JPA Query API, Design, Testing and Authorization

JPA Slides #5

Queries

Would like to ask more general question, not just CRUD

- How many of age < 21 living in Gothenburg and earning more than 30k kr/month?

Ad hoc queries handled by the JPA **Query API** in combination with

- JPQL, embedded query language (Strings), similar to SQL, or...
- ... **Criteria API**, type safe queries (Objects) or ...
- ...Native SQL (Strings, platform dependent), ...we don't!

Java Persistence Query Language JPQL

"Objectified" version of SQL, query strings, working with (collections of) objects

```
// JPQL is case insensitive, both ok
"select c from Country c where c.population > 2000000"
"SELECT c FROM Country c WHERE c.population > 2000000"
```

General form

```
// The below are “clauses” (FROM clause etc.)
SELECT ... FROM ...
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```

JPQL vs SQL

In the query language, an expression can traverse, or navigate, to related entities. These expressions are the primary difference between the Java Persistence query language and SQL.

- JPQL Queries navigates to related entities, whereas SQL joins tables
- The names of the objects must be the names from the Java classes (type, attribute names, not table/column names)

```
//Table PRODUCT column name (bad)
PRODUCT_CODE

// Attribute of class Product (ok)
p.productCode
```

JPQL Identification Variables

Identification variable(s) declared in FROM clause

```
// Assume classes Magazin, Article and Author
// All magazines that have articles authored by someone with the
first // name "John".
SELECT DISTINCT mag FROM Magazine mag JOIN mag.articles art JOIN art.
author auth WHERE auth.firstName = 'John'
```

- mag evaluates to type Magazine
- art evaluates to type Article reachable from Magazin
- auth evaluates to type Author reachable from Article

Possible type errors

```
// Type error: customerId an object (a customer) id is a Long
"select.... where order.customerId = customer.id"
// Correct, both objects
"select.... where order.customerId = customer"
```

JPQL Path Expression

Identification variable followed by the navigation operator (.) and a state-field or association-field is a path expression

```
// Assume mag has type Magazin then path expression is illegal
// Having a collection "in the middle" of path (collections must be
last)
mag.articles.author

// Solution
// Introduce extra identification variable to range over collection
// (also possible with joins)
// All authors that have any articles in any magazines (NOTE ',')
SELECT DISTINCT art.author FROM Magazin AS mag, IN(mag.articles) art
```

JPQL Where and Select Clause

WHERE clause consists of a conditional expression used to select objects or values that satisfy the expression

- Composed of other conditional expressions, comparison operations, logical operations, path expressions that evaluate to boolean values, boolean literals, and boolean input parameters. Arithmetic expressions can be used in comparison expressions. Arithmetic expressions are composed of other arithmetic expressions, arithmetic operations, path expressions that evaluate to numeric values, numeric literals, and numeric input parameters

The SELECT clause denotes the query result.

- More than one value may be returned
- The SELECT clause may contain one or more of the following elements: a single range variable or identification variable that ranges over an entity abstract schema type, a single-valued path expression, an aggregate select expression

JPQL Joins

Have seen examples on previous slides

```
// Join publishers and magazines
SELECT pub FROM Publisher pub JOIN pub.magazines mag WHERE
                                                       pub.revenue > 1000000
```

There are many types of joins in SQL, so are there in JPQL

SQL Join: Combines data from two or more tables in a database
JPQL Join: Combines data of (many) objects of two different types

JPQL in NetBeans

Possible to run JPQL from inside NetBeans!

- Mark persistence.xml (in src/main/resources)
- Right click > Run JPQL Query …

JPQL Samples, see lecture code jpa_query

JPA Query API

Interfaces

- **Query**, used to control query execution (unknown result type), avoid!
- **TypedQuery** extends Query, known result type, prefer..!
- **Result**, extracting result from query
- **ResultItem**, item returned from query
- **Parameter**, type for query parameters

JPQL using Query API

Use EM to get a (typed) query object
- Prefer typed queries, supply class object

```
// Single result (result object managed if inside transaction)
Customer customer = em
    .createQuery("select c from Customer c where c.name = 'Olle'",
                                                            Customer.class)
    .getSingleResult();


// Collection
List<Customer> customers = em
    .createQuery("select c from Customer c", Customer.class)
    .setMaxResults(20)
    .getResultList();
```

Query Parameters

```
// Named Parameters
String name = ...;
List<Customer> cs = em.createQuery(
    "select c from Customer c where c.name = :name", Customer.class)
    .setParameter("name", name)
    .getResultList();


// Positional Parameters
String name = ...;
List<Customer> cs = em.createQuery(
    "select c from Customer c where c.name = ?1", Customer.class).
    setParameter(1, name)
    .setMaxResults(20)
    .getResultList();
```

Bulk Update

Applies to entities of a single entity class (together with its subclasses, if any)

- Prefer this to EntityManager if (large) collections of objects...

