More on JPA: rest of Murach
Chapter 13, pizza2

From last time: A simple JPA entity

```java
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
}
```

We'll use `GenerationType.TABLE` for the database to use.

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</td>
<td>Inserts an entity into the database.</td>
</tr>
<tr>
<td>merge</td>
<td>Updates an entity in the database and returns an attached entity.</td>
</tr>
<tr>
<td>remove</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.
Entity life cycles

- As in pizza1/music1, a domain object can be born in the DAO layer and be returned to the calling service method.
- Under JPA, these objects are “managed”, that is, actively tracked by the runtime of JPA, until commit time. Then they are “detached”, no longer managed.
- Also as in pizza1/music1, a domain object can be born in the service layer, a new object, unknown to JPA. It becomes “managed” when we use em.persist(object) to save it to the database, and detached 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, roll back any changes. This ensures data integrity.
- JPA may flush (save to DB) 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.

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();
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();
}

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.

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();
}

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 set up to manage its lineitems 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 lineitems.
The UserDB class (pg. 449-451)

```java
public static void delete(User user) {
    EntityManager em = DBUtil.getEmFactory().createEntityManager();
    try {
        em.getTransaction().begin();
        em.remove(em.merge(user));
        trans.rollback();
    } finally {
        em.close();
    }
}
```

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

The UserDB class (continued)

```java
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);
    return q.getSingleResult();
}
```

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

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

Here the incoming User object is left over from before the transaction, thus detached. Merge makes it managed, so it can be removed.
```

Pizza2 infrastructure code

```java
package murach.data;

import javax.persistence.*;

import murach.business.User;

public class UserDB {

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

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

    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);
        return q.getSingleResult();
    }

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

```
The UserDB class (continued)
```

```java
public class UserDB {

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

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

    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);
        return q.getSingleResult();
    }

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

```
The UserDB class (continued)
```
Pizza2 Entity: class and id field annotations

```java
@Entity(name = "PizzaSize")
@Table(name="PIZZA_SIZES")
public class PizzaSize implements Serializable, Comparable<PizzaSize> {
    private static final long serialVersionUID = 1L;
    @Id
    @TableGenerator(name="SizeIdGen",
        table = "PIZZA_ID_GEN",
        pkColumnName = "GEN_NAME",
        valueColumnName = "GEN_VAL",
        pkColumnValue = "SizeId_Gen")
    @GeneratedValue(generator="SizeIdGen")
    @Column(unique=true, nullable=false)
    private int id;
```

Table needed for id generators: one row for each entity needing new ids

```sql
CREATE TABLE PIZZA_ID_GEN (GEN_NAME VARCHAR(50) NOT NULL, GEN_VAL INTEGER,
    PRIMARY KEY (GEN_NAME));
INSERT INTO PIZZA_ID_GEN (GEN_NAME, GEN_VAL) values ('Ordno_Gen', 0);
INSERT INTO PIZZA_ID_GEN (GEN_NAME, GEN_VAL) values ('SizeId_Gen', 0);
INSERT INTO PIZZA_ID_GEN (GEN_NAME, GEN_VAL) values ('ToppingId_Gen', 0);
INSERT INTO PIZZA_ID_GEN (GEN_NAME, GEN_VAL) values ('MenuSizeId_Gen', 0);
INSERT INTO PIZZA_ID_GEN (GEN_NAME, GEN_VAL) values ('MenuToppingId_Gen', 0);
```

JPA will do the select, update of the appropriate row automatically

PizzaOrder to PizzaSize relationship

```java
@OneToOne
@JoinColumn(name="SIZE_ID",
    nullable=false)
private PizzaSize pizzaSize;
```

PizzaOrder to PizzaTopping relationship: one-to-many

```java
@OneToMany(
    mappedBy="order")
private Set<PizzaTopping> pizzaToppings;
```

Pizza2 Code

• Same service API, so presentation code is same as pizza1
• Service method code: look at examples, see transactions started and finished in each service method. We use the domain objects as before.
• DAO: PizzaOrderDAO.java is 166 lines in pizza1, only 81 lines in pizza2 because JPA does all that messy JDBC code for us. We still have to code the queries of the finders of course, now in JPQL.

Pizza2 Code

• Same service API, so presentation code is same as pizza1
• Service method code: look at examples, see transactions started and finished in each service method. We use the domain objects as before.
• DAO: PizzaOrderDAO.java is 166 lines in pizza1, only 81 lines in pizza2 because JPA does all that messy JDBC code for us. We still have to code the queries of the finders of course, now in JPQL.
Some pointers

- Also see [JPA2Notes.html](JPA2Notes.html) for some of the comments on the book contained in these slides

**Notes on setting up the pizza2 JPA Project**

- Need to reload the databases for pizza2, because there's a new table
- Also new to database directory: 3 versions of persistence.xml, one for each DB
- See pizza2/README for how to run the project

**Creating a Project in eclipse** for non-web project using JPA, such as pizza2.

- For our purposes, you can treat a JPA project as a Java project, since all the needed libraries are in lib. Then you can edit the sources, use the debugger, etc., just as before.