Fundamentals of Java Enterprise Components

Session: 11

Java Persistence API

Objectives

- Define Java Persistence API
- Describe how Java Persistence API works
- Describe entities and their mutual interactions
- Describe managing and querying the entities
- Explain creation of database schema
- Explain the basics of Java Persistence Query Language

Introduction

Java Persistence API was introduced in Java EE 5 as standard persistence API.

An ER model can be mapped onto an object-oriented application, where entities are mapped onto objects and relationships among the objects are converted into methods.

ER model is the basis of Java Persistence API. javax.persistence.Entity manages the data as entities.

Entities 1-4

An entity is an object in the Java application whose state has to be saved to the database.

A class representing a set of objects can be mapped to the table definition.

According to the Persistence API, following are the requirements of the entity classes:

- Every entity class must have a default constructor with public or protected access.
- The entity classes cannot be defined as final, nor the methods of entity class can be final.
- If the persistent object is passed through a session bean's remote interface, then the entity should be synchronized.
- The variables of the persistent objects must be declared protected or private.
- An entity class may extend other entity or non-entity classes.

Entities 2-4

The entity class variables can assume any one of the primitive data types or wrappers of primitive data types. It can also be an object of any of the following types:

- java.math.BigInteger
- java.math.BigDecimal
- java.util.Date
- java.util.Calendar
- java.sql.Date
- java.sql.Time
- java.sql.TimeStamp
- User-defined serializable types
- byte[]
- Byte[]
- char[]
- Character[]
- Enumerated types
- Other entities and/or collections of entities

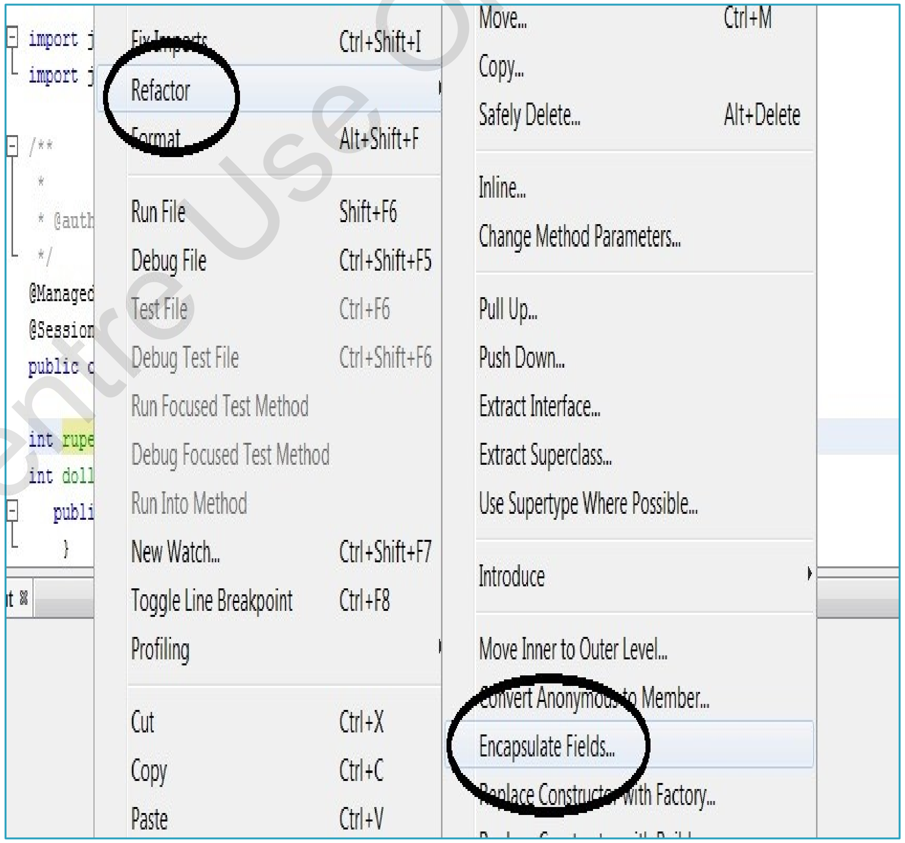
Entities 3-4

A persistent field is one whose values correspond to instance variables of an object and are obtained at runtime.

A persistent property gets the values through the getter and setter methods defined in the entity class.

The Netbeans IDE generates these methods for the attributes as shown in the figure.

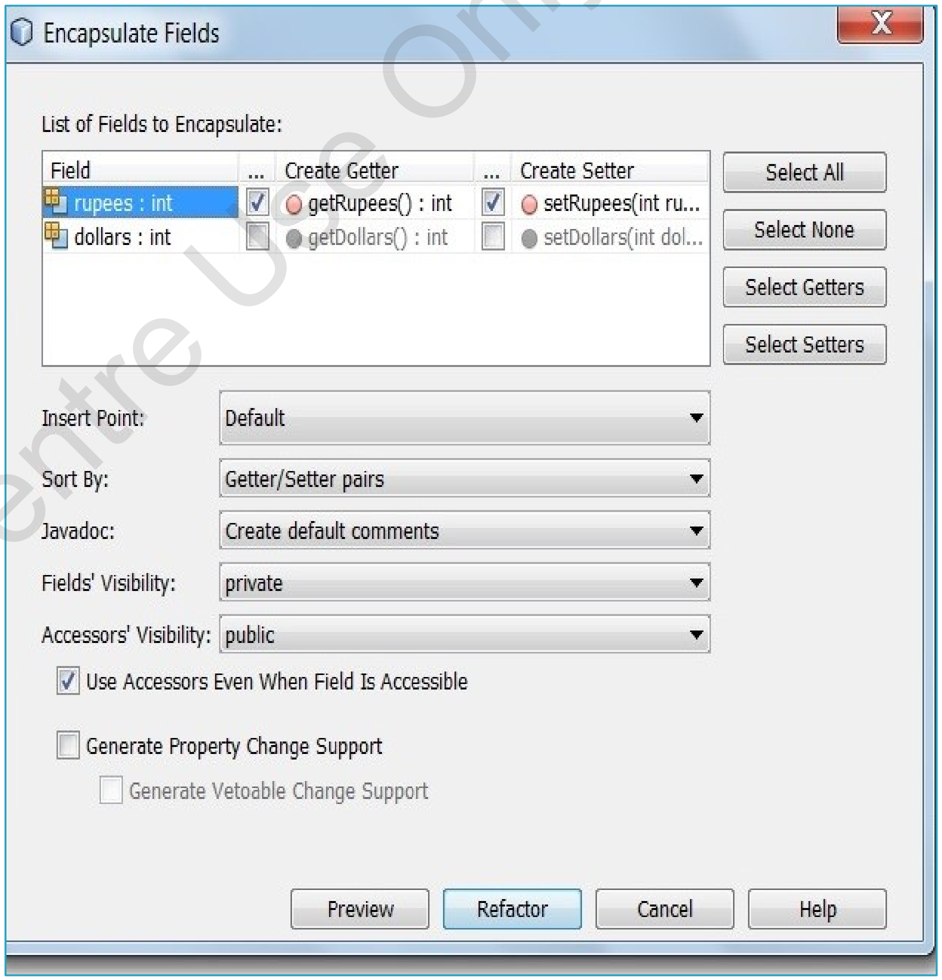
The menu in figure appears on right-clicking the variable that is to be encapsulated.



Entities 4-4

The screen shown in the figure appears through which the getter and setter methods can be created.

These getter and setter methods are used to fetch and return the values for the attributes.


Encapsulate Fields

List of Fields to Encapsulate:

Field	...	Create Getter	...	Create Setter
rupees : int	✓	getRupees() : int	✓	setRupees(int ru...
dollars : int		getDollars() : int		setDollars(int dol...