```
// JPQL
update Customer c set c.city = 'New' where c.city = 'New York'

// Query API
em.getTransaction().begin();
int nAffectedRows = em
    .createQuery("update Customer c set c.city = :newName
                                          where c.city = :oldName")
     .setParameter("newName", "New")
     .setParameter("oldName", "New York")
     .executeUpdate();
em.getTransaction().commit();
```

Bulk Delete

Applies to entities of a single entity class (together with its subclasses, if any)

- Prefer this to EntityManager if (large) collections of objects...

```
// JPQL
delete from DiscountCode dc where dc.discountCode = ‘X’


// Query API
em.getTransaction().begin();
int nAffectedRows = em
    .createQuery("delete from DiscountCode dc where dc.discountCode
              = :dc")
    .setParameter("dc", 'X')
    .executeUpdate();
em.getTransaction().commit();
```

Named Queries

Class annotation on entity classes (only)

```
@NamedQuery(name="MyEntity.findSalaryForNameAndDept",
                 query="select e.salary .... ")
@Entity
public class MyEntity ...{
    @Id
    ...
}

// Usage
  ...em.createNamedQuery(
   "MyEntity.findSalaryForNameAndDept", Customer.class)
```

- Checked by JPA at startup (embedded JPQL runtime)
- Parsed once, then reused (embedded re- parsed over an over)
- NetBeans can auto generate

Embedded JPQL

Embedded JPQL string problematic from design view

- Spelling
- Complexity, looooong strings
- Hard to find would like to have them collected (where some JPQL aficionado can find and optimize, Java programmer normally not database experts)
- Not perfect to have named queries on entities only (often query belong elsewhere)

Solutions?

- No common agreed upon
- Queries in XML, database, ...

Criteria API

JPQL queries are defined as strings
JPA criteria queries defined as objects that represent query elements

Criteria API errors can be detected during compilation (but for many developers string based JPQL queries, which are very similar to SQL queries, are easier to use and understand)

For simple static queries - string based JPQL queries (e.g. as named queries) may be preferred.
For dynamic queries that are built at runtime - the criteria API may be preferred.

Building a dynamic query based on fields that a user fills at runtime in a form that contains many optional fields - is expected to be cleaner when using the JPA criteria API, because it eliminates the need for building the query using many string concatenation operations.

String based JPQL queries and JPA criteria based queries are equivalent in power and in efficiency. Therefore, choosing one method over the other is also a matter of personal preference.

Criteria API

API comprised of a (vast) number of interfaces
- CriteriaBuilder
- Predicate
- Root
- Path
- Join
- From
- Parameter
- … non-optimal API design (understatement…)

An alternative (not typesafe) is EasyCriteria

Criteria API Metadata Model

Criteria API uses a metadata model

- The metamodel is a set of objects that describe your domain model
- A generic way to deal with an application's domain model (used by frameworks)
- Naming: Book.java (model), Book_.java (metadata underscore)
- Normally (!) auto generated from domain objects (NetBeans will (should) generate, ends up in Generated Sources automagically)

Criteria API Metadata Model Sample

File Book_.java (metamodel for Book class)

```
@javax.persistence.metamodel.StaticMetamodel(Book.class)
public class Book_ {
  public static volatile SingularAttribute<Book, Long> id;
  public static volatile SingularAttribute<Book, String> title;
  public static volatile SingularAttribute<Book, Float> price;
  public static volatile SingularAttribute<Book, String>
                                              description;
}
```

JPQL vs Criteria

```
// Query API + JPQL (em  = entitymanager)
Product product = em.createQuery(
                "select p from Product p where p.productId =
                                        980001", Product.class)
                .getSingleResult();


// Query API + CriteriaQuery API
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Product> qProduct = cb.createQuery(Product.class);
Root<Product> rProduct = qProduct.from(Product.class);
qProduct.select(rProduct).where(cb.equal(
                     product.get("productId"), 98001));

