



# Jakarta EE for Spring Boot Developers: Mapping Concepts and Paradigms



# Contents

Guide Updated: **June 2024**

<b>Introduction</b> .....	<b>1</b>
<b>Jakarta EE vs. Spring Boot</b> .....	<b>1</b>
<b>Core Jakarta EE Specifications</b> .....	<b>2</b>
Jakarta CDI .....	2
Core Injection Annotation .....	2
Common Ground .....	3
Key Differences .....	3
CDI's Core Scopes .....	3
Qualifiers vs. Autowiring .....	4
Philosophical Takeaway .....	4
Jakarta REST (RESTful Web Services): Building APIs .....	4
Mapping Requests .....	4
Philosophical Difference .....	5
Jakarta Persistence (Java Persistence API): Managing Data .....	5
<b>Philosophical Differences in Practice</b> .....	<b>6</b>
<b>Conclusions</b> .....	<b>6</b>

## Introduction

Are you a Spring Boot developer? Ever wondered what J2EE, Java EE or Jakarta EE is or how it compares to Spring Boot? If you answered yes to any of these questions, then it's time to expand your theoretical knowledge of the Java Platform. This quick guide aims to bridge your existing knowledge with the core principles that make up Jakarta EE applications. The focus of this document is helping you understanding how Jakarta EE components and philosophies translate from your familiar Spring framework background.

Importantly, this is not a step-by-step Spring-to-Jakarta EE migration guide. Instead, it is designed to support the building of a theoretical foundation. Understanding Jakarta EE's principles will help you make informed architectural decisions when choosing the right tool for your next project. You will also become a better Spring developer by knowing how the core Spring constructs are implemented elsewhere. Let's get started.

## Jakarta EE vs. Spring Boot

Jakarta EE is a collection of abstract specifications that together form a complete solution for building end-to-end, multi-tier enterprise applications. Jakarta EE is broken down into profiles, the closest to Spring Boot being the Web Profile, which is a sub-collection of the platform specifications geared towards building web applications.

While both Jakarta EE and Spring Boot offer comprehensive constructs for building modern enterprise Java applications, they approach core functionalities with different philosophies. Let's look at some of their key areas to understand how they conceptually differ:

- **Foundation: Standards vs. Opinion** Jakarta EE centres around a set of abstract specifications (Jakarta REST, Jakarta CDI, etc.). This approach provides flexibility in choosing implementations and ensures vendor independence. Spring Boot, on the other hand, favours convention over configuration and delivers a more opinionated framework experience. It comes with a comprehensive set of integrated components, making it ready to use out of the box.
- **Scope:** Jakarta EE encompasses a wide array of technologies for various aspects of enterprise application development. Spring Boot has a strong focus on web application development and microservices, while also gradually expanding its support into other areas.
- **Ecosystem:** Jakarta EE has a mature and standardised ecosystem. Spring Boot boasts a vast and rapidly evolving ecosystem with rich tooling and libraries for a wide array of scenarios.

## Core Jakarta EE Specifications

There are core specifications on the Jakarta EE platform that will be part of nearly every application you develop. This section highlights these APIs and their Spring Boot equivalents, along with some examples.

### Jakarta CDI

Jakarta CDI (Contexts and Dependency Injection) is the closest equivalent to Spring's core IoC (Inversion of Control) container. Key concepts to compare include:

#### Core Injection Annotation

##### CDI (@Inject)

Standard: Part of the JSR-330 specification (Dependency Injection for Java).

Primary Mechanism: The core way to inject dependencies in CDI managed beans.

Flexibility with Qualifiers: Often used in conjunction with custom qualifier annotations (@Secure, @Primary, etc.) for fine-grained selection when multiple beans of the same type exist.

##### Spring (@Autowired)

Spring-Specific: Belongs to Spring's dependency injection framework.

Autowiring: The most common way to inject beans. Primarily autowires beans by type, automatically resolving dependencies if a single bean of a matching type exists within the Spring context.

#### Additional Annotations:

@Component: The baseline annotation indicating a Spring-managed bean.

@Service, @Repository, etc.: Provide more semantic meaning (service layer, data access layer) but function largely the same way under the hood.

@Qualifier: For disambiguation in cases similar to CDI's qualifiers.

## Common Ground

Purpose: Both serve to declare and resolve dependencies in a typesafe way, promoting loose coupling and testability in your code.

## Key Differences

Standards-Based vs. Framework-Specific: CDI's approach adheres to a Java standard, while Spring provides its own solution.

Emphasis on Qualifiers vs. Autowiring: CDI leans heavily on qualifiers, Spring favours simpler autowiring by type.

## CDI's Core Scopes

A scope is a well-defined life cycle within which an instance of a bean can exist. Jakarta CDI has a number of built-in scopes for common application requirements. The table below lists these, along with their rough Spring equivalents.

