Enterprise Java Bean (EJB)

Written in the Java programming language, an enterprise bean is a server-side component that encapsulates the business logic of an application. The **business logic** is the code that fulfills the purpose of the application.

In an inventory control application, for example, the enterprise beans might implement the business logic in methods called `checkInventoryLevel` and `orderProduct`. By invoking these methods, clients can access the inventory services provided by the application.

Benefits of Enterprise Java Beans

- Enterprise beans simplify the development of large, distributed applications.
- The EJB container (Glassfish Server) provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container is responsible for system-level services, such as transaction management and security authorization.
- The enterprise beans are portable components, the application assembler can build new applications from existing beans.

When to Use Enterprise Beans?

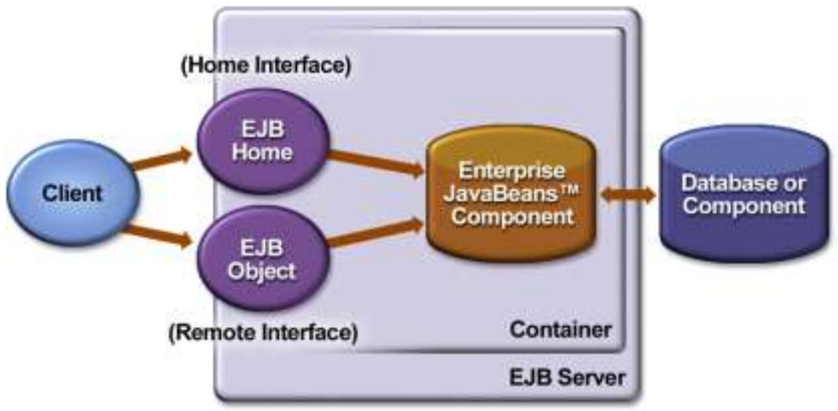
Developer should consider using enterprise beans if an application has any of the following requirements.

- The application must be scalable. To accommodate a growing number of users, you may need to distribute an application's components across multiple machines. Not only can the enterprise beans of an application run on different machines, but also their location will remain transparent to the clients.
- Transactions must ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.
- The application will have a variety of clients. With only a few lines of code, remote clients can easily locate enterprise beans.

Characteristics of EJBs:

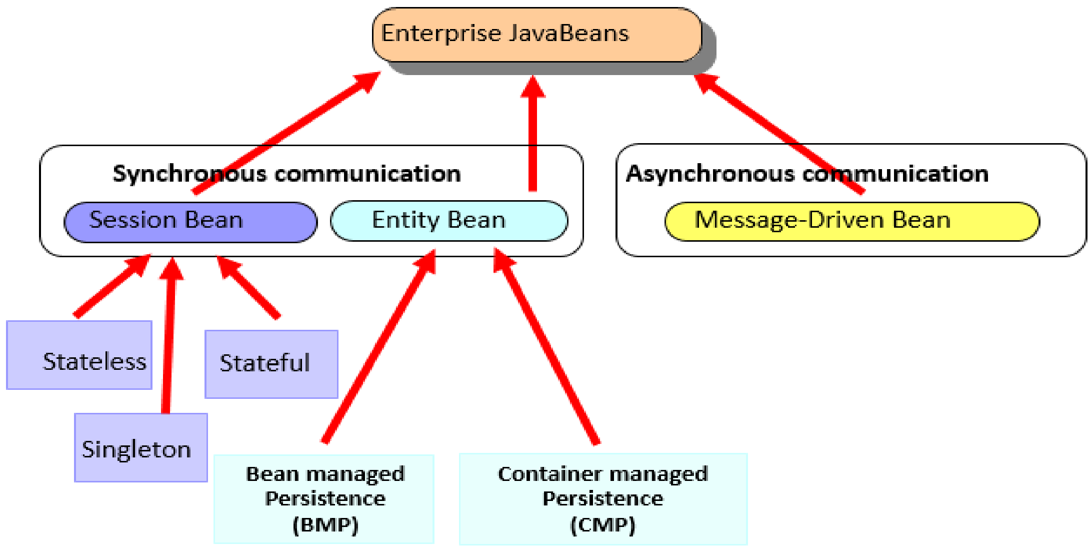
- They contain business logic that operates on the enterprise's data.
- A client never accesses an enterprise bean directly; the container environment mediates access for the client. This provides component-location transparency.
- They can be included in an assembled application without requiring source code changes or recompilation of them.
- Beans are always single threaded and the EJB container handles multiple client requests by load balancing and instantiating multiple instances of the single-threaded components.

