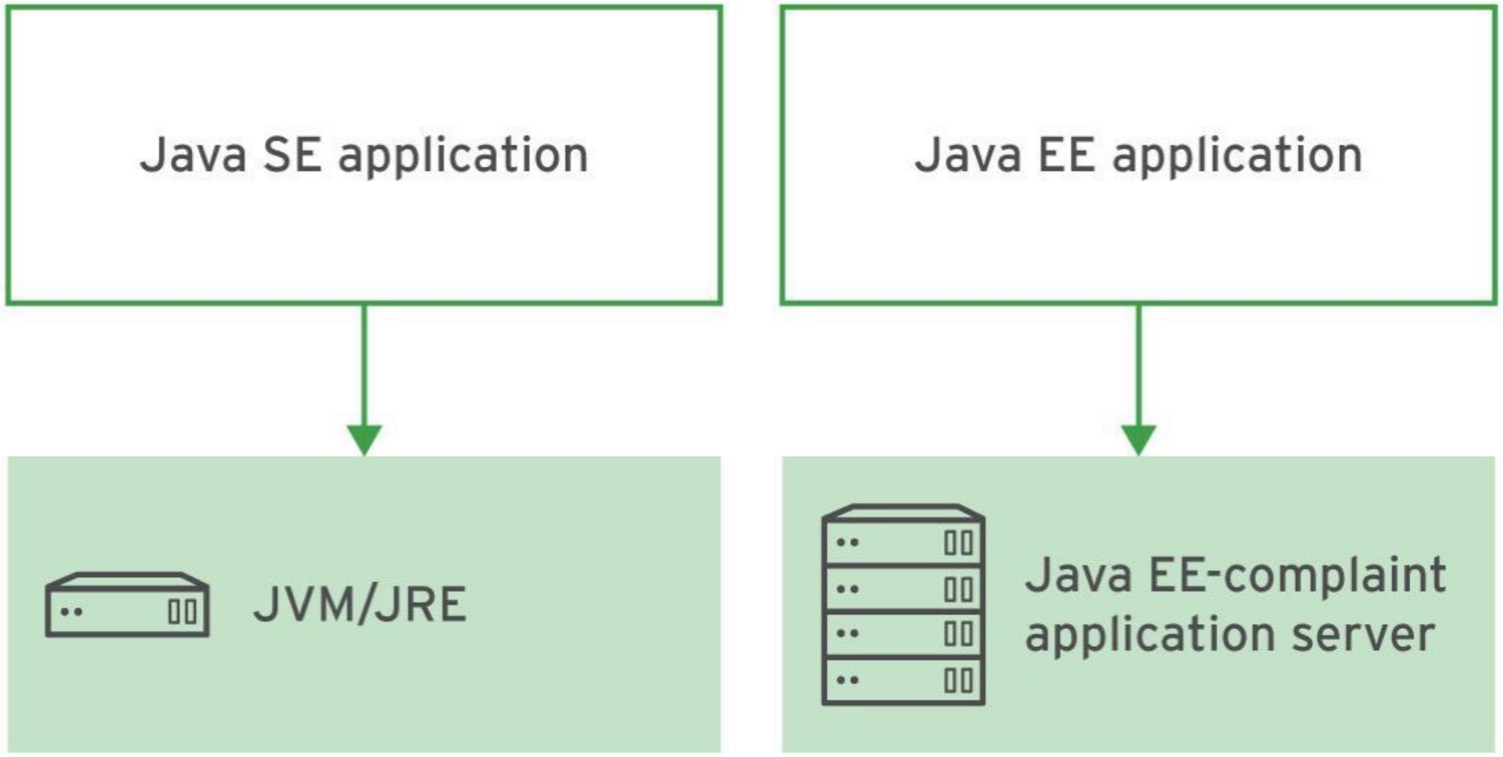
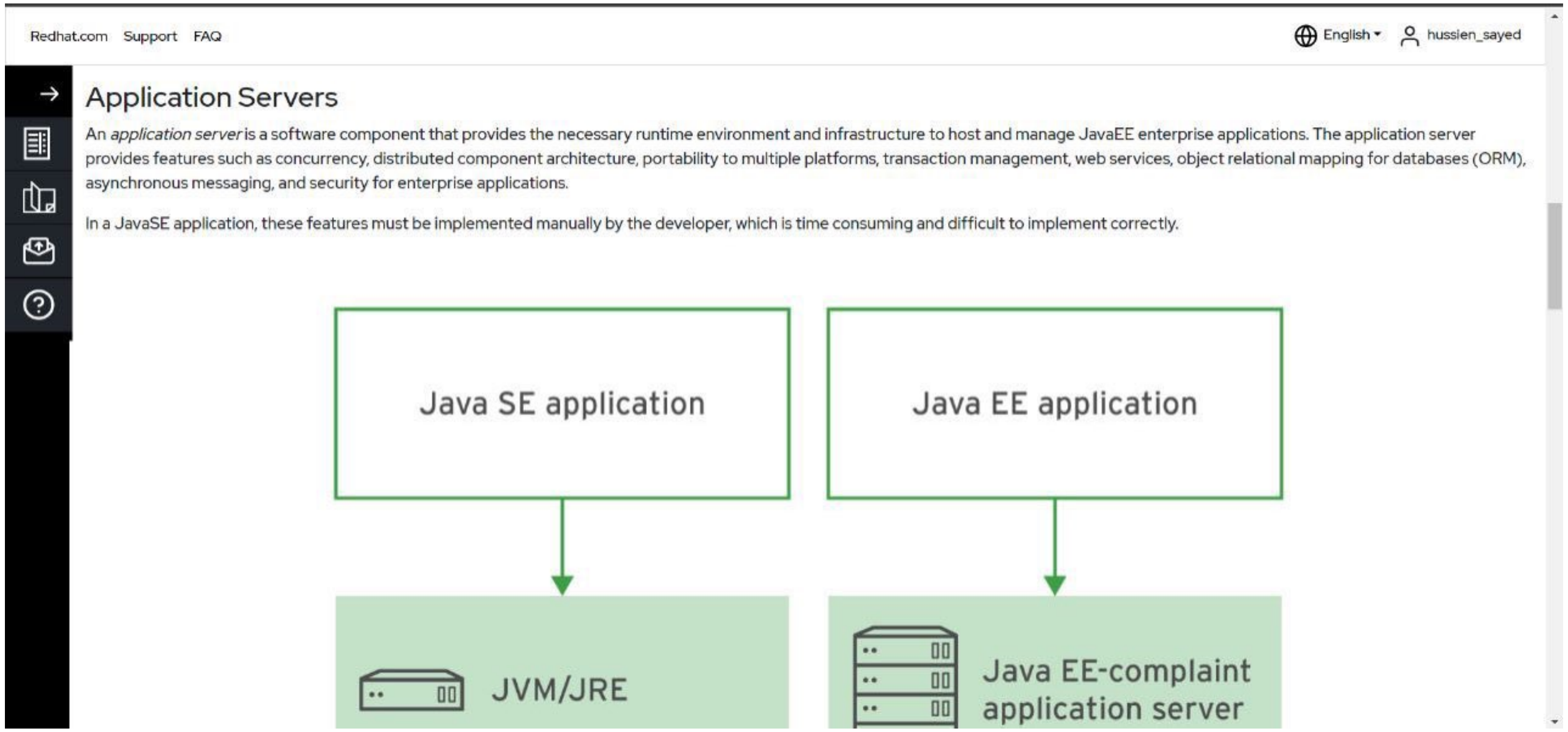
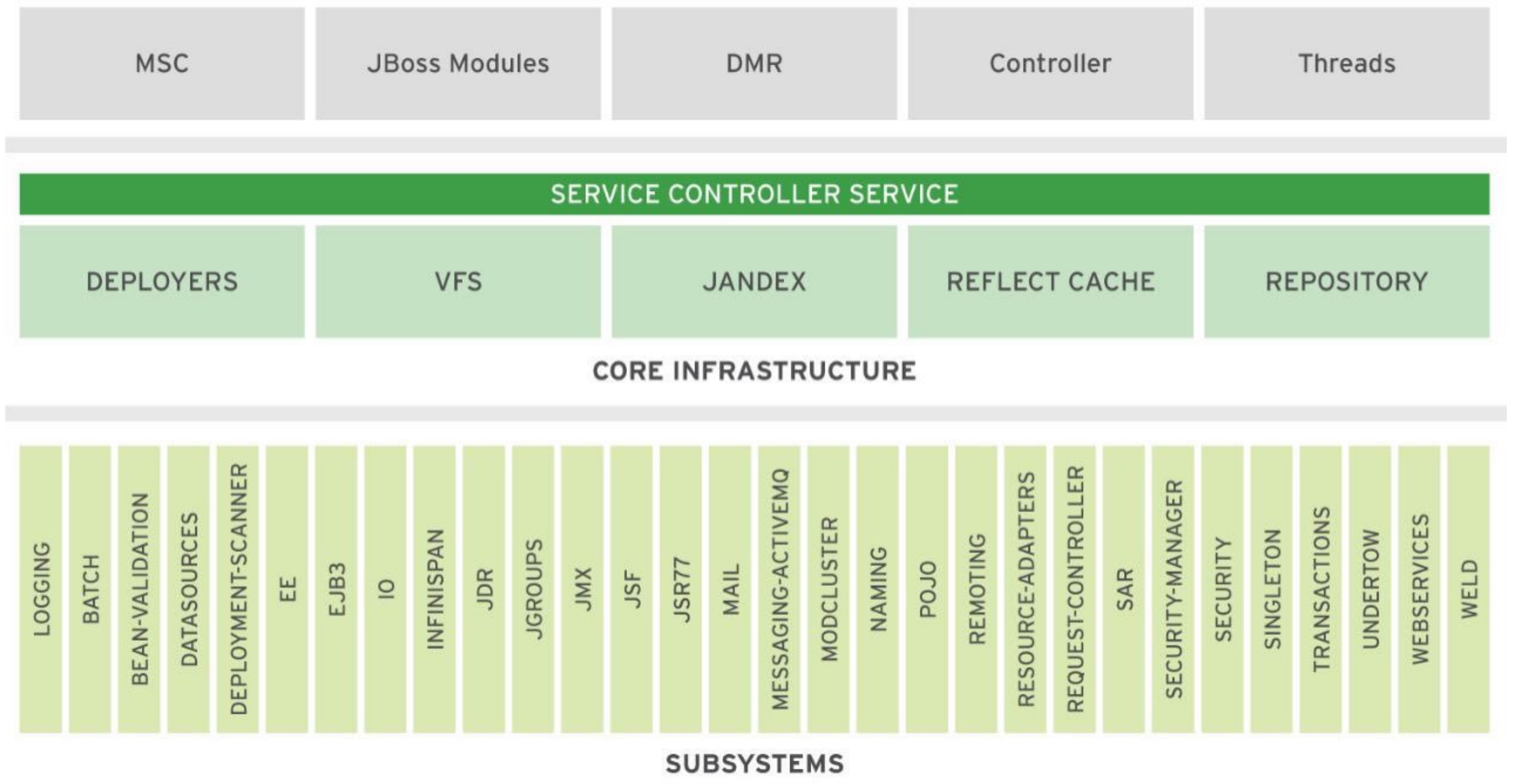

Java SE application
Java EE application
JVM/JRE
Java EE-complaint application server