// Use query API to execute criteria query
List<Product> prod = em.createQuery(qProduct).getResultList();
```

Note: We still have strings in query, more to come...

Criteria API Dissection

CriteriaBuilder
- A factory for all the individual pieces of the criteria (many..

CriteriaQuery<T>
- Type-safe way to express a query. Functionality specific to top-level queries. T is expected return type (any object type)

Root<T>
- Roots define the basis (type T) from which all joins, paths and attributes are available in the query. In a criteria query, a root is **always an entity**

Path navigation (the '.' in JPQL)
- product.get(...).get(...)... (call return another Path-object)
- Parameter to get: String (bad) or meta data class (good), more to come...

Multiple Roots

Criteria queries may define multiple roots

```
CriteriaQuery query = builder.createQuery();
Root<Person> men = query.from( Person.class );
Root<Person> women = query.from( Person.class );
Predicate menRestriction = builder.and(
    builder.equal( men.get( “gender” ), Gender.MALE ),
    builder.equal( men.get( “relationshipStatus” ), RelationshipStatus.SINGLE )
);
Predicate womenRestriction = builder.and(
    builder.equal( women.get( “gender” ), Gender.FEMALE ),
    builder.equal( women.get( “relationshipStatus” ), RelationshipStatus.SINGLE )
);
query.where( builder.and( menRestriction, womenRestriction ) );
```

Joins

```
// JPQL
select prod.description from PurchaseOrder pu join pu.productId prod
where pu.quantity = 10

// Same expressed as criteria
```

```
// Note: Query just used to get a root
CriteriaQuery<PurchaseOrder> qOrder = cb.createQuery(PurchaseOrder.class);
CriteriaQuery<Product> qProduct = cb.createQuery(Product.class);

Root<PurchaseOrder> rOrder = qOrder.from(PurchaseOrder.class);

Join<PurchaseOrder, Product> orderProductJoin = rOrder.join("productId");

// qProduct says we get a Product, select it from join
qProduct.select(orderProductJoin).where(cb.equal(rOrder.get("quantity"), 10));

// Run query
List<Product> prod = em.createQuery(qProduct)
               .getResultList();
```

A Type Safe Query

```
// Using meta data model
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Book> query =
               builder.createQuery(Book.class);
Root<Book> book = query.from(Book.class);

// Type safe!
Path<String> title = book.get(Book_.title);
Predicate cond = builder.equal(title, "abc");
query.where(cond);

TypedQuery<Book> q = em.createQuery(query);
Book b = q.getSingleResult();
```

Example Queries

```
// Many predicates (greater equal, less equal)
Root<Pub> pub = criteriaQuery.from(Pub.class)
Predicate p1 = criteriaBuilder.ge(pub.get("pint"), arg1);
Predicate p2 = criteriaBuilder.le(pub.get("pint"), arg2)
criteriaQuery.where(criteriaBuilder.and(p1, p2));

// Min, max, avg (getSingleResult)
Expression me = criteriaBuilder.min(pub.get("pint"));
CriteriaQuery<Object> select = criteriaQuery.select(me);

