



# User Guide

# Contents

<b>What is Jakarta EE?</b>	<b>1</b>
What is a Specification?	2
What is a Compatible Implementation?	2
<b>What is Eclipse MicroProfile?</b>	<b>3</b>
<b>Jakarta EE Application Development Process</b>	<b>3</b>
Development	4
Testing	4
Deployment	4
<b>Jakarta EE Testing</b>	<b>5</b>
Principles of Testing	5
Testing Shows the Presence of Defects, not their Absence	5
Exhaustive Testing is Impossible	5
Early Testing Saves Time and Money	5
Defects Cluster	5
Beware of the Pesticide Principle	6
Testing is Contextual	6
Absence-of-errors is a Fallacy	6
Types of Tests	6
Unit Tests	6
Integration Tests	6
Functional Tests	7
Performance Tests	7
Smoke Tests	7
Custom Test Combinations	7
Testing Libraries	8
JUnit	8
Mockito	9
Arquillian	12
Testcontainers	22
Other Test Libraries	25
<b>Summary</b>	<b>26</b>



The Java Platform has been the first choice for enterprise application development for many developers over the last two and half decades. There is no shortage of frameworks and platforms for developing all kinds of software applications using the Java Programming Language. One such platform that has stood the test of time is Jakarta EE (formerly Java EE).

This guide describes software testing on the Jakarta EE Platform. It provides a brief look at the theoretical foundations of Jakarta EE, followed by an overview of testing principles and finally exploring the most popular choices for creating automated tests of Jakarta EE developed applications.

## What is Jakarta EE?

[Jakarta EE](#) is a set of community developed, abstract specifications that together form a platform for developing end-to-end, multi-tier enterprise applications. Jakarta EE is built on the Java Standard Edition, and aims to provide a stable, reliable and vendor neutral platform on which to develop cloud native applications.

Hitherto, Jakarta EE was called Java EE and was a property of Oracle Inc., evolved through the Java Community Process (JCP). However, in late 2017, Oracle decided to move the platform to an open foundation for a much broader community-led evolution. The Eclipse Foundation got chosen and Java EE, after the transfer, got rebranded to Jakarta EE.

## What is a Specification?

As stated in the above definition, Jakarta EE is made up of a set of specifications that each cover a specific API for solving a specific software development need. For example, the Jakarta Contexts and Dependency Injection ([Jakarta CDI](#)) specification provides constructs for creating loosely coupled applications through dependency injection. These different specifications are combined into a single “umbrella” specification for each Jakarta EE release. As such, Jakarta EE 10 for instance, is released under the Jakarta EE 10 specification.

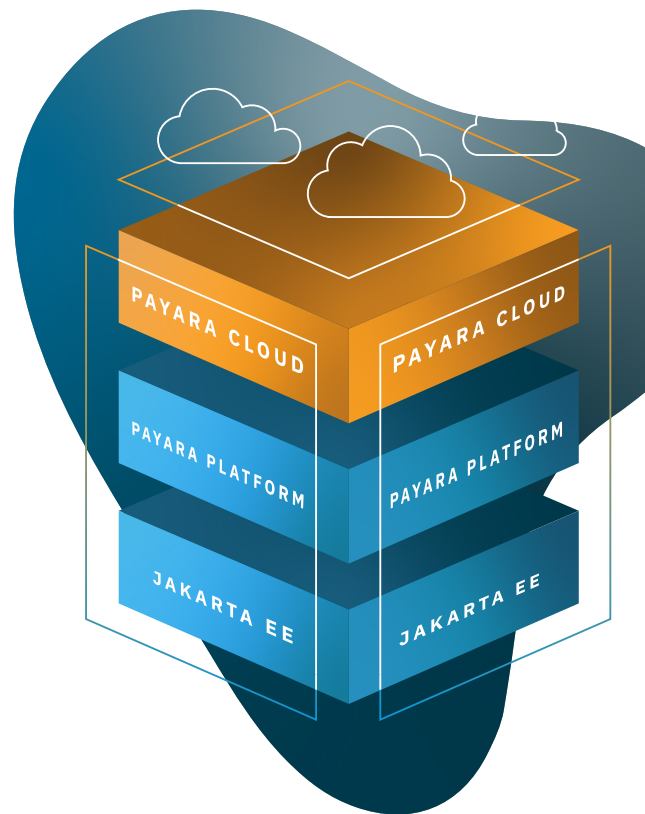
More technically, a specification is a formal proposal document made to the Jakarta EE Specification Committee through the Jakarta EE Specification Process (JESP) that outlines the functions of a given set of APIs. This document outlines what the expected behaviour should be for various invocations of the API. The specification then acts as the blueprint for the API.

## What is a Compatible Implementation?

As a specification is merely a document that outlines the behaviour of a given API, it needs an implementation that realises the actual outcomes for each invocation of the API. For instance, the Jakarta Persistence specification provides the EntityManager interface that has the persist() method. This method, when called and passed an instance of a Jakarta Persistence entity, persists that entity instance as a database row to the underlying database. The “library” that does the actual work of taking that instance and making sure it gets stored to the durable storage when the EntityManager#persist() method is invoked, is called a compatible implementation of the Jakarta Persistence specification.

Each specification that makes up the full Jakarta EE platform has an implementation. As a specification itself, the Jakarta EE platform also has an implementation in the form of [compatible products](#) and are free to pick any compatible implementation of the platform. With this abstraction, Jakarta EE implementation vendors can collaborate on the base, standard specifications and compete through innovations on top of the base platform.

An example of such invocation is the [Payara Cloud](#) offering from Payara. This innovation helps you realise the dream of true separation of your business domain application and the runtime that powers it. With Payara Cloud, you simply upload your Jakarta EE application web archive (.war file) and have it automatically deployed to the cloud, just as Jakarta EE was envisaged to have a separation of business domain from the runtime. Another example of custom features available on the Payara Platform is [remote CDI events](#). This feature, built on the Jakarta CDI specification, allows the firing of CDI events that can be observed by any listener in a given Hazelcast cluster.



## What is Eclipse MicroProfile?

The Jakarta EE Platform is a general purpose platform for developing all kinds of applications. As modern application development paradigms have changed a lot in the past years, there is a need to evolve the platform to meet such changes. One such paradigm is cloud native software application development.

