Dismiss the Java Myths

Get to Know the Real Jakarta EE

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

eBook

Contents

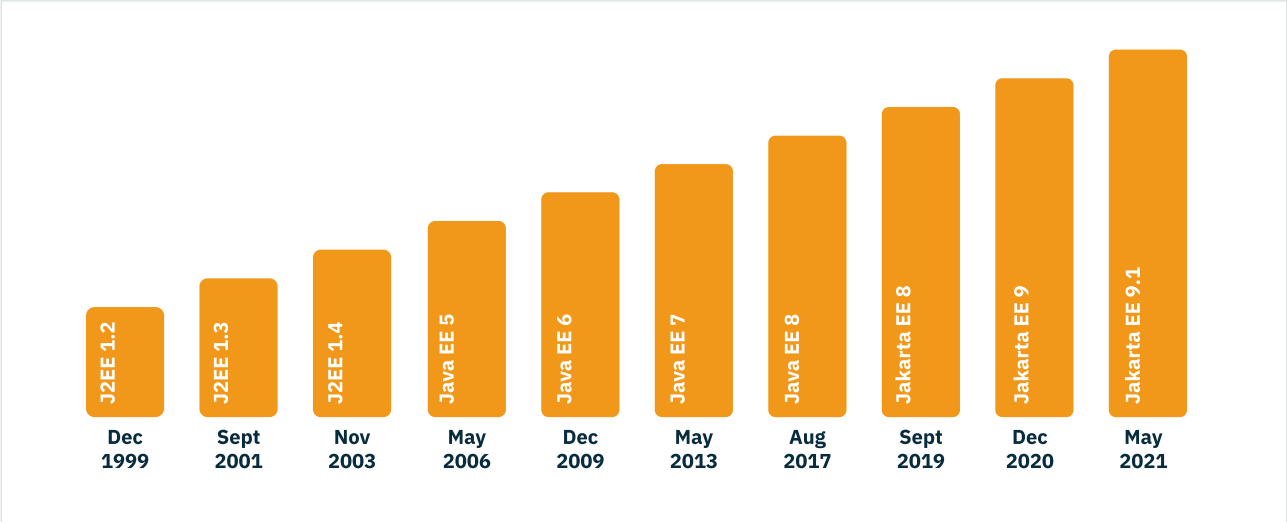
Java was created in 1995 and despite being over 25 years old, it's still one of the most popular and widely used programming languages in the world. Due to its longevity – many myths around Java EE (now Jakarta EE) have circulated.

In this eBook, we'll discuss the most common myths and show you why the programming language is still relevant, how it keeps up with the changes in the IT world and show there is a future in Jakarta EE.

MYTH: Java EE is Outdated and Dead

One of the most common myths about Java EE is that it's too old and outdated to use for modern application development. Fortunately, that's simply not true and you can continue using your existing Java EE development skills to modernize your application development.

Java EE and Java have a related history. Java EE was built for service-side enterprise web application processing. It was a set of standards that used to follow the Java timeline:



The Platform

J2EE 1.2 aligned with Java 1.2. In the initial versions, JSPs were created to compete with CGI scripts. In those days, you built applications with Perl or C code. When J2EE came along, it created the concept of a persistent server instead of CGI, which allowed you to store data and things in memory.

J2EE linked roughly with the release schedule of Java, coming typically 6-12 months after each Java release. The first couple of releases had a couple years between the releases and then it started to get a bit longer in between each release.

J2EE turned into Java EE 5 in May of 2006, and when people talk about how they don't like about J2EE, **they tend to be talking about things that happened *before* Java EE 5**. None of the accusations for the reasons why they don't like J2EE, such as deployment descriptors, XML, entity beans, etc. – none of those things existed after May of 2006 in Java EE 5. This was the first massive modernization of Java EE. So, if someone claims Java EE is old and outdated because of deployment descriptions or entity beans, it just means they haven't kept up with the changes in the industry themselves!

After 2006, Java EE began releasing new modernizations of the API that used predominately annotations. It completely removed the EJB CMP and entity beans model and brought in JPA and those sorts of capabilities.

Java EE added new capabilities for new platforms with each release. For example, Java EE 7 brought in web sockets, JSON processing, and the new web technologies available at that point in time.

Java EE 8 brought in servlet 4 which supported HTTP/2, http push, and new capabilities available in that time frame.

Looking at the timeline, you can see Java EE is not outdated because it has always been tracking the standards and technologies happening in the current times and building those capabilities into each new release.

Then, in September 2019, Java EE turned into Jakarta EE, which further modernized the technology. We'll cover what changed with Jakarta EE and how it's evolved within this eBook.

The Java EE Foundation Technology

Let's take a step back and talk about what Java EE is.

It's more than a platform – Java EE is a set of specifications. And those Java EE specifications are the foundation technology for a lot of server-side frameworks that may be seen as cooler, faster, or better than Java EE. But the reality is, because Java EE/Jakarta EE is the standards body that generates specifications which are appropriate for enterprise, server-side applications, many of the other modern technologies are using Java EE, such as:

- **Application Servers** - Application servers are very tied to the full Java EE platform specifications.
- **Tomcat** -Tomcat is a super powerful servlet container that supports a limited number of Java EE specifications.
- **Microservices Frameworks** -Several microservices frameworks use Java EE specifications, including:
  - Dropwizard - based on Jersey (an implementation of JAX-RS) and Hibernate (an implementation of the Java Persistence API), which are all implementations of Java EE specifications.
  - Micronaut - uses Java x and JAX and beans validation, so it's based on the core Java EE APIs.
  - Spring - often seen as a Java EE/Jakarta EE competitor, but Spring also uses a lot of the foundational Java EE/Jakarta EE technology.
  - Quarkus and ahead-of-time compilation - based on Java EE/Jakarta EE specifications.

Key Specifications

Jakarta EE is basically a group that creates specifications. The most heavily used specifications within application servers, and even outside of application servers, include:

- JPA
- Hibernate
- JPA
- EclipseLink
- JAX-RS
- Restful Web Services
- Dropwizard
- Inject (foundational API for dependency injection)
- Bean Validation
- XML Binding
- JSON-Processing

What is a Jakarta EE Specification?

A specification is different from a framework.

A specification consists of:

- Document - defines the purpose of an API, how it's used, various conditions, and how things should behave.
- API JAR - what you use as a dependency
- TCK - a test suite that can be run against an implementation to prove that it meets the specification

A specification allows for multiple implementations of the same specification and the same API.

For example, EclipseLink and Hibernate are implementations of JPA, and Payara Server and JBoss are implementations of the full platform specification.

A platform, or profile, is basically a specification that sweeps up lots of other specifications within it.

So, a Java EE/Jakarta EE full profile platform is a set of specifications and other requirements for how they work together, and a TCK that proves you can implement it and that your product meets the specification.

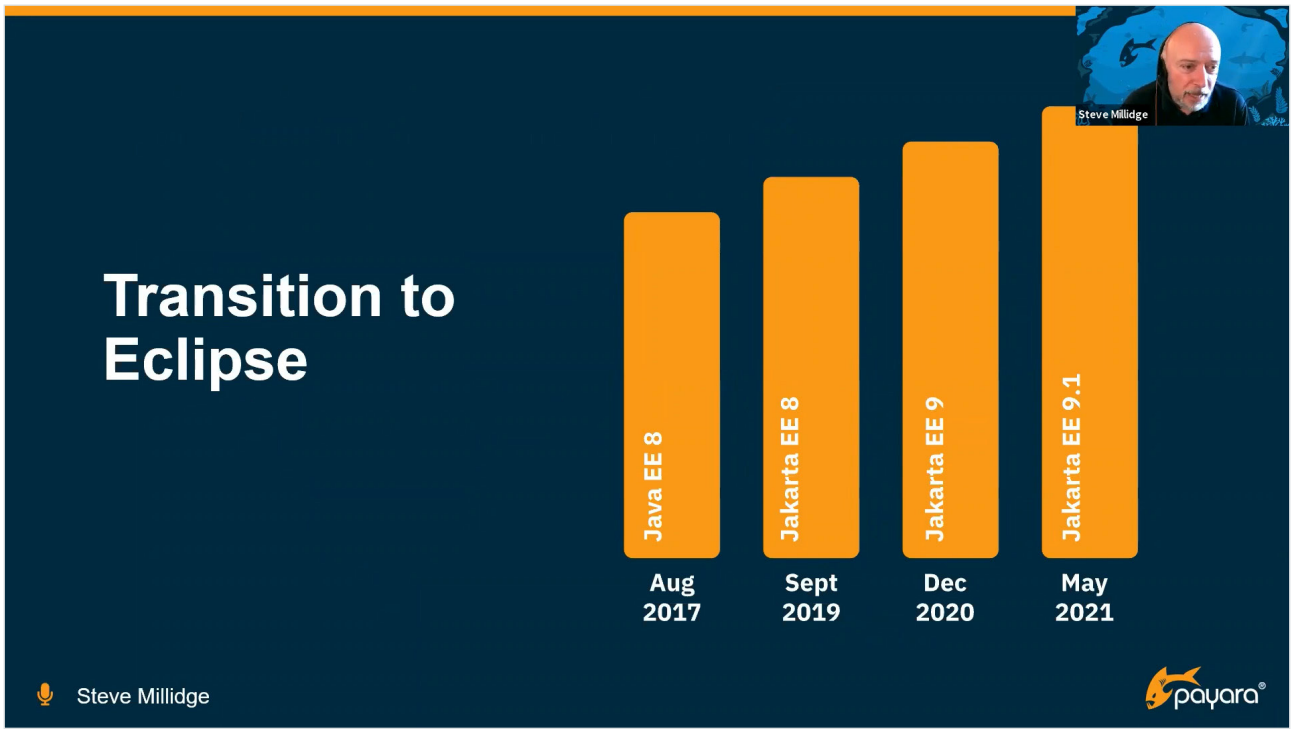
Benefits of Specifications

Why do we care about specifications?

We care because specifications allow for choice. Specifications opened the door for innovation and competition. There are many different technologies that now make use of Java EE/Jakarta EE, from lightweight runtimes to big enterprise application servers.

Having the specification offers longevity. Many specifications are focused on backwards compatibility. For example, it's possible to take an application built in Java EE 5 fifteen years ago and run it on a modern Payara Server on the cloud, an edge server, or a traditional machine - all because of the specification process.

Java EE Transitioned to the Eclipse Foundation



Java EE 8 came out in 2017 through the Java Community Process (JCP) led by Oracle. This is the last release by that body and the decision was made to move it to Eclipse Foundation, an open source foundation.

The Eclipse Foundation released Jakarta EE 8 in 2019. The process took a lot of time going through the legals, copyright, and all that business-y stuff, as well as transferring the code base to other servers and creating new release pipelines on the Eclipse Foundation's infrastructure.

Jakarta EE 8 is deliberately designed to be binary compatible to Java EE 8, so that anything compatible with Java EE 8 would also be compatible with Jakarta EE 8 - and would run Java EE 8 applications.

Between Jakarta EE 8 and Jakarta EE 9, there was a problem with trademarks and copyrights of, the *javax* namespace. The *javax* namespace for the APIs had to be changed to the *jakarta* namespace.

To give developers an opportunity to adopt the new namespace without also having to worry about making new features and compatibilities work, Jakarta EE 9 deliberately contains the same features as Jakarta EE 8. The only difference is that it was moved to the new *jakarta* namespace.

Any of the frameworks previously using stand-alone Java EE specifications, such as Spring and Dropwizard, would ultimately have to adopt the new namespace if they wanted to move forward if they planned to pick up any later versions of the specifications.

Jakarta EE 9.1 was released as a minor release in May 2021, which brought the test suite up to Java 11, which was the latest Long Term Support (LTS) version of Java at that time.

When you're looking at this timeline, it may look like nothing really happened in the enterprise Java EE space between 2017 and 2021. The myth that Java EE is outdated, and dead, may originate from this high-level view of what was going on with Java during this four-year time period – there were no new features released, so people assumed it must be outdated! But that's not true. What you don't see on the Jakarta EE timeline is the development of MicroProfile.

During this transition between Java EE and Jakarta EE, many of the vendors who build Jakarta EE (Payara, RedHat, Tomitribe, IBM, and others) came together again and discussed the inability to innovate on the specifications during this transition period. They wanted to address the changes going on in the industry, which included a move toward microservices architectures.

MicroProfile began during this Java EE 8 transition phase, and since then, MicroProfile has shipped many APIs specific to microservices use cases. It's based on Java EE API foundations including CDI, JSON-P, JSON-B, and it's built on top of Open Tracing, Open API and Config.

The enterprise Java industry has not been on a standstill since Java EE 8, which is one of the accusations people throw around to support their belief that Java EE is outdated – the reality is, MicroProfile and several new APIs were created during this timeframe to address the growth of microservices architecture.

Many traditional Java EE application servers support MicroProfile APIs, including:

- Payara Server and Payara Micro
- Apache TomEE
- OpenLiberty
- Quarkus
- Thorntail
- WildFly
- kumuluzEE

The application servers support the use of MicroProfile APIs on top of Java EE.

Payara and other application servers supporting MicroProfile APIs have been building new capabilities for developers to build new applications.

MicroProfile also allows for innovation in runtimes. It allowed for things like Quarkus, Helidon, and kumuluzEE to use a slightly different model for developing applications. So, these same MicroProfile APIs can be used in traditional application servers as well as other types of runtimes. (See also "Beginners Overview Guide to Java Runtimes")

Why Did Java EE Move to the Eclipse Foundation?

Developers are rarely interested in the governance of platforms, but from an enterprise and organizational perspective, moving Java EE to the Eclipse Foundation and making it open source as Jakarta EE, offers many advantages.

The APIs, the TCKs, and the specifications are open source, which means anyone can build an implementation without having to pay money to other people or organizations.

The competition among vendors building more implementations should further drive innovation.

The Eclipse Foundation is a not-for-profit organization that offers an open governance of Jakarta EE, which allows us to all sit together at a table with rules and collaborate with the other industry players as a group.

Because it's a specification and a standard, you get patent protection if you create an implementation. There's a very simple license for it because it's open source. The Eclipse Foundation is an independent referee between the different players allowing for independent stewardship. The model allows the creation of multiple, independent implementations.

As an end user of Jakarta EE, building an application, you don't want massive innovation on the API level because that means you have to change your application every time the API changes. But you do want solid backwards compatibility.

So, rather than innovate on APIs, vendors like Payara can compete on how the implementation works, and that offers you more choice when it comes to the different models of implementation and where your applications run. This is probably the biggest advantage of moving Java EE (or Jakarta EE, as it is now) to open source with the Eclipse Foundation.

On the flip side, backwards compatibility can make innovation and growth of APIs move a bit slower, as can the collaboration process. Collaborating with many vendors will always take longer than having a benevolent dictator leading a framework. But the other difference is, open governance offers longevity. If one vendor quits, you still have all the others contributing to the growth of Jakarta EE.

Moving Jakarta EE to the Eclipse Foundation is a huge industry achievement that has taken two to three years. It now serves as a foundation for the whole industry.

If you look at all of the compatible implementations of Jakarta EE 8 you can see there are 15 different implementations of the full platform created by 12 different companies. This is proof that Jakarta EE is definitely not a dead industry body.

There's a lot of work going on – 15 implementations and 12 companies! Ranging from Eclipse GlassFish (predominantly maintained as an implementation that passes the TCK to prove that we can release Jakarta EE), but also big industry players like RedHat, Oracle, and IBM, and important players like Payara, and regional companies that serve their regional markets as well.

All of these vendors have taken the open source specifications, built a product, and certified that their implementation works and meets the platform requirements and passes the TCK. As a developer you should be able to take your application, and if you don't like your current implementation, move it to any of the other implementations since they all are based on the same specifications.

What's Next for Jakarta EE?

Now that all of the work is done to move Jakarta EE to Eclipse, the next release is Jakarta EE 10 and that's planned in the first quarter of 2022. A lot of the foundational specifications that have moved from the javax to the jakarta namespace are now going to be improved. You can look at the various projects at Jakarta.ee and get involved. They provide a project release review to define what they want to improve in the future.

For example, Steve Millidge from Payara is heavily involved in Concurrency. You could go on the Github repo to see the work being done in this area and contribute to the improvements and growth. That's the big difference between Jakarta EE and the previous Java specifications.

Also in the Jakarta EE 10 release timeframe, a Core Profile is likely to happen. JDK 17 is the LTS version of Java as of September 2021. When Jakarta EE 10 releases it should run on JDK 17.

Core Profile

A Profile, or Platform, is a set of specifications.

Out of all the Jakarta EE specifications, an implementation might choose to package certain specifications together. The runtime can say it supports that profile if it has all of those APIs and passes the TCK for that profile or platform.

Full Profile

The Full Profile means every Jakarta EE specification, except for any specifications in incubation at the moment.

Web Profile

The Web Profile cuts out some of the more enterprise-integration type specifications, like connector architecture or JMS.

Development of the Core Profile

The Core Profile is very targeted at microservices applications. The idea is the Core Profile will potentially also support native compilation.

This will probably be very similar to MicroProfile and have a CDI called CDI-lite, updated JAX-RS, and JSON specifications - and possibly some others from the Jakarta EE family.

Myth: Java EE is Dead and Outdated

Dismiss the Myth: Jakarta EE is an Evolving Foundational Technology for Many Enterprise Frameworks

Java EE has now become Jakarta EE. So, yes, Java EE is effectively frozen at Java EE 8 – it's still supported but there won't be any new developments of Java EE – because it has moved to the new namespace and name: **Jakarta EE**.

Jakarta EE will be a foundational technology for a lot of the enterprise frameworks out there, even if you don't think you use Jakarta EE application servers, it will still affect you as a developer. **It's moving and evolving, and you still need to know about Jakarta EE**. It's definitely not dead.

Jakarta EE is alive and thriving. You can get involved, it's moving forward, and it will be the foundation of enterprise Java for the next 10 years.

Get Involved: https://jakarta.ee/

JAKARTA EE

How Will You Use the Payara Platform?

Migrating to from Another Application Server

Looking to migrate mission critical or production apps from a different application server (such as GlassFish, Jboss or WebLogic)?

Request Payara Enterprise

Planning to Run Payara Platform in Production

Looking for an reliable and modern application server for mission critical, production environments?