// Order by
CriteriaQuery<Object> select = criteriaQuery.select(from);
    select.orderBy(criteriaBuilder.asc(from.get("pbyte"))
       ,criteriaBuilder.desc(from.get("pint")));
```

JPA and Polymorphism

JPQL/Criteria queries (result) are automatically polymorphic
- Specifying generic class as root

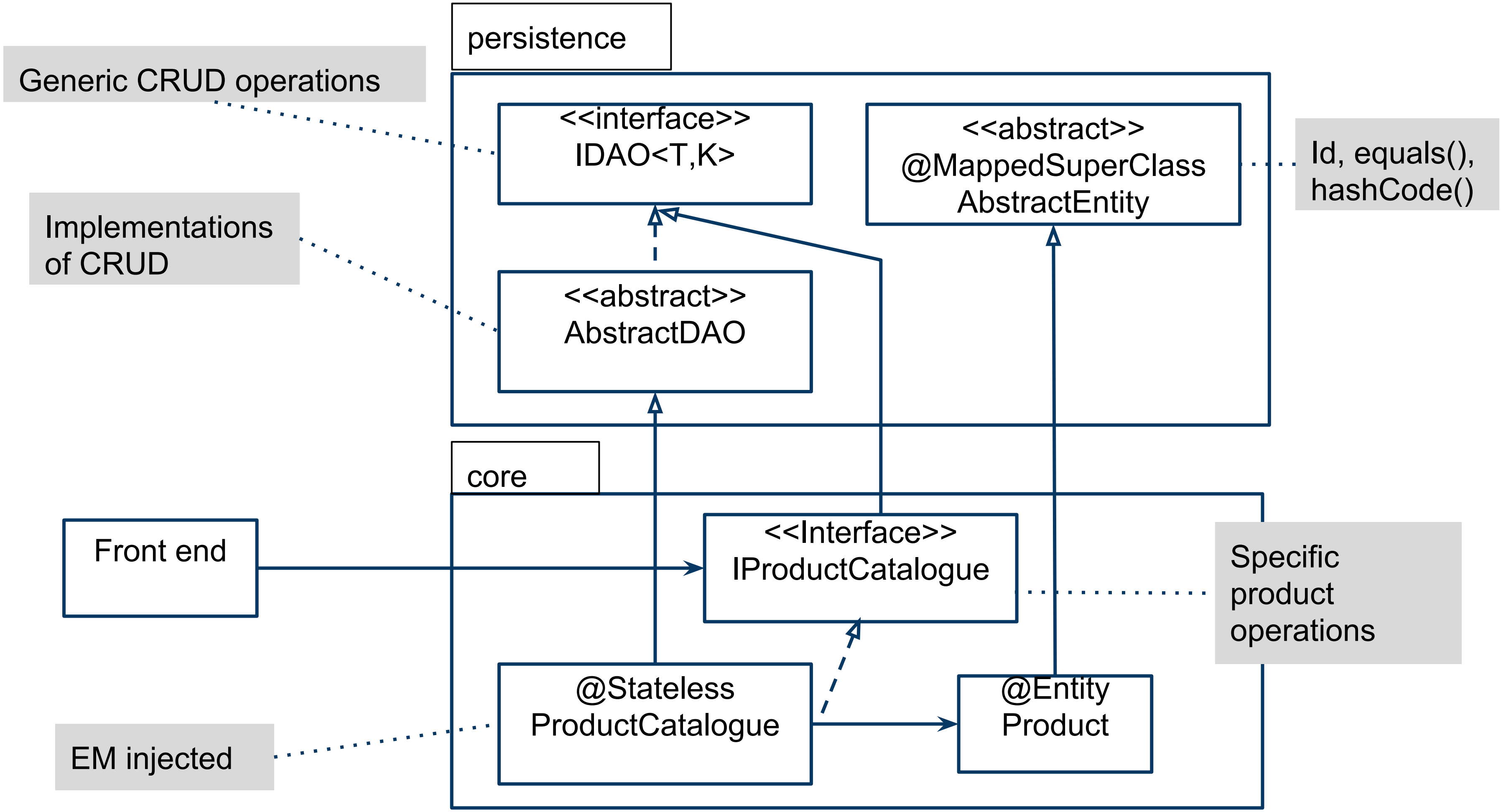
```
// JPQL possible ...
private Class<T> clazz  = …;
String className = clazz.getSimpleName();
Integer count = em
   .createQuery("select count(c) from " + className +
   "c  where c.name = 'Olle'", Integer.class)

// Criteria API, better
private Class<T> clazz  = …;

CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
Root<T> rt = cq.from(clazz);
cq.select(em.getCriteriaBuilder().count(rt));
```

Persistence Layer Design

Data Access Object (DAO). Generic term for object handling entities



CRUD Facade Interface

```
@Local  // Implemented by local EJB
public interface IDAO<T, K> { // T entity type, K key type
    void create(T t);
    void delete(K id);
    void update(T t);
    T find(K id);
    List<T> findAll();
    List<T> findRange(int firstResult, int maxResults);
    int count();
}
```

Domain Driven Design

Many use …

```
// The DAO
Database db = Database.getInstance();
db.save( myOrder );  // Persist
```

The above doesn't seem to be OO (implementation details)
- There should be some domain model class to transparently handle persistence

```
// Much more OO
OrderBook ob = ....getOrderBook();
ob.add( myOrder );  // Transparently persist
```

Ad Hoc Queries

Ad hoc queries, possibly don't have a natural location?

- Create QueryProcessor class or similar to collect unrelated queries?
- Organize from use cases?
- ...

Persistence Layer Testing

Complicated to test persistence layer (EJBs)
- Not possible with JUnit, not an JEE environment …

… Saved by Arquillian (tutorial)
- A testing platform for testing Java enterprise applications by enabling creation of integration, functional, behaviour tests …
- Think: JUnit for JEE
- Will use embedded container
- Possible embedded database (we don't because we wish to inspect tables). JavaBD must be running (but GlassFish mustn't)
- Quite a few test dependencies

Database Backed JEE Security

Previously used a file realm. Database (JDBC) realm better

- Must have database with table for users (some column must be specified as login and password columns and more...)
- Must create a GlassFish (JDBC) realm to connect server and database (use Admin Console)
- See code sample