Integration testing for Jakarta EE

Rudy De Busscher

Version 1.0.0, 08/10/2022

Table of Contents

Release notes

1.0.0

1. Initial version with support for Payara Micro, Open Liberty, Wildfly
2. Glassfish support is limited.

Introduction

Integration testing can be difficult as you need to run your application as it will run in production. This means that you need to set up and configure all dependencies but provide each of them with a predefined set of values to have repeatable and predictable tests.

The use of the Testcontainers project can help you in setting up and maintaining the test environments that you need.

The Jakarta Integration Tet framework helps you to run your application on the application container within Docker using the Testcontainers project. It helps you deploy the application with the runtime of your choice and provides you easy access to the endpoints of your application. But with a simple parameter, you can also remotely debug your application to get a fast solution to the issues that might be detected during the tests.

Supported Jakarta runtimes

This integration testing framework is designed to work with Jakarta Runtimes, Jakarta EE 8 as this is still the version that is used by the bulk of the applications that are developed today but actually it is not limited to this version (see further on). The runtimes that are supported are:

- Payara Micro - website
- OpenLiberty - website
- Wildfly - website
- Glassfish - website - Limited support, see further on

Configuration

To get started, at this dependency to your project

```
    <dependency>
        <groupId>be.atbash.test</groupId>
        <artifactId>integration-testing</artifactId>
        <version>1.0.0</version>
        <scope>test</scope>
    </dependency>
```

It uses JUnit5 and also brings the Testcontainers dependency into your project test scope. It also depends on Apache CXF and Jackson but more on that when I talk about calling the endpoints within your application.

On your machine, you also need a valid Docker client (although it also works with Docker Machine) that accesses a docker environment. Look at the Testcontainers requirement for Docker if you want more information.

Creating a test

When you want to write a test that deploys your application on a certain runtime, you need to have the following test class.

```
@ContainerIntegrationTest(runtime = SupportedRuntime.PAYARA_MICRO)
public class HelloPayaraIT extends AbstractContainerIntegrationTest {
// Should be public and not the JUnit 5 preferred package scope.
}
```

The `AbstractContainerIntegrationTest` class keeps the Container reference that is running your application and also registers an extension that will help you trace problems (see further on)

The class also requires a specific annotation that triggers the JUnit 5 extension but also has some members to define, for example, the Jakarta runtime on which your application will run. For an explanation of all members, see further on

As you can see from the comment in the snippet, the test class should have a scope `public`. The JUnit 5 extension makes use of reflection and to avoid issues on newer JDK versions, the accessibility of fields is not changed (no call of `field.setAccessible(true)`).

Access application endpoints

When you want to perform integration testing on your application, you typically call the endpoints and see if you get the expected results back. Calling endpoints or URLs can be done in several ways within Java, even with the classes from the JVM itself. But most methods are rather cumbersome and sensitive to configuration values like host, ports, paths, etc. And these vary which each run using Testcontainers, so a robust way of calling your endpoints was chosen using the MicroProfile rest Client functionality.

But as a developer writing the test, you don't need to know all the details to make use of it nor does the runtime that runs your application must support MicroProfile Rest Client since it is only used on the test side.

For each *controller* or the set of endpoints you want to test, you should create an interface with the JAX-RS annotations that define the endpoint.

```
@Path("/product")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public interface JsonService {

    @GET
    List<Product> getProducts();

    @POST
    void addProduct(Product product);

}
```

With the help of the Apache CXF and Jackson frameworks, the JUnit 5 extension will generate a proxy from this interface that is capable of calling your application endpoint. You only need to *inject* it into your test class. The Junit5 extensions use the RestClient annotation as a marker for this purpose.

```
    @RestClient
    public JsonService jsonService;
```

(also here the field must be public to avoid calls to `setAccessible`)

Calling your application endpoints in the test becomes calling Java methods. And the host, port, and root part of your URL is taken care of by the JUnit 5 extension. You only need to configure the rest of the âth with the `@Path` annotation on the interface.

Base Docker images

As you have seen in the example earlier in this document, you can define the runtime that runs your application using the `runtime` member of the `@ContainerIntegrationTest` annotation.

But you also can define the runtime using a java system property. When you do not specify the runtime with the annotation, the property `be.atbash.test.runtime` is used to determine the runtime. The value should, case-insensitive, math the enum name of `SupportedRuntime`. This allows you to run your application on different runtimes if you are developing a framework or library for example.

The value of the *SupportedRuntime* determines the base Docker image from which the JUnit 5 extension creates an Image to perform the test. These are the default values of those Docker images.

