Beginners Overview Guide to Java Runtimes

The Payara® Platform - Production-Ready,
Cloud Native and Aggressively Compatible.

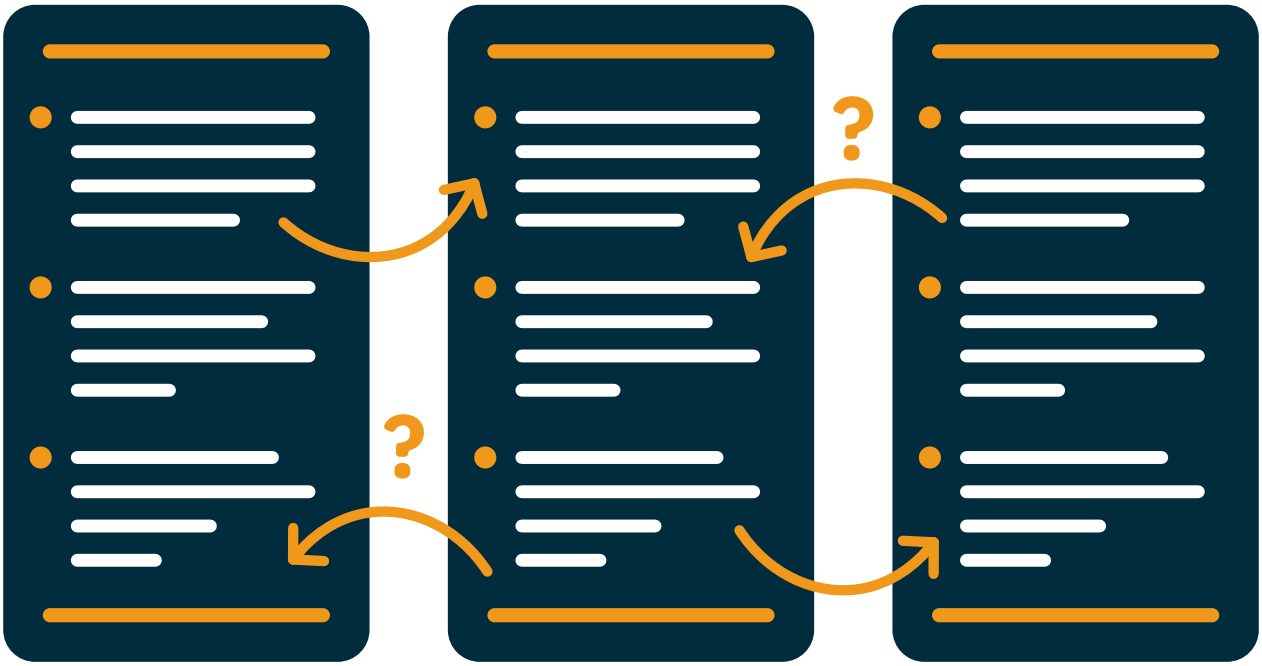
User Guide

Contents

Introduction

When running Java Web applications, you'll come across many terms including Web Server, Application Server, and Application Runtime, among others .... But what exactly are these and what is the difference between all of them? There is a lot of confusion around the terminology and the meaning of each word.

In this guide we dive a little bit deeper into this to help eliminate some of that confusion while indicating how and when to use the different types of Java Runtimes.



Terminology Explained

What is a Web Server?

You probably know that HTTP, HyperText Transfer Protocol, is the most important protocol used on the Internet to communicate between different machines. The protocol standardizes how information is sent between parties. The first version was text-based as the main parts in the exchange are human-readable characters and binary information is converted to text using a BASE64 encoding. It transfers the data and some meta-data that identifies the data or the format in which the data is sent. For example, HTTP indicates what needs to happen as you have methods like GET, POST, and DELETE, and what the requester's intentions are for retrieving, updating, or removing actions.

If you look up the definition of the word Web Server, you'll find that a server is capable of handling HTTP requests. Web Servers understand the protocol and can reply to those requests in a format the client understands.

In the Java world, the Tomcat Server is many times mentioned as an example of a Web Server. Although the Tomcat Server indeed can respond to HTTP requests through the servlet specification, it can do more than that. Since it also has an implementation available for the Java Server Pages (JSP) specification, you can also develop dynamic applications with Tomcat. Tomcat also supports more than just the HTTP protocol as it has the capability of handling WebSocket Connections which are using a different mechanism than HTTP.

What is an Application Server?

