Jakarta MVC 2.1 Technology Compatibility Kit

Documentation & Usage Guide

Jakarta MVC Project

Version 2.1.0

Table of Contents

Chapter 1. Introduction

This guide describes how to download, install and run the Technology Compatibility Kit (TCK) for the Jakarta MVC Specification 2.1.

1.1. TCK Primer

A TCK, or Technology Compatibility Kit, is one of the three required pieces for any JSR (the other two being the specification document and the reference implementation). The TCK is a set of tools and tests to verify that an implementation of the technology conforms to the specification. The tests are the primary component, but the tools serve an equally critical role of providing a framework and/or set of SPIs for executing the tests.

The tests in the TCK are derived from assertions in the written specification document. The assertions are itemized in an XML document, where they each get assigned a unique identifier, and materialize as a suite of automated tests that collectively validate whether an implementation complies with the aforementioned assertions, and in turn the specification. For a particular implementation to be certified, all the required tests must pass (meaning the provided test suite must be run unmodified).

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

1.2. Compatibility Testing

The goal of any specification is to eliminate portability problems so long as the program which uses the implementation also conforms to the rules laid out in the specification.

Executing the TCK is a form of compatibility testing. It's important to understand that compatibility testing is distinctly different from product testing. The TCK is not concerned with robustness, performance or ease of use, and therefore cannot vouch for how well an implementation meets these criteria. What a TCK can do is to ensure the exactness of an implementation as it relates to the specification.

Compatibility testing of any feature relies on both a complete specification and a complete reference implementation. The reference implementation demonstrates how each test can be passed and provides additional context to the implementor during development for the corresponding assertion.

1.3. About the MVC TCK

The Jakarta MVC 2.1 TCK is designed as a portable, configurable and automated test suite for verifying the compatibility of an implementation of the Model-View-Controller Specification. The test suite is built atop JUnit and provides a series of extensions that allow runtime packaging and deployment of Java EE artifacts for in-container testing (Arquillian).

Each test class in the suite acts as a deployable unit. The deployable units, or artifacts, are defined

in a declarative way using annotations.

1.4. TCK Components

The Jakarta MVC 2.1 TCK includes the following components:

- **The test suite**, which is a collection of JUnit tests and supplemental resources that configure the runtime and other software components.
- **The TCK audit** (`tck-audit.xml`) used to list out the assertions identified in the Jakarta MVC 2.1 specification. It matches the assertions to test cases in the test suite by unique identifier and produces a coverage report. The audit document is provided along with the TCK. Each assertion is defined with a reference to a chapter, section and paragraph from the specification document, making it easy for the implementor to locate the language in the specification document that supports the feature being tested.
- A **setup example** demonstrating Maven and Ant setups to run the TCK test suite.

1.5. Passing the Jakarta MVC 2.1 TCK

In order to pass the Jakarta MVC 2.1 TCK, you need to:

- Pass the Jakarta MVC 2.1 signature tests (see Running the signature tests) asserting the correctness of the Jakarta MVC 2.1 API used.
- Run and pass the test suite (see Running the TCK tests).

Chapter 2. Appeals Process

As stated by the TCK process,window=*blank*, _Specifications are the sole source of truth and considered overruling to the TCK in all senses. Therefore, during the implementation of a specification it can happen, that the implementing party is considering a test of this TCK as non-conform to the specification. To get this problem resolved, the implementing party can create a *challenge* for the test.

To file a challenge, please create an issue in the MVC TCK issue tracker,window=_blank} as described in the link:https://jakarta.ee/committees/specification/tckprocess#_filing_a_challenge[TCK process - Filing a challenge,window=_blank.

Chapter 3. Coverage report

A specification can be distilled into a collection of assertions that define the behavior of the software. This section introduces the Jakarta MVC 2.1 TCK coverage report, which documents the relationship between the assertions that have been identified in the Jakarta MVC 2.1 specification document and the tests in the TCK test suite.