Insert Point: Default

Sort By: Getter/Setter pairs

Javadoc: Create default comments

Fields' Visibility: private

Accessors' Visibility: public

[✓] Use Accessors Even When Field Is Accessible

[ ] Generate Property Change Support

[ ] Generate Vetoable Change Support

Collections as Entity Fields and Properties

- An entity may also have a collection as a property.
- The entity field may use any one of the following interfaces to represent a collection of objects:
  - java.util.Collection
  - java.util.Set
  - java.util.List
  - java.util.Map
- The getter and setter methods must be appropriately defined for these collections in the entity class.
- Given code demonstrates the getter and setter methods for a Collection object, coll.

```
package Product;
import java.util.Collection;
import java.util.Iterator;
import javax.ejb.Stateless;
public class ProductsBean implements
java.io.Serializable
{
   private String proName;
   private String description;
   private Collection coll = new
Collection() {
/*implement all the abstract methods*/
}
    public Collection getColl() {
        return coll;
    }
public void setColl(Collection coll) {
        this.coll = coll;
    }
```

Validating Persistent Fields and Properties 1-3

- The Java Beans Validation API provides mechanisms for validating application data.
- The validation mechanism is embedded into the Java EE containers and it can be applied to the persistent entity classes.
- Bean validation provides for defining a set of constraints, both default and custom constraints on the persistence fields and properties.

Validating Persistent Fields and Properties 2-3

Following table lists Bean validation's built-in constraints, defined in the javax.validation.constraints package that can be applied to the value of a field or property:

Constraint	Description	Example
@AssertFalse	The value must be false.	@AssertFalse boolean isAbsent;
@AssertTrue	The value must be true.	@AssertTrue boolean isActive;
@Digits	The value must be a number within a specified range.	@Digits(integer=5, fraction=2) BigDecimal amount;
@Max	The value must be an integer value lower than or equal to the number in the value element.	@Max(10) int total;
@Min	The value must be an integer value greater than or equal to the number in the value element.	@Min(5) int quantity;
@NotNull	The value must not be null.	@NotNull String firstname;
@Null	The value must be null.	@Null String tempData;

@Pattern	The value must match the regular expression defined in the regexp element.	@Pattern(regexp="\\(\\d{3}\\)\\d{3}
		−\\d{4}")
		String phone;

@Size	The size must match the specified boundaries.	@Size(min=3, max=150) String message;

Validating Persistent Fields and Properties 3-3

Following code demonstrates the usage of bean validation constraint to specify non-null persistent fields:

```
public class Login {
 @NotNull
 private String username;
@NotNull
private String password;
}
```

More than one constraints can be applied on a single Java beans component as shown in the following code:

```
public class Login {
@NotNull
@Size(min=1, max=20)
private String username;
@NotNull
@Size(min=1, max=20)
private String password;
}
```

Primary Keys

Every row in a database has to be uniquely identified through primary key.

There are two variants of a primary key:

- Simple primary key
- Composite primary key

If the primary key of an entity is implemented as a class, then following are the requirements of the class:

- The primary key class must be declared public or protected.
- The class must be serializable.
- The primary key class must have a default constructor defined.
- It must implement the hashCode() and equals() methods.
- If the primary key is a composite key, then it must be mapped to multiple fields or properties of the entity class. It should be represented as an embeddable class.
- The names and data types of the attributes of the entity class and names of the attributes of the composite primary key should match.

Implementing Entity Relationships

The entities in an object-oriented application are related with each other.

The EntityManager class manages the entities and their association in the domain.

Following are the variants of the entity relationships:

- An association between two types of entities is said to be one-to-one when one entity of a type is associated with exactly one entity of another entity type. It is represented in the Java program through @OneToOne annotation.
- One-to-many association is where an entity of one type can be associated with more than one entities of another type. This association is represented through @OneToMany annotation.
- Many-to-one association implies that many entities of certain type are associated with one entity of another type. This association in the code is represented through @ManyToOne annotation.
- Many-to-many association implies that every entity of certain type can be associated with many entities of the other type and vice-versa. In Java programs, this association is represented through @ManyToMany annotation.

Directionality of Relationships

The direction of relationship among entities can be **unidirectional** or **bidirectional**.

Unidirectional relationships

- In a unidirectional relationship, only one entity has a relationship field or property that refers to the other.
- There is only one owner of the relationship among the participating entities.

Bidirectional relationships

- In case of bidirectional relationship, among the two participating entities, one of the entities is the owning side and the other entity is termed as inverse side.
- In a bidirectional relationship, the inverse side of the relationship uses the mappedby attribute of the annotations.

Cascade Operations and Relationships

- Relationships among entities in a domain introduce dependencies in the application.
- Following table shows the cascade operations supported by the persistence API for the entities:

Cascade Operation	Description
ALL	The operation is cascaded to all the related entities.
DETACH	If the owner entity is removed from the persistence context, then the related entity is also removed.
MERGE	When the owner entity is merged with the persistence context, then the related entity is also merged.
PERSIST	If the owner entity is persisted to the storage, then the related entities are also persisted.
REFRESH	When the owner entity is refreshed in the persistence context, then the related entities are also refreshed.
REMOVE	When the parent entity is deleted from the persistence context, then all the related entities are also removed from the persistent context.

Embeddable Classes in Entities

Embeddable classes of entities are classes which represent a set of entities whose identity is dependent on the containing entity class.

Each embeddable entity class shares the identity of the containing entity class.

Given code demonstrates the usage of Embeddable class.

```
@Entity
public class Profile {
@Id
protected long SSN;
String first_name;
String last_name;
@Embedded
ZipCode zipCode;
String nationality;
...
}
```

Entity Inheritance

Inheritance is used in Java to ensure code reuse and to distinguish various groups of entities in the domain.

A database query is submitted for the base class of the entity hierarchy, then it is applied against the entire hierarchy.

Abstract entities:

- Abstract classes cannot be instantiated.
- Abstract entities can be queried.
- To declare an abstract class, @Entity annotation is specified.

Given code demonstrates an example of abstract entity.

```
@Entity
public abstract class ConceptCar{
@Id
protected String carId;
…..
}
@Entity
public class SupersonicConceptCar
extends ConceptCar{………}
@Entity
public class MinimumfuelConceptCar
extends CceptCar{………….}
```

Mapped Superclasses

Mapped superclasses are used when there is certain common state and mapping information pertaining to all the sub-classes of the super class.

Mapped superclasses can be instantiated but cannot be queried nor used in entity manager or query operations.

Mapped superclasses cannot be mapped onto a table.

Mapped superclasses can be abstract or concrete but cannot be targets of entity relationships.

To create a mapped superclass, use @MappedSuperClass annotation.

Following code demonstrates an example of Mapped superclass:

```
@MappedSuperClass
public class Student{
@Id
protected String studId;
……………..
}
public class CS_Student extends
Student{
……….
}
public class EC_Student extends
Student{
……….
}
```

Entity Inheritance Mapping Strategies

The entities of the application domain have to be systematically mapped to the tables. Following are the strategies of mapping the entities onto the database:

- A single table per class hierarchy
- A table per concrete entity class
- A join strategy

The default mapping strategy is single table per class hierarchy.

The strategy element of @Inheritance annotation is used to set the strategy. The possible values of strategy are defined in the javax.persistence.InheritanceType enumerated type as follows:

```
public enum InheritanceType {
    SINGLE_TABLE,
    JOINED,
    TABLE_PER_CLASS
};
```

A Single Table Per Class Hierarchy

This strategy corresponds to the default InheritanceType.SINGLE_TABLE.

All the classes in a hierarchy are mapped to a single table. The discriminator column in the table identifies which class the entity belongs to.

Following are the properties of a discriminator column:

- String name
- Length
- Stringcolumn Definition
- Discriminator type

Table Per Concrete Entity Class and Joined Subclass Strategy

Table per concrete entity class

- This strategy corresponds to InheritanceType.TABLE_PER_CLASS.
- Each concrete entity class is mapped to a corresponding table in the database.
- Every field and property of the entity class including the inherited classes and properties are translated to a column of the table.

