakarta.inject-api-
2.0.1.jar:./target/wildfly/modules/system/layers/base/jakarta/interceptor/api/main/jakarta.interceptor-api-
2.2.0.jar:./target/wildfly/modules/system/layers/base/jakarta/jms/api/main/jakarta.jms-api-
3.1.0.jar:./target/wildfly/modules/system/layers/base/jakarta/json/api/main/jakarta.json-api-
2.1.3.jar:./target/wildfly/modules/system/layers/base/jakarta/json/bind/api/main/jakarta.json.bind-api-
3.0.1.jar:./target/wildfly/modules/system/layers/base/jakarta/mail/api/main/jakarta.mail-api-
2.1.3.jar:./target/wildfly/modules/system/layers/base/jakarta/persistence/api/main/jakarta.persistence-api-
3.2.0.jar:./target/wildfly/modules/system/layers/base/jakarta/resource/api/main/jakarta.resource-api-
2.1.0.jar:./target/wildfly/modules/system/layers/base/jakarta/security/auth/message/api/main/jakarta.authentication-
-api
-3.1.0.jar:./target/wildfly/modules/system/layers/base/jakarta/security/enterprise/api/main/jakarta.security.enterp-
rise-api-
4.0.0.jar:./target/wildfly/modules/system/layers/base/jakarta/security/jacc/api/main/jakarta.authorization-api-
3.0.0.jar:./target/wildfly/modules/system/layers/base/jakarta/servlet/api/main/jakarta.servlet-api-
6.1.0.jar:./target/wildfly/modules/system/layers/base/jakarta/servlet/jsp/api/main/jakarta.servlet.jsp-api-
3.1.1.jar:./target/wildfly/modules/system/layers/base/jakarta/servlet/jstl/api/main/jakarta.servlet.jsp.jstl-3.0.1-
jbossorg-1.jar:./target/wildfly/modules/system/layers/base/jakarta/servlet/jstl/api/main/jakarta.servlet.jsp.jstl-
api-3.0.2.jar:./target/wildfly/modules/system/layers/base/jakarta/transaction/api/main/jakarta.transaction-api-
2.0.1.jar:./target/wildfly/modules/system/layers/base/jakarta/validation/api/main/jakarta.validation-api-
3.1.0.jar:./target/wildfly/modules/system/layers/base/jakarta/websocket/api/main/jakarta.websocket-api-
2.2.0.jar:./target/wildfly/modules/system/layers/base/jakarta/websocket/api/main/jakarta.websocket-client-api-
2.2.0.jar:./target/wildfly/modules/system/layers/base/jakarta/ws/rs/api/main/jakarta.ws.rs-api-
4.0.0.jar:./target/jdk-bundle/java.base:./target/jdk-bundle/java.rmi:./target/jdk-bundle/java.sql:./target/jdk-
bundle/java.naming
```

Example POM Snippets

```
<properties>
  <base.tck.dir>${project.build.directory}/jakartaee</base.tck.dir>
  <bin.dir>${base.tck.dir}/com/sun/ts/tests/signaturetest/signature-repository</bin.dir>
  <signature.file.dir>${base.tck.dir}/src</signature.file.dir>
</properties>

<plugins>
  <plugin>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
      <execution>
        <id>extract-mapping-file</id>
        <phase>process-test-resources</phase>
        <goals>
          <goal>unpack</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
```

```

    <configuration>
      <artifactItems>
        <artifactItem>
          <groupId>jakarta.tck</groupId>
          <artifactId>signaturevalidation</artifactId>
          <version>${version.jakarta.ee.platform.tck}</version>
          <overWrite>true</overWrite>
          <outputDirectory>${base.tck.dir}</outputDirectory>
          <includes>**/sig-test.map,**/sig-test-pkg-list.txt</includes>
        </artifactItem>
        <artifactItem>
          <groupId>jakarta.tck</groupId>
          <artifactId>signaturevalidation</artifactId>
          <version>${version.jakarta.ee.platform.tck}</version>
          <overWrite>true</overWrite>
          <outputDirectory>${signature.file.dir}</outputDirectory>
          <includes>**/*.sig</includes>
        </artifactItem>
      </artifactItems>
    </configuration>
  </execution>
</executions>
</plugin>

<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <configuration>
    <dependenciesToScan>
      <dependency>jakarta.tck:signaturevalidation</dependency>
    </dependenciesToScan>
  </configuration>
  <executions>
    <execution>
      <id>signaturevalidation</id>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
      <configuration>
        <includes>
          <include>**/*Test</include>
        </includes>

        <groups>${tck.profile}</groups>
        <additionalClasspathElements>
          <!-- Include the libraries from the server on the test class path -->
          <!--suppress MavenModelInspection -->
          <additionalClasspathElement>${jakarta.api.jars}</additionalClasspathElement>
        </additionalClasspathElements>
      </configuration>
    </execution>
  </executions>
</plugin>
</plugins>

```



In the above example the `signature.file.dir` has a `src` directory prefix which is important. Currently the path for the signature test files is `src/com/sun/ts/tests/signaturetest/tests/signaturetest/` and is hard-coded. This may change in a future release to not require such a property be set.

Backend Database Setup

The following sections address special backend database setup considerations:

- [Setup Considerations for MySQL](#)
- [Setup Considerations for MS SQL Server](#)

Setup Considerations for MySQL

The Jakarta Persistence API (formerly JPA) tests require delimited identifiers for the native query tests. If you are using delimited identifiers on MySQL, modify the `sql-mode` setting in the `my.cnf` file to set the `ANSI_QUOTES` option. After setting this option, reboot the MySQL server. Set the option as shown in this example:

```
sql-mode="STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION,ANSI_QUOTES"
```

Setup Considerations for MS SQL Server

If your database already exists and if you use a case-sensitive collation on MS SQL Server, execute the following command to modify the database and avert errors caused by case-sensitive collation:

```
ALTER DATABASE ctsdb  
COLLATE Latin1_General_CS_AS ;
```

Setup and Configuration for Testing with the Jakarta EE 11 Web Profile

This chapter describes how to configure the Jakarta EE 11 Platform TCK test suite to work with your Jakarta EE 11 Web Profile test environment. It is recommended that you first set up the testing environment using the Jakarta EE 11 Web Profile CI and then with your Jakarta EE 11 Web Profile server.

Configuring the Jakarta EE 11 Web Profile Test Environment

The instructions in this section and in [Configuring Your Application Server as the VI](#) step you through the configuration process for the Solaris, Microsoft Windows, and Linux platforms.

To Run Tests Against a Jakarta EE 11 Web Profile Implementation

The Jakarta EE 11 Platform TCK is the Technology Compatibility Kit (TCK) for the Jakarta Platform, Enterprise Edition as well as the Jakarta EE 11 Web Profile. Implementations of the full Jakarta Platform, Enterprise Edition must pass all of the tests as defined by Jakarta EE 11 Platform TCK Rules in [Procedure for Jakarta Platform, Enterprise Edition 11.0 Certification](#).

Implementations of the Jakarta EE 11 Web Profile must run the tests that verify requirements defined by the Jakarta EE 11 Web Profile Specification. These tests are defined by the Rules in [Procedure for Jakarta Platform, Enterprise Edition 11 Web Profile Certification](#). These requirements are a subset of the tests contained in the Jakarta EE 11 Platform TCK test suite. The test suite provides a mechanism whereby only those tests for the Jakarta EE 11 Web Profile will be run. The following steps explain how to use this mechanism.

1. Configure your runner pom.xml maven-surefire-plugin to specify a group of *web* as shown below:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>${maven.surefire.plugin.version}</version>
  <configuration>
    <dependenciesToScan>...</dependenciesToScan>
    </dependenciesToScan>
    <groups>web</groups>
  </configuration>
</plugin>
```

This setting will only allow WAR files (that is, no EAR files) to be passed to the Deployment Porting Package. This is the minimal set of requirements that vendors must support for Web Profile. Again, "web" only covers REQUIRED technologies for the Jakarta EE 11 Web Profile.

Executing Tests

The Jakarta EE 11 Platform TCK uses Arquillian and Junit5 as the test framework. The technology test artifacts have been split up into individual test artifacts based on the underlying component specification.

This chapter includes the following topics:

- [Jakarta EE 11 Platform TCK Operating Assumptions](#)
- [Running the Tests](#)
- [Using Keywords to Test Required Technologies](#)
- [Rebuilding Tests for Different Databases](#)
- [Test Reports](#)



The instructions in this chapter assume that you have installed and configured your test environment as described [Installation](#) and [Setup and Configuration](#) respectively.

Jakarta EE 11 Platform TCK Operating Assumptions

The following are assumed in this chapter:

- Jakarta EE 11 CI is installed and configured as described in this guide.
- Detailed configuration will vary from product to product. In this guide, we provide details for configuring the Jakarta EE CI, Eclipse GlassFish 8.0. If you are using another CI, refer to that product's setup and configuration documentation.
- Java SE 17/21 software is correctly installed and configured on the host machine.
- Jakarta EE 11 Platform TCK is installed and configured as described in this guide.
- Implementations of the technologies to be tested are properly installed and configured.

Running the Tests

There are two general ways to run Jakarta EE 11 Platform TCK using the Arquillian/Junit5 test harness software:

- Through a Java IDE that supports Junit5
- From a maven pom setup that configures the surefire plugin to run the tests

Running the tests in a Java IDE is useful for debugging and validating individual test setup.

Running the tests from a maven test runner pom is the standard way of automating the run of the complete set of tests and allows for producing test reports used to create a certification request.

