Enterprise Java Beans

Table of Contents

[Types of EJBs 2](#_Toc100623486)

[Session Beans 3](#_Toc100623487)

[Stateful Session Beans 3](#_Toc100623488)

[Stateless Session Beans 4](#_Toc100623489)

[Singleton Session Beans 4](#_Toc100623490)

[Dependency Injection 6](#_Toc100623491)

[Lifecycle Callback Methods 8](#_Toc100623492)

[Stateful Session Beans 9](#_Toc100623493)

[Stateless Session Beans 10](#_Toc100623494)

[Singleton Session Beans 11](#_Toc100623495)

[Accessing EJBs 11](#_Toc100623496)

[Local Clients 12](#_Toc100623497)

[Remote Clients 13](#_Toc100623498)

**Enterprise Java Beans** (EJBs) are standard building blocks that encapsulate the business logic for enterprise applications. We have previously seen how servlets and JSP pages reside in a Web Container. EJBs on the other hand, reside inside an **EJB container**. The EJB Container handles the system-level services (transaction management, security authorization, etc.), so the EJB itself can concentrate on the business logic. The fact that EJBs are modularized also helps with reuse.

We should use EJBs:

* For scalable applications
* For transaction management
* For different types of clients like web clients, application clients, etc.

Unfortunately, the Tomcat server we have used so far is unable to deploy EJB containers. Because of this, in order to use EJBs, we must use the [Wildfly](https://www.wildfly.org/) server.

## Types of EJBs

There are two types of EJBs:

* **Session Beans** – These perform a task for the client, which may implement a web service.
* **Message-Drive Beans** – These act as a listener for a particular message type, such as Java Message Service API.

We will only be studying Session Beans.

## Session Beans

**Session Beans** encapsulate business logic that can be invoked by a client. Session beans are **not persistent**, i.e. data is not saved to a database. They can be of three types, stateful, stateless and singleton.

### Stateful Session Beans

In **stateful session beans**, the ‘state’ represents the instance variables of individual clients. Since the client directly ‘talks’ to the bean, this state is also called the **conversational state**.

The state is only retained for the duration of the client session. If the client **terminates** or removes the bean, the **session ends** and the state disappears.

Stateful session beans should be used when the bean needs to hold information that will be used in future method invocations by the same client.

@Stateful  
public class TestBean implements *TestInterface*{}

JAVA

### Stateless Session Beans

**Stateless session beans** are the opposite of stateful ones. Any information from the client side only exists for the duration of a single method invocation. This allows better scalability. Stateless session beans can also implement web services, unlike stateful ones.

Stateless session beans should be used when the bean’s state does not have any information for a specific client. The methods that are invoked should be **generic**, for example to send an email to confirm an order.

@Stateless  
public class TestBean implements *TestInterface*{}

JAVA

### Singleton Session Beans

**Singleton session beans** are a variant of stateful ones, but there is only **one bean session per application**. These beans exist to be **shared** by all clients, and the fact that they exist for the lifecycle of the application means they can be used for **start-up and shutdown tasks**. Singleton session beans can also implement web services.

*// for concurrent access*

@ConcurrencyManagement(ConcurrencyManagementType.*CONTAINER*)

@Singleton  
public class TestBean implements *TestInterface*{}

JAVA

Singleton session beans can be accessed by **multiple clients simultaneously**, which introduces some issues with writing data. To deal with this, we use locks. We can add a read or a write lock to specific methods.

The **read lock** allows multiple concurrent accesses to the method.

@Lock(LockType.*READ*)  
public void incrementCount() {  
 count++;  
}

JAVA

The **write lock** prevents the method from being accessed by other clients when one client has accessed it. This client may only use it for the specified period of time at most.

@Lock(LockType.*WRITE*)  
@AccessTimeout(value = 60, unit = TimeUnit.*SECONDS*)  
public void setCount(int count) { this.count = count; }

JAVA

## Dependency Injection

Consider the code below:

public class SampleClass {  
 SampleService1 service = new SampleService1();  
 SampleClass() {}  
 String message = service.getService1Info();  
}

public class SampleService1{  
 public String getService1Info() {  
 return "This is sample service 1";  
 }  
}

JAVA

This is an example of a **hard dependency**, where one class contains an object of another class. Hard dependencies make code inflexible and difficult to maintain, extend, reuse test or collaborate on.