Architecture of EJB



- This picture shows a simplified architecture of EJB. The business logic components under EJB architecture are represented as EJB beans while the hosting environment is represented by EJB container (sometimes called as EJB server).
- The business component developer have to write EJB home interface which defines the methods that will be used by clients in order to create and locate your bean through the container. Second, EJB remote interface which defines the business methods of your bean and EJB class.
- The container, at the time of deployment of beans, will create two internal and intermediary objects, EJB home object and EJB remote object. These objects are implementation of home and remote interface. So when the client wants to invoke some business methods of the EJB bean you created, it is actually talking to these two intermediary objects instead. This is to allow the container to intercept the calls so that it can provide system services such as security, transaction, persistence, resource management, life cycle management, and so on.

Types of EJB


Enterprise JavaBeans
Synchronous communication
Session Bean
Entity Bean
Asynchronous communication
Message-Driven Bean
Stateless
Stateful
Singleton
Bean managed Persistence (BMP)
Container managed Persistence (CMP)

EJB has three types of bean - session bean, entity bean, and message driven bean. The session bean can be either stateful , stateless session bean and Singleton. The entity bean can be either bean managed or container managed. As a developer choose which bean type to use depending on the needs and requirements of your application.

Session Bean

A **session bean** encapsulates business logic that can be invoked programmatically by a client over local, remote, or web service client views. To access an application that is deployed on the server, the client invokes the session bean's methods.

Types of Session Beans

Session beans are of three types: 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**.
- As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. When the client terminates, its session bean appears to terminate and is no longer associated with the client. The state is retained for the duration of the client/bean session. If the client removes the bean, the session ends and the state disappears.

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.
- Because they can support multiple clients, stateless session beans 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
- Singleton session beans maintain their state between client invocations but are not required to maintain their state across server crashes or shutdowns.

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.

- A message-driven bean's instances retain no data or conversational state for a specific client.
- A single message-driven bean can process messages from multiple clients.
- Session beans allow you to send JMS messages and to receive them synchronously but not asynchronously.

Accessing Enterprise Beans

Clients access enterprise beans either through a no-interface view or through a business interface. A **no-interface view** of an enterprise bean exposes the public methods of the enterprise bean implementation class to clients. Clients using the no-interface view of an enterprise bean may invoke any public methods in the enterprise bean implementation class or any superclasses of the implementation class. A **business interface** is a standard Java programming language interface that contains the business methods of the enterprise bean.

Using Enterprise Beans in Clients

The client of an enterprise bean obtains a reference to an instance of an enterprise bean through either **dependency injection**, using Java programming language annotations, or **JNDI lookup**, using the Java Naming and Directory Interface syntax to find the enterprise bean instance.

Dependency Injection

It is the simplest way of obtaining an enterprise bean reference. Clients that run within a Java EE server-managed environment, or Java EE application clients, support dependency injection using the `javax.ejb.EJB` annotation.

JNDI lookup

Applications that run outside a Java EE server-managed environment, such as Java SE applications, must perform an explicit lookup. JNDI supports a global syntax for identifying Java EE components to simplify this explicit lookup.

