Module - IV

Enterprise Java Bean (EJB)

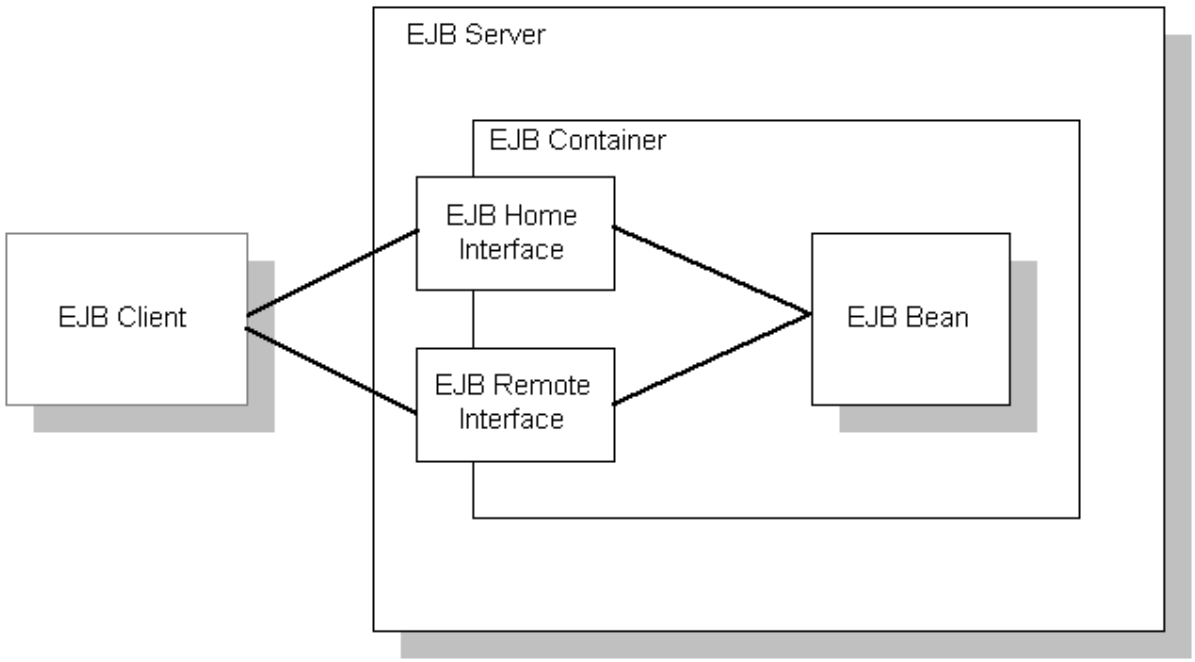
An Enterprise Java Bean is a server side component that encapsulates the business logic of an application.

EJB provides an architecture to develop and deploy component based enterprise applications considering robustness, high scalability, and high performance.

EJB Architecture:

An EJB architecture is composed of:

1. An enterprise bean server
2. Enterprise Bean Containers
3. Enterprise Beans
4. Enterprise Bean Client
5. Other system such as Java Naming and Directory Interface (JNDI) and Java Transaction Service (JTS).



- Client object makes a request for method on the bean.
- Container comes into picture and checks whether the client is in the approved list for calling a method on the bean.
- If the client is authorised, container either crates new interface or activates the requested bean from the pool.

- Container inform the EJB object that the bean is ready and passes the client's method request to bean.

Enterprise Bean Server: AN EJB Server is a component Transaction Server. An EJB server provides the framework for creating, deploying and managing middle-tire business logic and an environment that allows the execution of application deployed using EJB component.

Enterprise Bean Container: An EJB container is basically a software, provides an environment within which EJB component lives. Each time client makes a request for method on bean it performs the following operation-

- Register the object
- Providing a remote interface for the object.
- Creating and destroying object instances
- Checking security for the object.
- Managing active state for the object.
- Coordinating distributed transaction.

Enterprise Bean: Enterprise Bean is a server-side component that encapsulates the code that fulfils the purpose of the application. It sits behind the GUI and performs all the business logic such as database operations.

Enterprise Bean Client: An EJB client is a standalone application that provides the user interface logic on a client machine. The EJB client makes the call to remote EJB components on a server.

Benefits of Enterprise Beans

For several reasons, enterprise beans simplify the development of large, distributed applications.

First, because the EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container, rather than the bean developer, is responsible for system-level services, such as transaction management and security authorization.

Second, because the beans rather than the clients contain the application's business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

Third, because enterprise beans are portable components, the application assembler can build new applications from existing beans. Provided that they use the standard APIs, these applications can run on any compliant Java EE server.