To Run Testa in Command-Line Mode

Any framework that allows one to configure the run of Junit5 tests could be used, but in this section we will highlight the use of maven and the maven-surefire-plugin. The platform-tck project includes a set of maven runner projects for GlassFish 8.0 in the <https://github.com/jakartaee/platform-tck/tree/main/glassfish-runner> subproject. There is one runner project for each of the technology test artifacts. You can use these runners as a template for your own test runners, or you can create one complete test runner that includes all the technology test artifacts.

As a general rule, the test runner should include the following:

1. Dependencies on the Junit5 and Arquillian test frameworks as shown in [Maven Dependencies](#).

2. Configure your Arquillian container as shown in [Arquillian Container Configurations](#).
3. Set the `TS_HOME` environment variable to the directory in which Jakarta EE 11 Platform TCK was installed.
4. Ensure that the `ts.jte` file contains information relevant to your setup.
Refer to [Setup and Configuration](#) for detailed configuration instructions, or that your runner maven project surefire/failsafe plugin has defined the equivalent system properties.
5. Execute the test Maven goal to start the testsuite:

```
mvn test
```

This runs all tests in the current directory and any subdirectories.

Maven Dependencies

Your runner configuration should have a `dependencyManagement` section that imports the Junit5 and Arquillian and Jakarta EE API and Jakarta TCK test artifact bom dependencies. There is a `jakarta.tck:artifacts-bom` that contains the Jakarta EE 11 TCK test artifacts and their dependencies.

Maven `dependencyManagement` Section

```
<!-- Import jakarta.tck:artifacts-bom BOM to simplify dependency versions -->
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>jakarta.tck</groupId>
      <artifactId>artifacts-bom</artifactId>
      <version>11.0.0-M7</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

You then need to define the specific dependencies for the test runner without the version information, for example, this would be applicable to the Jakarta Enterprise Beans tests:

Maven `dependencies` Section

```
<dependencies>
  <!-- Jakarta EE APIs -->
  <dependency>
    <groupId>jakarta.platform</groupId>
    <artifactId>jakarta.jakartaee-api</artifactId>
    <scope>provided</scope>
  </dependency>

  <!-- The TCK test artifacts for components of interest -->
  <dependency>
    <groupId>jakarta.tck</groupId>
    <artifactId>ejb30</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>jakarta.tck</groupId>
    <artifactId>ejb32</artifactId>
    <scope>test</scope>
  </dependency>

  <!-- TCK common dependencies -->
  <dependency>
    <groupId>jakarta.tck</groupId>
```

```

        <artifactId>common</artifactId>
    </dependency>

    <!-- TCK Arquillian artifacts -->
    <dependency>
        <groupId>jakarta.tck.arquillian</groupId>
        <artifactId>arquillian-protocol-appliance</artifactId>
    </dependency>
    <dependency>
        <groupId>jakarta.tck.arquillian</groupId>
        <artifactId>arquillian-protocol-javatest</artifactId>
    </dependency>
    <dependency>
        <groupId>jakarta.tck.arquillian</groupId>
        <artifactId>tck-porting-lib</artifactId>
    </dependency>

    <!--
        The Arquillian connector that starts the VI and deploys archives to it. This is the GlassFish 8.0
version.
    -->
    <dependency>
        <groupId>org.omnifaces.arquillian</groupId>
        <artifactId>arquillian-glassfish-server-managed</artifactId>
        <version>1.8</version>
        <scope>test</scope>
    </dependency>
</dependencies>

```

You are allowed to update the dependencies to the latest version of the Jakarta EE API and Jakarta TCK test dependencies, but the jakarta.tck:* test artifact version should be the same as the version of the TCK you are running.

Available TCK Test Artifacts

The list of groupId:artifactId for the Jakarta EE 11 TCK test artifacts is:

- jakarta.tck:appliance
- jakarta.tck:assembly-tck
- jakarta.tck:common
- jakarta.tck:cdi-tck-ee-impl
- jakarta.tck:connector
- jakarta.tck:ejb30
- jakarta.tck:ejb32
- jakarta.tck:el-platform-tck
- jakarta.tck:integration
- jakarta.tck:javaee-tck
- jakarta.tck:rest-platform-tck
- jakarta.tck:javamail
- jakarta.tck:jdbc-platform-tck
- jakarta.tck:jms-platform-tck
- jakarta.tck:project
- jakarta.tck:persistence-platform-tck-tests
- jakarta.tck:persistence-platform-tck-common
- jakarta.tck:persistence-platform-tck-dbprocedures
- jakarta.tck:persistence-platform-tck-spec-tests

- jakarta.tck:jsonb-platform-tck
- jakarta.tck:jsonp-platform-tck
- jakarta.tck:pages-platform-tck
- jakarta.tck:transactions-tck
- jakarta.tck:tags-tck
- jakarta.tck:signaturetest
- jakarta.tck:websocket-tck-platform-tests
- jakarta.tck:xa

The version of these artifacts is the same as the service release version of the TCK. You can find the latest version of these artifacts in the Jakarta staging repository or in the Maven Central repository. You can search for the jakarta.tck:artifacts-bom to find the latest version that has been released.

Arquillian Container Configurations

Arquillian Container Configuration (javatest Protocol)

In addition to specifying the test framework and TCK test artifact dependencies, you need to configure the Arquillian container. The Arquillian container manages the VI being tested and handles starting the container, deploying the test archives, and then stopping the container.

The configuration will depend on the container you are using, but you will need to include a configuration of the Arquillian javatest protocol for web profile tests.

Maven javatest-arquillian.xml javatest Protocol Configuration

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="http://jboss.org/schema/arquillian"
            xsi:schemaLocation="http://jboss.org/schema/arquillian
http://jboss.org/schema/arquillian/arquillian_1_0.xsd">

    <engine>
        <property name="deploymentExportPath">target/deployments</property>
    </engine>
    <extension qualifier="glassfish-descriptors">
        <property name="descriptorDir">target</property>
    </extension>

    <group qualifier="glassfish-servers" default="true">
        <container qualifier="tck-javatest" default="true">
            <configuration> ①
                <property name="glassFishHome">target/glassfish8</property>
                <property name="debug">true</property>
                <property name="suspend">>false</property>
            </configuration>
            <protocol type="javatest"> ②
                <property name="trace">true</property>
                <property name="workDir">/tmp</property>
                <property name="tsJteFile">jakartaeeetck/bin/ts.jte</property>
                <property name="tsSqlStmtFile">sql/derby/derby.dml.sql</property>
            </protocol>
        </container>
    </group>

</arquillian>
```

① This is the managed container specific config, here for GlassFish 8.0.

- ② This is the javatest protocol configuration.
- `tsJteFile` property is the path to the `ts.jte` file that contains the configuration for the test run.
 - `tsSqlStmtFile` property is the path to the SQL file that contains the DML statements for the test run.
 - the `trace` property is used to enable additional logging for the test run.
 - the `workDir` property is the directory where the test TCK classes will put working files.

Failsafe Plugin Configuration (javatest Protocol)

And finally, you need to configure a surefire/failsafe plugin execution to use this container configuration. The following is an example of the surefire/failsafe plugin configuration for the javatest protocol:

Example javatest-arquillian.xml maven-failsafe-plugin Configuration

```
<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>3.5.0</version>
  <configuration>
    <trimStackTrace>>false</trimStackTrace>
    <dependenciesToScan>...</dependenciesToScan>
    <systemPropertyVariables>
      ...
    </systemPropertyVariables>
  </configuration>

  <executions>
    <execution>
      <id>jpa-tests-web</id>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
      <configuration>
        <includes>
          ...
        </includes>
        <excludeGroups>tck-appclient,arq-servlet</excludeGroups> ①

        <systemPropertyVariables>
          <arquillian.xml>javatest-arquillian.xml</arquillian.xml>②
        </systemPropertyVariables>
      </configuration>
    </execution>
  </executions>
</plugin>
```

- ① Exclude other protocol types used in TCK.
- ② Set the `arquillian.xml` property to the `javatest-arquillian.xml` file example above.

Arquillian Container Configuration (Servlet Protocol)

The new CDI/Persistence integration tests in the `jakarta.tck:persistence-platform-tck-tests` artifact `ee.jakarta.tck.persistence.ee.cdi` package require an Arquillian container with the Servlet protocol. The bom for the common Jakarta EE based protocols was included in the [Maven Dependencies](#) section example above.