Request Payara Enterprise

Try Payara Platform in Development

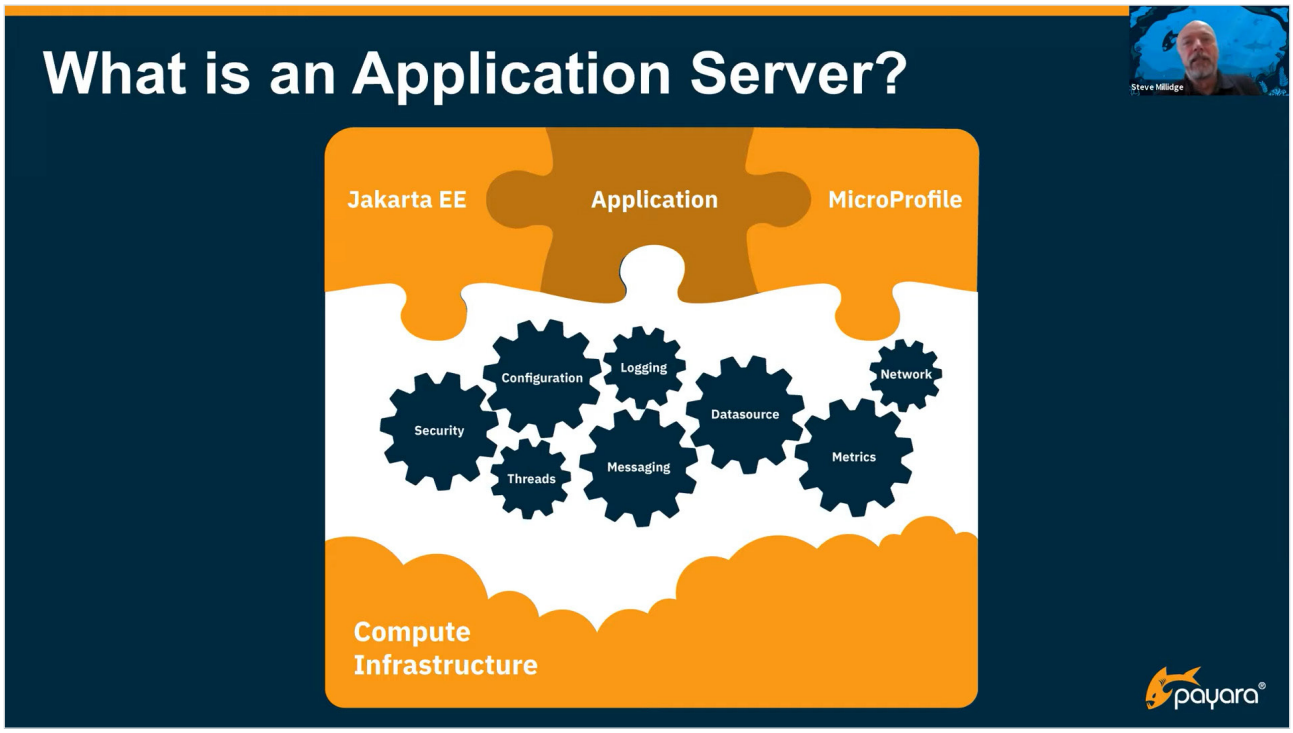
No plans of running Payara Platform in production and just looking to try it out, or test new features?

Download Community Edition

MYTH: Java EE Application Servers Are Heavy

People who have not used Java EE in many years or who don't know much about Java EE often say "Java EE application servers are resource hogs, and slow to start up." But it is possible to have a small footprint and low resource usage with Jakarta EE runtimes.

What is an Application Server?



Before we can tackle the myth, we should define what an application server is and the different application server deployment models. The goal of an application server is to provide a managed runtime for your application.

In the image above, your application is shown in the jigsaw puzzle piece at the top and center. And the application server provides a bunch of services for you, so you don't have to write the code to do them yourself, things like:

- how to handle thread management in the server
- how to do http processing
- how to handle the network
- database connectivity
- generate metrics and monitoring diagnostics
- logging
- complete security infrastructure

You need all these things if you are going to build an enterprise application.

From a Jakarta EE point of view, an application server also provides industry standard APIs and allow your applications to use the services. Jakarta EE supplies the Java Persistence API, for example, which allows you to write and read data from a database, which then uses underlying datasource implementations within the application server and the Java EE API to map the data into rows and use the datasource to connect to the database.

There are other things that go beyond the single instance of an application server that a managed runtime or application server provide, such as high availability, clustering capabilities, and the ability to store data in one node and retrieve it from another node. Many different enterprise functionalities are provided by the managed runtime. It is a bit like an operating system, but for server side applications.

That sits on top of some type of compute infrastructure. In the past few years, the compute infrastructure has become much more diverse. We've moved away from bare metal servers to virtual machines to containers to container orchestration in Kubernetes. Or Cloud, or container runtimes. All of these options are considered compute infrastructure to the application server, and it's the role of the application server to hide the complexities of the compute infrastructure from the developer so the developer can focus on writing the application, use the Jakarta EE or MicroProfile APIs and get all the benefits of the functionality provided for you within the APIs.

Using this as a definition of an application server, many other runtimes that would claim that Java EE application servers are 'heavy and slow' would fit that same definition.

Traditional Application Server

You take your application, and you go through a step where you deploy it to the application server. That's how you do this with Payara Server on a domain, you go through the Admin Console, upload your application, and click deploy. Similar to WebLogic or JBoss, or any other Jakarta EE application servers.

Fat Jar

Another model that is evolving. In this model, you take the application and package it tightly with your managed runtime, and then you can run it through java – jar. That will run the application and bootstrap the runtime at the same time.

It is remarkably similar to a traditional application server except that this is a different packaging system. You're combining your application with your runtime and creating a single artifact. Payara Micro uses this model of application deployment.

In the Fat Jar model, some runtimes allow you to pick and choose the components and self-assemble your runtime. In a Jakarta EE model with Payara, your runtime will be versioned and be a specific version of the Payara Platform. You don't choose your security components or datasource implementation, or which metrics provider or networking library. Some runtimes give you that freedom of choice, but it requires a level of self assembly.

Spring Boot is built on top of the Fat Jar model and makes opinionated decisions about what those components should be.

From a runtime perspective, some people favor the Fat Jar because you have a single artifact, and others prefer a Thin War where you deploy a small application on top of a managed runtime.

Thin Wars and Container Images

One advantage of the deployment process of separating the runtime layer from your application is when you start to look at containers and how you build container images so they can be executed with a container orchestrator (like Kubernetes). There are advantages to separating out the application layer from the application server or managed runtime or framework.

Typically, in containers there are several layers. The base layer, which is the operating system and how you've configured the operating system.

Then there is a runtime layer. For example, if you're using Payara Micro you can layer a specific version of Payara Micro on top of your base system layer.

And then you layer on your application code as a single, separate layer.

The operating system and runtime layers are stable and can be quite large. When you separate the application from the runtime and operating system layers, you gain the advantage of being able to push small application changes to a thin layer, rather than having to push the whole artifact over and over again with every small change. This is especially beneficial during development or pre-production stages when your application may go through a lot of changes. And since container images are built in layers, the build time is much faster if you're just pushing slight changes to a thin layer than if you were pushing the whole thing, including the operating system and runtime, with each change.

Advantages of a Managed Runtime

There is no self-assembly needed with a managed runtime like Payara Server or Payara Micro. Managed runtimes come with a versioned set of everything that works together, is certified, passes the Jakarta EE TCKs, and is a compatible implementation of Jakarta EE. This means you don't have to assemble a runtime from various parts.

It's also 'versioned', and from a security patching perspective, if you have different layers and you have a problem with something contained within Payara Micro or Payara Server, such as a security alert on some component, then you know you can upgrade just that layer to the latest version of Payara and solve that security issue.

This is a much trickier fix if you are packaging a Fat Jar yourself from different components – especially when they've gone into production. It's possible that the original development team that packaged the Fat Jar has moved on by then.

Because the Payara Platform is a managed runtime, there are a lot of other capabilities we can provide already baked into the server, like monitoring, health checks, high availability, and scalability. It allows the developer to focus on building the application and not the infrastructure.

Build Applications – Not Infrastructure

One of the great advantages of an application server is that it separates your application from the runtime. This means the runtime can hide a lot of complexity. For example, if you were going to package a WAR file within a Fat Jar and deploy it to Kubernetes, then you must worry about creating the Fat Jar, creating the Docker files, building container images, writing YAML file, creating your Pods, and controlling that through the Kubernetes API. That's a lot to worry about!

But when you move to something like Payara Server or Payara Cloud, there are many things that the runtime can do to isolate you from all that complexity. For example, in Payara Server, if you run it on a cloud provider like Azure or AWS using VM instances, it will automatically cluster without any special configuration.

We can provide things like helm charts to allow you to build a Payara Domain in Kubernetes. Or a Kubernetes Operator which can control your application. We can do a lot of that work for you within an application server, so you don't have to worry about it as a developer. You can worry about your actual job – building applications – and leave the infrastructure up to your application server/runtime.

What is Considered Heavy?

Java EE often gets accused of being 'heavy'. But what does that mean? There are four things that people could mean when they refer to Java EE as 'heavy':

1. **Memory Usage** - people claim Java EE or the Java EE application servers use lots of memory. That isn't necessarily true as we'll show you a bit later in this ebook. Application working sets, or the amount of memory your application is going to use, is going to typically dwarf the memory used by the server itself. Your working set is going to depend on how much data you need to hit in a single transaction or request, and that will typically be large compared to the application server.

2. **CPU Usage** - there is no reason why a runtime is going to add a lot of CPU over and above your raw algorithmic performance. If you build a 'hello world' application that uses all the layers of Java EE, you will typically get that running with a request processing time of 10-20 milliseconds. The CPU usage is not typically large for a Java EE application server.

3. **Deployment Size** - some people say deployment size is heavy. But because we separate the application server from the application, deployment sizes for Jakarta EE are quite small. A reasonable sized application, depending how many third-party dependencies you package and if you use raw Java EE APIs, for example, a 'hello world' rest application is only a few kilobytes in size. And that's not heavy.
For other runtimes where you package your runtime along with your application, the deployment size is typically larger than a Jakarta EE deployment.

4. **Installation Size** - people also talk about application servers having gigabytes of installation. If you look at Payara Micro as an example, this is just not true anymore. Payara Micro is about 80mb.

How Does Jakarta EE Compare?

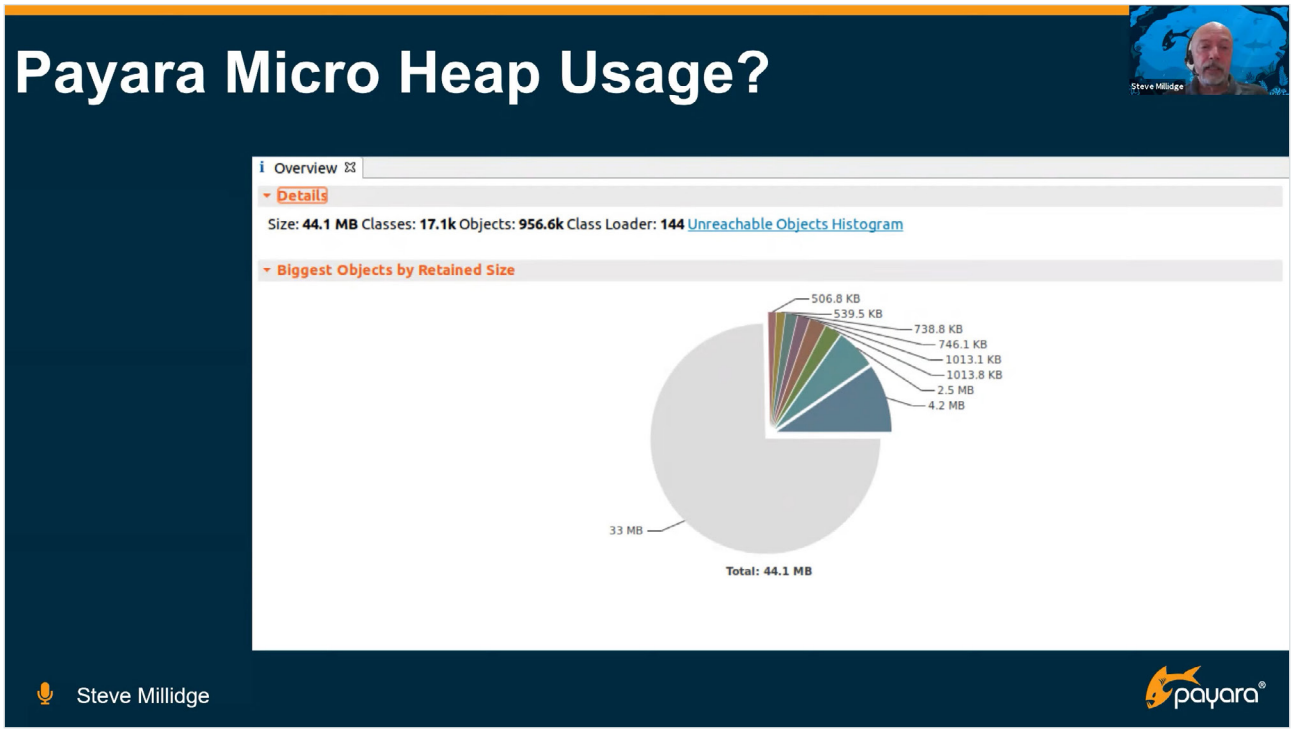
Steve Millidge took Pet Store, a Java EE standard application, and Pet Clinic, a Spring standard application, and built each of them to compare them. We're not trying to compare Jakarta EE versus Spring, because both are very small.

But if you look at Pet Clinic, the file size is between 50 and 80mb depending on what you choose to include in your runtime. If you add things like rest and web sockets you will be closer to the 80mb size. When you boot it up, it boots in 2 to 9 seconds. if you look at the heap usage, it shows around 30 mb. In that Spring application, if you hit a REST endpoint it will do around 2000 - 3000 requests per second. So, it's both fast and small.

If you take Payara Micro 5 and the Pet Store application which has REST endpoints and a JSF application, the application itself is around 8mb and Payara Micro is around 80 mb. So, it's a bit bigger than Springboot but certainly not considered huge. Its heap usage is between 17 and 45 mb when you boot it up and it boots up in 2 to 10 seconds.

If you were to run either Pet Clinic or Pet Store through a performance tool, the web sessions start gathering and using the majority of the heap. So, it's the application usage rather than the runtime that adds 'heaviness'.

Both Jakarta EE and Spring are incredibly small infrastructures which are very comparable. The runtimes use less resources than your browser does running JavaScript.



The above image shows a Payara Micro heap dump after starting the Pet Store application. This has gone through the Eclipse Memory Analyzer tool kit.

What Does it Mean When People Say Java EE is Slow?

“Slow” can mean many different things. When people promote the myth of Java EE being slow, they probably are referring to response times, performance problems, boot time, build time, or deployment time.

- **Response time** - most of the response time in real applications is used by the developer. Spring Boot and Payara Micro achieved 2000 - 3000 requests per second.
- **Performance problems** - if you have a performance problem, you must look at whether you have an algorithmic performance problem with a single slow request, or is it a scalability issue where the single request is fast but concurrent requests are slow? Application servers provide horizontal scalability.
- **Boot time** - is it that the boot time is slow? With Payara Micro, an application will boot between 2 and 10 seconds depending on your application, and that’s similar to other application runtimes. The more you add, the slower it is. There are ways to boot it a bit faster, such as with GraalVM and native compilation, but the question to ask yourself is if that really matters enough to rewrite your application? To move from a boot time of a few seconds to a few hundred milliseconds? The answer to that question is probably not.
- **Build time** - building in a Thin War is much faster than building a full container with all the different layers from scratch. Build time in Jakarta EE is very quick.
- **Deployment time** - how fast will it deploy an application? And again, this may take a few seconds up to tens of seconds depending on what you put in your application. The more you can use with the Java APIs the faster your deployment will be.

Most Java EE application servers will offer horizontal scalability right out of the box, giving you the ability to add multiple instances. Payara Server and Payara Micro provide automatic clustering for horizontal scalability without you having to do any work, whether that’s scaling on multiple Pods or Pod scaler in Kubernetes - we provide that capability to cluster on a Kubernetes structure - or whether it’s running another Java process on a physical machine. Because we have a managed runtime, we can provide that functionality for you.

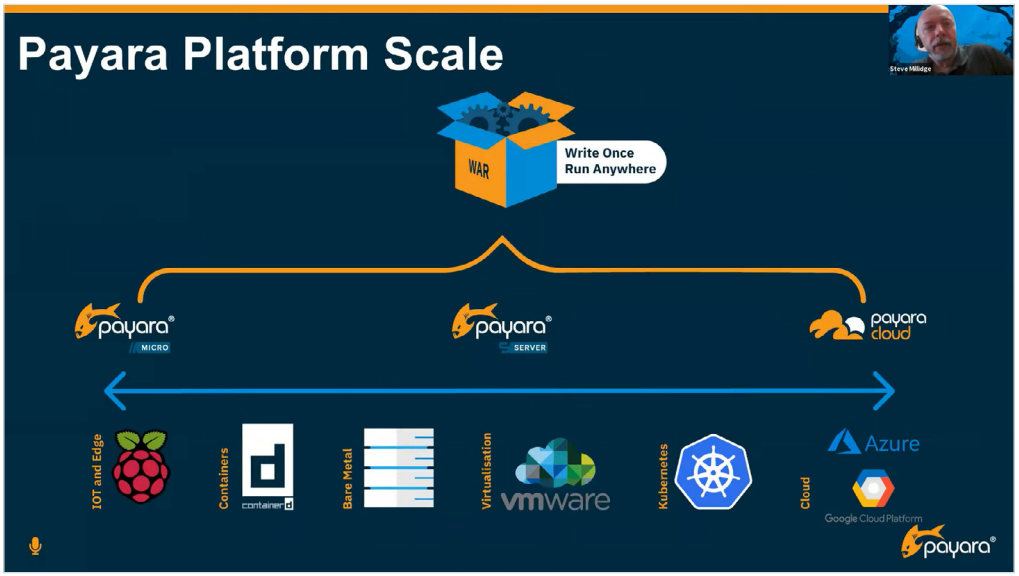
Myth: Java EE is Heavy

Dismiss the Myth: Java EE/ Jakarta EE is NOT Heavy, the Overhead Comes from Your Application

Java EE is NOT heavy. It's comparable to others. When you look at all the runtimes or application servers, regardless of which one you choose, a lot of the overhead comes from your application itself – not the runtime. Whether that's memory, CPU usage, disk space, or whatever else - often it is the application that will be the dominant factor.

