Chapter 13
How to use JPA to work with a database

Objectives

Applied
1. Develop data access classes that use JPA to provide all of the methods that your servlets need to work with a database.
2. Develop servlets that use the methods of your data access classes.

Objectives (continued)

Knowledge
1. Name three implementations of JPA.
2. Describe O/R (object-relational) mapping.
3. Describe the purpose of the persistence.xml file.
4. Describe the purpose of JPA annotations.
5. Distinguish between field annotations and getter annotations.
6. Describe how a web application can use the Persistence, EntityManagerFactory, and EntityManager classes to work with a database.
7. Describe how transactions work.

JPA...
- Is an object-relational mapping specification.
- Makes it easier to map objects to rows in a relational database.
- Shields the developer from having to write JDBC and SQL code.
- Runs on top of JDBC.
- Is compatible with any database that has a JDBC driver.

Three popular JPA implementations
- Hibernate
- EclipseLink
- TopLink

Entities and the entity manager
- Business classes intended to be used with JPA are called entities.
- You can convert a business class to an entity by adding JPA annotations to the class.
- Entities are managed by an entity manager.
- Full Java EE application servers such as Glassfish have a built-in entity manager that includes advanced features such as automatic transaction management.
- If you want to use JPA outside of a full Java EE application server, such as in Tomcat or a desktop application, you can create your own entity managers.

The persistence.xml file

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <persistence-unit name="emailListPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/murach_jpa"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value="sesame"/>
      <property name="javax.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```
A summary of the persistence.xml elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>persistence-unit</td>
<td>The name attribute specifies the name you use in your code to get a reference to the database. The transaction-type attribute specifies how the application works with entity managers. RESOURCE_LOCAL specifies that you will create and manage the entity managers yourself. This is necessary if you’re using Tomcat.</td>
</tr>
<tr>
<td>provider</td>
<td>Specifies the full class name of the JPA PersistenceProvider class.</td>
</tr>
<tr>
<td>exclude-unlisted-classes</td>
<td>A false value specifies that JPA uses all classes annotated as entities. Otherwise, you have to list each class you want JPA to use as an entity</td>
</tr>
<tr>
<td>shared-cache-mode</td>
<td>Determines the caching strategy used by JPA. Caching can improve performance. This is covered later in this chapter.</td>
</tr>
</tbody>
</table>

The class for a JPA entity

- The @Entity annotation specifies that this class is a managed bean that’s part of a persistence unit.
- The @Id annotation specifies which field in the class is the primary key.
- The @GeneratedValue annotation specifies how the primary key should be generated.
- To override the default table name, code the @Table annotation on the line immediately following the @Entity annotation.
- By default, JPA uses the same names for the columns in the database as the names of the fields in the class.
- If you want to override the default column name, you can code the @Column annotation immediately above the field.

How to code field annotations

```java
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long userId;

public Long getUserId() {
    return userId;
}

public void setUserId(Long userId) {
    this.userId = userId;
}
```

We’ll use field annotations, next slide.

How to code getter annotations

```java
private Long userId;

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
public Long getUserId() {
    return userId;
}

public void setUserId(Long userId) {
    this.userId = userId;
}
```

We’ll use field annotations, next slide.

Getter and field annotations

- Getter annotations use the get and set methods of the class to access the fields.
- Field annotations use reflection to access the fields in your class directly, even if they are declared as private. It does not call the get and set methods. As a result, any code in your get and set methods does not run when JPA accesses the fields.
- You cannot mix field and getter annotations in the same class.
A JPA entity with relationships

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Invoice implements Serializable {

    private Long invoiceNumber;
    private User user;
    @OneToMany(fetch=FetchType.EAGER, cascade=CascadeType.ALL)
    private List<LineItem> lineItems;

    // getters and setters for fields
}

Relationships in a JPA entity

- The @ManyToMany annotation specifies that many invoices can belong to one user.
- The @OneToMany annotation specifies that an invoice can have many line items.