As the base Jakarta EE Platform has always been geared towards enterprises, it has historically evolved at a much slower pace than changes in the software development space. It is for this reason that the [Eclipse MicroProfile](#) project was created as an extension to the base platform to provide cloud-native APIs for developing modern cloud-based applications.

Eclipse MicroProfile, built upon Jakarta CDI, Jakarta REST and Jakarta JSON Processing, comes with the following APIs

- OpenTracing
- OpenAPI
- REST Client
- Config
- Fault Tolerance
- Metrics
- JWT Propagation
- Health

These APIs augment the much larger Jakarta EE Platform APIs to provide the developer with a cohesive set of APIs for developing, testing and deploying cloud-native modern enterprise applications.

## Jakarta EE Application Development Process

Jakarta EE application development follows the general software development lifecycle process of requirements gathering and planning, design and/or prototyping, actual development, testing, deployment and maintenance. Different companies use different combinations of these steps. However, development, testing and deployment are steps that almost every company's application development process will entail.

*All code examples are available on the Payara GitHub: <https://github.com/payara/Payara-Examples>*

## Development

Developing applications on the Jakarta EE Platform with Eclipse MicroProfile requires that the application declares a dependency on those two APIs. As Maven is the de facto build tool for most Jakarta EE applications, a typical dependency declaration would look like that shown in Figure 1-1.



```
<dependencies>
  <dependency>
    <groupId>jakarta.platform</groupId>
    <artifactId>jakarta.jakartaee-api</artifactId>
    <version>10.0.0</version>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>org.eclipse.microprofile</groupId>
    <artifactId>microprofile</artifactId>
    <version>5.0</version>
    <type>pom</type>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

**Figure 1-1** showing Jakarta EE and MicroProfile dependency declaration.

With the above declaration in place, a compatible implementation like the [Payara Platform](#) can be used for development. Scoping the dependencies to ‘provided’ means the runtime, or compatible implementation, will provide the implementation of the various ‘abstract’ APIs used.

## Testing

Every non-trivial enterprise application will require testing. There are different types and combinations of testing that can be employed to assure a certain level of quality for software applications. The next section of this guide goes into detail about testing on the Jakarta EE Platform.

## Deployment

After testing, an application will need to be deployed. As shown in Figure 1-1, a Jakarta EE application normally does not ship with the runtime. The runtime is provided by the compatible implementation on which the application is deployed. There are different ways a Jakarta EE application can be deployed, each varying in their complexity and ease of use. Payara Cloud is one such deployment platform for Jakarta EE and Eclipse MicroProfile applications that takes an application archive and automatically provisions all the infrastructure needed to run the application in the cloud.

# Jakarta EE Testing

Testing is an integral, important part of software development. Testing assures that an application meets a certain minimum level of quality. Jakarta EE applications, like applications developed with other frameworks, require testing. As Jakarta EE is built on top of the Java Standard Edition, all the testing libraries and frameworks available for Java SE can be employed to craft different types and combinations of automated tests. Jakarta EE itself does not have any special testing API.

## Principles of Testing

As important as software application testing is, there is no universal criteria for determining what comprises good testing practices. The plethora of programming languages and application development frameworks out there mean testing is a highly heterogeneous activity that differs from one platform to the other and from one application to the other. The American Software Testing Qualifications Board (ASTQB) has identified seven principles that can help guide the creation of an effective testing regime. These principles are:

### Testing Shows the Presence of Defects, not their Absence

Tests can help reveal bugs in an application. However, tests cannot guarantee that an application is bug-free. All the tests in a test suite passing does not mean there are no defects that can be unearthed. This notwithstanding, having extensive tests can give reasonable assurance that the application will not fail under conditions that have been tested.

### Exhaustive Testing is Impossible

Enterprise applications are very large and complicated. It is not possible to have tests that cover every possible permutation of an application. Attempting any such activity will prove expensive. An analysis of different core features should be made and test efforts focused on critical areas.

### Early Testing Saves Time and Money

A quick smoke test of a new application release can help identify showstopper bugs that could have required the running of the entire test suite to unearth. This also applies to implementing test driven development as much as possible.

### Defects Cluster

A small number of components in an application will be responsible for the majority of bugs. Testing should then focus on these components that are responsible for the majority of the bugs. This is

essentially applying the Pareto Principle to testing application components – the idea that roughly 80% of consequences come from 20% of causes.

## **Beware of the Pesticide Principle**

The same set of tests repeated over a long period of time may end up not catching new defects. This principle is an analogy referring to the ineffectiveness of pesticide applied to the same area over a long period in killing insects. Tests will need to be updated, refactored with new input data and permutations to keep them effective at identifying application defects.

## **Testing is Contextual**

Testing is a highly context dependent activity. The tests applied to a business application will differ from the tests applied to an aerospace computer control application. This is important to keep in mind when taking inspiration for creating testing systems.

## **Absence-of-errors is a Fallacy**

As elucidated in the first principle, tests do not guarantee the absence of errors. As such, it is a mistaken notion to assume an application is free of defects if tests do not unearth any. Also identifying and fixing defects does not guarantee the system cannot fail.

## **Types of Tests**

There are many types of tests that can be employed in any given domain. Each company will employ a different combination of these tests depending on the overall objectives of testing and the type of application. The most popular types are listed below.

### **Unit Tests**

A unit test is an automated test that tests a single application component in isolation to verify if the component is working as expected. Unit tests are the most granular tests in any test suite. Depending on the particular component being unit tested, its dependencies might be mocked in order to be able to unit-test just that component in isolation. Unit tests are generally very fast to run because of their isolated nature.

### **Integration Tests**

An integration test is a test for asserting that different components of the application that together carry out a specific function work correctly. Integration tests span more than one component. Integration tests tend to be much more time consuming to run because they can span several components and most likely include several network calls. Integration tests are also automated.

## Functional Tests

A functional test focuses on the business aspect of an application. Functional tests will normally verify the results of the operation under test. There is a fine line between functional and integration tests because both tests will cause different components of the application to be invoked. In some organisations, functional and integration tests are one and the same. However, technically, functional tests assert the output of an operation to verify that it equals a preset, expected value. Integration tests, however, generally check if the components are working. Functional tests can be manual or automated.

## Performance Tests

A performance test checks how the application behaves when subjected to different workloads. Performance tests can help unearth bottlenecks because they measure things like the speed of the application, the reliability of a site, how scalable a component is among others. This kind of test can help optimise applications based on the data gathered from the tests. Performance tests are automated.

## Smoke Tests

A smoke test is mostly a manual test that checks if the basic functionality of an application is working as expected. Smoke tests generally follow a new build or release of the application and precede other types of checks.

## Custom Test Combinations

There are other tests, like end-to-end and acceptance testing that are different variations of integration and functional tests. In the end, every organisation will have a mix of different test types and modes based on the domain. The test types enumerated above are in no way an exhaustive list or all the test types out there. But these are the most common that cut across all types of organisations.

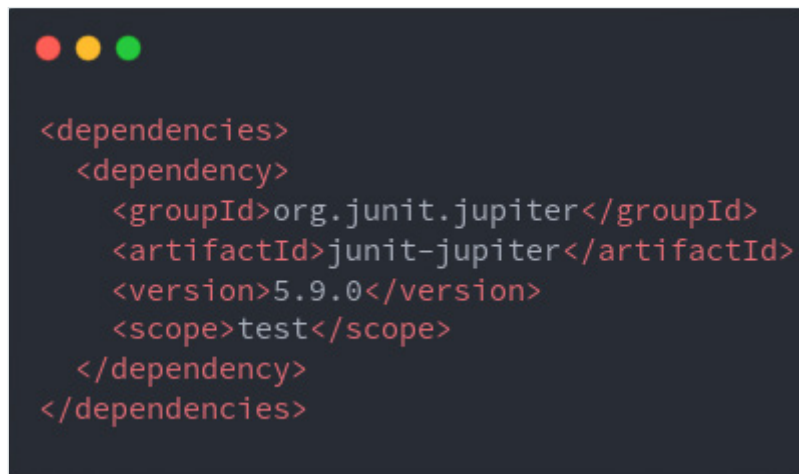
## Testing Libraries

As a Java based platform, there is a healthy collection of testing libraries that you can choose from to craft tests for a typical Jakarta EE application. You will use different combinations of these libraries to help test your application. This section takes a look at the most popular ones, and shows how you can use them in a Jakarta EE application. We will also look at tradeoffs in picking a library.

### JUnit

JUnit is a Java testing library for making assertions on test artefacts. It has a number of callbacks for setting up and tearing down test data. It also has an extension API that other testing libraries have built on top of to extend testing for different purposes. JUnit is arguably the most popular testing library in enterprise Java.

To add JUnit to a Jakarta EE project, a dependency on the junit-jupiter will be declared in the project object model file as shown in Figure 1-2



