SYSC 4806 – Lab 2: ORM

The goal of this lab is to use object-relational mapping, specifically JPA, to persist our address book and its list of buddies to a database. First, you need to read this article, all the way down to (but excluding) the section called "Reference Implementation". Armed with this knowledge, you should inspect the code in this example of using JPA to persist to a SQLite database that you can refer to and borrow from.

First, recall the layered cake that is ORM:

- First, you need a database. In this lab we're going to use SQLite, which doesn't require any client-server setup and just stores things in a single file. H2, an in-memory database, is another simple option, which we will explore in future labs.

- Next, you need a database-specific JDBC driver so that you can query the database from inside your Java program.

- JPA is an ORM layer on top of JDBC, but it really is only a standard. You need an actual implementation of JPA. For this lab we will use the EclipseLink implementation, but in future labs we will use Spring's. Trying things with different libraries and tools is a good exercise in figuring out the "magic" that happens behind the scenes!

- Finally, the Java classes need to be annotated with the appropriate JPA tags.

So, here are the various steps to follow:

1. Once you've loaded your AddressBook project in IntelliJ, edit the Maven pom.xml file to include the dependencies on the SQLite JDBC driver and the EclipseLink implementation of JPA. See the pom.xml of the example linked above to get the dependencies right. Every time you change the pom.xml file, you need to reload the project, to download and install the new dependencies. Press ⇧⌘I (macOS), or Ctrl+Shift+O

(Windows/Linux), or click the little Maven icon to get it to download and install the dependencies. Or in the Maven tool window, right-click on your project and select "Reload project".

2. Next, you need to set up a configuration file called `persistence.xml` and that you have to save in the folder called "META-INF". Maven may have already created this folder for you, but if it didn't, make sure to create the META-INF folder inside another folder called "resources" which itself should live inside the "src/main" folder. If you had to manually create the "resources" folder, mark the folder as a "Resources Root" by right-clicking on the folder and selecting the "Mark Directory as" option, which then offers you the "Resources Root" option. The content of the `persistence.xml` file can be adapted from the example in the link provided above. You can either create this file manually (use `new -> file` and type or copy-paste what you need in that file), or you can rely on IntelliJ to help you. For the latter option, right-click on your project, choose "add framework support", then choose "JavaEE persistence"[1]. It will ask for a version of persistence.xml; pick 2.1 or 2.2. It will then ask for a provider, for which you can specify "EclipseLink". No need to do anything else with the wizard. Either way, at this point you should have a persistence.xml file in the META-INF folder, and you need to modify the file for your purpose. We're going to see how in the next step.

3. Let's look at the persistence.xml file of the example in the second link above more closely. It defines a persistence unit (the name you give it is up to you, just know that you will refer to it later when you will invoke it in your java code, and to be safe don't use spaces or special characters in this name), which is comprised of:

---

[1] If you don't see the "JavaEE persistence" option, you must open project module settings, go to facets, and remove the JPA facet. The option should show up then!

2

- o the persistence provider (that's the full name of the class that implements JPA for you. If you used the wizard it probably set this for you already, and it should be identical to what is in the example). Just make sure that the transaction-type of your persistence unit is set to "RESOURCE_LOCAL" just like in the example. This is to specify that we are using persistence in a standalone desktop application.

- o the list of classes to be persisted. You'll need to modify this part with the full class names (including package name) of your AddressBook application;

- o then there's a set of properties to configure parameters that the persistence provider requires, i.e., the name of the JDBC driver, the location of the database, etc. These values should be identical to the ones in the example. Since we're using SQLite, we won't need to specify a username or password to access the database.

4. Now you can start annotating your AddressBook classes. Let's start small, and first only annotate and persist a BuddyInfo. You can use the Product.java class provided in the example for inspiration. You basically need to specify that BuddyInfo is an @Entity, and you need to provide a primary key identifier using the @Id tag. IntelliJ will help you import the right packages so that the tags are recognized. Some JavaBeans conventions will need to be respected: make sure to provide a default (i.e., no-arg) constructor (if you don't have any constructor at all you are fine, but if you only have non-default constructors you should also provide a default constructor), make sure your accessors and mutators are following proper getXXX and setXXX conventions.

5. Let's now test that we can persist a BuddyInfo object! In a separate class (how about a unit test?), create an entity manager (see the JPATest class in the example), and, within a transaction (see example again), use the entity manager to persist instances of your annotated class. Now you can query the table to see if you can retrieve those objects you just persisted.

In JPA, the query still needs to be written using a SQL variant, but in other ORM (such as Active Record in Rails, and to some extent in Spring) this is not usually necessary. In fact, even configuration files like persistence.xml are often not necessary when using a full-blown framework and the power of introspection and conventions, as we will see very soon!

- o Problem-tracking tip #1: if IntelliJ doesn't recognize your annotations, it could be because you haven't updated Maven with the dependencies on EclipseLink properly.

- o Problem-tracking tip #2: double-check that the persistence unit name you used when invoking the EntityManagerFactory matches the one you specified in your persistence.xml. Make sure the name you used doesn't contain spaces or other non-alphanumeric characters.

- o Problem-tracking tip #3: when building, Maven copies files stored in the "resources" folder over to the target folder. This means it may not pick up the persistence.xml folder if it is in the src folder. Two solutions:

  - ▪ Copy it into the resources folder so that the build works.

  - ▪ Or tell Maven where to look, using the <u>\<resources> tag</u> .

- o Problem-tracking tip #4: some students have reported a "not known entity" error. This could be because in very recent versions of Java, the javax.persistence.* package (and more generally, libraries that are part of Java EE) has been moved to jakarta.persistence.* So the easy fix would be to use an older version of Java (Java 17 works for me), and the more complicated but more sound fix would be to use the newer packages (and corresponding Maven dependencies, if needed).

6. You made it here? Congratulations, this means you successfully persisted a BuddyInfo object! Now let's annotate AddressBook so that we can persist an entire address book containing many buddy info objects. You can use the example in the <u>first link above</u> for inspiration, but basically,

you need to capture, using annotations, the fact that an AddressBook has a one-to-many relationship with BuddyInfo. If you do this successfully, you should be able to persist a few buddy infos, add them to your address book object, then persist the address book itself.

7. Bonus round: it is possible to create an address book first with non-yet-persisted buddyInfo objects, in such a way that when you persist the addressBook it also persists its buddyInfos. For this to work you need to investigate the "CascadeType" of your one-to-many annotation. It's just a one-liner; can you get this to work?

When you are done, show your work to the TA and put the entire directory in a zip file and submit on CuLearn.