When to Use Enterprise Beans

You should consider using enterprise beans if your 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. These clients can be thin, various, and numerous.

Types of EJB:

EJB Architecture defines three types of enterprise beans:

1. Session Bean
2. Entity Bean
3. Message Driven Bean

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. The session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.

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

Stateful Session Beans:

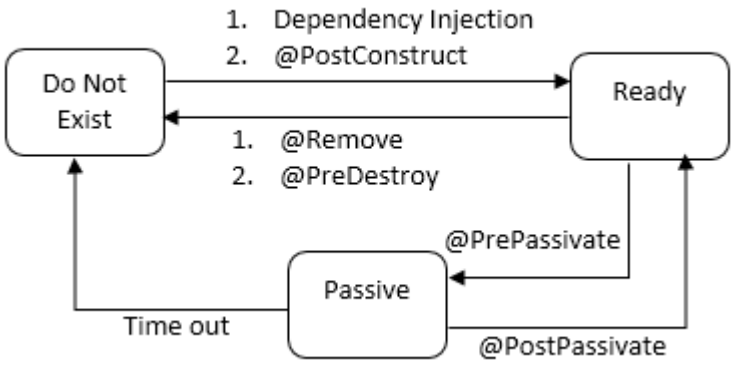
The state of an object consists of the values of its instance variables. In a stateful session bean, the instance variables represent the state of a unique client/bean session. Because the client interacts ("talks") with its bean, this state is often called the conversational state.

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.

Lifecycle of a Stateful Session Bean:

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.



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.

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.

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 across all clients.

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.

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.

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

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.

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.

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.

JMS (Java Message Service) is an API that provides the facility to create, send and read messages for application or software component. JMS is mainly used to send and receive message from one application to another.

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.

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

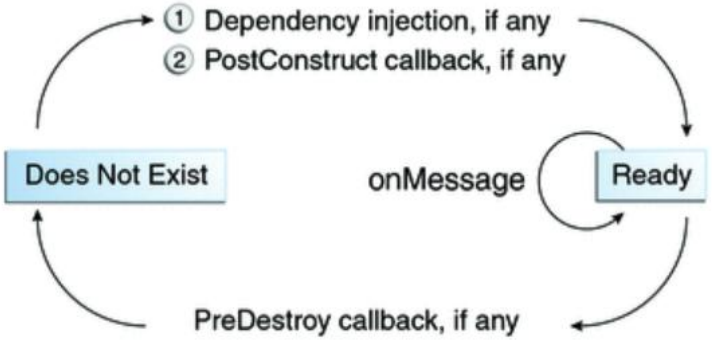
The most visible difference between message-driven beans and session beans is that

- Clients do not access message-driven beans through interfaces.
- Session beans allow you to send JMS messages and to receive them synchronously but not asynchronously.

Lifecycle of a Message-Driven Bean:

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`.

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.

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.

Entity Bean:

Entity bean represents data maintained by an enterprise, typically in a database. An instance of the entity bean represents a row in a table.

Entity Bean gives more control to EJB developer to save and retrieve data from database.

In EJB 2.x, there was two types of entity beans: **bean managed persistence** (BMP) and **container managed persistence** (CMP). Since EJB 3.x, it is deprecated and replaced by JPA (Java Persistence API).

In BMP more control is given to the EJB developer to save and retrieve data. But in CMP container is responsible for saving and retrieving the data from the underlying database.

Context and Dependency Injection:

Contexts and Dependency Injection (CDI) for the Java EE platform is one of several Java EE 6 features that help to knit together the web tier and the transactional tier of the Java EE platform. CDI is a set of services that, used together, make it easy for developers to use enterprise beans along with Java Server Faces technology in web applications. Designed for use with stateful objects, CDI also has many broader uses, allowing developers a great deal of flexibility to integrate various kinds of components in a loosely coupled but type safe way.

Contexts and Dependency Injection for Java EE (CDI) 1.0 was introduced as part of the Java EE 6 platform, and has quickly become one of the most important and popular components of the platform.

The most fundamental services provided by CDI are as follows:

Contexts: The ability to bind the lifecycle and interactions of stateful components to well-defined but extensible lifecycle contexts

Dependency injection: The ability to inject components into an application in a type safe way, including the ability to choose at deployment time which implementation of a particular interface to inject

CDI defines a powerful set of complementary services that help improve the structure of application code.