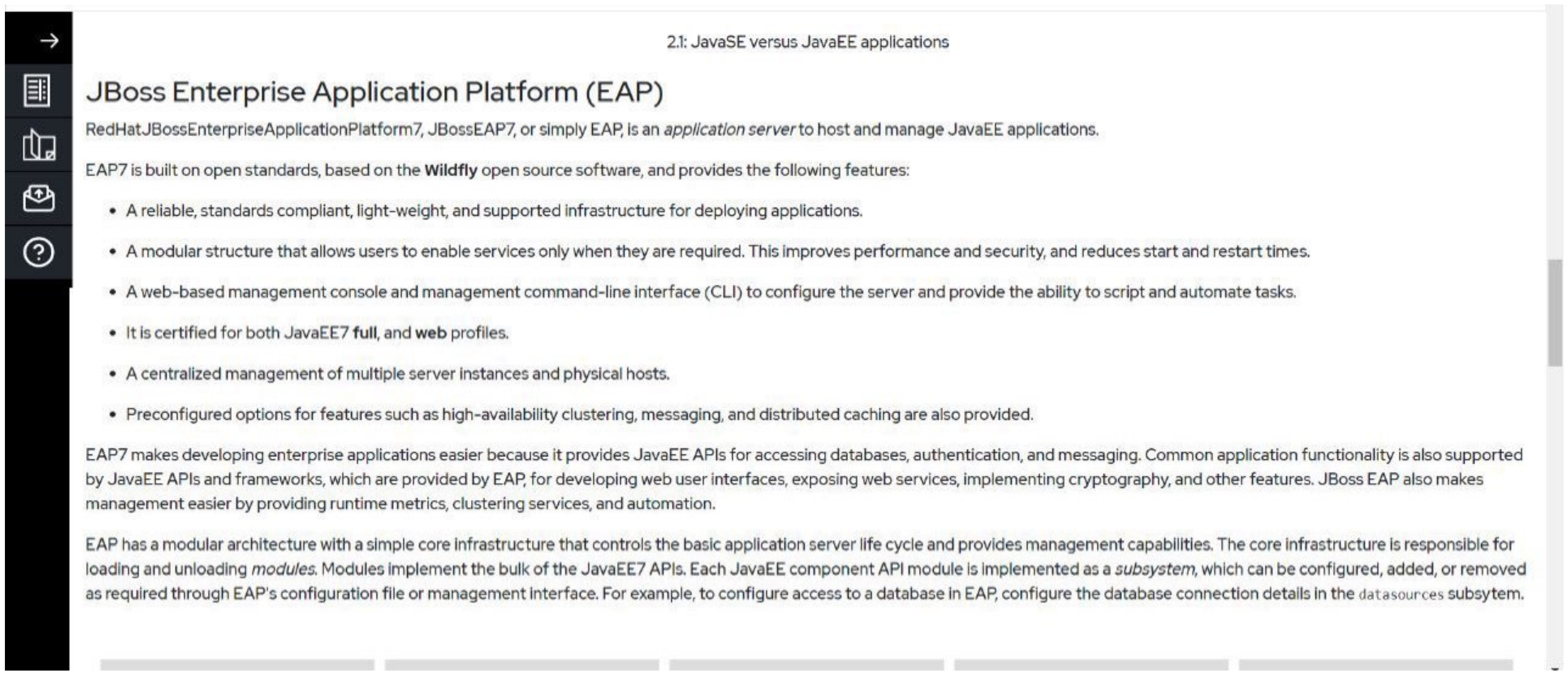


Application Servers

An *application server* is a software component that provides the necessary runtime environment and infrastructure to host and manage JavaEE enterprise applications. The application server provides features such as concurrency, distributed component architecture, portability to multiple platforms, transaction management, web services, object relational mapping for databases (ORM), asynchronous messaging, and security for enterprise applications.

In a JavaSE application, these features must be implemented manually by the developer, which is time consuming and difficult to implement correctly.


MSC
JBoss Modules
DMR
Controller
Threads
SERVICE CONTROLLER SERVICE
DEPLOYERS
VFS
JANDEX
REFLECT CACHE
REPOSITORY
CORE INFRASTRUCTURE
LOGGING
BATCH
BEAN-VALIDATION
DATASOURCES
DEPLOYMENT-SCANNER
EE
EJB3
IO
INFINISPAN
JDR
JGROUPS
JMX
JSF
JSR77
MAIL
MESSAGING-ACTIVEMQ
MODCLUSTER
NAMING
POJO
REMOTING
RESOURCE-ADAPTERS
REQUEST-CONTROLLER
SAR
SECURITY-MANAGER
SECURITY
SINGLETON
TRANSACTIONS
UNDERTOW
WEBSERVICES
WELD
SUBSYSTEMS



JBoss Enterprise Application Platform (EAP)

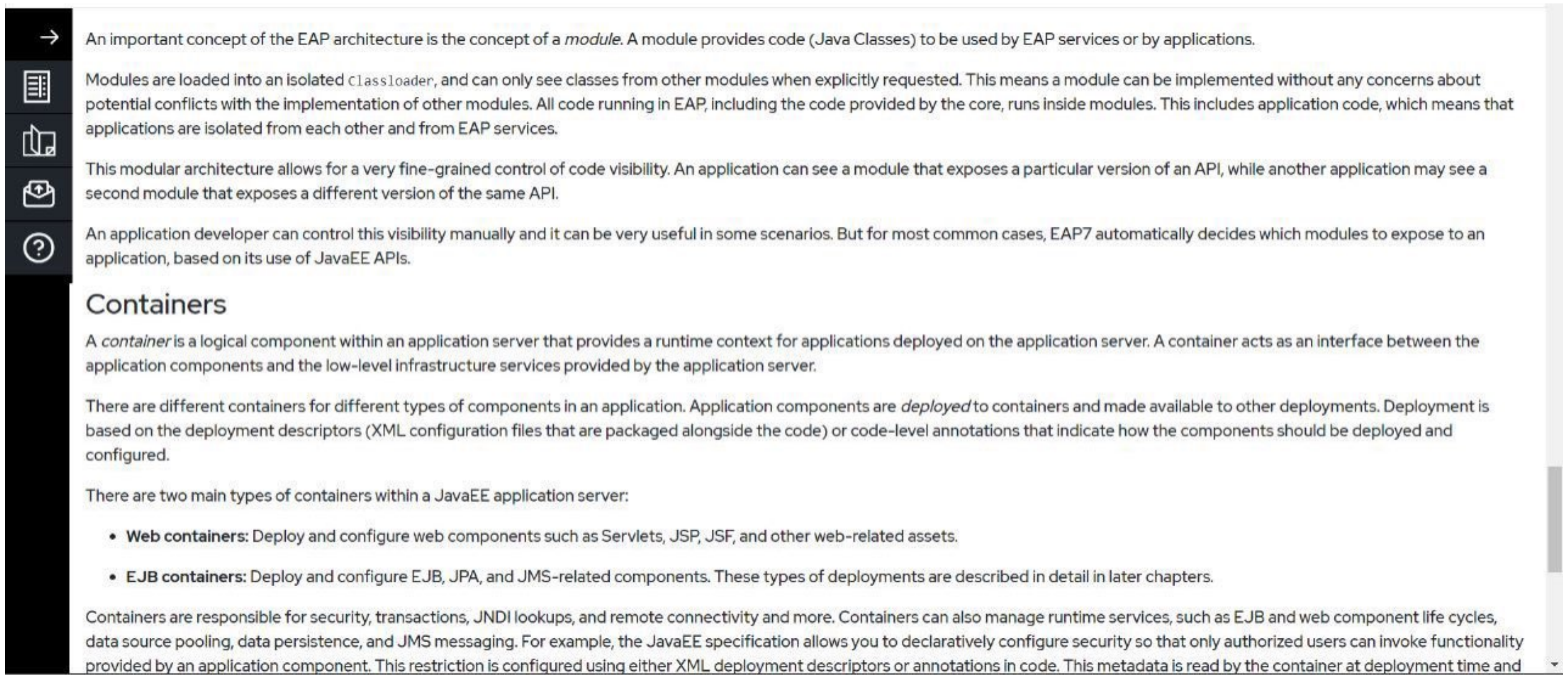
RedHatJBossEnterpriseApplicationPlatform7, JBossEAP7, or simply EAP, is an *application server* to host and manage JavaEE applications.

EAP7 is built on open standards, based on the **Wildfly** open source software, and provides the following features:

- A reliable, standards compliant, light-weight, and supported infrastructure for deploying applications.
- A modular structure that allows users to enable services only when they are required. This improves performance and security, and reduces start and restart times.
- A web-based management console and management command-line interface (CLI) to configure the server and provide the ability to script and automate tasks.
- It is certified for both JavaEE7 **full**, and **web** profiles.
- A centralized management of multiple server instances and physical hosts.
- Preconfigured options for features such as high-availability clustering, messaging, and distributed caching are also provided.

EAP7 makes developing enterprise applications easier because it provides JavaEE APIs for accessing databases, authentication, and messaging. Common application functionality is also supported by JavaEE APIs and frameworks, which are provided by EAP, for developing web user interfaces, exposing web services, implementing cryptography, and other features. JBoss EAP also makes management easier by providing runtime metrics, clustering services, and automation.

EAP has a modular architecture with a simple core infrastructure that controls the basic application server life cycle and provides management capabilities. The core infrastructure is responsible for loading and unloading *modules*. Modules implement the bulk of the JavaEE7 APIs. Each JavaEE component API module is implemented as a *subsystem*, which can be configured, added, or removed as required through EAP's configuration file or management interface. For example, to configure access to a database in EAP, configure the database connection details in the `datasources` subsytem.



An important concept of the EAP architecture is the concept of a *module*. A module provides code (Java Classes) to be used by EAP services or by applications.

Modules are loaded into an isolated `Classloader`, and can only see classes from other modules when explicitly requested. This means a module can be implemented without any concerns about potential conflicts with the implementation of other modules. All code running in EAP, including the code provided by the core, runs inside modules. This includes application code, which means that applications are isolated from each other and from EAP services.

This modular architecture allows for a very fine-grained control of code visibility. An application can see a module that exposes a particular version of an API, while another application may see a second module that exposes a different version of the same API.

An application developer can control this visibility manually and it can be very useful in some scenarios. But for most common cases, EAP7 automatically decides which modules to expose to an application, based on its use of JavaEE APIs.

Containers

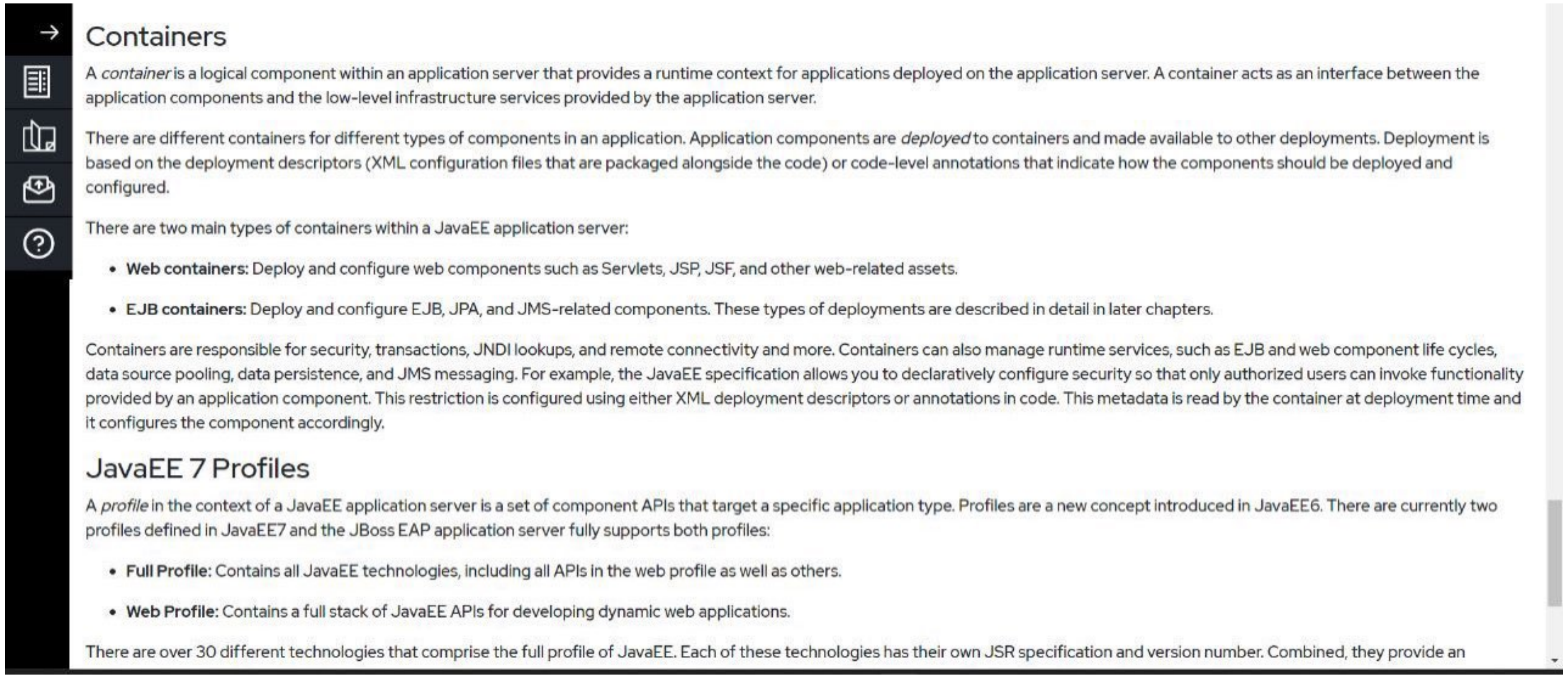
A *container* is a logical component within an application server that provides a runtime context for applications deployed on the application server. A container acts as an interface between the application components and the low-level infrastructure services provided by the application server.

