Detalii JPA

- **JPA (Java Persistence API)** este o specificare care descrie modul de lucru cu datele relationale pentru platformele Java SE si Java EE. Implementarea standart pentru JPA este **EclipseLink.** Ultima versiune JPA 2.1 : EclipseLink, Hibernate, etc

Exista trei aspecte legate de JPA:

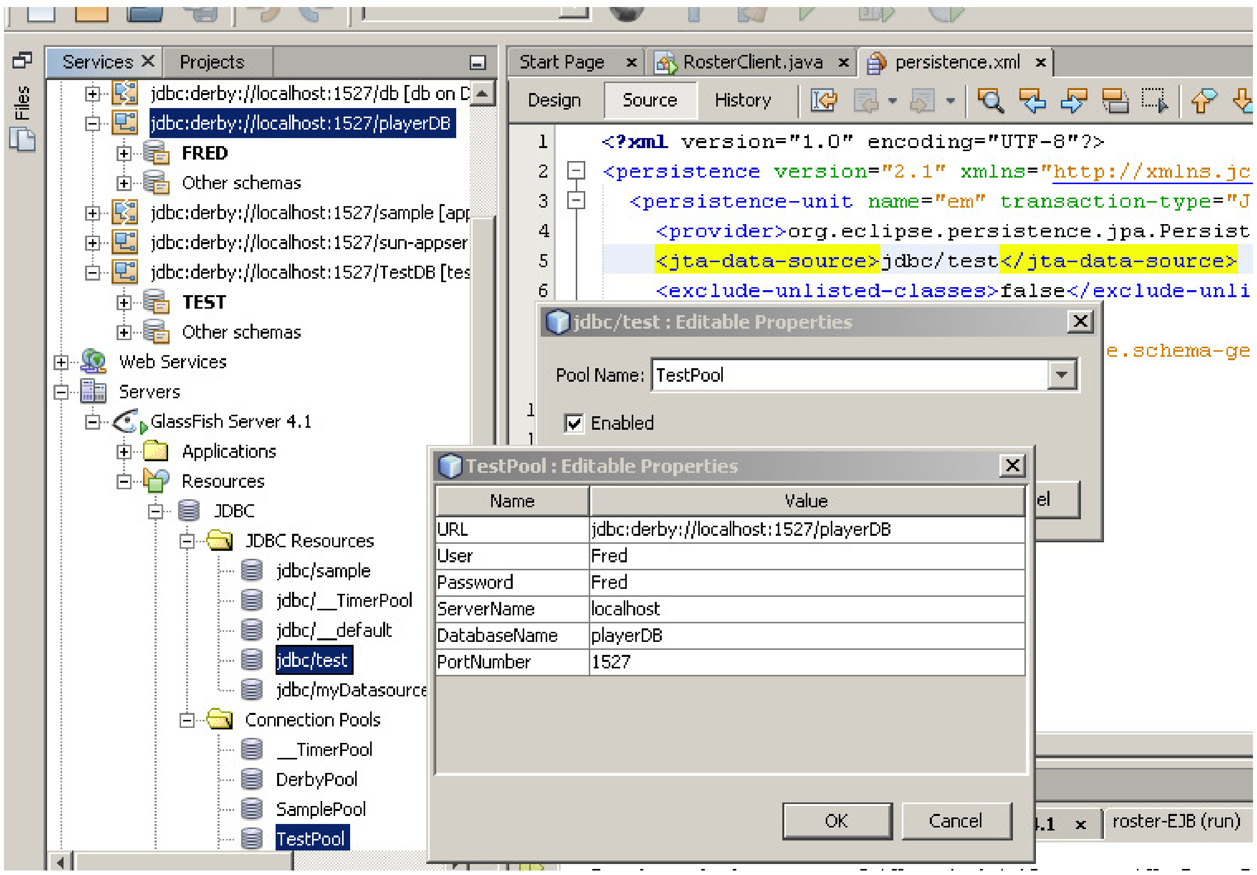
- Interfata propriuzisa de programare (API-ul) - se gaseste in package-ul " javax.persistence"
- JPQL(Java Persistence Query Language)- limbaj objec-oriented de interogare a bazelor de date relationare in care sunt memorate "entity"-urile (clase Java, adnotate cu @Entity. Instantele acestor clase se memoreaza in tabelele din RDB)
- Objec-relational metadata- relatii intre entitati, exprimate cu ajutorul adnotarilor si/sau fisiere xml.

