JAKARTA EE

You don’t always need an application server to run Jakarta EE applications

Depending on the requirements, you can do well with Helidon, Piranha, or Hammock.

by Arjan Tijms

Jakarta EE (formerly Java EE) and the concept of an application server have been intertwined for so long that it’s generally thought that Jakarta EE implies an application server. This article will look at whether that’s still the case—and, if Jakarta EE isn’t an application server, what is it?

Let’s start with definitions. The various Jakarta EE specifications use the phrase *application server* but don’t specifically define it. The phrase is often used in a way where it would be interchangeable with terms such as *runtime*, *container*, or *platform*. For instance, the specification documents from the following specs mention things like the following:

- Jakarta authorization: “The Application server must bundle or install the PolicyContext class…”
- Jakarta messaging: “A ServerSessionPool object is an object implemented by an application server to provide a pool of ServerSession objects…”
- Jakarta connectors: “This method is called by an application server (that is capable of lazy connection association optimization) in order to dissociate a ManagedConnection…”

The Jakarta EE 9 platform specification doesn’t explicitly define an application server either, but section 2.12.1 does say the following:

> A Jakarta EE Product Provider is the implementor and supplier of a Jakarta EE product that includes the component containers, Jakarta EE platform APIs, and other features defined in this specification. A Jakarta

> EE Product Provider is typically an application server vendor, a web server vendor, a database system vendor, or an operating system vendor. A Jakarta EE Product Provider must make available the Jakarta EE APIs to the application components through containers.

The term *container* is equivalent to *engine*, and very early J2EE documents speak about the “Servlet Engine.”

Thus, the various specification documents do not really specify what an application server is, and when they do mention it, it’s basically the same as a container or runtime. In practice, when someone speaks about an application server, this means something that includes all of the following:

- It is separately installed on the server or virtual machine.
- It listens to networking ports after it is started (and typically includes an HTTP server).
- It acts as a deployment target for applications (typically in the form of a well-defined archive), which can be both deployed and undeployed.
- It runs multiple applications at the same time (typically weakly isolated from each other in some way).
- It has facilities for provisioning resources (such as database connections, messaging queues, and identity stores) for the application to consume.
- It contains a full-stack API and the API’s implementation for consumption by applications.

In addition, an application server *may* include the following:

- A graphical user interface or command-line interface to administer the application server
- Facilities for clustering (so load and data can be distributed over a network)

Pushback against the full-fledged application server

The application server model has specific advantages when shrink-wrapped software needs to be deployed into an organization and integrated with other software running there. In such situations, for example, an application server can eliminate the need for users to authenticate themselves with every application. Instead, the application server might use a central Lightweight Directory Access Protocol (LDAP) service as its identity store for employees, allowing applications running on the application server to share that service.

This model, however, can be less ideal when an organization develops and operates its own public-facing web applications. In that case, the application needs to exert more control. For instance, instead of using the same LDAP service employees use, the organization’s customers would have a separate

registration system and login screen implemented by the developers.

Here, an application server can be an obstacle because part of the security would need to be done *outside* of the application, often by an IT operations (Ops) team, who may not even know the application developers.

For example, an application server that hosts production applications is often shielded from the developer (Dev) team and is touched by only the Ops team. However, some problems on the server really belong in the Dev domain because they apply to in-house application development. This can lead to tickets that bounce between the Dev and Ops teams as the Dev team tries to steer the actions that only the Ops team is allowed to perform. (Note: The DevOps movement is an attempt to solve this long-standing problem.)

Another issue concerns the installed libraries within the application server. The exact versions of these the libraries, and the potential need to patch or update them, is often a Dev concern. For instance, the Ops team manages routers, networks, Linux servers, and so on and that team might not be very aware of the intricate details of Mojarra 2.3.1 versus Mojarra 2.3.2 or the need to patch the `FacesServlet` implementation.

Modern obsolescence: Docker and Kubernetes

The need for having an installed application server as the prime mechanism to share physical server resources started to diminish somewhat with the rise of *virtual servers*. A decade ago, you might see teams deploying a single application to a single dedicated application server running inside a virtual server. This practice, though, was uncommon enough that in 2013 the well-known Java EE consultant Adam Bien wrote a dedicated blog post about this practice that received some pushback. One of the arguments against Bien's idea was that running an entire (virtual) operating system for a single application would waste resources.

Almost exactly at the same time as Bien wrote his post, the Docker container platform was released. Containers run on a single operating system and, therefore, largely take the resource-wasting argument away. While containers themselves had been around since the 1970s, Docker added a deployment model and a central repository for container images (Docker Hub) that exploded in popularity.