```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.9.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Figure 1-2 showing a JUnit dependency declaration in the pom.xml file

With this dependency in place, we can use the gamut of JUnit assertions and test constructs. However, you will seldom use JUnit in isolation. It will serve as the base test library for setting up test data, creating test classes and artefacts and then using its various `assert*` methods to make assertions.

## Mockito

Mockito is a Java library for creating mocked objects in unit tests. A component under test that has dependencies on other components will have those dependencies mocked. Mocking allows predictable values on the mocked objects to be returned and later asserted.

A typical Jakarta EE application will consist of runtime managed components. By runtime managed components, we mean application controller classes that are managed by the Jakarta Contexts and Dependency Injection runtime. As the “glue” that helps you develop loosely coupled applications, the CDI runtime manages the creation, injection and destruction of various application dependencies.

As such, you cannot simply unit test different components in isolation without some form of mocking of their dependencies. For such, the Mockito library is an excellent one. Using it entails adding it as a dependency as showing in Figure 1-3.



```
<dependencies>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.8.0</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>4.8.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

**Figure 1-3 shows Mockito dependency declaration**

Figure 1-3 shows a dependency declaration on the mockito-core and mockito-junit-jupiter artefacts. Mockito has JUnit extensions that for creating unit tests we will need. Figure 1-4 shows the component we seek to test.

```
1 @ApplicationScoped
2 public class GreetingService {
3
4     @Inject
5     private ConfigPropertyProvider propertyProvider;
6
7     public JsonObject greet(final String name) {
8         DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
9         return Json.createObjectBuilder()
10             .add("greeting", "Hello there " + name)
11             .add("message", "Getting started with Jakarta EE!")
12             .add("platform", "Jakarta EE")
13             .add("platformVersion", propertyProvider.getJakartaVersion())
14             .add("implementation", propertyProvider.getApplicationServer())
15             .add("date", LocalDateTime.now(ZoneOffset.UTC).format(formatter))
16             .build();
17     }
18 }
19
```

**Figure 1-4 showing GreetingService controller**

GreetingService is an application-scoped (Line 1) singleton that has a single method - greet(String name) that returns a jakarta.json.JsonObject. The controller declares a dependency (Line 5) on ConfigPropertyProvider, shown in Figure 1-5 below.

```
1 @ApplicationScoped
2 public class ConfigPropertyProvider {
3
4     @Inject
5     @ConfigProperty(name = "jakarta.version")
6     @Getter
7     private String jakartaVersion;
8
9     @Inject
10    @ConfigProperty(name = "application.server")
11    @Getter
12    private String applicationServer;
13 }
```

**Figure 1-5 showing ConfigPropertyProvider controller class.**

The `ConfigPropertyProvider` is also an application-scoped singleton that centralises `MicroProfile Config` properties. It has getter methods (Lines 6 and 11 from Lombok) for making these injected configuration properties available to its clients.

To unit-test the `GreetingService`, we need to first mock calls to its dependent, the `ConfigPropertyProvider`. This way, we can unit test it in isolation. Figure 1-6 shows the full JUnit/Mockito unit test class for the `GreetingService`.