There are different containers for different types of components in an application. Application components are *deployed* to containers and made available to other deployments. Deployment is based on the deployment descriptors (XML configuration files that are packaged alongside the code) or code-level annotations that indicate how the components should be deployed and configured.

There are two main types of containers within a JavaEE application server:

- **Web containers:** Deploy and configure web components such as Servlets, JSP, JSF, and other web-related assets.
- **EJB containers:** Deploy and configure EJB, JPA, and JMS-related components. These types of deployments are described in detail in later chapters.

Containers are responsible for security, transactions, JNDI lookups, and remote connectivity and more. Containers can also manage runtime services, such as EJB and web component life cycles, data source pooling, data persistence, and JMS messaging. For example, the JavaEE specification allows you to declaratively configure security so that only authorized users can invoke functionality provided by an application component. This restriction is configured using either XML deployment descriptors or annotations in code. This metadata is read by the container at deployment time and



Containers

A *container* is a logical component within an application server that provides a runtime context for applications deployed on the application server. A container acts as an interface between the application components and the low-level infrastructure services provided by the application server.

There are different containers for different types of components in an application. Application components are *deployed* to containers and made available to other deployments. Deployment is based on the deployment descriptors (XML configuration files that are packaged alongside the code) or code-level annotations that indicate how the components should be deployed and configured.

There are two main types of containers within a JavaEE application server:

- **Web containers:** Deploy and configure web components such as Servlets, JSP, JSF, and other web-related assets.
- **EJB containers:** Deploy and configure EJB, JPA, and JMS-related components. These types of deployments are described in detail in later chapters.

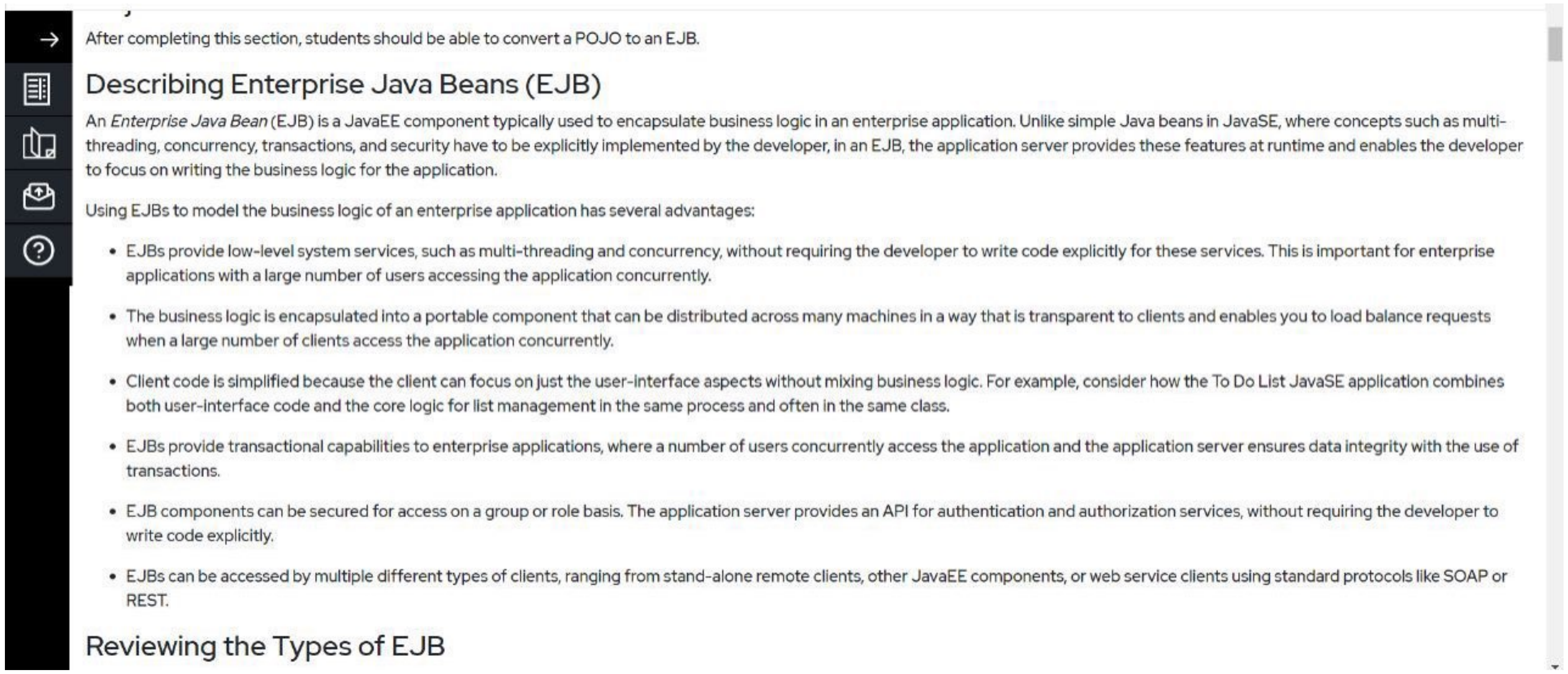
Containers are responsible for security, transactions, JNDI lookups, and remote connectivity and more. Containers can also manage runtime services, such as EJB and web component life cycles, data source pooling, data persistence, and JMS messaging. For example, the JavaEE specification allows you to declaratively configure security so that only authorized users can invoke functionality provided by an application component. This restriction is configured using either XML deployment descriptors or annotations in code. This metadata is read by the container at deployment time and it configures the component accordingly.

JavaEE 7 Profiles

A *profile* in the context of a JavaEE application server is a set of component APIs that target a specific application type. Profiles are a new concept introduced in JavaEE6. There are currently two profiles defined in JavaEE7 and the JBoss EAP application server fully supports both profiles:

- **Full Profile:** Contains all JavaEE technologies, including all APIs in the web profile as well as others.
- **Web Profile:** Contains a full stack of JavaEE APIs for developing dynamic web applications.

There are over 30 different technologies that comprise the full profile of JavaEE. Each of these technologies has their own JSR specification and version number. Combined, they provide an


After completing this section, students should be able to convert a POJO to an EJB.

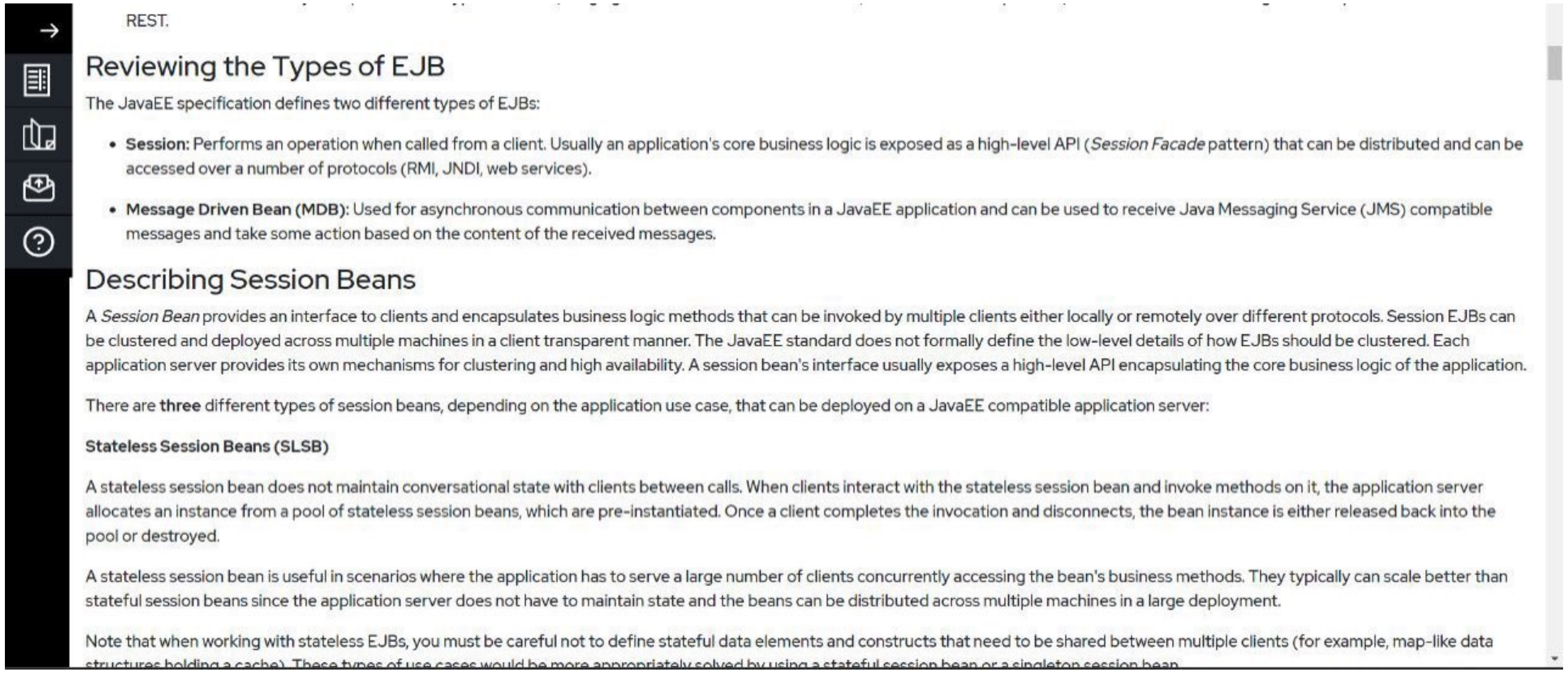
Describing Enterprise Java Beans (EJB)

An *Enterprise Java Bean* (EJB) is a JavaEE component typically used to encapsulate business logic in an enterprise application. Unlike simple Java beans in JavaSE, where concepts such as multi-threading, concurrency, transactions, and security have to be explicitly implemented by the developer, in an EJB, the application server provides these features at runtime and enables the developer to focus on writing the business logic for the application.

Using EJBs to model the business logic of an enterprise application has several advantages:

- EJBs provide low-level system services, such as multi-threading and concurrency, without requiring the developer to write code explicitly for these services. This is important for enterprise applications with a large number of users accessing the application concurrently.
- The business logic is encapsulated into a portable component that can be distributed across many machines in a way that is transparent to clients and enables you to load balance requests when a large number of clients access the application concurrently.
- Client code is simplified because the client can focus on just the user-interface aspects without mixing business logic. For example, consider how the To Do List JavaSE application combines both user-interface code and the core logic for list management in the same process and often in the same class.
- EJBs provide transactional capabilities to enterprise applications, where a number of users concurrently access the application and the application server ensures data integrity with the use of transactions.
- EJB components can be secured for access on a group or role basis. The application server provides an API for authentication and authorization services, without requiring the developer to write code explicitly.
- EJBs can be accessed by multiple different types of clients, ranging from stand-alone remote clients, other JavaEE components, or web service clients using standard protocols like SOAP or REST.

Reviewing the Types of EJB



REST.

Reviewing the Types of EJB

The JavaEE specification defines two different types of EJBs:

- **Session:** Performs an operation when called from a client. Usually an application's core business logic is exposed as a high-level API (*Session Facade* pattern) that can be distributed and can be accessed over a number of protocols (RMI, JNDI, web services).
- **Message Driven Bean (MDB):** Used for asynchronous communication between components in a JavaEE application and can be used to receive Java Messaging Service (JMS) compatible messages and take some action based on the content of the received messages.

Describing Session Beans

A *Session Bean* provides an interface to clients and encapsulates business logic methods that can be invoked by multiple clients either locally or remotely over different protocols. Session EJBs can be clustered and deployed across multiple machines in a client transparent manner. The JavaEE standard does not formally define the low-level details of how EJBs should be clustered. Each application server provides its own mechanisms for clustering and high availability. A session bean's interface usually exposes a high-level API encapsulating the core business logic of the application.

