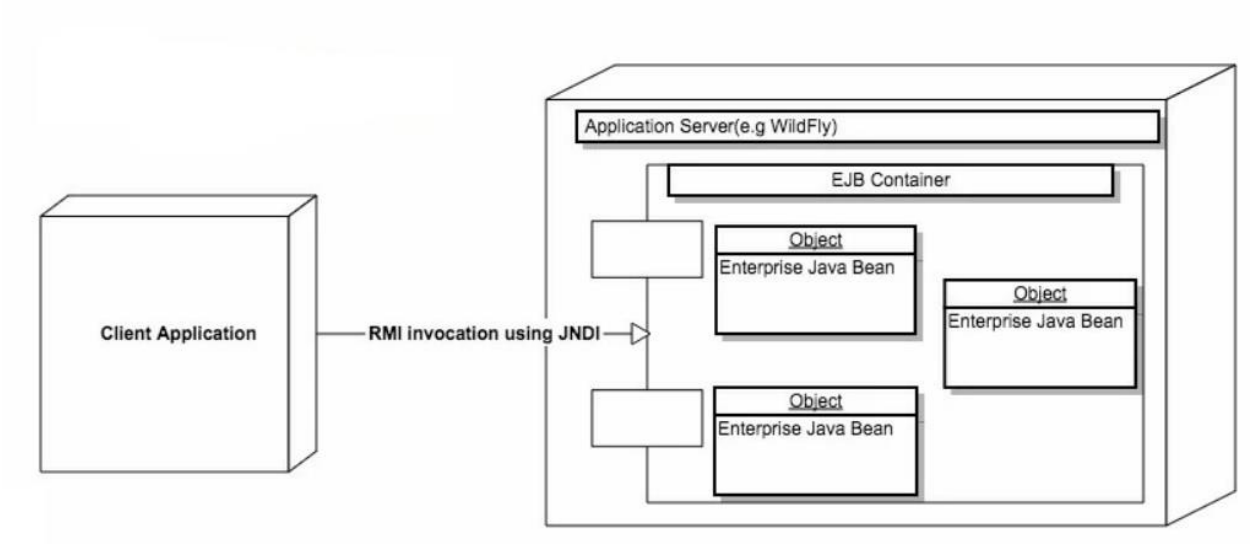
Introduction To Enterprise Javabeans

EJB Architecture



In this diagram, we can see the logical representation of the way EJBs are deployed and invoked by means of the remote method invocation (RMI). It is important to note the fact that EJB containers cannot be deployed without an application server. However, as from Java EE 7, it is now possible to configure an application in such a way that it is only made of Web components only. Further, this is known as the Web Profile within the Java EE space. Java EE 7 has got two types of profiles, the Web profile, and the Full Java EE profile. The Full Java EE profile is made up of the Web Profile and everything else that is not required by the Web Profile.

EJB is defined as an architecture for the development and deployment of component-based, robust, highly scalable business applications. By using EJB, you can write scalable, reliable, and secure applications without writing your own complex distributed component framework.

EJB is about rapid application development for the server side. You can quickly and easily construct server-side components in Java. This can be done by leveraging a prewritten distributed infrastructure provided by the industry.

EJB is designed to support application portability and reusability across Enterprise middleware services of any vendor.

BENEFITS OF EJB

Component portability - The EJB architecture provides a simple, elegant component container model. Java server components can be developed once and deployed in any EJB-compliant server.

Architecture independence - The EJB architecture is independent of any specific platform, proprietary protocol, or middleware infrastructure. Applications developed for one platform can be redeployed on other platforms.

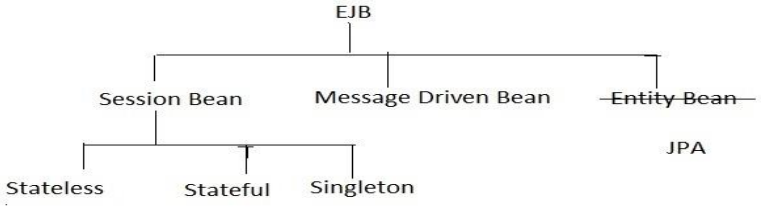
Developer productivity - The EJB architecture improves the productivity of application developers by standardizing and automating the use of complex infrastructure services such as transaction management and security checking. Developers can create complex applications by focusing on business logic rather than environmental and transactional issues.

Customization - Enterprise bean applications can be customized without access to the source code. Application behavior and runtime settings are defined through attributes that can be changed when the enterprise bean is deployed.

Multitier technology - The EJB architecture overlays existing infrastructure services.

Versatility and scalability - The EJB architecture can be used for small-scale or large-scale business transactions. As processing requirements grow, the enterprise beans can be migrated to more powerful operating environments.

Types of EJB



Accessing Enterprise Bean

In this tutorial you will learn that how to access Enterprise Beans (applicable only to session beans not to message-driven beans)

Enterprise beans are accessed by the client in two ways either through a no-interface view or through a business interface.

No-interface view

In this view the enterprise bean implemented class discloses the public methods to clients. Using this view enterprise bean implemented class's or any super classes of the implementation class's public methods may be accessed by an enterprise bean.

Business interface

This interface is an interface of Java programming language which keeps the business methods of enterprise bean.

To use a session bean client will have to required either bean's business's interface methods or enterprise bean's public methods which has a no-interface view.

How to use Enterprise Beans in clients

To know before the use of Enterprise Beans in clients let us first know about the clients. The can be Local clients, Remote clients and Web Service clients. To identify these clients they have some characteristic.

Characteristics of Local clients :

They must run in the same application.

Clients can be a web component or other enterprise bean.

Enterprise bean position which will be used is not diaphanous to the local client.

