Introduction to
Java Persistence API (JPA)
And
Enterprise Java Beans (EJBs)

Fall 2012
Department of Mathematics
and Computer Science
Lehman College, CUNY

JPA and EJB Introduction
Why EJB and JPA? (1)

- Most server-side applications have a lot in common
  - including business logic implemented
  - managing application state
  - storing and retrieving information/data from/to a relational database
  - managing transactions
  - implementing security
  - performing asynchronous processing
  - integrating systems, etc.

Why EJB and JPA? (2)

- Web application built based on JSP(EL/JSTL) and Sevlet technologies with JNDI/JDBC DataSource can be run in any web server such as Glassfish Server and Tomcat Server.
- You can continue develop web application using these technologies for small and medium sized applications.
  - Your application may need to have additional functionality such as thread-safety, transactions, and security (although such functionality may be provided by the backend DBMS).
Why EJB and JPA? (3)

- Data should be managed transparently
- The user interface should not perform business logic
  - Especially when there are multiple interfaces (web, mobile, desktop)
- The business layer
  - Implement business logic
  - Should be separated from the data managing layer
  - Should be separated from presentation layer
  - Add transaction management and security

Why EJB and JPA? (4)

- EJB technology is a server-side component architecture that enables rapid and simplified development of distributed, transactional, secure, and portable applications
  - The services provided by the technology remain transparent to the developer
  - Eliminating the tedious and often error-prone task of adding a lot of boiler plate code which would otherwise be required
MVC Architecture

- **MVC (Model View Controller) Architecture**
  - the model, business logic, and presentation layers
- **The model is encapsulated by entities**
  - Keep the entities as transparent as possible, free of business logic
    - They can have methods to validate their attributes
- **The business logic is encapsulated by EJBs...**
  - Model the actions of the application (reserve a seat, buy a ticket, etc)
- **The presentation layer**
  - Can be built with different technologies, depending on the target audience (web users? mobile devices? desktop apps?)
  - By implementing the business logic in EJBs, it can be used by other apps

EJB Containers

- **EJB container**
  - runtime environment that provides services, such as transaction management, concurrency control, pooling, and security authorization.
  - Historically, application servers have added other features such as clustering, load balancing, and failover.
- **Using Enterprise beans with JPA entity classes allows you focus on the business logic of your application while relying on solutions that have already been tried and tested.**
- **Some JEE Application Servers**
  - GlassFish (Sun/Oracle, open source edition)
  - WebSphere (IBM)
  - WebLogic (Oracle)
  - JBoss (RedHat)
  - WebObjects (Apple)
You can think of EJB both as components, or Java classes that are incorporated in your project, as well as a framework that provides numerous enterprise-related services.

- **Pooling**: For each EJB component, the EJB platform creates a pool of component instances that are shared by clients.
  - At any point in time, each pooled instance is only allowed to be used by a single client.
  - As soon as an instance is finished servicing a client, it is returned to the pool for reuse instead of being discarded for the garbage collector to reclaim.

- **Thread Safety**: EJB makes all components thread-safe in ways that are completely invisible.
  - This means that you can write your server components as if you were developing a single-threaded desktop application. It doesn't matter how complex the component itself is.

---

**EJB Services (2)**

- **Transactions**: EJB supports declarative transaction management that helps you add transactional behavior to components using simple configuration instead of coding.
  - You can designate any component method to be transactional.
  - If the method completes normally, EJB commits the transaction and makes the data changes made by the method permanent. Otherwise the transaction is rolled back.

- **Security**: EJB supports integration with the Java Authentication and Authorization Service (JAAS) API
  - It is easy to completely externalize security and secure an application using simple configuration instead of writing security code.
In the context of Java Enterprise, persistence refers to the act of automatically storing data contained in Java objects into a relational database.

- The Java Persistence API (JPA) is an object-relational mapping (ORM) technology that enables applications to manage data between Java objects and a relational database in a way that is transparent to the developer.
- This means that you can apply JPA to your projects by creating and configuring a set of Java classes (entities) that mirror your data model. Your application can then access these entities as though it were directly accessing the database.

**ORM Tools**
- Programmers can create their own using JDBC
- Hibernate, iBATIS, JDO
- ADO.NET Entity Framework (Microsoft)
- EclipseLink, etc...
Benefits to using JPA

- POJO persistence
- ORM is completely metadata-driven
- No complex data access objects (DAO).
- The API helps you manage transactions.
- Write standards-based code that interacts with any relational database, freeing you from vendor-specific code.
- JPA has its own rich, SQL-like query language for static and dynamic queries. Using the Java Persistence Query Language (JPQL), your applications remain portable across different database vendors.
- You can avoid the task of writing low-level, verbose JDBC/SQL code.
- JPA transparently provides services for data caching and performance optimization.
- You can also use the Java Persistence API for desktop application persistence -- not just for enterprise apps.