Steps to create a simple web application using Session Bean through Dependency Injection

Step 1: *create a new **bean class** named as EJBbinary.java in com.wipro.bean package.*

```
package com.wipro.bean;
import javax.ejb.Stateless;

@Stateless
public class EJBbinary implements LocalBean {
	public String binaryNumber(int num) {
		String ans = Integer.toBinaryString(num);
		return ans;
	}
}
```

Step 2: *create a Local Interface named as LocalBean.java in com.wipro.bean package.*

```
package com.wipro.bean;
import javax.ejb.Local;

@Local
public interface LocalBean {
	public String binaryNumber(int n);
}
```

Step 3: *create a new **servlet** named as BinaryServlet.java in com.wipro.servelt package.*

```
package com.wipro.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.wipro.bean.LocalBean;

@WebServlet("/BinaryServlet")
public class BinaryServlet extends HttpServlet {

	public void doPost(HttpServletRequest req, HttpServletResponse res)
throws IOException {
		res.setContentType("text/html");
		PrintWriter out = res.getWriter();
```

BinaryServlet is the URL to invoke servlet file. It should match with form action

```
        }
    }
```

Step 4: *In the same BinaryServelt file, create a reference to Local interface using @EJB annotation. The @EJB annotation inject a bean object and assign bean object to interface reference.*

```
package com.wipro.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.wipro.bean.LocalBean;

@WebServlet("/BinaryServlet")
public class BinaryServlet extends HttpServlet {
    @EJB
    private LocalBean ob;

    public void doPost(HttpServletRequest req, HttpServletResponse res)
throws IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();

    }
}
```

Step 5: *In the same BinaryServelt file, get the form data and invoke the bean class method.*

```
package com.wipro.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.wipro.bean.LocalBean;

@WebServlet("/BinaryServlet")
public class BinaryServlet extends HttpServlet {
    @EJB
    private LocalBean ob;
```

```
        public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws IOException {
                res.setContentType("text/html");
                PrintWriter out = res.getWriter();
                int n = Integer.parseInt(req.getParameter("num"));
                String result = ob.binaryNumber(n);
                out.println("Binary format: " + result);
        }
    }
```

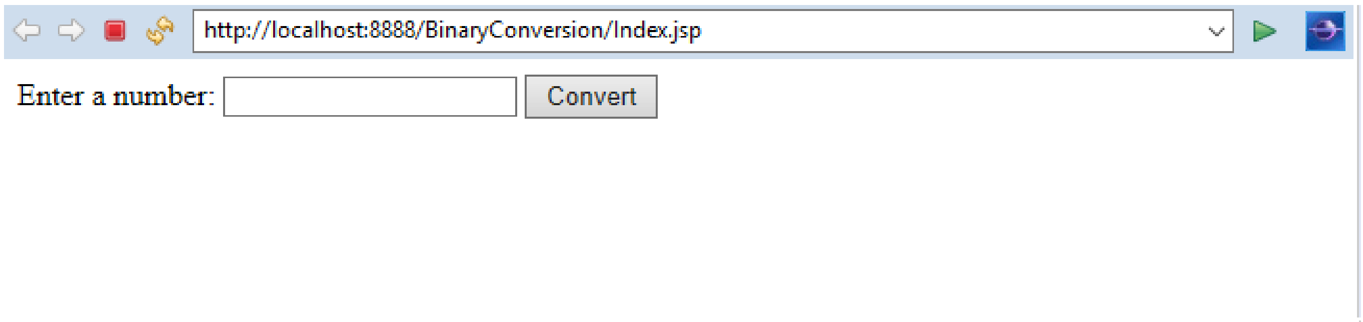
Step 6: *Create a HTML page containing a <**form**> element.*

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Binary Number</title>
</head>
<body>

