How to Modernize Java EE Applications

Published 15 November 2021 - ID G00757828 - 43 min read

By Analyst(s): Matt Brasier

Initiatives: Application Architecture and Platforms for Technical Professionals

Java EE (previously J2EE) was once a leading standard for application development, but now organizations want to adopt cloud-native architectures to increase agility. Application technical professionals must decide when and how they will modernize their Java applications and architectures.

Overview

Key Findings

- The popularity of the Java EE specification has fallen significantly since its peak, and continues to drop. The specification's recent move from Oracle's control to the Eclipse Foundation Jakarta EE project, its roots in n-tier application architectures, and the establishment of Eclipse MicroProfile as an alternative standard, are all contributing to uncertainty about its long-term future.

- Organizations seeking agility and dexterity are looking for flexible technology stacks, rather than using a monolithic Java EE application server that includes many services that their application doesn't use.

- Enterprises that have already successfully modernized their Java EE-based applications have taken an incremental approach, extracting functions that benefit from increased agility while leaving the stable, slower-changing core intact.

- Contemporary frameworks (such as Spring Boot and Quarkus) allow organizations to reuse their established Java EE skills, tools and expertise in a more modern application architecture.

Recommendations

As an application technical professional modernizing Java EE applications, you should:

- Adopt an incremental approach by modernizing individual application components or modules, rather than whole applications.
Problem Statement

For many years, Java Platform, Enterprise Edition (Java EE) was one of the most popular frameworks for developing server-side application components and traditional web applications. Java applications are built for and deployed using platforms that implement a set of shared/open-standard specifications that define commonly used application platform features. Many of the technologies that make up the Java EE specification (such as server-side generation of HTML using Jakarta Server Faces [JSF]) are now legacy and should not be used for new development work. Working on applications that make use of these legacy technologies makes it much harder for your developers to adopt modern agile delivery approaches.

This raises the following question:

How do I modernize my Java EE applications to improve their development agility?

This research answers that question by telling you if, when and how to move your applications from Java EE to a more modern cloud-native architecture.
The Gartner Approach

Modernization of your applications should be driven by specific business goals, rather than a technical desire to move from a legacy platform. A modernization effort involves significant planning, effort and costs. You must target your modernization effort at application capabilities where you can define and measure the business benefit you will receive.

When you are ready to modernize your Java EE application, follow these six steps:

- Prework: Identify business goals and constraints
- Step 1: Evaluate and classify your application
- Step 2: Define your application architecture’s future state
- Step 3: Modernize the most impactful components
- Follow-Up: Continually modernize

Modernization is not a one-size-fits-all task. You must design your modernization to deliver business value, rather than purely technical benefits. The primary benefit of modernizing away from Java EE is to increase the business agility of your application. Some application capabilities will benefit more from agility than others.

Following the guidance in this framework will result in different outcomes for different components. The right outcome for a component could be:

- Remaining on or upgrading to Java EE 8
- Upgrading to Jakarta EE 9
- Rearchitecting to use frameworks that overlap the features available from Java EE, such as Spring Boot or microprofile-based frameworks such as Quarkus, IBM WebSphere Liberty or Helidon
  - For more details of these frameworks and the impact of selecting them, see Step 3.
You will be more successful in your modernization approach if you have gained experience implementing modern architectures — such as mesh app and service architecture (MASA) — on “greenfield” development projects. Before you make disruptive changes to your applications, you must already have adopted modern software development practices and processes (see Solution Path for Continuous Delivery With DevOps). This experience will give you a clearer view of your target state, while smoothing out the bumps and barriers experienced by an organization’s early adopters of agile processes and modern architecture.

This guidance framework provides a methodology for evaluating a single Java EE application, identifying the components within that application that make good targets, and then modernizing them. However, modernization is not the end of your journey. By adopting modern development practices and processes, you should enter into a continuous feedback loop of improvement of both the applications themselves and your modernization approach.

**The Guidance Framework**

When modernizing your Java EE application, you should execute the six steps outlined in Figure 1.
Figure 1: The Guidance Framework

The Guidance Framework

Prework

Before you start to modernize your application, you must be clear about the reason for modernizing it. It would be nice if all our applications could be brought, and kept, in line with the latest architectural principles, but modernization costs money, so you must target your modernization efforts to achieve a return on investment. The Gartner TIME (tolerate, invest, modernize and eliminate) framework provides a mechanism for classifying applications. Application portfolio management is outside the scope of this research, but for more details, see Use TIME to Engage the Business for Application and Product Portfolio Triage.

During your prework, you should:

- Use pace layers to evaluate demand for change
Use Pace Layers to Identify Demand for Change

Gartner recommends a model called the Pace-Layered Application Strategy to identify areas where the pace of change impacts your business goals (see What Is Gartner’s Pace-Layered Application Strategy and Why Should You Use It?). Because modern architectures are designed to accelerate the pace of change of an application, applications that need to change regularly are more likely to benefit from modern architecture.

The pace layer model defines three categories of application (as shown in Figure 2):

- **Systems of innovation**: These are your prototypes and experimental projects. If they are successful, they will become your next generation of systems of differentiation. They require the fastest pace of change while you quickly try out new approaches and techniques.

- **Systems of differentiation**: These are the applications that differentiate you from your competitors, giving you a unique brand, process or experience. Systems of differentiation allow you to set yourself apart from the crowd by being faster, better, more personalized or cheaper, and they are often implemented as homegrown or heavily customized applications. You must regularly update your systems of differentiation to maintain your advantage and stand out.

- **Systems of record**: These are business applications necessary for the execution of your business, and for which you do not derive a business benefit by taking a unique approach. This category includes common back-office functionality such as human capital management (HCM) and finance processing, but may also include some industry-specific technologies.
Figure 2: Use Pace Layers to Identify Demand for Change

**Use Pace Layers to Identify Demand for Change**

When identifying targets for modernization, use the concepts of the Pace-Layered Application Strategy model to identify the components within your application that will derive a business benefit from having an architecture that enables fast-paced change.