The whole point of Java EE/Jakarta EE is to separate the application from your runtime, and then in the Payara Platform, our goal is to set it up for the developer so you can just build your application.

Payara Micro covers the Jakarta EE Web Profile APIs and Batch, Concurrency specifications and JMS client. It can run on very small Edge servers or Raspberry Pis, so it's a very small application server. Then we have Payara Server, which is a traditional domain mode with an Admin where you manage many server instances. This is ideal for bare metal servers through to Kubernetes. And then in 2022, Payara Cloud launches! Payara Cloud will run on top of Kubernetes but also will run on multiple Kubernetes clusters.



Myth: Java EE is Not Cloud-Native

Another common misconception about Java EE is that it is not a cloud-native technology. Before we can discuss whether it is cloud-native, we need to decide on a definition for cloud-native, as it's a term that's used in several different contexts.

What is Cloud-Native?

We'll look at three definitions of cloud-native:

> *InfoWorld - "In general usage, "cloud-native" is an approach to building and running applications that exploits the advantages of the cloud computing delivery model."*

The obvious question that comes from reading this definition of cloud-native is then 'what is cloud computing?' From our perspective, one of the unique aspects of cloud computing is the infrastructure on demand. That means infrastructure and compute infrastructure is elastic, meaning it can expand and contract rapidly, which is different from traditional on premise data centers where you

may have to raise a request or a ticket to gain access to compute, or VM, and wait for someone to provision that for you. With cloud, you can do that in seconds.

The other big difference between cloud computing and traditional on-premise environments is that a lot of the infrastructure is software-defined. You have APIs where you can set and configure all your compute network and storage infrastructure. And you can do this from continuous integration, continuous development, or from scripts and it can all be software-defined.

One of the other big areas of cloud, which is probably not a positive, but it is a reality, is that cloud platforms like Azure, AWS, Google Cloud, all come with a whole host of proprietary APIs. There aren't a lot of standards around cloud providers. Each API is different, so Amazon's API for configuring a load balancer will be completely different from Azure. So once you start working with a cloud provider, you begin to get tied into that provider and it is difficult to move to a different one.

When we talk about cloud-native in this context, it means exploiting the elastic software-defined infrastructure, and then running Java EE applications on that infrastructure.

> ***Cloud Native Computing Foundation - "Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach."***

They offer a similar definition to InfoWorld for cloud-native, but when they go on to talk about containers, service meshes, microservices, immutable infrastructure, and declarative APIs takes the definition a step too far. These are the aspects of the industry that the Cloud Native Computing Foundation focuses on, creating service mesh technologies and microservices, but it's not fundamental to cloud computing. You don't have to run a service mesh to run on public cloud or build elastic infrastructure.

> ***RedHat - "Cloud-native applications are a collection of small, independent, and loosely coupled services. They are designed to deliver well-recognized business value, like the ability to rapidly incorporate user feedback for continuous improvement.***

Steve Millidge disagrees with this definition for cloud native. This definition effectively links cloud native with microservices, and he feels that goes too far. You can build applications that run on cloud infrastructure that are not microservices. Microservice architecture can also be built on top of on-premise datacenters, so there is no need to link the two together.

There is a lot of confusion around the definition of cloud-native, and it seems the industry is linking four concepts together into the terminology:

- Microservices
- 12 Factor App
- Containers
- Public Cloud

While these are good practices on cloud, they are not necessarily a requirement for a cloud native application.

For our purposes, we define cloud-native by the things which are unique about public cloud:

- Elastic infrastructure so you can scale out and down rapidly
- Ability to get infrastructure on demand
- Ability to stand up a new machine for testing
- Proprietary APIs to manage infrastructure

When you focus on this as the definition of cloud-native, we can talk about deployment models for how you can use cloud infrastructure to deploy Jakarta EE applications. There are a lot of options for deploying Java EE/Jakarta EE applications on public cloud.

Application Server Models

There are two different models for deploying Jakarta EE applications: traditional application server or Fat Jar.

Traditional Application Server

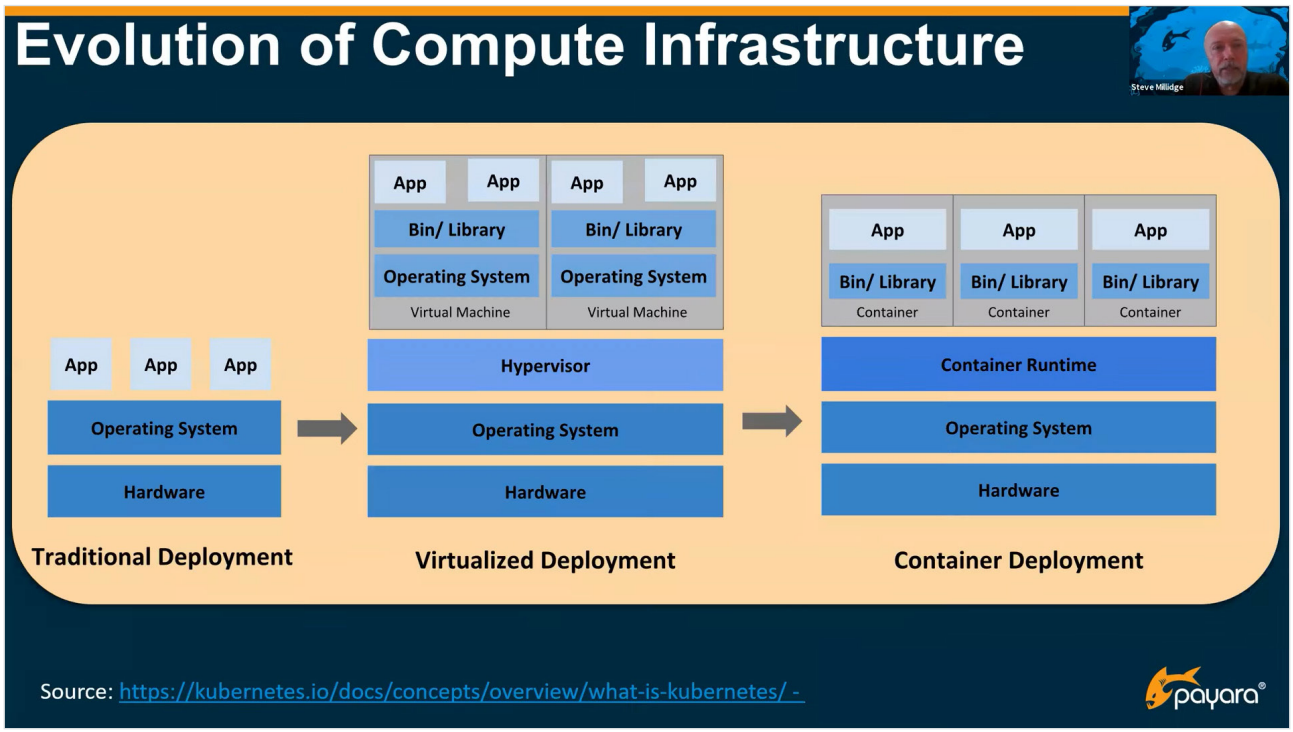
As a reminder, we've said that the traditional application server separates the application from the infrastructure, and you deploy the application into your application server. The application server takes care of many of the services that you need, like networking, security, database access, threading.

Fat Jar

The Fat Jar model packages your application with your application server into a single Jar file that can be run on the command line with a java -jar.

Payara Platform supports both models. Traditional deployment would be Payara Server with a traditional domain. And Payara Micro typically uses the Fat Jar deployment model.

Evolution of Compute Infrastructure



Kubernetes.io offers the above image which sums up the evolution of compute infrastructure and what has happened for cloud, where things are clashing and meeting in the middle to give us *cloud native*.

The traditional deployment model consists of your hardware and operating systems, and you can deploy multiple applications on your operating system.

Then virtualization came around, and the deployment model started to change. On top of the operating system, we got a hypervisor that allowed us to wrap full computer images into a virtual machine. The applications are deployed on top of the operating system, deployed on the virtual machine, on top of hypervisor, on top of the physical server running an operating system. This is what created the birth of cloud computing as opposed to the use of dedicated servers and hosting providers.

Virtual machines allow you to create multiple things and share the compute power of the underlying hardware. This resulted in the birth of public cloud providers, like Amazon EC2, which were effectively virtual machines running on Amazon's hardware.

Then we move to containers. Docker came around and we evolved to a container deployment model. The container runtime sits on top of the operating system, but it is not the same as a hypervisor. A hypervisor emulates an entire computer while a container runtime uses the logical file systems and creates a copy of the operating system rather than the whole machine. You package your application

into a container, which locks the configuration of your application and your file system layer and libraries that you're using and puts it on top of a logical instance of the operating system in the container.

Containers move us into where we're going now with containers and container instance services and Kubernetes. They are all running on the container level.

How do we take different application server deployment models, like traditional Payara domain or a Payara Micro, and map that to the cloud native infrastructure which is virtualization and container deployment? This is part of the application server's job.

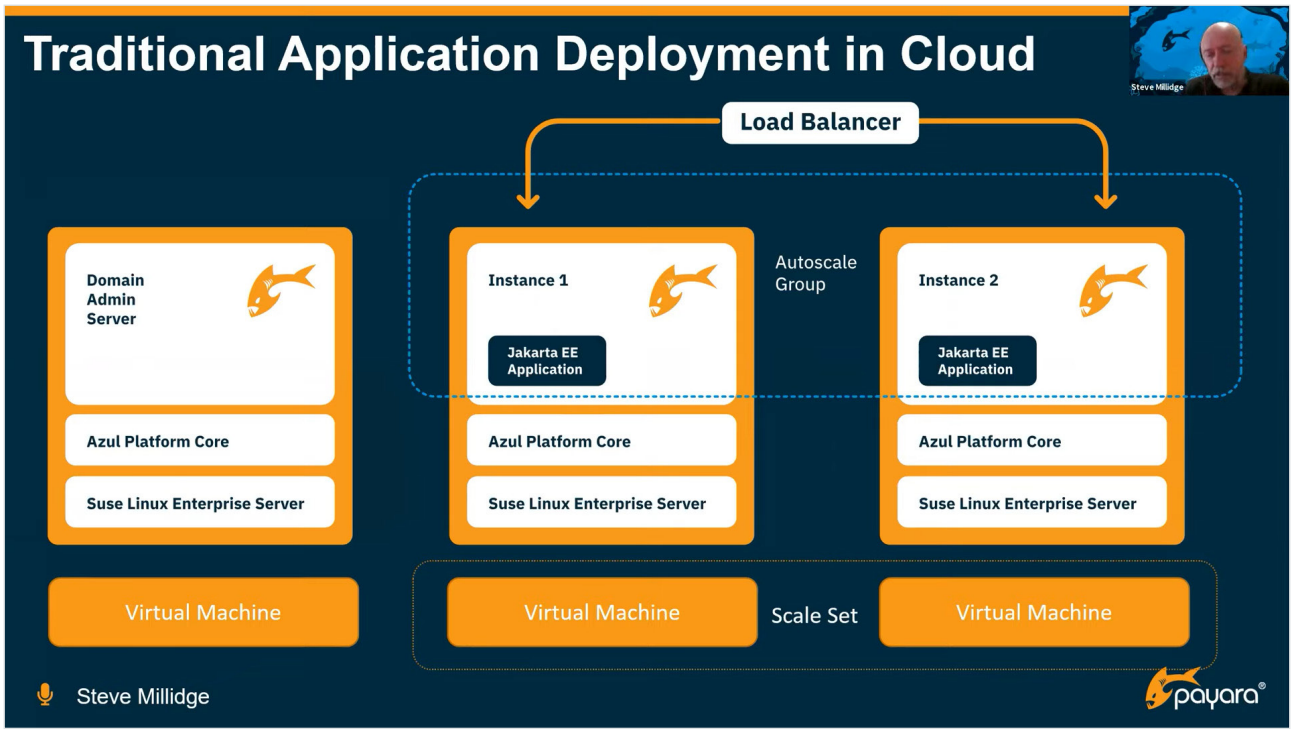
Traditional Application Deployment in Cloud

How would you deploy a typical Payara domain onto a public cloud like Amazon or Azure? Payara has a lot of documentation around this. Cloud-native is really a function in which the application server or platform you use to run your Jakarta EE applications supports these modern, containerized deployments.

Virtual Machines

For Azure virtual machines, you have Azure VMs, and for AWS you have EC2.

Payara Platform 5 has specific support for Azure VMs and Amazon's EC2, and the Payara domain is built specifically for using them.



In the above image, we see a Load Balancer. (On Amazon, that would be an elastic load balancer and on Azure it would be a standard load balancer.)

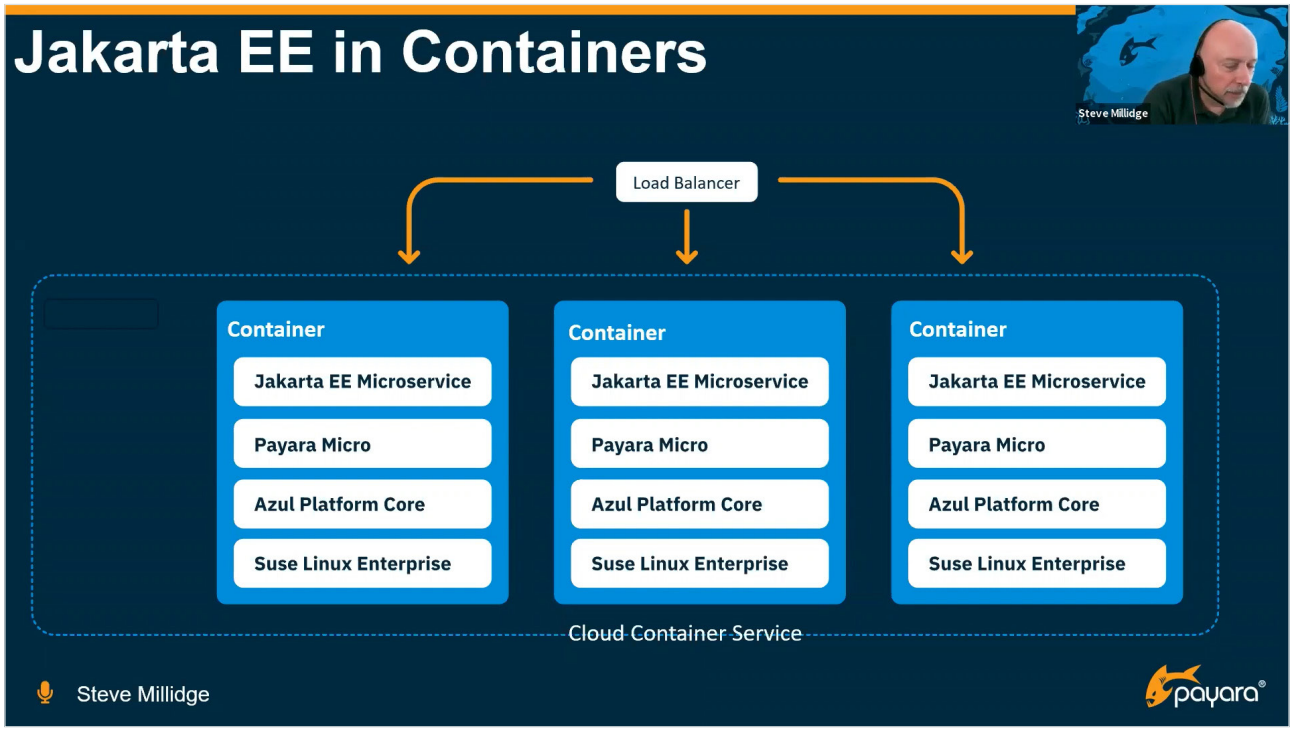
Most cloud providers offer scalability by building on the elasticity of the cloud infrastructure. On Azure, it's called a scale set, and on Amazon it's called an auto-scaling group. This allows you to create a virtual machine image and scale it out - so you can choose 4 or 5 or 6. These are typically linked into the load balancer and cloud computing so it will load balance depending on how many virtual machines you have.

Payara Platform 5 has done a lot of work to make it cloud native for this sort of architecture. We recently created a new feature called **Auto Scale Group**, which will eventually allow you to call the public cloud APIs to scale out virtual machines for you. But right now, out of the box, if you use Payara and create this architecture it will automatically cluster your application instances and they will do that on different public cloud platforms.

It will also enable you to create VM images with instances on, which we call Dynamic Instances, and that allows you to add and domain instances on the fly from the domain. We also have a Deployment Group and if you add an instance into the Deployment Group, then the applications that you configured and the configuration you set will automatically be propagated.

Out-of-the-box, Payara domain supports traditional application deployment. You don't have to have microservices to work in a public environment and gain the benefits of elasticity and auto clustering and scalability. Payara Server in a traditional domain can support that now. So, you can take a traditional application and run it on Payara Server and it would be cloud-native.

Containers



We have solutions for containers running natively in public cloud. The two main container services, Amazon Elastic Container Service, which allows you to create a container and spin it up without worrying too much about VMs and the compute infrastructure. Amazon provides it. And a similar feature is available on Azure; it's called Container Instances.