In your runner dependencies, you need to include the following dependency to enable the Servlet protocol: .Additional Maven dependency Servlet Protocol

```
<dependencies>
  ...
  <dependency>
    <groupId>org.jboss.arquillian.jakarta</groupId>
    <artifactId>arquillian-parent-jakarta</artifactId>
```

```

    </dependency>
</dependencies>

```

You will also need a container configuration for the Servlet protocol. The following is an example of the `arquillian.xml` file for the Servlet protocol for the GlassFish 8.0 container:

Example `servlet-arquillian.xml` Container Configuration

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://jboss.org/schema/arquillian"
    xsi:schemaLocation="http://jboss.org/schema/arquillian
http://jboss.org/schema/arquillian/arquillian_1_0.xsd">

    <defaultProtocol type="Servlet 5.0" /> ①
    <engine>
        <property name="deploymentExportPath">target/deployments</property>
    </engine>
    <extension qualifier="glassfish-descriptors">
        <property name="descriptorDir">target</property>
    </extension>

    <group qualifier="glassfish-servers" default="true">
        <container qualifier="tck-rest" default="true">
            <configuration>
                <property name="glassFishHome">target/glassfish8</property>
            </configuration>
        </container>
    </group>

</arquillian>

```

① This sets the default protocol to Servlet 5.0.

Failsafe Plugin Configuration (Servlet Protocol)

And finally, you need to configure a surefire/failsafe plugin execution to use this container configuration. The following is an example of the surefire/failsafe plugin configuration for the Servlet protocol:

Example `servlet-arquillian.xml` `maven-failsafe-plugin` Configuration

```

<plugin>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>3.5.0</version>
    <configuration>
        <trimStackTrace>>false</trimStackTrace>
        <dependenciesToScan>jakarta.tck:persistence-platform-tck-tests</dependenciesToScan>

        <systemPropertyVariables>
            ...
        </systemPropertyVariables>
    </configuration>

    <executions>
        <execution>
            <id>jpa-tests-cdi</id>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
            <configuration>
                <includes> ①
                    <include>ee/jakarta/tck/persistence/ee/cdi/*Test.java</include>
                </includes>
            </configuration>
        </execution>
    </executions>
</plugin>

```

```

        <systemPropertyVariables>
            <arquillian.xml>servlet-arquillian.xml</arquillian.xml>②
        </systemPropertyVariables>
    </configuration>
</execution>

```

- ① Restrict the tests run to only those in the `ee.jakarta.tck.persistence.ee.cdi` package.
- ② Set the `arquillian.xml` property to the `servlet-arquillian.xml` file example above.

Arquillian Container Configuration (applient Protocol) (Full Platform Only)

Full platform tests make use of the Application Client container. These tests require a special Arquillian protocol that is used to run the tests in the Application Client container. Thus, full platform TCK tests will require a test plugin configuration that enables the applient protocol.

Maven `applient-arquillian.xml` *applient Protocol Configuration*

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://jboss.org/schema/arquillian"
    xsi:schemaLocation="http://jboss.org/schema/arquillian
http://jboss.org/schema/arquillian/arquillian_1_0.xsd">

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

    <extension qualifier="glassfish-descriptors">
        <property name="descriptorDir">target</property>
    </extension>

    <group qualifier="glassfish-servers" default="true">
        <container qualifier="tck-applient" default="true">
            <configuration> ①
                <property name="glassFishHome">target/glassfish8</property>
            </configuration>
            <protocol type="applient"> ②
                <property name="runClient">true</property>
                <property name="runAsVehicle">true</property>
                <property name="clientEarDir">target/applient</property>
                <property name="unpackClientEar">true</property>
                <property name="clientCmdLineString">${env.JAVA_HOME}/bin/java \
                    --add-opens \
                    java.base/java.lang=ALL-UNNAMED \
                    -cp \
                    ${glassfish.home}/glassfish/lib/gf-client.jar:${clientStubJar}:target/lib/tck-porting-lib.jar \
                    -Djava.system.class.loader=org.glassfish.applient.client.acc.agent.ACCAgentClassLoader \
                    -Dcom.sun.aas.configRoot=${glassfish.home}/glassfish/config \
                    -Dcts.tmp=${ts.home}/tmp \
                    -Doracle.jdbc.J2EE13Compliant=true \
                    -Doracle.jdbc.mapDateToTimestamp \
                    -Dlog.file.location=${glassfish.home}/glassfish/domains/domain1/logs \
                    -Dri.log.file.location=${glassfish.home}/glassfish/domains/domain1/logs \
                    -DwebServerHost.2=localhost \
                    -DwebServerPort.2=8080 \
                    -javaagent:${glassfish.home}/glassfish/lib/gf-client.jar=arg=
                -configxml,arg=${glassfish.home}/glassfish/domains/domain1/config/glassfish
                -acc.xml,client=jar=${clientStubJar},arg=-name,arg=${clientAppArchiveName} \
                    org.glassfish.applient.client.AppClientGroupFacade
            </property>
            <property name="cmdLineArgSeparator">\\</property>
            <!-- Pass ENV vars here -->
            <property name="clientEnvString">PATH=${env.PATH};LD_LIBRARY_PATH=${glassfish.home}/lib;AS_DEBUG=true;

```

```

        APPCPATH=${clientEarLibClasspath}:${glassfish.home}/glassfish/modules/security.jar</property>
<property name="clientDir">${project.basedir}</property>
<property name="clientStubsCmdLine">${env.JAVA_HOME}/bin/java \
    -jar \
    ${glassfish.home}/glassfish/modules/admin-cli.jar \
    get-client-stubs \
    --appName \
    ${deploymentName} \
    target
</property>
<property name="clientStubsJarSuffix">Client</property>
<property name="workDir">/tmp</property>
<property name="tsJteFile">jakartaeetck/bin/ts.jte</property>
<property name="tsSqlStmtFile">sql/derby/derby.dml.sql</property>
<property name="trace">true</property>
<property name="clientTimeout">20000</property>
</protocol>
</container>
</group>

</arquillian>

```

① This is the managed container specific config, here for GlassFish 8.0.

② This is the appclient protocol configuration.

- runClient - set to true to run the client
- runAsVehicle - should be true for platform TCK
- clientEarDir - the directory where the client EAR test artifact will be created
- unpackClientEar - if true, the client EAR will be unpacked. If the VI appclient container does not support a path to the client jar inside of an EAR, then this should be set to true.
- clientCmdLineString - the command line the VI uses to launch and application client. This should be a space separated list of arguments, but long command lines can be broken up using a separator character that can be specified using the cmdLineArgSeparator property. There are special properties that can be used in the command line:
 - \${clientEarDir} : this will be replaced with the clientEarDir property value as an absolute path.
 - \${clientAppArchive} - this is the client app archive that contains the appclient main class. It is a relative path to the clientEarDir property.
 - \${clientAppArchiveName} - this is the client app archive minus the .jar extension.
 - \${clientEarLibClasspath} - this is the classpath for the client EAR lib directory. It is a colon separated list of jars in the clientEarDir/lib directory.
 - \${deploymentName} - the name of the deployment that is being tested. This is the name of the EAR file that is being deployed to the server and would reflect renaming of a deployment by the ear application.xml.
 - \${clientStubJar} - this is the client stub jar that is created by the appclient container if the clientStubsCmdLine property is set.
 - \${vehicleArchiveName} - the name of the archive in the ear deployment that contains the legacy javatest vehicle invocation target.
- cmdLineArgSeparator - the character used to split command line arguments into multiple lines. Each line should be a single argument and the last line should not end with the separator character. Each line will be trimmed of leading and trailing whitespace.
- clientEnvString - any environment variables that need to be set for the appclient process. This is a semicolon separated list of environment variables in the form VAR=VALUE. You can pass through runner environment variables if you have the xml file as a resource and have resource filtering on. Use the maven syntax of \${env.VARNAME} to access the runner VARNAME environment variable.
- clientDir - the working directory of the appclient container process.

- `clientTimeout` - a timeout in milliseconds for the appclient process to finish. The appclient process will be terminated if it does not finish in this time.
- `clientStubsCmdLine` - the command line to run to create the appclient stubs if needed. It supports property replacement for:
 - `${deploymentName}` - the name of the deployment that is being tested. This is the name of the EAR file that is being deployed to the server and would reflect renaming of a deployment by the `ear application.xml`.
- `clientStubsJarSuffix` - the suffix that the appclient container uses to create the client stub jar by appending this value to the `${clientStubJar}`. This is used to create the `clientStubsJar` property.
- `tsJteFile` property is the path to the `ts.jte` file that contains the configuration for the test run.
- `tsSqlStmtFile` property is the path to the SQL file that contains the DML statements for the test run.
- the `trace` property is used to enable additional logging for the test run.
- the `workDir` property is the directory where the test TCK classes will put working files.

FailSAFE Plugin Configuration (appclient Protocol) (Full Platform Only)

```
<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>3.5.0</version>
  <executions>
    <execution>
      <id>jpa-tests-appclient</id>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
      <configuration>
        <includes>
          <include>...</include>
        </includes>
        <groups>platform&tck-appclient</groups> ①
        <excludedGroups>tck-javatest,arq-servlet</excludedGroups> ②

        <systemPropertyVariables>
          <arquillian.xml>appclient-arquillian.xml</arquillian.xml> ③
          <ts.home>${ts.home}</ts.home>
        </systemPropertyVariables>
      </configuration>
    </execution>
  </executions>
</plugin>
```

- ① Restrict the tests run to only those tagged with both `platform` and `tck-appclient`, which is the group of tests required by the Full Platform that use an application client container. Only the Full Platform supports the appclient container.
- ② Exclude the protocol types other than `tck-appclient`.
- ③ Set the `arquillian.xml` property to the `appclient-arquillian.xml` file example above.

Example - Running a Single Test Package

To run a single test package, you would configure your surefire/failsafe plugin to have an execution that only includes the tests in the package you are interested in. For example, to run the tests in the `ee.jakarta.tck.persistence.ee.cdi` package, you would configure your surefire/failsafe plugin as shown in the following example:

```
<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>3.5.0</version>
```

```

<configuration>
  <trimStackTrace>false</trimStackTrace>
  <dependenciesToScan>jakarta.tck:persistence-platform-tck-tests</dependenciesToScan>
<executions>
  <execution>
    <id>jpa-tests-cdi</id>
    <goals>
      <goal>integration-test</goal>
      <goal>verify</goal>
    </goals>
    <configuration>
      <includes>
        <include>ee/jakarta/tck/persistence/ee/cdi/*Test.java</include>①
      </includes>

      <systemPropertyVariables>
        <arquillian.xml>cdi-arquillian.xml</arquillian.xml> ②
      </systemPropertyVariables>
    </configuration>
  </execution>
  ...
</executions>
</plugin>

```

- ① Restrict the tests run to only those in the `ee.jakarta.tck.persistence.ee.cdi` package.
- ② As described in the [Arquillian Container Configurations](#) section, you need to set the `arquillian.xml` property to a container configuration for the protocol used by the tests.

