Lab 5 - JPA Relationships

Summary

The purpose of this assignment is to expand our final project business domain with additional entities, implement relationships within our business domain, explore the JPA requirements for applications to manage both sides of bi-directional relationships, and demonstrate with JUnit test cases. Finally, this project also incorporates both a maven web application and JUnit testing within the same project, demonstrating the use of two separate persistence units, one JTA (for use by Payara while running our web application) and one RESOURCE_LOCAL (from use by Java SE while running unit tests).

Requirements

Documentation

You can take a break from documentation this week. No docs required for this lab. I will be asking you to write about your Final Project design in detail, including the relationships between your entities, on the written Midterm - so use this Lab to experiment with your relationships, and document the design on the written Midterm question.

Database Setup

Use your itmd4515 database and user from Lab 1.

Project Setup

Your uid-fp repository should already be setup, and you should continue pushing your commits into GitHub for this lab.

We will be working in this repository from now until the end of the semester. Please remember, I will be looking for multiple commits. I would suggest using the lab number in your commit message as a prefix so you can also review the history throughout the semester, for example:

- Lab 5 - Initial Commit
- Lab 5 - OneToOne unidirectional relationship between Foo and Bar
- Lab 5 - Test cases for unidirectional 1:1 relationship between Foo and Bar
- Lab 5 - ManyToMany bidirectional relationship between Widget and Gizmo
- Lab 5 - Test cases for bidirectional M:M relationship between Widget and Gizmo

Project Requirements

1. If you haven't already (from Lab 3), define a JDBC Resource for use by your application. There are many many ways to do this:

1. Creating a Payara JDBC Connection Pool and JDBC Resource via the admin console
2. Creating a Payara JDBC Connection Pool and JDBC Resource via the asadmin command line utility
3. Defining a JDBC Resource directly in Payara domain configuration file
4. Defining a JDBC Resource via web.xml
5. Defining a JDBC Resource via annotations in our code (this is what we are going to do. the others might be more useful if you are hosting multiple applications in one server that could share a data source)

2. Your @DataSourceDefinition should look something like below. If you have multiple parameters in your JDBC URL, you would include them as parameters via the annotation **Note** - depending your operating system, you may also need to add useSSL=false.

   Important - If you are re-using your DataSourceDefinition from Lab 3, make sure you don't forget to change the databaseName to itmd4515 like me!

```
@DataSourceDefinition(
        name = "java:app/jdbc/itmd4515DS",
        className = "com.mysql.cj.jdbc.MysqlDataSource",
        portNumber = 3306,
        serverName = "localhost",
        databaseName = "itmd4515",
        user = "itmd4515",
        password = "itmd4515",
        properties = {
            "zeroDateTimeBehavior=CONVERT_TO_NULL",
            "serverTimezone=America/Chicago",
            "useSSL=false"
        }
```

3. Consider the deployment of your MySQL JDBC Driver. Assuming you are also using a @DataSourceDefinition, you should be able to include the JDBC driver with your application by ensuring it has default scope in your pom.xml as I am showing in class. Alternatively, you can copy the MySQL JDBC jar file to your Payara domain's **lib/ext** (extensions) folder.
4. If necessary, move your **RESOURCE_LOCAL** Persistence Unit from Lab 4 named **itmd4515testPU** to an appropriate location for testing. In class, I will demonstrate that you can create this manually in the Files view by creating a src/test/resources folder, and then moving your persistence.xml. This may also necessitate specifying entities within the persistence unit - I will demonstrate this in class.
5. Create a new **JTA** Persistence Unit named **itmd4515PU** connecting to your **itmd4515** database using the **itmd4515** user. If you have any trouble creating this in NetBeans, you can also create this manually by copying your test persistence.xml to the right location and modifying the XML.

   Once complete, you should have two persistence.xml files:

src/main/resources/META-INF/persistence.xml *and* src/test/resources/META-INF/persistence.xml

Make sure the Persistence Unit defined in each file has a different name, and that your test class refer to the **itmd4515testPU** PU. We will move forward with test cases pointing to a "test" PU, and the web application pointing to a "production" PU.

Make sure you double-check your pom.xml in case the NetBeans persistence wizard added an additional JPA dependency. if so, remove it. Your dependencies should continue to look like Lab 4, except for JPA! Please update to the versions specified in class!

6. If necessary, re-factor your entity classes so that they reside in a dedicated domain or model sub-package. If necessary, update your test class(es) to reside in the same package.
7. Based on your design thoughts from Lab 4, introduce additional entities to your business domain. The requirements are:
    1. You must have at least four total entities in your **business domain**. You may have as many as you want, but at least four.
    2. Remember, your final project must support the logical concept of multiple types of users or roles, but ***do not*** create these as entities. We must consider this now so we can introduce security in later projects, and we will use the JEE security framework to introduce these security entities later. Likewise, do not create Admin as an entity. **Stay focused on entities within your domain.**
8. As before, all your entities should:
    1. Use an appropriate PK strategy, and use an appropriate data type for your PK (as per the options discussed in class)
    2. Include appropriate equals and hashCode methods for your PK strategy
    3. Include at least one temporal data type
    4. Include at least three different data types. There is no limit to the number of attributes you can include. Your attributes should be sufficient to represent your entity. Exercise good design and judgment.
    5. Include appropriate Constructors, accessors and mutators
    6. Include appropriate bean validation constraints based on your database types and sizes
    7. Include appropriate toString method for formatted output
    8. In other words, your entities should **make sense.** Do not use mine from class demo. I am coding fast and furious, to try and demonstrate all you need for 1 week in a few hours.
9. Include at least three relationships between your entities
    1. At least one of the relationships must be bi-directional
    2. Try and use different relationship types. I will not look favorably on 3 OneToOne uni-directional relationships. This is for you to learn about relationships, and to learn about relationships you need to introduce some real relationships into your business domain.

10. Include appropriate helper methods in your business domain to manage both sides of the relationship. Will be demonstrated in class.
11. Two JUnit test cases to illustrate that your relationships are working as expected:
    1. One Uni-directional relationship test case
    2. One Bi-directional relationship test case
        1. Consider the example I worked through in class. You can assert that persist was successful, and that collections on both sides of a bi-directional relationship contain the expected entity.
12. Submit to Canvas or Beacon
    1. Right your **uid-fp** project and select "Clean"
    2. Go to your NetBeans Projects directory. Create a zip file of the **uid-fp** folder and submit it to the Canvas or Beacon assignment.