Characteristics of Remote clients :

An enterprise bean which will access the client, called remote client if it run on an other machine or JVM however, running on different JVM is not necessary.

Clients can be a web component, an application client, or other enterprise bean.

Enterprise bean position which will be used is diaphanous to the remote client.

Remote clients which will use an enterprise bean that, enterprise bean will must have to implement the business interface.

Web Service clients :

Java EE application can be accessed by a web service client in two manner :

A web service made with JAX-WS can be used by the client.

By invoking stateless session bean's business methods.

Enterprise Beans are the server side component that encapsulates the logic of fulfillment of the purpose of an application. Instance of these Enterprise Beans in clients can be use/get by either the **dependency injection** or **JNDI lookup**.

Dependency Injection is a Java programming language annotations using which a instance of an Enterprise Bean can be get in a easy way. It is used when a client run inside the Java EE server-managed environment. Annotation that is used to support the dependency injection to the enterprise beans or the applications (JavaServer Faces web applications, Java EE application clients, JAX-RS web services, or other enterprise beans) which are runs within the Java EE server managed environment is **javax.ejb.EJB**.

JNDI, Java Naming and Directory Interface syntax is also used to get an instance of an Enterprise Bean. It is used when an application is running outside a Java EE server managed environment must required to perform an explicit lookup.

Packaging Enterprise Beans

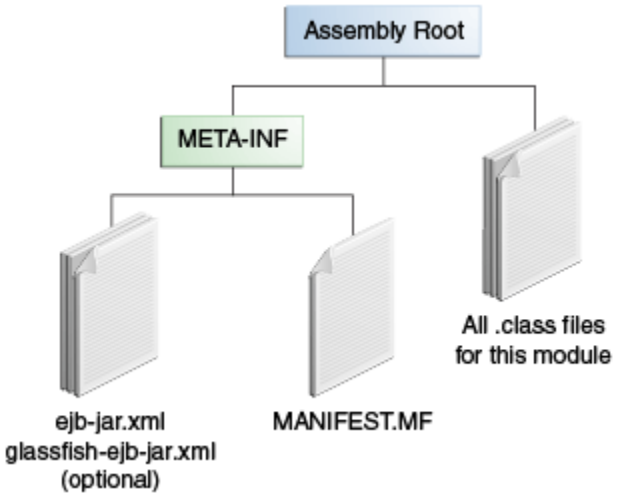
This section explains how enterprise beans can be packaged in EJB JAR or WAR modules.

Packaging Enterprise Beans in EJB JAR Modules

An EJB JAR file is portable and can be used for various applications.

To assemble a Java EE application, package one or more modules, such as EJB JAR files, into an EAR file, the archive file that holds the application. When deploying the EAR file that contains the enterprise bean's EJB JAR file, you also deploy the enterprise bean to GlassFish Server. You can also deploy an EJB JAR that is not contained in an EAR file. Figure 5-2 shows the contents of an EJB JAR file.

Figure 5-2 Structure of an Enterprise Bean JAR



Description of "Figure 5-2 Structure of an Enterprise Bean JAR"

Packaging Enterprise Beans in WAR Modules

Enterprise beans often provide the business logic of a web application. In these cases, packaging the enterprise bean within the web application's WAR module simplifies deployment and application organization. Enterprise beans may be packaged within a WAR module as Java programming language class files or within a JAR file that is bundled within the WAR module.

To include enterprise bean class files in a WAR module, the class files should be in the WEB-INF/classes directory.

To include a JAR file that contains enterprise beans in a WAR module, add the JAR to the WEB-INF/lib directory of the WAR module.

WAR modules that contain enterprise beans do not require an ejb-jar.xml deployment descriptor. If the application uses ejb-jar.xml, it must be located in the WAR module's WEB-INF directory.

JAR files that contain enterprise bean classes packaged within a WAR module are not considered EJB JAR files, even if the bundled JAR file conforms to the format of an EJB JAR file. The enterprise beans contained within the JAR file are semantically equivalent to enterprise beans located in the WAR module's WEB-INF/classes directory, and the environment namespace of all the enterprise beans are scoped to the WAR module.

For example, suppose that a web application consists of a shopping cart enterprise bean, a credit card–processing enterprise bean, and a Java servlet front end. The shopping cart bean exposes a local, no-interface view and is defined as follows:

```
package com.example.cart;
```

@Stateless

public class CartBean { ... }

The credit card–processing bean is packaged within its own JAR file, cc.jar, exposes a local, no-interface view, and is defined as follows:

package com.example.cc;

@Stateless

public class CreditCardBean { ... }

The servlet, com.example.web.StoreServlet, handles the web front end and uses both CartBean and CreditCardBean. The WAR module layout for this application is as follows:

WEB-INF/classes/com/example/cart/CartBean.class

WEB-INF/classes/com/example/web/StoreServlet

WEB-INF/lib/cc.jar

WEB-INF/ejb-jar.xml

WEB-INF/web.xml

Working With Session Beans

When to Use Session Beans

In general, you should use a session bean if the following circumstances hold:

- At any given time, only one client has access to the bean instance.
- The state of the bean is not persistent, existing only for a short period (perhaps a few hours).
- The bean implements a web service.
- Stateful session beans are appropriate if any of the following conditions are true:
- The bean's state represents the interaction between the bean and a specific client.
- The bean needs to hold information about the client across method invocations.
- The bean mediates between the client and the other components of the application, presenting a simplified view to the client.
- Behind the scenes, the bean manages the work flow of several enterprise beans.
- To improve performance, you might choose a stateless session bean if it has any of these

traits:

- The bean's state has no data for a specific client.
- In a single method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to send an email that confirms an online order.

Types of Session Bean

There are three types of session beans: stateful, stateless and singleton.