**Identify the Business Drivers for Modernization**

You may think that application modernization is a purely technical affair, that your legacy architecture is getting out of date and that you should move to something newer. However, technical drivers — such as the cost of support, standardizing on platforms and approaches, or updating your skills — are unlikely to deliver sufficient value to give a return on investment. Your primary drivers for modernization should be based on the agility that a more modern architecture can deliver to the business. You must identify the components within your application that could deliver business advantages if they were moved to a more modern architecture.

These factors are illustrated in Figure 3.
Set concrete goals based on these business drivers (such as “increase customer retention by 2%”), so that you can deliver measurable returns on investment. These will help you prioritize which components to modernize first and how much change they will need to meet your business objectives.

Updating only your runtime platform will not deliver the business benefits you are looking for. How much you will need to update will depend on how far up the cloud-native tower you are looking to get, and where you currently are.

At minimum, you should:

- Mature your agile and DevOps development practices
- Update your development and build tools

**Mature Your Agile and DevOps Development Practices**

Source: Gartner

75031_C

---

**Figure 3: Drivers for Modernization**

<table>
<thead>
<tr>
<th>Drivers for Modernization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>New Capabilities</strong></td>
</tr>
<tr>
<td>Process Automation</td>
</tr>
<tr>
<td>Improved Stability</td>
</tr>
<tr>
<td>Experiment More</td>
</tr>
<tr>
<td>New User Experiences</td>
</tr>
<tr>
<td><strong>Faster Changes</strong></td>
</tr>
<tr>
<td>Respond to Customers</td>
</tr>
<tr>
<td>Bug Fixes</td>
</tr>
<tr>
<td>Scaling</td>
</tr>
<tr>
<td>Market Changes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Supporting Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obsolescence</td>
</tr>
<tr>
<td>Skills</td>
</tr>
<tr>
<td>Recruitment</td>
</tr>
<tr>
<td>Standardizing</td>
</tr>
<tr>
<td>Support</td>
</tr>
</tbody>
</table>
Agility in software development is about being able to react to changing stakeholder needs quickly. Modern application architectures and frameworks are focused on enabling development agility by simplifying the development process, speeding up round-trip testing and improving testability. However, you will not see any benefits from these if your skills and processes do not enable you to be agile. Before you invest heavily in modernization, you should ensure you have a good level of maturity with agile development practices.

For more details of agile approaches, see Solution Path for Continuous Delivery With DevOps.

**Update Your Development and Build Tools**

Depending on the age of your application, you may be using one of several build technologies (such as Apache Ant, Gradle or Apache Maven). You may or may not be using a modern integrated development environment (IDE) because many application server vendors provide an IDE with extensions that make it easier to link to their application server (such as Red Hat JBoss Developer Studio or Oracle JDeveloper). In that case, your IDE is likely the same age as your application server.

To achieve your agility goals, you will need to update your development and build tools, including IDEs, testing frameworks, code quality frameworks and security scanning tools. Before selecting tools, you must define the scope of your build process, and this requires an understanding of your outer architecture and runtime platform (see below). A modern, cloud-native application platform is a system that takes a deliverable from developers and turns it into a set of running instances. You must decide what form that deliverable takes — is it a container image, source code or a compiled application executable?

For more details on building your delivery pipeline, see How to Architect Continuous Delivery Pipelines for Cloud-Native Applications.

**Step 1: Evaluate and Classify Your Application**

Successful modernization is a journey. The Prework section looked at the business outcome that defines your endpoint, but now you need to know the point you are starting from.

Use the following questions to evaluate the current status of your application:

1. What is the current support status of your application?
2. Do you have access to the source code, and can you build it?

3. What Java and Java EE versions does the application currently use?

4. What libraries and frameworks does the application currently run on?

5. Does the application make use of cloud-hostile or cloud-unfriendly patterns?

**What Is the Current Support Status of Your Application?**

Many aspects of an application and the platform it runs on affect your ability to maintain and support it effectively. This research focuses on only those related to the Java Virtual Machine (JVM) and Java EE application servers.

In September 2021, Oracle updated its Java Licensing to provide free access to bug fixes and patches for long-term support (LTS) for two years (the period between LTS releases). This reverses some of the most impactful changes made in April 2019, when Oracle changed how it licensed and supported the Java platform so that obtaining bug fixes and patches for Java releases over six months old required a subscription. The net result of these changes is that, if you are using Oracle Java releases, there are significant benefits to upgrading to LTS releases for production workloads.

For more details, see [Evaluate Your Options Now to Minimize the Impact of Costly Subscription Fees for Oracle Java SE Licensing](#).

Of more importance is the support status of your application server. The most commonly used Java EE application servers are IBM WebSphere Application Server, Oracle WebLogic Server and Red Hat JBoss Enterprise Application Platform. Each of these vendors outlines its support roadmap and offers several tiers of support. In addition, all three vendors offer agreements that allow you to continue to purchase limited support after the official end-of-life date for the platform, but such support is usually limited to critical production issues, rather than ongoing bug fixes.
When determining the support status of your application server, the key date you need to identify is the end-of-life date for the platform. This is the date from which you will no longer receive regular security patches or bug fixes. It is not easy to compare the end-of-life dates for different vendors (because they have different tiers with different levels of patch and security fix availability that do not neatly map to each other). However, Figure 4 shows a timeline of the availability and end of mainstream support dates for commonly used Java EE platforms. Releases of major versions of Java EE application servers are usually tied to releases of the Java EE specification.