With a deployment tool at hand, and the knowledge that fewer resources are wasted, deploying an application server running a single application finally went mainstream. Specifically, the native deployment feature and, above all, the *undeployment* feature of an application server are not really used, because most Docker deployments are static.

In 2015 the Kubernetes container orchestration system appeared, and it quickly became popular for managing many instances of (mostly) Docker containers. For Java EE application servers, Kubernetes means that Java EE's native clustering facilities are not really used, because tasks such as load balancing are now managed by Kubernetes.

Around the same time, the *serverless* microservices execution model became more popular with cloud providers. This meant that the deployment unit didn't need its own HTTP server. Instead, the deployment unit contained code that is called by the *serverless server* of the cloud provider. The result was that for such an environment, the built-in HTTP server of a Java EE or Jakarta EE application server is not needed anymore. Obviously, such code needs to provide an interface the cloud provider can call. There's currently no standard for this, though Oracle is working with the Cloud Native Computing Foundation on a specification for this area.

The Jakarta EE APIs

Without deployments, without running multiple applications, without an HTTP server, and without clustering, a Jakarta EE application server is essentially reduced to the Jakarta EE APIs.

Interestingly, this is how the Servlet API (the first Java EE API) began. In its early versions, servlet containers had no notion of a deployed application archive and they didn't have a built-in HTTP server. Instead, servlets were functions that were individually added to a servlet container, which was then paired with an existing HTTP server.

Despite some initial resistance from within the Java EE community, the APIs that touched the managed-container application server model started to transition to a life outside the application server. This included HttpServletRequest#login, which began the move away from the strict container-managed security model, and @DataSourceDefinition, which allowed an application to define the same type of data source (and connection pool) that before could be defined only on the application server.

In Java EE 8 (released in 2017), application security received a major overhaul with Jakarta Security, which had an explicit goal of making security fully configurable without any application server specifics.

For application servers, several choices are available, such as Oracle WebLogic Server.

What if you're not looking for a full application server? Because there is tremendous value in the Jakarta EE APIs themselves, several products have sprung up that are *not* application servers but that do provide Jakarta EE APIs. Among these products are

Helidon, Piranha Cloud, and Hammock, which shall be examined next.

Project Helidon

Project Helidon is an interesting runtime that's not an application server. Of the three platforms I'll examine, it's the only one fully suitable for production use today.

Helidon is from Oracle, which is also known for Oracle WebLogic Server—one of the first application servers out there (going back all the way to the 1990s) and one that best embodies the application server concept.

Helidon is a lightweight set of libraries that doesn't require an application server. It comes in two variants: Helidon SE and Helidon MP (MicroProfile).

Helidon SE. Helidon SE does not use any of the Servlet APIs but instead uses its own lightweight API, which is heavily inspired by Java functional programming. The following shows a very minimal example:

```
WebServer.create(
    Routing.builder()
      .get(
          "/hello",
          (request, response) -> response.sen
      .build())
      .start();
```

Here's an example that uses a full handler class, which is somewhat like using a servlet.

```
WebServer.create(
    Routing.builder()
           .register(
               "/hello",
               rules -> rules.get("/", new He
           .build())
           .start();

public class HelloHandler implements Handler
    @Override
    public void accept(ServerRequest req, Ser
        res.send("Hi");
    }
}
```

In the Helidon native API, many types are functional types (one abstract method), so despite a class being used for the `Handler` in this example to contrast it to a servlet, a lambda could have been used. By default, Helidon SE launches an HTTP server in these examples at a random port, but it can be configured to use a specific port as well.

Helidon MP. With Helidon MP, you declare a dependency to Helidon in the `pom.xml` file and add it to the Helidon parent POM. You then write normal MicroProfile/Jakarta EE code (for the Jakarta EE libraries that Helidon MP supports). After the build, you get a runnable JAR file including your code. A minimal example for such a `pom.xml` file looks as follows:

```
<project xmlns="http://maven.apache.org/POM/4
    xmlns:xsi=
        "http://www.w3.org/2001/XMLSchema-ins
    xsi:schemaLocation=
        "http://maven.apache.org/POM/4.0.0 ht
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>io.helidon.applications</gro
        <artifactId>helidon-mp</artifactId>
        <version>2.2.2</version>
        <relativePath />
    </parent>

    <groupId>com.example</groupId>
    <artifactId>example</artifactId>
    <version>1.0</version>

    <dependencies>
        <dependency>
            <groupId>io.helidon.microprofile.
            <artifactId>helidon-microprofile<
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plu
                <artifactId>maven-dependency-
                <executions>
                    <execution>
                        <id>copy-libs</id>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

This runnable JAR file neither contains nor loads the Helidon libraries. Instead, those libraries are referenced via `META-INF/MANIFEST.MF` to be in the `/libs` folder relative to where the executable JAR file exists.

After building, such as by using the `mvn` package, you can run the generated JAR file using the following command:

```
java -jar target/example.jar
```

This command will start the Helidon server, and this time you do get a default port at 8080. Besides the MicroProfile APIs, which is what Helidon focuses on primarily, the following Jakarta EE APIs are supported:

- Jakarta CDI: Weld
- Jakarta JSON-P/B: Yasson
- Jakarta REST: Jersey
- Jakarta WebSockets: Tyrus
- Jakarta Persistence: EclipseLink, Hibernate
- Jakarta Transactions: Narayana

You can read more about Helidon in "Helidon: A simple cloud native framework," by Todd Sharp, and "Fast, flexible data access in Java using the Helidon microservices platform," by Paul Parkinson.

Piranha Cloud

Piranha Cloud is a relatively new project; although it started as a very low-key project a couple of years ago, it didn't pick up pace until October 2019, as you can see in **Figure 1**, which shows the GitHub commit graph.

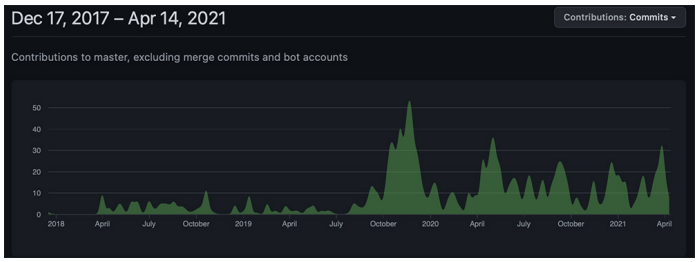

Figure 1. The GitHub commit graph for Piranha Cloud

As of mid-April 2021, Piranha was not yet production-ready, and it is mostly of interest to developers who want to see how a Jakarta EE and MicroProfile runtime is being built and is evolving.

Piranha comes in four versions: Nano, Embedded, Micro, and Server. Each version adds features essentially following the list mentioned earlier for application server functionality. **Figure 2** is a high-level overview of the Piranha architecture with respect to the four versions.

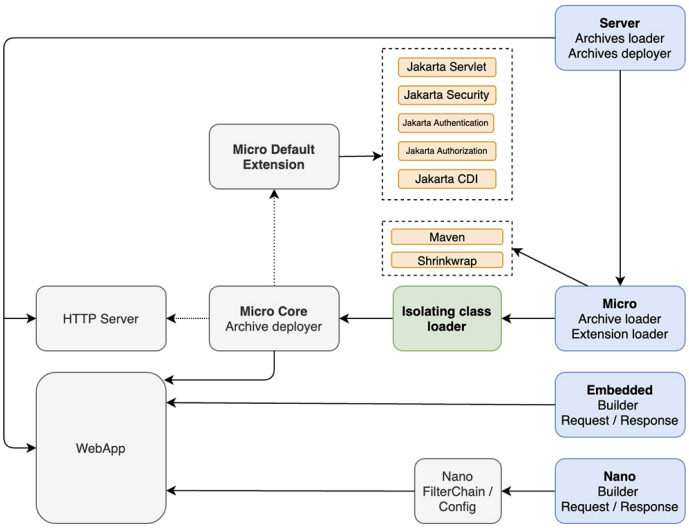

Figure 2. The Piranha Cloud architecture

Here's more detail about each version.

Piranha Nano. Piranha Nano essentially runs only a single servlet on a flat classpath, forgoing several servlet features, for example, many of the listeners and support for sessions. It's therefore specifically suited for serverless computing using a familiar Servlet API that's a subset of the full API. This means servlets created for Piranha Nano will run on other servlet containers as well, but Nano will not run all regular servlet applications.

Piranha Nano is programmatically set up, for instance, from a `main()` method in a regular Java class. Here is a somewhat contrived Hello World example; running this from a `main` method will yield "Hello, World!"

