Module 5 EJB and Server Side Component Models

Problem Domain

- Characteristics that good software implies
  - Secure
  - Sound/maintains integrity
  - Scalable
  - Interoperable
  - Robust/resilient
  - Correct/functions as specified

5.1 Container Services

- The EJB Container can continue to make our jobs easier by providing a wide range of generic and configurable services.
- Enterprise applications generally handle a large number of concurrent users, and each request might need to be handled carefully so it does not violate the security or integrity of the application.
- wiring of modules together is taken care by container itself.
- EJB offers a range of container services
  - Dependency Injection
  - Concurrency
  - Instance pooling/caching
  - Transactions
  - Security
  - Timers
  - Naming and object stores
  - Interoperability
  - Lifecycle callbacks
  - Interceptors
  - Platform Integration
  - Bringing it Together

➢ Dependency Injection (DI)

- A component-based approach to software design brings with it the complication of inter-module communication. Tightly coupling discrete units together violates module independence and separation of concerns
- As EJB is a component-centric architecture, it provides a means to reference dependent modules in decoupled fashion.

- **Concurrency**
  - Assuming each Service is represented by one instance, dependency injection alone is a fine solution for a single-threaded application; only one client may be accessing a resource at a given time.
  - However, this quickly becomes a problem in situations where a centralized server is fit to serve many simultaneous requests.

- **Instance Pooling/ Caching**
  - Because of the strict concurrency rules enforced by the Container, an intention a bottleneck is often introduced where a service instance may not be available for processing until some other request has completed.
  - If the service was restricted to a singular instance, all subsequent requests would have to queue up until their turn was reached needed to achieve optimal throughput.

- **Transactions**
  - A transaction is a single unit of work items, which follows the ACID properties. ACID stands for Atomic, Consistent, Isolated, and Durable.
    - **Atomic** − If any of the work item fails, the whole unit will be considered failed. Success meant, all items execute successfully.
    - **Consistent** − A transaction must keep the system in consistent state.
    - **Isolated** − Each transaction executes independent of any other transaction.
    - **Durable** − Transaction should survive system failure if it has been executed or committed.

- **Security**
  - 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.

- **Timers**
  - Timer Service is a mechanism by which scheduled application can be build. For example, salary slip generation on the 1st of every month. EJB 3.0 specification has specified @Timeout annotation, which helps in programming the EJB service in a stateless or message driven bean. EJB Container calls the method, which is annotated by @Timeout.
  - EJB Timer Service is a service provided by EJB container, which helps to create timer and to schedule callback when timer expires.

- **Naming and Object Stores**

  - JNDI stands for Java Naming and Directory Interface. It is a set of API and service interfaces. Java based applications use JNDI for naming and directory services. In context of EJB, there are two terms.

    - **Binding** − This refers to assigning a name to an EJB object, which can be used later.
    - **Lookup** − This refers to looking up and getting an object of EJB.

- **Interoperability**
  - Interoperability is a vital part of EJB. The specification includes the required support for Java RMI-IIOP for remote method invocation and provides for transaction, naming, and security interoperability.
  - EJB also requires support for JAX-WS, JAX-RPC, Web Services for Java EE, and Web Services Metadata for the Java Platform specifications.

- **Lifecycle callbacks**
  - Callback is a mechanism by which the life cycle of an enterprise bean can be intercepted. EJB 3.0 specification has specified callbacks for which callback handler methods are created. EJB Container calls these callbacks. We can define callback methods in the EJB class itself or in a separate class. EJB 3.0 has provided many annotations for callbacks.

- **Interceptors**
  - While it's really nice that EJB provides aspectized handling of many of the container services, the specification cannot possibly identify all cross-cutting concerns facing your project.
  - For this reason, EJB makes it possible to define custom *interceptors* upon business methods and lifecycle callbacks. This makes it easy to contain some common code in a centralized location and have it applied to the invocation chain without impacting your core logic.

- **Platform Integration**
  - As a key technology within the Java Enterprise Edition (JEE) 6, EJB aggregates many of the other platform frameworks and APIs:
    - Java Transaction Service
    - Java Persistence API
    - Java Naming and Directory Interface (JNDI)

- Security Services
- Web Services

5.3 Developing your first EJB

- Developing an EJB involves most of the same steps and concepts that you have become familiar with when developing plain old Java objects (POJOs), with a few minor differences. The following steps illustrate the typical process to develop and deploy an EJB:
  1. Write the classes and interfaces for your enterprise bean
  2. Write a deployment descriptor
  3. Package the enterprise bean and associated files inside of a jar file
  4. Deploy the bean

5.4 Models

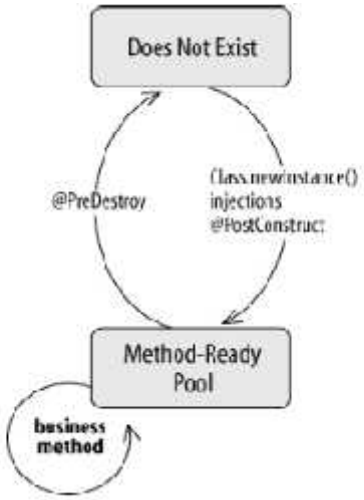
Session beans are divided into three basic types: stateless, stateful, and singleton.