```
1 @Log
2 @ExtendWith(MockitoExtension.class)
3 public class GreetingServiceTest {
4
5     @InjectMocks
6     protected GreetingService greetingService;
7
8     @Mock
9     protected ConfigPropertyProvider propertyProvider;
10
11
12
13     @BeforeEach
14     void init() {
15         when(propertyProvider.getApplicationServer()).thenReturn("payara-6-community");
16         when(propertyProvider.getJakartaVersion()).thenReturn("10.0.0");
17     }
18
19     @Test
20     void testGreetingService() {
21
22         var jsonObject = greetingService.greet("John");
23         log.log(Level.INFO, jsonObject.toString());
24
25
26         assertEquals("Hello there John", jsonObject.getString("greeting"));
27         assertEquals("Getting started with Jakarta EE!", jsonObject.getString("message"));
28         assertEquals("Jakarta EE", jsonObject.getString("platform"));
29         assertEquals("10.0.0", jsonObject.getString("platformVersion"));
30         assertEquals("payara-6-community", jsonObject.getString("implementation"));
31         assertNotNull(jsonObject.get("date"));
32     }
33 }
34 }
```

**Figure 1-6 showing the `GreetingServiceTest`**

The `GreetingServiceTest` is a JUnit test that is extended with Mockito (Line 2) through the `@ExtendWith` annotation. The class under test (CUT) is injected on Line 6 as a Mockito managed object through `@InjectMocks` (Line 5) annotation. Lines 8-9 show an injection of a mocked instance of the `ConfigPropertyProvider` controller class. The mocked instance will be injected into the mockito managed `GreetingService` object.

With these injections in place, we tell the Mockito runtime what to return if a given method is invoked on the mock. Lines 13-17 use the JUnit `@BeforeEach` callback method to set up response data when certain methods are invoked on the mocked `ConfigPropertyProvider` controller instance. In this case, we return the values that would have been returned if the controller class was running in a managed environment like the Payara Platform or Payara Cloud. In that case, the MicroProfile Config runtime will inject config values into the various fields in the `ConfigPropertyProvider` (Figure 1-5). But in a unit test environment, we need to mock such a function because everything is running in isolation, independent of any runtime provided features.

Lines 19-33 finally show the actual test. Line 22 shows the invocation of the `greet` method on the `GreetingService` Mockito managed instance. Lines 26-31 makes assertions on the returned `JsonObject`, using predetermined values that we expect to be available in the returned `JsonObject` if the `GreetingService` is working correctly.

The entire `GreetingServiceTest` class is a simple unit test. As shown in Figure 1-6, there is nothing related to Jakarta EE in it. It is a simple, plain unit test that is testing a single component in isolation. This kind of test is also very fast because of its very isolated, single-unit nature. This is an example of how you can unit test Jakarta EE components using JUnit in combination with the Mockito library. Core components of your application should be heavily unit tested because by their nature, unit tests are cheap and thus lend themselves to faster build pipelines.

## Arquillian

[Arquillian](#) is an integration testing library for creating application deployments in a test context. Unlike the unit test discussed in the previous section, integration testing requires a deployed instance of the application because, as the name implies, it tests how a given component behaves within the context of its relationship with other components of the application. Figure 1-7 shows the `HelloResource` class.



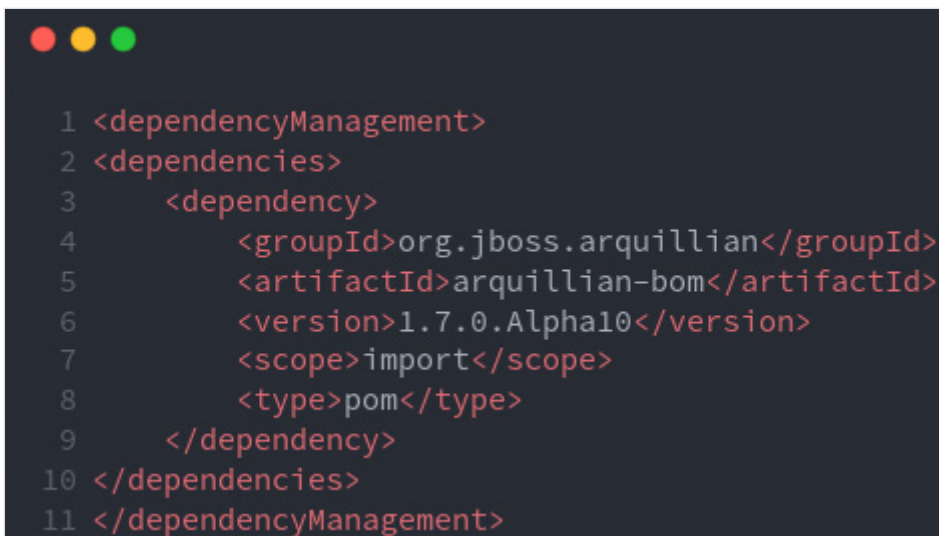
Figure 1-7 showing the `HelloResource`

The `HelloResource` is a Jakarta REST service that depends upon the `GreetingService` (Lines 4-5). It has one resource method hosted at the path `/hello-world/{visitor}` where `visitor` is a path parameter variable that is substituted at runtime with the value that is passed. The resource method delegates the greeting to the `GreetingService` instance. Though this class is a very simple, boring one, there are three different Jakarta APIs at play. The Jakarta CDI, Jakarta REST and MicroProfile Config APIs. CDI providing the wiring of the dependencies, Jakarta REST providing the infrastructure to accept HTTP requests, route to this class, marshal and return a response to the client, MicroProfile Config providing the reading of the configuration values provided by the `ConfigPropertyProvider`.

To test if all these disparate parts that make up the `HelloResource` work correctly when the application is deployed to a compatible runtime, we need to have such a deployment environment in the test context. Effectively, testing that the `JsonObject` returned when the `HelloResource` method is invoked, will, in actual fact, indirectly be asserting that all the dependencies of the class are behaving correctly, at least as far as the returned data is correct.

Arquillian provides the constructs for creating application deployments in a test context such that we can make different test assertions on different components of the application without needing to mock. Essentially with Arquillian, we get a “live view” of the application in the test context and can then test if the different parts are working well, both individually and as a cohesive whole.

To use Arquillian, we need to declare a dependency on it in the `pom.xml` file. The first part of this is to declare the `arquillian-bom` in a dependency-management block, as shown in Figure 1-8.



```
1 <dependencyManagement>
2 <dependencies>
3   <dependency>
4     <groupId>org.jboss.arquillian</groupId>
5     <artifactId>arquillian-bom</artifactId>
6     <version>1.7.0.Alpha10</version>
7     <scope>import</scope>
8     <type>pom</type>
9   </dependency>
10 </dependencies>
11 </dependencyManagement>
```

Figure 1-8 showing the `arquillian-bom` declaration.

With the bill of materials declared, we can then declare the rest of the dependencies in the dependencies block as shown in Figure 1-9.