The structure of this report is controlled by the assertion document, so we'll start there.

3.1. Jakarta MVC 2.1 TCK Assertions

The Jakarta MVC 2.1 TCK developers have analyzed the Jakarta MVC 2.1 specification document and identified the assertions that are present in each chapter.

The assertions are listed in an XML file which is generated from the specification document. Each assertion is identified by the section of the specification document in which it resides and a unique paragraph identifier to narrow down the location of the assertion further.

See the following example of an example:

```
<section id="controllers" title="Controllers" level="2">
    <!-- ... -->
    <assertion id="ctrl-method">
        <text>An MVC controller is a JAX-RS resource method decorated by @Controller</text>
    </assertion>
    <!-- ... -->
</section>
```

The strategy of the Jakarta MVC 2.1 TCK is to write a test which validates this assertion when run against an implementation. A test case (a method annotated with `@Test`) is correlated with an assertion using the `@org.jboss.test.audit.annotations.SpecAssertion` annotation as follows:

```
@RunWith(Arquillian.class)
@SpecVersion(spec = "mvc", version = "1.0")
public class ControllerAnnotationTest {

    /* more tests */

    @Test
    @SpecAssertion(section = Sections.MVC_CONTROLLERS, id = "ctrl-method")
    public void controllerMethod() throws IOException {
        // test implementation
    }

    /* more tests */

}
```

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

3.2. The Coverage Report

The coverage report is an HTML report generated as part of the TCK project build. Specifically, it is generated by an annotation processor that attaches to the compilation of the classes in the TCK test suite. The report is written to the directory `tests/target/coverage-report/`.

The report itself has three main sections:

Chapter Summary

List the chapters in the specification document along with total assertions, tests and coverage percentage.

Section Summary

Lists the sections in the specification document along with total assertions, tests and coverage percentage.

Coverage Detail

Each assertion and the test that covers it, if any.

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

Covered

Test exists for this assertion

Not covered

No test exists for this assertion

Unimplemented

A test exists, but is unimplemented

Untestable

the assertion has been deemed untestable, a note, explaining why, is normally provided

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

Chapter 4. Running the TCK tests

This chapter describes how to run and configure the TCK test suite against a given Jakarta MVC 2.1 implementation in a given Java EE container. The testsuite uses Arquillian to execute tests against real Java EE containers. It is strongly recommended making yourself familiar with the Arquillian documentation. It will give you a deeper understanding of the different parts described in the following sections.

4.1. The TCK runner template

The TCK contains a directory `sample` which contains an example project for running the TCK and which should be used as a template for creating new projects. This sample project is using the reference implementation Eclipse Krazo and runs the TCK against Eclipse Glassfish 6.0.0.

The project is using Apache Maven is contains just three files:

`pom.xml`

: The Maven POM file imports the TCK BOM, declares all required dependencies and configure the Maven Surefire Plugin to run the tests defined by the Jakarta MVC 2.1 TCK.

`arquillian.xml`

: This file configures Arquillian to deploy the web applications which are used by the TCK tests to Eclipse Glassfish.

`KrazoGlassfishProvider.java`

: This Java class is an implementation of the `ee.jakarta.tck.mvc.api.BaseArchiveProvider` interface which is part of the TCK. The responsibility of the archive provider is to create a base Shrinkwrap archive which is used by all TCK tests. The archive produced by the provider depends on the specific environment you are running the tests against. As Jakarta MVC 2.1 API and implementation is not provided by Eclipse Glassfish, the `KrazoGlassfishProvider` builds and archive which adds the API JAR and Eclipse Krazo to the `/WEB-INF/lib` directory of the archive so that both are deployed as part of the web application.