Proiectele care utilizeaza JPA trebuie sa contina un fisier de configurare "persistence.xml". In exemplul nostru(aplicatia roster), se defineste in acest fisier o unitate de persistenta cu numele "em" (in cazul nostru nu a fost necesar sa facem referire la unit prin acest nume deoarece nu exista decat un singur unit). Atributul "transaction-type" este JTA daca aplicatia ruleaza pe serverul JavaEE si RESOUCE_LOCAL pentru aplicatii JavaSE(laboratorul 4)- http://www.adam-bien.com/roller/abien/entry/don_t_use_jpa_s

In sectiunea <**jta-data-source>** se specifica baza de date in care sunt tabelele aplicatiei. In cadrul precedentului laborator am utilizat resursa per-configurata "jdbc/__default" pentru a identifica "Data Source". In poza de mai jos se arata cat de usor se modifica DataSource. In loc de jdbc/__default, doresc sa folosesc alta resursa (creata administrativ) cu numele "jdbc/test":

- modific in persistence.xml : <jta-data-source>jdbc/test</jta-data-source>
- dupa cum se vede in poza, in GlassFish/Resources/JDBC/JDBC Resources, daca fac click-dreapta pe jdbc/test si Properties apare fereastra din care aflu numele "pool"-ului de conexiuni utilizat de catre jdbc/test: aici este "TestPool"
- la "Connections Pool" fac click-dreapta pe "TestPool" si apare fereastra cu informatiile ce identifica baza de date la care optin o conexiune: "playerDB"(am selectat-o in partea stanga sus).

OBSERVATIE: crearea unei noi resurse (ex:jdbc/test) se face simplu dar nu are rost sa pierdem timp cu asta acum ! Folositi jdbc/__default sau jdbc/sample.



```
jdbc:derby://localhost:1527/db [db on D
jdbc:derby://localhost:1527/playerDB
    FRED
    Other schemas
jdbc:derby://localhost:1527/sample [app
jdbc:derby://localhost:1527/sun-appser
jdbc:derby://localhost:1527/TestDB [tes
    TEST
    Other schemas
Web Services
Servers
  GlassFish Server 4.1
    Applications
    Resources
      JDBC
        JDBC Resources
          jdbc/sample
          jdbc/__TimerPool
          jdbc/__default
          jdbc/test
          jdbc/myDatasource
        Connection Pools
          __TimerPool
          DerbyPool
          SamplePool
          TestPool
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jc
  <persistence-unit name="em" transaction-type="J
    <provider>org.eclipse.persistence.jpa.Persist
    <jta-data-source>jdbc/test</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unli
```

jdbc/test : Editable Properties

Pool Name: TestPool

[✓] Enabled

TestPool : Editable Properties

Name	Value
URL	jdbc:derby://localhost:1527/playerDB
User	Fred
Password	Fred
ServerName	localhost
DatabaseName	playerDB
PortNumber	1527

- ## Adnotarile din clasele entity

Modul de utilizare a adnotarilor este descris foarte concis in documentul: Essential object-relational mapping in JPA.pdf

Rezultatul adnotarilor:

CREATE TABLE EJB_ROSTER_PLAYER (ID VARCHAR(255) NOT NULL, NAME VARCHAR(255), POSITION VARCHAR(255), SALARY FLOAT, PRIMARY KEY (ID));

CREATE TABLE EJB_ROSTER_LEAGUE (ID VARCHAR(255) NOT NULL, DTYPE VARCHAR(31), NAME VARCHAR(255), SPORT VARCHAR(255), PRIMARY KEY (ID));

CREATE TABLE EJB_ROSTER_TEAM (ID VARCHAR(255) NOT NULL, CITY VARCHAR(255), NAME VARCHAR(255), LEAGUE_ID VARCHAR(255), PRIMARY KEY (ID));

CREATE TABLE EJB_ROSTER_TEAM_PLAYER (PLAYER_ID VARCHAR(255) NOT NULL, TEAM_ID VARCHAR(255) NOT NULL, PRIMARY KEY (PLAYER_ID, TEAM_ID));

ALTER TABLE EJB_ROSTER_TEAM ADD CONSTRAINT fk_TeamToLeague FOREIGN KEY (LEAGUE_ID) REFERENCES EJB_ROSTER_LEAGUE (ID);

ALTER TABLE EJB_ROSTER_TEAM_PLAYER ADD CONSTRAINT fk_TP_TeamID FOREIGN KEY (TEAM_ID) REFERENCES EJB_ROSTER_TEAM (ID);

ALTER TABLE EJB_ROSTER_TEAM_PLAYER ADD CONSTRAINT fk_TP_PlayerID FOREIGN KEY (PLAYER_ID) REFERENCES EJB_ROSTER_PLAYER (ID);

Ce este in spatele **JPA-ului** ? : http://tomee.apache.org/jpa-concepts.html .