**Figure 4: Support Life Cycle of Common Java EE Application Servers**

In September 2019, the Java EE specification transitioned to the Eclipse Foundation and became Jakarta Enterprise Edition (Jakarta EE). The Jakarta EE 8 specification is fully compatible with the Java EE 8 specification. The Jakarta EE 9 specification was released in December 2020. However, we have seen slow adoption of support for Jakarta EE 9 in commercial application servers. Support is stronger with open-source application servers such as WildFly, Open Liberty and GlassFish.
Jakarta EE 9 is not backward-compatible with Java EE 8, so rework will be needed to migrate your existing applications, if you plan to adopt it.

Do You Have Access to the Source Code, and Can You Build It?

If you do not have access to your application source code, your modernization will be limited to rebuilding it from scratch using its observed (and perhaps documented) behavior as a template. If you do have access to the source code, you need to make sure it builds. For most applications, the application code itself is not enough; you will need all the necessary libraries, dependencies, build tools and scripts. If you have the source code, you can at least use this as a definitive definition of what the application should do. And if you are lucky, you will be able to reproduce or replace any missing libraries or dependencies — but this will consume a significant part of your modernization budget.

What Java and Java EE Versions Does the Application Currently Use?

Java EE and Java SE have been around for a long time, and over that time, they have evolved significantly. It will be much simpler to modernize an application that is using Java EE 8 than one using J2EE 1.4 because you will be able to find modern equivalents for more of the features and frameworks from Java EE 8. An application that uses Java EE 7 or 8 is also more likely to run on an application server that can run in a containerized environment or on cloud infrastructure.

Applications that use versions of the Java Development Kit (JDK) or Java EE older than version 5 will require significant rearchitecting to modernize.

What Libraries and Frameworks Does the Application Currently Run On?

Java EE contains a wide set of specifications and capabilities, from core features such as Enterprise JavaBeans (EJBs), servlets and transaction management to less-used features such as Java Message Service (JMS) (messaging) and JavaMail (email). Therefore, you must determine which features of Java EE your application uses to ensure you know what features the platforms for your modernized application will need to offer.
You may be able to get this information by looking at the build dependencies (such as Apache Maven Project Object Model (POM), Apache Ant build.xml or Gradle file). However, it is likely your application just has a dependency on a higher-level Java EE artifact, rather than specifying individual components. When this is the case, you should check the imported package names used in source code to understand what features you are using. Java EE packages begin with the “javax” prefix, so this is a good place to start. For example, a source file that contains an import for “javax.jms” makes use of JMS. You can also get a good understanding of the dependencies that your application has by using static analysis tools such IBM Cloud Transformation Advisor, Red Hat Migration Toolkit for Applications and VMware Spring Migration Analyzer.

It is very rare for an application to use only features available from the Java SE and EE specifications. Most applications will make use of third-party libraries, and you must identify those libraries and their transitive dependencies too. You must also identify and catalog those dependencies, but in this case, you can get them by looking at your build dependencies. Be aware that the libraries you depend on will themselves have transitive dependencies, including on Java EE APIs, that you must be aware of. Your build tool may offer a way to produce a list of all dependencies (such as the mvn dependency:tree option), which provides an excellent starting point.

**Does the Application Make Use of Cloud-Hostile or Cloud-Unfriendly Patterns?**

The principles of modern application architecture are those of cloud-native architecture. One of your modernization goals is to move your application from a traditional n-tier architecture to a cloud-native one. There are some patterns or behaviors that applications can exhibit that either do not work in a cloud (or containerized) environment, or that will detrimentally affect your ability to adopt cloud-native approaches.

Assess whether your application uses any of these patterns or behaviors:

- **Multicast broadcasting** — This is a technique for sending one or many messages across a network. If you use this technique within your application, you should consider publish-subscribe topics or messaging queues as an alternative approach.

- **Embedded logging** — Log messages are written to local logs that are not part of a wider log management infrastructure. When modernizing your application, you should update your logging to write to standard out or to an external logging system.
- **Shared storage** — This storage is assumed to be available at specific locations and is shared between application components such as shared file storage, shared caches and so on. Remove direct file paths from your application, and use block storage where possible. Volume mounts are also an option, but they will require additional operational management. Adopt eventual consistency to shared state where possible.

- **Local host IP address** — The loopback IP address (127.0.0.1) is not compatible with many cloud or cloud-native platforms. Ensure your application can look up its own hostname and route traffic to itself that way.

- **Full trust code/root user** — Code running in a privileged mode with full access to system resources is not compatible with cloud approaches to security. Identify components in the application to see what specific system resources are required, and provide alternative mechanisms of access.

- **Embedded configuration details** — Application configuration details are embedded in the application and can only be set at install time. Update the application to externalize configuration and use a service discovery mechanism (such as environment variables or a configuration repository) to extract relevant configuration data from the external environment.

- **DNS/hostname dependencies** — These are hard-coded addresses for services or other components. Update the application to use a dynamic service discovery mechanism.

- **Use of atomicity, consistency, isolation and durability (ACID) transactions** — Java EE (and in particular, EJBs) are tightly coupled to ACID transaction semantics. Both distributed and local transactions will need consideration. Assess whether transactions are really needed or were just used because they were there for free. Use the saga pattern as an alternative to distributed transactions.

- **Legacy monitoring** — Monitoring of traditional Java EE applications often relies on the application server automatically exposing metrics via Java Management Extensions (JMX) and having a static topology. Use modern monitoring technology such as Prometheus. Change your monitoring objective from monitoring servers to monitoring overall service availability.
Step 2: Define Your Application Architecture’s Future State

Having established your starting point in Step 1, you must now establish an application architecture that delivers the required business agility and use the revise, rearchitect and rebuild approaches to deliver it.

Many organizations approach modernization as a one-size-fits-all endeavor. They select a target architecture, such as microservices or serverless functions, and then start trying to modernize their whole monolith to suit that architecture. A modern application architecture should embrace heterogeneity. Although there are advantages that come from your application teams not needing to know or support too many languages, frameworks, tools and architectures, you must balance these against picking the right tool or architecture for the job.