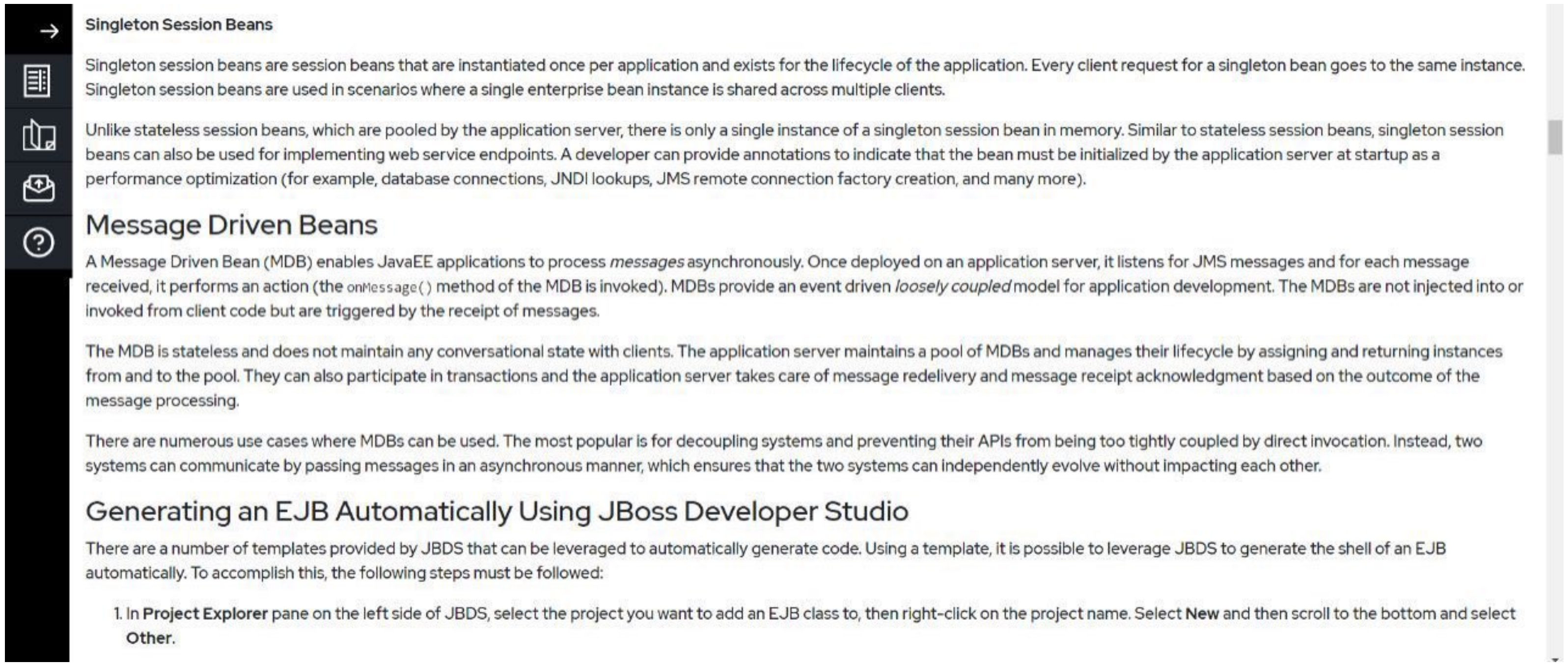
There are **three** different types of session beans, depending on the application use case, that can be deployed on a JavaEE compatible application server:

Stateless Session Beans (SLSB)

A stateless session bean does not maintain conversational state with clients between calls. When clients interact with the stateless session bean and invoke methods on it, the application server allocates an instance from a pool of stateless session beans, which are pre-instantiated. Once a client completes the invocation and disconnects, the bean instance is either released back into the pool or destroyed.

A stateless session bean is useful in scenarios where the application has to serve a large number of clients concurrently accessing the bean's business methods. They typically can scale better than stateful session beans since the application server does not have to maintain state and the beans can be distributed across multiple machines in a large deployment.

Note that when working with stateless EJBs, you must be careful not to define stateful data elements and constructs that need to be shared between multiple clients (for example, map-like data structures holding a cache). These types of use cases would be more appropriately solved by using a stateful session bean or a singleton session bean.



Singleton Session Beans

Singleton session beans are session beans that are instantiated once per application and exists for the lifecycle of the application. Every client request for a singleton bean goes to the same instance. Singleton session beans are used in scenarios where a single enterprise bean instance is shared across multiple clients.

Unlike stateless session beans, which are pooled by the application server, there is only a single instance of a singleton session bean in memory. Similar to stateless session beans, singleton session beans can also be used for implementing web service endpoints. A developer can provide annotations to indicate that the bean must be initialized by the application server at startup as a performance optimization (for example, database connections, JNDI lookups, JMS remote connection factory creation, and many more).

Message Driven Beans

A Message Driven Bean (MDB) enables JavaEE applications to process *messages* asynchronously. Once deployed on an application server, it listens for JMS messages and for each message received, it performs an action (the `onMessage()` method of the MDB is invoked). MDBs provide an event driven *loosely coupled* model for application development. The MDBs are not injected into or invoked from client code but are triggered by the receipt of messages.

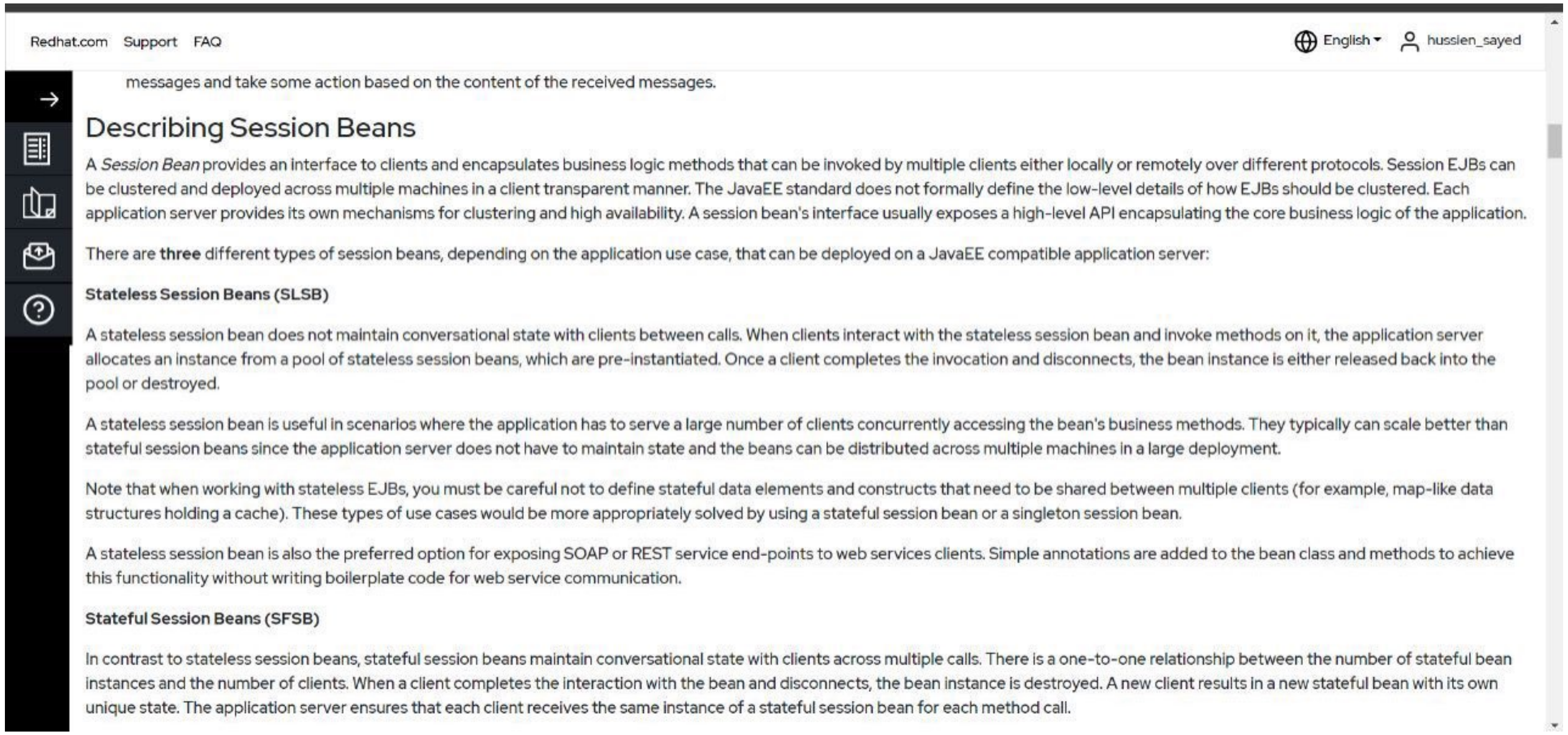
The MDB is stateless and does not maintain any conversational state with clients. The application server maintains a pool of MDBs and manages their lifecycle by assigning and returning instances from and to the pool. They can also participate in transactions and the application server takes care of message redelivery and message receipt acknowledgment based on the outcome of the message processing.

There are numerous use cases where MDBs can be used. The most popular is for decoupling systems and preventing their APIs from being too tightly coupled by direct invocation. Instead, two systems can communicate by passing messages in an asynchronous manner, which ensures that the two systems can independently evolve without impacting each other.

Generating an EJB Automatically Using JBoss Developer Studio

There are a number of templates provided by JBDS that can be leveraged to automatically generate code. Using a template, it is possible to leverage JBDS to generate the shell of an EJB automatically. To accomplish this, the following steps must be followed:

1. In **Project Explorer** pane on the left side of JBDS, select the project you want to add an EJB class to, then right-click on the project name. Select **New** and then scroll to the bottom and select **Other**.



messages and take some action based on the content of the received messages.

Describing Session Beans

A *Session Bean* provides an interface to clients and encapsulates business logic methods that can be invoked by multiple clients either locally or remotely over different protocols. Session EJBs can be clustered and deployed across multiple machines in a client transparent manner. The JavaEE standard does not formally define the low-level details of how EJBs should be clustered. Each application server provides its own mechanisms for clustering and high availability. A session bean's interface usually exposes a high-level API encapsulating the core business logic of the application.

There are **three** different types of session beans, depending on the application use case, that can be deployed on a JavaEE compatible application server:

Stateless Session Beans (SLSB)

A stateless session bean does not maintain conversational state with clients between calls. When clients interact with the stateless session bean and invoke methods on it, the application server allocates an instance from a pool of stateless session beans, which are pre-instantiated. Once a client completes the invocation and disconnects, the bean instance is either released back into the pool or destroyed.

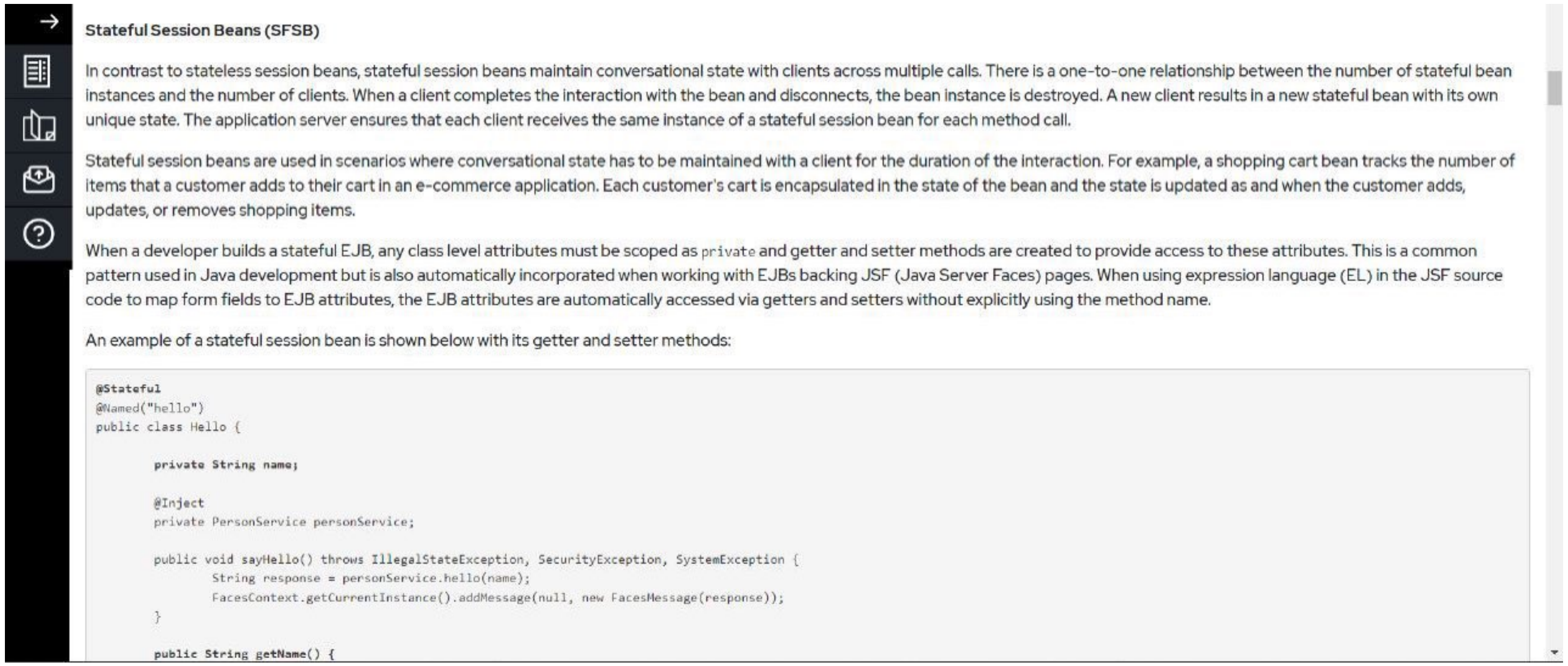
A stateless session bean is useful in scenarios where the application has to serve a large number of clients concurrently accessing the bean's business methods. They typically can scale better than stateful session beans since the application server does not have to maintain state and the beans can be distributed across multiple machines in a large deployment.