JPA Concepts (1)

- JPA maps objects to a database through metadata.
- Associated with every entity is metadata that describes the mapping.
- Metadata can be written in two different formats:
  - Annotations:
    - The code of the entity is directly annotated with all sorts of annotations described in the javax.persistence package.
  - XML descriptors:
    - The mapping is defined in an external XML file that will be deployed with the entities.
    - Can be used instead of (or in addition to) annotations
    - Better portability and maintenance in some cases
JPA Concepts (3)

- **Entity**: an application-defined object with the following characteristics
  - it can be made persistent
  - it has a persistent identity (i.e. a key)
  - it’s partially transactional... that is: an entity is created, updated and deleted within a transaction. However, in-memory entities can be changed without the changes being persisted
  - it’s not a primitive, a primitive wrapper, or built-in object

- **Entity metadata**: describes every entity
- **Entity manager**: enables API calls to perform operations on an entity
  - Until an entity manager is used to create, read, or write an entity
    - the entity is just a regular non-persistent Java object.

JPA Concepts (4)

- **The Entity-Relationship Model** has been around since 1975 by Peter Chen.
  - Entities are the objects or things in the model
  - Relationships are the connections or links between entities
- **JPA requires** that you identify the classes of those objects that you will store in a database.
  - Use the annotation `@Entity` to define classes that it will map to a relational database.
- **Relationships have cardinality constraints**
  - Use the annotations `@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany` to define relationships among entities.
Entity Class

- Entity classes make up the model in the MVC design pattern.
  - Essentially an entity is a plain old Java object (POJO) that holds data for the purposes of saving that data in a database.
- An entity class has the following characteristics:
  - The class is annotated with the @Entity annotation so that the Enterprise container will recognize it as an entity class.
  - The class's member variables should not be public. Access to the variables should be through accessor (getter) and mutator (setter) methods.
  - The class should have a no-argument constructor. It may have other constructors, but the no-argument constructor must exist.
- The member variables that will be persisted to a database can be
  - primitive types, serializable classe types, or Multi value types (Collections, Maps and Arrays).

GuestBook.java Entity Class (1)

```java
@Entity
public class GuestBook implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;
    @Column (name="FIRSTNAME")
    private String firstName;
    private String lastName;
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }
}
```
```java
@override
public boolean equals(Object object) {
    // TODO: Warning - this method won't work in the case the id fields are not set
    if (!(object instanceof GuestBook)) {
        return false;
    }
    GuestBook other = (GuestBook) object;
    if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.id))) {
        return false;
    }
    return true;
}

@override
public String toString() {
    return "entity.GuestBook[ id=" + id + "]";
}

/**
 * @return the firstName
 */
public String getFirstName() {
    return firstName;
}

/**
 * @param firstName the firstName to set
 */
public void setFirstName(String firstName) {
    this.firstName = firstName;
}

/**
 * @return the lastName
 */
public String getLastName() {
    return lastName;
}

/**
 * @param lastName the lastName to set
 */
public void setLastName(String lastName) {
    this.lastName = lastName;
}
```

```java
/**
 * @param firstName the firstName to set
 */
public void setFirstName(String firstName) {
    this.firstName = firstName;
}

/**
 * @return the lastName
 */
public String getLastName() {
    return lastName;
}

/**
 * @param lastName the lastName to set
 */
public void setLastName(String lastName) {
    this.lastName = lastName;
}
```
An entity class is a simple POJO class decorated with some annotations.

- These annotations are essential for the object-relational mapping work.
- Entity classes should have the @Entity annotation applied so that Java can tell that it is supposed to be an entity class instead of a simple non-persistent class.
- The id member variable has two annotations, @Id and @GeneratedValue.
  - Each entity must have an @Id annotation.
  - This tells JPA that the data element is the primary key in the database. Without a primary key JPA would have no way to distinguish one record from another in the database.
  - The @GeneratedValue annotation is optional

The firstName instance variable is annotated with @Column.

- This is an optional annotation that tells JPA that the database column name for this variable is firstName. If this annotation was not present JPA would look for a column with the same name as the variable. Because the "id" has no @Column annotation JPA will look for a column named "id" to store the contents of the id.

An entity class may use either persistent fields or persistent properties depending on where the annotations are placed.