5.4.1 Stateless session bean

- A stateless session bean is a type of enterprise bean, which is normally used to perform independent operations.
- A stateless session bean as per its name does not have any associated client state.
- EJB Container normally creates a pool of few stateless bean's objects and use these objects to process client's request. Because of pool, instance variable values are not guaranteed to be same across lookups/method calls.

The Lifecycle of a stateless session bean

- The lifecycle of a stateless session bean is very simple. It has only two states: *Does Not Exist* and *Method-Ready Pool*.



The Does Not Exist State

- When a bean is in the Does Not Exist state, it is not an instance in the memory of the system. In other words, it has not been instantiated yet.

The Method-Ready Pool

- Stateless bean instances enter the Method-Ready Pool as the container needs them. When the EJB server is first started, it may create a number of stateless bean instances and enter them into the Method-Ready Pool. (The actual behavior of the server depends on the implementation.) When the number of stateless instances servicing client requests is insufficient, more can be created and added to the pool.

5.4.2 Stateful session bean

- A stateful session bean is a type of enterprise bean, which preserve the conversational state with client. A stateful session bean as per its name keeps associated client state in its instance variables. EJB Container creates a separate stateful session bean to process client's each request. As soon as request scope is over, stateful session bean is destroyed.

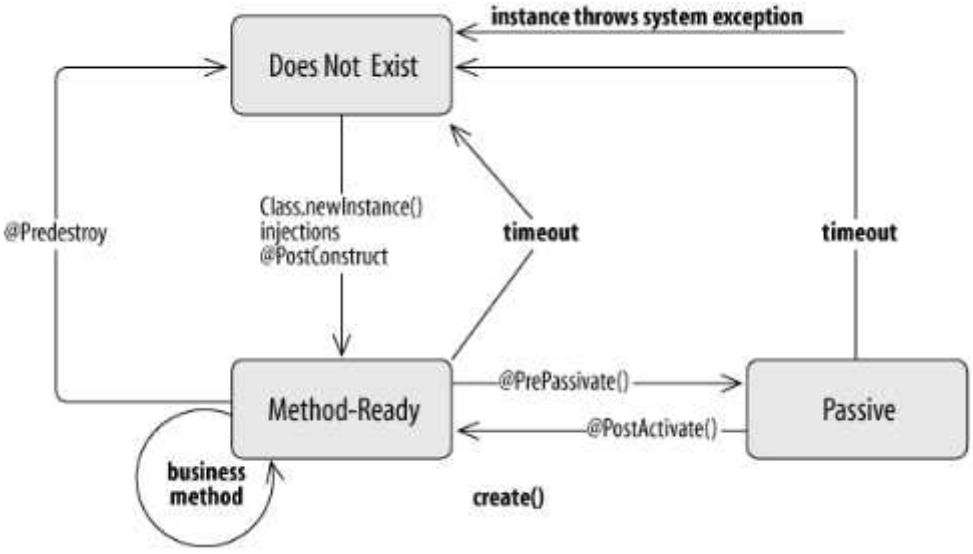
Lifecycle of Stateful Session Bean

The Does Not Exist State

- A stateful bean instance in the Does Not Exist state has not been instantiated yet. It doesn't exist in the system's memory.

The Method-Ready State

- The Method-Ready state is the state in which the bean instance can service requests from its clients. This section explores the instance's transition into and out of the Method-Ready state.



The Passivated State

- During the lifetime of a stateful session bean, there may be periods of inactivity when the bean instance is not servicing methods from the client. To conserve resources, the container can passivate the bean instance by preserving its conversational state and evicting the bean instance from memory. A bean's conversational state may consist of primitive values, objects that are serializable, and the following special types:
  - javax.ejb.SessionContext
  - javax.jta.UserTransaction (bean transaction interface)
  - javax.naming.Context (only when it references the JNDI ENC)
  - javax.persistence.EntityManager
  - javax.persistence.EntityManagerFactory
  - References to managed resource factories (e.g., javax.sql.DataSource)
  - References to other EJBs

The types in this list (and their subtypes) are handled specially by the passivation mechanism. They do not need to be serializable; they will be maintained through passivation and restored automatically when the bean instance is activated.

5.4.3 Singleton session bean

- These bean types service requests independently of one another in separate bean Sometimes, however, it's useful to employ a scheme in which a single shared instance is used for all clients, and new to the EJB 3.1 Specification is the singleton session bean to fit this requirement.