Note that when working with stateless EJBs, you must be careful not to define stateful data elements and constructs that need to be shared between multiple clients (for example, map-like data structures holding a cache). These types of use cases would be more appropriately solved by using a stateful session bean or a singleton session bean.

A stateless session bean is also the preferred option for exposing SOAP or REST service end-points to web services clients. Simple annotations are added to the bean class and methods to achieve this functionality without writing boilerplate code for web service communication.

Stateful Session Beans (SFSB)

In contrast to stateless session beans, stateful session beans maintain conversational state with clients across multiple calls. There is a one-to-one relationship between the number of stateful bean instances and the number of clients. When a client completes the interaction with the bean and disconnects, the bean instance is destroyed. A new client results in a new stateful bean with its own unique state. The application server ensures that each client receives the same instance of a stateful session bean for each method call.



Stateful Session Beans (SFSB)

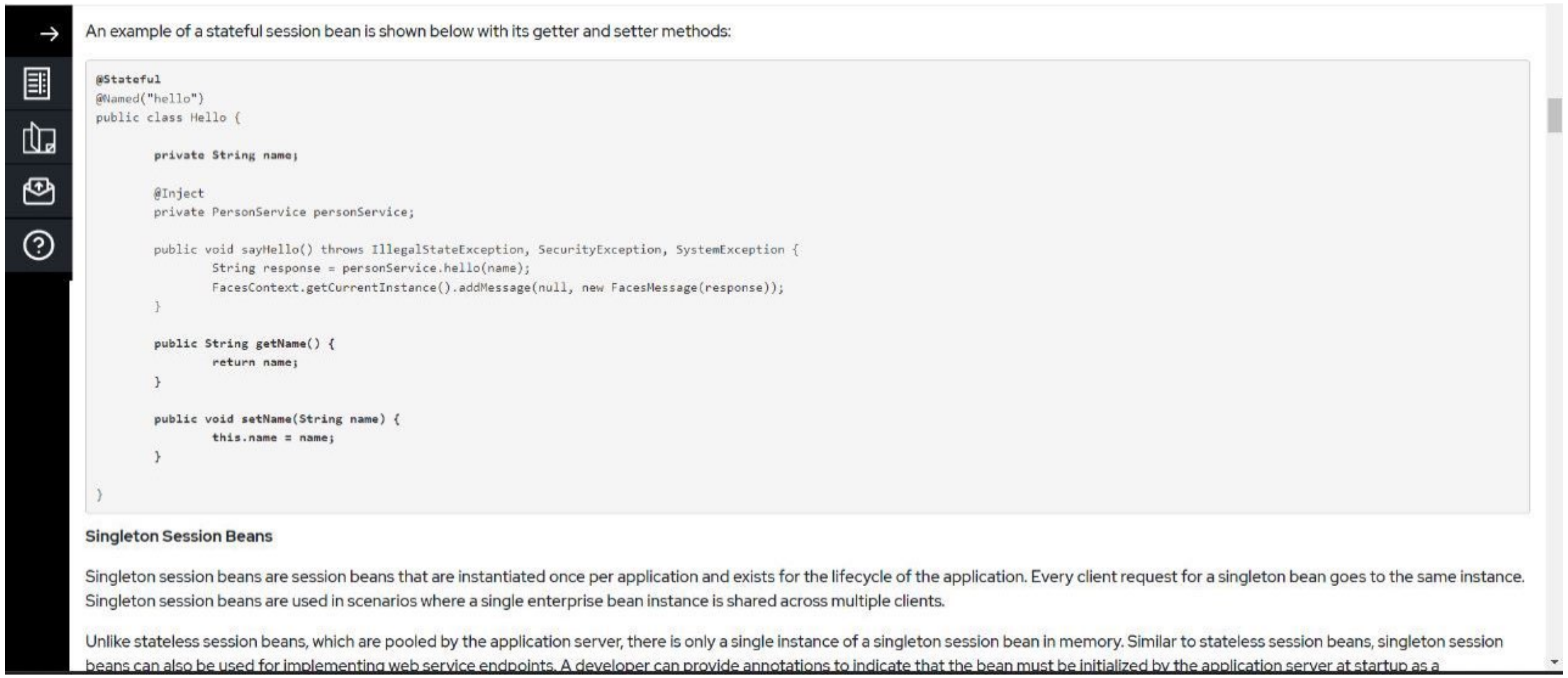
In contrast to stateless session beans, stateful session beans maintain conversational state with clients across multiple calls. There is a one-to-one relationship between the number of stateful bean instances and the number of clients. When a client completes the interaction with the bean and disconnects, the bean instance is destroyed. A new client results in a new stateful bean with its own unique state. The application server ensures that each client receives the same instance of a stateful session bean for each method call.

Stateful session beans are used in scenarios where conversational state has to be maintained with a client for the duration of the interaction. For example, a shopping cart bean tracks the number of items that a customer adds to their cart in an e-commerce application. Each customer's cart is encapsulated in the state of the bean and the state is updated as and when the customer adds, updates, or removes shopping items.

When a developer builds a stateful EJB, any class level attributes must be scoped as `private` and getter and setter methods are created to provide access to these attributes. This is a common pattern used in Java development but is also automatically incorporated when working with EJBs backing JSF (Java Server Faces) pages. When using expression language (EL) in the JSF source code to map form fields to EJB attributes, the EJB attributes are automatically accessed via getters and setters without explicitly using the method name.

An example of a stateful session bean is shown below with its getter and setter methods:

```
@Stateful
@Named("hello")
public class Hello {

        private String name;

        @Inject
        private PersonService personService;

        public void sayHello() throws IllegalStateException, SecurityException, SystemException {
                String response = personService.hello(name);
                FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(response));
        }

        public String getName() {
```



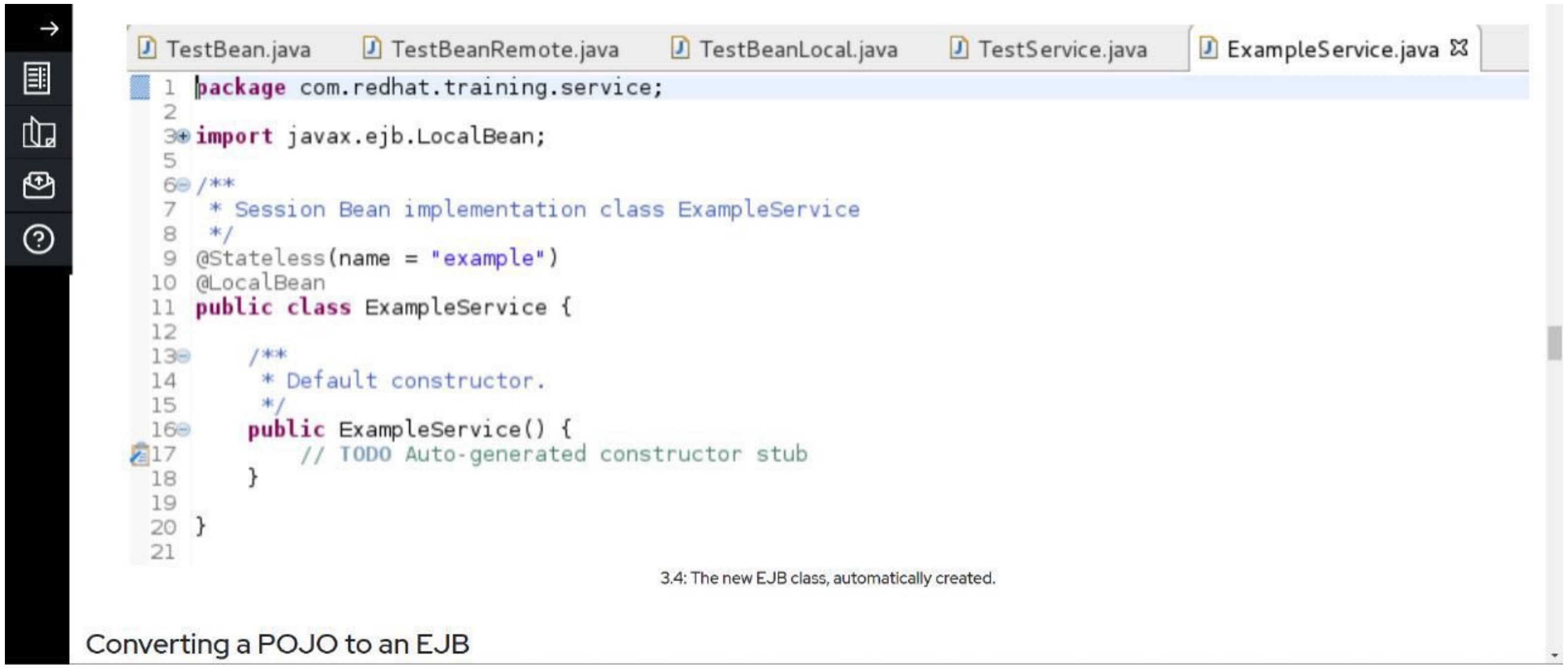
An example of a stateful session bean is shown below with its getter and setter methods:

```
@Stateful
@Named("hello")
public class Hello {

        private String name;

        @Inject
        private PersonService personService;

        public void sayHello() throws IllegalStateException, SecurityException, SystemException {
                String response = personService.hello(name);
                FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(response));
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

}
```

Singleton Session Beans

Singleton session beans are session beans that are instantiated once per application and exists for the lifecycle of the application. Every client request for a singleton bean goes to the same instance. Singleton session beans are used in scenarios where a single enterprise bean instance is shared across multiple clients.

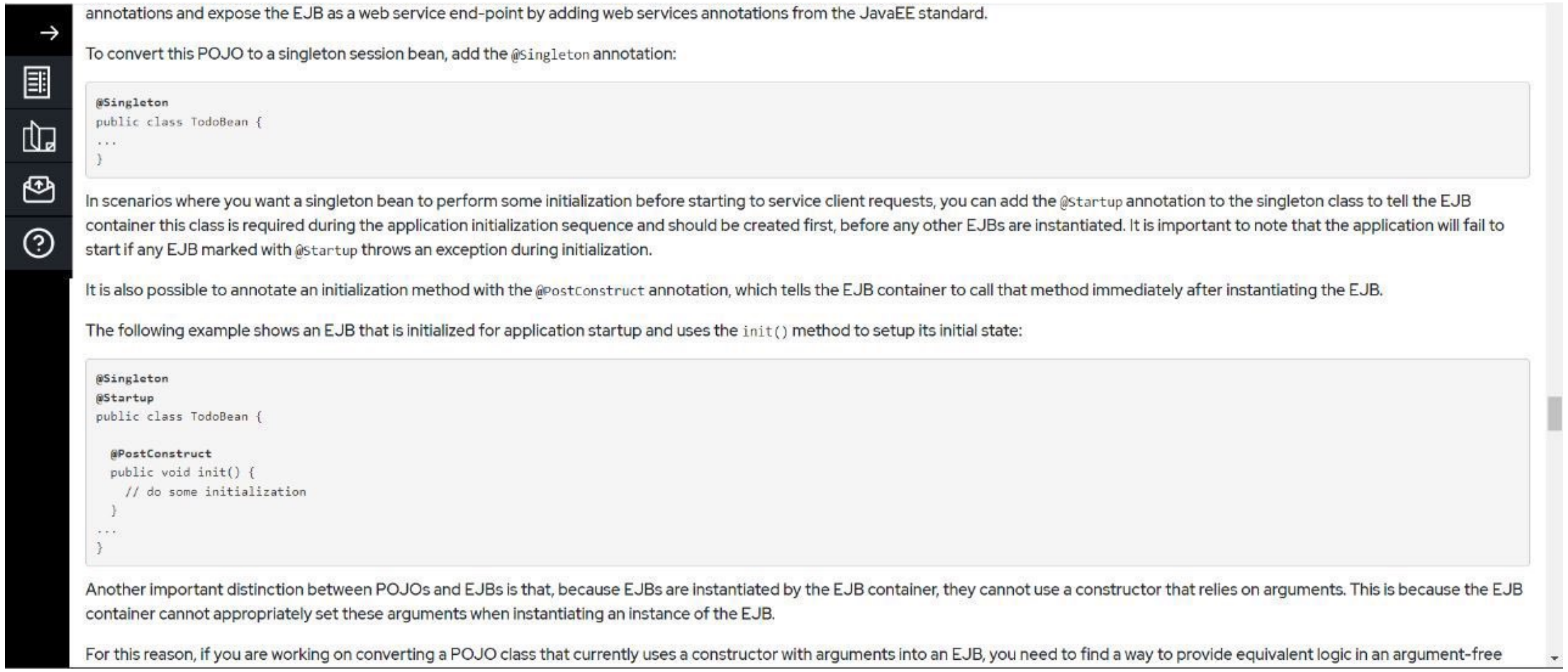
Unlike stateless session beans, which are pooled by the application server, there is only a single instance of a singleton session bean in memory. Similar to stateless session beans, singleton session beans can also be used for implementing web service endpoints. A developer can provide annotations to indicate that the bean must be initialized by the application server at startup as a



```java
package com.redhat.training.service;

import javax.ejb.LocalBean;

/**
 * Session Bean implementation class ExampleService
 */
@Stateless(name = "example")
@LocalBean
public class ExampleService {

    /**
     * Default constructor.
     */
    public ExampleService() {
        // TODO Auto-generated constructor stub
    }

}

```

3.4: The new EJB class, automatically created.