Stateful Session Beans

The state of an object consists of the values of its instance variables. In a **stateful** session bean, the instance variables represent the state of a unique client-bean session. Because the client interacts ("talks") with its bean, this state is often called the **conversational state**.

The state is retained for the duration of the client-bean session. If the client removes the bean or terminates, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends there is no need to retain the state.

Stateless Session Beans

A **stateless** session bean does not maintain a conversational state with the client. When a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client, but only for the duration of the invocation. When the method is finished, the client-specific state should not be retained. Clients may, however, change the state of instance variables in pooled stateless beans, and this state is held over to the next invocation of the pooled stateless bean. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. That is, the state of a stateless session bean should apply accross all clients.

Because stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

Singleton Session Beans

A *singleton session bean* is instantiated once per application and exists for the lifecycle of the application. Singleton session beans are designed for circumstances in which a single enterprise bean instance is shared across and concurrently accessed by clients.

Singleton session beans offer similar functionality to stateless session beans but differ from them in that there is only one singleton session bean per application, as opposed to a pool of stateless

session beans, any of which may respond to a client request. Like stateless session beans, singleton session beans can implement web service endpoints.

Singleton session beans maintain their state between client invocations but are not required to maintain their state across server crashes or shutdowns.

Applications that use a singleton session bean may specify that the singleton should be instantiated upon application startup, which allows the singleton to perform initialization tasks for the application. The singleton may perform cleanup tasks on application shutdown as well, because the singleton will operate throughout the lifecycle of the application.

The Lifecycles of Enterprise Beans

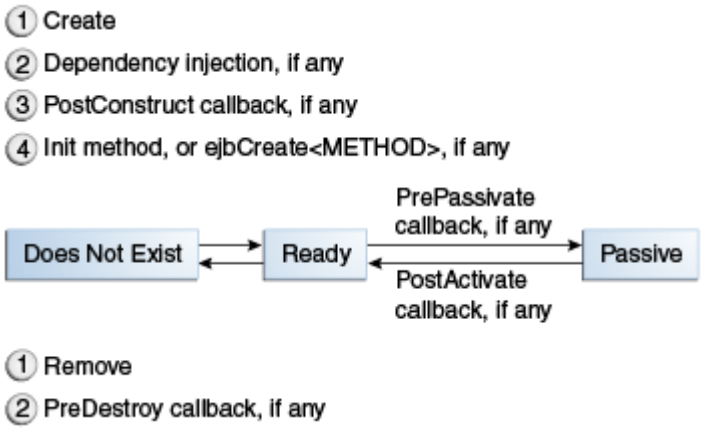
An enterprise bean goes through various stages during its lifetime, or lifecycle. Each type of enterprise bean (stateful session, stateless session, singleton session, or message-driven) has a different lifecycle.

The descriptions that follow refer to methods that are explained along with the code examples in the next two chapters. If you are new to enterprise beans, you should skip this section and run the code examples first.

The Lifecycle of a Stateful Session Bean

Figure 35-2 illustrates the stages that a stateful session bean passes through during its lifetime. The client initiates the lifecycle by obtaining a reference to a stateful session bean. The container performs any dependency injection and then invokes the method annotated with `@PostConstruct`, if any. The bean is now ready to have its business methods invoked by the client.

Figure 35-2 Lifecycle of a Stateful Session Bean



While in the ready stage, the EJB container may decide to deactivate, or passivate, the bean by moving it from memory to secondary storage. (Typically, the EJB container uses a least-recently-used algorithm to select a bean for passivation.) The EJB container invokes the method annotated `@PrePassivate`, if any, immediately before passivating it. If a client invokes a business method on the bean while it is in the passive stage, the EJB container activates the bean, calls the method annotated `@PostActivate`, if any, and then moves it to the ready stage.

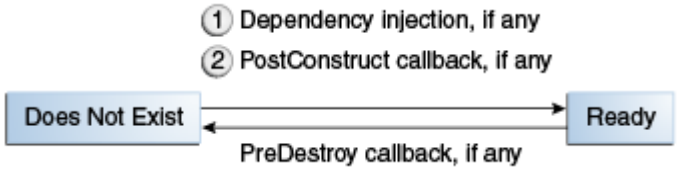
At the end of the lifecycle, the client invokes a method annotated `@Remove`, and the EJB container calls the method annotated `@PreDestroy`, if any. The bean's instance is then ready for garbage collection.

Your code controls the invocation of only one lifecycle method: the method annotated `@Remove`. All other methods in Figure 35-2 are invoked by the EJB container.

The Lifecycle of a Stateless Session Bean

Because a stateless session bean is never passivated, its lifecycle has only two stages: nonexistent and ready for the invocation of business methods. Figure 35-3 illustrates the stages of a stateless session bean.

Figure 35-3 Lifecycle of a Stateless or Singleton Session Bean



The EJB container typically creates and maintains a pool of stateless session beans, beginning the stateless session bean's lifecycle. The container performs any dependency injection and then invokes the method annotated `@PostConstruct`, if it exists. The bean is now ready to have its business methods invoked by a client.

At the end of the lifecycle, the EJB container calls the method annotated `@PreDestroy`, if it exists. The bean's instance is then ready for garbage collection.

The Lifecycle of a Singleton Session Bean

Like a stateless session bean, a singleton session bean is never passivated and has only two stages, nonexistent and ready for the invocation of business methods, as shown in Figure 35-3.

