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.

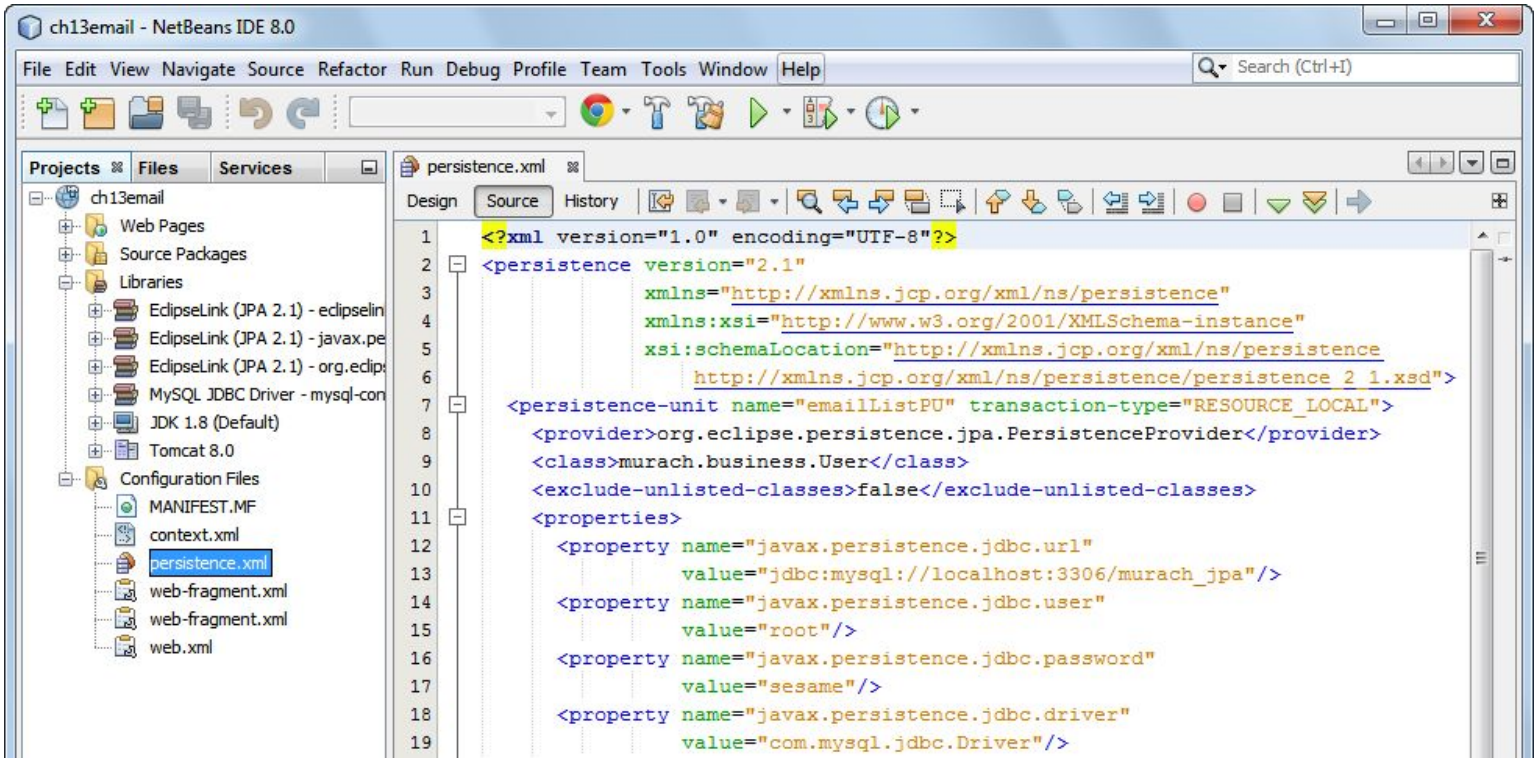
Three popular JPA implementations

- 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.

NetBeans with a JDBC driver, JPA library, and persistence.xml file



How to add a JDBC driver and the JPA library

- To add the MSQL JDBC driver to a project, right-click on its Libraries folder, select Add Library, and select the library.
- To add a JPA library to a project, right-click on its Libraries folder, select Add Library, and select the library.

How to work with the persistence unit

- To add a persistence.xml file to a project, select File→New File. Then, select the Persistence category, Persistence Unit file type, and configure the persistence unit.
- To view the source code for a persistence unit, expand the Configuration Files folder, double-click the persistence.xml file, and click Source.

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>
```

A summary of the persistence.xml elements

Element	Description
`persistence-unit`	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.
`provider`	Specifies the full class name of the JPA PersistenceProvider class.
`exclude-unlisted-classes`	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.
`shared-cache-mode`	Determines the caching strategy used by JPA. Caching can improve performance. This is covered later in this chapter.

A simple JPA entity

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class User implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long userId;
    private String firstName;
    private String lastName;
    private String email;

    public Long getUserId() {
        return userId;
    }
    public void setUserId(Long userId) {
        this.userId = userId;
    }
    // the rest of the get and set methods for the fields
}
```

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 getter annotations

```
private Long userId;

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

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

How to code field annotations

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

public Long getUserId() {
    return userId;
}

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

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 {

    @ManyToOne
    private User user;

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

    @Temporal(javax.persistence.TemporalType.DATE)
    private Date invoiceDate;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long invoiceNumber;

    private boolean isProcessed;

    // getters and setters for fields
}
```

Two elements of the @OneToMany annotation

Element	Values
`fetch`	FetchType.EAGER 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.
`cascade`	CascadeType.ALL specifies that all operations that change the invoice should also update all of the line items. CascardType.PERSIST specifies that any time 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.REMOVE specifies that any time an invoice is removed from the database, all of its line items should also be removed.