You could further restrict the included test to a specific test class by changing the includes to a specific test class name.

Running the Jakarta EE 11 Platform TCK Signature Tests

To run the Jakarta EE 11 Platform TCK signature tests, configure a runner as outlined in the following example segments from the `glassfish-runner/signature/pom.xml` runner.

Example Maven dependencyManagement and dependencies Section

```

<properties>
  <!-- Properties set in the JTE file -->
  <base.tck.dir>${project.build.directory}/jakartaee</base.tck.dir>
  <bin.dir>${base.tck.dir}/com/sun/ts/tests/signaturetest/signature-repository</bin.dir>

  <!-- Note that currently, this must have src as the first directory as it is hard-coded in the test -->
  <signature.file.dir>${base.tck.dir}/src</signature.file.dir>
  <version.jakarta.tck>{jakartaee_tck_version}</version.jakarta.tck>
  <!-- A classpath type property containing all Jakarta EE API jars -->
  <jakarta.api.jars>fill-in-api-jars</jakarta.api.jars>
</properties>
<!-- Import the tck relevant boms -->
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>jakarta.tck</groupId>
      <artifactId>artifacts-bom</artifactId>
      <version>${version.jakarta.tck}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>

```

```

    <groupId>jakarta.tck</groupId>
    <artifactId>signaturevalidation</artifactId>
</dependency>

<dependency>
    <groupId>jakarta.tck</groupId>
    <artifactId>signaturetest</artifactId>
</dependency>
<dependency>
    <groupId>jakarta.tck</groupId>
    <artifactId>sigtest-maven-plugin</artifactId>
    <version>2.6</version>
</dependency>

<!-- Jakarta TCK tools dependencies -->

<dependency>
    <groupId>jakarta.tck.arquillian</groupId>
    <artifactId>arquillian-protocol-javatest</artifactId>
</dependency>
<dependency>
    <groupId>jakarta.tck.arquillian</groupId>
    <artifactId>tck-porting-lib</artifactId>
</dependency>

<dependency>
    <groupId>org.jboss.arquillian.junit5</groupId>
    <artifactId>arquillian-junit5-container</artifactId>
</dependency>
<dependency>
    <groupId>org.omnifaces.arquillian</groupId>
    <artifactId>arquillian-glassfish-server-managed</artifactId>
    <version>1.7</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.jboss.shrinkwrap</groupId>
    <artifactId>shrinkwrap-api</artifactId>
</dependency>

</dependencies>

```

The signature map and signature files are included in the `jakarta.tck:signaturevalidation` artifact. These should be extracted to the `runner.base.tck.dir` and `signature.file.dir` as shown in this `maven-dependency-plugin` segment:

Example Maven maven-dependency-plugin Section

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>3.8.1</version>
  <executions>

    <execution>
      <id>extract-sigtest-files</id>
      <phase>process-test-resources</phase>
      <goals>
        <goal>unpack</goal>
      </goals>
      <configuration>
        <artifactItems>
          <artifactItem>
            <groupId>jakarta.tck</groupId>
            <artifactId>signaturevalidation</artifactId>
            <version>${version.jakarta.tck}</version>
            <overWrite>true</overWrite>
          </artifactItem>
        </artifactItems>
      </configuration>
    </execution>
  </executions>
</plugin>

```

```

        <outputDirectory>${base.tck.dir}</outputDirectory>
        <includes>**/sig-test.map,**/sig-test-pkg-list.txt</includes>
    </artifactItem>
    <artifactItem>
        <groupId>jakarta.tck</groupId>
        <artifactId>signaturevalidation</artifactId>
        <version>${version.jakarta.tck}</version>
        <overWrite>true</overWrite>
        <outputDirectory>${signature.file.dir}</outputDirectory>
        <includes>**/*.sig</includes>
    </artifactItem>
</artifactItems>
</configuration>
</execution>
</executions>
</plugin>

```

The `jakarta.api.jars` property should be set to a classpath type property that contains all the Jakarta EE API jars. This is passed in as a `additionalClasspathElements` to set the classpath for the tests. It should be built up from the jars included in the VI server distribution.

Using that, you can then configure the failsafe plugin to run the tests as is shown in the following is an example:

Example Maven maven-failsafe-plugin Section

```

<plugin>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>3.5.2</version>
    <configuration>
        <dependenciesToScan>
            <dependency>jakarta.tck:signaturevalidation</dependency>
        </dependenciesToScan>
    </configuration>
    <executions>
        <execution>
            <id>signaturevalidation-web</id>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
            <configuration>
                <includes>
                    <include>**/*Test</include>
                </includes>

                <groups>${tck.profile}</groups> ①
                <additionalClasspathElements>
                    <!-- Include the libraries from the server on the test class path -->
                    <additionalClasspathElement>${jakarta.api.jars}</additionalClasspathElement>
                </additionalClasspathElements>

                <systemPropertyVariables>
                    <glassfish.home>${glassfish.home}</glassfish.home>
                    <javaee.level>web</javaee.level> ②

                <optional.tech.packages.to.ignore>jakarta.resource,jakarta.resource.cci,jakarta.resource.spi,jakarta.resource.spi.w
ork,jakarta.resource.spi.endpoint,jakarta.resource.spi.security,jakarta.mail,jakarta.mail.event,jakarta.mail,jakart
a.mail.event,jakarta.mail.internet,jakarta.mail.search,jakarta.mail.util,jakarta.security.jacc,jakarta.security.aut
h.message,jakarta.security.auth.message.callback,jakarta.security.auth.message.config,jakarta.security.auth.message
.module</optional.tech.packages.to.ignore> ③
            </systemPropertyVariables>
        </configuration>
    </execution>
</executions>
</plugin>

```

- ① The groups to run; should be either `web` for the Web Profile or `platform` for the Full Platform.
- ② A system property that is used to set the Java EE level for the test run. This should be set to `web` for the Web Profile and `platform` for the Full Platform.
- ③ A system property that is used to set the optional technology packages to ignore. This should be set to the optional technology packages that are included in the Full Platform but not the Web Profile. You can simply copy the setting as shown here.

With your runner configured, you can run the signature tests using `mvn verify`.

Using Keywords to Test Required Technologies

The Jakarta EE TCK includes JUnit5 `@Tag` annotations that allow you to select a subset of tests based on the tag name. Each test in TCK has keywords associated with it. The keywords are used to create groups and subsets of tests. At test execution time, a user can tell the test harness to only run tests with or without certain groups.

The full list of available tags are:

- `@Tag("arq-servlet")`
- `@Tag("assembly")`
- `@Tag("connector")`
- `@Tag("connector_standalone")`
- `@Tag("connector_web")`
- `@Tag("ejb")`
- `@Tag("ejb30")`
- `@Tag("ejb32")`
- `@Tag("ejb_web")`
- `@Tag("ejb_web_profile")`
- `@Tag("el")`
- `@Tag("integration")`
- `@Tag("jaxrs")`
- `@Tag("jdbc")`
- `@Tag("jms")`
- `@Tag("jms_web")`
- `@Tag("jsonb")`
- `@Tag("jsonp")`
- `@Tag("jsp")`
- `@Tag("jsp_security")`
- `@Tag("jstl")`
- `@Tag("jta")`
- `@Tag("mail")`
- `@Tag("persistence")`
- `@Tag("platform")`
- `@Tag("security")`
- `@Tag("signaturetest")`
- `@Tag("tck-appclient")`
- `@Tag("tck-javatest")`

- @Tag("web")
- @Tag("websocket")
- @Tag("xa")

The tags are used to group tests by the technology they are testing. For example, the `ejb` tag is used to group all tests that are related to EJB. None of the technology related tags are used by the TCK. They exist to help filter out tests while debugging test run issues.

The non-technology related tags are: * "platform" is used to run tests that are required by the Full Platform. To be able to run all tests you will need plugin execution configurations for the servlet, appclient, and javatest protocols. * "web" is used to run tests that are required by the Web Profile. To be able to run all tests you will need plugin execution configurations for the servlet, and javatest protocols. * "arq-servlet" is used to run tests that use the Arquillian Servlet protocol. * "tck-appclient" is used to run tests that use the custom Arquillian appclient protocol used in the TCK. * "tck-javatest" is used to run tests that use the custom Arquillian javatest protocol used in the TCK.

The protocol tags are generally used to exclude tests that are not relevant to the protocol a given failsafe/surefire plugin execution is using. Examples of this were given in the [Arquillian Container Configurations](#) section.

To Use Keywords to Run Required Technologies

You use the `<groups>` element in the failsafe/surefire plugin configuration as described in the [Failsafe Plugin documentation](#) to specify the tags you want to run.

Example - Running Tests for Required Technologies in the Full Platform

To restrict the tests to the group of tests that are required by the Full Platform, use the `platform` keyword.

```
...
  <plugins>
    ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <version>3.5.2</version>
      <configuration>
        <groups>platform</groups>
      </configuration>
    </plugin>
  </plugins>
```

Only tests that are required by the Full Platform will be run.

Example - Running Tests for All Required Technologies in the Web Profile

To restrict the tests to the group of tests that are required by the Web Profile, use the `web` keyword.