- A well-defined lifecycle for stateful objects bound to lifecycle contexts, where the set of contexts is extensible
- A sophisticated, type safe dependency injection mechanism, including the ability to select dependencies at either development or deployment time, without verbose configuration
- Integration with the Unified Expression Language (EL), allowing any contextual object to be used directly within a JSF or JSP page

Ex (Dependency Injection in Java):

Services.java

```
public class Services{
	int add(int a,int b){
		return a+b;
	}
	int sub(int a,int b){
		return a-b;
	}
}
```

Client.java

```
public class Client{
	private Services s;
	int a=5,b=2;
	Client(Services s){
		this.s=s;
	}
	void operation(){
		System.out.println("Sum="+s.add(a,b));
		System.out.println("Difference="+s.sub(a,b));
	}
}
```

Injector.java

```
public class Injector{
	public static void main(String a[]){
		Services s=new Services();
		Client c=new Client(s);
		c.operation();
	}
}
```

Security in Java EE:

Security is a major concern of any enterprise level application. It includes identification of user(s) or system accessing the application. Based on identification, it allows or denies the access to resources within the application. An EJB container manages standard security concerns or it can be customized to handle any specific security concerns.

Features of a Security Mechanism

A properly implemented security mechanism will provide the following functionality:

- Prevent unauthorized access to application functions and business or personal data (authentication)
- Hold system users accountable for operations they perform (non-repudiation)
- Protect a system from service interruptions and other breaches that affect quality of service

Ideally, properly implemented security mechanisms will also be

- Easy to administer
- Transparent to system users
- Interoperable across application and enterprise boundaries

Characteristics of Application Security:

- **Authentication** − This is the process ensuring that user accessing the system or application is verified to be authentic.
- **Authorization** − This is the process ensuring that authentic user has right level of authority to access system resources.
- **Data integrity –** The means used to prove that information has not been modified by third party, an entity other than the source of information.

- **Confidentiality -** It ensure that information is made available only to the user who are authorised to access.
- **Non-repudiation:** It is used to prove that a user who performed some action cannot reasonably deny having done so.
- **Quality of Service**: The means used to provide better service to selected network traffic over various technologies.
- **Auditing**: The means used to capture a tamper-resistant record of security-related events for the purpose of being able to evaluate the effectiveness of security policies and mechanisms. To enable this, the system maintains a record of transactions and security information.

Java EE Security Mechanisms

1. Application-Layer Security

In Java EE, component containers are responsible for providing application-layer security, security services for a specific application type tailored to the needs of the application. At the application layer, application firewalls can be used to enhance application protection by protecting the communication stream and all associated application resources from attacks.

2. Transport-Layer Security

Transport-layer security is provided by the transport mechanisms used to transmit information over the wire between clients and providers; thus, transport-layer security relies on secure HTTP transport (HTTPS) using Secure Sockets Layer (SSL). Transport security is a point-to-point security mechanism that can be used for authentication, message integrity, and confidentiality. When running over an SSL-protected session, the server and client can authenticate each other and negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. Security is active from the time the data leaves the client until it arrives at its destination, or vice versa, even across intermediaries. The problem is that the data is not protected once it gets to the destination. One solution is to encrypt the message before sending.

3. Message-Layer Security

In message-layer security, security information is contained within the SOAP message and/or SOAP message attachment, which allows security information to travel along with the message or attachment. For example, a portion of the message may be signed by a sender and encrypted for a particular receiver. When sent from the initial sender,

the message may pass through intermediate nodes before reaching its intended receiver. In this scenario, the encrypted portions continue to be opaque to any intermediate nodes and can be decrypted only by the intended receiver. For this reason, message-layer security is also sometimes referred to as end-to-end security.

Security Features in EJB:

In Java EE, the component containers are responsible for providing application security. A container provides two types of security: declarative and programmatic.

Annotations to Specify Security Information:

EJB 3.0 has specified following attributes/annotations of security, which EJB containers implement.

DeclareRoles − Indicates that class will accept the declared roles. Annotations are applied at class level.

RolesAllowed − Indicates that a method can be accessed by user of role specified. Can be applied at class level resulting which all methods of class can be accessed buy user of role specified.

PermitAll − Indicates that a business method is accessible to all. It can be applied at class as well as at method level.

DenyAll − Indicates that a business method is not accessible to any of the user specified at class or at method level.

1. Declarative Security

Declarative security can express an application component's security requirements by using deployment descriptors. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the Java EE server reads the deployment descriptor and acts upon the corresponding application, module, or component accordingly. Deployment descriptors must provide certain structural information for each component if this information has not been provided in annotations or is not to be defaulted.

2. Programmatic Security

Programmatic security is embedded in an application and is used to make security decisions. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application. The API for programmatic security consists of methods of the `EJBContext` interface and the `HttpServletRequest` interface. These methods allow components to make business-logic decisions based on the security role of the caller or remote user.