Joined subclass strategy

- This strategy corresponds to InheritanceType.JOINED.
- Uses SQL join operation to merge the data from different classes in the hierarchy.
- Root of the class hierarchy is represented by a single table and every subclass of the hierarchy is represented through a different table.

Managing Entities

Java Persistence API uses entity managers to manage the entities, which are instances of javax.persistence.EntityManager interface.

A persistence context implies a set of entities in the data store of the application, which is managed by the entity manager.

Container Managed Entity Managers

- All the components of an application are present within a container and the lifecycle of the entity manager is managed by the container.

Application Managed Entity Managers

- The lifecycle of the entity manager is managed by the application.
- An application managed entity manager is used when an isolated persistence context is required.
- The transaction manager has to explicitly start the transaction and define the scope of the entities while performing the entity operations.

Entity Lifecycle

There are four stages in the lifecycle of an entity instance:

- New entity instance
- Managed entity instance
- Detached entity instance
- Removed entity instance

EntityManager interface has various methods which can be used to perform operations and manipulate the state of the entities in the persistence context.

- find()
- persist()
- remove()
- flush()

Persistence Unit 1-2

- Persistence unit refers to all the entity classes in the application, which are stored in a single data store and are managed by a single EntityManager.
- The persistence unit of an application is defined by the persistence.xml configuration file.
- Following code illustrates how to define a persistence unit in a persistence.xml file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence 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_1_0.xsd"
version="1.0">

   <persistence-unit name="SamplePersistentUnit">
```

Persistence Unit 2-2

```
<description>For illustration</description>
     <provider>com.objectdb.jpa.Provider</provider>
     <mapping-file>META-INF/mappingFile.xml</mapping-file>
     <jar-file>Company.jar</jar-file>
     <class>Company.Employee</class>
     <class>Company.Department</class>
     <properties>
       <property name="javax.persistence.jdbc.url"
                 value="objectdb://localhost/my.odb"/>
       <property name="javax.persistence.jdbc.user" value="admin"/>
       <property name="javax.persistence.jdbc.password" value="admin"/>
     </properties>
   </persistence-unit>

</persistence>
```

Querying Entities

Java Persistence API has various methods as follows:

- Java Persistence Query Language (JPQL) - Java Persistence Query Language is a string based language used to query databases like SQL from the object-oriented context.
- Criteria API - Criteria API are also used to create type safe queries from a Java program.

Database Schema Creation

Java Persistence API provides various properties which can be used in the process of creating the database schema.

Following properties of the Persistence API can be used along with the XML tags:

javax.persistence.schema-generation.database.action

- This property can assume any one of the following four values – none, create, drop-and-create, drop

The properties javax.persistence.schema-generation.create-source and javax.persistence.schema-generation.drop-source are used to define how the database source is created.

- There are four options to choose from while creating and dropping the source – metadata, script, metadata-then-script, script-then-metadata

javax.persistence.sql-load-script-source is used to define the sql script to load the data from the data source.

Java Persistence Query Language 1-2

Java Persistence Query Language enables creation of SQL like queries which can be used by the application to access and manipulate a database.

Entity manager creates required queries for the application using the methods createQuery and createNamedQuery.

Given code demonstrates creation of a dynamic query using createQuery method.

```
……
public List findStudent(String name)
{
return em.createQuery(
"SELECT s FROM Student c WHERE c.name
LIKE :Name")
.setParameter("Name", name)
.setMaxResults(10)
.getResultList();
}
……………
```

Java Persistence Query Language 2-2

The createNamedQuery method is used to create static queries, which implies that the parameters for the query are not accepted dynamically at runtime.

There are two variants of parameters of the queries:

- Named parameters
- Positional parameters