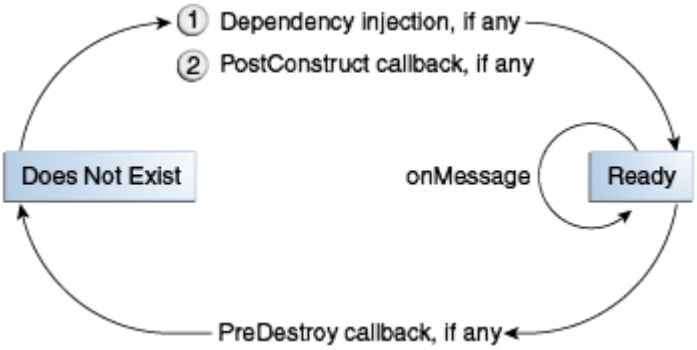
The EJB container initiates the singleton session bean lifecycle by creating the singleton instance. This occurs upon application deployment if the singleton is annotated with the `@Startup` annotation. The container performs any dependency injection and then invokes the method annotated `@PostConstruct`, if it exists. The singleton session bean is now ready to have its business methods invoked by the client.

At the end of the lifecycle, the EJB container calls the method annotated `@PreDestroy`, if it exists. The singleton session bean is now ready for garbage collection.

The Lifecycle of a Message-Driven Bean

Figure 35-4 illustrates the stages in the lifecycle of a message-driven bean.

Figure 35-4 Lifecycle of a Message-Driven Bean



The EJB container usually creates a pool of message-driven bean instances. For each instance, the EJB container performs these tasks.

If the message-driven bean uses dependency injection, the container injects these references before instantiating the instance.

The container calls the method annotated `@PostConstruct`, if any.

Like a stateless session bean, a message-driven bean is never passivated and has only two states: nonexistent and ready to receive messages.

At the end of the lifecycle, the container calls the method annotated `@PreDestroy`, if any. The bean's instance is then ready for garbage collection.

Working with Message Driven Beans

What Is a Message-Driven Bean?

A **message-driven bean** is an enterprise bean that allows Java EE applications to process messages asynchronously. This type of bean normally acts as a JMS message listener, which is similar to an event listener but receives JMS messages instead of events. The messages can be sent by any Java EE component (an application client, another enterprise bean, or a web component) or by a JMS application or system that does not use Java EE technology. Message-driven beans can process JMS messages or other kinds of messages.

What Makes Message-Driven Beans Different from Session Beans?

The most visible difference between message-driven beans and session beans is that clients do not access message-driven beans through interfaces. Interfaces are described in the section Accessing Enterprise Beans. Unlike a session bean, a message-driven bean has only a bean class.

In several respects, a message-driven bean resembles a stateless session bean.

- A message-driven bean's instances retain no data or conversational state for a specific client.
- All instances of a message-driven bean are equivalent, allowing the EJB container to assign a message to any message-driven bean instance. The container can pool these instances to allow streams of messages to be processed concurrently.
- A single message-driven bean can process messages from multiple clients.

The instance variables of the message-driven bean instance can contain some state across the handling of client messages, such as a JMS API connection, an open database connection, or an object reference to an enterprise bean object.

Client components do not locate message-driven beans and invoke methods directly on them. Instead, a client accesses a message-driven bean through, for example, JMS by sending messages to the message destination for which the message-driven bean class is the MessageListener. You assign a message-driven bean's destination during deployment by using GlassFish Server resources.

Message-driven beans have the following characteristics.

- They execute upon receipt of a single client message.
- They are invoked asynchronously.
- They are relatively short-lived.
- They do not represent directly shared data in the database, but they can access and update this data.
- They can be transaction-aware.
- They are stateless.

When a message arrives, the container calls the message-driven bean's onMessage method to process the message. The onMessage method normally casts the message to one of the five JMS message types and handles it in accordance with the application's business logic. The onMessage method can call helper methods or can invoke a session bean to process the information in the message or to store it in a database.

A message can be delivered to a message-driven bean within a transaction context, so all operations within the onMessage method are part of a single transaction. If message processing is rolled back, the message will be redelivered.

When to Use Message-Driven Beans

Session beans allow you to send JMS messages and to receive them synchronously but not asynchronously. To avoid tying up server resources, do not to use blocking synchronous receives in a server-side component; in general, JMS messages should not be sent or received synchronously. To receive messages asynchronously, use a message-driven bean.

Example of Message driven bean

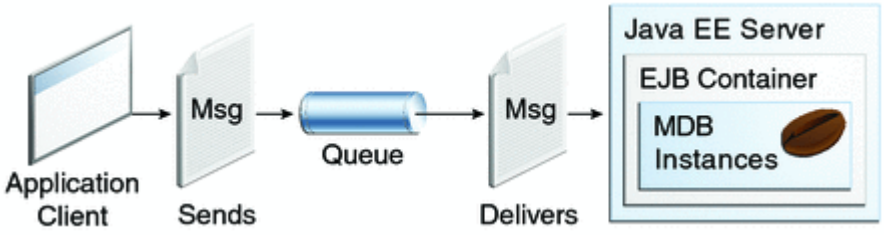
The simplemessage application has the following components:

SimpleMessageClient: An application client that sends several messages to a queue

SimpleMessageBean: A message-driven bean that asynchronously receives and processes the messages that are sent to the queue

Figure 25-1 illustrates the structure of this application. The application client sends messages to the queue, which was created administratively using the Administration Console. The JMS provider (in this case, the GlassFish Server) delivers the messages to the instances of the message-driven bean, which then processes the messages.

Figure 25-1 The simplemessage Application



The source code for this application is in the *tut-install*/examples/ejb/simplemessage/ directory.