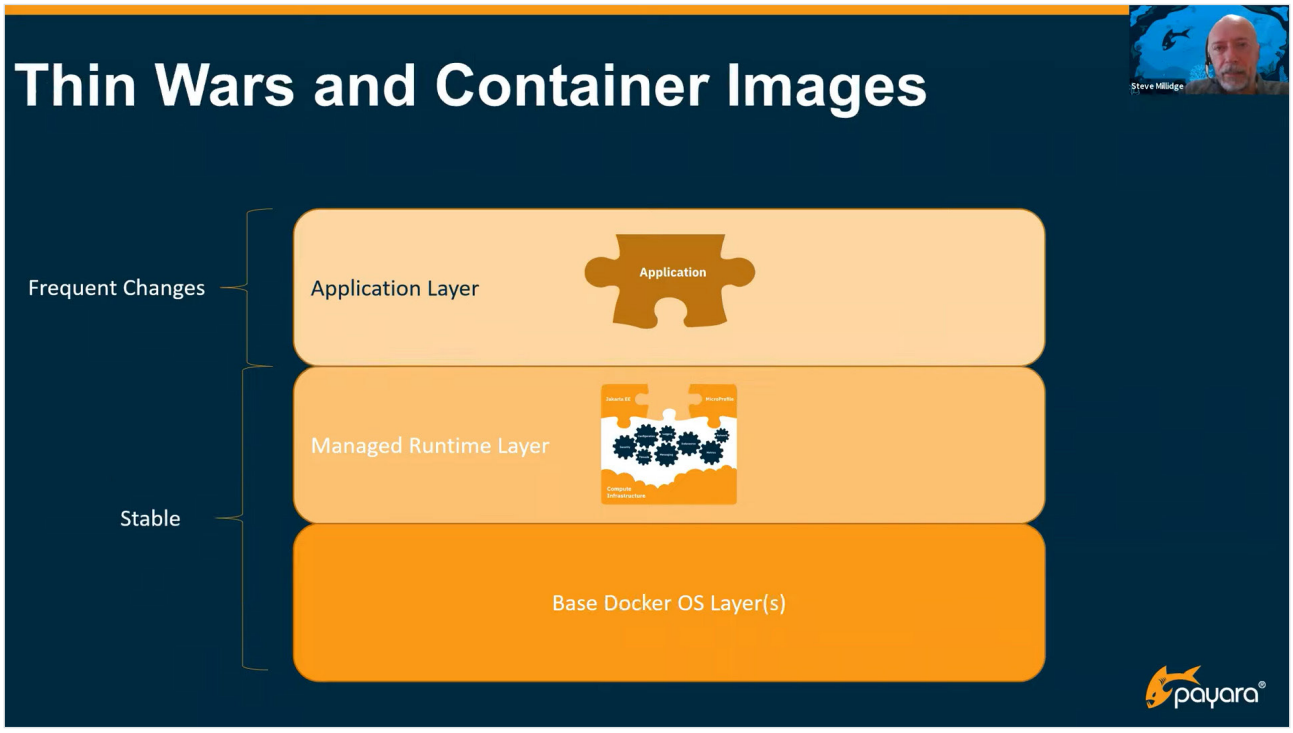
Payara Platform fully supports both. So, you create your container image using Docker or something like that, and then, in the above example, you package your Linux, Azul Platform Core (which is included with a Payara Enterprise subscription), and then you can put your microservice on top of it or any web profile application over it. You could also use Payara Server, and that would allow you to run any Jakarta EE compatible application.

So, all of that is packaged up and the package is sent to the container service, and it will fire up a container and run it for you without you worrying about server management or host management or virtual machines.

Then, on top of all of that you put a cloud load balancer, which is a bit more complex to load balance across multiple containers, but typically the cloud provider will allow you to use multiple containers for the same application and load balance across them. Again, Payara Platform has specific functionality within the infrastructure of Payara Server or Payara Micro for clustering and session replication across this sort of architecture.

So, you can also take advantage of container services within public cloud using the Payara Platform.

Jakarta EE Application Deployment Model



Jakarta EE has advantages when you are building this architecture. If you are building container images, you have three layers.

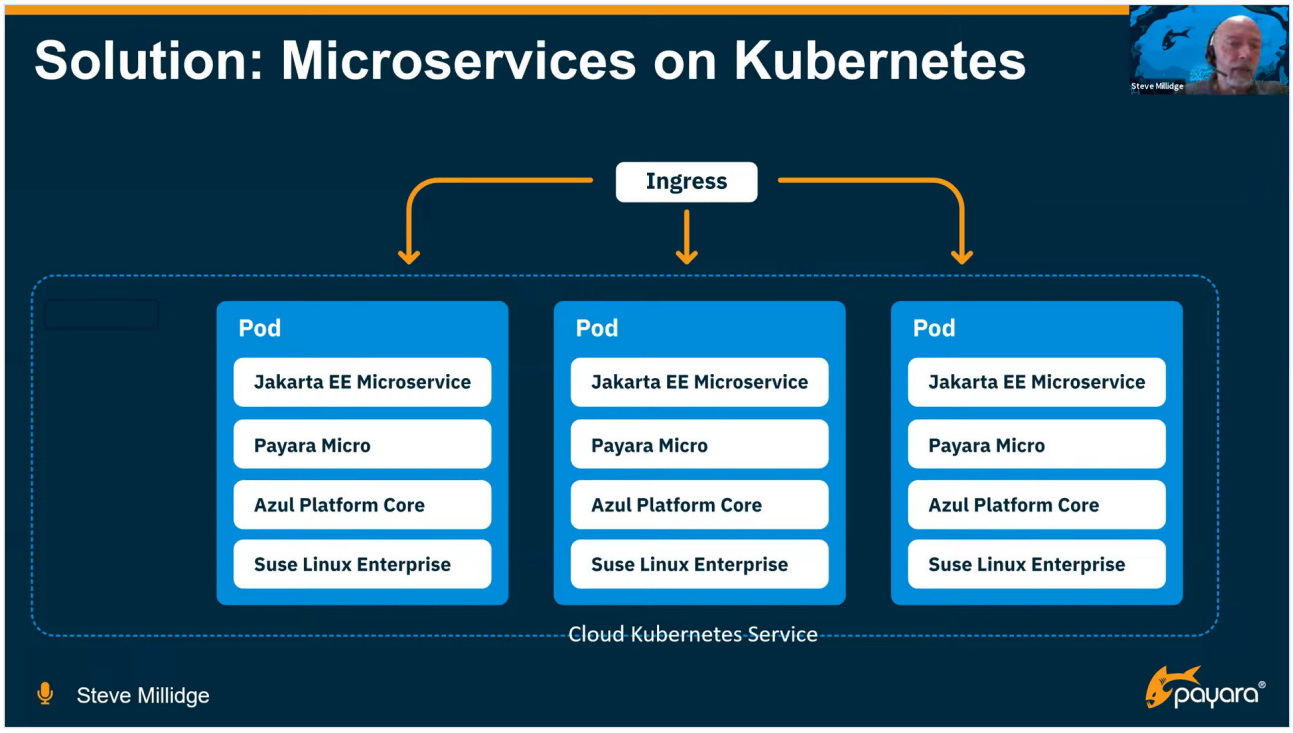
Container images are built from layers, and they are controlled by configuration.

- The bottom layer is your base operating system.
- Then you have the application server or managed runtime layer.
- And then you have your application layer.

Container images are typically quite large because they package an entire operating system image and drop it into a single file. That file can be megabytes to gigabytes in size.

The layers are there so you only have to change small pieces of the file system to make updating the top layer easier. From a Jakarta EE perspective, the application layer changes often, but it's separated from the application server tier. This allows you to build containers quickly and update small changes and push them to the cloud rapidly. Typically, the application layer changes are small.

Microservices on Kubernetes



The next level up from containers on cloud is Kubernetes service. So, the next logical question is what is Kubernetes? Kubernetes is a platform which is 'standard' in that most cloud providers support it, and it allows you to manage containers at scale. It's not like an app development solution, it is an infrastructure technology that allows you to manage containers at scale. It supplies services for load balancing containers, service discovery so you can create what's called 'a service' (which are multiple pods), and then you can scale it out.

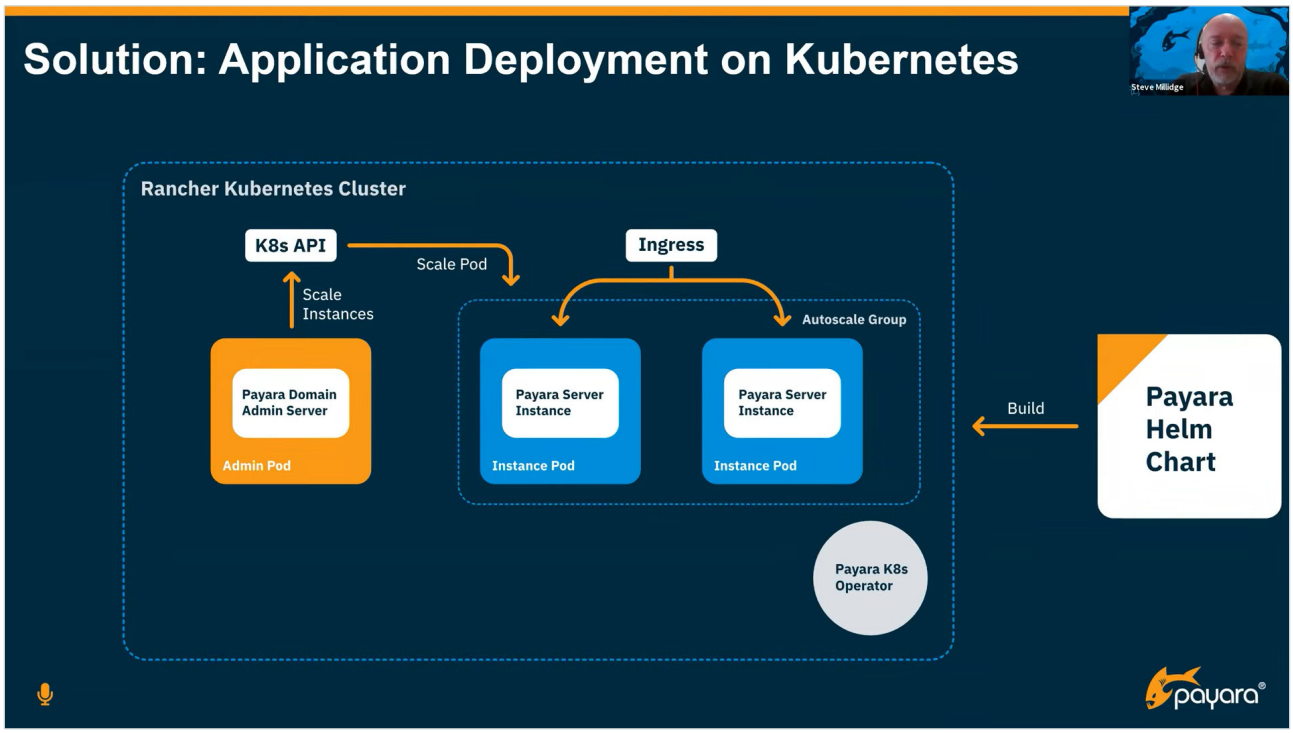
Kubernetes manages storage and lets you attach and remove storage. It can do roll outs and roll backs of different configurations of your containers and pods. It can also do bin packing, which means it can work out which physical compute is the best one to run your containers on. It can do some Secrets and security management as well.

Running a Jakarta EE application on Kubernetes in this model is very similar to running on an Azure Container Instance. The difference is Kubernetes is a standard in that it is supported by Azure, AWS, and Google Cloud Provider. If you build your infrastructure on Kubernetes then it is likely to be a bit more portable between the different cloud providers. If you build it on Azure Container Instances, then it will be set up in a completely proprietary way and difficult to move it later.

In this model, we build a container image and include our Jakarta EE application, Payara Micro, JVM, and our operating system. Then we create the Pod definition and load that in through Kubernetes through some cryptic YAML and then we can scale the pod up and down using Pod auto-scalers.

Payara Platform has this functionality built in! Payara Platform can cluster Kubernetes using the service names and you can build container images using Payara Micro or Payara Server, and scale them on a Kubernetes platform.

Application Deployment on Kubernetes



In another traditional architecture, we can build capabilities that work similarly to a domain. You don't need to build microservices, you can build a traditional server domain into Kubernetes.

We have something called a **Payara Kubernetes Operator**. The operator builds the infrastructure. If you drop that into Kubernetes, it will create the instances and Pods that you need to build a traditional Payara Server domain on top of Kubernetes. If you're using a cloud Kubernetes service, you can take a normal Payara domain and build it out onto Kubernetes.

As mentioned previously, we are building more functionality in the Payara Platform that will allow scaling out and down from the Admin Server, but you can currently do that with pod scalers and dynamic instances. You will be able to build clustered Jakarta EE application servers on top of Kubernetes.

Later, we'll produce Helm Charts which will allow you to install all of this from scratch.

Java EE is Definitely Cloud-Native

So, this brings us back to our question, is Java EE cloud-native?

From a cloud-native perspective, Java EE APIs don't really matter. If you have a Java EE application, what really makes it cloud-native is what you choose as a platform to take your WAR or EAR file and deploy it. You need an application server to manage the infrastructure for you, and then you have a Java EE cloud-native application because it's going to run natively on cloud infrastructure and you can scale it out, down, and use the elasticity of cloud. You can build out and deploy the application natively into public cloud.

There are a lot of proprietary APIs within cloud. At Payara, we have built many messaging connectors for Java EE developers. For example, we've created a set of JCA adapters, Apache Kafka, and MQTT (a protocol that many event services on cloud platforms use).

The two main advantages of using our connectors over using the service API:

1. The service API is proprietary
2. Need to integrate things with Jakarta EE model

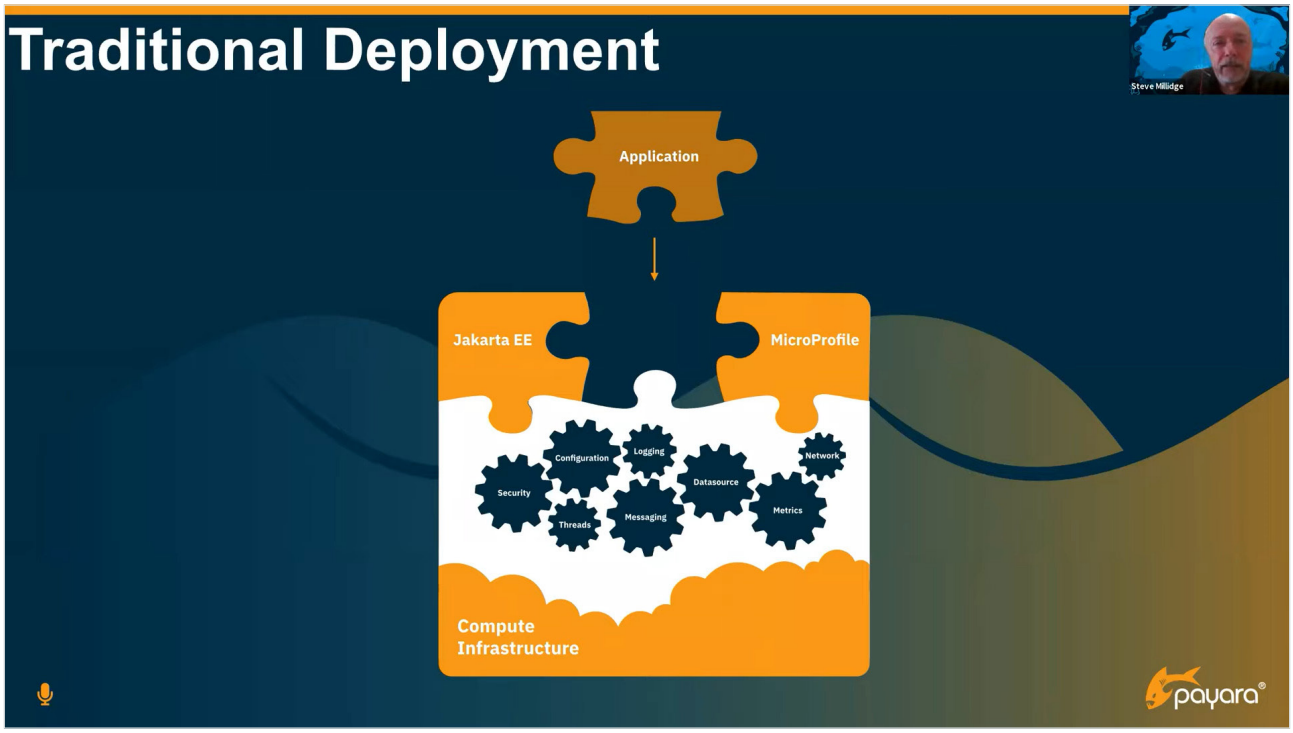
For example, if you use the standard API to get a message from Amazon SQS , then you would need to set up a Java EE context and then use entity managers and session beans or CDI. Payara Platform connectors **do all of that for you so you can just integrate with the technologies**.

There are also different security protocols used by the various cloud providers. Payara Platform integrates with standard security providers like OAuth2, OpenID Connect and LDAP. Our job is to build this sort of ecosystem that makes your applications natively run with the services provided by the cloud providers.

Using standard Jakarta EE, such as Jakarta security, Payara builds connectors that allow your applications to use the security protocols and be in the right context for the rest of the application code.

Cloud Native Runtime - Applying the Jakarta EE Model to the Cloud

So, what if we accepted that all of this is software-defined infrastructure? Is there a way we could completely build a cloud-native runtime for a developer? Or as an architect that needs to deploy a Jakarta EE application?



If we look back at the traditional deployment model, what we see with Jakarta EE is that we are separating the application from the compute infrastructure and our runtime. You create a WAR file or EAR file and deploy it.

> *Some of the work Payara has been doing recently is instead of taking the models that we've just described here (packaging up Kubernetes and creating containers and Pod definitions, and YAML) - instead of doing all of that infrastructure work -* ***what if we took this Jakarta EE traditional deployment model and applied it to the cloud?***

This is what we've done with Payara Cloud.

Payara Cloud: Upload. Deploy. Run.