Two elements of the @OneToMany annotation

<table>
<thead>
<tr>
<th>Element</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch</td>
<td>FetchType.EAGER specifies that all of the line items for the invoice should be loaded when the invoice is loaded from the database.</td>
</tr>
<tr>
<td>cascade</td>
<td>CascadeType.ALL specifies that all operations that change the invoice should also update all of the line items. CascadeType.PERSIST specifies that any time a new invoice is inserted into the database, any line items it has should also be inserted. CascadeType.REMOVE specifies that any time an invoice is removed from the database, all of its line items should also be removed.</td>
</tr>
</tbody>
</table>

A static method of the Persistence class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createEntityManagerFactory()</td>
<td>Returns an EntityManagerFactory object for the specified persistence unit. The persistence unit name must match the unit name defined in the persistence.xml file.</td>
</tr>
</tbody>
</table>

A utility class that gets an entity manager factory

import java.persistence.EntityManagerFactory;
import java.persistence.Persistence;

public class DBUtil {
    // code

    @Override
    public static User getUserById(long userId) {
        // code

        return user;
    }

    // code
}

How to retrieve an entity by primary key

import java.persistence.EntityManager;
import murach.business.User;

How to retrieve an entity by primary key

public class UserDB {
    public static User getUserById(long userId) {
        // code
    }

    // code
}

How to retrieve an entity by primary key

public class DBUtil {
    public static User getUserById(long userId) {
        // code
    }

    // code
}

Description

- Entity managers are not thread-safe, so you need to create local entity managers for each method that needs one. It's true that entity managers are not thread-safe, so we'll arrange that each thread has its own entity manager, created when that thread's transaction starts. It's not true that each method needs this em creation, especially if they look like DAO methods like this one above.

Description

- Entity managers are not thread-safe, so you need to create local entity managers for each method that needs one. It's true that entity managers are not thread-safe, so we'll arrange that each thread has its own entity manager, created when that thread's transaction starts. It's not true that each method needs this em creation, especially if they look like DAO methods like this one above.
**Method of the EntityManagerFactory class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createEntityManager()</td>
<td>Returns an EntityManager object</td>
</tr>
</tbody>
</table>

**Two methods of the EntityManager class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find(entityClass, primaryKey)</td>
<td>Returns an object of the specified entity class that has the specified primary key. If the specified primary key doesn’t exist, this method returns a null value. Closes the entity manager object and releases its resources. This prevents resource leaks.</td>
</tr>
</tbody>
</table>

**A JPQL statement that selects one field**

\[
\text{SELECT i.invoiceDate FROM Invoice i WHERE i.isProcessed = 'n'}
\]

**Two methods of the EntityManager class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createQuery(queryString)</td>
<td>Returns a Query object that returns the result as an Object or list of Object types. This version is not type safe.</td>
</tr>
<tr>
<td>createQuery(queryString, resultClass)</td>
<td>Returns a TypedQuery object that returns the result as an object of the specified result class or a list of objects of the specified result class.</td>
</tr>
</tbody>
</table>

**How to retrieve multiple entities**

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.Query;

public class InvoiceDB {
    public static List<Invoice> selectUnprocessedInvoices() {
        EntityManager em = DBUtil.getEmFactory().createEntityManager();
        try {
            TypedQuery<Invoice> q = em.createQuery("SELECT i.invoiceDate FROM Invoice i WHERE i.isProcessed = 'n'");
            List<Invoice> invoices = q.getResultList();
            return invoices;
        } finally {
            em.close();
        }
    }
}

**JPQL and the getResultList method**

- **JPQL (Java Persistence Query Language)** is an object-oriented query language defined as part of the JPA specification. It works similarly to SQL.
- JPQL uses path expressions to refer to the fields of an entity. These expressions don’t refer to the columns of a table.
- The `getResultList` method may automatically perform joins or additional queries to satisfy the relationships between entities.

For example, public class `PizzaOrders` findOrdersByRoom(…) in pizza2 uses "select o from PizzaOrder o where o.roomNumber = '2' + roomNumber + '2' and o.day = '3 + day + ' order by o.id" and returns PizzaOrders with PizzaToppings and a PizzaSize hanging off of them.

**Some exceptions thrown by getSingleResult**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoResultException</td>
<td>The query returned no results.</td>
</tr>
<tr>
<td>NonUniqueResultException</td>
<td>The query returned more than one result.</td>
</tr>
</tbody>
</table>
Named parameters and the setParameter method

- To specify a named parameter in a query string, code a colon (:) followed by the name of the parameter.
- To set a parameter, code the setParameter method and specify the name of the parameter as the first argument and the value of the parameter as the second argument.