The simplemessage Application Client

The SimpleMessageClient sends messages to the queue that the SimpleMessageBean listens to. The client starts by injecting the connection factory and queue resources:

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;


@Resource(mappedName="jms/Queue")
private static Queue queue;
```

Next, the client creates the connection, session, and message producer:

```
connection = connectionFactory.createConnection();
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
messageProducer = session.createProducer(queue);
```

Finally, the client sends several messages to the queue:

```
message = session.createTextMessage();


for (int i = 0; i < NUM_MSGS; i++) {
    message.setText("This is message " + (i + 1));
    System.out.println("Sending message: " + message.getText());
    messageProducer.send(message);
}
```

The Message-Driven Bean Class

The code for the SimpleMessageBean class illustrates the requirements of a message-driven bean class:

It must be annotated with the @MessageDriven annotation if it does not use a deployment descriptor.

The class must be defined as public.

The class cannot be defined as abstract or final.

It must contain a public constructor with no arguments.

It must not define the finalize method.

It is recommended, but not required, that a message-driven bean class implement the message listener interface for the message type it supports. A bean that supports the JMS API implements the javax.jms.MessageListener interface.

Unlike session beans and entities, message-driven beans do not have the remote or local interfaces that define client access. Client components do not locate message-driven beans and invoke methods on them. Although message-driven beans do not have business methods, they may contain helper methods that are invoked internally by the onMessage method.

For the GlassFish Server, the @MessageDriven annotation typically contains a mappedName element that specifies the JNDI name of the destination from which the bean will consume messages. For complex message-driven beans, there can also be an activationconfig element containing @ActivationConfigProperty annotations used by the bean.

A message-driven bean can also inject a MessageDrivenContext resource. Commonly you use this resource to call the setRollbackOnly method to handle exceptions for a bean that uses container-managed transactions.

Therefore, the first few lines of the SimpleMessageBean class look like this:

```
@MessageDriven(mappedName="jms/Queue", activationConfig =  {
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
                              propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(propertyName = "destinationType",
                              propertyValue = "javax.jms.Queue")
})
public class SimpleMessageBean implements MessageListener {
    @Resource
    private MessageDrivenContext mdc;
    ...
```

NetBeans IDE typically creates a message-driven bean with a default set of @ActivationConfigProperty settings. You can delete those you do not need, or add others. Table 25-1 lists commonly used properties.

Table 25-1 @ActivationConfigProperty Settings for Message-Driven Beans

Property Name	Description
acknowledgeMode	Acknowledgment mode; see Controlling Message Acknowledgment for information
destinationType	Either javax.jms.Queue or javax.jms.Topic
subscriptionDurability	For durable subscribers, set to Durable; see Creating Durable Subscriptions for information
clientId	For durable subscribers, the client ID for the connection

subscriptionName	For durable subscribers, the name of the subscription
messageSelector	A string that filters messages; see JMS Message Selectors for information, and see An Application That Uses the JMS API with a Session Bean for an example
addressList	Remote system or systems with which to communicate; see An Application Example That Consumes Messages from a Remote Server for an example

The onMessage Method

When the queue receives a message, the EJB container invokes the message listener method or methods. For a bean that uses JMS, this is the onMessage method of the MessageListener interface.

A message listener method must follow these rules:

The method must be declared as public.

The method must not be declared as final or static.

The onMessage method is called by the bean's container when a message has arrived for the bean to service. This method contains the business logic that handles the processing of the message. It is the message-driven bean's responsibility to parse the message and perform the necessary business logic.

The onMessage method has a single argument: the incoming message.

The signature of the onMessage method must follow these rules:

The return type must be void.

The method must have a single argument of type javax.jms.Message.

In the SimpleMessageBean class, the onMessage method casts the incoming message to a TextMessage and displays the text:

```
public void onMessage(Message inMessage) {
    TextMessage msg = null;
```

```
    try {
        if (inMessage instanceof TextMessage) {
            msg = (TextMessage) inMessage;
            logger.info("MESSAGE BEAN: Message received: " +
                msg.getText());
        } else {
            logger.warning("Message of wrong type: " +
                inMessage.getClass().getName());
        }
    } catch (JMSException e) {
        e.printStackTrace();
        mdc.setRollbackOnly();
    } catch (Throwable te) {
        te.printStackTrace();
    }
}
```

Running the simplemessage Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the simplemessage example.

Administered Objects for the simplemessage Example

This example requires the following:

A JMS connection factory resource

A JMS destination resource

Java Message Service Concepts and have not deleted the resources, you already have these resources. Otherwise, the resources will be created automatically when you deploy the application.

To Run the simplemessage Application Using NetBeans IDE

From the File menu, choose Open Project.

In the Open Project dialog, navigate to:

tut-install/examples/ejb/

Select the simplemessage folder.

Select the Open as Main Project check box and the Open Required Projects check box.

Click Open Project.

In the Projects tab, right-click the simplemessage project and choose Build.

This task packages the application client and the message-driven bean, then creates a file named simplemessage.ear in the dist directory.

Right-click the project and choose Run.

This command creates any needed resources, deploys the project, returns a JAR file named simplemessageClient.jar, and then executes it.

The output of the application client in the Output pane looks like this (preceded by application client container output):

Sending message: This is message 1

Sending message: This is message 2

Sending message: This is message 3

To see if the bean received the messages,

check <install_dir>/domains/domain1/logs/server.log.

The output from the message-driven bean appears in the server log (*domain-dir*/logs/server.log), wrapped in logging information.

MESSAGE BEAN: Message received: This is message 1

MESSAGE BEAN: Message received: This is message 2

MESSAGE BEAN: Message received: This is message 3

The received messages may appear in a different order from the order in which they were sent.

Interceptors

This is an example of how to use the javax.interceptor.* in an EJB.
Interceptors are used, as the name suggests, to intercept ejb methods calls using methods annotated with interceptor annotation (@AroundInvoke , @AroundTimeout , @PostConstruct etc).
An interceptor method is called by Ejb Container before ejb method call it is intercepting.
The Interceptors specification defines two kinds of interception points:

- business method interception, and
- lifecycle callback interception.

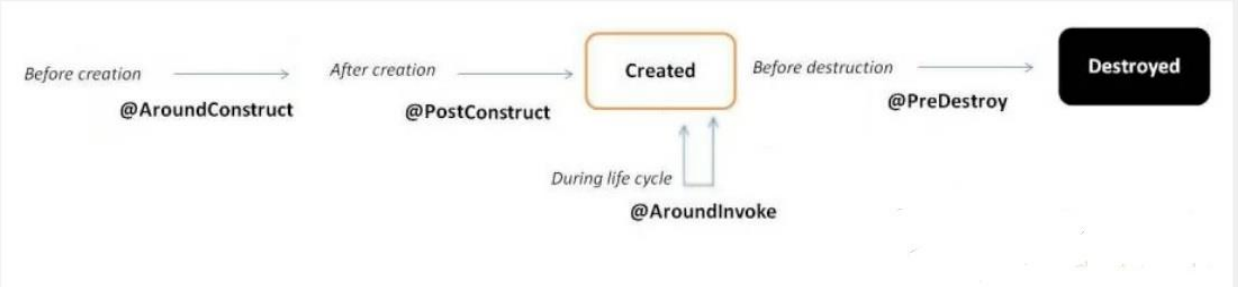


Figure 1: Interceptor Types

Interceptor methods can be applied or bound at three levels.

- **Default** − Default interceptor is invoked for every bean within deployment.Default interceptor can be applied only via xml (ejb-jar.xml).
- **Class** − Class level interceptor is invoked for every method of the bean. Class level interceptor can be applied both by annotation of via xml(ejb-jar.xml).

- **Method**– Method level interceptor is invoked for a particular method of the bean. Method level interceptor can be applied both by annotation of via xml(ejb-jar.xml).

We are discussing Class level interceptor here.

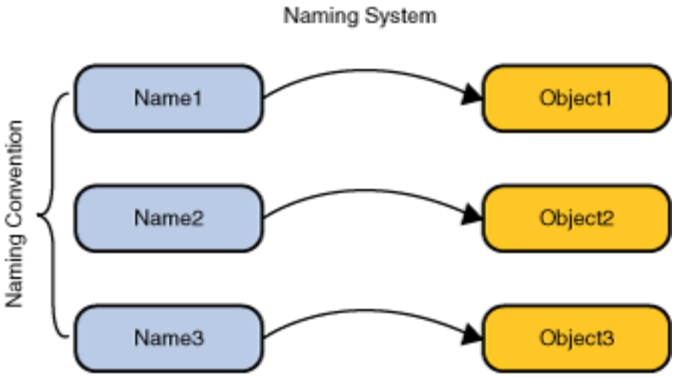
- Example already shared in google classroom

Java Naming and Directory Interface

What is Naming Service?

Naming Concepts

A fundamental facility in any computing system is the *naming service*--the means by which names are associated with objects and objects are found based on their names. When using almost any computer program or system, you are always naming one object or another. For example, when you use an electronic mail system, you must provide the name of the recipient. To access a file in the computer, you must supply its name. A naming service allows you to look up an object given its name.



A naming service's primary function is to map people friendly names to objects, such as addresses, identifiers, or objects typically used by computer programs.

For example, the Internet Domain Name System (DNS) maps machine names to IP Addresses:

www.example.com ==> 192.0.2.5

Bindings

The association of a name with an object is called a *binding*. A file name is *bound* to a file.

JNDI Naming

In a distributed application, components need to access other components and resources such as databases. For example, a servlet might invoke remote methods on an enterprise bean that retrieves information from a database. In the J2EE platform, the Java Naming and Directory Interface (JNDI) naming service enables components to locate other components and resources. To locate a JDBC resource, for example, an enterprise bean invokes the JNDI lookup method. The JNDI naming service maintains a set of bindings that relate names to objects. The lookup method passes a JNDI name parameter and returns the related object.

JNDI provides a *naming context*, which is a a set of name-to-object bindings. All naming operations are relative to a context. A name that is bound within a context is the JNDI name of the object. In Specifying a Resource Reference, for example, the JNDI name for the JDBC resource (or data source) is jdbc/ejbTutorialDB. A Context object provides the methods for binding names to objects, unbinding names from objects, renaming objects, and listing the bindings. JNDI also provides subcontext functionality. Much like a directory in a file system, a *subcontext* is a context within a context. This hierarchical structure permits better organization of information. For naming services that support subcontexts, the Context class also provides methods for creating and destroying subcontexts.

Table 31-1 describes JNDI subcontexts for connection factories in the Sun Java System Application Server Platform Edition 8.

Table 31-1 JNDI Subcontexts for Connection Factories

Resource Manager Type	Connection Factory Type	JNDI Subcontext
JDBC	javax.sql.DataSource	java:comp/env/jdbc
JMS	javax.jms.TopicConnectionFactory javax.jms.QueueConnectionFactory	java:comp/env/jms
JavaMail	javax.mail.Session	java:comp/env/mail
URL	java.net.URL	java:comp/env/url
Connector	javax.resource.cci.ConnectionFactory	java:comp/env/eis

JAXR Resource Adapter	javax.xml.registry.ConnectionFactory	java:comp/env/eis/JAXR

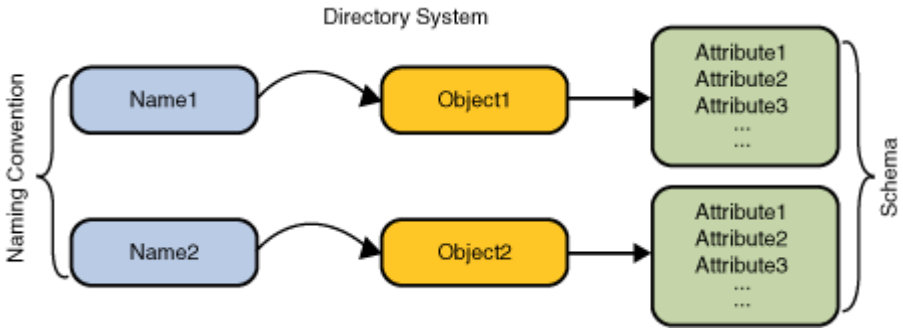
What is Directory Service?

Directory Concepts

Many naming services are extended with a *directory service*. A directory service associates names with objects and also associates such objects with *attributes*.

directory service = naming service + objects containing attributes

You not only can look up an object by its name but also get the object's attributes or *search* for the object based on its attributes.