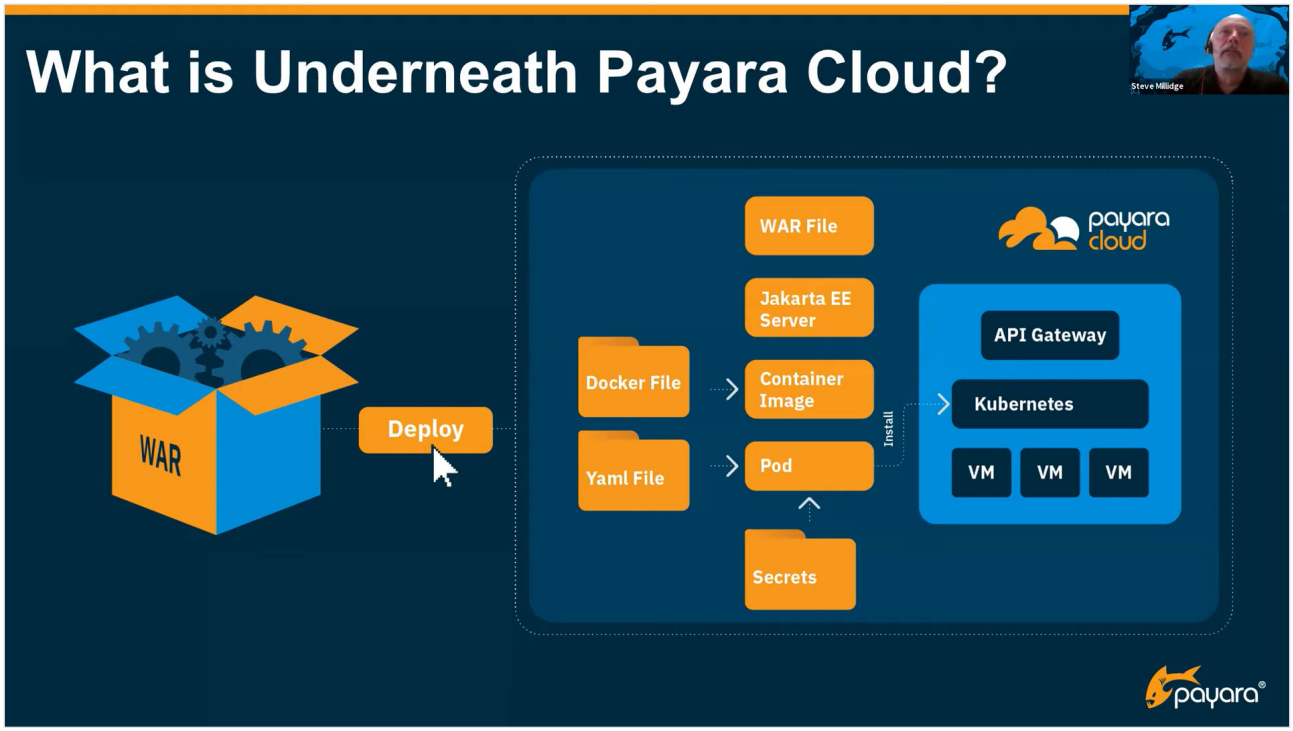
The idea of Payara Cloud is that you can deploy applications to the cloud in a three-step process. We want to take the deployment process of the Jakarta EE model and deploy it to the cloud.

Our view was all you need to do is upload your EAR or WAR file for your application, go onto an Admin Console and click Deploy.

When you log into Payara Cloud, you have a set of namespaces (a grouping so if you have microservices in a single application you could create them in a single namespace, and they'd be able to talk to each other). You upload the application and hit deploy and it just deploys it for you and gives you a URL to access your application - just like you would do with a standard application server.

Ultimately, all of the Payara products are designed to make running on the cloud easier for developers, from Payara Server to Payara Micro and now, Payara Cloud.

How Payara Cloud Works



You may be wondering how Payara Cloud reduces the steps of deploying an application to the cloud. Behind the scenes, Payara Cloud is building all of the infrastructure and deployment models that you can build yourself if you want (as mentioned, Payara Platform is cloud-native and will run in cloud environments) but Payara Cloud does it for you.

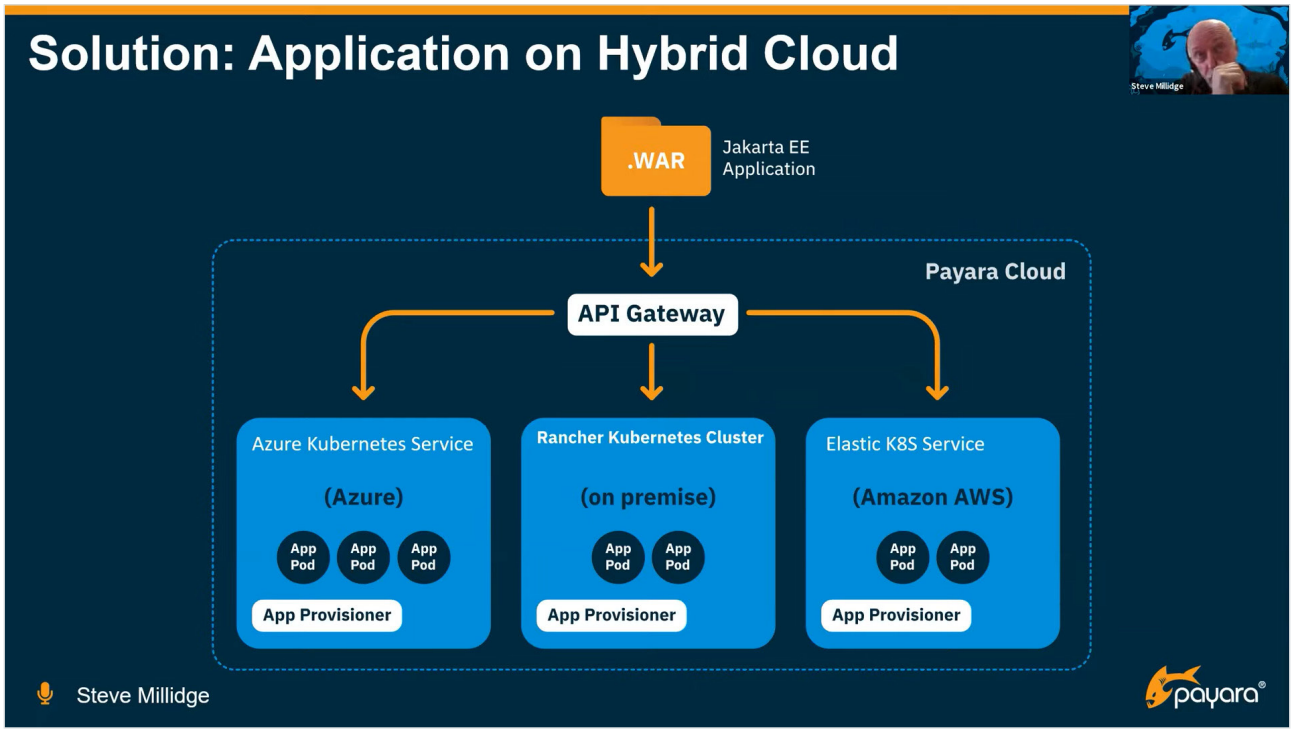
When you upload the application and click deploy, Payara Cloud does all of the tedious infrastructure configuration that needs doing to run. And we sit it all on top of Kubernetes so we can put it on Azure, AWS, or GCP, or potentially, on-premises in the future. Basically, we can take this and run it anywhere.

Payara Cloud packages your WAR file with Payara Micro for the Java EE application server,

builds it into a container image and creates the Docker files, Pods, YAML files, works out Secrets, and allows you to configure external configuration sources, connect to databases. Then, Payara Cloud runs it on top of Kubernetes and sorts out the Ingress for you with an API gateway.

Payara Cloud shows that the Jakarta EE model is absolutely suited for cloud-native deployment.

In fact, we could take this model a step further.



We could take this and deploy it on a complete hybrid cloud. There's nothing stopping us from building infrastructure that lets you take your application and drop it into one of multiple Kubernetes clusters or spread it across Kubernetes clusters. Currently, you can choose regions, but you can't deploy the same application into multiple regions at this time. This is a potential future feature of Payara Cloud.

Because your Jakarta EE application is just a WAR file, we can deploy it and drop it on Azure or Elastic K8 (Amazon Kubernetes), or we could put it on an on-premise Kubernetes cluster like OpenShift. That would run!

Basically, what Payara is doing as a Jakarta EE vendor is managing all of this for the developer so you can build the application and not deal with the infrastructure.

The biggest change in cloud-native is this infrastructure layer.

Myth: Java EE is Not Cloud-Native

Dismiss the Myth: Jakarta EE Lets You Write Your Application Once and Run it Anywhere – Including a Cloud Platform

Developers Can Write Once and Run Anywhere

Ultimately, we want to separate your application from the infrastructure, and Payara Platform provides a platform that can scale from edge servers to containers to bare metal to virtualization to Kubernetes and put it right onto a cloud platform so you can run your application wherever you want.

Java EE is cloud-native. Jakarta EE applications run cloud-natively from traditional deployment on VMs, where Payara has done the work to ensure clustering and elasticity works right out of the box, right through taking an application and deploying it on Kubernetes.

MYTH: Java EE Doesn't Do Microservices

Before we can talk about whether or not Java EE can do microservices we should briefly discuss what microservices are:

> ***"Microservices architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery..."***
>
> ***https://martinfowler.com/articles/microservices.html***

Martin Fowler's definition is one of the first microservices definitions from back in 2014. Microservices are an evolution, to be honest, but they evolved from remote procedure calls from way back in the 1980s and 1990s through Corbo, remote EJBs, web services, service buses, and restful architectures. And now we've arrived at microservices.

In the Martin Fowler definition, microservices is an architectural style about how to architect applications and not necessarily about the individual business code, but how you architect the system.

What people often call a monolithic application, one 'big lump', and split it up into a suite of small services. There isn't a definition for what a "small" service is, and you could spend weeks arguing about what that means, but effectively, this suite of services typically run in their own process in the operating system. They communicate with lightweight mechanisms, typically HTTP.

The idea is you build each service around a specific business capability so that each one manages its own domain. And because they are separate processes they will be independently deployable, so you can take one microservice down without affecting the suite of other microservices - which is unlike a monolithic application.

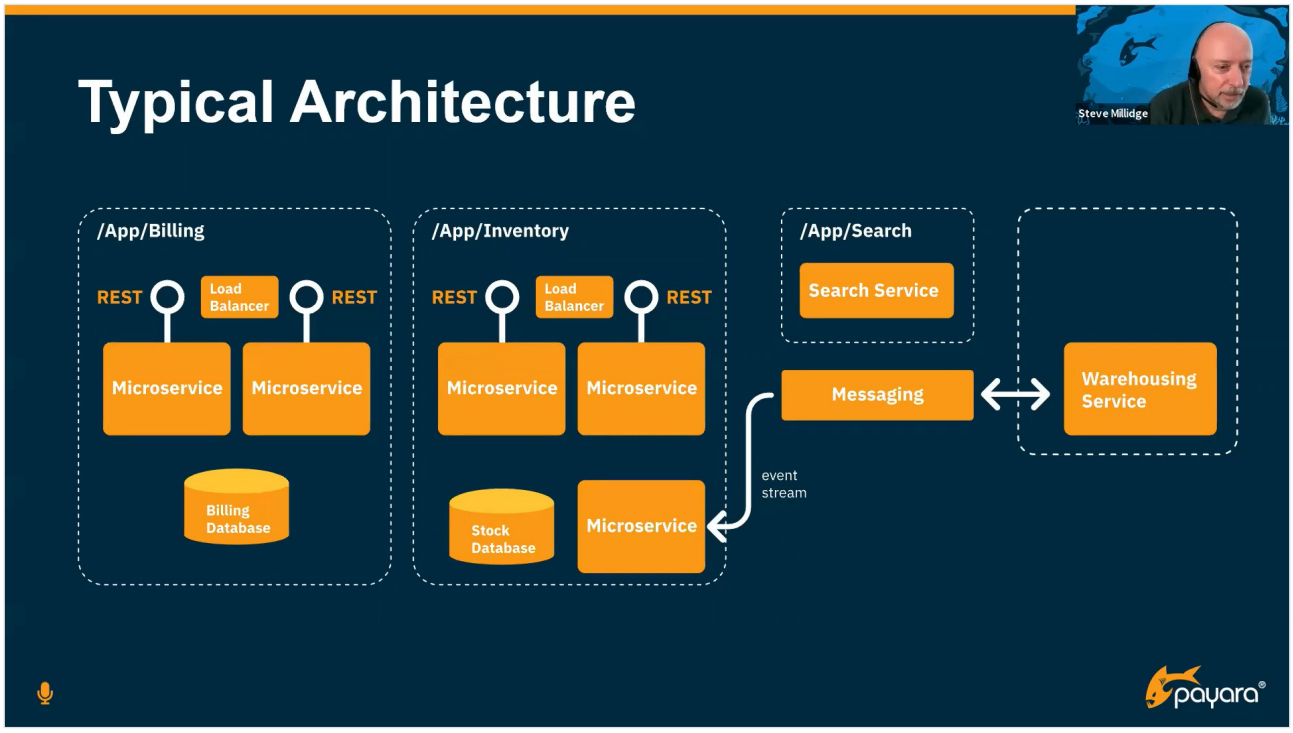
Then, because they are independently deployable, there is a link to automated deployment which means you could potentially roll out each service automatically if you do an update to that service.

The move to microservices has gone hand in hand with the evolution of how compute infrastructure works. As mentioned previously, the hardware that you run your application on has changed in the past 10 or more years. Previously we deployed on a physical server, and then a hypervisor, and then deployed using virtual machines - but now we deploy on containers in a container orchestration platform like Kubernetes.

Microservices can package a service into a container and deploy it automatically onto some compute infrastructure. So, all of the trends come together to form the microservices architecture.

You don't have to use containers with microservices, but it helps with the automation deployment piece of this definition.

Typical Microservices Architecture



Let's imagine you have an ordering section of an ecommerce website where you sell stuff online, and you need to take the various business domains associated with that, for example, billing. The billing domain might involve credit card payments or raising invoices. What we can do in microservices architecture is take that business domain and build a number of operations in the microservice that would serve the billing domain.

Within the microservice you would potentially have a rest API, a load balancer to independently scale your microservice out, and potentially a datastore.

You could do the same thing for inventory. So if you needed to work out how much stock you have, you could have an inventory service that works with a database to see if you have items on hand or if they need to be ordered.

Then you take your application and split it up into these business domains. You could also potentially have search or warehousing to send orders to be picked and shipped.

In this illustration above, it shows microservices typically use REST, but it's not required. REST is used to retrieve or update data. But they can also talk through messaging streams and middleware like JMS, or Apache Kafka streaming technology. They can also update microservices through event streams from warehousing, for example.

If you were running your ecommerce website through a monolithic application, all of the various business domains would be built in the same deployment module.

The idea of microservices is that it's split out and each microservice does one thing. Each one could be developed by a separate team, and you tend to have a separate domain database. It's not just about your service architecture, microservices is also about your data architecture, where each data would be associated with one or more microservices and be very specific to one business domain.

Microservices has also changed the user interface technology. Today, we build a lot of UI for web applications using Javascript frameworks. This allows us to take a microservice and call multiple REST APIs from a Javascript and then assemble and render a user interface that fuses data from each of the different services (client-side rendering).

Previously, we did a lot of service side rendering. Service side rendering is much more difficult if you had a microservices architecture because you would need to do a lot of calls on the service side from something like JSF or some other service side rendering framework to pull data from various services, make it into HTML and throw it to the client.

Microservices has evolved to complement the Javascript framework.

The challenge from an architectural point of view is working out what the business domains are, what the services are, and what constitutes a microservice and which domain it should be in. You have to figure out whether to create 10 microservices or 1000. The challenge is figuring out how to split up your requirements into good components of software.

Advantages of Building Microservices

The advantages of building an application as a set of microservices include:

Modularity

If you're doing something that is a well-defined domain, like billing, then you can concentrate on that domain and make it modular and not closely coupled with other domains. This allows you to break them down and analyze them in pieces.

Separately scalable

From a deployment perspective, each microservice is separately scalable. If you had a lot of search traffic on your website, then you could fire up 10 or 20 of the search microservice, while you may only need 1 instance for the warehouse. If you put it all on a single application, then you'd have to scale the entire application. Monoliths are actually scalable too, but you can't scale different parts of the application.

Independent development

You can break your development team up arranged around your microservice. Each team can concentrate on a single group of microservices. There are some challenges with this in that your architecture will reflect how you split the development team up, so you need to be careful that you build your architecture around how you want to split your microservices.

Independent deployment

You can independently deploy the microservices. For example, you can take your search service and deploy it completely separate from your warehouse service. If you package it all into a single application, then you'd have to redeploy the whole application.

Loosely coupled

Microservices are loosely coupled, meaning changes to the design, implementation, or behavior in one won't cause changes in another.

Replaceable

If you want to replace your search service with version 2, then you can create it and move it in and replace your version 1. This isn't something you get for free, you have to architect your interfaces to your microservice so version 2 will comply with version 1, otherwise you would get coupling with different services. So, you do have to architect that capability, but it is possible to replace a single service rather than the whole application.

Highly testable

Each microservice should be testable. This means each unit can be tested individually from the rest of the microservices.

Polyglot

If you want to, you can write microservices as a polyglot rather than Java. Some organizations find advantages for doing this.

Small teams

You can create small teams to work on individual projects.

Challenges of Microservices

Microservices are an evolution of architecture and so you'll need to keep an eye on certain aspects of distributed computing. You have to worry about the fallacies of distributed computing. Whenever you split something up into a suite of microservices that are independently deployable, then you are basically building a distributed architecture. And therefore, you have to be concerned about issues affecting distributed computing issues, including:

- Network reliability
- Latency is zero
- Bandwidth is infinite
- Network security
- Topology doesn't change
- Having one administrator
- Transport cost is zero
- Network is homogeneous

When you move into microservices you need to be concerned about these things and the tradeoff between course-grained services and fine-grained services. There are APIs that can help.

Disadvantages of microservices compared to monolithic applications include:

Operational complexity

Even though each service is independently deployable and scalable, on it's own compute infrastructure, this introduces a lot of moving parts. How you manage that and monitor it becomes more complex.

Distributed

Distributed computing presents a number of challenges as described above.

Eventual consistency

When you split a database it can lead to issues around eventual consistency. For example, if you update the warehouse database and send a message to the actual ordering system to update inventory, that message could take latency to get there, during which time the databases will be out of date. Where as if it was all in a single scheme, that would be an unlikely issue to deal with.

Maintenance complexity

The multiple moving parts of microservices lead to much more complexity than trying to maintain a monolithic application.

Complex system testing

Single microservice testing can be easier in a microservice, however system testing across the entire thing is more complex because of the distribution.

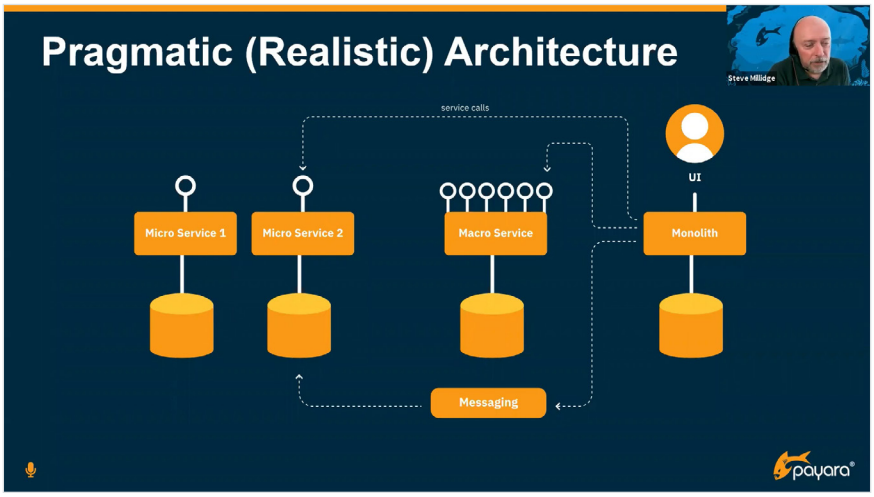
Lifecycle decoupling

You can build terrible microservice architectures where you get a lot of lifecycle coupling, for example if microservice 1 calls microservice 2 - you're now in a distributed situation and you need to know what happens when microservice 2 is not available. Developers working on microservice 1 need to handle that complexity which would not exist if this was in a monolith because the second service would be deployed together with the first one.