        <form action="BinaryServlet" method="post">
                Enter a number: <input type="text" name="num" />
                <input type="submit"  value="Convert" />
        </form>
</body>
</html>
```

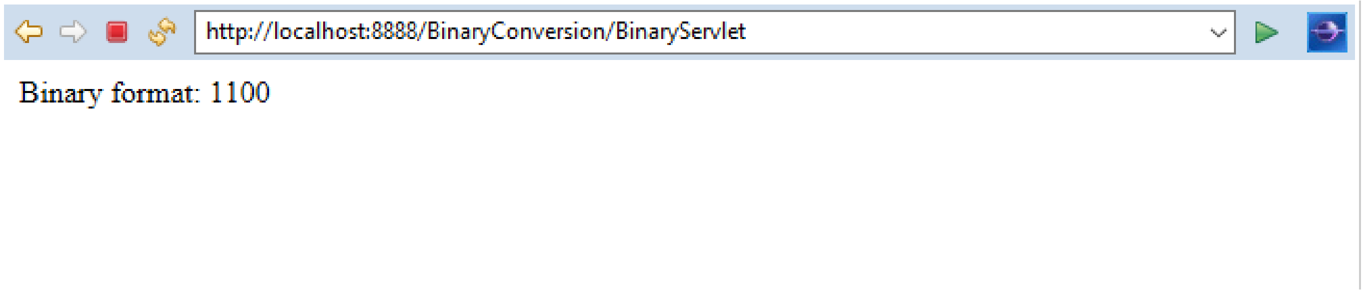
BinaryServlet is the URL to invoke servlet file

Step 7: *Run the HTML file.*



http://localhost:8888/BinaryConversion/Index.jsp

Enter a number: [ ] Convert



http://localhost:8888/BinaryConversion/BinaryServlet

Binary format: 1100

Steps to create a simple web application using Session Bean through **Java Naming and Directory Interface (JNDI)**

Step 1: *Create a JSP page containing a <**form**> element.*

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Input</title>
    </head>
    <body>
        <form action="BeanLookup.jsp">
            Enter the Integer:<input type="text" name="integer"/><br/>
            <input type="submit" value="reverse"/>
        </form>
    </body>
</html>
```

Step 2: *create a new **bean class** named as EJBbean.java in com.wipro.bean package.*

```
package com.wipro.bean;

import javax.ejb.*;
@Stateless
@EJB(name = "java:global/EJBbean", beanInterface = RemoteBean.class)
public class EJBBean implements RemoteBean {
      @Override
      public String reverseInt(String a) {
            String r = "";
            for (int i = a.length() - 1; i >= 0; i--) {
                  r = r + a.charAt(i);
            }
            return r;
      }
}
```

URL to represent Bean class to Remote Users.

URL to represent the interface

Step 3: *create a Remote Interface named as RemoteBean.java in com.wipro.bean package.*

```
package com.wipro.bean;

import javax.ejb.*;
@Remote
public interface RemoteBean {
      public String reverseInt(String a);
}
```

Step 4: *Create a new **JSP** named as BeanLookup.jsp in.*

```
<%@page import="com.wipro.bean.*"%>
<%@page import="javax.naming.*"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Reversed Integer</title>
</head>
<body>
      <%
            String a = request.getParameter("integer");
      %>
      <%
            Context c = new InitialContext();
            RemoteBean e = (RemoteBean) c.lookup("java:global/EJBbean");
      %>
      Reversed Integer:<%=e.reverseInt(a)%>
</body>
</html>
```

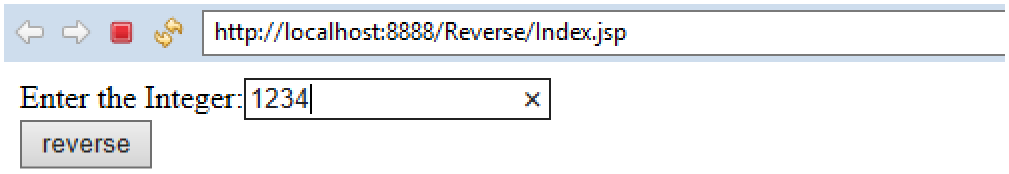
URL to represent Bean class to Remote Users. It should match with the EJB annotation of Bean Class

Step 5: *Create a new **JSP** named as BeanLookup.jsp in.*

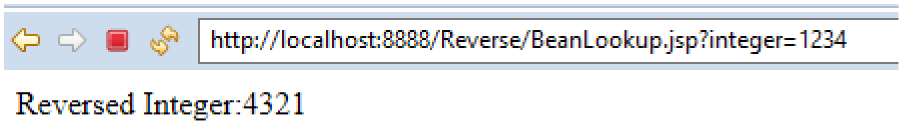
```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Input</title>
    </head>
    <body>
        <form action="BeanLookup.jsp">
            Enter the Integer:<input type="text" name="integer"/><br/>
            <input type="submit" value="reverse"/>
        </form>
    </body>
</html>
```

The file BeanLookup.jsp will be invoked

Step 6: *Run the HTML file.*



Enter the Integer: 1234



Reversed Integer:4321