Instead, we should use a **soft dependency**. Essentially, this means the first class should include an **interface** instead of a second class directly. This immediately removes all of the issues we see with hard dependencies.

public class SampleClass {  
 private *SampleService* service;  
 SampleClass(*SampleService* service) {  
 this.service = service;  
 }  
 String message = service.getServiceInfo();  
 public void printMessage() {  
 System.*out*.println(message);  
 }  
}

public interface *SampleService* {  
 public String getServiceInfo();  
}  
  
public class SampleService1 implements *SampleService* {  
 public String getServiceInfo() {  
 return "This is sample service 1";  
 }  
}

public class SampleService2 implements *SampleService* {  
 public String getServiceInfo() {  
 return "This is sample service 2";  
 }  
}  
  
public class SampleService3 implements *SampleService* {  
 public String getServiceInfo() {  
 return "This is sample service 3";  
 }  
}

JAVA

An alternative to this is to use **dependency injection**, where the exact object to be used is injected later on by an **injector**, either through the object’s constructor or through a setter method.

public class SampleClass {  
 private *SampleService* service;  
 public SampleClass() {};  
 public SampleClass(*SampleService* service) { *// constructor injection* this.service = service;  
 }  
 public void setService(*SampleService* service) { *// setter injection* this.service = service;  
 }  
 public void printMessage() {  
 System.*out*.println(message);  
 }  
}

public class Injector {  
 *SampleService* service2 = new SampleService2();  
 SampleClass sampleClass1 = new SampleClass(service2);SampleClass sampleClass2 = new SampleClass();  
  
 public Injector() {  
 sampleClass2.setService(service2);sampleClass2.printMessage();  
 }  
}

JAVA

## Lifecycle Callback Methods

There are several stages in the lifecycle of an EJB. At each of these stages, we can cause a method to be executed. These methods are not specified beforehand. Instead, we can create our own methods and simply add an **annotation** to cause that method to be executed. Depending on which annotation we use, we can cause the method to be executed at a different stage of the lifecycle.

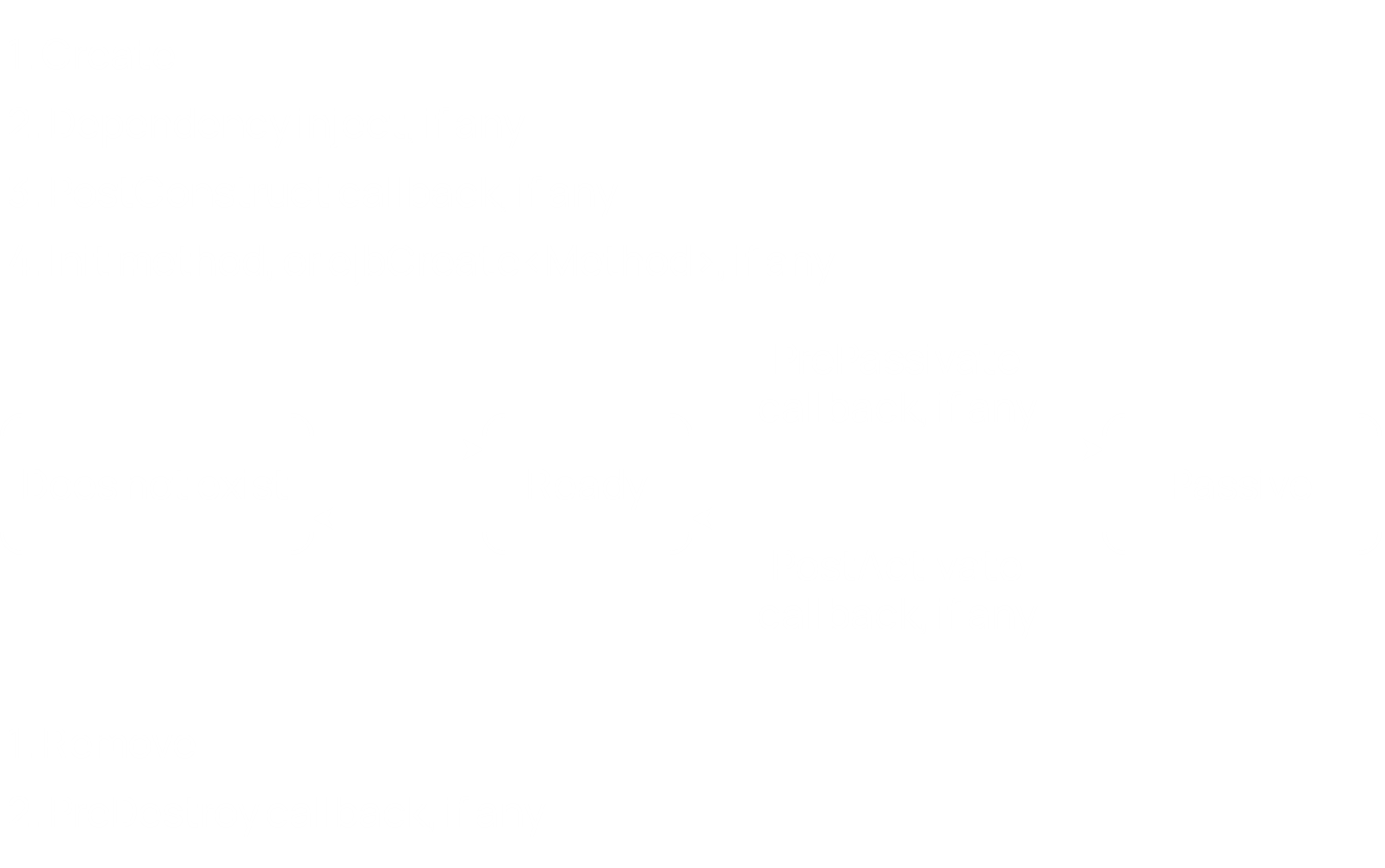
Note that all lifecycle methods must return void and cannot have any parameters.