```
ByteArrayOutputStream outputStream = new Byte

new NanoPiranhaBuilder()
    .servlet("HelloWorldServlet", new HttpSer
        protected void doGet(HttpServletReque
            throws IOException, ServletExcept
            response.getWriter().print("Hello
        }})
    .build()
    .service(
        new NanoRequestBuilder()
            .method("GET")
            .servletPath("/index.html")
            .build(),
        new NanoResponseBuilder()
            .outputStream(outputStream)
            .build()
    );

System.out.println(outputStream.toString());
```

As mentioned earlier, Piranha Nano will not run every existing servlet. It can run Jakarta Server Pages and Apache Wicket pages.

Piranha Embedded. Like Piranha Nano, Piranha Embedded runs on a flat classpath and is programmatically set up. It supports the full Servlet API—well, that's the goal. As of the time of this writing, it passes about 92% of the Jakarta EE 9.1 Servlet TCK (Technology Compatibility Kit).

Piranha Embedded doesn't start an HTTP server. Requests and responses can be programmatically created and passed in, but there are also various convenience methods using default versions of those. Using the programmatic API, a representation of a web application archive (WAR) is created, and the API can create elements such as `web.xml` when needed. No classes or JAR files need to be added. Because of the flat classpath, those are directly available to Piranha Embedded from the classpath used by the code that embeds it.

Because of this, Piranha Embedded can be used in the same way as a mocking framework for various Jakarta libraries, with the important difference that it's testing against an actual implementation. Here's an example.

```
System.out.println(
    new EmbeddedPiranhaBuilder()
        .stringResource("/index.xhtml", """
            <!DOCTYPE html>

            <html lang="en" xmlns:h="http://x
                <h:head>
                    <title>Hello Jakarta Face
                </h:head>
                <h:body>
                    Hello Jakarta Faces
                </h:body>
            </html>
            """)
        .initializer(MojarraInitializer.class
        .buildAndStart()
        .service("/index.xhtml")
        .getResponseAsString());
```

In this example, a resource named `index.xhtml` with a simple Facelets message as content is set in the root of a temporary web application. In addition to this single resource, a `ServletContainerInitializer` is set; in this example, it's one that initializes Mojarra (a Jakarta Server Faces implementation). Mojarra itself is on the classpath of this code.

The `buildAndStart()` runs the initializer, causing Mojarra to start and install the `FacesServlet` mapped to `*.xhtml`. No HTTP server is started. When the code calls the `service()` method for `index.xhtml`, this results in an `EmbeddedRequest` being created and passed into the

`EmbeddedPiranha` code, eventually reaching the `FacesServlet` code. This servlet will then try to find `index.xhtml` via the internal `ServletContext`, which will return the content set via the builder used in the code above. Mojarra then processes the template and writes the response, which is the object returned by the `service()` method.

If you need more control over the request and even the response, you can create those objects yourself and pass them into the service method.

```
EmbeddedPiranha piranha =
    new EmbeddedPiranhaBuilder()
    .stringResource("/index.xhtml", """
        <!DOCTYPE html>

        <html lang="en" xmlns:h="http://xmlns
            <h:head>
                <title>Hello Jakarta Faces</t
            </h:head>
            <h:body>
                Hello Jakarta Faces
            </h:body>
        </html>
        """)
    .initializer(MojarraInitializer.class)
    .buildAndStart();

EmbeddedRequest request =
    new EmbeddedRequestBuilder()
        .contextPath("")
        .servletPath("/index.html")
        .build();
EmbeddedResponse response = new EmbeddedRespo

piranha.service(request, response);
```

Piranha Micro. Piranha Micro builds on the same core as Piranha Embedded, but it adds several major components, including an isolating class loader, the ability to run full WAR files, an optional HTTP server, the ability to run from the command line, and an extension mechanism for Jakarta EE and Jakarta MP components. When no such extension is specified, Piranha Micro uses a default extension that supports the following minimal set of Jakarta EE components:

- Jakarta Servlet (and transitive dependencies of Jakarta Server Pages and Jakarta Expression Language)
- Jakarta Security (and transitive dependencies of Jakarta Authentication and Jakarta Authorization)
- Jakarta CDI (and transitive dependencies for Jakarta DI and Jakarta Interceptors)

Unlike many other servlet containers, Piranha Micro does not include a native security implementation on top of which the Jakarta security APIs are layered. Instead, Jakarta Security directly provides the implementation for servlet security. This

means, for example, that the `FORM` authentication mechanism configured in `web.xml` is backed by the same code as the one configured by `@FormAuthenticationMechanismDefinition`.

For programmatic and embedded usage, Piranha Micro's isolating class loader is an important asset. It fully shields the code running within Piranha Micro from the environment in which it is embedded. This is very useful for those situations where, for instance, CDI beans and service loaders from the environment should not be picked up by the code running within Piranha Micro. It's a trade-off, though, because the full isolation makes any communication between the embedding and embedded code more difficult. Plus, there's a potential higher cost in memory usage.

Piranha Micro uses ShrinkWrap as its native format to handle WAR files. ShrinkWrap is easy to use to programmatically create archives of all kinds, and it directly connects to tools such as the Arquillian microservices test suite, which uses the same archive format.

Using a builder similar to the one used by Piranha Embedded, you can mimic the example used above but this time using Piranha Micro.