CDI Scope	Description	Spring Equivalent
@ApplicationScoped	Live as long as the application, shared across users.	singleton (default)
@RequestScoped	Exist for a single HTTP request.	request
@SessionScoped	Tied to a specific user's session.	session
@ConversationScoped	Live across multiple requests in a logical conversation.	Complex in Spring, may require custom management

Jakarta CDI has the @Dependent pseudo-scope that creates a new bean instance for each request. This is roughly equivalent to Spring's prototype scope.

## Qualifiers vs. Autowiring

### CDI Qualifiers

They provide fine-grained control when multiple beans of the same type exist.

You create custom annotations (@Fast, @Secure, etc.) and apply them both at the injection point and the bean definition.

CDI selects the bean whose qualifier matches the qualifier at the injection point.

### Spring's Autowiring

Autowiring by type is the most common approach. If there's only one bean of a matching type in the Spring context, it's automatically injected.

Autowiring by name (@Autowired with field/method name) is possible for more specific selection.

@Qualifier in Spring functions similarly to CDI's qualifiers when finer control than by-type or by-name is needed.

## Philosophical Takeaway

CDI provides more fine-grained control over bean lifecycles and selection mechanisms, particularly with qualifiers. Spring often defaults to simpler autowiring with the option of using @Qualifier for advanced scenarios.

## Jakarta REST (RESTful Web Services): Building APIs

Think of Jakarta REST as conceptually similar to Spring MVC. Both use annotations to define web endpoints.

Jakarta REST uses @Path on classes to define resource paths, similar to Spring's @Controller or more specifically @RestController and @RequestMapping.

## Mapping Requests

Both Jakarta REST and Spring MVC offer powerful constructs to map HTTP requests to specific methods in your Java code, but they do so with slightly different annotation styles. In Jakarta REST, you'll find dedicated annotations like @GET, @POST, @PUT and @DELETE, each explicitly handling its respective HTTP method. These annotations are often combined with @Path to specify the URI path and can be supplemented with @QueryParam for handling query parameters.

Spring MVC, on the other hand, takes a more composed approach. Instead of individual method annotations, it provides combined annotations like `@GetMapping` and `@PostMapping`, which integrate the HTTP method and path information into a single declaration. Path variables are handled with `@PathVariable`, while query parameters use `@RequestParam`. Spring MVC also does have additional flexibility, allowing for regex-based path matching and other advanced features.

## Philosophical Difference

Both Jakarta REST and Spring MVC enable effective web development but differ in their underlying philosophies. Jakarta REST strongly adheres to REST architectural principles, providing focused annotations for building resource-oriented APIs that use HTTP verbs and URIs.

Spring MVC takes a more flexible approach, accommodating a wider range of web architectures. It supports Model-View-Controller (MVC) patterns, where Jakarta EE offers alternatives like Jakarta Server Faces (JSF) and Jakarta MVC.

In essence, Jakarta REST is specialised for REST APIs, while Spring MVC is a generalist for various web development styles. Your choice depends on whether you prioritise strict adherence to REST or require flexibility for diverse architectures.

## Jakarta Persistence (Java Persistence API): Managing Data

Both Jakarta EE and Spring Boot leverage the Java Persistence API (JPA) as their foundation for database interactions. JPA provides a standardised way to map Java objects to relational databases, handle transactions and execute queries. In the Jakarta EE ecosystem, you work directly with JPA, offering full flexibility and control over the persistence layer. While currently lacking a high-level abstraction like Spring Data JPA, the upcoming Jakarta EE 11 is expected to include Jakarta Data, which will simplify common data access tasks.

Conversely, Spring Boot provides Spring Data JPA, a JPA abstraction that simplifies many aspects of data access with features like repositories and automatic query generation. While this adds convenience, it also introduces some constraints due to its opinionated approach. The choice between using JPA directly or opting for Spring Data JPA's abstractions ultimately depends on the balance you seek between flexibility and rapid development.

## Philosophical Differences in Practice

The contrasting philosophies of Jakarta EE and Spring Boot have tangible consequences in everyday development. Rooted in standards, Jakarta EE gives you the flexibility to handpick implementations for its various components. For instance, if your project demands JPA for data persistence, you can freely choose between Hibernate, EclipseLink or other compliant providers. This freedom, however, comes with a trade-off: setting up a highly customised Jakarta EE project might entail slightly more initial configuration and integration effort to assemble your tailored stack.

In contrast, Spring Boot is built for speed. By bundling a curated set of technologies and providing sensible defaults, it accelerates the development process, especially for applications that adhere to its conventions. In you want to create a REST API, Spring Boot offers seamless integration with Spring MVC, taking care of many details behind the scenes. This comes at a slight cost in flexibility compared to Jakarta EE's à la carte approach, but the streamlined experience often leads to faster initial development and prototyping.