Pe scurt:

Persistence Context is the in-memory cache of entity objects maintained by an Entity Manager

Here's a quick cheat sheet of the JPA world:

- A **Cache** is a **copy of data**, copy meaning pulled from but living outside the database.
- **Flushing** a Cache is the act of putting modified data back into the database.
- A **PersistenceContext** is essentially a Cache. It also tends to have it's own non-shared database connection.
- An **EntityManager** represents a PersistenceContext (and therefore a Cache)
- An **EntityManagerFactory** creates an EntityManager (and therefore a PersistenceContext/Cache)

Comparing `RESOURCE_LOCAL` and `JTA` persistence contexts

With <persistence-unit transaction-type="**RESOURCE_LOCAL**"> **you** are responsible for EntityManager (PersistenceContext/Cache) creating and tracking...

- You **must** use the **EntityManagerFactory** to get an EntityManager
- The resulting **EntityManager** instance **is** a PersistenceContext/Cache
- An **EntityManagerFactory** can be injected via the **@PersistenceUnit** annotation only (not @PersistenceContext)
- You are **not** allowed to use @PersistenceContext to refer to a unit of type RESOURCE_LOCAL
- You **must** use the **EntityTransaction** API to begin/commit around **every** call to your EntityManger
- Calling entityManagerFactory.createEntityManager() twice results in **two** separate EntityManager instances and therefor **two** separate PersistenceContexts/Caches.
- It is **almost never** a good idea to have more than one **instance** of an EntityManager in use (don't create a second one unless you've destroyed the first)

With <persistence-unit transaction-type="**JTA**"> the **container** will do EntityManager (PersistenceContext/Cache) creating and tracking...

- You **cannot** use the **EntityManagerFactory** to get an EntityManager
- You can only get an **EntityManager** supplied by the **container**
- An **EntityManager** can be injected via the **@PersistenceContext** annotation only (not @PersistenceUnit)
- You are **not** allowed to use @PersistenceUnit to refer to a unit of type JTA
- The **EntityManager** given by the container is a **reference** to the PersistenceContext/Cache associated with a JTA Transaction.
- If no JTA transaction is in progress, the EntityManager **cannot be used** because there is no PersistenceContext/Cache.
- Everyone with an EntityManager reference to the **same unit** in the **same transaction** will automatically have a reference to the **same PersistenceContext/Cache**
- The PersistenceContext/Cache is **flushed** and cleared at JTA **commit** time

Cache == PersistenceContext

The concept of a database cache is an extremely important concept to be aware of. Without a copy of the data in memory (i.e. a cache) when you call account.getBalance() the persistence provider would have to go read the value from the database. Calling account.getBalance() several times would cause several trips to the database. This would obviously be a big waste of resources. The other side of having a cache is that when you call account.setBalance(5000) it also doesn't hit the database (usually). When the cache is "flushed" the data in it is sent to the database via as many SQL updates, inserts and deletes as are required. That is the basics of java persistence of any kind all wrapped in a nutshell. If you can understand that, you're good to go in nearly any persistence technology java has to offer.

Complications can arise when there is more than one PersistenceContext/Cache relating the same data in the same transaction. In any given transaction you want exactly one PersistenceContext/Cache for a given set of data. Using a JTA unit with an EntityManager created by the container will always guarantee that this is the case. With a RESOURCE_LOCAL unit and an EntityManagerFactory you should create and use exactly one EntityManager instance in your transaction to ensure there is only one active PersistenceContext/Cache for the given set of data active against the current transaction.

Caches and Detaching

Detaching is the concept of a persistent object **leaving** the PersistenceContext/Cache. Leaving means that any updates made to the object are **not** reflected in the PersistenceContext/Cache. An object will become Detached if it somehow **lives longer** or is **used outside** the scope of the PersistenceContext/Cache.
For a JTA unit, the PersistenceContext/Cache will live as long as the transaction does. When a transaction completes (commits or rollsback) all objects that were in the PersistenceContext/Cache are Detached. You can still use them, but they are no longer associated with a PersistenceContext/Cache and modifications on them will **not** be reflected in a PersistenceContext/Cache and therefore not the database either.
Serializing objects that are currently in a PersistenceContext/Cache will also cause them to Detach.

In some cases objects or collections of objects that become Detached may not have all the data you need. This can be because of lazy loading. With lazy loading, data isn't pulled from the database and into the PersistenceContext/Cache until it is requested in code. In many cases the Collections of persistent objects returned from an javax.persistence.Query.getResultList() call are completely empty until you iterate over them. A side effect of this is that if the Collection becomes Detached before it's been fully read it will be permanently empty and of no use and calling methods on the Detached Collection can cause strange errors and exceptions to be thrown. If you wish to Detach a Collection of persistent objects it is always a good idea to iterate over the Collection at least once.

You **cannot** call EntityManager.persist() or EntityManager.remove() on a Detached object.
Calling EntityManager.merge() will re-attach a Detached object.