```
System.out.println(new MicroEmbeddedPiranhaBu
    .archive(
        ShrinkWrap
            .create(WebArchive.class)
              .addAsWebResource(new StringAss
                  <!DOCTYPE html>

                  <html lang="en" xmlns:h="ht
                      <h:head>
                          <title>Hello Jakart
                      </h:head>
                      <h:body>
                          Hello Jakarta Faces
                      </h:body>
                  </html>
                  """), "index.xhtml")
              .addAsWebInfResource(EmptyAsset
              .addAsLibraries(
                  Maven.resolver()
                       .resolve(
                           "org.glassfish:jak
                           "jakarta.websocket
                       .withTransitivity().as
    .buildAndStart()
    .service("/index.xhtml")
    .getResponseAsString());
```

This code creates a ShrinkWrap archive containing the same Facelet as in the previous example, but it also creates an empty `faces-config.xml` to trigger the initialization of Jakarta Server Faces and it includes Mojarra (a Jakarta Server Faces implementation JAR file) in the archive.

Using the same ShrinkWrap API, you can load an existing WAR file from disk, add individual files from disk, add copies of classes from the classpath, and so on.

For command-line usage, a JAR file named `piranha-micro.jar` is available, which can be used to start a web application from a file, for example. This version starts an HTTP server by default. Using it looks as follows:

```
java -jar piranha-micro.jar --war someapp.war
```

The default port is 8080, so assuming the archive shown in the example above was saved to `someapp.war` on disk, you would be able to request the same page using the following:

```
wget localhost:8080/index.xhtml
```

An interesting aspect of `piranha-micro.jar` is that it has the feel of a hollow JAR file (prepackaged runtime with all its dependencies in one JAR file), but it's actually a loader. The JAR file contains its own (shaded) copy of Maven, which it uses to load the Piranha Micro core classes and dependencies, as well as any extensions. Furthermore, `piranha-micro.jar` contains ShrinkWrap to load a `.war` from a file or, in exploded form, from a directory. Dependency JAR files and the application archive are loaded and executed from memory, which provides a particular advantage: Contrary to some hollow JAR solutions, no unpacking to a temporary folder is needed.

Piranha Server. As its name implies, Piranha Server comes closest to a traditional application server, although with a twist.

Like a traditional application server, Piranha Server is the only member of the Piranha family that is installed, that functions as a deployment target, and that runs multiple applications. But if those traditional products support a hollow JAR version, that hollow JAR version is technically speaking (almost) the full server, with the server facilities just hidden. For Piranha Server, it's the other way around.

The Piranha Server variant is a small shell that starts an HTTP server and instantiates an embedded Piranha Micro instance (without an HTTP server, obviously) for each deployed application. Because of the strongly isolating class loader used by a Piranha Micro instance, each application uses its own version of the Jakarta EE libraries. This is significantly different from a traditional Jakarta EE server, where those libraries are loaded once and shared by every application that is deployed.

Because it loads a fresh set of Jakarta EE libraries for each application, Piranha Server can support different versions of Jakarta EE simultaneously, as shown in **Figure 3**.

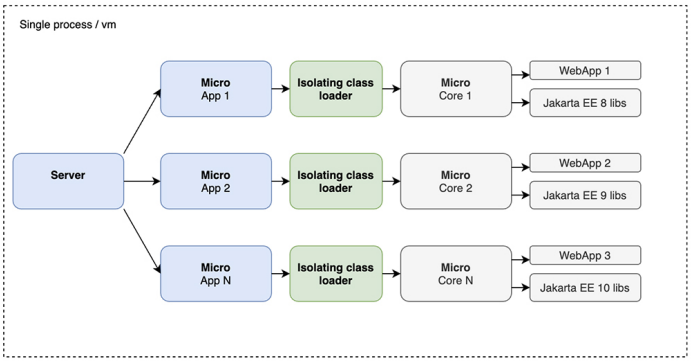

Figure 3. The architecture of Piranha Server running instances of Piranha Micro

Piranha Server obviously uses much more memory when running multiple applications than a traditional application server would, especially when many applications are deployed (say, tens or even hundreds). However, it uses fewer resources than running many servlet runtimes, each with a single application.

By the way, Piranha Server can be configured to use Piranha Embedded instead of Piranha Micro, thus allowing the applications to share the Jakarta EE libraries, as shown in **Figure 4**. A future variant of Piranha Server will allow you to mix Piranha Micro and Piranha Embedded applications.

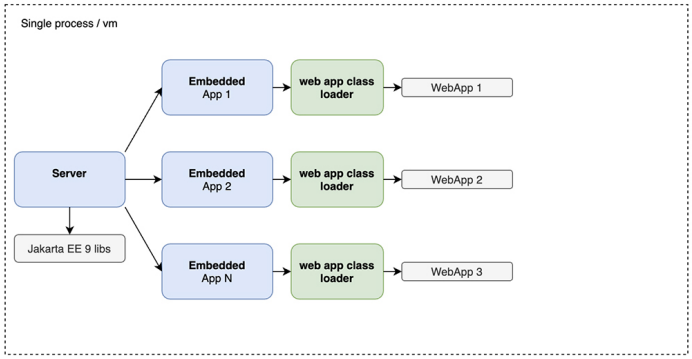

Figure 4. The architecture of Piranha Server running instances of Piranha Embedded

Hammock

Hammock was one of first runtimes to use the Java EE libraries without being a full-fledged application server. The Hammock project was started in 2014 by John Ament as a combination of RESTEasy (Jakarta REST), Undertow (Jakarta Servlet), and Weld (Jakarta CDI).

Hammock describes itself as follows:

> Hammock is not an application server. It has no concept of EJB support, it doesn't support any of the management extensions or deployment requirements. It doesn't run WAR files, it uses uber-jars to have a simple executable or can be deployed exploded.

Hammock thus focuses on the uber-JAR concept, where a developer adds Hammock as a dependency to a project, and the build then results in a runnable JAR file that contains both Hammock and the application code. With Hammock, there is no concept whatsoever of installing anything, functioning as a deployment target, or running multiple applications.

Something that sets Hammock aside is that it started to support alternative implementations for all the Jakarta APIs that it uses via pluggable Maven dependencies, such as `ws.ament.hammock:bootstrap-weld3` for CDI.

Hammock supports the following:

- Jakarta CDI: Weld, OpenWebBeans
- Jakarta JSON-P: Johnzon
- Jakarta Servlet: Tomcat, Undertow, Jetty
- Jakarta REST: CFX, Jersey, RESTEasy
- Jakarta Persistence: Hibernate, EclipseLink, OpenJPA
- Jakarta Messaging: Artemis

Hammock was among the first products to support the initial MicroProfile specification (which consisted only of Java EE APIs then). It added support for later MicroProfile versions by incorporating Apache components.

Unfortunately, Hammock hasn't been updated for some time; the latest release, version 2.1, was published in July 2018.

Conclusion

This article began by exploring exactly what an application server is, and then it discussed that for those seeking a solid runtime for Jakarta EE code, full application servers are not always needed anymore. Indeed, it explained that the Jakarta EE APIs don't specifically require an application server at all.

The article also looked at several runtime platforms for Jakarta EE that are not application servers but which provide many Jakarta EE APIs. These runtimes show that Jakarta EE is well positioned to transition to a world that doesn't need application servers.

Dig deeper

- Transition from Java EE to Jakarta EE
- Java for the enterprise: What to expect in Jakarta EE 10
- How to build applications with the WebSocket API for Java EE and Jakarta EE
- Helidon: A simple cloud native framework
- Fast, flexible data access in Java using the Helidon microservices platform
- Arquillian: Easy Jakarta EE testing

Arjan Tijms

Arjan Tijms was a JSF (JSR 372) and Security API (JSR 375) EG member and is currently project lead for a number of Jakarta projects including Jakarta- Security, Authentication, Authorization, Faces, and Expression Language. He is the co-creator of the popular OmniFaces library for JSF that was a 2015 Duke's Choice Award winner and is the author of two books: *The Definitive Guide to JSF* and *Pro CDI 2* in Java EE 8. Arjan holds an MSc degree in computer science from the University of Leiden, The Netherlands. Follow Arjan on Twitter at @arjan_tijms.

Share this Page