- If the annotations are applied to the member variables then the entity will use persistent fields.
- If persistent properties are used, then the member variable's accessor (getter) method would be annotated.
- Do not mix two different annotations
- Using persistent fields is preferred (see chapter 32 of EE 6 tutorial)
Annotations (3)

- A primary key in a database table need not be a single column. Likewise an entity's Id doesn't have to be a single variable.
- Simple PK
  - Use @Id annotation on an attribute, can be one of the typical scalar data types (including String, Date, Integer)
  - Use @GeneratedValue to have the value of the PK be generated automatically by the persistence provider
- Composite PK -- two ways to annotate it
  - @EmbeddedId (recommended): create a Java class for the PK annotated with @Embeddable and then create a field of this type in the Entity annotated with @EmbeddedId
    - Embeddable objects do not have a persistent identity on their own; they can be embedded only within owning entities.
  - @IdClass: create a Java class for the PK w/no annotations, then annotate the entity with @IdClass
  - Both approaches get mapped to the same table structure

@Column Annotation

- Some important defaults:
  - name of column is the same as that of the field
  - null values allowed on non-primitive data types
  - Strings have a length of 255
- Sample use...
  - @Column(name="firstName")
  - @Column(name="firstName", unique=false, length=100)
Other annotations

- **@Transient**
  - To annotate a field that is not to be persisted

- **@Access**
  - To annotate whether FIELD or PROPERTY access type is to be used... avoid using unless necessary!

- **@ElementCollection and @CollectionTable**
  - For annotating a field that is a Collection Java types
  - In the DB, a table will be created for the values in the collection

Entity Relationships (1)

- Relationships between entities can be unidirectional or bidirectional

- **Unidirectional relationships**
  - only one entity knows about the other entities in the relationship
  - have an "owning" side which maintains the relationship in the DB

- **Bidirectional relationships**
  - have both an owning and an "inverse" side.

- The owning side determines how and when updates affect the relationship.

- Also, the owning side usually contains the foreign key to the other entity.
Entity Relationships (2)

When bidirectional relationships are used one of the entities must be specified as the owner.
- Ownership simply means that one of the entities is responsible for maintaining the relationship.
- There is no extra programming involved in maintaining the relationship, it is a matter of which of the tables in the relational database has the primary key and which one as the foreign key.
- In many to one relationships the many side is always the owning side (that is, the side with the @ManyToMany annotation).
- The one to many side cannot be the owning side in JPA. For one to one and many to many relationships either side may be the owning side.

Entity Relationships (3)
Entity Relationships (4)

To make clear which side is the owning side of a relationship we add some additional parameters to the side that does not own the relationship.

- The entity which does not own the relationship will have the mappedBy="XXX" property in the relationship annotation where XXX is the name of the field or property in the other entity that refers to this one.

Example

--In CustomerOrder.java entity

```java
@OneToMany(cascade = CascadeType.ALL, mappedBy = "customerOrder")
    private Collection<OrderedProduct> orderedProductCollection;
```

--In OrderedProduct.java entity

```java
@ManyToMany(optional = false)
    private CustomerOrder customerOrder;
```

OrderedProduct.java (Affablebean)

```java
@Entity
@Table(name = "ordered_product")
@NamedQueries({
    @NamedQuery(name = "OrderedProduct.findAll", query = "SELECT o FROM OrderedProduct o"),
    @NamedQuery(name = "OrderedProduct.findByCustomerOrderId", query = "SELECT o FROM OrderedProduct o WHERE o.orderedProductPK.customerOrderId = :customerOrderId"),
    @NamedQuery(name = "OrderedProduct.findByProductId", query = "SELECT o FROM OrderedProduct o WHERE o.orderedProductPK.productId = :productId"),
    @NamedQuery(name = "OrderedProduct.findByQuantity", query = "SELECT o FROM OrderedProduct o WHERE o.quantity = :quantity")})
public class OrderedProduct implements Serializable {
    private static final long serialVersionUID = 1L;
    @EmbeddedId
    protected OrderedProductPK orderedProductPK;
    @Basic(optional = false)
    @NotNull
    @Column(name = "quantity")
    private short quantity;
    @JoinColumn(name = "product_id", referencedColumnName = "id", insertable = false, updatable = false)
    @ManyToOne(optional = false)
    private Product product;
    @JoinColumn(name = "customer_order_id", referencedColumnName = "id", insertable = false, updatable = false)
    @ManyToOne(optional = false)
    private CustomerOrder customerOrder;
```
Elements in Relationship Annotations

- **Important elements**
  - **cascade**: CascadeType.{MERGE,PERSIST,REFRESH,REMOVE,ALL}
    - specifies which operations will be propagated to the target of the relationship
  - **fetch**: FetchType.{LAZY, EAGER}
    - whether the persistence provider will automatically fetch the target of the relationship whenever the source is retrieved
    - default is LAZY for 1-to-M, and M-to-M; EAGER for M-to-1 and 1-to-1
  - **optional**: true (default) or false
    - if set to false, then a non-null relationship must always exist