```
1 <dependencies>
2   <dependency>
3     <groupId>org.jboss.arquillian.junit5</groupId>
4     <artifactId>arquillian-junit5-container</artifactId>
5     <scope>test</scope>
6
7   </dependency>
8
9   <dependency>
10    <groupId>fish.payara.arquillian</groupId>
11    <artifactId>arquillian-payara-server-embedded</artifactId>
12    <version>3.0.alpha7</version>
13    <scope>test</scope>
14  </dependency>
15
16  <dependency>
17    <groupId>fish.payara.extras</groupId>
18    <artifactId>payara-embedded-all</artifactId>
19    <version>6.2022.1</version>
20    <scope>test</scope>
21  </dependency>
22 </dependencies>
```

**Figure 1-9 showing the other Arquillian dependencies.**

Similar to Mockito, Arquillian has a JUnit extension that we pull in with the `arquillian-junit5-container`. The `arquillian-payara-server-embedded` tells Arquillian which container we want to use to run the test. As discussed above, running an integration test entails deploying the actual application artefact, in our case a `.war` file, to a real application server, or compatible Jakarta EE runtime.

There are different ways an implementation can be used with Arquillian. The `arquillian-payara-server-embedded` is a [connector](#) from Payara that tells the library how to start, deploy and stop the container. The `payara-embedded-all` dependency then pulls in the embedded server. The last bit of setup to get everything in place is to configure the `arquillian.xml` file, shown in Figure 1-10 below.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3             xmlns="http://jboss.org/schema/arquillian"
4             xsi:schemaLocation="http://jboss.org/schema/arquillian
5                                 https://jboss.org/schema/arquillian/arquillian_1_0.xsd">
6     <defaultProtocol type="Servlet 5.0"/>
7
8     <container qualifier="payara-embedded" default="true"/>
9
10 </arquillian>
```

**Figure 1-10 showing the arquillian.xml configuration file.**

The arquillian.xml file is where the library is configured. The sample shown in Figure 1-10 sets the payara-embedded as the default container to be used in the test. This file can be used to configure different aspects of the library and containers that are available to it. For our purposes, the above sample suffices. With everything in place, let's take a look at the HelloResource integration test as shown in Figure 1-11.

```
1 @Log
2 @ExtendWith(ArquillianExtension.class)
3 public class HelloResourceIT {
4
5     @Inject
6     protected HelloResource helloResource;
7
8     @ArquillianResource
9     protected URL contextPath;
10
11     protected static HttpClient httpClient;
12
13
14     static Jsonb jsonb;
15     @BeforeAll
16     public static void initAll() {
17         httpClient = HttpClient.newBuilder().build();
18         jsonb = JsonbBuilder.create();
19     }
20
21
22     @Deployment
23     public static WebArchive createLocalDeployment() throws URISyntaxException {
24         final JavaArchive javaArchive = ShrinkWrap.create(JavaArchive.class, "jumpstart-ee.jar")
25             .addPackage(HelloApplication.class.getPackage())
26             .addAsResource("test-persistence.xml", "META-INF/persistence.xml")
27             .addAsManifestResource(new FileAsset(new File("src/test/resources/META-INF/beans.xml")),
28                 "beans.xml")
29             .addAsResource(
30                 new UrlAsset(Objects.requireNonNull(
31                     HelloResourceIT.class.getResource("/microprofile-config.properties")),
32                     "/META-INF/microprofile-config.properties");
33
34         javaArchive.getContent().forEach((key, value) -> log.log(Level.INFO, (key + "-->" + value)));
35         return ShrinkWrap.create(WebArchive.class, "jakarta-jumpstart.war").addAsLibrary(javaArchive);
36     }
37
38     @Test
39     void testHello() {
40         var jsonObject = helloResource.hello("John Jakes");
41         assertNotNull(jsonObject);
42
43         log.log(Level.INFO, jsonObject.toString());
44
45         assertEquals("Hello there John Jakes", jsonObject.getString("greeting"));
46         assertEquals("Getting started with Jakarta EE!", jsonObject.getString("message"));
47         assertEquals("Jakarta EE", jsonObject.getString("platform"));
48         assertEquals("10.0.0", jsonObject.getString("platformVersion"));
49         assertEquals("payara-6-community", jsonObject.getString("implementation"));
50         assertNotNull(jsonObject.get("date"));
51     }
52 }
53 }
54 }
```

**Figure 1-11** showing the `HelloResourceIT` class.

Figure 1-11 shows the full `HelloResourceIT` class, which starts by declaring `ArquillianExtension` as a JUnit extension we want to use. Lines 5-6 use the inject the `HelloResource`, which is our class under test, using the `jakarta.inject.Inject`. Lines 22-35 declare a method annotated `@Deployment`. This annotation marks the `createLocalDeployment` method as a deployment producer. An Arquillian deployment is an archive of everything that should be deployed as part of the application. The

method uses the library's ShrinkWrap API to create a `org.jboss.shrinkwrap.api.spec.WebArchive`. This archive contains all the classes, descriptors, properties files and everything else we need to ship with our deployed application. Arquillian takes this returned `WebArchive` and then deploys it to the configured container, in our case Payara embedded.

With the deployment method in place, Lines 37-52 create the actual test method and assertions. Line 39 calls the `hello` method on the injected `HelloResource`, passing in "John Jakes" as the parameter. The rest of the method makes assertions on the returned `JsonObject` with predetermined values that should be returned if the various controllers are working in unison.