```
…….
students =
em.createNamedQuery("SELECT s FROM
Student c WHERE c.name LIKE :Name")
    .setParameter("StudentName",
"Alice")
    .getResultList();
………
```

Writing SQL Queries in JPQL

Select query is the most commonly used SQL query.

Following is the format for writing a SQL query:

- `SQL_statetement ::= select_clause from`
- `[where][group by][having][orderby]`

The from clause, holds the reference to the data source objects.

The where clause, represents the condition based on which objects are expected to be retrieved from the entity classes.

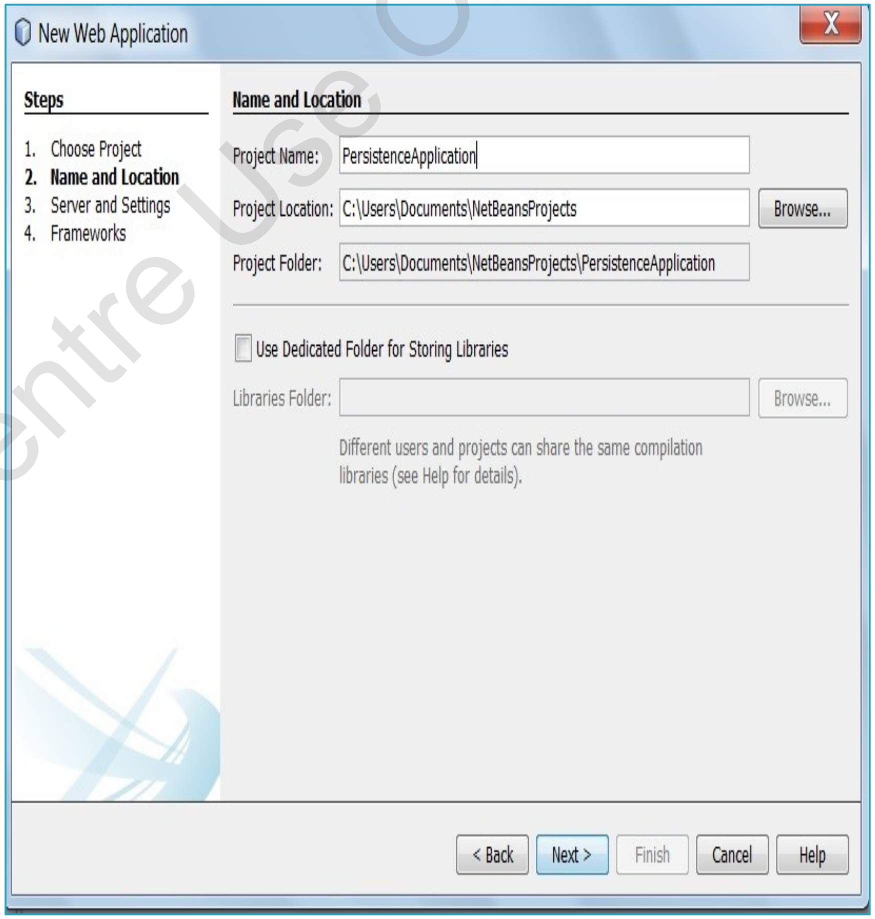
The group by clause is used to create logical groups among the data objects.

The orderby clause is used to specify some order on all the entities of the entity class that are being retrieved by the query.

Application Using Persistence API 1-6

Creating a Web Application Using JSF

- To understand the use of the Persistence API, create a Web application using JSF in the IDE by selecting File → New Project → Java Web → Web application. The New Web Application dialog box is displayed as shown in the figure.
- Specify the name of the application as PersistenceApplication and click Next.
- Select the GlassFish server and Java EE 7 version and click Next.
- Select the JavaServer Faces checkbox and click Finish.


New Web Application

Steps
1. Choose Project
2. Name and Location
3. Server and Settings
4. Frameworks

Name and Location

Project Name: PersistenceApplication

Project Location: C:\Users\Documents\NetBeansProjects

Project Folder: C:\Users\Documents\NetBeansProjects\PersistenceApplication

☐ Use Dedicated Folder for Storing Libraries

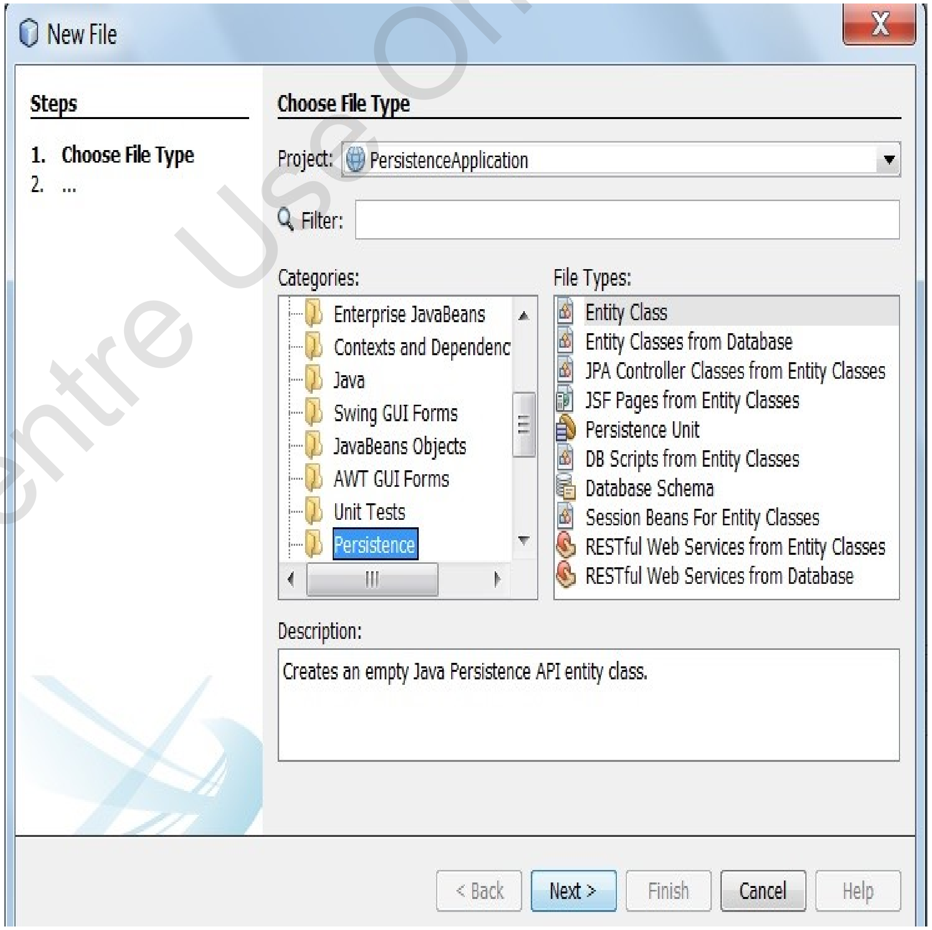
Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

Application Using Persistence API 2-6

Creating Entity Class and Persistence Unit

- Once the Web application is created, add an entity class named Message to the application to store the messages.
- To create the entity class, right-click the project and select New → Other → Persistence → Entity Class from the New File dialog box as shown in the figure.


New File

Steps
1. Choose File Type
2. ...

Choose File Type

Project: PersistenceApplication

Filter:

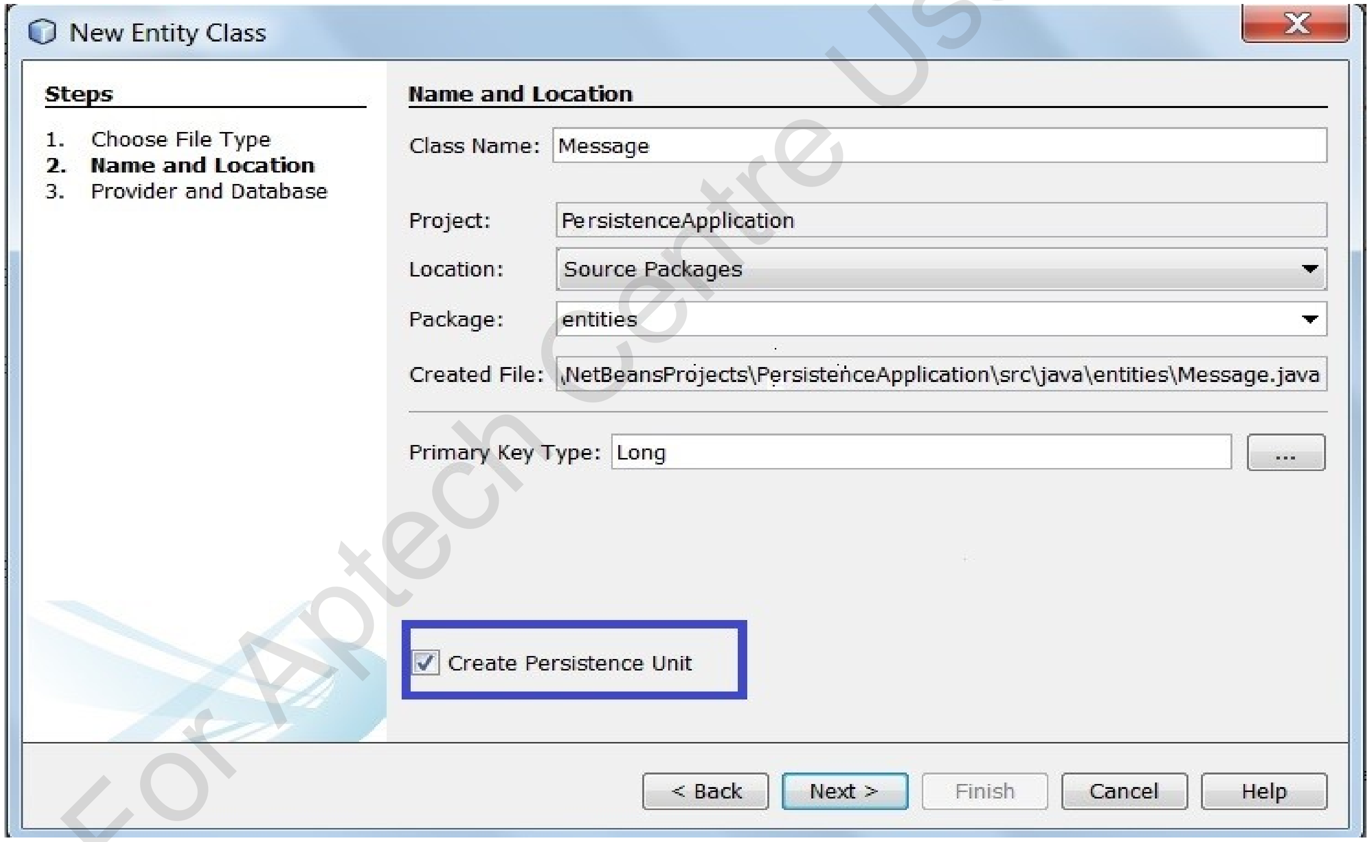
Categories: Enterprise JavaBeans, Contexts and Dependenc, Java, Swing GUI Forms, JavaBeans Objects, AWT GUI Forms, Unit Tests, Persistence

File Types: Entity Class, Entity Classes from Database, JPA Controller Classes from Entity Classes, JSF Pages from Entity Classes, Persistence Unit, DB Scripts from Entity Classes, Database Schema, Session Beans For Entity Classes, RESTful Web Services from Entity Classes, RESTful Web Services from Database

Description:
Creates an empty Java Persistence API entity class.

Application Using Persistence API 3-6

Specify the Name and Location details as shown in the following figure:



New Entity Class

Steps

1. Choose File Type
2. **Name and Location**
3. Provider and Database

Name and Location

Class Name: Message

Project: PersistenceApplication

Location: Source Packages

Package: entities

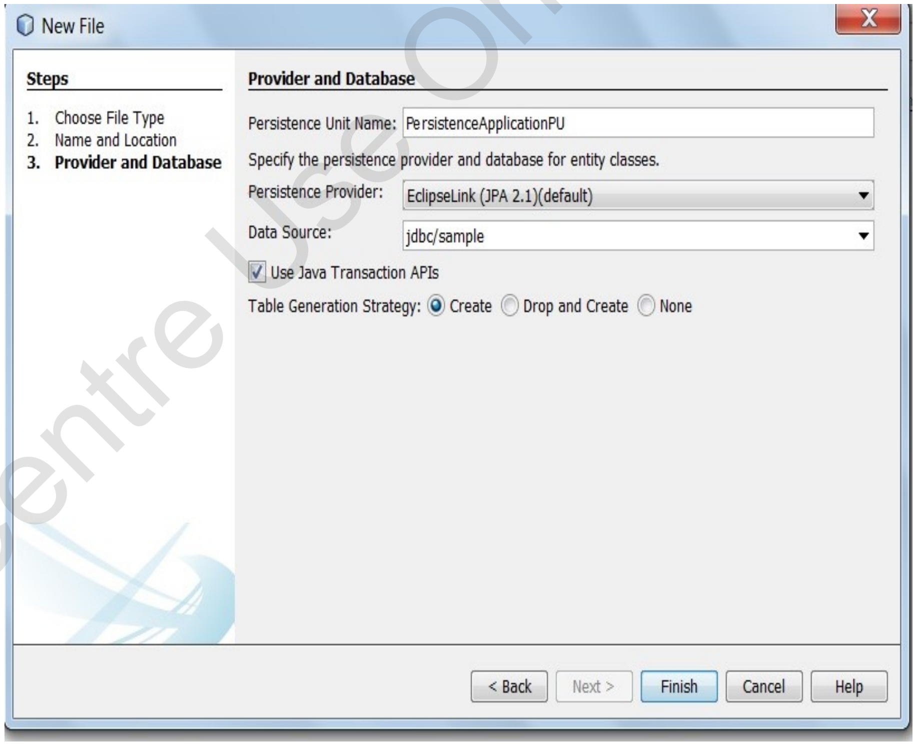
Created File: \NetBeansProjects\PersistenceApplication\src\java\entities\Message.java

Primary Key Type: Long

☑ Create Persistence Unit

Application Using Persistence API 4-6

Click Next. The Provider and Database screen is displayed. Specify the settings as shown in the figure.



Persistence Unit Name: PersistenceApplicationPU

Specify the persistence provider and database for entity classes.

Persistence Provider: EclipseLink (JPA 2.1)(default)

Data Source: jdbc/sample

[✓] Use Java Transaction APIs

Table Generation Strategy: [● Create] ○ Drop and Create ○ None

Select the Data Source as jdbc/sample from the drop-down list. The jdbc/sample database comes bundled with JavaDB in the server installation directory. Ensure that the Use Java Transaction APIs checkbox is selected and the Table Generation Strategy is set to Create.
Click Finish.

Application Using Persistence API 5-6

Following code is added to the default code created by the IDE:

```
package entities;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity
public class Message implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String message;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getMessage() {
        return message;
    }
```

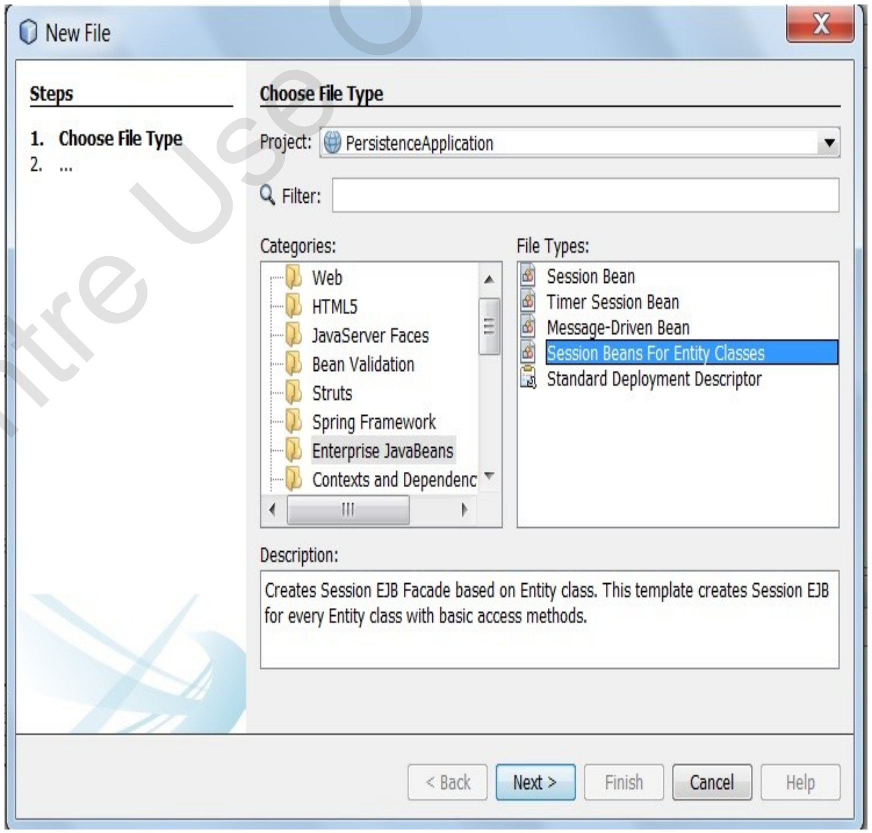
Application Using Persistence API 6-6