Converting a POJO to an EJB



annotations and expose the EJB as a web service end-point by adding web services annotations from the JavaEE standard.

To convert this POJO to a singleton session bean, add the `@Singleton` annotation:

```
@Singleton
public class TodoBean {
...
}
```

In scenarios where you want a singleton bean to perform some initialization before starting to service client requests, you can add the `@Startup` annotation to the singleton class to tell the EJB container this class is required during the application initialization sequence and should be created first, before any other EJBs are instantiated. It is important to note that the application will fail to start if any EJB marked with `@Startup` throws an exception during initialization.

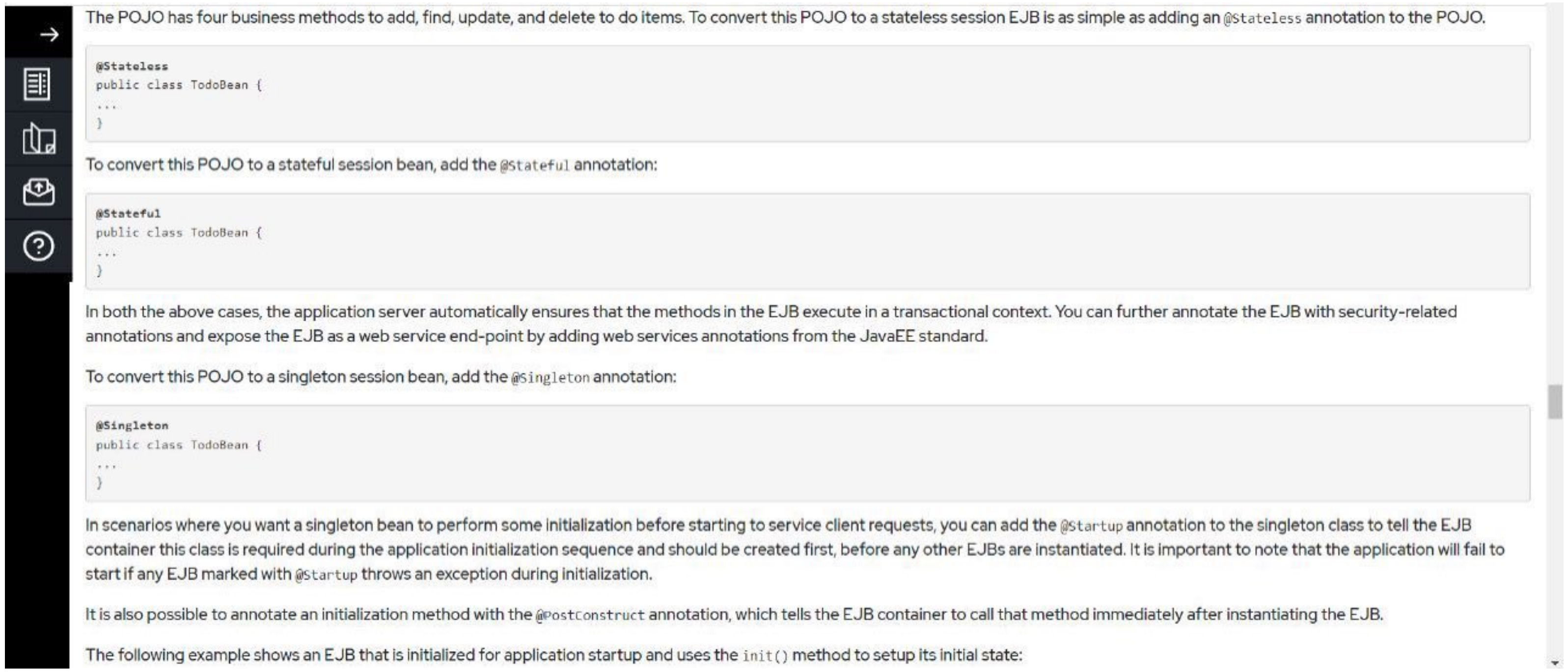
It is also possible to annotate an initialization method with the `@PostConstruct` annotation, which tells the EJB container to call that method immediately after instantiating the EJB.

The following example shows an EJB that is initialized for application startup and uses the `init()` method to setup its initial state:

```
@Singleton
@Startup
public class TodoBean {

  @PostConstruct
  public void init() {
    // do some initialization
  }
...
}
```

Another important distinction between POJOs and EJBs is that, because EJBs are instantiated by the EJB container, they cannot use a constructor that relies on arguments. This is because the EJB container cannot appropriately set these arguments when instantiating an instance of the EJB.

For this reason, if you are working on converting a POJO class that currently uses a constructor with arguments into an EJB, you need to find a way to provide equivalent logic in an argument-free



The POJO has four business methods to add, find, update, and delete to do items. To convert this POJO to a stateless session EJB is as simple as adding an `@Stateless` annotation to the POJO.

```
@Stateless
public class TodoBean {
...
}
```

To convert this POJO to a stateful session bean, add the `@Stateful` annotation:

```
@Stateful
public class TodoBean {
...
}
```

In both the above cases, the application server automatically ensures that the methods in the EJB execute in a transactional context. You can further annotate the EJB with security-related annotations and expose the EJB as a web service end-point by adding web services annotations from the JavaEE standard.

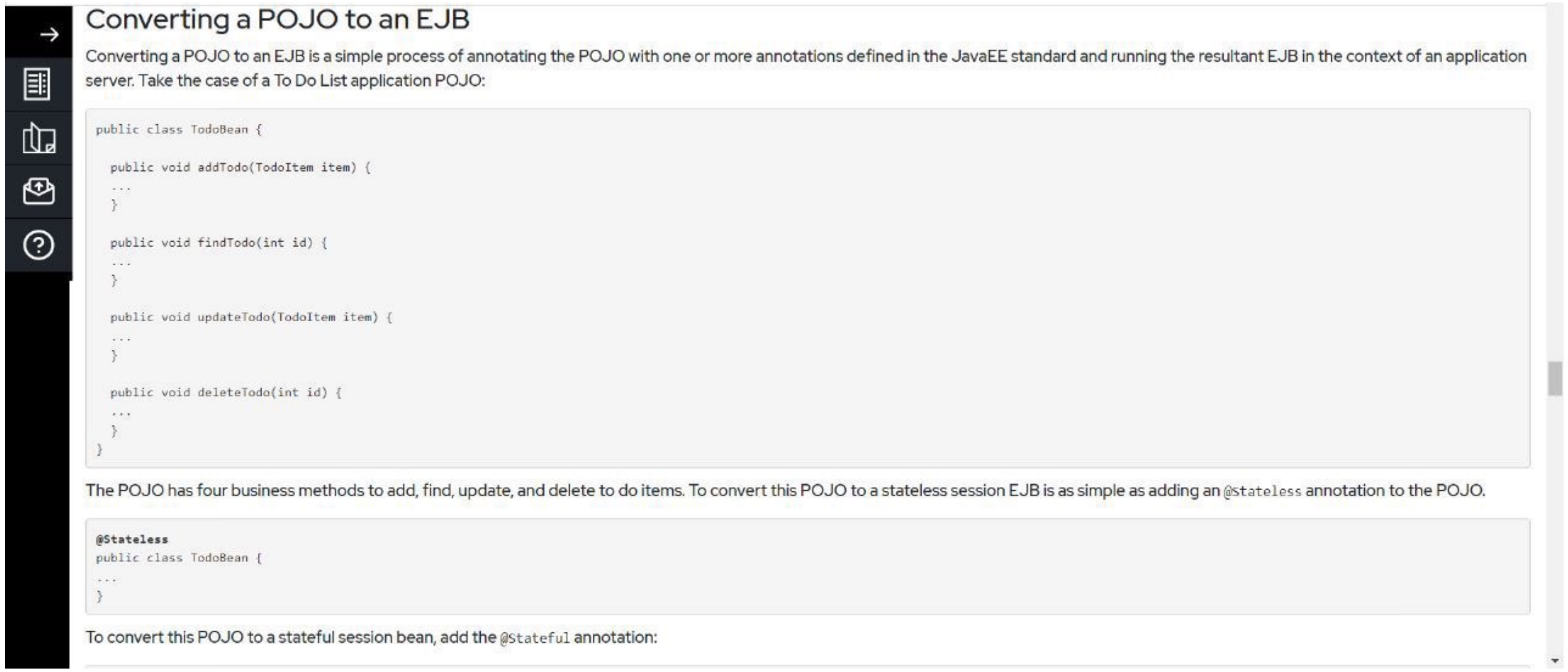
To convert this POJO to a singleton session bean, add the `@Singleton` annotation:

```
@Singleton
public class TodoBean {
...
}
```

In scenarios where you want a singleton bean to perform some initialization before starting to service client requests, you can add the `@Startup` annotation to the singleton class to tell the EJB container this class is required during the application initialization sequence and should be created first, before any other EJBs are instantiated. It is important to note that the application will fail to start if any EJB marked with `@Startup` throws an exception during initialization.

It is also possible to annotate an initialization method with the `@PostConstruct` annotation, which tells the EJB container to call that method immediately after instantiating the EJB.

The following example shows an EJB that is initialized for application startup and uses the `init()` method to setup its initial state:



Converting a POJO to an EJB

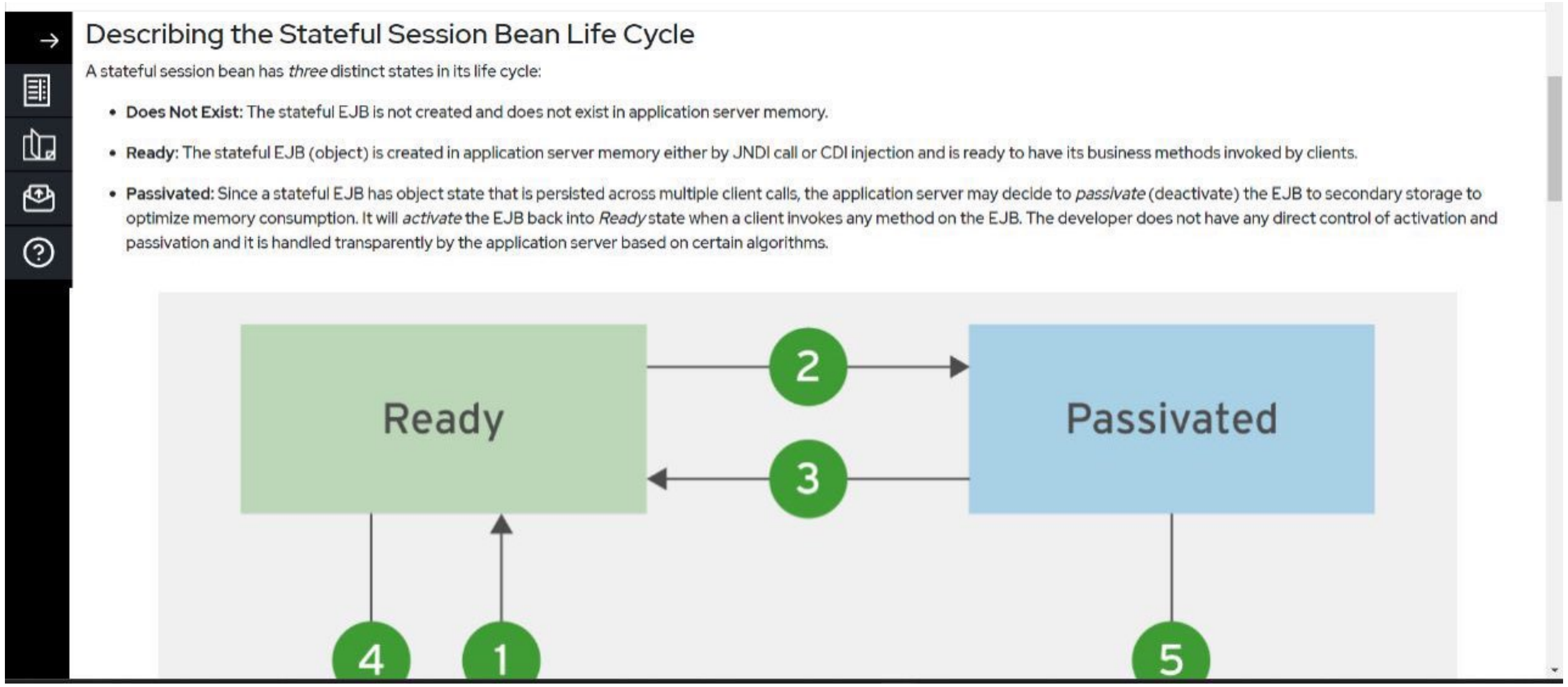
Converting a POJO to an EJB is a simple process of annotating the POJO with one or more annotations defined in the JavaEE standard and running the resultant EJB in the context of an application server. Take the case of a To Do List application POJO:

```
public class TodoBean {

  public void addTodo(TodoItem item) {
  ...
  }

  public void findTodo(int id) {
  ...
  }

  public void updateTodo(TodoItem item) {
  ...
  }

  public void deleteTodo(int id) {
  ...
  }
}
```