Lines 48 and 49 are particularly interesting because for the unit test, we mocked the expected values that should be returned when the `ConfigPropertyProvider` is invoked. But for this integration test, the actual MicroProfile Config values are read by the runtime. The `microprofile-config.properties` file from which the config values to be injected into the `ConfigPropertyProvided` are read is shown in Figure 1-12.



```
1 jakarta-version=10.0.0
2 application-server=payara-6-community
```

**Figure 1-12 showing the MP config properties file.**


Figure 1-11 shows a full fledged Jakarta EE integration test using Arquillian, JUnit and the Payara Platform. The integration test in the example makes the call directly on the instance of the `HelloResource` injected into the test. However, we can also make REST calls to the `HelloResource` in the test and then make test assertions on the returned response. Figure 1-13 shows a test that uses the HTTP client introduced in Java 11 to make a REST call to the `HelloResource`.

```
1 @Log
2 @ExtendWith(ArquillianExtension.class)
3 public class HelloResourceIT {
4
5     @Inject
6     protected HelloResource helloResource;
7     @ArquillianResource
8     protected URL contextPath;
9     protected static HttpClient httpClient;
10
11     @Inject
12     protected PersistenceService persistenceService;
13
14     static Jsonb jsonb;
15
16     @Test
17     @RunAsClient
18     void testHelloResource() throws Exception {
19         var request = HttpRequest.newBuilder()
20             .uri(URI.create(contextPath.toExternalForm() + "api/hello-world/John"))
21
22             .header("Content-Type", "application/json")
23             .GET().build();
24         var response = httpClient.send(request, HttpResponse.BodyHandlers.ofString());
25         assertNotNull(response);
26         assertNotNull(response.body());
27
28         log.log(Level.INFO, response::body);
29
30         JsonObject jsonObject = jsonb.fromJson(response.body(), JsonObject.class);
31
32         assertNotNull(jsonObject);
33         assertEquals("Hello there John", jsonObject.getString("greeting"));
34         assertEquals("Getting started with Jakarta EE!", jsonObject.getString("message"));
35         assertEquals("Jakarta EE", jsonObject.getString("platform"));
36         assertEquals("10.0.0", jsonObject.getString("platformVersion"));
37         assertEquals("payara-6-community", jsonObject.getString("implementation"));
38         assertNotNull(jsonObject.get("date"));
39
40         log.log(Level.INFO, jsonObject::toString);
41     }
42 }
43
44 }
```

**Figure 1-13 showing another test method. Already discussed parts omitted for brevity.**

Lines 78 of Figure 1-13 shows the use of `@ArquillianResource` annotation to inject a resource into the field. Line 17 annotates the `testHelloResource` method with `@RunAsClient`. This annotation tells Arquillian to run this method outside of the container. We do this to simulate a real REST client firing against the `HelloResource` method. The rest of the method makes an HTTP call to the deployed `HelloResource`, converts the response from `String` to `JsonObject` and makes assertions on it.

Arquillian is not limited to just this. We can also test persistence. Figure 1-14 shows a simple HelloEntity, a Jakarta Persistence entity that will be mapped to a database table.



```
1 @Entity
2 @Getter
3 @Setter
4 public class HelloEntity implements Serializable {
5
6     @Id
7     @GeneratedValue(strategy = GenerationType.AUTO)
8     private Long id;
9
10    @Version
11    private Long version;
12
13    private String name;
14    private String greeting;
15    private LocalDateTime greetingDate;
16    private LocalDateTime dateCreated;
17
18    @PrePersist
19    private void init() {
20        dateCreated = LocalDateTime.now(ZoneOffset.UTC);
21    }
22 }
```

**Figure 1-14 showing the HelloEntity.**

The HelloEntity is persisted through the EntityManager injected into the PersistenceService shown in Figure 1-15.

```
1 @Singleton
2 public class PersistenceService {
3
4     @PersistenceContext
5     private EntityManager entityManager;
6
7     public HelloEntity save(final HelloEntity helloEntity) {
8         entityManager.persist(helloEntity);
9         return helloEntity;
10    }
11
12    public HelloEntity find(final Long id) {
13        return entityManager.find(HelloEntity.class, id);
14    }
15 }
16
```

**Figure 1-15** showing the PersistenceService.

The PersistenceService is a simple singleton EJB that persists and finds HelloEntity instances. With this in place, we can have another test to test that this controller is working as expected. Figure 1-16 shows another test that directly calls the methods on the PersistenceService and makes test assertions on the returned objects.

```
1 @Log
2 @ExtendWith(ArquillianExtension.class)
3 public class HelloResourceIT {
4
5     @Inject
6     protected HelloResource helloResource;
7     @ArquillianResource
8     protected URL contextPath;
9     protected static HttpClient httpClient;
10
11     @Inject
12     protected PersistenceService persistenceService;
13
14     static Jsonb jsonb;
15
16     @Test
17     void testPersistence() {
18         final var helloEntity = new HelloEntity();
19         helloEntity.setGreeting("Hello, world!");
20         helloEntity.setName("Arquillian");
21         helloEntity.setGreetingDate(LocalDateTime.now(ZoneOffset.UTC));
22         assertNull(helloEntity.getId());
23         assertNull(helloEntity.getDateCreated());
24         assertNull(helloEntity.getVersion());
25
26         log.log(Level.INFO, () -> jsonb.toJson(helloEntity));
27
28         final var savedEntity = persistenceService.save(helloEntity);
29
30         assertNotNull(savedEntity);
31         assertNotNull(savedEntity.getId());
32         assertNotNull(savedEntity.getDateCreated());
33         assertNotNull(savedEntity.getVersion());
34
35         assertEquals(helloEntity.getGreeting(), savedEntity.getGreeting());
36         assertEquals(helloEntity.getName(), savedEntity.getName());
37         assertEquals(helloEntity.getGreetingDate(), savedEntity.getGreetingDate());
38
39         log.log(Level.INFO, () -> jsonb.toJson(savedEntity));
40
41         final var loadedEntity = persistenceService.find(savedEntity.getId());
42
43         assertNotNull(loadedEntity);
44         assertNotNull(loadedEntity.getId());
45         assertNotNull(loadedEntity.getDateCreated());
46         assertNotNull(loadedEntity.getVersion());
47
48         assertEquals(helloEntity.getGreeting(), loadedEntity.getGreeting());
49         assertEquals(helloEntity.getName(), loadedEntity.getName());
50         assertEquals(helloEntity.getGreetingDate(), loadedEntity.getGreetingDate());
51
52         log.log(Level.INFO, () -> jsonb.toJson(savedEntity));
53     }
54 }
55
56 }
```

Figure 1-16 shows the PersistenceService test.

Lines 11-12 of Figure 1-16 injects the `PersistenceService` with the `@Inject` annotation. The `test-Persistence` method, declared on Lines 16-54 then calls the `save` and `find` methods on the injected controller and makes test assertions on the returned objects.