- **@JoinColumn**
  - Allows you to customize the join column (i.e. foreign key)

Storing and Manipulating Entities

- **The persistence context is the collection of managed entities in a particular data store.**
  - The persistence context is manipulated by means of the EntityManager interface.
  - The EntityManager is the means by which an application accesses the persistence context.
  - Unlike session beans, entities are not directly managed by the enterprise container. They are managed by the persistence context (which does not require an enterprise container).
  - **Example**
    - In the Java SE environment, the application manages the life cycle of an entity manager -- there is no container
    - Creating an entity manager in Java SE
      ```java
      EntityManagerFactory emf = Persistence.createEntityManagerFactory("SportsPU");
      EntityManager em = emf.createEntityManager();
      ```
    - Start a transaction from which to modify data
      ```java
      em.getTransaction().begin();
      ```
    - A transaction is not needed if you just query to retrieve data
Entity Life Cycle (1)

- When you create a new instance of an entity class you have an entity in the new state.
- An entity object becomes Managed when it is persisted to the database via an EntityManager's persist method, which must be invoked within an active transaction.
  - On transaction commit, the owning EntityManager stores the new entity object to the database.

Entity Life Cycle (2)

- Entity objects retrieved from the database by an EntityManager are also in the Managed state.
- If a managed entity object is modified within an active transaction the change is detected by the owning EntityManager and the update is propagated to the database on transaction commit.
- A managed entity object can also be retrieved from the database and marked for deletion, by using the EntityMananger's remove method within an active transaction.
- The entity object changes its state from Managed to Removed, and is physically deleted from the database during commit.
- The last state, Detached, represents entity objects that have been disconnected from the EntityManager.
  - All the managed objects of an EntityManager become detached when the EntityManager is closed.
  - Working with detached objects, including merging them back to an EntityManager is possible.
Persistence Context

- The persistence context is the collection of all the managed objects of an EntityManager.
  - If an entity object that has to be retrieved already exists in the persistence context, the existing managed entity object is returned without actually accessing the database (except retrieval by refresh, which always requires accessing the database).
- The main role of the persistence context is to make sure that a database entity object is represented by no more than one in-memory entity object within the same EntityManager.
- Every EntityManager manages its own persistence context. Therefore, a database object can be represented by different memory entity objects in different EntityManager instances.
  - Retrieving the same database object more than once using the same EntityManager should always result in the same in-memory entity object.

EntityManager API

- persist:
  - make an instance managed and persistent
- find:
  - search for an entity of a specified class and PK value
- getReference:
  - like find but with lazy fetch
- remove:
  - remove the entity from the persistence context and from the DB
- flush:
  - forces any changes to be flushed to the database, otherwise they will be done at commit time of the transaction -- changes can still be rolled back.
- refresh:
  - overwrites the current state of a managed entity with data that is in the DB
Querying via JPQL

- The results obtained from SQL are in the form of rows and columns (tables), whereas JPQL uses an entity or a collection of entities.
- JPQL syntax is object-oriented
- JPQL is mapped to SQL and JDBC calls
  > Done automagically!

JPQL Example

```java
@WebServlet(name = "GuestBookServlet", urlPatterns = {"/GuestBook"})
public class GuestBookServlet extends HttpServlet {

    @PersistenceContext(unitName = "SimpleJPAPU")
    private EntityManager em;

    @Resource
    UserTransaction utx;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
            try {
                utx.begin();
                GuestBook g = new GuestBook();
                g.setFirstName(request.getParameter("firstname"));
                g.setLastName(request.getParameter("lastname"));
                em.persist(g);
                List<GuestBook> guestList = em.createQuery("select g from GuestBook g").getResultList();
                request.setAttribute("guestList", guestList);
                utx.commit();
                this.getServletContext().getRequestDispatcher("/index.jsp").forward(request, response);
            } catch (Exception e) {
                e.printStackTrace();
                try {
                    utx.rollback();
                } catch (Exception ex) {
                    ------
                }
            }
        }
```
GuestBook App based on JPA

index.jsp (JPAGuestBookK App)

```jsp
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Guest Book Based on Java Persistence API</title>
</head>
<body>
<h1>Guest Users</h1>
<c:forEach var="guest" items="${guestList}">
<p>${guest.firstName} ${guest.lastName}</p>
</c:forEach>
<form method="post" action="GuestBook">
<h1>Sign the Guest Book</h1>
<table>
<tr><td>First Name:</td><td><input type="text" name="firstname" /></td></tr>
<tr><td>Last Name:</td><td><input type="text" name="lastname" /></td></tr>
<tr><td colspan="2">
<input type="hidden" name="action" value="add"/>
<input type="submit" value="Sign on Guest Book" />
</td></tr>
</table>
</form>
</body>
</html>
```
Create GuestBookServlet