The term Application Server is sometimes associated with cumbersome, slow and big. But modern Application Servers are not slow or cumbersome, and in most cases perform much better than a Web Server where all the required frameworks are added to match the functionality of Application Servers. But more on that later.

Although Tomcat is capable of hosting Java applications it is not considered an application server in the true sense of the word. Another set of servers is denoted as Application Servers that have much broader support for developing and running applications on them.

Looking up the definition of Application Server, the main differentiator between Web Servers and Application Servers is that Application Servers one can handle a whole range of connection types, while Web Servers can only handle HTTP connections. Application Servers are based on HTTP but can also support RMI and other types of connections like JMS and SOAP.

What is an Application Runtime?

To avoid some of the confusion around Web Servers and Application Servers, the term Application Runtime is now used in many cases. An Application Runtime indicates that you have a program that runs your application. It does not use the term 'Server' with all its negative connotations and the name makes it clear that it runs your application. Since almost everything is an application today and there is no need to supply static web pages anymore, this makes sense. Some projects serve the static content like javaScript files and images through a Web Server, but most of the time the Application Runtime handles the static content as well.

How can we classify the Application Runtimes if the terms Web Server and Application Server are not appropriate, as there are different groups of runtimes today? The following sections propose an aggregation of the type of applications that are most suitable for each runtime.

Application Runtimes

Web Server Runtimes

There is a group of runtimes that are sometimes still referred to as Web Servers, including Tomcat, Jetty, Undertow, and Netty. This group have some functionality on board to run an application, which means they are sometimes referred to as application servers or application runtimes. But unlike Application Servers, they don't share a common set of functionalities or libraries. The Tomcat, Jetty, and Undertow libraries all have the Servlet specification on board. Since we already explained that HTTP, supported by the Servlet specification, is the most important communication, it is enough for some types of applications. The browser, or the client program in general, can send an HTTP request to the server that responds with a Web page. The user interacts with it and sends some information back to the server.

Although this kind of functionality might be enough to run application s, it will not be the easiest way to build your application. You will need to develop a lot of the basic functionality and infrastructure code yourself, which is not an ideal way to build applications, and the reason Tomcat, Jetty, and Undertow are part of larger runtimes. They still handle the more low-level details, but the larger runtimes they are part of have additional features to make developing applications easier and faster, so you don't need to deal with the low-level stuff manually.

Netty is a bit different from other Web Servers in the sense that it is based on what is called Non-blocking IO (NIO). In traditional programming, you often encounter the situation of your code needing to wait for the response of some other party. This can be a request to the database or even the client that reads the response, and your application needs to wait before it sends more data. Netty uses

an asynchronous event-driven approach which makes it ideal to write network applications. While the asynchronous event-driven approach results in more efficient applications, they are also harder to write, maintain, and extend. But the downside is if some of the parts of your application are not using the reactive approach, such as database access, you'll lose some of the performance benefits yet still have to deal with having more complex code.

Runtimes with a Minimal Core

As mentioned in the previous section, using Tomcat and other Web Servers require writing a lot of the code and infrastructure code yourself as it is not available on the runtime.

But it's simply not efficient to write the code yourself, particularly when you need to implement the REST specification, for example. There are many libraries that offer a minimal core of libraries, including the ability to implement REST on top of the HTTP specification, so you should use that instead of trying to do it yourself with a Web Server like Tomcat.

Spring Framework

The Spring Framework is an example of this type of runtime. It is a set of libraries centered around Spring Core that provide Dependency Injection for your application. With the Dependency Injection, you are not instantiating services yourself, but the system will instead provide you with a suitable implementation of the functionality you require. So, when your code needs a Payment Service, for example, it requests a suitable implementation from the system, like implementing the Java Interface you use in your code. The system is then responsible for creating and maintaining that instance of the Payment Service. It can also create a Proxy for that instance to add functionality based on some annotations or configuration, such as transaction management, logging, and security.

The Spring Framework consists of more than 30 modules, each implementing specific functionality like REST functionality. You can add each of them to your application to reduce the amount of code you need to write. This way, you can assemble a runtime with just enough functionality required for your application. Spring applications can be run with any runtime that support Servlets, including Tomcat and Jetty.

Assembling only the libraries you need to run your applications has some advantages over Application Runtimes that load all libraries whether you use them or not, but there are also some drawbacks associated:

- The different Spring modules have dependencies on each other. As mentioned, they all use the Spring Core module to provide the basic Dependency Injection and Bean functionality, for example. But the version of Module B that you want to use can depend on another version of the Core module than the version of Module A that you have. And those different versions of Core might contain incompatible changes. This makes it sometimes difficult, or even impossible, to determine the correct combination of modules that you can use for your application when attempting to add only the modules you need for your application.
- Upgrading can also be challenging due to the diverse set of Modules and dependencies. Using a new version of Module A might not be possible as one of the dependencies has an incompatible version of the dependencies that are also used in Module B.