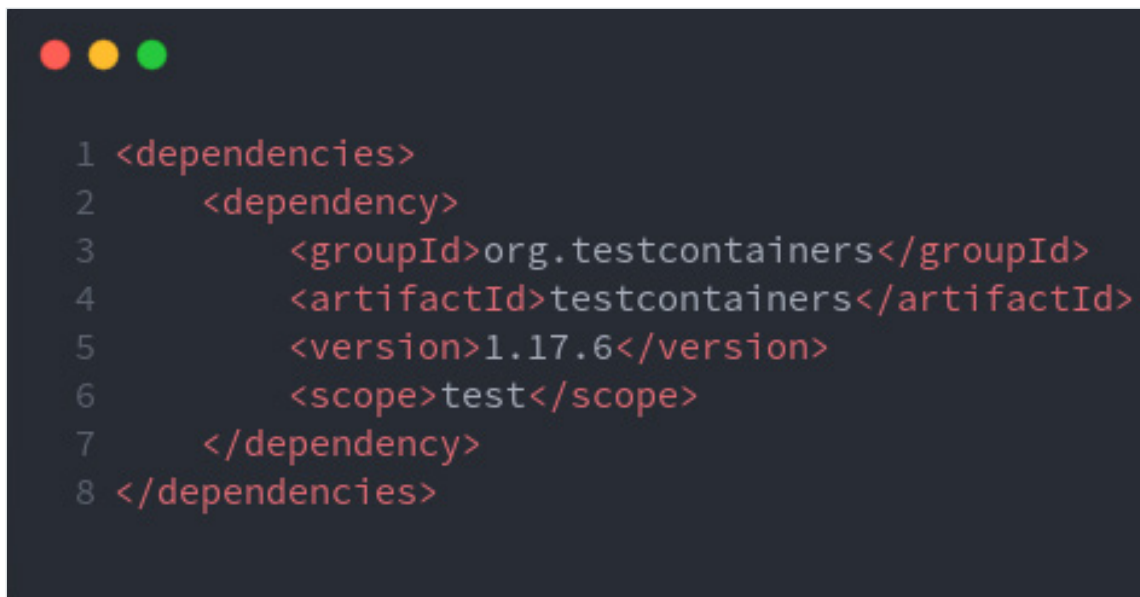
As you have seen so far, Arquillian gives you a way to test Jakarta EE managed components in your tests with minimum setup. All that is needed is to create an archive of your application with all that is needed and then leave the deployment to the library. You can also see how the JUnit library is a constant all through.

Arquillian tests, however, can be expensive to run since Arquillian needs to spin up a container on which to deploy your application archive. This cost can be reduced through the use of remote containers. This is when you tell the library to bind to an already running container. For instance a CI/CD pipeline can have a running Payara Server to which Arquillian based integration tests can be pointed to for test artefact deployment.

Whichever container option that is chosen is not as important as having adequate integration tests that touch all core aspects of an application. Even though testing does not guarantee the absence of defects, integration tests generally assure that critical application components that fail can be found out much faster for remedial action.

## Testcontainers

[Testcontainers](#) is another Java library that allows you to run test infrastructure in “lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container.” Effectively TestContainers allows you to deploy your application and its dependent services within a test as Docker containers, fully managed by the library. To use it, declare a dependency on it as shown in Figure 1-17.



```
1 <dependencies>
2     <dependency>
3         <groupId>org.testcontainers</groupId>
4         <artifactId>testcontainers</artifactId>
5         <version>1.17.6</version>
6         <scope>test</scope>
7     </dependency>
8 </dependencies>
```

Figure 1-17 showing TestContainers dependency declaration.

With the dependency declaration in place, we are ready to use the library. TestContainers has different ways of spawning Docker containers. You can spawn containers programmatically using the API, through an existing Dockerfile or through a docker-compose file. I find the docker-compose feature the most powerful of them all. This allows you to run the same set of containers that will be run in production in the test context without needing to glue together anything. Listing 1-18 shows the use of the TestContainers API to create the infrastructure for testing the HelloResource.

```
1 @Log
2 @Testcontainers
3 public class HelloResIT {
4
5     protected static HttpClient httpClient;
6     protected static String restUrl;
7     final static String serviceName = "app_1";
8     final static int targetPort = 8080;
9     static Jsonb jsonb;
10
11     @Container
12     public static DockerComposeContainer composeContainer = new DockerComposeContainer(
13         new File("docker-compose.yml"))
14         .withExposedService(serviceName, targetPort,
15             Wait.forListeningPort().withStartupTimeout(Duration.ofSeconds(30)))
16         .withExposedService(serviceName, 4848,
17             Wait.forListeningPort().withStartupTimeout(Duration.ofSeconds(30)));
18
19     @BeforeAll
20     public static void initAll() {
21         httpClient = HttpClient.newBuilder().build();
22         var serviceName = "app_1";
23         var targetPort = 8080;
24         var resourcePathTemplate = "http://%s:%d/jee-jumpstart/api";
25
26         composeContainer.start();
27
28         restUrl = String.format(resourcePathTemplate,
29             composeContainer.getServiceHost(serviceName, targetPort),
30             composeContainer.getServicePort(serviceName, targetPort));
31
32         jsonb = JsonbBuilder.create();
33     }
34 }
35
36 }
```