Java Persistence API:

The Java Persistence API provides Java developers with an object/relational mapping facility for managing relational data in Java applications. Java Persistence consists of four areas:

- The Java Persistence API
- The query language
- The Java Persistence Criteria API
- Object/relational mapping metadata

The Java Persistence API:

The Java Persistence API (JPA) is a specification of Java. It is used to persist data between Java object and relational database. JPA acts as a bridge between object-oriented domain models and relational database systems.

Persistence domain objects are known as Entity and represented by @Entity annotation. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table.

Entities are managed by the `EntityManager`. Each `EntityManager` instance is associated with a persistence context: a set of managed entity instances that exist in a particular data store. A persistence context defines the scope under which particular entity instances are created, persisted, and removed. The `EntityManager` interface defines the methods that are used to interact with the persistence context.

The query language:

The Java Persistence Query Language (JPQL) is a platform-independent object-oriented query language defined as part of the Java Persistence API (JPA) specification.

JPQL is used to make queries against entities stored in a relational database. It is heavily inspired by SQL, and its queries resemble SQL queries in syntax, but operate against JPA entity objects rather than directly with database tables.

The Java Persistence Criteria API:

The Criteria API is a predefined API used to define queries for entities. It is the alternative way of defining a JPQL query. These queries are type-safe, and portable and

easy to modify by changing the syntax. Similar to JPQL it follows abstract schema (easy to edit schema) and embedded objects. The metadata API is mingled with criteria API to model persistent entity for criteria queries.

The major advantage of the criteria API is that errors can be detected earlier during compile time. String based JPQL queries and JPA criteria based queries are same in performance and efficiency.

Object Relation Mapping:

Object Relational Mapping (ORM) is a functionality which is used to develop and maintain a relationship between an object and relational database by mapping an object state to database column. It is capable to handle various database operations easily such as inserting, updating, deleting etc.

Various frameworks that function on ORM mechanism: - Hibernate, TopLink, ORMLite, iBATIS, JPOX

Types of Mapping:

Following are the various ORM mappings: -

- **One-to-one -** This association is represented by @OneToOne annotation. Here, instance of each entity is related to a single instance of another entity.
- **One-to-many -** This association is represented by @OneToMany annotation. In this relationship, an instance of one entity can be related to more than one instance of another entity.
- **Many-to-one -** This mapping is defined by @ManyToOne annotation. In this relationship, multiple instances of an entity can be related to single instance of another entity.
- **Many-to-many -** This association is represented by @ManyToMany annotation. Here, multiple instances of an entity can be related to multiple instances of another entity. In this mapping, any side can be the owing side.

Java EE Supporting Technologies

The Java EE platform includes several technologies and APIs that extend its functionality. These technologies allow applications to access a wide range of services in a uniform manner. These technologies are, Transactions, Resources and Resource Adapters, Java Message Service Concepts, Java Message, Bean Validation.

Transactions in Java EE Applications

In a Java EE application, a transaction is a series of actions that must all complete successfully, or else all the changes in each action are backed out. Transactions end in either a commit or a rollback.

The Java Transaction API (JTA) allows applications to access transactions in a manner that is independent of specific implementations. JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the transactional application, the Java EE server, and the manager that controls access to the shared resources affected by the transactions.

A financial program, for example, might transfer funds from a checking account to a savings account by using the steps listed in the following pseudo code:

```
begin transaction
   debit checking account
   credit savings account
   update history log
commit transaction
```

Either all or none of the three steps must complete. Otherwise, data integrity is lost. Because the steps within a transaction are a unified whole, a transaction is often defined as an indivisible unit of work.

A transaction can end in two ways: with a commit or with a rollback. When a transaction commits, the data modifications made by its statements are saved. If a statement within a transaction fails, the transaction rolls back, undoing the effects of all statements in the transaction. In the pseudo code, for example, if a disk drive were to crash during the credit step, the transaction would roll back and undo the data modifications made by the debit statement. Although the transaction fails, data integrity would be intact because the accounts still balance.

Container-Managed Transactions: In an enterprise bean with container-managed transaction demarcation, the EJB container sets the boundaries of the transactions. Container-managed transactions can be used with any type of enterprise bean: session or message-driven. Container-managed transactions simplify development because the enterprise bean code does not explicitly mark the transaction's boundaries. The code does not include statements that begin and end the transaction. By default, if no transaction demarcation is specified, enterprise beans use container-managed transaction demarcation.

