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

Objectives The 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 Arquillian tests covering all methods in the ProductCatalogue. We will use;

- GlassFish and a relational database, JavaDB (aka Derby bundled with GlassFish).
- Java Persistence API (JPA) for ORM-mappings, EntityManages, queries, ...
- Enterprise Java Beans (EJBs) to host the JPA code (container managed persistence).
- Arquillian (embedded container) to test the persistence layer.

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

Start with a quick look at a sample database.

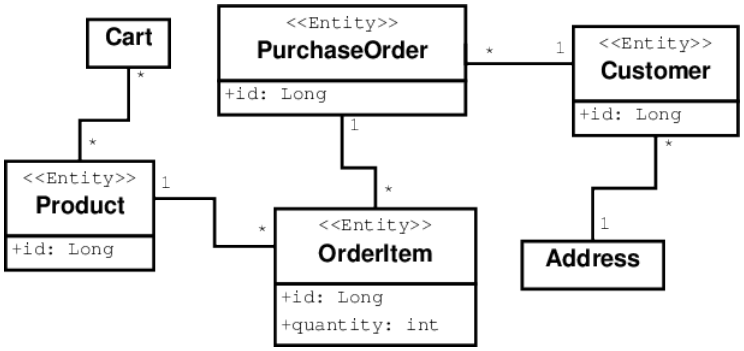
1. In NetBeans go to Services tab > Databases > Right click "jdbc:derby.../sample" > Connect (possible, login/passwd = app/app).
2. Expand the node "jdbc".... > APP > Tables > CUSTOMER.
3. Mark CUSTOMER > Right click > View data. Change the SQL to the below and click run (button with small green triangle);

```
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.

2 Design

The parts of the model that should be persisted (stereotypes <<Entity>>). The Cart should not be persistet but the content should be tranformed into OrderItems which are in turn persisted.



3 Creating the database

First we'll create the database. This is an application external operation.

1. In NetBeans go to Services tab > Databases > Java DB (right click) > Create Database...
2. Fill in (and then 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.

4 Implementation

Final project structure in appendix.

4.1 Creating the persistence layer

If database deleted, recreate.

1. Download skeleton code. Open in NetBeans, the will be errors.
2. Copy core-classes from the original (non JPA) shop model and put in core-package.

3. Copy util-classes from the original (non JPA) shop model and put in persistence-package.

4. Rework Abstract Entity to an JPA mapped superclass. Let class handle id, automatic generation of id, equals and hashCode (all entity classes should inherit Abstract Entity, check).

5. Rework and rename IEntityContianer to IDAO. Only modification is to remove bound on type parameter T.

6. Remove IEntity (replaced by entity classes).

7. Rework and rename AbstractEntityContainer to AbstractDAO. Let it implement IDAO. Remove List elems and empty all methods (put a return null or similar to be able to compile). Add attribute; private final Class<T> clazz (used to get correct type), a constructor with clazz as parameter and a method; protected abstract EntityManager getEntityManager().

8. Rework IProductCatalog. Let it extend IDAO<Product, Long>. Add Local annotation to make it a local interface for EJBs

9. Make ProductCatalogue a statless EJB. Let it inherit AbstractDAO and implement IProductCatalogue. Inject an EntityManager and add a constructor calling the superconstructor with Product.class as argument. Implement getEntityManger method to return the EntityManager.

10. Do the same procedure with CustomerRegistry and OrderBook.

11. Let Shop be an applications scoped CDI bean and inject the previously created EJB's (use @EJB) There should be no use of "new" anywhere in the model, i.e. all objects managed by GlassFish.

12. All errors should be gone, if not contact asssitant.

4.2 Mapping the model

Now it should be possible to start the mapping.

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

2. For all classes to be persisted; Add JPA annotations as needed, Entity, Embedded, Column, OneToMany, JoinColumn, etc.

3. Start implemening methods in AbstractDAO (except; findAll, findRange and count, needs queries, see below). After each method write a test ...

5 Testing the Persistence Layer

We'll use Arquillian. It's possible to use both an embedded EJB container and an embedded database. To be able to inspect the tables we prefer a non-embedded database i.e. JavaDB must be running.

Note This can be tricky, really have to check everything!

1. Inspect files in src/test/resources (Arquillian config files).
2. Use the supplied Arquillian tests as a start. Comment out testProductGetByName.
3. If create()-method (in AbstractDAO) implemented it should be possible to run testPersistAProduct (also; all mappings must be correct). Try! If success inspect generated tables and data.
4. Add tests to cover all operations, implemented so far, on the ProductCatalogue.

6 Queries

6.1 Queries with 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.

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!

1. Implement findAll, findRange and count in AbstractDAO using JPQL queries. This involves some string hacking (use clazz.getSimpleName()). Will generate warnings, accept for now. Test!
2. Add search facilities to the ProductCatalogue. It's should be possible to search on any Product attribute (getByName, getById(), .. getByAny() or similar). Test.
3. Try to add some data into the database using bulk update. Use a Servlet (inject a DAO) and hard code some input.

6.2 Queries with Criteria API

(Optional) Test same as above using the Criteria API (no need for string hacking). Keep old code, only comment out.

7 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_skel in pom.xml. Give it a try (probably need some tweaking).

Appendix

Final project content (name will be jpa_shop_skel).