**Figure 1-18** showing the setup for TestContainers.

Figure 1-18 starts by annotating the test class with `@TestContainers`. This annotation is a JUnit extension that activates automatic start and stop of containers used in the test. Lines 11-17 create a `org.testcontainers.containers.DockerComposeContainer` annotated `@Container`, passing in the `docker-compose.yml` file bundled with the application. Lines 28-30 then use methods on the created `DockerComposeContainer` to get the host and port of the service within the `docker-compose.yml` file that we are interested in. The returned host and port information is then used to create a URL to the deployed resource method in the container. With the setup in place, we use the Java HTTP client to make a call to the resource and then make assertions on the returned data. This is shown in Figure 1-19.

```
1 @Log
2 @Testcontainers
3 public class HelloResIT {
4
5     @Test
6     void testHello() throws Exception {
7         var request = HttpRequest.newBuilder()
8             .uri(URI.create(restUrl + "/hello-world/John"))
9             .header("Content-Type", "application/json")
10            .GET().build();
11         log.log(Level.INFO, request.uri().toURL().toExternalForm());
12
13         var response = httpClient.send(request, HttpResponse.BodyHandlers.ofString());
14         assertNotNull(response);
15         assertNotNull(response.body());
16
17         log.log(Level.INFO, response::body);
18
19         JsonObject jsonObject = jsonb.fromJson(response.body(), JsonObject.class);
20
21         assertNotNull(jsonObject);
22         assertEquals("Hello there John", jsonObject.getString("greeting"));
23         assertEquals("Getting started with Jakarta EE!", jsonObject.getString("message"));
24         assertEquals("Jakarta EE", jsonObject.getString("platform"));
25         assertEquals("10.0.0", jsonObject.getString("platformVersion"));
26         assertEquals("payara-6-community", jsonObject.getString("implementation"));
27         assertNotNull(jsonObject.get("date"));
28
29         log.log(Level.INFO, jsonObject::toString);
30     }
31
32 }
```

**Figure 1-19 showing test assertions. Previously discussed code left out for brevity.**

The testHello method shown in Figure 1-19 uses the Java HTTP client to invoke the hello-world resource method in the HelloResource class. The rest of the method uses the same assertions we have seen throughout this guide to assert that the returned data is correct. Note that the assertions on Lines 25 and 26 assert for the real data as returned by the ConfigPropertyProvider controller. This is because this test is firing against an actual deployment of the application in a Docker container.

Testcontainers has different ways of spawning different services. I encourage you to check out the [docs](#) page for in-depth guides on how to use the library for your testing needs. Arquillian also does have an extension similar to Testcontainers for spawning containers. Take a look at [Arquillian Cube](#) for more information on how to use this extension.

## Other Test Libraries

The libraries covered in this guide are by no means an exhaustive list of all the options available to you for testing Jakarta EE applications. However, these are the most popular ones that are very pervasive in the ecosystem. JUnit for instance has influenced almost every other testing library on the Java platform. An important take away from the discussions in this guide is that testing a Jakarta EE application is no different from testing any Java based application. Some test types naturally require some initial plumbing, but so will any non-trivial test.

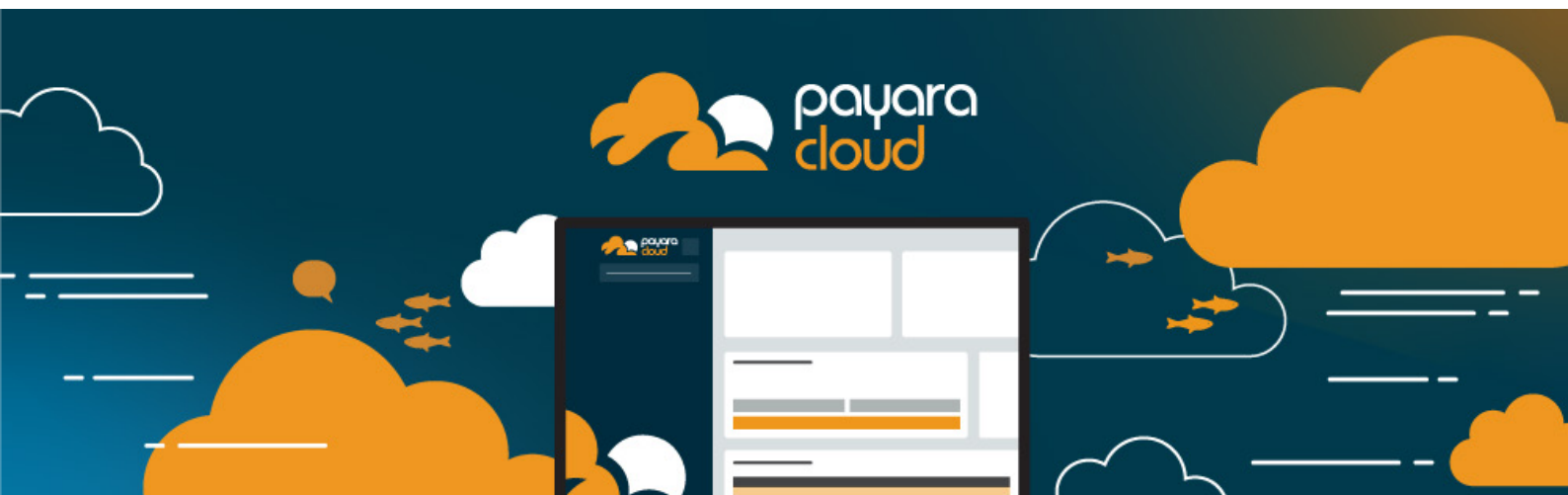
You might be wondering if I have any recommendations as to which library to use. My answer is that it mostly depends. Is the application a greenfield one? Is it a legacy for which you are adding tests? What is the mix of existing and proposed tests? Are there test reports and coverage? Every application domain is different and highly contextual. As such the best way to create a good, reliable and maintainable testing regime is to carry out an analysis of the application within the context of business objectives.

## Summary

In this guide, we have taken a look at the theoretical foundations of the Jakarta EE application development platform, how it relates to the Eclipse MicroProfile project, then looked at principles of testing as listed by the ISTQB. We've then investigated the types of tests and finally the most popular, far reaching Java libraries for creating unit and integration tests.

Since you are reading this guide, you most likely are a Jakarta EE developer. Do you know there is a new, easier, convenient, faster and relatively cheaper way to deploy and host your Jakarta EE applications in the cloud? This new Jakarta EE focused PAAS handles all the DevOps related tasks and infrastructure for you - it completely abstracts you from Docker, Kubernetes and other such complex infrastructure. All you need to do is simply upload your application artefact or .war file and watch it deploy and run instantly in the cloud - almost like magic.

Check out the new Payara Cloud and speak to us today on how we can help you leverage the power of this platform to cut down on your application development costs today.



**CLICK HERE**



**sales@payara.fish**



**UK: +44 800 538 5490  
Intl: +1 888 239 8941**



**www.payara.fish**

Payara Services Ltd 2022 All Rights Reserved. Registered in England and Wales; Registration Number 09998946  
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