How to wrap an operation in a transaction

```java
EntityTransaction trans = em.getTransaction();
try {
    trans.begin();
    em.persist(user);
    trans.commit();
} catch (Exception ex) {
    trans.rollback();
} finally {
    em.close();
}
```

How to insert a single entity

```java
em.persist(user);
```

How to update a single entity

```java
em.merge(user);
```

How to delete a single entity

```java
em.remove(em.merge(user));
```

Methods of the EntityManager object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>persist(entity)</td>
<td>Inserts an entity into the database.</td>
</tr>
<tr>
<td>merge(entity)</td>
<td>Updates an entity in the database and returns an attached entity.</td>
</tr>
<tr>
<td>remove(entity)</td>
<td>Deletes an entity from the database.</td>
</tr>
<tr>
<td>flush()</td>
<td>Forces any unsaved changes to synchronize to the database.</td>
</tr>
</tbody>
</table>

Merge is tricky to use correctly, so we will avoid it. No extra call to the em is needed to update an entity already known to the em (i.e. a managed entity) — it will happen automatically at commit.

Transactions

- If you aren’t using a Java EE server, code database operations within a transaction. If the transaction is successful, commit the changes to the database. If the transaction isn’t successful, rollback any changes. This ensures data integrity.
- JPA may flush unsaved changes before you finish a transaction. However, if the rollback method of that transaction is called, JPA can still rollback those changes.
- A transaction can be rolled back any time before the commit method is called, or if the commit method is called but fails.

We have a method rollbackAfterException to handle the usual case of a DB problem. It’s a little tricky because rollback itself can throw.
How to update multiple entities

EntityTransaction trans = em.getTransaction();
Query q = em.createQuery("UPDATE Invoice i SET i.isProcessed = 'y' WHERE i.id < :id");
q.setParameter(id, 200);
int count = 0;
try {
    trans.begin();
    count = q.executeUpdate();
    trans.commit();
} catch (Exception ex) {
    trans.rollback();
} finally {
    em.close();
}

Alternatively, use a JPA query to get the Invoice objects, set isProcessed via the setter for each, and commit.

There was discussion in class about performance of these approaches. JPA usually does all the updates together at commit, and may use JDBC batching for this. There are no JPA-defined settings for batching, but Hibernate and EclipseLink both have settings. See this post

The UserDB class (continued)

public static void update(User user) {
    EntityManager em = DBUtil.getEmFactory().createEntityManager();
    try {
        em.merge(user);
        trans.commit();
    } catch (Exception ex) {
        trans.rollback();
    } finally {
        em.close();
    }
}

This takes a detached object and forces it into the database using merge.
We would do:
Find it before doing the update, so it's managed
Do the update in the same transaction
Then em.commit() in the service layer.

The UserDB class (continued)

public static void delete(User user) {
    EntityManager em = DBUtil.getEmFactory().createEntityManager();
    try {
        em.remove(em.merge(user));
        trans.commit();
    } catch (Exception ex) {
        trans.rollback();
    } finally {
        em.close();
    }
}

Alternatively, use a JPA query to get the Invoice objects, call em.remove(o) for each, and commit.

How to delete multiple entities

EntityTransaction trans = em.getTransaction();
Query q = em.createQuery("DELETE FROM Invoice i WHERE i.id < :id");
q.setParameter(id, 200);
int count = 0;
try {
    trans.begin();
    count = q.executeUpdate();
    trans.commit();
} catch (Exception ex) {
    trans.rollback();
} finally {
    em.close();
}

Alternatively, use a JPA query to get the Invoice objects, call em.remove(o) for each, and commit.

The executeUpdate method

- The executeUpdate method returns a count of the number of entities affected by the query.
- These queries may trigger additional automatic updates or deletions. For example, deleting an invoice will automatically delete all of its line items.

In Murach’s setup, the entity Invoice is setup to manage its line items using cascade on the relationship: In Invoice.java:

```java
@OneToMany(fetch=FetchType.EAGER, cascade=CascadeType.ALL)
private List<LineItem> lineItems;
```

Alternatively, without cascade, we would separately delete the line items.

We would start and end the transaction in the service layer.

The executeUpdate method returns a count of the number of entities affected by the query. These queries may trigger additional automatic updates or deletions. For example, deleting an invoice will automatically delete all of its line items.
The UserDB class (continued)

```java
public static User selectUser(String email) {
    EntityManager em = DBUtil.getEmFactory().createEntityManager();
    String qString = "SELECT u FROM User u " +
            "WHERE u.email = :email";
    TypedQuery<User> q = em.createQuery(qString, User.class);
    q.setParameter("email", email);
    User user = q.getSingleResult();
    return user;
}
```

```java
public static boolean emailExists(String email) {
    User u = selectUser(email);
    return u != null;
}
```