The different annotations we can use are:

* @PostConstruct – These methods are invoked by the container on newly constructed bean instances after all dependency injection has completed and before the first business method is invoked.
* @Remove – These methods will cause the EJB to be removed once the method completes. The removal may be normal or abnormal.
* @PreDestroy – These methods are invoked after any @Remove method and before the container removes the EJB.
* @PostActivate – These methods are invoked after the container moves the bean from secondary storage to the active status.
* @PrePassivate – These methods are invoked before the bean is put in the passive state.

### Stateful Session Beans

For **Stateful Session Beans**, the lifecycle is initiated by the client obtaining a reference to the session bean. Next, we have dependency injections, followed by the @PostConstruct methods. At this point, the session bean is in the **ready state**, where business methods can be invoked.



The EJB container might decide to **deactivate** or **passivate** the bean by moving it from memory to secondary storage. Before doing this, the @PrePassivate methods are called. If the client calls a business method on a deactivated bean, it is brought back into memory. The @PreActivate methods are called, after which the bean is in the **ready stage**.

At the end of the lifecycle, the @Remove methods are called, after which the @PreDestroy methods are called. Once this is done, the bean is ready for **garbage collection**.

### Stateless Session Beans

A **Stateless Session Bean** is never passivated. Thus, it only has two stages, non-existent and ready.

The EJB container usually has a pool of stateless session beans, which is where the lifecycle begins. Like before, dependency injection and the @PreConstruct methods are invoked, which puts the bean in the **ready stage**. At the end of the bean’s lifecycle, the @PreDestroy method is invoked, and the bean is ready for garbage collection.

### Singleton Session Beans

**Singleton Session Beans** also have just two stages, non-existent and ready. The lifecycle starts when the container creates the singleton instance. If the @Startup annotation is used with the bean, then this happens at application deployment. This is followed by dependency injection and the @PostConstruct methods, at which point the bean is in the **ready stage**. At the end of the lifecycle, the @PreDestroy methods are called, which puts the bean up for garbage collection.

## Accessing EJBs

The business methods of a session bean can be exposed to clients in one of three ways, over a local view, a remote view or a no-interface view.

The difference between a local and a remote client is that a **local client** resides on the same machine, while a **remote client** resides on a different machine. Having remote clients gives us the benefit of loose coupling and increased scalability, but introduces some increased latency.

We also have the concept of **no-interface EJBs**, where, instead of exposing an interface for the client to use, the actual **classes** are exposed. In this case, the client has a reference to an instance of an enterprise bean either through dependency injection, or through the Java Naming and Directory Interface (JNDI) lookup.

### Local Clients

For **local clients**, the location of the enterprise bean is **not transparent**. If we choose to use **no-interface EJBs**, we just have to mark the EJB as stateless.

@Stateless  
public class TestBean {}

JAVA

Alternatively, we can create an **interface** and mark it as local. We can then create classes that implement that interface, with the classes also being marked as local.

@Local  
public interface *TestInterface* {}

JAVA

@Local(*TestInterface*.class)  
public class TestBean implements *TestInterface*{}

JAVA

To use TestBean in a servlet, we use the @EJB annotation.

@EJB  
*TestInterface* testInterface;

JAVA

We can directly use this interface and call its methods now, without having to deal with the actual classes. We are not having to initialize the class. The EJB container does that for us.

An alternative to the @EJB annotation is the @Inject annotation, but it is recommended to not use this.

However, what if we use the interface approach and we have multiple beans that implement that interface? In that case, there will be an **ambiguity error**. To avoid this, we can specify exactly which class to use.

@EJB(beanName = "TestBean")  
*TestInterface* testInterface;

JAVA

### Remote Clients

For **remote clients**, the process is exactly the same except that we use the @Remote attribute instead of the @Local attribute.