The POJO has four business methods to add, find, update, and delete to do items. To convert this POJO to a stateless session EJB is as simple as adding an `@stateless` annotation to the POJO.

```
@Stateless
public class TodoBean {
...
}
```

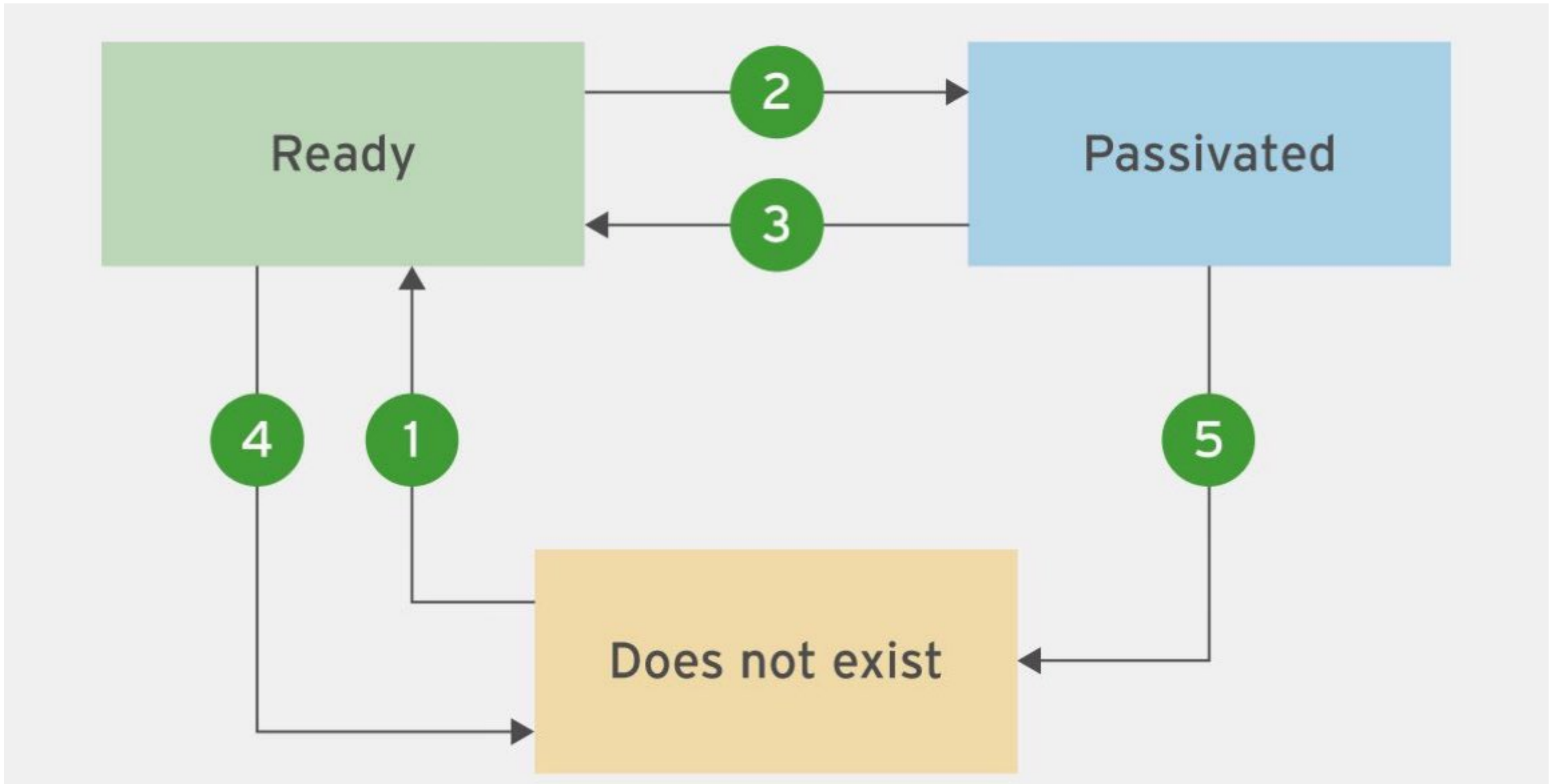
To convert this POJO to a stateful session bean, add the `@Stateful` annotation:



Describing the Stateful Session Bean Life Cycle

A stateful session bean has *three* distinct states in its life cycle:

- **Does Not Exist:** The stateful EJB is not created and does not exist in application server memory.
- **Ready:** The stateful EJB (object) is created in application server memory either by JNDI call or CDI injection and is ready to have its business methods invoked by clients.
- **Passivated:** Since a stateful EJB has object state that is persisted across multiple client calls, the application server may decide to *passivate* (deactivate) the EJB to secondary storage to optimize memory consumption. It will *activate* the EJB back into *Ready* state when a client invokes any method on the EJB. The developer does not have any direct control of activation and passivation and it is handled transparently by the application server based on certain algorithms.



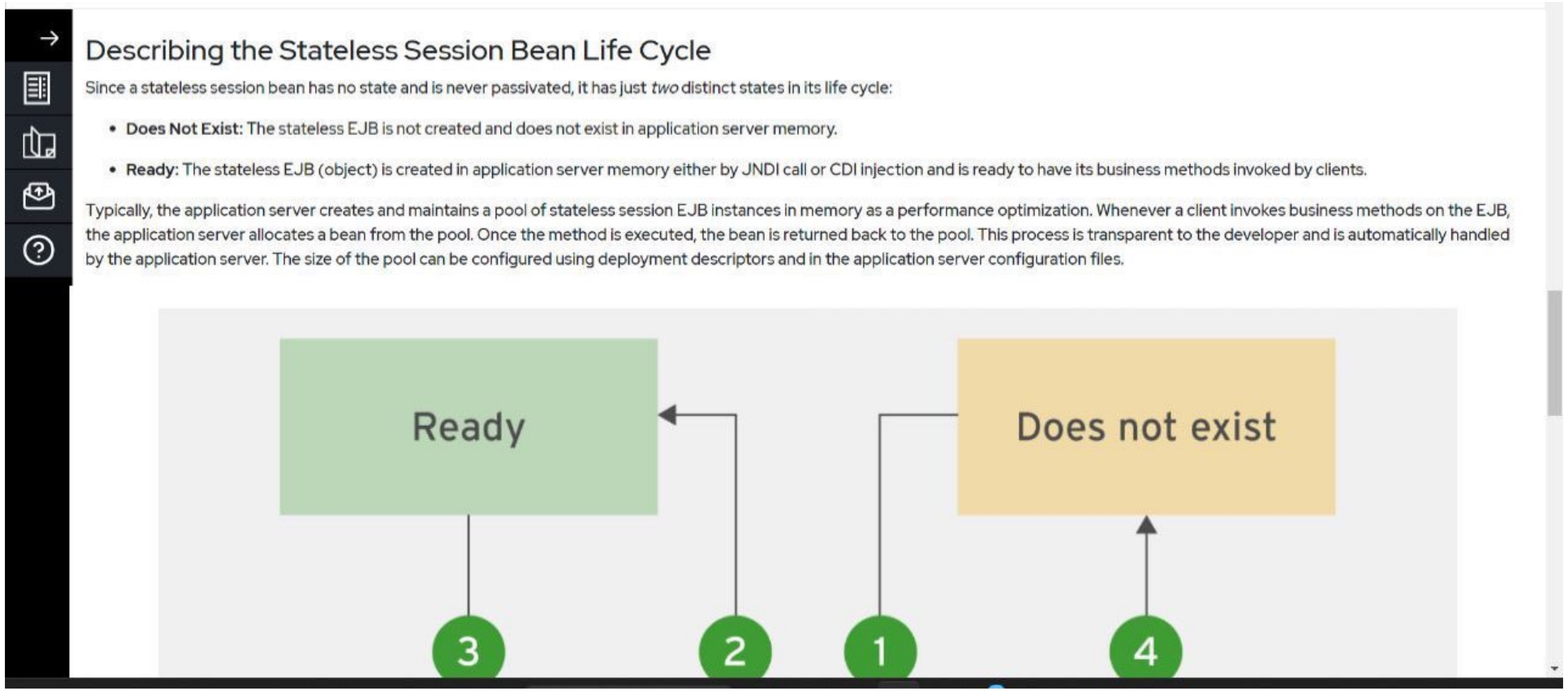
1 - The EJB moves to the Ready state after a JNDI call or a CDI injection.

2 - The container passivates the EJB if it is no longer being used.

3 - The container activates the EJB if any of its methods are invoked.

4 - The EJB is removed after a timeout, a client removal, or a container shutdown or crash.

5 - The EJB is removed after reaching the idle timeout threshold.