![Image](image.png)

GuestBookServlet Code

```java
@WebServlet(name = "GuestBookServlet", urlPatterns = {"/GuestBook")
public class GuestBookServlet extends HttpServlet {

@PersistenceContext(unitName = "JPAGuestBookPU")
private EntityManager em;

@Resource
UserTransaction utx;

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        try {
            utx.begin();
            GuestBook g = new GuestBook();
            g.setFirstName(request.getParameter("firstname"));
            g.setLastName(request.getParameter("lastname"));
            em.persist(g);
            List<GuestBook> guestList = em.createQuery("select g from GuestBook g").getResultList();
            request.setAttribute("guestList", guestList);
            utx.commit();
            this.getServletContext().getRequestDispatcher("/index.jsp").forward(request, response);
        }
    }
```
GuestBookServlet (cont’d)

```java
try {
    utx.rollback();
} catch (Exception ex) {
    ex.printStackTrace();
}
```

Create Database (MySQL) for the App

```java
protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    try {
        utx.begin();
        GuestBook g = new GuestBook();
        g.setFirstName(request.getParameter("fname"));
        g.setLastName(request.getParameter("lname"));
        g.persist();
        List<GuestBook> guestList = ex.createServlet();
        request.setAttribute("guestList", guestList);
        utx.commit();
    } catch (Exception e) {
        e.printStackTrace();
    }
```
JPAGuestBook database

Connect to the database
Connect to the database (JDBC/JNDI)

Create GuestBook.java entity class
GuestBook.java entity class created

Add two additional instance variables to the entity class:
private String firstName;
private String lastName;
Import GuestBook entity class

```java
try {
    utx.begin();

    GuestBook gb = new GuestBook();
    gb.setFirstName("firstname");
    gb.setLastName("lastname");

    List<GuestBook> guestList = gb.createQuery("select g from GuestBook g where g.firstName = 1 and g.lastName = 1");
    if (guestList.isEmpty()) {
        this.setRequestContext().getRequestDispatcher("/index.jsp").forward();
        throw ServletException, IOException {
            try {
                utx.rollback();
            } catch (Exception e) {
```
Import other Necessary Interfaces

```java
import java.annotation.Resource;
import java.persistence.PersistenceContext;
import java.servlet.ServletException;
import java.servlet.annotation.WebServlet;
import java.servlet.http.HttpServlet;
import java.servlet.http.HttpServletRequest;
import java.servlet.http.HttpServletResponse;
import java.transaction.UserTransaction;

/**
 * @WebServlet(name = "GuestBookServlet", urlPatterns = {"/GuestBook"})
 * @PersistenceContext(unitName = "SimpleJPA")
 * public class GuestBookServlet extends HttpServlet {
 *  private EntityManager em;
 *  
 *  // Add import for java.persistence.EntityManager
 *  // Create class "EntityManager" in package controller
 *  // Create class "GuestBookServlet" in controller
 */
```

Run JPAGuestBook App

Guest Users
Bob Schneider
Katherine St John
MMkGirlCat

Sign the Guest Book
First Name: 
Last Name: 
Sign on Guest Book
guestbook table (in MySQL DBMS)

EJB
(Enterprise Java Beans)
Types of EJBs

- **SessionBean:**
  - EJB used for implementing high-level business logic and processes
  - Session beans handle complex tasks that require interaction with other components (servlet controllers, entities, web services, messaging, etc.)
- **Timer Service**
  - EJB used for scheduling tasks
- **Message Driven Bean**
  - EJB used to integrate with external services via asynchronous messages using JMS.
  - Usually, delegate business logic to session beans

Session Beans

- Enterprise session beans are invoked by a client in order to perform a specific business operation.
  - The name *session* implies that a bean instance is available for the duration of a "unit of work".
- **The EJB 3.1 specification describes a typical session object as having the following characteristics:**
  - Executes on behalf of a single client
  - Can be transaction-aware
  - Updates shared data in an underlying database
  - Does not represent directly shared data in the database, although it may access and update such data
  - Is relatively short-lived
  - Is removed when the EJB container crashes. The client has to re-establish a new session object to continue computation.
Types of Session Beans

- EJB provides three types of session beans: stateful, stateless, and singleton.
  - **Stateless**: Stateless beans are used for operations that can occur in a single method call.
    - When the method finishes processing, the client-specific state of the bean is not retained. A stateless session bean therefore does not maintain a conversational state with the client.
  - **Stateful**: The state of the bean is maintained across multiple method calls. Useful for tasks that have to be done in several steps.
    - The "state" refers to the values of its instance variables. Because the client interacts with the bean, this state is often called the conversational state.
  - **Singleton**: A singleton session bean is instantiated once per application, and exists for the lifecycle of the application.
    - Singleton session beans are designed for circumstances where a single enterprise bean instance is shared across and concurrently accessed by clients.