Helidon or Quarkus

There are other solutions using the same minimal core concept as the Spring Framework, including Quarkus and Helidon, but they use a slightly different approach. Helidon and Quarkus are based on the MicroProfile specifications, and they bundle a few Jakarta EE specifications with the different implementations that come in handy in large microservices architecture, such as fault tolerance.

If you need additional functionality, such as database access, you need to add certain dependencies to your application to make use of it. The difference between solutions like Quarkus or Helidon compared to the Spring Framework is that all the different modules are always aligned with each other. The modules are not independently maintained, as is the case with the Spring Modules. All the modules are released together which means there are no compatibility issues between modules, and upgrades can be carried out smoother than with Spring.

These runtimes also have the GraalVM native compatibility capability in common. The applications can be combined with the runtime as a native executable for your target platform. This makes them a good candidate if you are running a serverless architecture where you need a fast startup of your application to respond to a user request. The downside is that you need to follow some guidelines which requires that you rewrite your applications, and you can't use all Java functionality, like reflection, as those are not supported by the native compilation.

Jakarta EE Compatible Application Runtimes

Jakarta EE is the new home of the Java Enterprise specifications of Java EE. Jakarta EE compatible runtimes are similar to the Spring Framework in that they provide you with many of the implementations you need in each project like REST and database access but takes it a bit further in that you don't have to specify the modules needed or code everything yourself. The Jakarta EE Full Profile, (or the Jakarta EE Web Profile if you only need a limited set of specifications), integrates all modules. This also means the Web application WAR file is very small and you can deploy them very fast on the Jakarta EE runtimes. The runtimes themselves don't need to be large, either. For example, Payara Micro is only about 75Mb large and contains all Jakarta EE Web Profile specifications in combination with the concurrency, JBatch, cache specification, and all MicroProfile specifications. A Spring application supporting all these specifications in combination with Tomcat is larger.

Another advantage of Jakarta EE is the backward compatibility of the specifications. With the release of Jakarta EE 9 in December 2020, there was a breaking change performed in the namespaces due to legal reasons associated with the donation of the Jakarta EE code from Oracle to the Eclipse Foundation. But now that the code is with Eclipse, there will be no breaking changes over time.

Many Jakarta EE compatible runtimes, including the Payara Platform products, ensure you can run the older and newer versions on the same runtime. Currently, you can run a Jakarta EE 9 application, using the Jakarta package names, on Payara Server and Payara Micro, which is also capable of running Jakarta EE 8 or older programs. And in the future, you will still be able to run older versions of Jakarta EE applications on the latest versions of Payara, with no need to rewrite applications, even though there is a difference in the package names.

This allows you to keep using the same runtime for applications using the older and newer namespace. You can perform the namespace change gradually without the need to use different versions of the product as you can use the same Payara version for both of them. This simplifies the Ops handling during the transition period and allows you to concentrate on the business value and logic instead of infrastructure.

Which Type of Runtime is Right for You?

The term Application Runtime is more appropriate today as most of the systems we use are serving applications to the end-users. Serving static pages and resources for most modern applications is uncommon or not sufficient for an application. So, the term Web Server, although still valid, should not be used for a product that serves Java applications like Tomcat. However, Tomcat has only a limited set of features onboard to develop applications , which means you need to add them unless you want to write infrastructure code yourself.

Using the Spring Framework instead of an Application Runtime like Tomcat will handle this for you, but since all modules are maintained individually, combining and upgrading them can be a very challenging task.

Other products like Quarkus and Helidon solve this problem in a better way since all modules are released together all the time. While these products are geared toward Java Native compilation to achieve fast startup and lower memory consumption, it comes with a huge cost as you need to follow their guidelines to achieve those benefits, requiring a complete rewrite of the applications from scratch.

The Jakarta EE runtimes like the Payara Platform have some advantages compared to the other products. They have everything you need on board already, which means you do not need to struggle to assemble your runtime. They also provide backward compatibility to make it easy to upgrade from one version to another. And if you want to achieve a faster startup this can be realised through the Class Data Sharing option of the JVM without the need to rewrite or adapt your application.

Learn more about the Payara Platform: www.payara.fish

sales@payara.fish

+44 207 754 0481

www.payara.fish