Mesh App and Service Architecture

Gartner recommends that modern applications are built using a MASA. This is a service-based approach in which your application is composed of a set of multigrained back-end services that are exposed as APIs. Small user interface components (mesh apps) consume these APIs to deliver the required functionality (see Figure 5).

For more details of MASA, see MASA: How to Create an Agile Application Architecture With Apps, APIs and Services.
An important aspect of MASA is that your back-end services are multigrained. They are not all microservices, but are sized and scoped to meet the business requirements for scalability and deployment velocity. Each service can be implemented using the most appropriate approach. You should have the fewest number of services needed to meet your goals for agility and scalability.

Not every component of your Java EE application needs to be modernized to the same extent and approach.

The Cloud-Native Tower
Modernization can be thought of as lifting your application components up a tower toward a truly cloud-native state. VMware defines a model of cloud readiness. By updating this to include cloud-hostile and adding the use of technologies or approaches to it (as in Figure 6), you can assess where your application component currently is and where it needs to move to.

**Figure 6: The Cloud-Native Tower**

- **Cloud-Hostile**
  - Local Storage
  - IP Multicast
  - IP Addresses

- **Cloud-Ready**
  - Dynamic Service Discovery and Routing

- **Cloud-Friendly**
  - XA Transactions
  - SOAP Interfaces
  - DNS Routing

- **Cloud-Resilient**
  - Horizontal Scalability
  - Uses Cloud HA Features
  - REST Interfaces

- **Cloud-Native**
  - API-First Design
  - Composable Services
  - Consumes Resources as Services
  - Web APIs

- **Anti-fragile**
  - Proactive Testing
  - Circuit Breakers
  - Dumb Pipes
  - Idempotent Operations

Source: Adapted from VMware/Pivotal
For each component (or set of dependent components), you must assess its current position on the tower and how far up the tower it makes sense to move it. Although you may initially think you want your whole application to be cloud-native, achieving this for every component will be very expensive and many components will deliver no additional business benefit from being truly cloud-native. Use the business drivers identified in the prework, and the existence of cloud-hostile patterns and technologies, to assess where your application currently is and where you will get the best business benefit for the effort expended in modernizing.

The Five Rs

The five Rs framework is commonly used when assessing an application for cloud migration. Each R represents an approach to application migration:

- Rehost
- Revise
- Rearchitect
- Rebuild
- Replace

When your goal is to modernize an application, the first (rehost) and last (replace) options are not applicable. This leaves you with three approaches you can take when modernizing your Java EE application.

Figure 7 illustrates the five approaches.
For each application component, you must decide whether to revise, rearchitect or rebuild it, depending on how many levels up the cloud-native tower it needs to move:

- **Revise**: Your goal is to change the components that surround the application without making significant changes to the application itself. For a Java EE application, this may involve using a cloud messaging service, rather than the built-in JMS capabilities of your application server, or perhaps deploying already modularized components in their own containerized runtimes.

- **Rearchitect**: When rearchitecting, you are prepared to make significant changes to your application architecture, but you still want to reuse application assets such as source code and build artifacts where possible. This could involve breaking a monolithic application down into smaller components, refactoring code or changing application components from EJBs to Spring or Contexts and Dependency Injection (CDI) beans.
Step 3: Modernize the Most Impactful Components

Your next task is to identify the components that you will modernize and separate them from your application. There are four aspects to implementing a modernization strategy:

- Apply a decomposition approach
- Modularize from the top down
- Decide whether to revise, rearchitect or rebuild each capability
- Select your target frameworks

Apply a Decomposition Approach

Decomposition is about identifying the boundaries around which you will break down your application into smaller components. There are three main strategies for decomposing an application: split, strangle and extend. Of these, the extend pattern leaves the original application intact and adds new functionality as modern services, which is not relevant if your goal is modernization of the existing application. But it does present an opportunity to establish and prove your new platform, frameworks, tools and processes while minimizing the impact on your core application.

Strangle
To implement the strangle pattern, identify the business function of your application that is holding you back the most because you cannot deploy it fast enough or scale it independently of the rest of your application. You should break this function out into its own service, then come back to the monolith and identify the next service that is holding you back. You repeat this process until you can no longer identify a component that is holding your business back. At that point, you either stop (if you have broken your application down enough) or apply the split pattern (if you need to break your application down further to meet development agility goals).

Split

In the split pattern, you break the application down into an initial set of services based on their business functionality. For example, an e-commerce application can be broken down into services that deal with check-out, search, inventory management, recommendations and comparisons. This approach exposes you to the risk that you break your application down into more components than is really necessary, leaving you to manage and govern an overly complex architecture.

The extend, strangle and split patterns are illustrated in Figure 8.
Modularize From the Top Down

To modernize your application components using a top-down approach, you should start by putting modern, consumer-centric, designed APIs in front of it. Then you start to break down the application internally into modules, first allowing each module to be developed individually but still requiring that they be combined into a monolith for deployment before gradually removing deployment time and runtime dependencies. Figure 9 illustrates the approach of modernizing from the top down.

For more about breaking down your application into components using a top-down approach, see Designing Services and Microservices to Maximize Agility.
Decide Whether to Revise, Rearchitect or Rebuild Each Capability

In Step 2, you selected an approach of revise, rearchitect or rebuild to update your application components to meet your business objectives. You must now implement that approach by updating and replacing the libraries and frameworks that your application uses and then making the code changes necessary to make the components work. Do not focus only on the programmatic API and functional differences when moving to your new architecture, because you will likely have to account for other significant changes, such as differences in quality-of-service and error-handling approaches.
Use a test-driven development approach when modernizing your application components, and ensure that you thoroughly test error scenarios, including networking and service failures.