```
public void setMessage(String message) {
        this.message = message;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Message)) {
            return false;
        }
        Message other = (Message) 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 "entities.Message[ id=" + id + " ]";
    }}
```

Creating the Session Façade (Session Bean) 1-7

Define the access to the entity through a session bean by creating a session bean for the entity class.

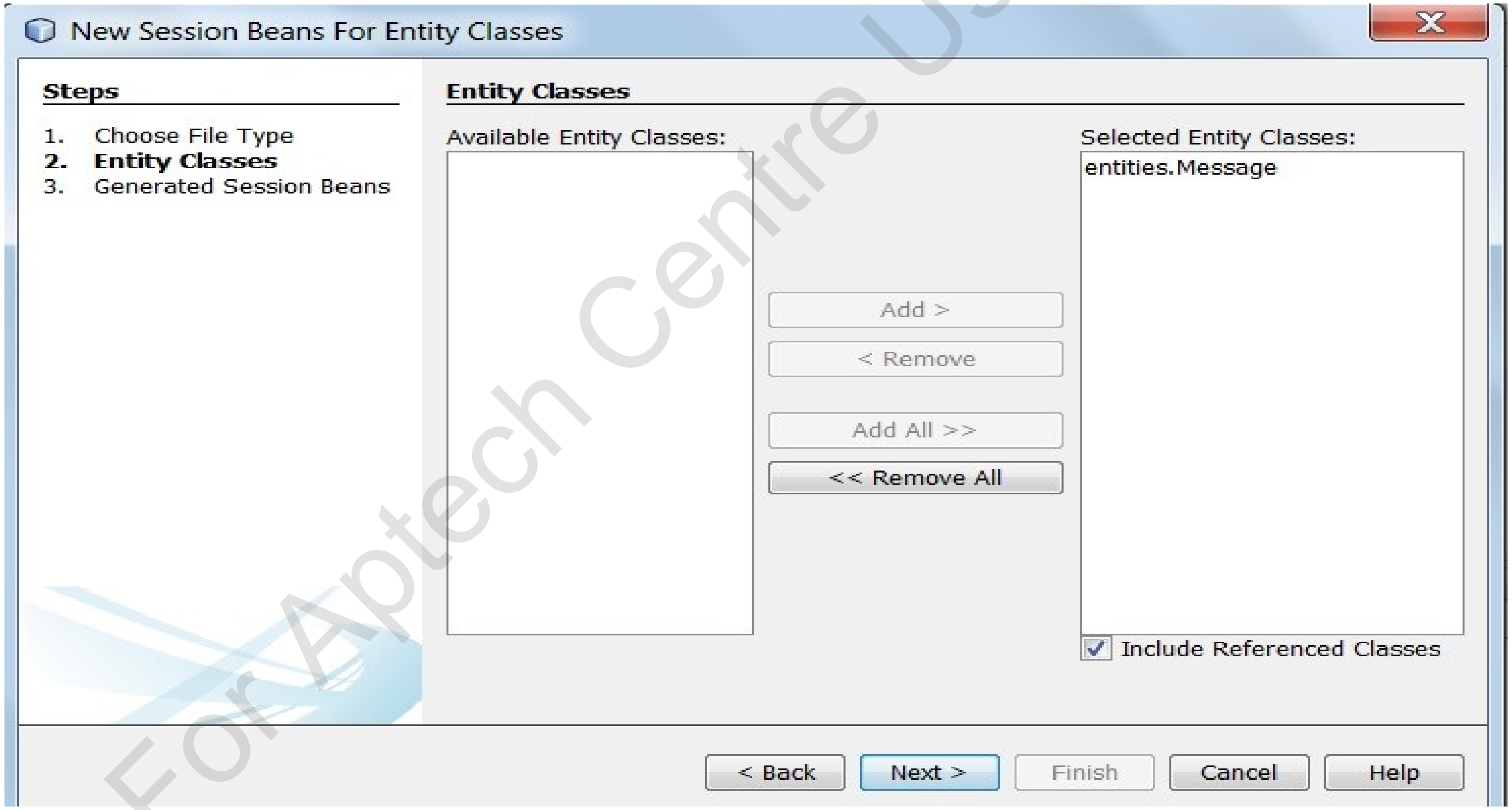
To create a session bean, right-click the project and select New → Other → Enterprise JavaBeans → Session Beans For Entity Classes as shown in the figure.



Creates Session EJB Facade based on Entity class. This template creates Session EJB for every Entity class with basic access methods.

Creating the Session Façade (Session Bean) 2-7

Click Next. Select entities.Message from the Available Entity Classes and click Add. The Message entity will be added to the Selected Entity Classes list as shown in the following figure:



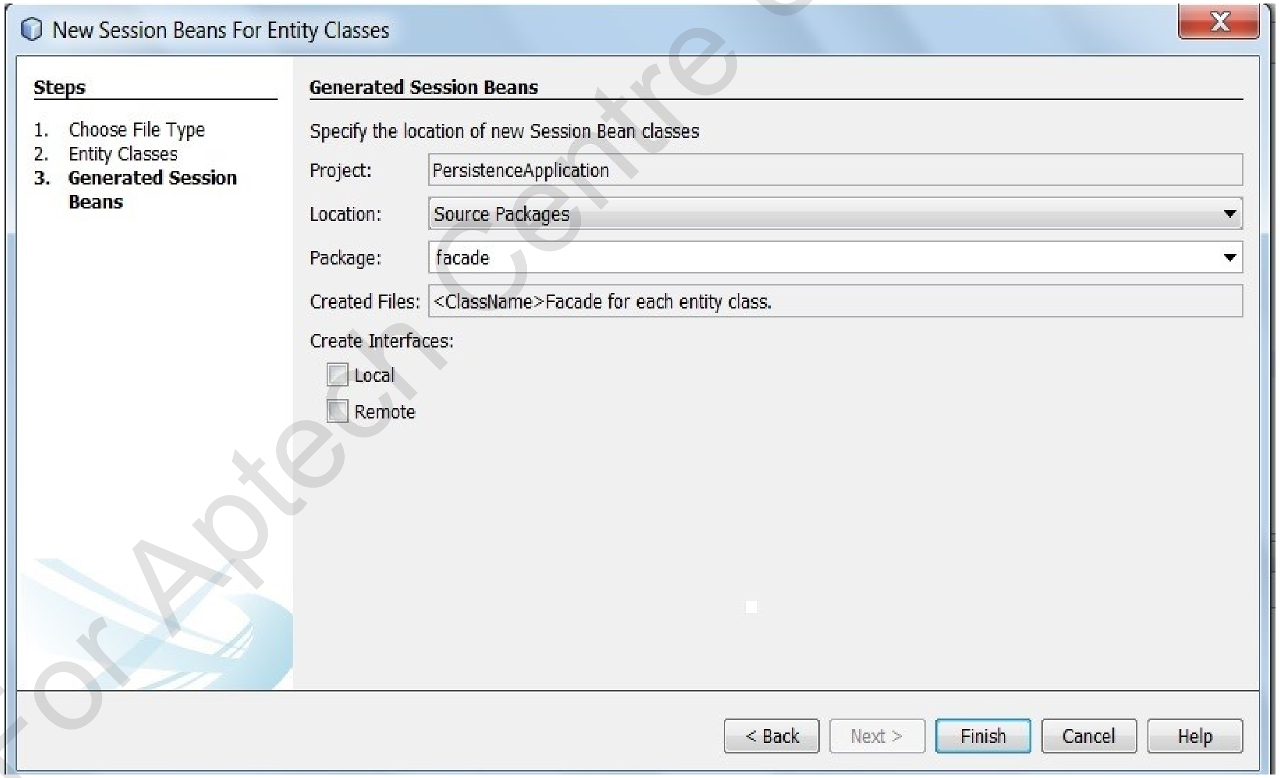
entities.Message

✓ Include Referenced Classes

Creating the Session Façade (Session Bean) 3-7

Click Next. Create the session bean in a package 'facade', this package name is based on the developer's choice.

Following figure shows how the session bean is created using the wizard:



New Session Beans For Entity Classes

Steps

1. Choose File Type
2. Entity Classes
3. **Generated Session Beans**

Generated Session Beans

Specify the location of new Session Bean classes

Project:	PersistenceApplication
Location:	Source Packages
Package:	facade
Created Files:	<ClassName>Facade for each entity class.

Create Interfaces:

- [ ] Local
- [ ] Remote

Creating the Session Façade (Session Bean) 4-7

The IDE generates the session facade class named MessageFacade.java and AbstractFacade.java.

The annotation @Stateless is used to declare a stateless session bean component.

MessageFacade.java extends AbstractFacade.java, which contains the business logic and manages the transaction.

A local managed bean is created.

Creating the Session Façade (Session Bean) 5-7

Following code demonstrates the AbstractFacade class, generated by the IDE:

```
package facade;
import java.util.List;
import javax.persistence.EntityManager;
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);
    }
```

Creating the Session Façade (Session Bean) 6-7

