Chapter 13

How to use JPA
to work with a database

Objectives

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 (domain 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 (feature of EJBs).
• 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. (We’ll do this.)

The persistence.xml file

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <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>

Similar to pizza2/database/mysql/persistence.xml
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 simple JPA entity

```java
import java.io.Serializable;
import java.sql.ResultSet;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import java.util.List;
import java.util.Date;

@Entity
public class User implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long userId;
    private String firstName;
    private String lastName;
    private String email;
    // the rest of the get and set methods for the fields
}
```

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 EntityManager yourself. This is necessary if you're using Tomcat. Specifies the full class name of the JPA PersistenceProvider class.</td>
</tr>
<tr>
<td>provider</td>
<td></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>
<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 EntityManager yourself. This is necessary if you're using Tomcat. Specifies the full class name of the JPA PersistenceProvider class.</td>
</tr>
<tr>
<td>provider</td>
<td></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;
}
```

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 java.util.List;

@Entity
public class Invoice implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long invoiceNumber;
    private boolean isProcessed;
    @OneToMany(fetch = FetchType.LAZY)
    private List<LineItem> lineItems;
    private User user;
    @Temporal(javax.persistence.TemporalType.DATE)
    private Date invoiceDate;
    private static final EntityManagerFactory emf = Persistence.createEntityManagerFactory("emailListPU");

    public static EntityManagerFactory getEmFactory() {
        return emf;
    }

    public static User getUserById(long userId) {
        EntityManager em = DBUtil.getEmFactory().createEntityManager();
        User user = em.find(User.class, userId);
        em.close();
        return user;
    }
}

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.LAZY specifies that all of the line items for the invoice should be loaded when the invoice is loaded from the database. FetchType.LAZY requests, but does not guarantee, that line items for the invoice only be loaded when the application actually accesses them.</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.ALL is true when a new invoice is inserted into the database, any line items it has should also be inserted. CascadeType.MERGE specifies that any time an invoice is updated, any changes to its line items should also be updated. CascadeType.PERSIST specifies that any time a new invoice is created, all of its line items should also be created.</td>
</tr>
</tbody>
</table>

A utility class that gets an entity manager factory

import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class DBUtil {
    public static EntityManagerFactory getEmFactory() {
        return Persistence.createEntityManagerFactory("emailListPU");
    }
}

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>

How to retrieve an entity by primary key

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

public class UserDB {
    public static User getUserById(long userId) {
        EntityManager em = DBUtil.getEmFactory().createEntityManager();
        User user = em.find(User.class, userId);
        em.close();
        return user;
    }
}

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.
### How to retrieve multiple entities

**import javax.persistence.EntityManager;**

```java
public class InvoiceDB {  
    public static List<Invoice> selectUnprocessedInvoices() {  
        EntityManager em = DBUtil.getEmFactory().createEntityManager();  
        String qString = "SELECT i.invoiceDate FROM Invoice i WHERE i.isProcessed = 'n'";  
        TypedQuery<Invoice> q = em.createQuery(qString, Invoice.class);  
        return q.getResultList();  
    }  
}
```

### How to retrieve a single entity

**import murach.business.*;**

```java
import murach.business.Invoice;  
import murach.business.User;  
import javax.persistence.EntityManager;  
import javax.persistence.TypedQuery;  
import java.util.List;  
import javax.persistence.*;  

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

### 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.

- **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 List<PizzaOrder> findOrdersByRoom(…) in pizza2` uses `"select o from PizzaOrder o where o.roomNumber = " + roomNumber + " and o.day = " + 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();
}
```

Note that trans.commit() and trans.rollback() can both throw, and if rollback throws here, this code throws to its caller. But if an ordinary DB problem occurs, the transaction is just quietly rolled back, with no notification to the caller. So this code (pg. 445) needs work.

**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>Force 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.