Stateless Session Bean example
**ConverterBean.java**

```java
import java.math.BigDecimal;
import javax.ejb.Stateless;

@Stateless
public class ConverterBean {

    // Add business logic below. (Right-click in editor and choose
    // "Insert Code > Add Business Method")

    private BigDecimal euroRate = new BigDecimal("0.0100168");
    private BigDecimal yenRate = new BigDecimal("79.3915");
    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = dollars.multiply(yenRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = yen.multiply(euroRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}
```

**ConverterServlet.java**

```java
@WebServlet(name = "ConverterServlet", urlPatterns = {"/ConverterServlet"})
public class ConverterServlet extends HttpServlet {

    @EJB
    ConverterBean converter;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        // Output the results
        "<h1>Servlet ConverterServlet at " + request.getContextPath() + 
        
        try {
            String amount = request.getParameter("amount");
            BigDecimal d = new BigDecimal(amount);
            BigDecimal result = amount.multiply(euroRate);
            return result.setScale(2, BigDecimal.ROUND_UP);
        }
    }
}
```
Stateless Session Bean

Creating a new enterprise application in a standard project. You can also create an EAR module project and Web application project in the enterprise application. A standard project uses an IDE-generated Ant build script to build and run your projects.
HelloSessionBean.java

Right Click inside the Session bean code, then click InsertCode option

```
package ejb;

import java.ejb.Stateless;
import java.ejb.LocalBean;

/**
 * @Author gjiang
 */
@Stateless
@LocalBean
public class HelloSessionBean {

    // Add business logic below. (Right-click in edit
    // 'Insert Code > Add Business Method')
```

helloSessionbean.java coding
Adding Servlet

**HelloServlet.java**

1. Choose File Type
2. Name and Location
3. Configure Servlet Deployment

- Class Name: `HelloServlet`
- Project: `HelloServlet-maven`
- Location: `Source Packages`
- Package: `hello`

Created Files:
- `/HelloServlet-maven/target/HelloServlet-maven.jar` (JUnit test)
- `/src/main/java/HelloServlet.java`
Coding inside `processRequest()` method

```java
protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
        // TODO: output your code here. You may use the following sample code.

        HelloServlet.setTitle();
        HelloServlet.set(request, getServletContext().getContextPath() + "/'M'>");
        // more code...
    } catch (IOException e) {
        out.close();
    }
}
```
Run HelloSessionBean app

![Servlet HelloServlet at /HelloSessionBean-war](image)

Stateful and Singleton Session Beans

- **Stateful:**
  - See the cart example in EE 6 tutorial (chapter 24, pp. 465)
- **Singleton:**
  - See the counter example in EE 6 tutorial (chapter 24, pp. 472)
Affablebean Project
(ch7 of the main reference R2)

Affable App based on EJB/JPA

- A Java EE container contains three essential components: a web (JSP/Servlet) container, an EJB container, and a persistence provider.
- Entity classes that you create are managed by the persistence provider.
  - The session beans that you create are managed by the EJB container.
  - Views are rendered in JSP pages (dispatched by the Controller Servlet), which are managed by the web container.
Adding Entity Classes (1)

Adding Entity Classes (2)
Adding Entity Classes (3)

Adding Entity Classes (4)
Adding Entity Classes (5)

- The class names for the entities are based on database tables.
  - For example, the CustomerOrder entity will be mapped to the customer_order database table.

- The 'Generate Named Query Annotations for Persistent Fields' option is selected by default.
  - We can use various named queries.

- The 'Create Persistence Unit' option is selected by default.
  - A persistence unit is a collection of entity classes that exist in an application.
  - The persistence unit is defined by a persistence.xml configuration file, which is read by your persistence provider.
  - Enabling this option therefore means that the wizard will also generate a persistence.xml file and populate it with default settings.