Lifecycle of Singleton Session Bean

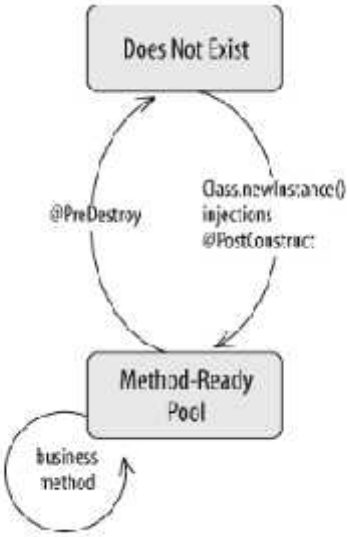
- The life of a singleton bean is very similar to that of the stateless session bean; it is either not yet instantiated or ready to service requests. In general, it is up to the Container to determine when to create the underlying bean instance, though this must be available before the first invocation is executed. Once made, the singleton bean instance lives in memory for the life of the application and is shared among all requests.

5.5 Message Driven Bean

- A message driven bean (MDB) is a bean that contains business logic. But, it is invoked by passing the message. So, it is like JMS Receiver.
- JMS provides two types of messaging models.
  - publish-and -subscribe(*topic*): messaging, one producer can send a message to many consumers through a virtual channel called a *topic*.
  - point -to-point (*queue*): on the other hand, is intended for a message that is to be processed once

LifeCycle of a Message Driven Bean

- The MDB instance's lifecycle has two states: *Does Not Exist* and *Method-Ready Pool*.
- The Method-Ready Pool is similar to the instance pool used for stateless session beans.



The Does Not Exist State

When an MDB instance is in the Does Not Exist state, it is not an instance in the memory of the system. In other words, it has not been instantiated yet.

The Method-Ready Pool

MDB instances enter the Method-Ready Pool as the container needs them. When the EJB server is first started, it may create a number of MDB instances and enter them into the Method-Ready Pool

5.6 EJB and Persistence: Persistence Entity Manager

- Entity bean represents the persistent data stored in the database. It is a server-side component.
- Following are the key actors in persistence API –
    - **Entity** − A persistent object representing the data-store record. It is good to be serializable.
    - **EntityManager** − Persistence interface to do data operations like add/delete/update/find on persistent object(entity). It also helps to execute queries using **Query** interface.
    - **Persistence unit (persistence.xml)** − Persistence unit describes the properties of persistence mechanism.
    - **Data Source (*ds.xml)** − Data Source describes the data-store related properties like connection url. user-name,password etc.

Mapping Persistence

The Programming model

- Entities are plain Java classes in Java Persistence.
- declare and allocate these bean classes just as you would any other plain Java object.

- interact with the entity manager service to persist, update, remove, locate, and query for entity beans.
- The entity manager service is responsible for automatically managing the entity beans' state.
- This service takes care of enrolling the entity bean in transactions and persisting its state to the database.

The Employee Entity

The Employee class is a simple entity bean that models the concept of an employee within a company. Java Persistence is all about relational databases.

The Bean Class

The Employee bean class is a plain Java object that you map to your relational database. It has fields that hold state and, optionally, it has *getter* and *setter* methods to access this state.

XML Mapping File

If you do not want to use annotations to identify and map your entity beans, you can alternatively use an XML mapping file to declare this metadata.

Entity Relationship

In order to model real-world business concepts, entity beans must be capable of forming relationships.

The Seven Relationship Types

- Seven types of relationships can exist between entity beans.
- There are four types of cardinality: *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many*.
  - ***One-to-one unidirectional***
    - The relationship between an employee and an address. You clearly want to be able to look up an employee's address, but you probably don't care about looking up an address's employee.
  - ***One-to-one bidirectional***
    - The relationship between an employee and a computer. Given an employee, we'll need to be able to look up the computer ID for tracing purposes. Assuming the computer is in the tech department for servicing, it's also helpful to locate the employee when all work is completed.
  - ***One-to-many unidirectional***
    - The relationship between an employee and a phone number. An employee can have many phone numbers (business, home, cell, etc.). You might need

to look up an employee's phone number, but you probably wouldn't use one of those numbers to look up the employee.

- ***Many-to-one unidirectional***
  - The relationship between an employee (manager) and direct reports. Given a manager, we'd like to know who's working under him or her. Similarly, we'd like to be able to find the manager for a given employee. (Note that a many-to-one bidirectional relationship is just another perspective on the same concept.)
- ***one-to-many bidirectional***
  - The relationship between a customer and his or her primary employee contact. Given a customer, we'd like to know who's in charge of handling the account. It might be less useful to look up all the accounts a specific employee is fronting, although if you want this capability you can implement a many-to-one bidirectional relationship.
- ***Many-to-many bidirectional***
  - The relationship between employees and tasks to be completed. Each task may be assigned to a number of employees, and employees may be responsible for many tasks. For now we'll assume that given a task we need to find its related employees, but not the other way around. (If you think you need to do so, implement it as a bidirectional relationship.)
- ***Many-to many unidirectional***
  - The relationship between an employee and the teams to which he or she belongs. Teams may also have many employees, and we'd like to do lookups in both directions.