An example is the telephone company's directory service. It maps a subscriber's name to his address and phone number. A computer's directory service is very much like a telephone company's directory service in that both can be used to store information such as telephone numbers and addresses. The computer's directory service is much more powerful, however, because it is available online and can be used to store a variety of information that can be utilized by users, programs, and even the computer itself and other computers.

A *directory object* represents an object in a computing environment. A directory object can be used, for example, to represent a printer, a person, a computer, or a network. A directory object contains *attributes* that describe the object that it represents.

Attributes

A directory object can have *attributes*. For example, a printer might be represented by a directory object that has as attributes its speed, resolution, and color. A user might be represented by a directory object that has as attributes the user's e-mail address, various telephone numbers, postal mail address, and computer account information.

An attribute has an *attribute identifier* and a set of *attribute values*. An attribute identifier is a token that identifies an attribute independent of its values. For example, two different computer accounts might have a "mail" attribute; "mail" is the attribute identifier. An attribute value is the contents of the attribute. The email address, for example, might have:

Attribute Identifier : Attribute Value

mail john.smith@example.com

Directories and Directory Services

A *directory* is a connected set of directory objects. A *directory service* is a service that provides operations for creating, adding, removing, and modifying the attributes associated with objects in a directory. The service is accessed through its own interface.

Many examples of directory services are possible.

Network Information Service (NIS)

NIS is a directory service available on the UNIX operating system for storing system-related information, such as that relating to machines, networks, printers, and users.

Oracle Directory Server

The Oracle Directory Server is a general-purpose directory service based on the Internet standard LDAP.

Java Naming and Directory Interface (JNDI)

This section discusses the Java Naming and Directory Interface (JNDI). JNDI is an application programming interface (API) for accessing different kinds of naming and directory services. J2EE components locate objects by invoking the JNDI lookup method.

This section covers the following topics:

JNDI Names and Resources

J2EE Naming Services

Naming References and Binding Information

JNDI Names and Resources

JNDI is the acronym for the Java Naming and Directory Interface API. By making calls to this API, applications locate resources and other program objects. A resource is a program object that provides connections to systems, such as database servers and messaging systems. (A JDBC resource is sometimes referred to as a data source.) Each resource object is identified by a unique, people-friendly name, called the JNDI name. A resource object and its JNDI name are bound together by the naming and directory service, which is included with the Application Server. To create a new resource, a new name-object binding is entered into the JNDI.

J2EE Naming Services

A JNDI name is a people-friendly name for an object. These names are bound to their objects by the naming and directory service that is provided by a J2EE server. Because J2EE components access this service through the JNDI API, the object usually uses its JNDI name. For example, the JNDI name of the Derby database is jdbc/Derby. When it starts up, the Application Server reads information from the configuration file and automatically adds JNDI database names to the name space.

J2EE application clients, enterprise beans, and web components are required to have access to a JNDI naming environment.

The application component's naming environment is a mechanism that allows customization of the application component's business logic during deployment or assembly. Use of the application component's environment allows the application component to be customized without the need to access or change the application component's source code.

A J2EE container implements the application component's environment, and provides it to the application component instance as a JNDI naming context. The application component's environment is used as follows:

The application component's business methods access the environment using the JNDI interfaces. The application component provider declares in the deployment descriptor all the environment entries that the application component expects to be provided in its environment at runtime.

The container provides an implementation of the JNDI naming context that stores the application component environment. The container also provides the tools that allow the deployer to create and manage the environment of each application component.

A deployer uses the tools provided by the container to initialize the environment entries that are declared in the application component's deployment descriptor. The deployer sets and modifies the values of the environment entries.

The container makes the environment naming context available to the application component instances at runtime. The application component's instances use the JNDI interfaces to obtain the values of the environment entries.

Each application component defines its own set of environment entries. All instances of an application component within the same container share the same environment entries. Application component instances are not allowed to modify the environment at runtime.

Naming References and Binding Information

A resource reference is an element in a deployment descriptor that identifies the component's coded name for the resource. More specifically, the coded name references a connection factory for the resource. In the example given in the following section, the resource reference name is jdbc/SavingsAccountDB.

The JNDI name of a resource and the name of the resource reference are not the same. This approach to naming requires that you map the two names before deployment, but it also decouples components from resources. Because of this de-coupling, if at a later time the component needs to access a different resource, the name does not need to change. This flexibility also makes it easier for you to assemble J2EE applications from preexisting components.

The following table lists JNDI lookups and their associated references for the J2EE resources used by the Application Server.

Table 6–1 JNDI Lookups and Their Associated References

JNDI Lookup Name	Associated Reference
java:comp/env	Application environment entries
java:comp/env/jdbc	JDBC DataSource resource manager connection factories
java:comp/env/ejb	EJB References
java:comp/UserTransaction	UserTransaction references
java:comp/env/mail	JavaMail Session Connection Factories

JNDI Lookup Name	Associated Reference
java:comp/env/url	URL Connection Factories
java:comp/env/jms	JMS Connection Factories and Destinations
java:comp/ORB	ORB instance shared across application components

JNDI Namespaces

Three JNDI namespaces are used for portable JNDI

lookups: java:global, java:module, and java:app.

The java:global JNDI namespace is the portable way of finding remote enterprise beans using JNDI lookups. JNDI addresses are of the following form:

```
java:global[/application name]/module name/enterprise bean name[/interface name]
```

Application name and module name default to the name of the application and module minus the file extension. Application names are required only if the application is packaged within an EAR. The interface name is required only if the enterprise bean implements more than one business interface.

The java:module namespace is used to look up local enterprise beans within the same module. JNDI addresses using the java:module namespace are of the following form:

```
java:module/enterprise bean name/[interface name]
```

The interface name is required only if the enterprise bean implements more than one business interface.

The java:app namespace is used to look up local enterprise beans packaged within the same application. That is, the enterprise bean is packaged within an EAR file containing multiple Java EE modules. JNDI addresses using the java:app namespace are of the following form:

```
java:app[/module name]/enterprise bean name[/interface name]
```

The module name is optional. The interface name is required only if the enterprise bean implements more than one business interface.

Datasource Resource Definition in Java EE.

DataSource resources are used to define a set of properties required to identify and access a database through the JDBC API. These properties include information such as the URL of the database server, the name of the database, and the network protocol to use to communicate with the server. **DataSource** objects are registered with the Java Naming and Directory Interface (JNDI) naming service so that applications can use the JNDI API to access a **DataSource** object to make a connection with a database.

Prior to Java EE 6, DataSource resources were created administratively as described in "Configuring WebLogic JDBC Resources" in *Configuring and Managing JDBC Data Sources for Oracle WebLogic Server*. Java EE 6 provides the option to programmatically define DataSource resources for a more flexible and portable method of database connectivity.

The **name** element uniquely identifies a **DataSource** and is registered with JNDI. The value specified in the **name** element begins with a namespace scope. Java EE 6 includes the following scopes:

java:comp—Names in this namespace have per-component visibility.

java:module—Names in this namespace are shared by all components in a module, for example, the EJB components defined in an a ejb-jar.xml file.

java:app—Names in this namespace are shared by all components and modules in an application, for example, the application-client, web, and EJB components in an .ear file.

j**ava:global**—Names in this namespace are shared by all the applications in the server.

You can programmatically declare DataSource definitions using one of the following methods:

Creating DataSource Resource Definitions Using Annotations

The javax.annotation.sql package provides **@DataSourceDefinition** and **@DataSourceDefinitions** for defining DataSource resource definitions in application component classes such as application clients, servlets, or Enterprise JavaBeans (EJB).

When the DataSource resource is injected, a **DataSource** object is created and registered with JNDI. Use annotation elements to configure the **DataSource** object. You can specify additional Java EE and WebLogic configuration attributes in the **properties** element of the annotation. See Using WebLogic Configuration Attributes.

Use **@DataSourceDefinition** to create a single datasource definition. For example:

```
. . .


@DataSourceDefinition(

name = "java:module/ExampleDS",

    className = "org.apache.derby.jdbc.ClientDataSource",

    portNumber = 1527,

    serverName = "localhost",

    databaseName = "exampleDB",

    user = "examples",

    password = "examples",

    properties={"create=true", "weblogic.TestTableName=SQL SELECT 1 FROM SYS.SYSTABLES"})


@WebServlet("/dataSourceServlet")

public class DataSourceServlet extends HttpServlet {
```

```
. . .


  @Resource(lookup = "java:module/ExampleDS")


. . .
```

Use the **@DataSourceDefinitions** to create multiple datasource definitions. For example:

```
. . .


@DataSourceDefinitions(
      value = {


        @DataSourceDefinition(name = "java:app/env/DS1",
           minPoolSize = 0,
           initialPoolSize = 0,
           className = "org.apache.derby.jdbc.ClientXADataSource",
           portNumber = 1527,
           serverName = "localhost",
           user = "examples",
           password = "examples",
           databaseName = "exampleDB",
           properties={"create=true", "weblogic.TestTableName=SQL SELECT 1 FROM
SYS.SYSTABLES"}
                  ),


       @DataSourceDefinition(name = "java:comp/env/DS2",
```

```
        minPoolSize = 0,

        initialPoolSize = 0,

        className = "org.apache.derby.jdbc.ClientDataSource",

        portNumber = 1527,

        serverName = "localhost",

        user = "examples",

        password = "examples",

        databaseName = "examplesDB",

        properties={"create=true", "weblogic.TestTableName=SQL SELECT 1 FROM
SYS.SYSTABLES"}

      )

    }

  )



. . .
```

For a complete example, see the "Creating a DataSource using the @DataSourceDefinition Annotation" in the WebLogic Server Code Examples.

Creating DataSource Resource Definition Using Deployment Descriptors

You can create DataSource resource definitions using deployment descriptors in **application.xml**, **application-client.xml**, **web.xml**, and **ejb-jar.xml** files. For example:

```
. . .



 <data-source>

    <name>java:module/ExampleDS</name>

    <class-name>org.apache.derby.jdbc.ClientDataSource</class-name>

    <server-name>localhost</server-name>
```

```
    <port-number>1527</port-number>
    <database-name>exampleDB</database-name>
    <user>examples</user>
    <password>examples</password>
    <property>
      <name>create</name>
      <value>true</value>
    </property>
    <property>
      <name>weblogic.TestTableName</name>
      <value>SQL SELECT 1 FROM SYS.SYSTABLES</value>
    </property>
  </data-source>
. . .
```