```
...
  <plugins>
    ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <version>3.5.2</version>
```

```

        <configuration>
          <groups>web</groups>
...
        <executions>
          ...protocol execution setups for servlet and javatest...
        </executions>
      </configuration>
    </plugin>
  </plugins>

```

Only tests that are required by the Web Profile will be run.

Rebuilding Tests for Different Databases

The following packages in the `jakarta.tck:ejb30` test artifact require rebuilding if you are using a database other than Derby with the default settings:

- `com/sun/ts/tests/ejb30/lite/packaging/war/datasource`
- `com/sun/ts/tests/ejb30/misc/datasource/twojars`
- `com/sun/ts/tests/ejb30/misc/datasource/twowars`

The tests under these packages use classes with `@DataSourceDefinition` annotations with settings appropriate for the Derby database. If you are using a different database, you will need to update the `@DataSourceDefinition` annotations in these classes to match the settings for your database.

The typical settings that need to be changed are:

```

    @DataSourceDefinition(name="java:global/jdbc/DB3",
        className="org.apache.derby.jdbc.ClientDataSource", ①
        portNumber=1527, ②
        serverName="localhost", ③
        databaseName="derbyDB;create=true", ④
        user="cts1", ⑤
        transactional=false,
        password="cts1", ⑥
        properties={})

```

- ① The class name of the JDBC driver for your database.
- ② The port number for your database.
- ③ The server name for your database.
- ④ The database name for your database. This may include the path to the database file.
- ⑤ The username for your database.
- ⑥ The password for your database.

JMS (Full Platform Only)

The `com/sun/ts/tests/jms/ee20/resourcedefs` tests may need to be updated and rebuilt for resource definitions...

The database properties in the TCK bundle are set to Derby database. If any other database is used, ...

The following directories require rebuilding: `src\com\sun\ts\tests\appclient\deploy\metadataacomplete\testapp`.

Test Reports

The Maven failsafe/surefire plugins generate test class reports in the `target/failsafe-reports` and `target/surefire-reports`

directory respectively. The reports are in XML format and can be viewed in any XML viewer. They follow a naming convention of TEST-<test-class-name>.xml.

Creating Summary Test Reports

To create a summary report in html format, use the `surefire-report` Maven plugin with:

- `surefire-report:report-only` - for reports on tests run with surefire
- `surefire-report:failsafe-report-only` - for reports on tests run with failsafe

For example, to create a summary report for tests run with failsafe, use the following command:

```
mvn mvn surefire-report:failsafe-report-only
```

This will produce a `target/reports/failsafe.html` file that provides an overview summary as well as the individual test details.

For example, to create a summary report for tests run with surefire, use the following command:

```
mvn mvn surefire-report:report-only
```

This will produce a `target/reports/surefire.html` file that provides an overview summary as well as the individual test details.

Debugging Test Problems

There are a number of reasons that tests can fail to execute properly. This chapter provides some approaches for dealing with these failures.

This chapter includes the following topics:

- [Overview](#)
- [Report Files](#)
- [Debugging Details](#)

Overview

When a test fails, the first thing to do is to look at the test output which can be found in either the `target/surefire-reports` directory or the `target/failsafe-reports` directory. The test output provides a wealth of information about the test run, including the test results, the test output, and the stack trace of any exceptions that occurred during the test run. The CI server log may also provide information if the test fails during deployment or execution within the CI server.

Report Files

You can generate test summary reports in HTML format by running the Maven surefire-reports plugin. See [Test Reports](#) for the details on how to generate test reports. The resulting html summary allows you to browse which tests are failing and decide if more detail debugging is needed.

Debugging Details

Jakarta EE TCK platform tests generally execute in two different JVMs. The first JVM is the client JVM, which is the JVM that runs the test client. The second JVM is the server JVM, which is the JVM that runs the test server. The client JVM is responsible for running the test client, which is the code that interacts with the test server. The server JVM is responsible for running the test server, which is the code that is being tested. The client JVM and the server JVM communicate with each other using the Arquillian protocol. The client JVM sends commands to the server JVM to start and stop the test server, and the server JVM sends test results back to the client JVM.

Debugging the Client JVM

To debug a test that is failing in the client JVM, use the `mvnDebug` command rather than `mvn` to start the client JVM in debug mode. By default, the client JVM listens for debugger connections on port 5005. You can setup a remote debugger in your IDE to connect to the client JVM on port 5005. You would typically want to have the platform-tck repository loaded into your IDE so that you can set breakpoints in the test client code.

Debugging the Server JVM

If tests are failing in the server JVM, you need to start the server JVM in debug mode. To do this, you need to consult the documentation for the Arquillian connector you are using to start the server being tested. You would then setup a remote debugger in your IDE to connect to the server JVM on the port specified in the Arquillian configuration. You would also want to have the platform-tck repository loaded into your IDE so that you can set breakpoints in the server side components the test is calling.

Troubleshooting

This chapter explains how to debug test failures that you could encounter as you run the Jakarta Platform, Enterprise Edition Compatibility Test Suite.

Common TCK Problems and Resolutions

This section lists common problems that you may encounter as you run the Jakarta Platform, Enterprise Edition Test Compatibility Kit software on the Jakarta Platform, Enterprise Edition CI, Eclipse GlassFish 8.0, and other implementations. It also proposes resolutions for the problems, where applicable.

- Problem:

When you start the Jakarta Platform, Enterprise Edition CI, Eclipse GlassFish 8.0 on Windows by using the `javaee -verbose` command, the system may not find the specified path and could display one of the following errors:

```
"Verify that JAVA_HOME is set correctly"
"Verify that JAKARTAEE_HOME is set correctly"
```

Resolution:

Set `JAVA_HOME` to the path where the version of Java being used was installed and set `JAKARTAEE_HOME` to the location of the Jakarta Platform, Enterprise Edition installation directory.

- Problem:

When running the Jakarta Persistence TCK tests you see an error log message containing:

```
LogFileProcessor setup failed
Please verify that the property log.file.location exists in ts.jte

SVR: Log file not found, using default location:/tmp/JPALog.xml
Log File : /tmp/JPALog.xml does not exists
SVR-ERROR: Check permissions for log file
SVR-ERROR: See User guide for Configuring log file permissions
```

Resolution:

You may have the `log.file.location` set in the `ts.jte` file, but if your Jakarta Persistence integration eagerly loads the testcase custom persistence provider, it may do so before the `log.file.location` property is set by the TCK harness code. To resolve this issue, you can set the `log.file.location` system property in your server configuration so the JVM has this property set before a test deployment is processed.

- Problem:

When running signature tests you see the test has failed, but there is no indication as to why.

Resolution:

The signature test driver uses a `java.lang.System.Logger`. Since the tests runs in a deployment, the failure messages are logged via the servers log configuration. If you prefer to see the failure reason with the assertion failure, simply add a console handler/appender to your logging configuration.

- Problem:

When running the Jakarta Platform, Enterprise Edition TCK tests you see an error log message containing:

```
[ERROR] Client2PmservletTest » Validation DeploymentScenario contains a target (tck-javatest) not matching any
defined Container in the registry.
Possible causes are: None of the 1 Containers are marked as default or you have defined a @Deployment with a
@TargetsContainer of value (tck-javatest) that does not match any found/configured Containers (tck-applient), see
arquillian.xml container@qualifier
```

Resolution:

As mentioned in [Arquillian Container Configurations](#), you will need to configuration your Arquillian container configuration to setup one of the Arquillian protocols used in the TCK. If you run a set of tests using the wrong Arquillian protocol, you will see this error. For this `Client2PmservletTest` in the error, an execution configuration that specifies the `tck-javatest` protocol is required. You would need to look at the [Failsafe Plugin Configuration \(javatest Protocol\)](#) section for an example of how to configure the `tck-javatest` protocol.



Further details behind this can be found in this platform-tck issue: <https://github.com/jakartaee/platform-tck/issues/2138>

Support

Jakarta EE is a community sponsored and community supported project. If you need additional assistance, you can reach out to the specific developer community. You will find the list of all Eclipse EE4J projects at <https://projects.eclipse.org/projects/ee4j>. All the sub-projects are listed. Each project page has details regarding how to contact their developer community.

For Jakarta EE TCK specific issues, you can reach out to the Jakarta EE TCK project team via the resources listed at <https://projects.eclipse.org/projects/ee4j.jakartaee-tck>.

Building the Tests

For final certification and branding, you need to run the tests as shipped in the Jakarta TCK EE distribution bundle without modification. However, you can build and run the tests for debugging purposes. The one exception is the rebuildable tests, which you can modify and rebuild.

Build the Platform TCK EE Tests

While the Jakarta TCK EE distribution bundle contains the test sources, they are not in a buildable Maven project form. To obtain a buildable version of the test sources from the Jakarta TCK EE GitHub repository, follow these steps:

1. Clone the Jakarta TCK EE GitHub repository and checkout the release tag of interest:

```
git clone https://github.com/jakartaee/platform-tck.git
cd platform-tck
git checkout jakartaee-tck-{jakartaee_tck_version}
```

2. Set your `JAVA_HOME` environment variable to point to a JDK 17+ installation.
3. Build the test sources:

```
mvn install
```

Your local Maven repository will now have the Jakarta TCK EE tests installed.

Implementing the Porting Package

Some functionality in the Jakarta Platform, Enterprise Edition platform is not completely specified by an API. To handle this situation, the Jakarta EE test suite defines an abstract class

`tck.arquillian.porting.lib.spi.AbstractTestArchiveProcessor`, which serves as a base class for vendor implementation-specific code to augment test deployment artifacts with vendor specific descriptors. A sample implementation is provided by...

In addition, there is a set of interfaces in the `com.sun.cts.porting` package. TODO: which of these interfaces is still in use:
`com.sun.ts.lib.porting.TSLoginContextInterface`
`com.sun.ts.lib.porting.TSURLInterface`
`com.sun.ts.lib.porting.TSJSAdminInterface` (Full Platform Only)
`com.sun.ts.lib.porting.TSURLConnectionInterface`

You must create your own implementations of the porting package abstract classes and interfaces to work with your particular Jakarta Platform, Enterprise Edition server environment.

Overview

The Jakarta Platform, Enterprise Edition CI uses a set of component specific deployment descriptors to configure the runtime files that are associated with each deployable component. GlassFish supports several runtime XML files: `sun-application_1_4-0.xml`, `sun-application-client_1_4-0.xml`, `sun-ejb-jar_2_1-0.xml`, and `sun-web-app_2_4-0.xml`, for vendor specific information.

To specify your implementation of the `tck.arquillian.porting.lib.spi.AbstractTestArchiveProcessor` class, you need to create an Arquillian `org.jboss.arquillian.core.spi.LoadableExtension` service implementation that registers your subclass of the `tck.arquillian.porting.lib.spi.AbstractTestArchiveProcessor` class. An example is shown in the following listing:

```
package wildfly.arquillian;

import org.jboss.arquillian.container.test.spi.client.deployment.ApplicationArchiveProcessor;
import org.jboss.arquillian.core.spi.LoadableExtension;
import org.jboss.arquillian.test.spi.enricher.resource.ResourceProvider;

public class JBossTckExtension implements LoadableExtension {
    @Override
    public void register(ExtensionBuilder builder) {
        builder.service(ResourceProvider.class, JBossXmlProcessor.class);①
        builder.observer(JBossXmlProcessor.class);②
    }
}
```

① Call the `builder.service` method passing in your subclass of `tck.arquillian.porting.lib.spi.AbstractTestArchiveProcessor`.

② Call the `builder.observer` method passing in your subclass of `tck.arquillian.porting.lib.spi.AbstractTestArchiveProcessor`.

In the above listing, the `tck.arquillian.porting.lib.spi.AbstractTestArchiveProcessor` subclass is the `JBossXmlProcessor.class`.

The following `com.sun.cts.porting` porting interfaces may need to be implemented if the defaults are not sufficient:

- `TSLoginContextInterface`
- `TSURLInterface`
- `TSJSAdminInterface` (Full Platform Only)
- `TSHttpURLConnectionInterface` (Full Platform Only)

To use specific implementations of these classes, you simply modify the following `porting.ts.*.class.1` entries of the `ts.jte` environment file to identify the fully-qualified class names:

```
porting.ts.login.class.1=[vendor-login-class]①
porting.ts.url.class.1=[vendor-url-class]②
porting.ts.jms.class.1=[vendor-jms-class]③
porting.ts.HttpURLConnection.class
.1=[vendor-httpsURLConnection-class]④
```

- ① [TSLoginContextInterface](#)
- ② [TSURLInterface](#)
- ③ [TSJMSAdminInterface \(Full Platform Only\)](#)
- ④ [TSHttpsURLConnectionInterface \(Full Platform Only\)](#)

You can find these interfaces in the `jakarta.tck:common` artifact and the `src/tools/common/src/main/java/com/sun/ts/lib/porting` directory of the TCK distribution.

Include the implementation of the previous interfaces in the classpaths of the test clients, and the test server components in the classpath of your Jakarta Platform, Enterprise Edition server

Note that because the test harness VM calls certain classes in the TCK porting package directly, porting class implementations are not permitted to exit the VM (for example, by using the `System.exit` call).

Additional implementation details for the porting package are provided in the following sections.

Porting Package APIs

The following sections describe the API in the Jakarta EE 11 Platform TCK porting package. The implementation classes used with the Jakarta Platform, Enterprise Edition CI are located in the <https://github.com/jakartaee/platform-tck/tree/main/tools/common/src/main/java/com/sun/ts/lib/porting/implementation> directory. You are encouraged to examine these implementations before you create your own.

Detailed API documentation for the porting package interfaces is available in the `<TS_HOME>/docs/api` directory. The API included in this section are:

- [AbstractTestArchiveProcessor](#)
- [TSJMSAdminInterface \(Full Platform Only\)](#)
- [TSLoginContextInterface](#)
- [TSURLInterface](#)
- [TSHttpsURLConnectionInterface \(Full Platform Only\)](#)

AbstractTestArchiveProcessor

The test harness test classes use Arquillian to deploy test artifacts to the containers of the Jakarta EE server being tested. The deployment methods include a `TestArchiveProcessor` interface as shown in the following code fragment:

```
@ExtendWith(ArquillianExtension.class)
@Tag("platform")
@Tag("ejb_remote_async_optional")
@Tag("web_optional")
@Tag("tck-javatest")

@TestMethodOrder(MethodOrderer.MethodName.class)
public class JsfcClientEjblitejsfTest extends com.sun.ts.tests.ejb30.bb.async.stateful.metadata.JsfcClient {
    static final String VEHICLE_ARCHIVE = "ejbbb_async_stateful_metadata_ejblitejsf_vehicle";
}
```

```

...
    @TargetsContainer("tck-javatest")
    @OverProtocol("javatest")
    @Deployment(name = VEHICLE_ARCHIVE, order = 2)
    public static WebArchive createDeploymentVehicle(@ArquillianResource TestArchiveProcessor archiveProcessor)
{
    ...
}

```

The TestArchiveProcessor interface is what the AbstractTestArchiveProcessor abstract class vendors should subclass implements. The TestArchiveProcessor interface method of interest are shown in the following code fragment:

```

public interface TestArchiveProcessor {
    /**
     * Called to process a client archive (jar) that is part of the test deployment.
     * @param clientArchive - the appclient archive
     * @param testClass - the TCK test class
     * @param sunXmlUrl - the URL to the sun-application-client.xml file
     */
    void processClientArchive(JavaArchive clientArchive, Class<?> testClass, URL sunXmlUrl);
    /**
     * Called to process a ejb archive (jar) that is part of the test deployment.
     * @param ejbArchive - the ejb archive
     * @param testClass - the TCK test class
     * @param sunXmlUrl - the URL to the sun-ejb-jar.xml file
     */
    void processEjbArchive(JavaArchive ejbArchive, Class<?> testClass, URL sunXmlUrl);
    /**
     * Called to process a web archive (war) that is part of the test deployment.
     * @param webArchive - the web archive
     * @param testClass - the TCK test class
     * @param sunXmlUrl - the URL to the sun-web.xml file
     */
    void processWebArchive(WebArchive webArchive, Class<?> testClass, URL sunXmlUrl);
    /**
     * Called to process a resource adaptor archive (rar) that is part of the test deployment.
     * @param rarArchive - the resource archive
     * @param testClass - the TCK test class
     * @param sunXmlUrl - the URL to the sun-ra.xml file
     */
    void processRarArchive(JavaArchive rarArchive, Class<?> testClass, URL sunXmlUrl);
    /**
     * Called to process a persistence unit archive (par) that is part of the test deployment.
     * @param parArchive - the resource archive
     * @param testClass - the TCK test class
     * @param persistenceXmlUrl - the URL to the sun-ra.xml file
     */
    void processParArchive(JavaArchive parArchive, Class<?> testClass, URL persistenceXmlUrl);
    /**
     * Called to process an enterprise archive (ear) that is part of the test deployment.
     * @param earArchive - the application archive
     * @param testClass - the TCK test class
     * @param sunXmlUrl - the URL to the sun-application.xml file
     */
    void processEarArchive(EnterpriseArchive earArchive, Class<?> testClass, URL sunXmlUrl);
}

```

For each type of Jakarta EE component archive that is included in a test deployment, one or more of these methods will be called with the component archive, the test harness test class, and a possibly null URL for the GlassFish/Sun version of the vendor descriptor. Not all test deployments include a GlassFish/Sun version of the vendor descriptor. Those that do not will pass in a null descriptor URL. Vendors could choose to transform the GlassFish version of the descriptor, or use some other scheme such as the test package/class name to locate their equivalent vendor specific descriptor.

TSJMSAdminInterface (Full Platform Only)

Jakarta Messaging-administered objects are implementation-specific. For this reason, the creation of connection factories and destination objects have been set up as part of the porting package. Each Jakarta Platform, Enterprise Edition implementation must provide an implementation of the `TSJMSAdminInterface` to support their own connection factory, topic/queue creation/deletion semantics.

The `TSJMSAdmin` class acts as a Factory object for creating concrete implementations of `TSJMSAdminInterface`. The concrete implementations are specified by the `porting.ts.jms.class.1` and `porting.ts.jms.class.2` properties in the `ts.jte` file.

There are two types of Jakarta Messaging-administered objects:

1. A `ConnectionFactory`, which a client uses to create a connection with a JMS provider
2. A `Destination`, which a client uses to specify the destination of messages it sends and the source of messages it receives

TSLoginContextInterface

The `TSLoginContext` class acts as a Factory object for creating concrete implementations of `TSLoginContextInterface`. The concrete implementations are specified by the `porting.ts.login.class.1` property in the `ts.jte` file. This class is used to enable a program to login as a specific user, using the semantics of the Jakarta Platform, Enterprise Edition CI. The certificate necessary for certificate-based login is retrieved. The keystore file and keystore password from the properties that are specified in the `ts.jte` file are used.

TSURLInterface

The `TSURL` class acts as a Factory object for creating concrete implementations of `TSURLInterface`. The concrete implementations are specified by the `porting.ts.url.class.1` property in the `ts.jte` file.

Each Jakarta Platform, Enterprise Edition implementation must provide an implementation of the `TSURLInterface` to support obtaining URL strings that are used to access a selected Web component. This implementation can be replaced if a Jakarta Platform, Enterprise Edition server implementation requires URLs to be created in a different manner. In most Jakarta Platform, Enterprise Edition environments, the default `com.sun.ts.lib.porting.implementation.SunRIURL` implementation of this class from the `jakarta.tck:common` artifact can be used.

TSHttpsURLConnectionInterface (Full Platform Only)

The `TSHttpsURLConnection` class acts as a Factory object for creating concrete implementations of `TSHttpsURLConnectionInterface`. The concrete implementations are specified by the `porting.ts.HttpsURLConnection.class.1` and `.2` properties in the `ts.jte` file.

You must provide an implementation of `TSHttpsURLConnectionInterface` to support the class `HttpsURLConnection`.



The `SunRIHttpsURLConnection` implementation class uses `HttpsURLConnection` from Java SE 17.

Jakarta TCK Test Appeals Process

Jakarta has a well established process for managing challenges to its TCKs. Any implementor may submit a challenge to one or more tests in the Jakarta EE TCK as it relates to their implementation. Implementor means the entity as a whole in charge of producing the final certified release. **Challenges filed should represent the consensus of that entity.**

Valid Challenges

Any test case (e.g., test class, @Test method), test case configuration (e.g., deployment descriptor), test beans, annotations, and other resources considered part of the TCK may be challenged.

The following scenarios are considered in scope for test challenges:

- Claims that a test assertion conflicts with the specification.
- Claims that a test asserts requirements over and above that of the specification.
- Claims that an assertion of the specification is not sufficiently implementable.
- Claims that a test is not portable or depends on a particular implementation.

Invalid Challenges

The following scenarios are considered out of scope for test challenges and will be immediately closed if filed:

- Challenging an implementation's claim of passing a test. Certification is an honor system and these issues must be raised directly with the implementation.
- Challenging the usefulness of a specification requirement. The challenge process cannot be used to bypass the specification process and raise in question the need or relevance of a specification requirement.
- Claims the TCK is inadequate or missing assertions required by the specification. See the Improvement section, which is outside the scope of test challenges.
- Challenges that do not represent a consensus of the implementing community will be closed until such time that the community does agree or agreement cannot be made. The test challenge process is not the place for implementations to initiate their own internal discussions.
- Challenges to tests that are already excluded for any reason.
- Challenges that an excluded test should not have been excluded and should be re-added should be opened as a new enhancement request

Test challenges must be made in writing via the {TechnologyShortName} specification project issue tracker as described in [TCK Test Appeals Steps](#)

All tests found to be invalid will be added to the Excluded Tests for that version of the {TechnologyShortName} TCK.

TCK Test Appeals Steps

1. Challenges should be filed via the Jakarta EE Platform specification project's issue tracker using the label challenge and include the following information:
 - The relevant specification version and section number(s)
 - The coordinates of the challenged test(s)
 - The exact TCK version
 - The implementation being tested, including name and company
 - The full test name

- A full description of why the test is invalid and what the correct behavior is believed to be
 - Any supporting material; debug logs, test output, test logs, run scripts, etc.
2. Specification project evaluates the challenge.

Challenges can be resolved by a specification project lead, or a project challenge triage team, after a consensus of the specification project committers is reached or attempts to gain consensus fails. Specification projects may exercise lazy consensus, voting or any practice that follows the principles of Eclipse Foundation Development Process. The expected timeframe for a response is two weeks or less. If consensus cannot be reached by the specification project for a prolonged period of time, the default recommendation is to exclude the tests and address the dispute in a future revision of the specification.
 3. Accepted Challenges.

A consensus that a test produces invalid results will result in the exclusion of that test from certification requirements, and an immediate update and release of an official distribution of the TCK including the new excluded tests. The associated `challenge` issue must be closed with an `accepted` label to indicate it has been resolved.
 4. Rejected Challenges and Remedy.

When a `challenge` issue is rejected, it must be closed with a label of `invalid` to indicate it has been rejected. There appeal process for challenges rejected on technical terms is outlined in Escalation Appeal. If, however, an implementer feels the TCK challenge process was not followed, an appeal issue should be filed with specification project's TCK issue tracker using the label `challenge-appeal`. A project lead should escalate the issue with the Jakarta EE Specification Committee via email (jakarta.ee-spec@eclipse.org). The committee will evaluate the matter purely in terms of due process. If the appeal is accepted, the original TCK challenge issue will be reopened and a label of `appealed-challenge` added, along with a discussion of the appeal decision, and the `challenge-appeal` issue will be closed. If the appeal is rejected, the `challenge-appeal` issue should be closed with a label of `invalid`.
 5. Escalation Appeal.

If there is a concern that a TCK process issue has not been resolved satisfactorily, the [Eclipse Development Process Grievance Handling](#) procedure should be followed to escalate the resolution. Note that this is not a mechanism to attempt to handle implementation specific issues.

A Common Applications Deployment (Needs Rewrite)

TODO: figure out if Common Applications Deployment tests should be removed since they cannot rely on (removed) Jakarta Deployment in EE 9.

Some tests in the test suite require the deployment of additional applications, components, or resource archives that are located in directories other than the test's directory.

[Table A-1 Required Common Applications](#) lists the test directories and the directories that contain the common applications that are required by the test directories.

Table A-1 Required Common Applications

Directory Under <code>com/sun/ts/tests</code>	Directory Under <code>com/sun/ts/tests</code> With Associated Common Applications
<code>ejb/ee/tx/session</code>	<code>ejb/ee/tx/txbean</code>
<code>ejb/ee/tx/entity/pm</code>	<code>ejb/ee/tx/txEPMbean</code>
<code>connector/ee/localTx/msginflow</code>	<code>common/connector/whitebox</code>
<code>connector/ee/mdb</code>	<code>connector/ee/localTx</code>
<code>common/connector/whitebox</code>	<code>connector/ee/noTx</code>
<code>common/connector/whitebox</code>	<code>connector/ee/xa</code>
<code>common/connector/whitebox</code>	<code>connector/ee/connManager</code>
<code>common/connector/whitebox</code>	<code>xa/ee</code>
<code>compat13/connector/localTx</code>	<code>compat13/connector/whitebox</code>
<code>compat13/connector/noTx</code>	<code>compat13/connector/whitebox</code>
<code>compat13/connector/xa</code>	<code>compat13/connector/whitebox</code>
<code>interop/tx/session</code>	<code>interop/tx/txbean</code>
<code>interop/tx/entity</code>	<code>interop/tx/txEbean</code>
<code>interop/tx/webclient</code>	<code>interop/tx/txbean</code>
<code>ejb/ee/pm/ejbql</code>	<code>ejb/ee/pm/ejbql/schema</code>

Configuring Your Backend Database

This appendix explains how to configure a backend database to use with a Jakarta Platform, Enterprise Edition server being tested against the Jakarta EE 11 TCK.

The topics included in this appendix are as follows:

- [Overview](#)
- [The `init.<database>` Ant Target](#)
- [Database Properties in `ts.jte`](#)
- [Database DDL and DML Files](#)

Overview

All Jakarta Platform, Enterprise Edition servers tested against the Jakarta EE 11 TCK must be configured with a database and JDBC 4.1-compliant drivers. Note that the Jakarta Platform, Enterprise Edition CI, Eclipse GlassFish 8.0 includes the Apache Derby database.

To perform interoperability testing, you need to configure two Jakarta Platform, Enterprise Edition servers and two databases, one of which must be the Jakarta Platform, Enterprise Edition RI with the bundled Apache Derby database. See [Jakarta Platform, Enterprise Edition Server Configuration Scenarios](#) for more information.

For the purposes of Jakarta EE 11 Platform TCK testing, all database configuration properties required by the TCK are made in the `<TS_HOME>/bin/ts.jte` file. The TCK `init.<`database>` Ant target of the `<TS_HOME>/bin/xml/initdb.xml` file uses the properties you set in `ts.jte` to generate one or more SQL statement files that are in turn used create and populate database tables and configure procedures required by the TCK. You don't need to explicitly use this script, however you will have to achieve the same effect via other means if you do not use the script.

The database configuration process comprises four general steps:

1. Set database-related properties in the `<TS_HOME>/bin/ts.jte` file.
2. Configure your Jakarta Platform, Enterprise Edition server implementation for your database and for TCK.
3. Start your database.
4. Run the `init.<database>` Ant target to initialize your database for TCK where `<database>` is the name of your database, e.g. `init.derby`, `init.javadb`, `init.postgresql` etc.

The procedure for configuring your Jakarta Platform, Enterprise Edition server for your database is described in [Configuring a Jakarta EE 11 Server](#). The final step, initializing your database for TCK by running `init.<`database>` target, is explained more in the next section.

The `init.<database>` Ant Target

Before your Jakarta Platform, Enterprise Edition server database can be tested against the Jakarta EE 11 Platform TCK, the database must be initialized for TCK by means of the Ant `init.<database>` target. For example, the `init.javadb` Ant task is used to initialize the Apache Derby database for TCK.

This Ant target references database properties in `ts.jte` file and database-specific DDL and DML files located under `<TS_HOME>/bin/sql` to generate SQL statement files that are read by the Jakarta EE 11 Platform TCK when you start the test suite. The DDL and DML files are described later in this appendix, in [Database DDL and DML Files](#).

The Jakarta EE 11 Platform TCK includes the following database-specific Ant targets (possibly more):

- `init.cloudscape`

- `init.db2`
- `init.oracle`
- `init.oracleDD`
- `init.oracleInet`
- `init.derby`
- `init.javadb`
- `init.sybase`
- `init.sybaseInet`
- `init.mssqlserver`
- `init.mssqlserverInet`
- `init.mssqlserverDD`
- `init.postgresql`

Each Ant target uses a database-specific JDBC driver to configure a backend for a specific database; for example, OracleInet/Oracle Inet driver; OracleDD/Oracle DataDirect driver. These targets are configured in the `<TS_HOME>/xml/initdb.xml` file.

Database Properties in ts.jte

Listed below are the names and descriptions for the database properties you need to set for TCK testing.

Note that some properties take the form `property`.ri``. In all cases, properties with an `.ri` suffix are used for interoperability testing only. In such cases, the property value applies to the Jakarta Platform, Enterprise Edition VI server (the server you want to test) and the property `.ri`` value applies to the Jakarta Platform, Enterprise Edition CI, Eclipse GlassFish 8.0 server. For example:

```
db.dml.file=VI_DML_filename
db.dml.file.ri=RI_DML_filename
```

The property `.ri`` properties are only used in two-server configurations; that is, when you are performing interoperability tests.

Table D-1 ts.jte Database Properties

Property	Description
<code>database`.classes`</code>	CLASSPATH to JDBC driver classes.
<code>database`.dataSource`</code>	DataSource driver.
<code>database`.dbName`</code>	Database Name.
<code>database`.driver`</code>	DriverManager driver.
<code>database`.password`</code>	User password configured.
<code>database`.poolName`</code>	Name of pool configured in the RI (do not change!).
<code>database`.port`</code>	Database Server port.

Property	Description
database`.properties`	Additional properties required by the defined data source for each driver configuration in <code>ts.jte</code> . You should not need to modify this property.
database`.server`	Database Server.
database`.url`	URL for the TCK database; the <code>dbName</code> , <code>server</code> , and <code>port</code> properties are automatically substituted in to build the correct URL. You should never need to modify this property.
database`.user`	User ID configured.
db.dml.file	Tells <code>init.`database`</code> which DML file to use for the VI database; for example, <code>`db.dml.file=\${javadb.dml.file}</code> .
db.dml.file.ri	Tells <code>init.`database`</code> which DML file to use for the RI database; for example, <code>`db.dml.file=\${javadb.dml.file}</code> .
jdbc.lib.class.path	Used by the database`.classes` properties to point to the location of the JDBC drivers.
jdbc.poolName	Configures the connection pool that will be used in the TCK test run; for example, <code>jdbc.poolName=\${javadb.poolName}</code> . Set this property when running against the RI if using a database other than Apache Derby.
password1	Password for the JDBC/DB1 resource; for example, <code>password1=\${javadb.passwd}</code> .
password2	Password for the JDBC/DB2 resource; for example, <code>password2=\${javadb.passwd}</code> .
password3	Password for the JDBC/DBTimer resource; for example, <code>password3=\${javadb.passwd}</code> .
user1	User name for the JDBC/DB1 resource; for example, <code>user1=\${javadb.user}</code> .
user2	User name for the JDBC/DB2 resource; for example, <code>user2=\${javadb.user}</code> .
user3	User name for the JDBC/DBTimer resource; for example, <code>user3=\${javadb.user}</code> .

Database DDL and DML Files

For each supported database type, the Jakarta EE 11 Platform TCK includes a set of DDL and DML files in subdirectories off the `<TS_HOME>/bin/sql` directory. The `config.vi` and `config.ri` targets use two `ts.jte` properties, `db.dml.file` and `db.dml.file.ri` (interop only), to determine the database type, and hence which database-specific DML files to copy as `<TS_HOME>/bin/tssql.stmt` and `tssql.stmt.ri` (for interop) files.

The `tssql.stmt` and `tssql.stmt.ri` files contain directives for configuring and populating database tables as required by the TCK tests, and for defining any required primary or foreign key constraints and database-specific command line terminators.

In addition to the database-specific DML files, the Jakarta EE 11 Platform TCK includes database-specific DDL files, also in subdirectories off `<TS_HOME>/bin/sql`. These DDL files are used by the ``init.`database` target to create and drop database tables and procedures required by the TCK.

The SQL statements in the `tssql.stmt` and `tssql.stmt.ri` files are read as requested by individual TCK tests, which use the statements to locate required DML files.

The DDL and DML files are as follows:

- database ``ddl.sql``: DDL for BMP, Session Beans
- database ``ddl.sprocs.sql``: DDL for creating stored procedures
- database ``ddl.interop.sql``: DDL for interop tests
- database ``dml.sql``: DML used during test runs

Each DDL command in each `<TS_HOME>/sql/`database` is terminated with an ending delimiter. The delimiter for each database is defined in the ``<TS_HOME>/bin/xml/initdb.xml` file. If your configuration requires the use of a database other than the databases that `initdb.xml` currently supports, you may modify `initdb.xml` to include a target to configure the database that you are using.

An example of the syntax for a database target in `initdb.xml` is shown below:

```
<target name="init.sybase">
  <antcall target="configure.backend">
    <param name="db.driver" value="${sybase.driver}"/>
    <param name="db.url" value="${sybase.url}"/>
    <param name="db.user" value="${sybase.user}"/>
    <param name="db.password" value="${sybase.passwd}"/>
    <param name="db.classpath" value="${sybase.classes}"/>
    <param name="db.delimiter" value="!"/>
    <param name="db.name" value="sybase" />
  </antcall>
</target>
```

The database ``name`` property should be added to your `ts.jte` file. The `db.name` property is the name of a subdirectory in `<TS_HOME>/sql`. After updating `initdb.xml`, you invoke the new target with:

```
ant -f <TS_HOME>/bin/xml/initdb.xml init.databasesname
```

Testing a Standalone Jakarta Messaging Resource Adapter (Full Platform Only)

This appendix explains how to set up and configure a Jakarta EE 11 CI and Jakarta EE 11 Platform TCK so a standalone Jakarta Messaging resource adapter can be tested.

This appendix covers the following topics:

- [Setting Up Your Environment](#)
- [Configuring Jakarta EE 11 Platform TCK](#)
- [Configuring a Jakarta EE 11 CI for the Standalone Jakarta Messaging Resource Adapter](#)
- [Modifying the Runtime Deployment Descriptors for the Jakarta Messaging MDB and Resource Adapter Tests](#)
- [\[f.5-running-the-jakarta-messaging-tests-from-the-command-line\]](#)
- [\[f.6-restoring-the-runtime-deployment-descriptors-for-the-jakarta-messaging-mdb-and-resource-adapter-tests\]](#)
- [\[f.7-reconfiguring-jakarta-ee-8-CI-for-jakarta-ee-11-platform-tck-after-testing-the-standalone-jakarta-messaging-resource-adapter\]](#)

Setting Up Your Environment

Before you can run the Jakarta Messaging TCK tests against a standalone Jakarta Messaging Resource Adapter using a Jakarta EE 11 CI, you must install the following components:

- Java SE 17 or 21 software
- Jakarta EE 11 CI software such as Eclipse GlassFish 8.0
- Jakarta EE 11 Platform TCK software

Complete the following steps to set up Eclipse GlassFish 8.0 in your environment:

1. Set the following environment variables in your shell environment:
 - JAVA_HOME to the directory where the Java SE software has been installed
 - JAKARTAE_HOME to the directory where the Jakarta EE 11 CI (Eclipse GlassFish 8.0) software has been installed
 - TS_HOME to the directory where the Jakarta EE 11 Platform TCK software has been installed
2. Update your PATH environment variable to include the following directories: JAVA_HOME/bin, JAKARTAE_HOME/bin, TS_HOME/bin, and ANT_HOME/bin.

Configuring Jakarta EE 11 Platform TCK

The `ts.jte` file includes properties that must be set for testing a standalone Jakarta Messaging Resource Adapter using the Jakarta EE 11 CI. The Jakarta Messaging Resource Adapter documentation in the `ts.jte` file should help you understand what you need to set in this step of the testing process.

1. Set the following properties in the `ts.jte` file:
 - `javaee.home` to the location where the Jakarta EE 11 CI is installed
 - Use the default value or enter a new host name for the `orb.host` property
 - Use the default value or enter a new port number for the `orb.port` property
 - `test.sa.jmsra` to `true`
 - `jmsra.rarfile` to the location of the standalone Jakarta Messaging Resource Adapter RAR file
 - `jmsra.jarfile` to the location of the standalone Jakarta Messaging Resource Adapter JAR file
 - `jmsra.name` to the name of the Jakarta Messaging Resource Adapter under test
2. Add `${jmsra.jarfile}` to the beginning or at the end of the `AppClient` classpath:

APPCPATH= list of classes and jars followed by `${pathsep}${jmsra.jarfile}\`

The `jmsra.jarfile`, which contains all the Jakarta Messaging Resource Adapter classes, needs to be added to the `AppClient` classpath in your `arquillian.xml` for your `appclient` container.

file. This JAR file should also be copied to the appropriate directory in your Jakarta EE 11 environment.

Configuring a Jakarta EE 11 CI for the Standalone Jakarta Messaging Resource Adapter

Review the `glassfish-runner/messaging-tck` maven runner the the `platform-tck` repo for the Jakarta Messaging Resource Adapter setup tasks.

Modifying the Runtime Deployment Descriptors for the Jakarta Messaging MDB and Resource Adapter Tests

After the standalone Jakarta Messaging Resource Adapter has been configured and deployed and the required Jakarta Messaging connector connection pools, Jakarta Messaging connector resources, and Jakarta Messaging administration objects have been created, the `sun-ejb-jar` runtime deployment descriptor XML files must be modified for the Jakarta Messaging MDB and Resource Adapter tests.

The descriptor XML files in the distribution directory of the Jakarta Messaging MDB and Resource Adapter test directories that exist under the `com.sun.ts.tests.jms.ee.mdb` and `com.sun.ts.tests.jms.ee20.ra` packages of the `jakarta.tck:jms-platform-tck` artifact.