Fault tracing

Similarly, if you have something bouncing events around a network and services are calling other services, if there is a fault you don't get a straight stack track like you would from a monolith. In microservices you have to trace and track across multiple services and machines to work out what happened.

Pragmatic (Realistic) Architecture



In reality, you're more likely to end up with a situation where you have a combination of all different architectural models. You're going to end up with a monolith for some part of your enterprise architecture, and a macro service with multiple microservices deployed together, and you can have microservices for different parts of architecture.

As a software engineer or architect, or technical architect, your job is to create the most optimal solution that meets the needs of your organization and not to just follow some paradigm or architectural pattern. So it may be that for efficiency, operational, or performance, you group microservices

together and deploy them together. You might do server side rendering or create a specific monolithic application because it's the most efficient for your use case.

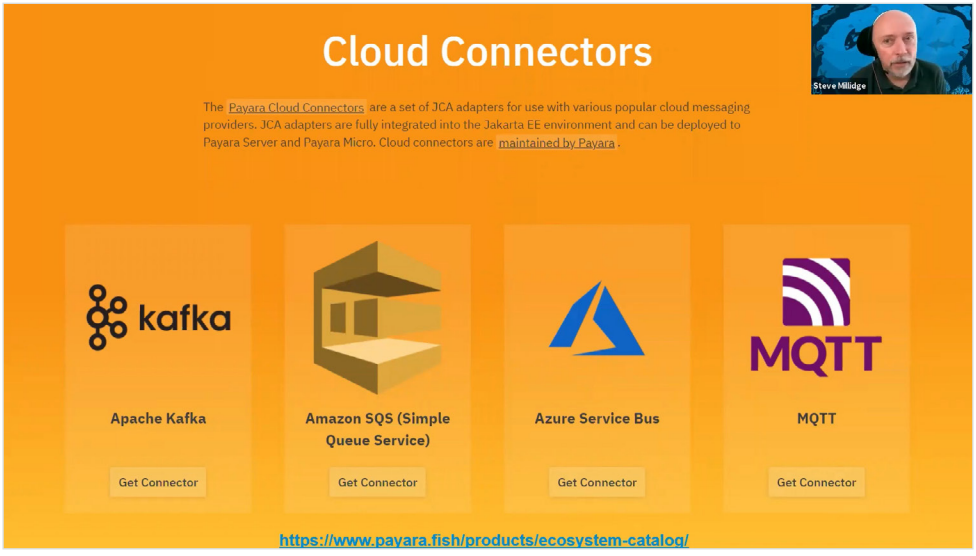
For example, if you don't have large teams and you don't need to independently scale services, and you don't have hundreds of thousands of users or spikes in traffic then some of the disadvantages of microservices come into play and may outweigh the benefits. As the architect, it's your job to track the right course.

Jakarta EE APIS for Microservices

Typically, when we are talking about microservices it is very closely associated with REST APIs. Jakarta EE has Jakarta REST, standard REST API for building RESTFUL web services and is used in other frameworks like Dropwizard. We have data transfer, so we can support JSON-P and JSON-B. Many REST web services for a user interface will be talking to Javascript and moving data across the network through JSON.

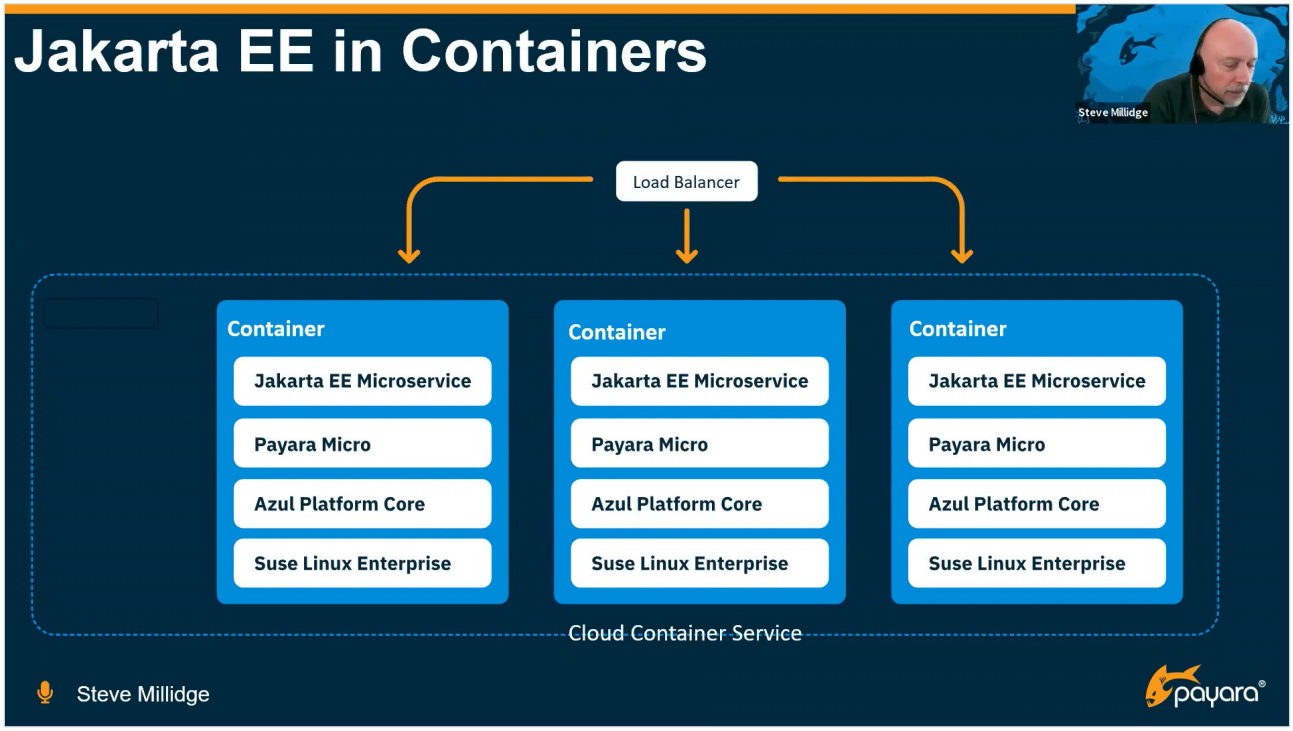
Jakarta EE has CDI which is a dependency injection framework for building your business logic. From the transaction side through integration of data, we have standard JPA (hibernate for example), and JMS for messaging. We also have JCA, which doesn't come up very often, but can be used to build connections to other systems to send messages.

In Payara Platform we have standard JCA connectors for different cloud messaging so you can deploy Kafka or Amazon AWS into your Service Bus and talk to different things that are not JMS compliant and send messages and event streams across different technologies.



All of these things can be integrated with Jakarta EE applications. Payara Platform has all of the APIS that you could have to build microservices.

Deployment of Microservices in Jakarta EE

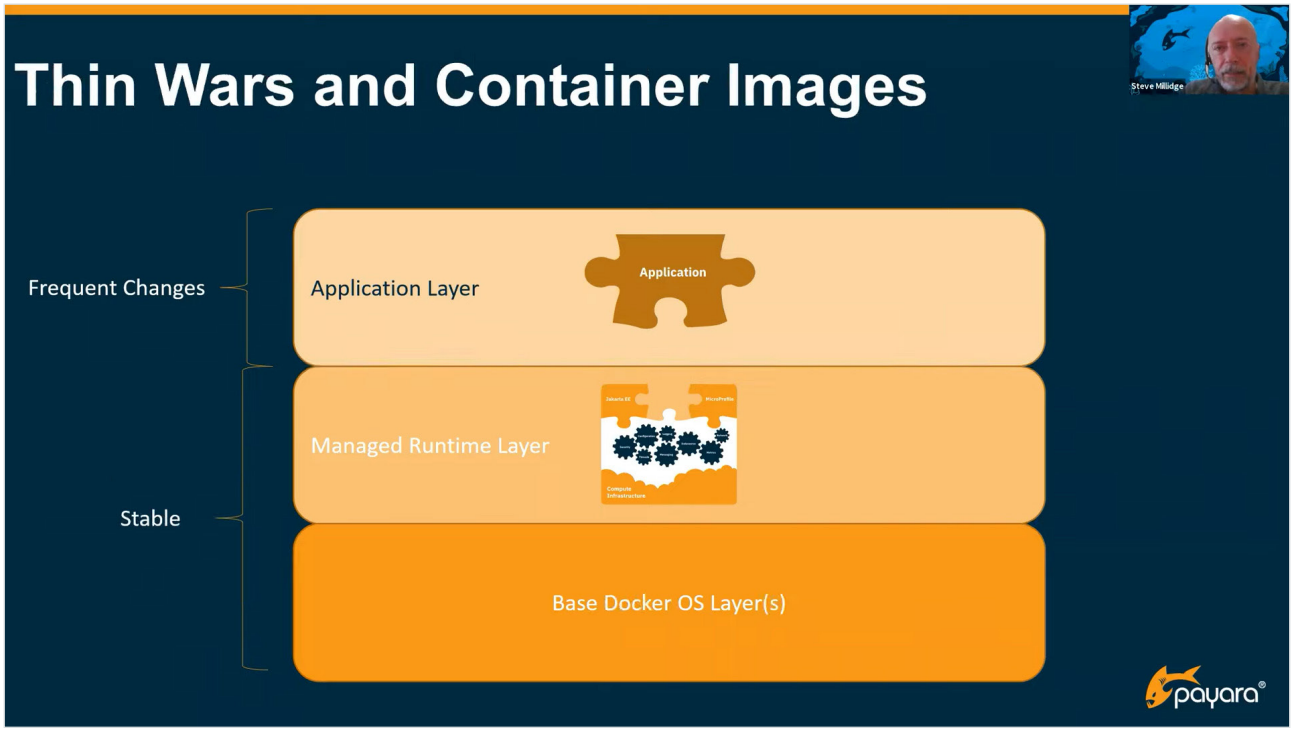


The microservices architecture works really well with containers because you can take your micro-service and package it into a container, which packages the application configuration, and deploy it onto a cloud container service or a container service - like Kubernetes.

Jakarta EE has evolved completely from only supporting domain-style application deployment into runtimes like Payara Micro (a single Jar that you can package a microservice as a WAR file on top of Payara Micro and push it on top of a JVM and wrap it in a container and throw it at a container service).

You can run multiple containers into a single machine or VM or whatever you choose to run it on as a compute infrastructure. These can be independently scalable so you can scale out your container service, microservice, and that would scale out your Jakarta EE microservice.

There are some advantages of Jakarta EE microservices over other sort of Fat Jar technologies in that when you build containers you build them in multiple layers, as we discussed previously. Typically, you would have the base operating system layer, the managed runtime layer (in our case, the Jakarta EE runtime like Payara Micro), and you package the microservice on the application layer:

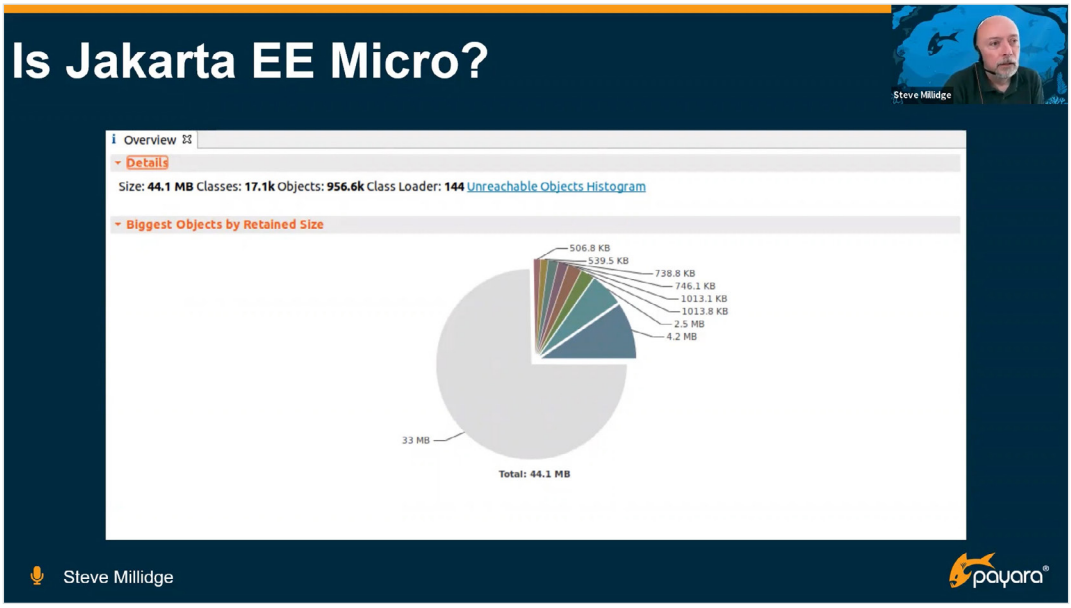


Because your application changes more often than your runtime, and changes in your applications are quite small, this setup allows you to push out the containers across the network faster because most of the container technologies just push the changes across when deploying new versions. You can rapidly deploy new microservice versions by changing just the application layer.

Microservices on Kubernetes Deployment Model

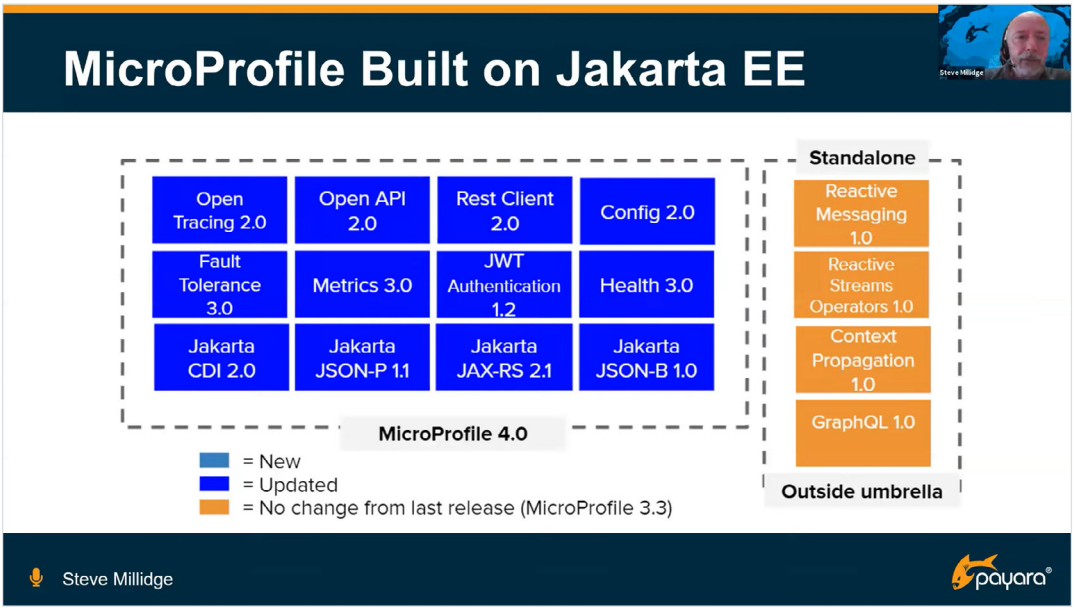
This also scales onto Kubernetes, a platform for managing containers at scale. It typically works on containers and a Pod (a definition of a microservice along with its container image). You can scale out Pods independently from microservices. And you can do that with Jakarta EE with things like Payara Micro.

An earlier myth that we already debunked, believing Jakarta EE is heavy, would lead people to believe there is no point in creating a container for a microservice for Jakarta EE. Here is a heap dump of Payara Micro running a microservice in Jakarta EE:



Jakarta EE is very small and can be packaged into containers. It has REST API. You can do all the things you need to do for microservices. You can build applications and connect to databases. In addition to Jakarta EE, there is an initiative called MicroProfile.

MicroProfile is a set of APIs that track the challenges of microservices architecture and build on top of Jakarta EE APIs to provide things to help with those challenges of deploying and running micro-services architecture.



For example, one of the challenges to microservices mentioned previously is that it's difficult to debug and trace what's happening in a distributed system. Open Tracing provides APIs where you can build traces of remote calls and get them for debugging purposes.

To mitigate the fallacies of distributed computing, Fault Tolerance allows you to add annotations to a remote call and if it finds it's not available, you can provide fall back to a different service, or APIs for retrying the call and doing things to limit the level of concurrency you're getting through your service.

MicroProfile has a REST Client for calling other REST services. Open API defines REST endpoints and JSON documents. Config gives you an externalized config so if you're deploying containers, they can configure themselves from external sources, like Kubernetes Secrets or other external config sources like your database.

These can all be built on top of the Jakarta EE APIs.

Metrics helps gather all of the data and metrics from different services and combine them into a centralized system. And metrics API allows you to build business metrics.

Health is a simple API that lets you expose whether your service is alive or dead and enables container orchestration services to know whether to kill and restart your service or not.