## Conclusions

Jakarta EE and Spring Boot are both powerful frameworks for building modern enterprise Java applications. They share core principles, such as dependency injection, web development and data persistence, yet their actual low-level implementations differ. Jakarta EE, anchored in standards, offers flexibility and customization for projects that demand flexibility. Spring Boot, with its opinionated structure and integrated components, prioritises rapid development and a streamlined developer experience.

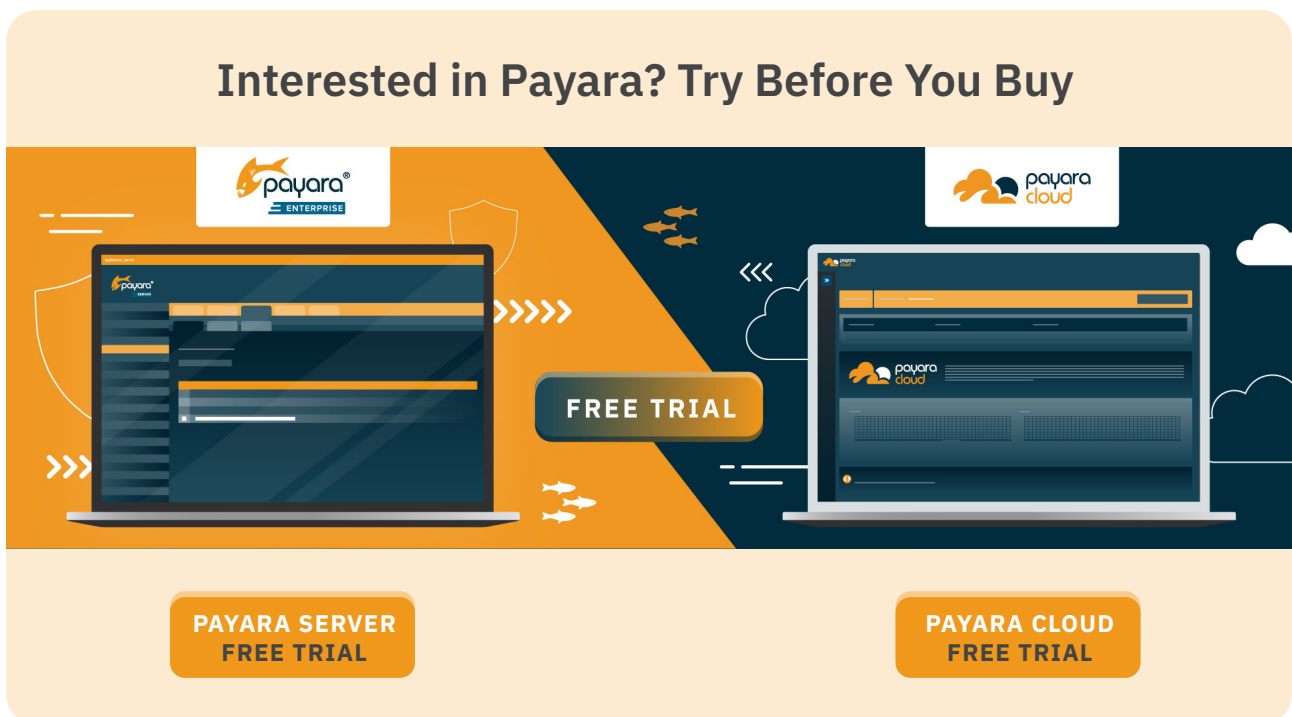
Payara Platform is a perfect example of Jakarta EE's philosophy in action. It provides a powerful and flexible runtime environment that allows you to tailor your Jakarta EE stack to your specific needs. Whether you require the full power of Payara Server for large-scale enterprise applications or the



lightweight footprint of Payara Micro for microservices, you benefit from a standards-compliant platform offering the flexibility and customization that Jakarta EE delivers. This allows you to tailor your technology stack to your precise needs, ensuring your application is built on an extensible foundation.

Ultimately, the "best" choice between Jakarta EE and Spring Boot isn't a simple matter of one being superior to the other. It depends entirely on your project's unique requirements, your team's existing expertise and the development priorities you have set. An understanding of the capabilities and philosophies of each framework allows you to make an informed decision, selecting the platform that best aligns with your goals and sets the stage for a successful and maintainable project.

## Interested in Payara? Try Before You Buy



[sales@payara.fish](mailto:sales@payara.fish)



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



[www.payara.fish](http://www.payara.fish)

Payara Services Ltd 2024 All Rights Reserved. Registered in England and Wales; Registration Number 09998946  
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