Workshop 4: The Back End, Java Persistence API and EJB component model

Objectives

Now we will map the shop model to a persistent storage (a relational database). If combining this workshop with any front-end from the previous workshops we'll get at fully functional web shop (or near). We will use;

- GlassFish.
- Java Persistence API (JPA) to map persistent objects to database tables and create relationships between tables and more.
- Test driven development with embedded database and JUnit.
- Introduction to EJB (EJBLite).

PLEASE: INSPECT CODE SAMPLES FROM THE LECTURES (ON COURSE PAGE)! EVERYTHING YOU NEED SHOULD BE THERE. WILL HOPEFULLY SAVE YOU A LOT OF TIME!

Final date: See course page

1 Inspecting a sample database

For now we're going to use the Java DB (aka Derby) which is bundled with NetBeans. Start with a quick look at the sample database.

1. In NetBeans go to Services tab > Databases > Right click "jdbc:derby.../sample" > Connect (possible, login/passwd = app/app).
2. Expand the node "jdbc".... > APP > Tables > CUSTOMER.
3. Mark CUSTOMER > Right click > View data. Change the SQL to the below and click run (button with small green triangle);

```
select * from APP.CUSTOMER where customer_id < 150;
```

4. It's also possible to add, delete or update data in the table. Use icons in the tool bar or cursor over the table data > Right click > .... or just click in a cell and alter the value. Click commit-button in tool bar (icon with table and small check mark) to commit the change (really write it).

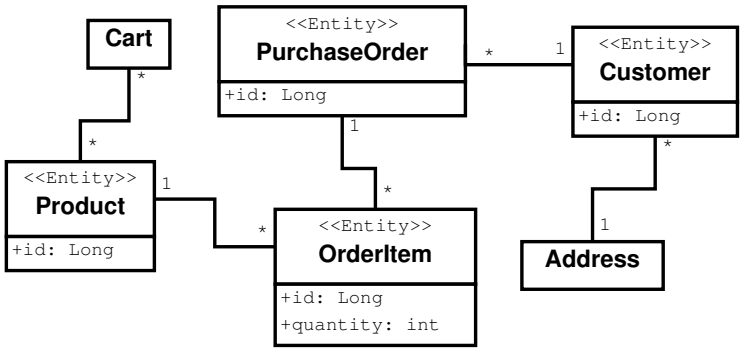
2 Persisting the model

The overall goal is to replace the usage of the in-memory shop-model with a real relational database. This will force us to do some modifications of the model and replace the in-memory-code with real database code using JPA. The final outcome should be JUnit tests covering all methods in the ProductCatalogue.

NOTE No skeleton code for this workshop, just two JUnit test, download and add to project. Test are added under way, implement methods in indicated order (getCont() and getRange() will be the last). Final result in Appendix.

2.1 The persistence model

The parts of the model that should be persisted.



2.2 Creating the database

First we'll create the database.

1. In NetBeans go to Services tab > Databases > Java DB (right click) > Create Database...
2. Fill in (and than OK):
   a) Database Name: shop
   b) User Name and Password: app
3. A new connection should show up.
4. Connect (right click) and inspect the (empty) database.

Deleting a database

You possible have need for this. Test now or remember for future use.

1. Mark jdbc:derby://.../shop > right click, Delete.

If in trouble, stop database server, go to .netbeans-derby directory and delete the folder named as the database. Restart database server.

2.3 Mapping the model

If database deleted, create and connect again. Now we'll transform the persistent classes in the model into JPA Entity classes. For now we use default mappings.

1. Make a clone of the shop-project (the model), rename to jpa_shop. NOTE: The clone is not an web application.
2. Add dependencies for eclipselink-2.3.0, derbyclient-10.6.1.0, javax.peristence-2.0.3 and org.eclipse.persistence.jpa.modelgen.processor-2.3.0.jar.
3. Copy all classes from the core package in the original shop model to jpa_shop (put in "core"-package).
4. Do the following;
   a) Rename Shop-class to JPAShop and ShopFactory to JPAShopFactory.
   b) Rename/modify IEntityContainer to public interface IDAO<T, K> (i.e. Data Access Object). Remove sort method.
   c) Rename AbstractEntityContainer to AbstractDAO, let it implement IDAO. Modify class as follows ...
      i. Change to; public abstract class AbstractDAO<T, K> implements IDAO<T, K> { ...
      ii. Remove List<T> elems and remove content from all methods. Add return statement as needed (to be able to compile).
      iii. Let constructor have a parameter; Class<T> clazz and String puName (persistence unit name). Use like this;

```
private EntityManagerFactory emf;
private final Class<T> clazz;
protected AbstractDAO(Class<T> clazz, String puName) {
  this.clazz = clazz;
  emf = Persistence.createEntityManagerFactory(puName);
}
```

   d) Let all container interfaces (IOrderBook, etc.) extend IDAO and let all containers (OrderBook, etc.) extend AbstractDAO and implement interface. Let constructor pass the class-object for the "contained" class to super constructor. Example;

```
public final class CustomerRegistry extends
          AbstractDAO<Customer, Long> implements ICustomerRegistry {
public CustomerRegistry(String puName) {
  super(Customer.class, puName);
```

```
}
```

e) Modify factory to JPAShopFactory.getShop(String persistenceUnitName);

5. For all classes to be persisted; add default constructor and remove any "final".

   a) Delete interface IEntity, and replace it with @Entity on implementing classes.

   b) Remove the random generation of id's in AbstractEntity class. Replace with @Id and @GeneratedValue.