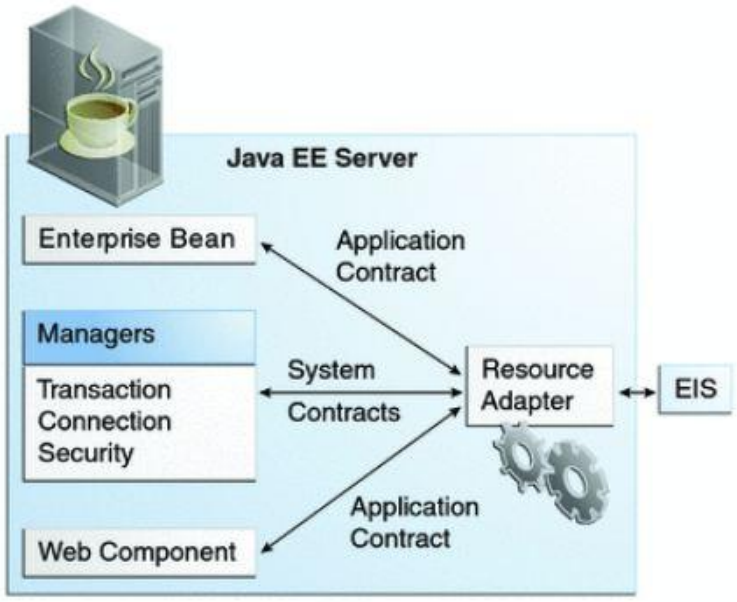
Bean-Managed Transactions: In bean-managed transaction demarcation, the code in the session or message-driven bean explicitly marks the boundaries of the transaction. Although beans with container-managed transactions require less coding, they have one limitation: When a method is executing, it can be associated with either a single

transaction or no transaction at all. Using bean-managed transactions this drawbacks can be avoided.

Resources and Resource Adapters:

Java EE components can access a wide variety of resources, including databases, mail sessions, Java Message Service objects, and URLs. The Java EE 6 platform provides mechanisms that allow you to access all these resources in a similar manner.

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 Java EE platform, the Java Naming and Directory Interface (JNDI) naming service enables components to locate other components and resources.



A resource is a program object that provides connections to systems, such as database servers and messaging systems. (A Java Database Connectivity resource is sometimes referred to as a data source.) Each resource object is identified by a unique, people-friendly name, called the JNDI name.

To store, organize, and retrieve data, most applications use a relational database. Java EE 6 components may access relational databases through the JDBC API.In the JDBC API, databases are accessed by using `DataSource` objects. A `DataSource` has a set of properties that identify and describe the real-world data source that it represents. These properties include such information as the location of the database server, the name of the database, the network protocol to use to communicate with the server, and

so on. In the GlassFish Server, a data source is called a JDBC resource.Applications access a data source by using a connection, and a `DataSource` object can be thought of as a factory for connections to the particular data source that the `DataSource` instance represents. In a basic `DataSource` implementation, a call to the `getConnection` method returns a connection object that is a physical connection to the data source.A `DataSource` object may be registered with a JNDI naming service. If so, an application can use the JNDI API to access that `DataSource` object, which can then be used to connect to the data source it represents.

A resource adapter is a Java EE component that implements the Java EE Connector architecture for a specific EIS (Executive information system). Examples of EISs include enterprise resource planning, mainframe transaction processing, and database systems. In a Java EE server, the Java Message Server and JavaMail also act as EISs that you access using resource adapters. As illustrated in Figure, the resource adapter facilitates communication between a Java EE application and an EIS.

References:

1. Java Server Programming for Professionals: Ivan Baross, Sharanam Shah, Cynthis Bayross, Vaishali Shah

2. Professional Java Server Programming: with Servlets, JavaServer Pages (JSP), XML, Enterprise JavaBeans (EJB), JNDI, CORBA, Jini and Javaspaces Paperback –Danny Ayers (Author), Hans Bergsten (Author), Jason Diamond (Author), Mike Bogovich (Author), Matthew Ferris (Author), Marc Fleury (Author), Ari Halberstadt (Author), Paul Houle (Author), Sing Li (Author), Piroz Mohseni(Author), Ron Phillips (Author), Krishna Vedati (Author), Mark Wilcox (Author), Stefan Zeiger (Author), Andrew Patzer(Author)
3. Eric Jendrock, D. Carson, I. Evans, D. Gollapudi, K. Haase, C. Srivastha, " The Java EE6 Tutorial",Volume-1, Fourth Edition, 2010, Pearson India, New Delhi.
4. https://www.javatpoint.com/
5. https://www.geeksforgeeks.org/
6. https://www.tutorialspoint.com
7. https://javabeat.net/
8. https://docs.oracle.com/
9. https://en.wikipedia.org/
10. www.google.com