Running the TCK is simple. First you have to download, unpack and start Eclipse GlassFish 6.0.0. Then just execute the following command in the sample project folder to run the TCK against this Eclipse GlassFish 6.0.0 instance:

```
mvn verify
```

After a few minutes you should get an output like this:

```
[...]
Running org.mvcspec.tck.tests.mvc.controller.mediatype.MediaTypeTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.109 sec - in
org.mvcspec.tck.tests.mvc.controller.mediatype.MediaTypeTest
Running org.mvcspec.tck.tests.mvc.controller.inject.InjectParamsTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.047 sec - in
org.mvcspec.tck.tests.mvc.controller.inject.InjectParamsTest

Results :

Tests run: 132, Failures: 0, Errors: 0, Skipped: 0

[...]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:02 min
[INFO] Finished at: 2019-11-15T08:21:54+01:00
[INFO] ------------------------------------------------------------------------
```

Congratulations! You just ran the Jakarta MVC 2.1 TCK!

4.2. Adjusting the template project

If you want to run the Jakarta MVC 2.1 TCK against a different environment, you should start from the template project and adjust it like described in the following sections.

4.3. Update Arquillian configuration

If you want to run the TCK against a different container, you will typically have to adjust the Arquillian configuration. In most cases this is really easy, because Arquillian already supports a wide range of containers. However, it is strongly recommended having a deeper look at the Arquillian documentation to learn more.

Generally, you will have to follow these steps:

- Replace the `arquillian-glassfish-remote-3.1` Arquillian adapter with the adapter for your environment in the `pom.xml` file.
- Then adjust the `arquillian.xml` file as described in the Arquillian adapter documentation. This typically includes adjusting port numbers and providing credentials for performing remote deployments.

As mentioned above, the details depend on the specific environment.

4.4. Provide a custom BaseArchiveProvider

The `BaseArchiveProvider` is used by the TCK to create a base web application archive capable of running Jakarta MVC 2.1 application in the specific target environment. The TCK then just adds controllers, views and other artifacts to that base archive and deploys it to the target container.

As the template project runs the TCK against Eclipse Glassfish 6.0.0 which doesn't provide support

for Jakarta MVC 2.1 out of the box, the corresponding archive provider creates a web application archive which includes both the Jakarta MVC 2.1 API JAR and the reference implementation Eclipse Krazo in `/WEB-INF/lib`.

The default implementation provided with the template project looks like this:

```
public class KrazoGlassfishProvider implements BaseArchiveProvider {

    @Override
    public WebArchive getBaseArchive() {

        File[] dependencies = Maven.resolver()
                .resolve(
                        "javkarta.mvc:jakarta.mvc-api:2.1.0",
                        "org.eclipse.krazo:krazo-core:3.0.0",
                        "org.eclipse.krazo:krazo-jersey:3.0.0"
                )
                .withoutTransitivity()
                .asFile();

        return ShrinkWrap.create(WebArchive.class)
                .addAsLibraries(dependencies);

    }

}
```

You will have to create a similar class which does the same for the environment you want to run the test against.

If your container provides Jakarta MVC 2.1 support out of the box, you would have to create an implementation which returns an empty archive like this:

```
public class EmptyArchiveProvider implements BaseArchiveProvider {

    @Override
    public WebArchive getBaseArchive() {
        return ShrinkWrap.create(WebArchive.class);
    }

}
```

The TCK uses a Java system property to learn about the implementation that should be used for building archives. This system property is configured using the Maven Surefire Plugin configuration in your `pom.xml`. The relevant section in the template project looks like this:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.19.1</version>
    <configuration>
        <dependenciesToScan>jakarta.mvc.tck:mvc-tck-tests</dependenciesToScan>
        <systemProperties>
            <BaseArchiveProvider>
                jakarta.mvc.tck.runner.KrazoGlassfishProvider
            </BaseArchiveProvider>
        </systemProperties>
    </configuration>
</plugin>
```

If you provide a custom implementation of `ArchiveBaseProvider`, you will have to adjust the configuration and change the FQCN if the implementation class.

Chapter 5. Running the signature tests

One of the requirements of an implementation passing the TCK is for it to pass the Jakarta MVC 2.1 signature test. This section describes how to run it against your implementation.

5.1. Obtaining the sigtest tool

To run the signature tests against an implementation, you will need the Sigtest tool. You can download the latest version here:

https://download.java.net/sigtest/download.html

The process has been tested against `sigtest-3_0-dev-bin-b09-24_apr_2013.zip`.

Download the corresponding ZIP file and unpack it to some location:

```
mkdir ${HOME}/sigtest && cd $_
wget http://download.java.net/sigtest/3.0/PreRel/sigtest-3_0-dev-bin-b09-24_apr_2013.zip
unzip sigtest-3_0-dev-bin-b09-24_apr_2013.zip
```

Now set the following environment variables:

`SIGTEST_HOME`
: Directory of the just downloaded Sigtest tool.

`JAVA_HOME`
: Location of the JDK you want to use for the test.

If you use the installation directory from the example above:

```
export SIGTEST_HOME=${HOME}/sigtest/sigtest-3.0
export JAVA_HOME=/lib/jvm/java-1.8.0
```

It is strongly recommended to use Java 8 for running the signature tests, because the Sigtest tool doesn't work well with newer Java versions.

5.2. Preparing the environment

To run the signature tests, you will need obtain three files. To simplify the invocation of the Sigtest tool, it is recommended to place all files in the same directory and to run the Sigtest from this directory:

```
mkdir ${HOME}/sigtest/mvc/ && cd $_
```

First you will need to get the Java EE 8 API JAR. You can download this directly from Maven Central:

```
wget https://repo1.maven.org/maven2/jakarta/platform/jakarta.jakartaee-api/9.0.0/jakarta.jakartaee-api-9.0.0.jar
```

Next, you will need the implementation JAR of the Jakarta MVC 2.1 API. In this example we will simply use the official API JAR.

```
wget https://repo1.maven.org/maven2/jakarta/mvc/jakarta.mvc-api/2.1.0/jakarta.mvc-api-2.1.0.jar
```

Finally you will need the `.sigtest` definition file, which you can also get from Maven Central:

```
wget https://repo1.maven.org/maven2/jakarta/mvc/tck/mvc-tck-sigtest/2.1.0/mvc-tck-sigtest-2.1.0.sigfile
```

Now you are ready to run the signature tests.

5.3. Running the signature test

```
${JAVA_HOME}/bin/java -jar ${SIGTEST_HOME}/lib/sigtestdev.jar SignatureTest \
  -Classpath "${JAVA_HOME}/jre/lib/rt.jar:./jakarta.jakartaee-api-9.0.0.jar:./jakarta.mvc-api-2.1.0.jar" \
  -Package jakarta.mvc \
  -FileName ./mvc-tck-sigtest-2.1.0.sigfile \
  -static
```

If the test is successful, you should see an output like this:

```
SignatureTest report
Base version: 2.1.0
Tested version:
Check mode: src [throws normalized]
Constant checking: on


STATUS:Passed.
```

5.4. Forcing a signature test failure

To verify that the signature test works correctly, you can modify the signature file:

```
CLSS public abstract interface jakarta.mvc.Models
intf java.lang.Iterable<java.lang.String>
meth public abstract <%0 extends java.lang.Object> {%%0} get(java.lang.String,java.lang.Class<{%%0}>)
meth public abstract java.lang.Object get(java.lang.String)
meth public abstract java.util.Map<java.lang.String,java.lang.Object> asMap()
meth public abstract jakarta.mvc.Models put(java.lang.String,java.lang.Object)
```

You can for example delete the line containing the definition of the `get` method. Now run the

signature test again. The check should fail like this:

```
SignatureTest report
Base version: 2.1.0
Tested version:
Check mode: src [throws normalized]
Constant checking: on


Added Methods
-------------

jakarta.mvc.Models:   method public abstract java.lang.Object jakarta.mvc.Models.get(java.lang.String)


STATUS:Failed.1 errors
```