      Note This is possible a disputable design. Suffices for this course.

      i. During this NetBeans will show a notification (light bulb). Click > Create Persistence Unit. Accept. A dialog pop up....(if not check persistence units from samples).

      ii. ... fill in. Name: shop_pu. Select the created database "shop" for database connection > Create. Inspect Other Sources/META-INF/persistence.xml. Note: Transaction type; RESOURCE_LOCAL, application will handle transactions (not container).

      iii. Add all Entity classes to the shop_pu.

   c) We need to add some annotations in the entity classes to inform JPA that; There are attributes of type Date, Cart should not be persisted and that Address should be stored together with Customer (same table). Add the annotations.

   d) Implement method add() in AbstractDAO. Remember; RESOURCE_LOCAL all transactions must be handled by application (for write operations).

Testing the model

a) Go to Files tab. Copy src/main/resources/META-INF/persistence.xml to src/test/resources/META-INF/persistence.xml. Rename PU to shop_test_pu. In Projects-tab there should show up an icon "Other Test Sources" (i.e. right now we have one PU for the real database and one for test)

b) Use the TestProductCatalogue, the test-PU and the JUnit-test as a starting point for testing. Comment out and/or add more tests as needed. Test the add()- method (if fail; read exception messages!).

c) Inspect generated database tables.

Note This can be tricky, really have to check everything! Spelling! Database must run etc.!

3 Test driven development with the embedded Java DB

Generally it's much more efficient to test the persistence layer in an JSE environment using JUnit test's either using a running database or...

... use the embedded Java DB (don't need to run the database server). The possibility to drop and create tables will give us fresh data for every test.

Note When running the embedded database it's not possible to see the outcome (tables are all in-memory). Must use assertTrue(..) etc.

Warning Using the embedded database will create a tmp directory in project dir. Never put this under version control (seems machine dependent)!!! Delete before commit (or better use .gitignore). If strange things happens or disaster remove it.

Tip Of course it's possible to debug the tests (Debug Test File).

1. Add a *test* dependency for the embedded derby database in POM (derby-10.6.1.0.jar).
2. Create a another persistence in unit src/test/resources/META-INF/persistence.xml (in same file).
   a) name="shop_test_embedded_pu"
   b) Specify in PU to use the embedded Derby, see code samples.
   c) Add the Product class to the PU.
3. Stop database server.
4. Modify the test for add (need some assert-statement) and run.
5. Implement methods delete, update and find in AbstractDAO.
6. Run the JUnit-test with PU for embedded Derby.

4 Customizing Mappings

For now we have mostly (only) used default mapping. If we're happy no problems! But we're not...

1. Add mapping to remove any extra tables possible generated by JPA (any other than; Customer, Product, OrderItem and PurchaseOrder). The SEQUENCE table will be created by Derby for keys, so ok). Use @ManyToOne, etc.

   Tip When adding annotations light bulbs will pop up. Click for suggestions. Remember no mutual dependencies between classes!

2. Keys are never allowed to be null, add JPA annotations (not CDI annotations, this is persistent data, not control layer objects).

3. We need the "cascade". The cascade will store complete object graphs (so we don't need to store or delete each object separately in the correct order). What should be cascaded?

4. Try to persist a minimal object graph using the TestCascade JUnit-test.

5 Queries

Now we would like to do some more general querying. For this we'll better use the Query API with JPQL (query strings) or the Criteria API (criteria objects).

5.1 Queries with JPQL[1]

1. Add search facilities to the ProductCatalogue. It's should be possible to search on any Product attribute (getByName, getById(), .. getByAny() or similar). Use JPQL.

   Tip It's possible to run JPQL from within NetBeans (no ';' - char last in queries, I'll think...)

   - Mark the persistence.xml > Right click > Run JPQL. Enter query and click Run icon.
   - Also possible to run SQL. Note case sensitivity!

   Note Beware of types! JPQL queries must be type correct. Example (po = PurchaseOrder, c = Customer);

   SQL (Tables): po.id = c.id, both integers, OK!

   JPQL (Objects): po.id = c.id, wrong, po.id is a reference to a Customer object, c.id is a Long!

2. Test.

5.2 Queries with Criteria API

(Optional) The criteria API can replace JPQL thus creating the same queries but in an object oriented and type safe way. To make the queries type-safe the API uses static metadata-model classes, see Projects tab > Generated Sources.

1. The methods getRange and getCount in AbstractDAO must use the clazz attribute (because AbstractDAO is generic). It's possible with JPQL using clazz.getSimpleName() in query string ...

2. ... or using the Critera API ...

3. Chose one of the above. Implement and test.

Finished Now you may report the workshop ...

[1] JPQL should be database independent. So should be possible to switch database.

6 Connecting back and front-end

(Optional) Now it's possible for any front-end to use the JPAShop as a back end. Just replace the dependency on shop with jpa_shop in pom.xml. Give it a try (probably need some tweaking).

7 Enterprise Java Beans and container managed persistence

(Optional). This is the preferred, and optimal, style if using an application server. Recommended for project! The goal is to turn all containers (OrderBook, etc) into EJB's thereby handling off many responsibilities to the container (life cycle, transactions, persistence context propagation, concurrency, ...).

1. Clone any of the front ends and rename to nn_ejb_jpa_shop (nn = servlet or ws or jsf).
2. Copy model and database package from jpa_shop to the clone.
3. Create an persistence unit with type JTA. PU should have a <jta-data-source> named jdbc/shop (holding database info, connection etc.). NetBeans possible ask if you would like to create a data source, if so do. Else New > GlassFish > JDBC Resource, name jdbc/shop. A folder "setup" should be created in Other Sources (possible when running..).
4. Delete shop and shop factory. Inject the EJB's where you need them.
5. Add annotations to make the containers stateless EJB's. Add @Local to corresponding interfaces.
6. Clean up AbstractDAO by removing all code for transactions (just keep the call to the EntityManager). Inject an EntityManager context.
7. Try to get it up and running.

Appendix

Final project content.