```
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] + 1);
        q.setFirstResult(range[0]);
        return q.getResultList();
    }
    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();
    }}
```

Creating the Session Façade (Session Bean) 7-7

Following code demonstrates the MessageFacade class:

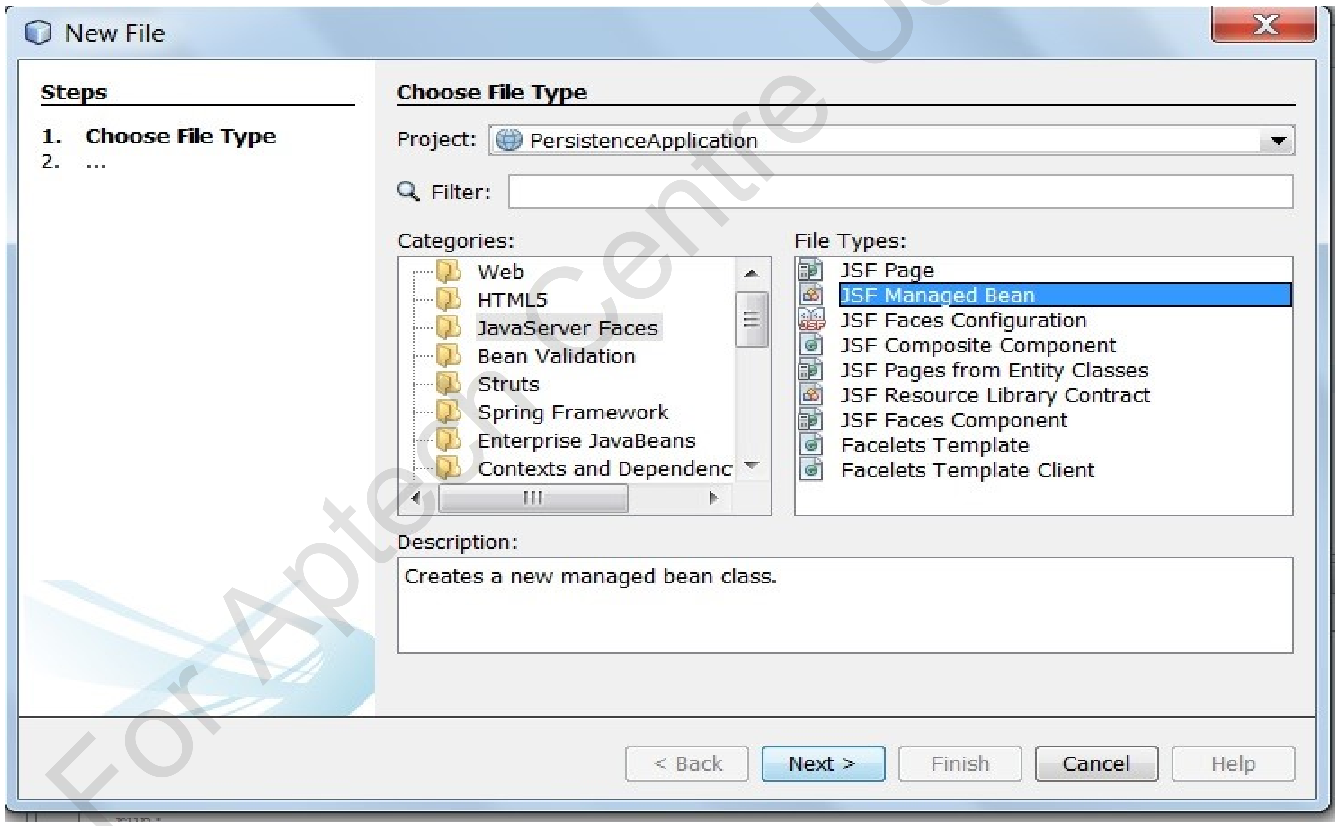
```
package facade;
import entities.Message;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
@Stateless
public class MessageFacade extends AbstractFacade<Message> {
    @PersistenceContext(unitName = "PersistenceApplicationPU")
    private EntityManager em;
    @Override
    protected EntityManager getEntityManager() {
        return em;
    }
    public MessageFacade() {
        super(Message.class);
    }

}
```

Defining the User Interface 1-6

The user interface is created using JSF.

First, create a JSF managed bean by right-clicking the project and selecting New → Other → JavaServer Faces → JSF Managed Bean as shown in the following figure:



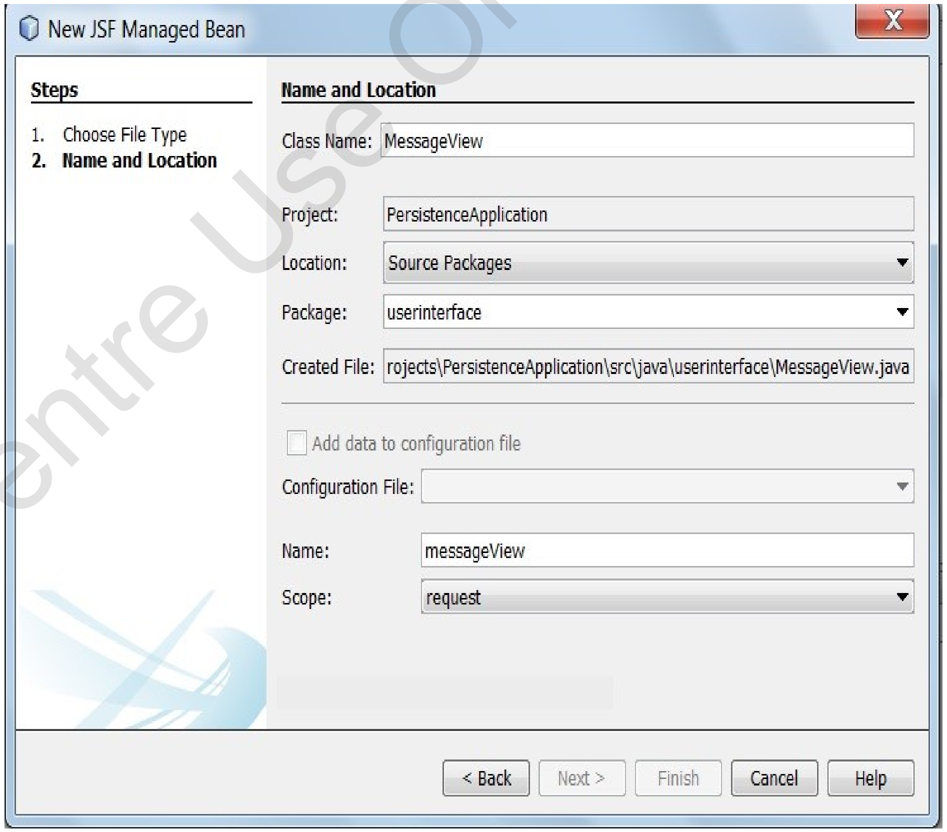
Creates a new managed bean class.

Defining the User Interface 2-6

Click Next. Specify appropriate values for the options in the wizard as shown in the figure.

Click Finish. The JSF ManagedBean is created and opened in the editor.

The IDE adds the @ManagedBean and @RequestScoped annotations and the name of the bean.



Class Name: MessageView

Project: PersistenceApplication

Location: Source Packages

Package: userinterface

Created File: rojects\PersistenceApplication\src\java\userinterface\MessageView.java

Add data to configuration file [ ]

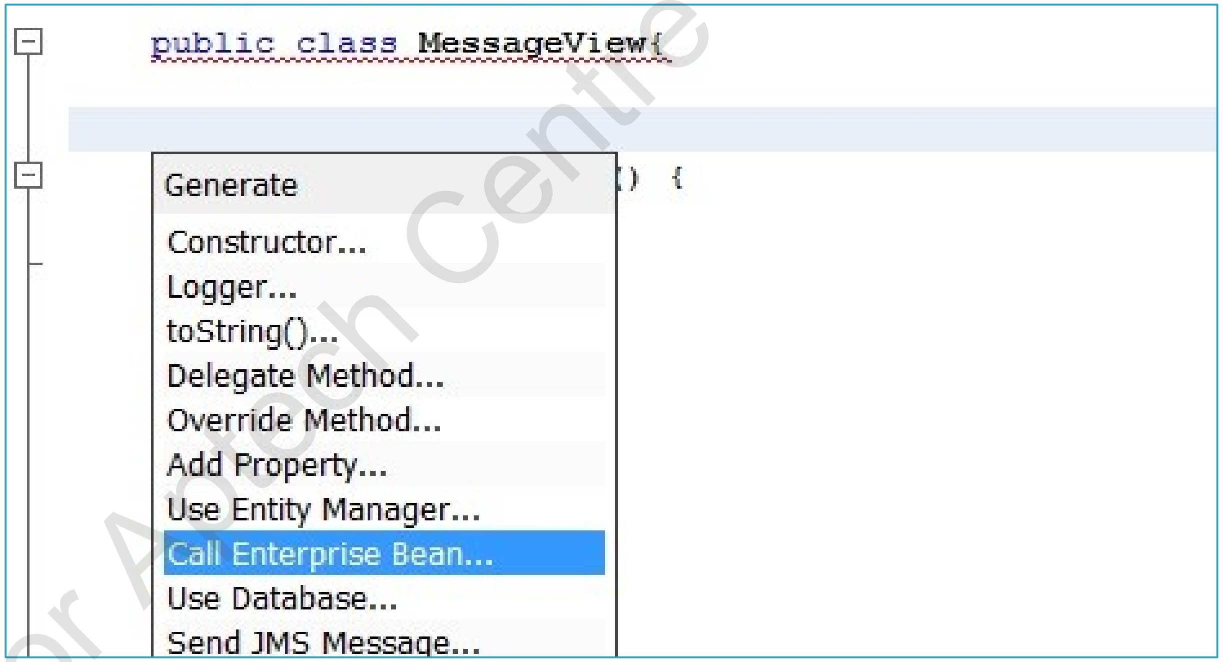
Configuration File:

Name: messageView

Scope: request

Defining the User Interface 3-6

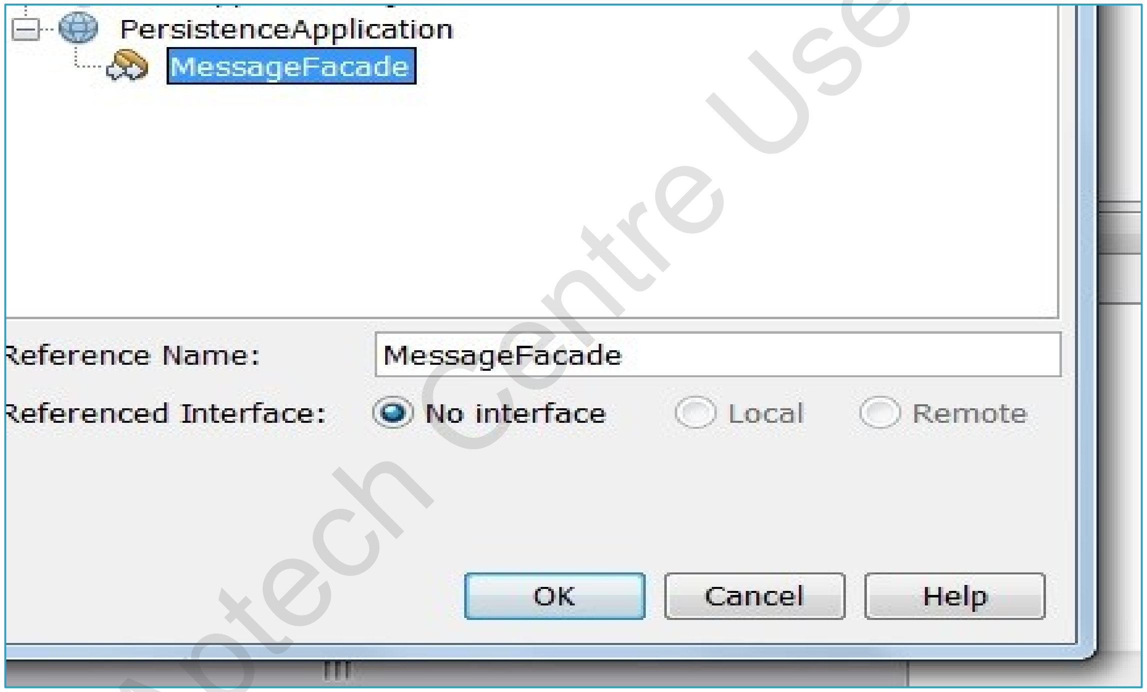
To add the session bean reference, press Alt+Insert. This provides options of code that can be added in the MessageView class. Select 'Call Enterprise Bean' option in the menu as shown in the following figure:



```
public class MessageView{
```

Defining the User Interface 4-6

Select the MessageFacade bean as shown in the following figure:



Reference Name: MessageFacade

Referenced Interface: [✓ No interface] Local Remote

Click OK.

Defining the User Interface 5-6

Following code demonstrates adding a reference to the Session bean to the MessageView class using the @EJB annotation:

```
public class MessageView {
    @EJB
    private MessageFacade messageFacade;
    /**
     * Creates a new instance of MessageView
     */
    public MessageView() {
    }

}
```

Defining the User Interface 6-6

Given code modifes the class by adding the message variable, initializing it in the constructor, getter method, and adding the getNumberOfMessages and postMessage methods.

```
public class MessageView {
    @EJB
    private MessageFacade messageFacade;
// Creates a new field
    private Message message;
    /**
     * Creates a new instance of MessageView
     */
    public MessageView() {
        this.message = new Message();
    }
     // Calls getMessage to retrieve the
message
    public Message getMessage() {
       return message;
    }
    // Returns the total number of messages
    public int getNumberOfMessages(){
       return messageFacade.findAll().size();
    }
    // Saves the message and then returns the
string "theend"
    public String postMessage(){
       this.messageFacade.create(message);
       return "response";
    }
```

Creating the Index Page

Modify the index page of the application as demonstrated in the following code to add the user interface components:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Persistence Application</title>
    </h:head>
    <h:body>
        <h:form>
            <h:outputLabel value="Message:"/><h:inputText
value="#{messageView.message.message}"/>
        <h:commandButton action= "#{messageView.postMessage()}" value="Post
Message"/>
        </h:form>
    </h:body>
</html>
```

Creating the Response Page

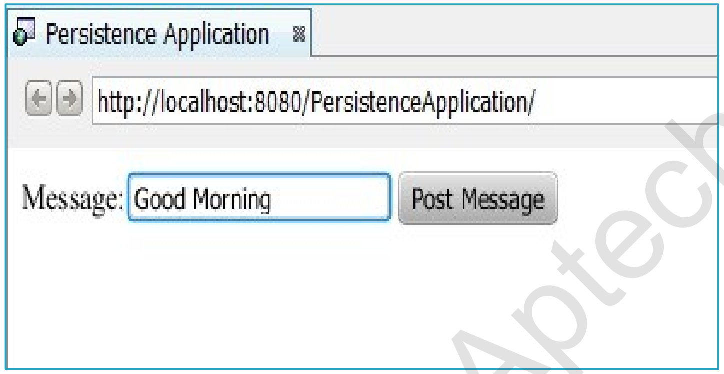
The following code demonstrates adding code to a new JSF page with the name, response.xhtml:

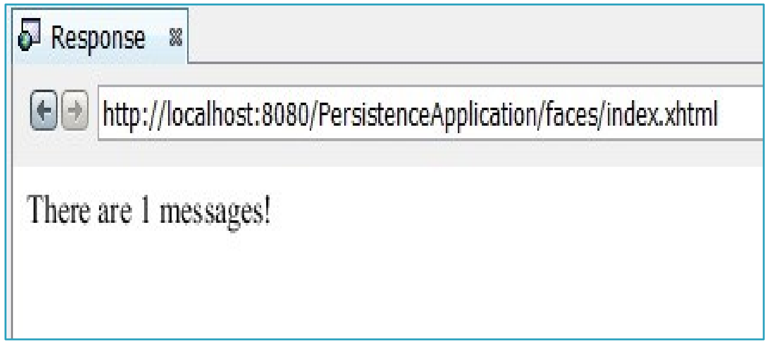
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Response</title>
    </h:head>
    <h:body>
     <h:outputLabel value="There are "/>
     <h:outputText value="#{messageView.numberOfMessages}"/>
    <h:outputLabel value=" messages!"/>
    </h:body>
</html>
```

Application Execution

On executing the application, the index.xhtml page is displayed. Type the message 'Good Morning' in the textbox as shown in the following figure:

Click the Post Message button. The response page is displayed as shown in the following figure:




Response
http://localhost:8080/PersistenceApplication/faces/index.xhtml

There are 1 messages!

Summary

- Java Persistence API performs the object-relational mapping of data, where the database stores the data in the form of tables and the application handles the data in the form of objects.
- The entities are modelled as entity classes by the JPA.
- The entity classes can accept the data in both static and dynamic forms through entity properties and fields.
- The entities in the application can be associated with each other in one-to-one, one-to-many, many-to-one, and many-to-many relationships.
- The entity inheritance in the object-oriented mode can be mapped to relations through single table per class hierarchy, a table per concrete entity class, or through join strategy of the relations.
- JPQL is used to implement the SQL operations on the database.