Select Your Target Frameworks

Java EE applications are built on the Java EE framework, although they may also make use of other frameworks. When modernizing your application, you can move it to a different framework and will often be able to reuse some or all of the business-logic code when doing so. However, your new framework will need an equivalent for each service that you used from Java EE (or you will need to use a third-party library to supply that functionality). Unfortunately, it is not quite as simple as swapping out one messaging or logging framework for another. Different frameworks and libraries usually have very different nonfunctional characteristics (such as thread safety guarantees, transactional behavior or ordering characteristics). These changes often do not appear under optimal operating circumstances, but become very relevant when the system behaves differently under failure conditions.

When migrating to a new framework or library, ensure that you thoroughly test any changes in the failure modes of your application.

There are four main approaches to modern service-based Java applications that allow you to reuse some part of your Java EE investment. In addition, you may want to consider adopting an approach that does not leverage your Java EE investment.

The approaches to consider are:

- Remain on or upgrade to Java EE 8
- Upgrade to Jakarta EE 9
- Migrate to Eclipse MicroProfile-based frameworks such as Quarkus, Liberty and Helidon
Migrate to Spring Boot and Spring Framework 5

The approach you pick for a particular feature or service should be driven by your modernization goals. How far up the cloud-native tower (see Figure 6) are you aiming and are you going to revise, rearchitect or rebuild your components? This is determined by how different the frameworks are from your existing Java EE platform, the particular features of Java EE you are using, and how similarly the target framework offers those features. You must also take into account your existing skill set (such as your experience of using the Spring Framework) and your existing vendor ecosystem (such as whether you use VMware Tanzu or Red Hat OpenShift environments, or cloud platforms such as those of Amazon Web Services or Microsoft [Azure]).

Table 1 shows core features of the Jakarta EE platform and the equivalents for Eclipse MicroProfile and Spring Boot.
### Table 1: Core Features of Jakarta EE Platform and Equivalents for Eclipse MicroProfile and Spring Boot

*(Enlarged table in Appendix)*

<table>
<thead>
<tr>
<th>Feature</th>
<th>Jakarta EE 9</th>
<th>Eclipse MicroProfile 4</th>
<th>Spring Boot</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/HTTPS</td>
<td>Java net</td>
<td>Java net</td>
<td>Java net</td>
</tr>
<tr>
<td>Transactions</td>
<td>Java Transaction API (JTA)</td>
<td>Non-equivalent</td>
<td>JTA, Spring Transactions</td>
</tr>
<tr>
<td>Interoperability</td>
<td>JAX-RR (JPA), Jakarta interoperability</td>
<td>Non-equivalent</td>
<td>Non-equivalent</td>
</tr>
<tr>
<td>Persistence</td>
<td>Java Persistence API (JPA)</td>
<td>Non-equivalent</td>
<td>Spring Data, Spring ORM integration</td>
</tr>
<tr>
<td>Messaging</td>
<td>Java Message Service (JMS)</td>
<td>Non-equivalent</td>
<td>Spring Core, Spring Template</td>
</tr>
<tr>
<td>Service Location</td>
<td>Java Naming and Directory service (JNDI)</td>
<td>Non-equivalent</td>
<td>Java (different from Java EE)</td>
</tr>
<tr>
<td>Email</td>
<td>Jakarta Mail (same as Java EE)</td>
<td>Non-equivalent</td>
<td>JavaMail</td>
</tr>
<tr>
<td>XML Processing</td>
<td>Java API for XML Processing (JAXP), Java architecture for XML Binding (JAXB)</td>
<td>Non-equivalent</td>
<td>Java API, JAXB (same as Java EE)</td>
</tr>
<tr>
<td>Resource Adapters</td>
<td>Java Cryptography Architecture (JCA)</td>
<td>Non-equivalent</td>
<td>Java Cryptography</td>
</tr>
<tr>
<td>Security</td>
<td>Java Authentication and Authorization Service (JACC)</td>
<td>Non-equivalent</td>
<td>Spring Security (different from Java EE)</td>
</tr>
<tr>
<td>SDAP Web Services</td>
<td>Java API for SDAP web services (JAX-WS)</td>
<td>Non-equivalent</td>
<td>Spring Web Services (different from Java EE)</td>
</tr>
<tr>
<td>Batch</td>
<td>Jakarta Batch API (same as Java EE)</td>
<td>Non-equivalent</td>
<td>Spring Batch, Spring Cloud Data Flow (different from Java EE)</td>
</tr>
<tr>
<td>Management</td>
<td>Java Management Extensions (JMX)</td>
<td>Non-equivalent</td>
<td>JMX, Actuator Management, Indicators</td>
</tr>
<tr>
<td>JSON Processing</td>
<td>WS-Addressing</td>
<td>Java RESTful Web services (JAX-RS)</td>
<td>Java RESTful Web services (JAX-RS)</td>
</tr>
<tr>
<td>REST Web Services</td>
<td>Java API for RESTful Web services (JAX-RS), Jersey</td>
<td>Non-equivalent</td>
<td>Java REST, Spring Web, Spring Web services (different from Java EE)</td>
</tr>
<tr>
<td>Service Configuration</td>
<td>Non-equivalent</td>
<td>MicroProfile Client</td>
<td>Spring Cloud Config (different from Java EE)</td>
</tr>
<tr>
<td>Component Lifecycle</td>
<td>Bean wiring extensions (BWI)</td>
<td>Non-equivalent</td>
<td>CDI (different from Java EE)</td>
</tr>
</tbody>
</table>

**Color key:**
- Green = same as Java EE; amber = different from Java EE; red = no equivalent

**Remain on or Upgrade to Java EE or Jakarta EE 8**

**Applies to:** Revise, rearchitect

**Achieves:** Up to cloud-friendly

**Level of effort:** Low
Application servers that support Java EE 8 will continue to be supported by vendors until at least 2025. There is no urgency about moving to a new framework, and you can modernize your application by modularizing your existing codebase and splitting your monolith. However, Java EE was designed for monolithic n-tier applications, and Java EE applications usually rely on ACID transactions. Java EE also carries a significant amount of legacy baggage, either explicitly (such as the JSF framework for server-side generation of HTML) or implicitly (such as how its original design goals of being compatible with the Common Object Request Broker Architecture [CORBA] continue to influence its programming model). The highest level of the cloud-native tower you will achieve is cloud-friendly. Jakarta EE 8 is fully compatible with Java EE 8, allowing you to move your applications from Java EE 8 to Jakarta EE 8 with only minor configuration changes.