- Payara Micro : `payara/micro:5.2022.2-jdk11`
- OpenLiberty : `openliberty/open-liberty:22.0.0.6-full-java11-openj9-ubi`
- WildFly : `quay.io/wildfly/wildfly:26.1.1.Final`
- Glassfish : `airhacks/glassfish:5.1.0`

You can use a different base docker image by defining a System Property.

Defining version number

When you specify the System property `be.atbash.test.runtime.version`, it is used to determine the base Docker image.

You can specify just the tagname to select the same Docker image but another version (like *5.2022.2* for the Java 8 version of the Payara Micro image). When this value contains a `:` or `/`, it will be used as the value for the *FROM* command in the Docker file. This allows you to use your own Docker image for testing your application.

Jakarta runtime specifics

The current version of the framework is designed to work with any Jakarta EE 8 compatible runtime and the application is running with Java 11. For a few runtimes, there are some specific additional requirements or limitations.

OpenLiberty

The Docker images for OpenLiberty require that you supply a *server.xml* file to configure the process. The testing framework expects this file within the *src/main/liberty/config* directory (the standard location when using the liberty tooling.) For more information on this file, look at OpenLiberty documentation page and the examples in this repository also have a minimal example.

Important here is the element `webApplication` that makes sure the application under test is deployed on the root.

Glassfish

Since there is no official Glassfish Docker image available, the framework uses the image that is created by Adam Bien, the Docklands images.

However, this image runs on Java 8 and has no support for remote debugging. So consider the support for Glassfish as very limited for the moment due to the lack of an official Docker Image for it.

Remote Debug

The testing framework supports remote debugging of your application. This makes it easier to research what is wrong with your code based on a failing test.

To activate it, set the `Debug` member of the `@ContainerIntegrationTest` annotation to true.

During the time that the framework code waits until the application is up and running, you can connect your Java debugger to port 5005. The start of the JVM is halted due to the `suspend=y` option that is passed to the JVM as part of the Debug configuration.

If you do not connect the debugger 'on time', the Testing framework reports the test as failed because the container did not start up correctly within 60 seconds of waiting time.

Other features

Some additional features available with the testing framework

Volume Mapping

It is also possible to define a volume mapping between the host running the test and the container running the application. This is the easiest way when you need to send to or retrieve files from the container. The mapping can be defined within the `@ContainerIntegrationTest`

```
@ContainerIntegrationTest(volumeMapping = {"path/on/host", "/path/within/container"})
```

You can define 1 or multiple mappings by defining sets of 2, 4, 6, ... strings.

The first one is the directory on the host. It can be a relative path and is resolved against the current directory of the current process. It might also be a absolute path and the JVM logic is used to derive the absolute path for the value you specify (using `File.getAbsolutePath`). The second string is the directory within the container and must always be absolute.

Live logging

It is possible to show the output of the runtime in the test output log. To have this info, specify it through the annotation. You should have already the logging for Testcontainers set up probably to have this working.

First, let us quickly recap the logging configuration of TestContainers. You can also read more on the Testcontainers documentation page.

Make sure you add an SLF4J logging output dependency to your project, like *Logback*.

```
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.11</version>
        <scope>test</scope>
    </dependency>
```

And have a proper configuration file for Logback appenders.

```
@ContainerIntegrationTest(liveLogging = true)
```

With the above definition, the container log will show within the output as defined in the Logback configuration at the moment the log entry is generated.

You can always access the log of the container from within your test code by using the statement.

```
   String logs = AbstractContainerIntegrationTest.testContainer
.AbstractContainerIntegrationTest.testContainer.getLogs();
```

Container log when test fails

Without any additional configuration needed, the container log will be shown in certain cases of failure of your test. This will help you to determine what went wrong and how you can fix the problem. With the following types of failures, the log is shown.

- The test code throws a `java.lang.AssertionError` error.
- The MicroProfile Rest client code encounters a status 404 when calling an endpoint.
- The MicroProfile Rest client code detects an Internal Server Error within the container.

Jakarta EE 10 support

As indicated, the current version of the test framework runs runtime versions of Jakarta EE 8 compatible products on JDK 11.

But since the only *connection* between the test and the endpoints of your application within the container is HTTP based, there is no requirement on the application that can be tested.

When you define the version/tag name of the container that is started your application can make use of Jakarta EE 9.x, Jakarta EE 10, and run on any JDK that is supported by the runtime. So it is easy to use this framework with the upcoming Jakarta EE 10 release.