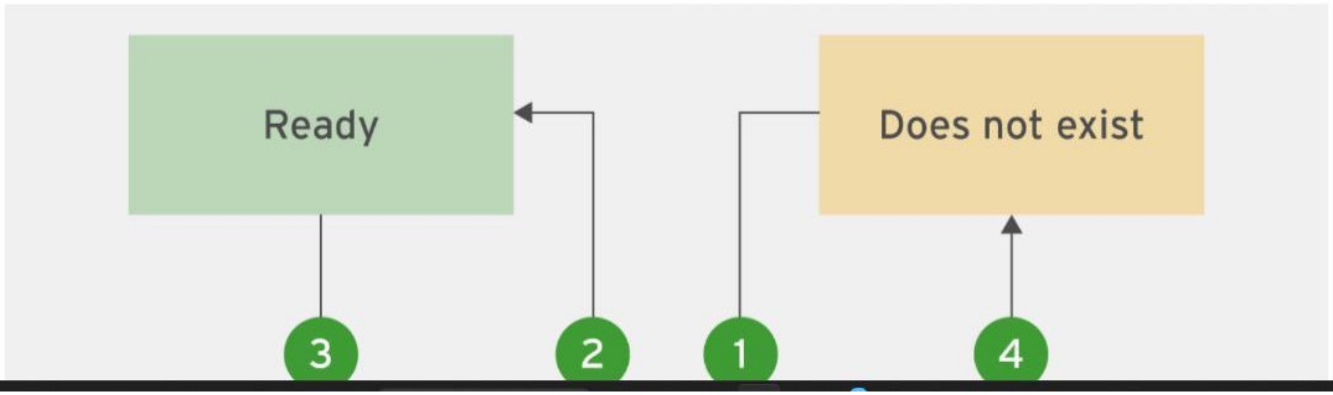


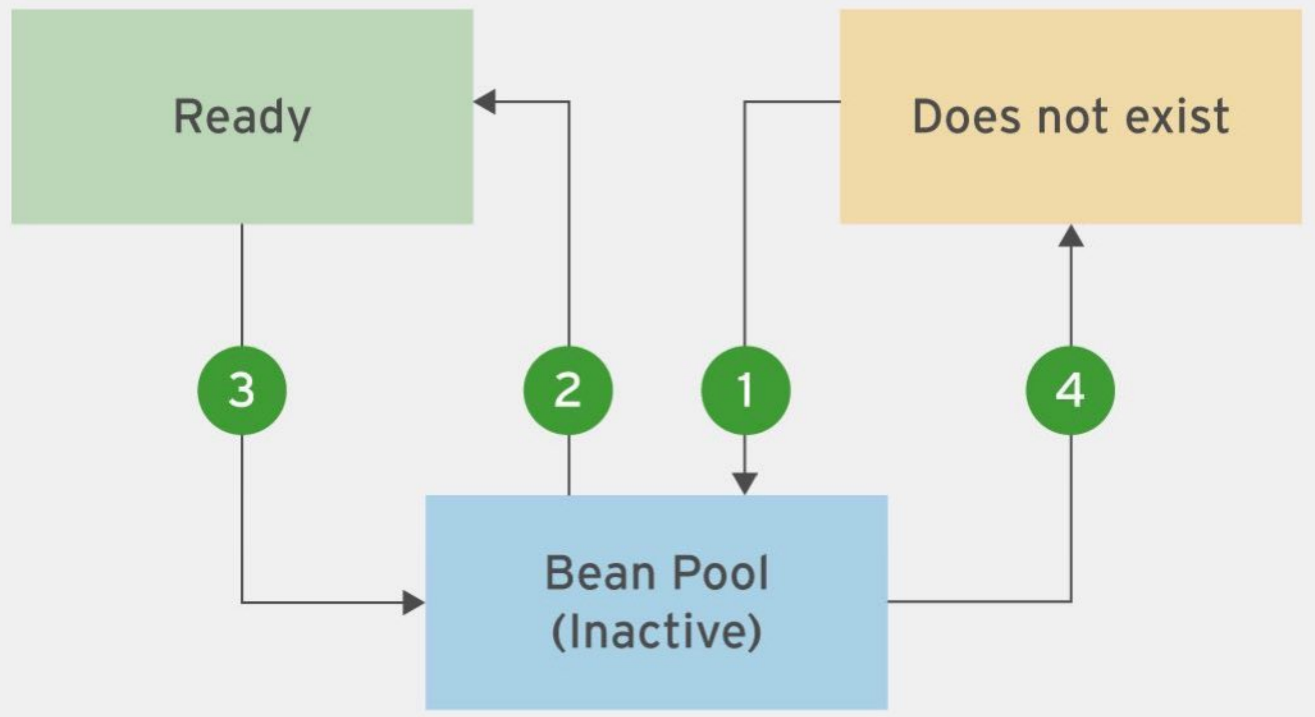
Describing the Stateless Session Bean Life Cycle

Since a stateless session bean has no state and is never passivated, it has just *two* distinct states in its life cycle:

- **Does Not Exist:** The stateless EJB is not created and does not exist in application server memory.
- **Ready:** The stateless EJB (object) is created in application server memory either by JNDI call or CDI injection and is ready to have its business methods invoked by clients.

Typically, the application server creates and maintains a pool of stateless session EJB instances in memory as a performance optimization. Whenever a client invokes business methods on the EJB, the application server allocates a bean from the pool. Once the method is executed, the bean is returned back to the pool. This process is transparent to the developer and is automatically handled by the application server. The size of the pool can be configured using deployment descriptors and in the application server configuration files.



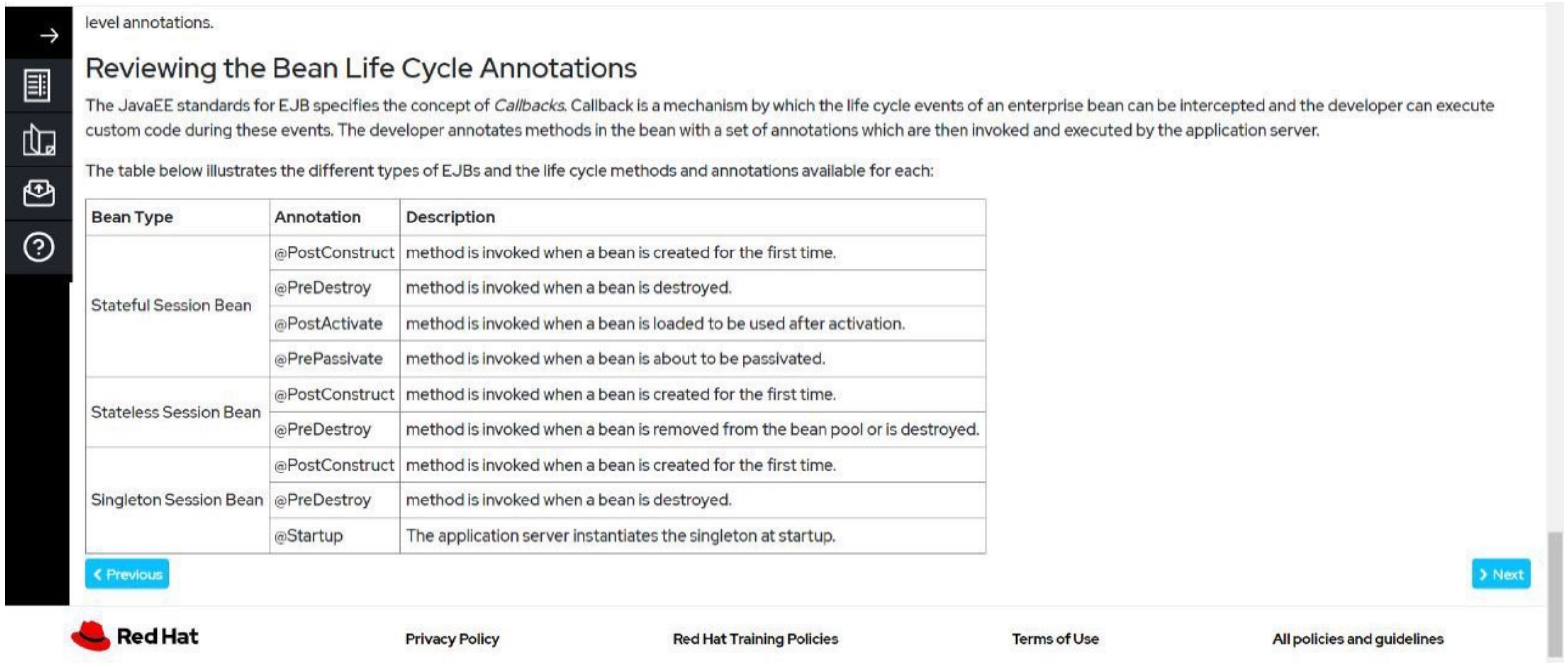


1 - The Container creates a bean and adds it to the pool.

2 - The Container allocates a bean from the pool when it is needed for execution.

3 - The Container returns the bean to the pool when it is no longer in use.

4 - Unused beans are destroyed after a timeout, a client removal, or a container shutdown or crash.



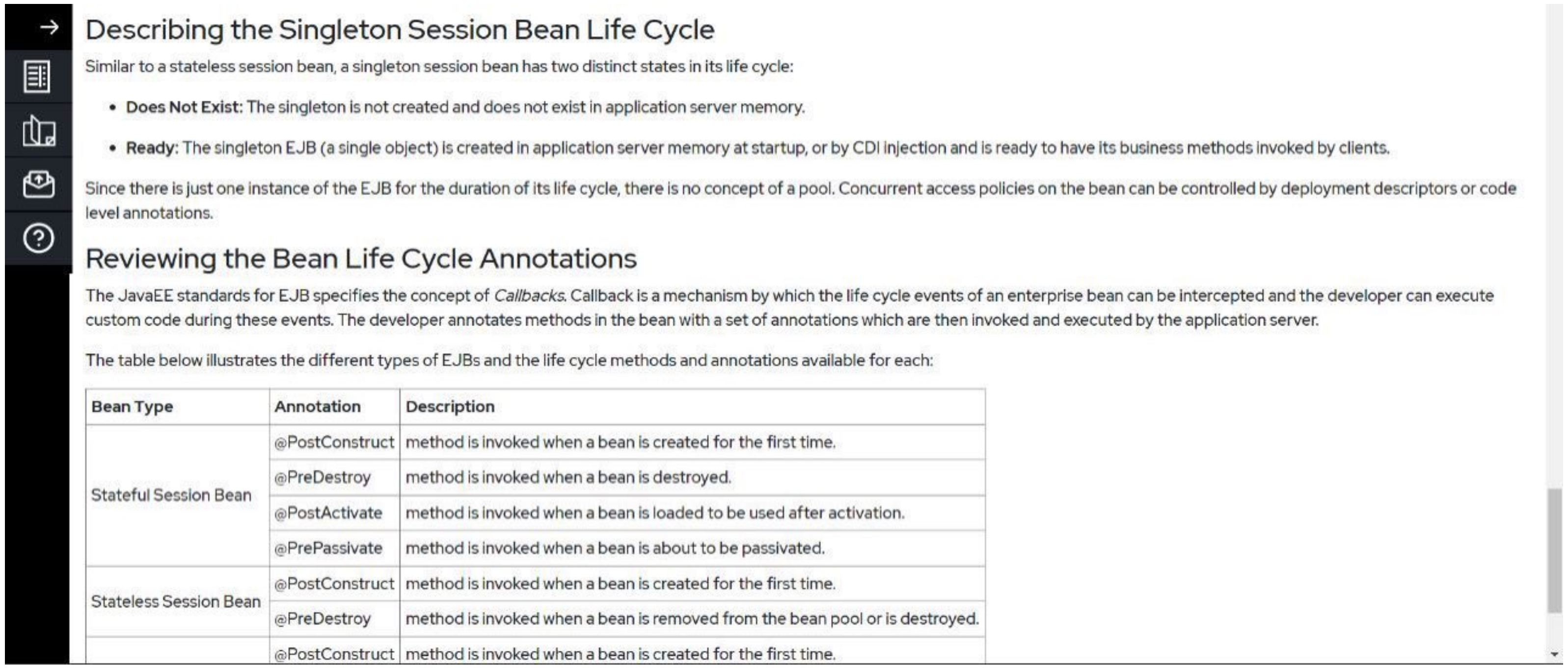
level annotations.

Reviewing the Bean Life Cycle Annotations

The JavaEE standards for EJB specifies the concept of *Callbacks*. Callback is a mechanism by which the life cycle events of an enterprise bean can be intercepted and the developer can execute custom code during these events. The developer annotates methods in the bean with a set of annotations which are then invoked and executed by the application server.

The table below illustrates the different types of EJBs and the life cycle methods and annotations available for each:

Bean Type	Annotation	Description
Stateful Session Bean	@PostConstruct	method is invoked when a bean is created for the first time.
	@PreDestroy	method is invoked when a bean is destroyed.
	@PostActivate	method is invoked when a bean is loaded to be used after activation.
	@PrePassivate	method is invoked when a bean is about to be passivated.
Stateless Session Bean	@PostConstruct	method is invoked when a bean is created for the first time.
	@PreDestroy	method is invoked when a bean is removed from the bean pool or is destroyed.
Singleton Session Bean	@PostConstruct	method is invoked when a bean is created for the first time.
	@PreDestroy	method is invoked when a bean is destroyed.
	@Startup	The application server instantiates the singleton at startup.



Describing the Singleton Session Bean Life Cycle

Similar to a stateless session bean, a singleton session bean has two distinct states in its life cycle:

- **Does Not Exist:** The singleton is not created and does not exist in application server memory.
- **Ready:** The singleton EJB (a single object) is created in application server memory at startup, or by CDI injection and is ready to have its business methods invoked by clients.

Since there is just one instance of the EJB for the duration of its life cycle, there is no concept of a pool. Concurrent access policies on the bean can be controlled by deployment descriptors or code level annotations.

Reviewing the Bean Life Cycle Annotations

The JavaEE standards for EJB specifies the concept of *Callbacks*. Callback is a mechanism by which the life cycle events of an enterprise bean can be intercepted and the developer can execute custom code during these events. The developer annotates methods in the bean with a set of annotations which are then invoked and executed by the application server.

The table below illustrates the different types of EJBs and the life cycle methods and annotations available for each:

Bean Type	Annotation	Description
Stateful Session Bean	@PostConstruct	method is invoked when a bean is created for the first time.
	@PreDestroy	method is invoked when a bean is destroyed.
	@PostActivate	method is invoked when a bean is loaded to be used after activation.
	@PrePassivate	method is invoked when a bean is about to be passivated.
Stateless Session Bean	@PostConstruct	method is invoked when a bean is created for the first time.
	@PreDestroy	method is invoked when a bean is removed from the bean pool or is destroyed.
	@PostConstruct	method is invoked when a bean is created for the first time.