**Upgrade to Jakarta EE 9**

**Applies to:** Revise

**Achieves:** Up to Cloud-friendly

**Level of effort:** Low to medium

You may feel that the simplest thing to do is to continue to use the Jakarta EE specification as it evolves through the Eclipse Foundation. The Jakarta EE 9 specification was released in December 2020.

Jakarta EE 9 does not aim to be backward-compatible with Java EE 8, and due to licensing restrictions, many packages have been renamed, requiring you to update and rebuild your applications. A design goal of Jakarta EE 9 is to drop much of the legacy baggage of Java EE 8 described above, but the heritage of Java and Jakarta EE still holds it back, compared with more modern approaches. The large vendors (such as Oracle, IBM and Red Hat) that were the center of the Java EE community are working on their next generations of cloud-native application platforms and are less focused on their legacy application servers. The release of Jakarta EE 9 went largely unnoticed by the enterprise Java development community, and appears not to have been widely adopted. Jakarta EE 9 should be considered an option for components where you do not need additional agility but you want to ensure you are using a platform that will continue to be supported for the next five years.

**Migrate to Eclipse MicroProfile**

**Applies to:** Rearchitect, rebuild
Achieves: Up to cloud-native

Level of effort: Medium to high

During 2016 and 2017, Oracle noticeably reduced its focus on updating the Java EE specification, while it concentrated on becoming more of a cloud platform. The enterprise Java community came together to create a set of standards that can be used in enterprise Java applications that use a microservices architecture. This effort became Eclipse MicroProfile. MicroProfile implementations are available as traditional application servers (such as WildFly), as well as in more modern frameworks (such as Quarkus). Many MicroProfile implementations can run either on a standard Kernel-based Virtual Machine (KVM) such as the Oracle JDK or OpenJDK, or on GraalVM (see Note 1).

MicroProfile does not aim to be compatible with the Java EE specifications, although it does reuse some of the standards that make up Java EE, such as CDI. In other areas, it offers its own standards, such as MicroProfile Config.

If you do not already use the Spring Framework, migrating to MicroProfile reuses more of the skills and artifacts from your existing Java EE application than a move to Spring Boot. However, it will still require a significant amount of work. With Jakarta EE 9 now also being governed by the Eclipse Foundation, a move toward convergence between the Jakarta EE and MicroProfile standards is possible.

Migrate to Spring Boot and Spring Framework 5

Applies to: Revise (if already using Spring Framework), rearchitect, rebuild

Achieves: Up to cloud-native

Level of effort: High

The Spring Framework was developed as an easier-to-use alternative to Java EE and EJBs in 2002, and has continued to evolve and expand in scope. Over the past two decades, it has competed to innovate, simplify and equalize the development of enterprise Java applications.
Spring Boot is a framework that brings together popular parts of the Spring Framework. Spring Boot relies heavily on the use of “sensible default” values to allow you to rapidly bring features into your application without complex configuration. Rather than building your application and then deploying it into an application server (that offers a set of common capabilities), you compile those common capabilities into your application as libraries. Spring Boot can run on a traditional JVM or on GraalVM (see Note 1).

Adoptions of Spring Boot and Spring Framework 5 vary in complexity, depending on whether your existing applications are already making use of the Spring Framework or whether they are pure Java EE applications.

**Follow Common Migration Paths**

There are many target frameworks and libraries that you can migrate to in order to modernize components of your Java EE application. However, the modernization process will be easier if you break it down into a sequence of steps and reuse skills and artifacts. To help with this, Gartner has identified some of the common migration paths that we see clients adopting, based on their starting application server.

---

**Common Migration Paths for JBoss Application Server**

Many organizations that are currently operating applications running on JBoss are targeting Red Hat’s Quarkus framework as their future cloud-native Java runtime. Recognizing the increasing popularity of Kubernetes as a container management platform, Quarkus aims to offer a modern developer experience for building Kubernetes-native Java applications. Quarkus has many similarities to Spring Boot, while offering a set of frameworks more familiar to developers with a history of creating Java EE applications (particularly if they used the Red Hat JBoss application server). It comprises a set of frameworks (some standards-based and others not) that expose different programming models (such as Eclipse MicroProfile-based microservices, reactive programming, REST-based services and integration components) that can be built into your application as required.
Quarkus is still in the early phases of its life cycle. As of September 2021, Gartner has only talked to a small number of clients that are using Quarkus in production environments, although many clients have identified it as their future target platform. The typical migration path to Quarkus involves migrating your application from the Java EE standard to MicroProfile on WildFly or JBoss EAP first, and then moving it to Quarkus as a second step. It is common for applications also to move from running on virtual machines to OpenShift during this migration effort. Alternatively, if you have selected Spring Boot as a target runtime for new Java applications, you may wish to rebuild parts of your application in Spring Boot.

Figure 10 illustrates these migration paths.

**Figure 10: Common JBoss Modernization Roadmaps**

Common Migration Paths for Oracle WebLogic Server

VM = virtual machine
Oracle recently released Verrazzano as a container-based runtime intended to give customers a common platform for applications that run either on-premises or in the cloud. Verrazzano supports several application runtimes, including WebLogic Server and Helidon (a MicroProfile runtime from Oracle that is similar to Red Hat's Quarkus). This gives you a potential target stack of Helidon on Verrazzano, which will give you a good migration path to Oracle Cloud. Note that because WebLogic does not support the MicroProfile specification, you must move your application to Helidon and refactor it to use MicroProfile at the same time, rather than as sequential steps. An alternative is to migrate your application from WebLogic to JBoss and then target Quarkus, but migration from WebLogic to JBoss is complex if your application uses many third-party libraries or relies heavily on the default behaviors of WebLogic. A third option is to rewrite your application using Spring Framework and Spring Boot, but this will likely involve rebuilding significant parts.