Persistence.xml (1)

![Persistence.xml Configuration](image)
### persistence.xml(2)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="AffableBeanPU" transaction-type="JTA">
    <jta-data-source>jdbc/affablebean</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties/>
  </persistence-unit>
</persistence>
```

### Category.java Entity (1)

```java
@Entity
@Table(name = "category")
@NamedQueries({
  @NamedQuery(name = "Category.findAll", query = "SELECT c FROM Category c"),
  @NamedQuery(name = "Category.findById", query = "SELECT c FROM Category c WHERE c.id = :id"),
  @NamedQuery(name = "Category.findByName", query = "SELECT c FROM Category c WHERE c.name = :name")
})
public class Category implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Basic(optional = false)
  @Column(name = "id")
  private Short id;
  @Basic(optional = false)
  @NotNull
  @Size(min = 1, max = 45)
  @Column(name = "name")
```
Category.java Entity (2)

```java
private String name;

@OneToMany(cascade = CascadeType.ALL, mappedBy = "categoryId")
private Collection<Product> productCollection;

public Category() {
}

public Category(Short id) {
    this.id = id;
}

public Category(Short id, String name) {
    this.id = id;
    this.name = name;
}

public Short getId() {
    return id;
}
```

89

Category.java Entity (3)

```java
public void setId(Short id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

@XmlTransient
public Collection<Product> getProductCollection() {
    return productCollection;
}

public void setProductCollection(Collection<Product> productCollection) {
    this.productCollection = productCollection;
}
```

90
Category.java Entity (4)

```java
@Override
public int hashCode() {
    int hash = 0;
    hash += (id != null ? id.hashCode() : 0);
    return hash;
}

@Override
public boolean equals(Object object) {
    // TODO: Warning - this method won't work in the case the id fields are not set
    if (!(object instanceof Category)) {
        return false;
    }
    Category other = (Category) object;
    if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.id))) {
        return false;
    }
    return true;
}
```

```java
@Override
public String toString() {
    return "entity.Category[ id=" + id + "]";
}
```

OrderedProduct.java entity

```java
private static final String QUERY = "SELECT o FROM OrderedProduct o WHERE o.productPK.productOrder = :productOrder";
```
Note that an additional entity class, `OrderedProductPK`, is generated.

- Recall that the data model’s `ordered_product` table uses a composite primary key that comprises the primary keys of both the `customer_order` and `product` tables.
- Because of this, the persistence provider creates a separate entity class for the composite key, and embeds it into the `OrderedProduct` entity. You can open `OrderedProduct` in the editor to inspect it. JPA uses the `@EmbeddedId` annotation to signify that the embeddable class is a composite primary key.

---

Adding Session Beans (1)

- Each session bean will contain basic access methods for its respective entity class.
- A session facade is a design pattern advertised in the Enterprise BluePrints program.
  - As stated in the Core J2EE Pattern Catalog, it attempts to resolve common problems that arise in a multi-tiered application environment, such as:
    - Tight coupling, which leads to direct dependence between clients and business objects
    - Too many method invocations between client and server, leading to network performance problems
    - Lack of a uniform client access strategy, exposing business objects to misuse
A session facade abstracts the underlying business object interactions and provides a service layer that exposes only the required functionality.

- It hides from the client's view the complex interactions between the participants.
- The session bean (representing the session facade) manages the relationships between business objects.
- The session bean also manages the life cycle of these objects by creating, locating, modifying, and deleting them as required by the workflow.

Adding Session Beans (3)

[Image of a software interface demonstrating the creation of a session bean]

- Creates Session EJB Facade based on Entity class. This template creates Session EJB for every Entity class with basic access methods.
Adding Session Beans (4)

Adding Session Beans (5)
public abstract class AbstractFacade<T> {
    private Class<T> entityClass;

    public AbstractFacade(Class<T> entityClass) {
        this.entityClass = entityClass;
    }

    protected abstract EntityManager getEntityManager();

    public void create(T entity) {
        getEntityManager().persist(entity);
    }

    public void edit(T entity) {
        getEntityManager().merge(entity);
    }

    public void remove(T entity) {
        getEntityManager().remove(getEntityManager().merge(entity));
    }

    public T find(Object id) {
        return getEntityManager().find(entityClass, id);
    }

    public List<T> findAll() {
        javax.persistence.criteria.CriteriaQuery cq =
            getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        return getEntityManager().createQuery(cq).getResultList();
    }

    public List<T> findRange(int[] range) {
        javax.persistence.criteria.CriteriaQuery cq =
            getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        q.setMaxResults(range[1] - range[0]);
        q.setFirstResult(range[0]);
        return q.getResultList();
    }
}
AbstractFacade.java (3)