There are a bunch of extension APIs on top of Jakarta EE which are explicitly for building microservices architectures. MicroProfile is a multi-vendor initiative and includes implementations like:

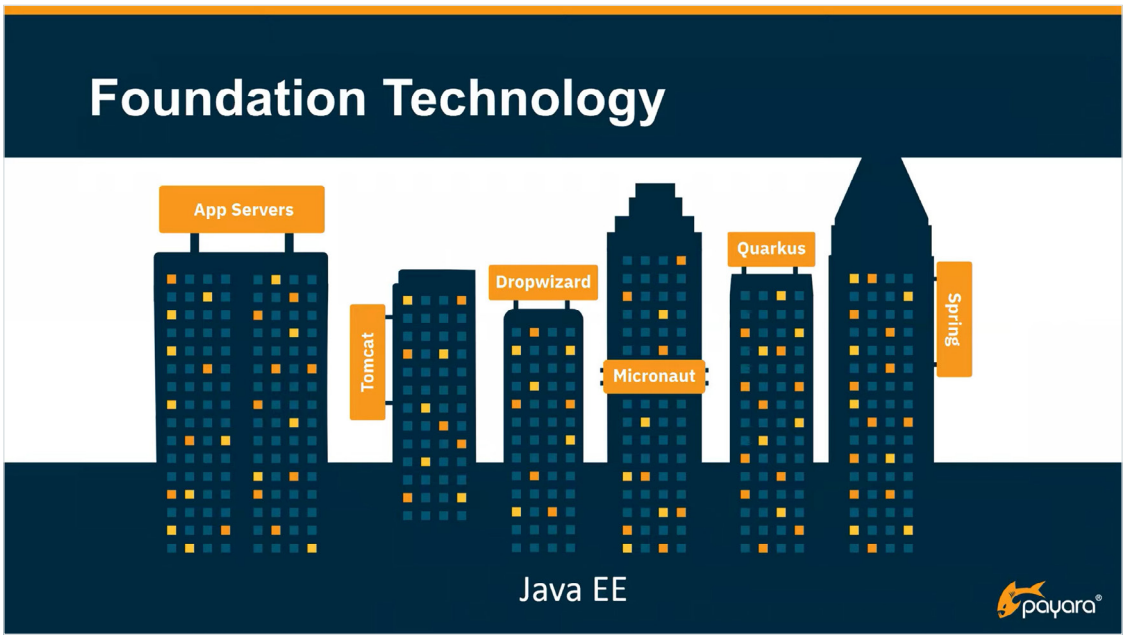
MicroProfile are industry standard APIs that work with Payara Platform.

Myth: Java EE Doesn't Do Microservices

Dismiss the Myth: Jakarta EE and MicroProfile Easily Deliver Microservices

To get back to the myth that Java EE cannot do microservices and that it's only for monolithic architecture - you can now see that Jakarta EE is small and has many APIs. Jakarta EE provides the foundation and MicroProfile provides extensions which are specific for microservices architectures. We can build them into small containers, deploy them and have all of the characteristics of a micro-services architecture using Jakarta EE.

MYTH: Java EE Standards Don't Matter



One of the key things to understand about Java EE standard is that it is a foundational technology for a lot of server-side Java. It's not just around Java EE/Jakarta EE application servers, like Payara Server and Payara Micro, but it is a foundation of other technologies, for example, Tomcat is a servlet container.

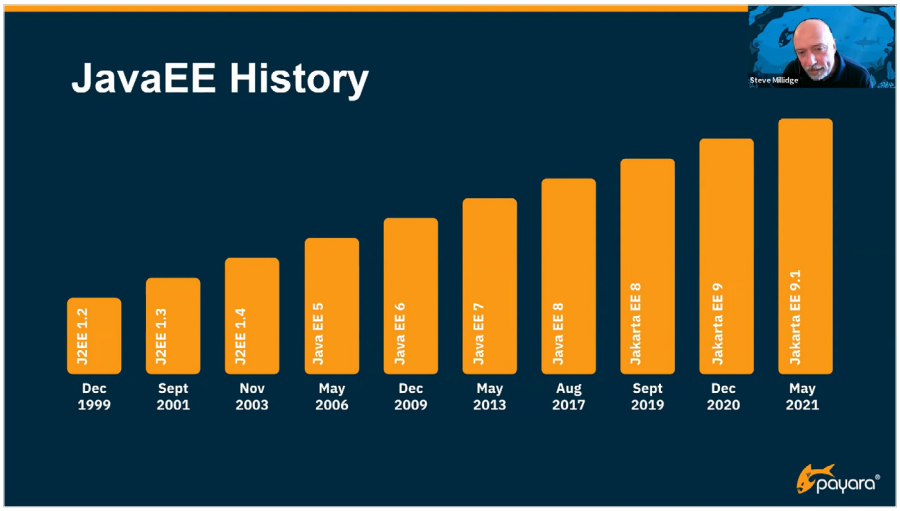
Some of the microservices frameworks, like Dropwizard or Micronaut, or Quarkus, all use the Jakarta EE specifications to some extent. For example, Dropwizard is heavily based on Jersey, which is an implementation of Jakarta REST API. Micronaut uses dependency injection, which is part of Jakarta EE. Quarkus is by RedHat and uses quite a few Jakarta EE specifications. Similarly, Spring uses a large number of Jakarta EE specifications even though it is not a Jakarta EE application framework. It uses things like Hibernate, which is a JPA implementation and can integrate with a lot of Jakarta EE specs.

When we're talking about Jakarta EE, we're not just talking about application servers. So we'll cover how the individual specifications get developed, and why standards are a good thing and how they help provide innovation to these frameworks and to things like Payara which are full implementations of the Jakarta EE platform.

Java EE History

We've looked at the history of Java EE previously. It goes back 22 years to 1999 with the release of J2EE 1.2. This was a basic server specification. Java EE has evolved over the years right up to the latest release, which is Jakarta EE 9.1 in May 2021. Payara 6 Alpha supports and is compatible with Jakarta EE 9.1.

There have been a number of major changes to Java throughout history. Java EE 5 in 2006 was when Java EE adopted its modern name and its modern technique of development through annotations. Java EE still gets criticized for things that happened before 2006. But by Java EE 5, EJB, JPA, and modern foundations for the framework were put in place.



The name change from Java EE to Jakarta EE happened between 2017 and 2021. During that time, Java EE transitioned into the Eclipse Foundation including the standards and API. Previously it had been managed by Oracle.

There have been a number of releases since Java EE moved to the Eclipse Foundation. Between Java EE 8 and Jakarta EE 9 was the namespace change between *javax* which was owned and trademarked by Oracle to *jakarta* which is independently owned by the Eclipse Foundation.

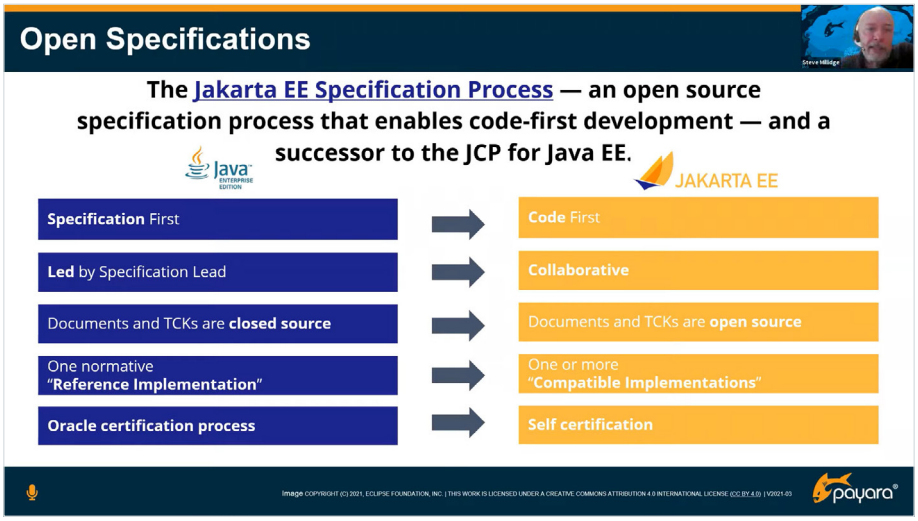
Java EE 8 was down under Oracle's stewardship and the JCP (Java Community Process) released in 2017. The first release from the Eclipse Foundation was Jakarta EE 8 in 2019 and it was exactly the same as Java EE 8. It was just released under completely different stewardship, governance, and license. This marked an important transition. It meant you could take a Java EE 8 application and run it as a Jakarta EE 8 application under open source license and artifacts under the Eclipse Foundation.

This also marked a major milestone for the Payara team as we could release and make Payara Server a compatible Jakarta EE 8 application server. Previously, under Java EE, we would have had to enter a license agreement with Oracle and given Payara was founded in 2016 and we knew Jakarta EE was coming, we didn't get the licensing. The open licensing with Eclipse has helped other implementations, too, like TomEE and TomCat. Because Jakarta EE is open source it is easier to become a compatible implementation.

Jakarta EE 8 was released under the *javax* namespace. Jakarta EE 9 was released in the jakarta namespace. Jakarta EE 9 is similar to Jakarta EE 8 from an API perspective to allow developers to simply change the namespace to get their applications to work on the new namespace. Then Jakarta EE 9.1 came out in May 2021, which was a minor release that allows you to run TCKs against Java 11.

Open Specifications

The biggest difference in the transition of Java EE from Oracle to the Eclipse Foundation is that the specifications are now open.



Previously, Java EE was developed under the JCP. The JCP operated in an open way, but it wasn't open source. In the JCP, typically specifications are developed specifications first. People would develop the specification document and APIs in a closed group. Now, in Jakarta EE, we are code first through open source development mechanisms.

In the JCP, there was a specification lead which doesn't exist under Jakarta EE. A specification lead would gather a group and if you wanted to be involved you had to apply to be a member of the spec group. Jakarta EE is completely different and is collaborative and open source.

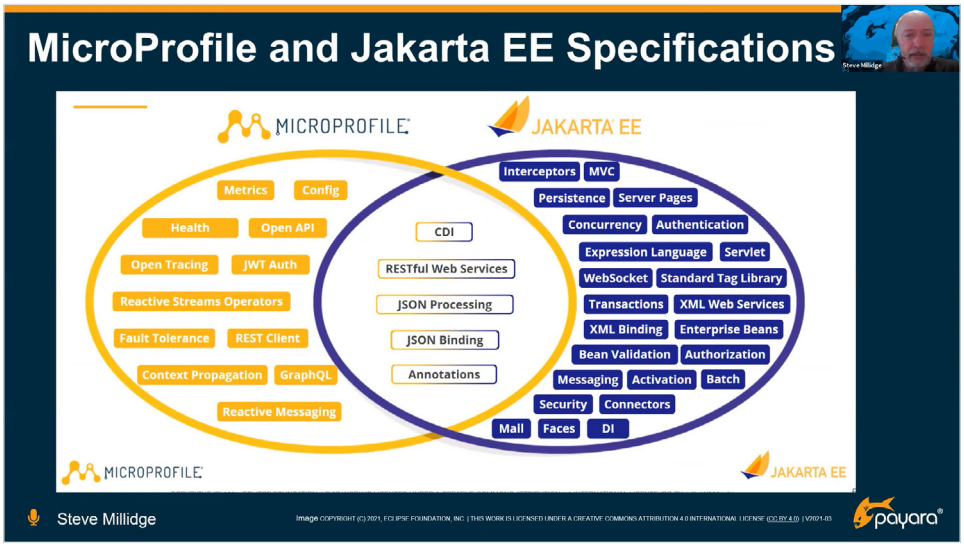
Under JCP, all of the spec documents and TCKs are closed source and you couldn't get access to them. Before the transition, the Payara team did not have access to the Java EE TCKs, so we developed our own test suites. But under Jakarta EE, the spec documents and TCKs are open source and under open licenses.

JCP also had a concept of a "Reference Implementation", and that was GlassFish for the Java EE platform. Jakarta EE doesn't have a reference implementation, but it has compatible implementations. They can use any implementation that passes the TCKs and now that the spec documents and TCKs are open source, anyone can get a hold of them and pass the TCK.

In the JCP, Oracle controlled the certification process, while in Jakarta EE anyone can certify. There are some requirements for self-certification but they're not closed.

The goal of the transition was to move everything into an open source licensing, testing, and take Eclipse Foundations best practice collaborative open source models and apply them to specification creation.

MicroProfile and Jakarta EE Standards



MicroProfile and Jakarta EE are both working groups within the Eclipse Foundation. There are a huge range of specifications and APIs that are covered among the two working groups. They share many as a foundation, the CDI, RESTful Web Services, JSON-P, JSON-B, and Annotations. On the Jakarta EE side, there are a lot of fundamental server side Java APIs:

Servlet API which is the foundation for TomCat and transaction APIs.

Some of the APIs have been moved out of the JDK, like XML Binding, which is now under control of Jakarta EE working group.

Fundamental APIs that have existed for 20+ years are now part of the specifications of the Jakarta EE working group.

On the MicroProfile side, the APIs are basically developed as extensions of Jakarta EE and are specifically for microservices architecture. They are a little more experimental than Jakarta EE. Also, MicroProfile carried Jakarta EE forward from an application server point of view during the transition between Java EE 8 and Jakarta EE 9 where there were no API changes happening under the Jakarta EE side there were a lot of changes happening on the MicroProfile side. This brings products like Payara Micro and Payara Server up to speed and provided capabilities to provide microservices.

What is a Working Group?

There are two sides of the Eclipse Foundation. In Jakarta EE, there is the working group, which is a group of companies that effectively funds the work on the specifications. It's an industry consortium which controls the brand, establishes part of a technical roadmap, defines the rules around compatibility and what makes a compatible product. Ultimately it will approve specifications and provides funding so the Eclipse Foundation can build the community, manage the process, outreach, and conferences.

The work itself for advancing the code is done in the open source side of the Eclipse Foundation in the Spec Project.

What is a Specification?

When you're creating a specification you essentially have to come up with three things:

- Spec
- API Jar
- TCK

A specification requires an API Jar, you have to create an API like the servlet API, or if you're creating one from scratch, like the NBC API. The API Jar defines the API developers will use when using this specification.

You have to create a Spec Documentation to give the contextual information about the specification, how it should be used, and any other things not in the Java Doc. Some specifications have long documents and some are short.

You also have to create a TCK - which is a Technology Compatibility Kit. This defines a suite of tests that someone has to pass to say they are compatible with an API. Once they've passed the TCK they can claim to be a compatible implementation of that specification.

The TCK ensures that anyone who claims to be compatible, like Payara, you can prove that they are because you can take that TCK and run it against Payara independently of Payara, and prove that we are compatible.

To release a specification you need these three things and a compatible implementation that has passed the TCK. That means while the API, Spec, and TCK is being developed, someone somewhere (IBM, Payara, independent open source project, etc) has to implement the API and pass the tests to prove the API you created is implementable. That means you can't go off as an API team and create an API that no one can actually implement. It also provides feedback into the APIs to make sure it makes sense.

There can be any number of compatible implementations, but there has to be at least one implementation freely available for download. Then the spec can be released.

A platform, like the Jakarta EE full platform containing all specs, or the web profile, contains these three things. To release a platform spec you also need a compatible implementation that fully passes the TCK.

How A Specification is Developed

The specification process is fully defined by the Eclipse Foundation. Basically, anyone can do the first step, which is to raise a proposal for a potential specification with Jakarta EE. You don't have to go through a specific vendor. If you have an idea you can create a proposal.

The proposal goes to the Creation Review Working Group Committee. Then the proposal follows the open source development process:

- Plan
- Plan Review
- Development
- Milestones
- Progress Review

When you want to create and release the spec, then you have to go through a full release review. That checks that you have an API, a TCK, and a spec document and you have a compatible implementation. At that point, it's voted on to become a ratified spec.

The voting on the ratification of the final spec. is when all of the intellectual property flows into the Eclipse Foundation and away from anyone who has collaborated on the creation of the spec, and from a licensing perspective, if you want to implement or use the API in a product - the IP flows are at the Eclipse Foundation with open source licensing including both patent and copyright licenses.

The Eclipse Foundation provides a trusted foundation for specifications. If you want to go off and implement one of these APIs, you know you have a robust legal framework for you to do so without any risk.

Open Source TCK License and Process

Prior to moving to the Eclipse Foundation, the TCKs were behind licensee paywall and it was impossible to get access to any of the TCKs for the specifications without becoming a licensee. Now, this is completely open source. The full TCKs for all the specs for the full platform are transitioned to Eclipse and it is an open source project.

Prior to the move, Payara had no access to the TCKs. This gives us transparency. Before, people would complain that the code couldn't move from one implementation to another because of the grey areas of the specification or TCK. Now, we fix that easily. We can do a pull request on GitHub for the next release of the TCK to fix it or add coverage.

It's also completely vendor neutral. The burden is not on a single vendor. Now, the burden is shared across all of the vendors and the whole community.

To claim compatibility, someone like Payara has to publish the TCK results. Under the JCP, compatibility was managed by Oracle. Now, compatibility is self-certification. For an organization like Payara, to prove we are Jakarta EE 9.1 compatible implementation, we have to publish our TCK results on a public forum.

You don't need to join the Eclipse Foundation to get a compatible implementation. You just create your own project, run your implementation against the TCKs, and if you pass the TCKs you are compatible. This opens the market for further innovation. The barrier to creating an implementation of any of the specs is now just the challenge of creating the implementation itself, rather than the commercial challenges and licenses that we had before.

The TCK development is a huge job and the community is welcome to contribute through GitHub issues and PRs.

Myth: Java EE Standards Don't Matter

Dismiss the Myth: Java EE Standards Offer a Choice of Runtime, Vendor Neutrality, and Longevity

Now that you see how the specifications process works, you can see there are several benefits to having the standards.

Advantages of Standards and the Open Source Process

1. Choice of Runtime

The unique thing about Jakarta EE and MicroProfile is that there are many implementations. There are implementations of the entire platform as well as implementations of individual APIs. Your code will work with any and all of the implementations because they all pass the same TCKs.

2. Vendor Neutrality

Jakarta EE has a lot of people on the Working Group. The Working Group manages the process, not the development of the APIs. There are many organizations involved in managing the process of creating APIs, it's a huge industry initiative. We have complete vendor neutrality because it's at the Eclipse Foundation, which is an open source, not for profit organisation. If anyone wanted to drop out of the process, none of the specifications need to stop or disappear. You are not at the whim of a single vendor since it's governed by the Eclipse Foundation. It allows a small organization, like Payara, to sit alongside larger organizations like IBM or RedHat.

3. Longevity

The specifications are building backwards compatibility. The companies may come and go, but the specifications can continue. That gives developers or someone who wants to run production systems on this technology confidence that it will last a long time.