Figure 11 illustrates these possible paths.
VM = virtual machine

**Common Migration Paths for IBM WebSphere Application Server**

IBM WebSphere Application Server (WAS) receives continuous updates, with much of the innovation and support for new architectures is happening on the open-source IBM WebSphere Liberty runtime that is included with all editions of WAS. IBM also owns Red Hat, and so has three Java EE application servers in its portfolio: WAS, Liberty and JBoss. IBM also has Quarkus, through Red Hat, giving many possible migration paths and future directions. The most typical modernization path for WebSphere applications is to Liberty. Spring Boot is still the most popular framework for modern Java applications, and remains an option for components where rebuilding is a viable option.

Figure 12 illustrates these possibilities.
Follow-Up: Continually Modernize

Getting your application components up and running on a modern platform and framework using agile practices and processes is the start, not the end, of a journey. The goal of modern application architectures is to enable faster and more regular change, which helps you continually modernize. To get the most out of your new architecture, you must continually monitor each component (not only from a performance monitoring point of view, but also to ensure that it is meeting your architectural goals) and modernize as required.

Continually Modernize
Your business requirements will not remain static. Components that are your systems of differentiation today may become your systems of record tomorrow, reducing their cadence of change. New frameworks, platforms and architectures may emerge that allow you to meet new business goals in better ways. You must continually assess whether your application is meeting your business goals and using the most appropriate frameworks and platforms.

Do not let the technical debt of outdated frameworks and architectures build up. Treat modernization debt the same as any other form of technical debt by regularly reviewing and addressing it.

Regularly look for new hot spots in your application where a particular service or feature is not delivering the agility that the business requires. Use the strangler pattern to break the service or feature out into a new service with a more appropriate architecture and set of frameworks. See A Primer on Technical Debt for more details of how to manage the buildup of technical debt.

Risks and Pitfalls

Modernizing your application is a risk-reward balancing act. To maximize your risk and minimize your reward, you must be aware of the following risks and pitfalls:

- Treating modernization as a one-off effort
- Failing to retire applications when appropriate
- Adopting new technologies but not modernizing

Treating Modernization as a One-Off Effort

Organizations often view modernization of an application as a discrete project. Once it is complete, the team moves on to a new set of work, and the application enters a cycle of limited-to-no maintenance. This risks an application accruing technical debt once again.
To mitigate this risk, ensure that application performance monitoring is in place to quickly highlight any performance, scalability or capacity issues with the application. Manage the application as a product, rather than as a project, and regularly invest in reducing technical debt. Regularly review your delivery cadence to ensure that it is matched to the business need for change. Ensure there is an ongoing maintenance program to keep the underlying infrastructure up-to-date, and ensure periodic reviews of the application are undertaken.

**Failing to Retire Applications When Appropriate**

Organizations can be tempted to keep an application in production for as long as possible, rather than retire it and bring in a new solution that better meets business needs.

To mitigate this risk, fully analyze an application and evaluate the operational, technical and business costs of not retiring it. Does the application require a dedicated operations team or highly specialized skill sets for maintenance? Such applications are candidates for a rebuild or for replacement by an off-the-shelf SaaS or a commercial off-the-shelf solution, rather than for revision or rearchitecting.

**Adopting New Technologies but Not Modernizing**

Some modernization efforts (especially those that treat modernization as an entirely technology-focused endeavor) will make changes to the technology stack but not the architecture. This is not modernization because it does not reflect the fact that changes in how you deliver modern software are heavily dependent on modernizing the architecture. Deploying your n-tier monolith application into a Kubernetes cluster does not make it more agile, nor does moving your code to Spring Boot without changing your architecture.

To mitigate this risk and get the benefits of a modern application architecture, you must identify where your old architecture does not meet your needs and apply architectural changes targeted at remedying those shortcomings. You cannot achieve modernization by adopting the latest technologies without changing your architecture. For more details on adopting a container-based approach, see [Solution Path for Implementing Containers and Kubernetes](#).

**Evidence**


2. [WebSphere Application Server — Support Lifecycle FYI](#), IBM.

3. [Oracle Information-Driven Support](#), Oracle.
4 Red Hat Application Services Product Update and Support Policy, Red Hat.

5 Replatforming: Moving Custom Apps to the Cloud, VMware.

**Note 1: Oracle GraalVM Enterprise**

Oracle GraalVM Enterprise is a modern application runtime able to execute Java code, as well as JavaScript, Python, Ruby, R, Scala, Groovy, Kotlin, Clojure, C and C++. It can execute Java bytecode using a traditional just-in-time compilation approach, but is also able to compile a Java application into a native executable. It is popular with vendors that are creating Java runtimes optimized for deployment in a containerized environment (such as Red Hat with Quarkus and VMware Tanzu with Spring Framework 5), due to its smaller memory footprint and faster startup times, compared with OpenJDK. However, it should be noted that performance benchmarks of GraalVM place its performance overall as slightly lower than OpenJDK for long-running application processes.

GraalVM is still in the early phases of its life cycle and is likely to become the runtime of choice for the future generation of enterprise Java applications. In its current state, you should consider GraalVM an option for situations where startup performance is a critical priority (such as when writing serverless functions), rather than as a general replacement for OpenJDK. If you plan to adopt GraalVM as the runtime for components in your Java application, ensure you have the necessary level of support from your vendors.