```java
public int count() {
    javax.persistence.criteria.CriteriaQuery cq =
    getEntityManager().getCriteriaBuilder().createQuery();
    javax.persistence.criteria.Root<T> rt = cq.from(entityClass);
    cq.select(getEntityManager().getCriteriaBuilder().count(rt));
    javax.persistence.Query q = getEntityManager().createQuery(cq);
    return ((Long) q.getSingleResult()).intValue();
}
```

Session Façade Classes

- Session Beans for Entity Classes wizard generates facade classes. Namely, boiler-plate code that is common to all classes is factored out into an abstract class named AbstractFacade.
  - All other Façade classes extends AbstractFacade.

- All of the generated session facades
  - instantiate an EntityManager using the @PersistenceContext annotation. @PersistenceContext(unitName = "AffableBeanPU") private EntityManager em
  - The @PersistenceContext annotation is used to inject a container-managed EntityManager into the class.
    - In other words, we rely on GlassFish' EJB container to open and close EntityManagers as and when needed.
    - The unitName element specifies the AffableBeanPU persistence unit, which has been defined in the application's persistence.xml file.
ProductFacade.java

@Stateless
public class ProductFacade extends AbstractFacade<Product> {
    @PersistenceContext(unitName = "AffableBeanPU")
    private EntityManager em;

    @Override
    protected EntityManager getEntityManager() {
        return em;
    }

    public ProductFacade() {
        super(Product.class);
    }
}

Entity Manager Revisited

- The EntityManager is an integral component of the Java Persistence API, and is responsible for performing persistence actions on the database.
  - The JPA EntityManager interface manages entities in terms of actually providing persistence services.
  - While entities tell a JPA provider how they map to the database, they do not persist themselves.
  - The EntityManager interface reads the ORM metadata for an entity and performs persistence operations.

- Affablebean application now
  - Contains a persistence model of the affablebean database in the form of JPA entity classes.
  - It also contains a session facade consisting of Enterprise beans that can be used to access the entity classes.
Using Session and Entity Beans

- We use the data access methods provided by the session beans, and store the data in scoped variables so that it can be retrieved from front-end page views.

index.jsp

- The index page requires data for the four product categories. In our current setup, the JSTL `<sql>` tags query the database for category details each time the index page is requested.
  - Since this information is rarely modified, it makes more sense from a performance standpoint to perform the query only once after the application has been deployed, and store the data in an application-scoped attribute.
  - We can accomplish this by adding this code to the ControllerServlet's init method.
ControllerServlet

```
import javax.ejb.EJB;
import session.CategoryFacade;

public class ControllerServlet extends HttpServlet {

    @EJB
    private CategoryFacade categoryFacade;

    @Override
    public void init() throws ServletException {

        // store category list in servlet context
        getServletContext().setAttribute("categories", categoryFacade.findAll());
    }
}
```

categoryFacade.findAll()

- We apply the facade class' findAll method to query the database for all records of Category.
- We then set the resulting List of Category objects as an attribute that can be referenced by the "categories" string.
  - Placing the reference in the ServletContext means that the reference exists in a scope that is application-wide.
Modify index.jsp

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<sql:query var="categories" dataSource="jdbc/affablebean">
    SELECT * FROM category
</sql:query>
```

```
Will be
```
```
<c:forEach var="category" items="${categories.rows}"
Will be
```
```
<c:forEach var="category" items="${categories}"
```

DataSource resource-ref removed

```
In web.xml
```
```
<resource-ref>
    <res-ref-name>jdbc/affablebean</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```
```
Modify category.jsp

- The category page requires three pieces of data in order to render properly:
  - category data: for left column category buttons
  - selected category: the selected category is highlighted in the left column, and the name of the selected category displays above the product table
  - product data for selected category: for products displayed in the product table
- comment out (Ctrl-/) the JSTL <sql> statements that are listed at the top of the file.
- <c:forEach var="category" items="${categories.rows}">
  - <c:forEach var="category" items="${categories}">

Selected category

- To retrieve the selected category, we can use the categoryFacade that we already created to find the Category whose ID matches the request query string.
- ControllerServlet
  if (userPath.equals("/category")) {
    // get categoryId from request
    String categoryId = request.getQueryString();
    if (categoryId != null) {
      // get selected category
      Category selectedCategory = categoryFacade.find(Short.parseShort(categoryId));
      // place selected category in request scope
      request.setAttribute("selectedCategory", selectedCategory);
    }
  }
- Category.jsp
  <p id="categoryTitle">${selectedCategory.rows[0].name}</p>
  <p id="categoryTitle">${selectedCategory.name}</p>
In order to retrieve all products for a selected category, we'll make use of the Category entity's `getProductCollection()` method.

- Start by calling this method on `selectedCategory` to get a collection of all Products associated with the `selectedCategory`.
- Then store the collection of products as an attribute in the request scope.
- Finally reference the scoped attribute from the `category.jsp` page view.

```java
// get all products for selected category
Collection<Product> categoryProducts = selectedCategory.getProductCollection();

// place category products in request scope
request.setAttribute("categoryProducts", categoryProducts);
```

```xml
<c:forEach var="product" items="${categoryProducts.rows}" varStatus="iter">
  <!-- your code here -->
</c:forEach>
```