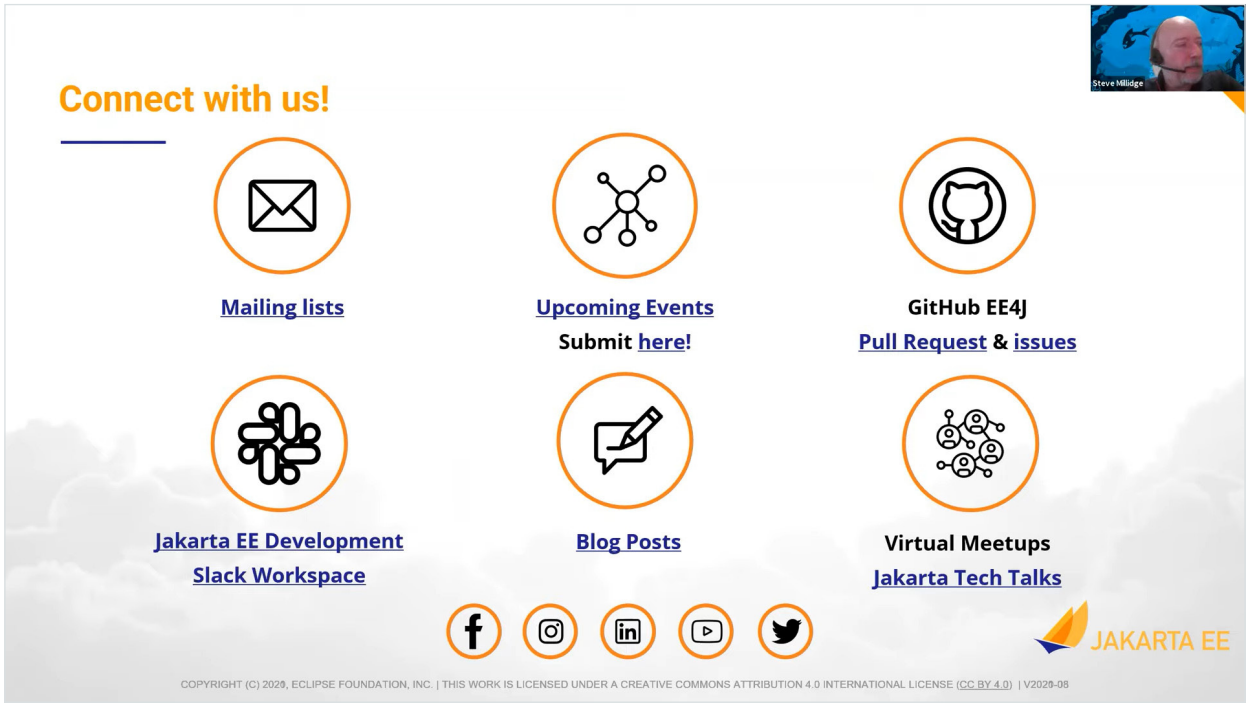
Some other advantages if you want to become an implementer of any of the APIs:

- Open source
- Open governance
- Patent protection
- Simple license
- Independent stewardship
- Industry standard
- Independent implementations

How to Get Involved with Jakarta EE

Because everything is done through the Eclipse Foundation, everyone can get involved. You can help with marketing, technical work, collateral work, or just report bugs you discover when using any of the APIs.

Every project has a mailing list. You can choose the project you want to get involved in and add yourself to the mailing list.



If you want to commit code, you don't need to be a committer on the project. First, you would need to have a contributor agreement to have a PR merged. That makes sure you sign the intellectual property rights over to the Eclipse Foundation.

You can also become an Eclipse Foundation member and if you're an organization, you can join the Jakarta EE Working Group.

MYTH: The Java EE Deployment Model is Out of Date

Another common myth about Java EE is that the deployment model is out of date. To show you why this is not true, first we'll review what's in the Jakarta EE applications (and what isn't!), and review the runtime models that we've discussed in depth previously.

What is in a Java EE/Jakarta EE Application?

In Jakarta EE, you build your application separately from your application runtime. The application has your application code (classes), 3rd party application dependencies, and optionally, you may have deployment descriptions if you're using an older application.

It's packaged into a single archive, either an EAR or WAR file.

From a developer perspective - your work is about done! You wouldn't typically have any Jars from jakarta EE APIs.

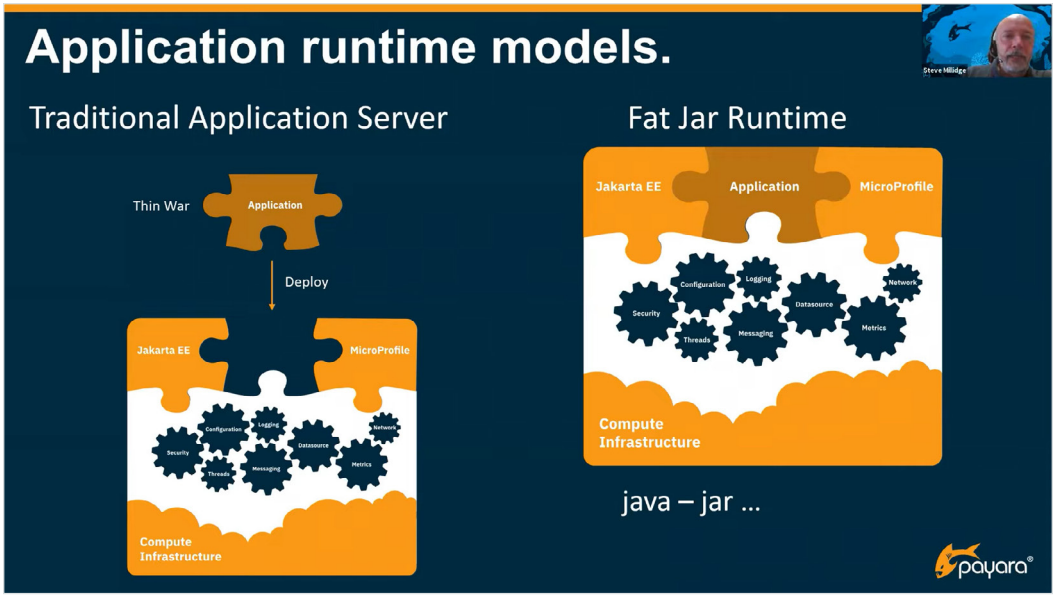
With this deployment model you build the archive.

What is NOT in a Jakarta EE Application?

Your application does not contain anything that isn't directly related to your application. For example, you wouldn't typically have code to do connection pooling or datasources. You wouldn't have different messaging code, you would use JMS connectors. You wouldn't bring in third party code from metrics or network code or to embed a web server. You just concentrate on building your application and not the infrastructure.

All of that other stuff is provided by your Jakarta EE runtime, like Payara. The runtime provides a bunch of services to your application.

Jakarta EE Runtime Models



There are a few different types of runtime models that you can build and they apply to Jakarta EE and into other Fat Jar type frameworks (like Springboot).

Traditional Application Server

Typically, in Jakarta EE, you would package your application into a Thin War, which only has application code and any third party dependencies your application needs. And then you can deploy it to a runtime somewhere. The runtime provides a whole host of services for you and your application, so you don't have to package them with the application.

Fat Jar Runtime

The alternative model is that you would package everything up together into a single archive. In some frameworks you would have to assemble which 'steak sauce' you want for your technology, which connection pooling, how you want to do metrics, messaging, and configuration, security. You would pull it all together and assemble the runtime alongside your application and package it into a single archive. Then you take the archive and deploy it into your compute infrastructure, either running it on the command line with a java - jar, or potentially package it into a container.

You basically have everything in one archive, which is why it gets called 'fat' since it tends to have a lot of code inside it.

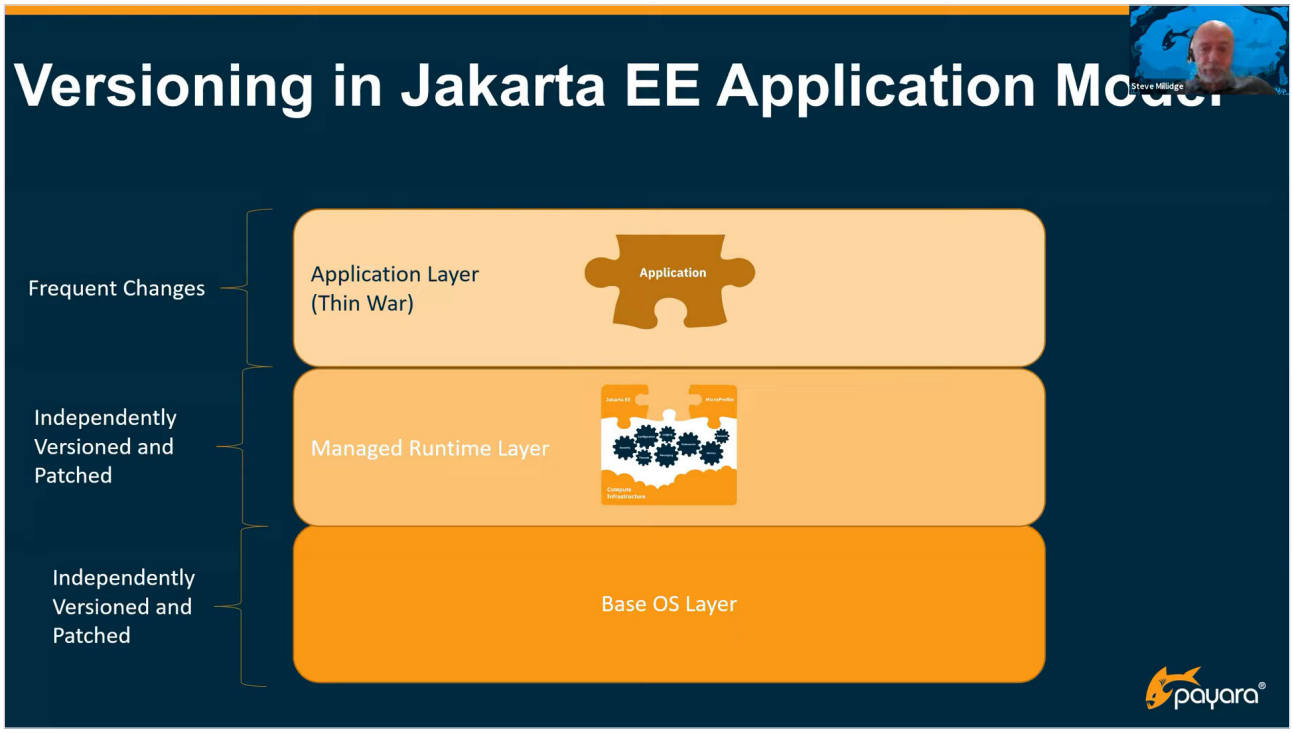
Advantages and Disadvantages of the Jakarta EE Deployment Model

The Jakarta EE deployment model is that you build your application separately from your runtime and ship it separately as opposed to an old fat jar where you assemble your runtime from a menu of parts.

There are advantages and disadvantages of the Jakarta EE deployment model. In Payara Micro, it supports the bundling of your Payara Micro container with an application into a single Jar if that's how you want to package it. But it also supports running and deploying your application separately to the runtime.

Versioning in the Jakarta EE Application Model

One of the biggest advantages of the Jakarta EE model is versioning.



As a developer it might seem easy to take your application components, your runtime, and assemble it how you want it, and then put it into a single archive and run that on your compute infrastructure. But the problem or disadvantage of doing that comes when you need to make changes to your application or microservice in production or understand what you have.

For example, if there was a critical CVE on a component that is running in your Fat Jar, who knows what's in that Fat Jar? Who knows whether that CVE has been fixed or whether the service with a single Jar file has that component within it?

If you break down the model into three layers, then you have different versioning more easily. For example, you have a base operating system layer that you either package in your container or run on a VM or bare metal server. That will be independently versioned and patched from your application. The developer often doesn't have control over what version of the operating system is running in the data center or cloud. And you would hope someone else is patching and maintaining that layer of the operating system.

Similarly, when you have a managed runtime like Payara Micro or Payara Server, or any other Jakarta EE server like WebLogic or WebSphere, then you can have an independently versioned and patched runtime as well. That includes all the security code, the HTTP handling, and all the layers shown previously that is delivered for you by your Jakarta EE runtime. For example, at Payara, we maintain the runtime and issue new releases monthly and monitor for security issues and provide patches or alerts or fix problems in the next release of the Payara Platform runtime.

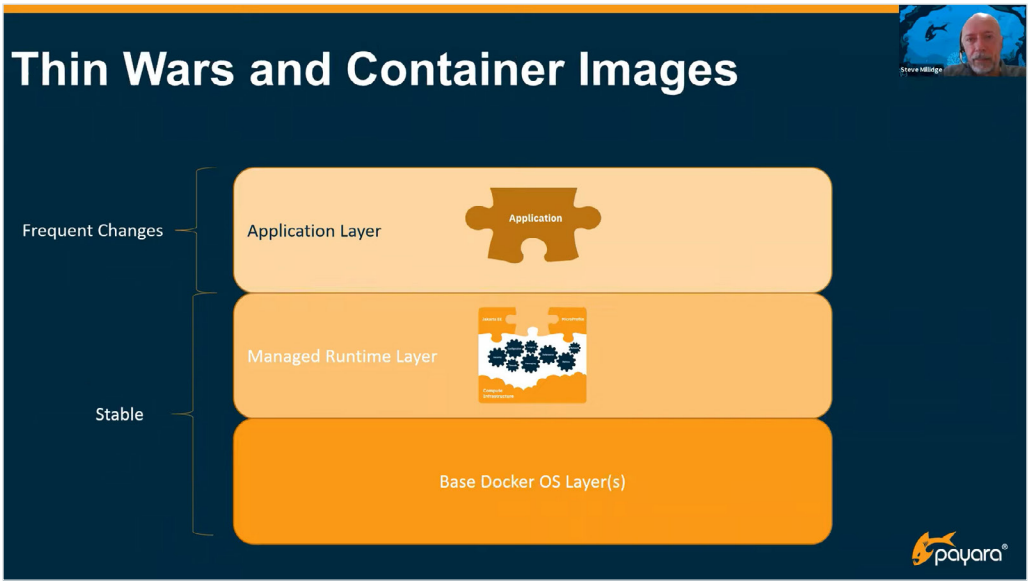
If someone does come to you in operations and says they've seen a critical CVE in the XYZ library, do we have that? Then you can talk to your runtime vendor to make sure that's fixed and you don't have to troll through running services and open Jar files to decide whether or not you have that.

Similarly, the developers of runtimes, like the Payara Team, support and track issues that need updates as well. You can have confidence based on backwards compatibility and things that you would be able to move to the next release of the runtime without huge concerns.

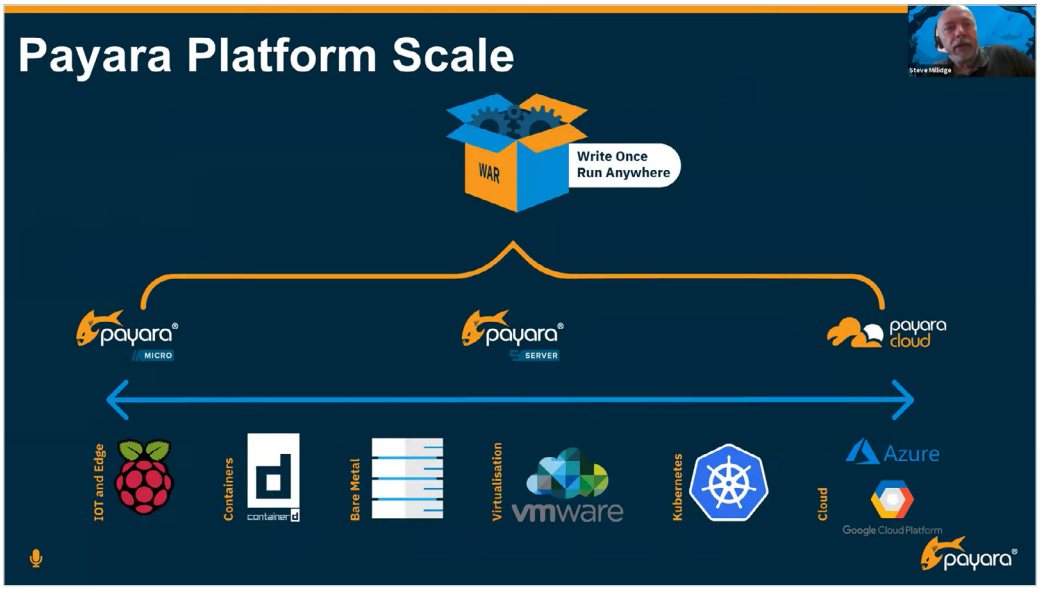
The developer or architect then only needs to manage the application layer itself, along with any third party dependencies the application requires. If you've got Jakarta EE, you would typically just have one Jakarta EE API dependencies, which makes it easier to manage the single API than pulling in half the internet to assemble a runtime.

Your application layer may change rapidly and more often than the Payara Platform or the Operating System base layer. So the layers help you when you need to make changes.

The other advantage you get from splitting the application layer away from the runtime and base layer is that if you do decide to run in a VM or bare metal environment, then you need to package and build a container image like Docker. These technologies are typically built in layers. So when you make a change and deploy it on the cloud for example, the technology is clever enough to only push the changes. If you can maintain a stable operating system layer and runtime layer and just frequently change the Thin War application which is only a few kilobytes in size, then the updates can be faster and less data is flying around. The development cycle can be much faster.



Another advantage is looking at how you run your application. If you separate the application from the application runtime, then your application runtime can take your application and run it on lots of services for you. Payara Platform allows you to write the code once, and run anywhere:



You don't have to assemble anything specific to run on Azure or Kubernetes. Payara Platform can take your WAR file written to Jakarta EE standards and deploy it on Payara Micro, and Edge device, run in a container, etc.

We're about to launch Payara Cloud, which is essentially a PaaS for Jakarta EE applications. You just take your WAR cloud and push it to the cloud and the application runs. It just takes the Thin War and uses the concept that you have a separately packaged application from the infrastructure.

Myth: The Java EE Deployment Model is Out of Date

Dismiss the Myth: Jakarta EE Deployment Model is Not Outdated - It Has Many Advantages!

There are so many advantages of the deployment model, achieved by separating the runtime and deployment as we've mentioned. Here's a quick summary of the advantages:

- No self assembly
- All pieces work together
- Versioned runtime
- Security
- Monitoring
- Health
- High Availability
- Scalability
- Focus on application (not infrastructure)

The myth that Jakarta EE deployment model is outdated is just that, a myth.

sales@payara.fish **+44 207 754 0481** **www.payara.fish**