GraalVM is open-source (see GraalVM Homepage), with a commercial version (GraalVM Enterprise Edition) available from Oracle (see Oracle GraalVM Enterprise Edition 20.0). Red Hat provides a distribution of GraalVM called Mandrel, which it supports as a Java runtime for Quarkus.

**Document Revision History**

A Guidance Framework for Modernizing Java EE Applications - 29 April 2020

---

**Recommended by the Author**

Some documents may not be available as part of your current Gartner subscription.

MASA: How to Create an Agile Application Architecture With Apps, APIs and Services

How to Modernize Your Application to Adopt Cloud-Native Architecture

Guidance Framework for Modernizing Microsoft .NET Applications
<table>
<thead>
<tr>
<th>Feature</th>
<th>Jakarta EE 9</th>
<th>Eclipse MicroProfile 4</th>
<th>Spring Boot</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/HTTPS</td>
<td>Java.net</td>
<td>Java.net</td>
<td>Java.net</td>
</tr>
<tr>
<td></td>
<td>[Same as Java EE]</td>
<td>[Same as Java EE]</td>
<td>[Same as Java EE]</td>
</tr>
<tr>
<td>Transactions</td>
<td>Java Transaction API (JTA)</td>
<td>No equivalent</td>
<td>JTA, Spring Transactions</td>
</tr>
<tr>
<td></td>
<td>[Same as Java EE]</td>
<td></td>
<td>[Different from Java EE]</td>
</tr>
<tr>
<td>Interoperability</td>
<td>RMI-IIOP, IDL, Jakarta Interoperability</td>
<td>No equivalent</td>
<td>No equivalent</td>
</tr>
<tr>
<td></td>
<td>[Same as Java EE]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Persistence</td>
<td>Java Persistence API (JPA)</td>
<td>No equivalent</td>
<td>Spring Data, Spring ORM Integration</td>
</tr>
<tr>
<td></td>
<td>[Same as Java EE]</td>
<td></td>
<td>[Different from Java EE]</td>
</tr>
<tr>
<td>Messaging</td>
<td>Java Message Service (JMS)</td>
<td>No equivalent</td>
<td>Spring JmsTemplate</td>
</tr>
<tr>
<td></td>
<td>[Same as Java EE]</td>
<td></td>
<td>[Different from Java EE]</td>
</tr>
<tr>
<td>Service Location</td>
<td>Java Naming and Directory Interface (JNDI)</td>
<td>CDI</td>
<td>Spring Core</td>
</tr>
<tr>
<td></td>
<td>[Same as Java EE]</td>
<td>[Different from Java EE]</td>
<td>[Different from Java EE]</td>
</tr>
<tr>
<td>Email</td>
<td>Jakarta Mail</td>
<td>No equivalent</td>
<td>JavaMail</td>
</tr>
<tr>
<td></td>
<td>[Same as Java EE]</td>
<td></td>
<td>[Same as Java EE]</td>
</tr>
<tr>
<td>Feature</td>
<td>Jakarta EE 9</td>
<td>Eclipse MicroProfile 4</td>
<td>Spring Boot</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>XML Processing</td>
<td>Java API for XML Processing (JAXP), Java Architecture for XML Binding (JAXB) [Same as Java EE]</td>
<td>No equivalent</td>
<td>JAXP, JAXB [Same as Java EE]</td>
</tr>
<tr>
<td>Resource Adaptors</td>
<td>Java Cryptography Architecture (JCA) [Same as Java EE]</td>
<td>No equivalent</td>
<td>No equivalent</td>
</tr>
<tr>
<td>Security</td>
<td>Java Authentication and Authorization Service (JAAS) [Same as Java EE]</td>
<td>MicroProfile JWT Authentication [Different from Java EE]</td>
<td>Spring Security [Different from Java EE]</td>
</tr>
<tr>
<td>SOAP Web Services</td>
<td>Java API for XML Web Services (JAX-WS) [Same as Java EE]</td>
<td>No equivalent</td>
<td>Spring Web Services [Different from Java EE]</td>
</tr>
<tr>
<td>Batch</td>
<td>Jakarta Batch API [Same as Java EE]</td>
<td>No equivalent</td>
<td>Spring Batch, Spring Cloud Data Flow [Different from Java EE]</td>
</tr>
<tr>
<td>Management</td>
<td>Java Management Extensions (JMX) [Same as Java EE]</td>
<td>MicroProfile Health Check, MicroProfile Metrics, MicroProfile OpenTracing [Different from Java EE]</td>
<td>JMX, Actuator Management Endpoints [Same as Java EE]</td>
</tr>
<tr>
<td>Feature</td>
<td>Jakarta EE 9</td>
<td>Eclipse MicroProfile 4</td>
<td>Spring Boot</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>JSON Processing</td>
<td>JSON Processing (JSON-P), JSON Binding (JSON-B) [Same as Java EE]</td>
<td>JSON-P, JSON-B [Same as Java EE]</td>
<td>JSON-P, JSON-B [Same as Java EE]</td>
</tr>
<tr>
<td>REST Web Services</td>
<td>Java API for RESTful Web Services (JAX-RS), Restful-WS [Same as Java EE]</td>
<td>MicroProfile OpenAPI, MicroProfile REST Client, JAX-RS [Different from Java EE]</td>
<td>JAX-RS, Spring Web, Spring WebFlux [Same as Java EE]</td>
</tr>
<tr>
<td>Service Configuration</td>
<td>No equivalent</td>
<td>MicroProfile Config [Different from Java EE]</td>
<td>Spring Cloud Config [Different from Java EE]</td>
</tr>
<tr>
<td>Component Life Cycle</td>
<td>Enterprise JavaBeans (EJB), CDI [Same as Java EE]</td>
<td>CDI [Different from Java EE]</td>
<td>Spring Framework [Different from Java EE]</td>
</tr>
</tbody>
</table>

Color key: green = same as Java EE; amber = different from Java EE; red = no equivalent

Source: Gartner