Relationships in a JPA entity

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

The annotation for a temporal type

```
@Temporal(TemporalType.TIMESTAMP)
private Date invoiceDate;
```

SQL mappings for temporal annotations

Annotation value	SQL type
`TemporalType.DATE`	`java.sql.Date`
`TemporalType.TIME`	`java.sql.Time`
`TemporalType.TIMESTAMP`	`java.sql.TimeStamp`

Description

- The @Temporal annotation specifies the SQL type for the java.util.Date and java.util.Calendar types.

A utility class that gets an entity manager factory

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

public class DBUtil {
    private static final EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("emailListPU");

    public static EntityManagerFactory getEmFactory() {
        return emf;
    }
}
```

Description

- Use a utility class to make it easy to get an EntityManagerFactory object for the specified persistence unit.

A static method of the Persistence class

Method	Description
`createEntityManagerFactory()`	Returns an EntityManagerFactory object for the specified persistence unit. The persistence unit name must match the unit name defined in the persistence.xml file.

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();
        try {
            User user = em.find(User.class, userId);
            return user;
        } finally {
            em.close();
        }
    }
}
```

Description

- Entity managers are not thread-safe, so you need to create local entity managers for each method that needs one.

Method of the EntityManagerFactory class

Method	Description
`createEntityManager()`	Returns an EntityManager object.

Two methods of the EntityManager class

Method	Description
`find(entityClass, primaryKey)`	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.
`close()`	Closes the entity manager object and releases its resources. This prevents resource leaks.

How to retrieve multiple entities

```
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;

import music.business.Invoice;

public class InvoiceDB {
    public static List<Invoice> selectUnprocessedInvoices() {
        EntityManager em = DBUtil.getEmFactory().createEntityManager();
        String qString = "SELECT i from Invoice i" +
                         "WHERE i.isProcessed = 'n'";
        TypedQuery<Invoice> q = em.createQuery(qString, Invoice.class);

        List<Invoice> invoices;
        try {
            invoices = q.getResultList();
            if (invoices == null || invoices.isEmpty())
                invoices = null;
        } finally {
            em.close();
        }
        return invoices;
    }
}
```

A JPQL statement that selects one field

```
SELECT i.invoiceDate FROM Invoice i WHERE i.isProcessed = 'n'
```

Two methods of the EntityManager class

Method	Description
`createQuery(queryString)`	Returns a Query object that returns the result as an Object or list of Object types. This version is not type safe.
`createQuery(queryString, resultClass)`	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.

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.

How to retrieve a single entity

```
import javax.persistence.*;

import murach.business.User;

public class UserDB {
    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 = null;
        try {
            user = q.getSingleResult();
        } catch (NoResultException e) {
            System.out.println(e);
        } finally {
            em.close();
        }
        return user;
    }
}
```

Some exceptions thrown by getSingleResult

Exception	Description
`NoResultException`	The query returned no results.
`NonUniqueResultException`	The query returned more than one result.

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

```
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

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

How to update a single entity

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

How to delete a single entity

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

Methods of the EntityManager object

Method	Description
`persist(entity)`	Inserts an entity into the database.
`merge(entity)`	Updates an entity in the database and returns an attached entity.
`remove(entity)`	Deletes an entity from the database.
`flush()`	Force any unsaved changes to synchronize to the database.

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, *roll back* 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 roll back 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.

How to update multiple entities

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

How to delete multiple entities

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

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.

The UserDB class

```
package murach.data;

import javax.persistence.*;

import murach.business.User;

public class UserDB {

    public static void insert(User user) {
        EntityManager em = DBUtil.getEmFactory().createEntityManager();
        EntityTransaction trans = em.getTransaction();
        trans.begin();
        try {
            em.persist(user);
            trans.commit();
        } catch (Exception e) {
            System.out.println(e);
            trans.rollback();
        } finally {
            em.close();
        }
    }
```

The UserDB class (continued)

```
public static void update(User user) {
    EntityManager em = DBUtil.getEmFactory().createEntityManager();
    EntityTransaction trans = em.getTransaction();
    trans.begin();
    try {
        em.merge(user);
        trans.commit();
    } catch (Exception e) {
        System.out.println(e);
        trans.rollback();
    } finally {
        em.close();
    }
}
```

The UserDB class (continued)

```
public static void delete(User user) {
        EntityManager em = DBUtil.getEmFactory().createEntityManager();
        EntityTransaction trans = em.getTransaction();
        trans.begin();
        try {
            em.remove(em.merge(user));
            trans.commit();
        } catch (Exception e) {
            System.out.println(e);
            trans.rollback();
        } finally {
            em.close();
        }
    }
```

The UserDB class (continued)

```
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);
        try {
            User user = q.getSingleResult();
            return user;
        } catch (NoResultException e) {
            return null;
        } finally {
            em.close();
        }
    }

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