Intro to JPA

Last time: Intro to servlets and tomcat

A servlet is Java code executed inside a web server for a certain application. One app could have several servlets.

Tomcat is a web server that can handle servlets (unlike many Apache servers)

Tomcat gets an HTTP request, figures out which servlet it's for, calls into that servlet (our Java code) to handle it and produce HTML for tomcat to send back to the user.

URL accessing my pizza3-specific tomcat installation: http://pe07.cs.umb.edu:9002/welcome.html (recall hw3 page flow problem)

Here the pizza3 app owns the whole website served by this tomcat, an "embedded tomcat".

We'll use it this simpler (embedded) way when developing webapp projects, using Spring Boot 2.

There is a shared tomcat web server running on port 8080 on pe07. It can handle multiple apps, much like Apache, but also supports Java. We looked at some simple JSPs running there. Each JSP compiles into a servlet. We finally saw code (JSP) that captures user input from a form.

Intro to JPA (not on exam next week)

JPA stands for Java Persistence Architecture, i.e., a standard API of Java that interfaces to various providers of its implementation, similar how the Java standard JDBC API interfaces to the database drivers that implement that API.

JPA provides an API for ORM, Object-Relational Mapping

ORM idea: take advantage of the similarities in data model and object model and automate the conversion of database data to/from objects.

Example: employees table, with name, salary, etc. vs. Employee object with those properties

Although JPA2 is considered part of Java EE (Enterprise Edition), not Java SE (Standard edition, the usual JDK), it does not depend on anything else in Java EE, so can be used as an extension of Java SE, as we do for pizza2, the JPA version of the Pizza project.

JPA2, released in late ’09, is widely used and supported by Hibernate ORM and Eclipselink, among other “providers”, i.e., implementations.

Our JPA implementation: Hibernate

We'll be using Hibernate 5.4, version-managed by spring-boot-starter-parent in pom.xml.

The original Hibernate project started in 2001 evolved into JPA1, then JPA2 standards.

Your JPA2 (now 2.2) knowledge can be used for Hibernate and Eclipselink projects.

JPA works with object-relational mappings: employees table ←→ Employee domain object, etc.

These mappings can be specified by annotations in domain classes or by XML files—we'll use annotations, the programmer's favorite method, for example:

```
@Entity
@Table(name="PIZZA_ORDERS")
public class PizzaOrder implements Serializable {
…
@Column(name="room_number")            ←can map to any column name
private int roomNumber;
```

JPA supports “persistent objects”

- The idea is that an Employee domain object becomes a **persistent object**
- This is standard terminology, but possibly misleading. Our domain objects are not themselves persistent (and stay that way under JPA). Persistent means long-lasting.
- They have a short lifetime – “scratch copies” of DB Data or objects for data on way to DB
- But because they are related through the software to DB data, they are called "persistent"
- The domain objects, themselves short-lived, are tied to truly persistent row(s) in the **entity table** in the DB via the object-relational mapping.

Identity of these “persistent objects”

Recall the idea of Java object identity: its ref’s value, the address of the object in memory

- but this changes from one scratch copy to another

Persistent identity: PK value of the row of the entity table in DB.

Example: id column value of pizza_orders table row, becomes the value in id field of the corresponding PizzaOrder object, and provides its persistent identity

Worry: We could end up with 2 Java objects with the same persistent id, apply change to one of them, get mixed up, fumble the edit.

One thing the JPA runtime system does for us is make sure there’s only one, by tracking each domain object it knows about, using its persistent identity.

Changes in objects vs. changes in DB rows

Changes in a Java object can happen anytime via method calls by the app's program

vs.

Changes in the database row(s) via transactions, which happen only at the end of our service calls.

These changes are not perfectly synchronized, so we end up with persistent objects being "quasi-transactional"

This is not usually a problem in our case of short transactions related to individual service methods.

We will see that the JPA runtime system can figure out what has changed in a persistent object and commit those changes to the database in important cases.

How does JPA track our domain objects?

We have to tell JPA about a domain object before it can help us with persistence.

In particular, a new object is unknown to JPA, so we make certain calls for it (e.g. "persist") before transaction commit.

How do we interact with the JPA runtime system?

Answer:

- First, we obtain an EntityManagerFactory (emf) object by special configuration code at startup.
- Later, at transaction start, we call on that emf to obtain an EntityManager (em) object
- Then we use the em's methods to talk to JPA's runtime: em.persist(domain-object) for ex., or tell it to commit the transaction.

JPA can help with relationships

- We can annotate the fact that one pizza order has many toppings, but only one size.
- Recall how pizza1 had DAO finders that filled out the toppings for an order, and others that didn’t.
- With JPA in use, we can specify which way we would like the DAO to do it, EAGER or LAZY, although JPA runtime can decide to fill out the details even if we said we didn’t want them.

Setting up the pizza2 JPA Project

- Need to reload the databases for pizza2, because there's a new table, so see its README for directions on running the project.
- To set it up in eclipse, just treat it as a Java Maven project: use Open Project from Filesystem after unzipping it.
- You can see it has the same scripts as pizza1, like runOnH2 and database/runH2Script, so it should be easy to work with.
- It is using Hibernate as the JPA provider, i.e., implementation. JPA itself is just an API, like JDBC.
- Will be covered in upcoming Homework 4

- Now Look at Chap. 13 slides.