1. Api "Criteria" de JPA 2

Apports et inconvénients de l'API "Criteria"

L'api "Criteria" (qui est apparue avec la version 2 de JPA) correspond à une syntaxe alternative à JQPL pour exprimer et déclencher des requêtes retournant des entités persistantes "JPA" .

Contrairement à JPQL , l'api des criteria ne s'appuie plus sur un dérivé de SQL mais s'appuie sur **une série d'expression de critères de recherche qui sont entièrement construits via des méthodes "java" .**

L'api "Criteria" est plus "orientée objet" et plus "fortement typée" que JPQL.

Le **principal avantage** réside dans **la détection de certaines erreurs dès la phase de compilation**.

A l'inverse, la plupart des erreurs dans l'expression d'une requête JPQL ne sont en général détectées que lors de l'exécution des tests unitaires.

Du coté négatif , **l'API "Criteria" de JPA n'est syntaxiquement pas très simple** (voir franchement complexe) et **il faut un certain temps pour s'y habituer (le code est moins "lisible" que JPQL au premier abord)** .

Exemple basique (pour la syntaxe et le déclenchement)

```
public List<Devise> getAllDevise() {
	/*return  this.entityManager.createQuery(
		"select d from Devise d",Devise.class).getResultList();*/

	//construction:
	CriteriaBuilder cb = entityManager.getCriteriaBuilder();
	CriteriaQuery<Devise> criteriaQuery = cb.createQuery(Devise.class);

	//préparation (expression des critères de recherche)
	Root<Devise> deviseRoot = criteriaQuery.from(Devise.class);
	criteriaQuery.select(deviseRoot);

	//déclechement (assez semblable) à celui d'une requête JPQL :
	return this.entityManager.createQuery(criteriaQuery).getResultList();
	}
```

Exemple avec prédicat pour partie ".where"

```
public Devise getDeviseByName(final String deviseName) {
      /* return  this.entityManager.createQuery(
            "select d from Devise d where d.monnaie = :deviseName",
Devise.class)
            .setParameter("deviseName",deviseName)
            .getSingleResult();*/
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Devise> criteriaQuery = cb.createQuery(Devise.class);

Root<Devise> deviseRoot = criteriaQuery.from(Devise.class);

Predicate pEqMonnaie = cb.equal(deviseRoot.get("monnaie") , deviseName);

criteriaQuery.select(deviseRoot);
criteriaQuery.where(pEqMonnaie);

return this.entityManager.createQuery(criteriaQuery).getSingleResult();
}
```

Exemple relativement simple avec jointure:

```
public List<Compte> findComptesByNumCli(long numCli) {
/*
return this.getEntityManager().createQuery("SELECT cpt FROM
Client cli  JOIN cli.comptes cpt WHERE cli.numero= :numCli",Compte.class)
.setParameter("numCli", numCli)
.getResultList();
*/
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Compte> criteriaQuery = cb.createQuery(Compte.class);

Root<Client> clientRoot = criteriaQuery.from(Client.class);


Predicate pEqNumCli = cb.equal(clientRoot.get("numero") , numCli);
//Predicate pEqNumCli = cb.equal(clientRoot.get(Client_.numero) , numCli);

Join<Client, Compte> joinComptesOfClient =  clientRoot.join("comptes");
//Join<Client, Compte> joinComptesOfClient =  clientRoot.join(Client_.comptes);


criteriaQuery.select(joinComptesOfClient);
criteriaQuery.where(pEqNumCli);

return this.entityManager.createQuery(criteriaQuery).getResultList();
}
```

NB: L'exemple précédent n'est pas "fortement typé" et certaines erreurs ("types incompatibles , "propriété n'existant pas ou mal orthographiée") peuvent passées inaperçues lors de la phase de compilation.

Classes "modèles" pour des expressions rigoureuses des critères de recherche :

```
...

//Predicate pEqNumCli = cb.equal(clientRoot.get("numero") , numCli);
Predicate pEqNumCli = cb.equal(clientRoot.get(Client_.numero) , numCli);

//Join<Client, Compte> joinComptesOfClient =  clientRoot.join("comptes");
Join<Client, Compte> joinComptesOfClient =  clientRoot.join(Client_.comptes);


....
```

Au sein de cette écriture beaucoup plus rigoureuse ,

Client_.numero sera interprété comme une propriété ".numero" qui existe bien sur une entité de type "Compte" et qui de plus est de type long .

Ces informations sont issue d'une classe java "modèle" dont le nom se termine (par convention) par un caractère "underscore" et dont le début coïncide avec la classe de l'entité persistante décrite.

(La classe modèle "Client_" décrit la structure de la classe "Client" préfixée par @Entity).

Une classe modèle doit absolument être placée dans le même package java que la classe d'entité persistante à décrire. On pourra cependant paramétrer maven et/ou l'IDE eclipse de façon à placer les classes "modèles" dans un autre répertoire car celles-ci sont sont générées automatiquement.

Exemple de classe "modèle"

```
package tp.myapp.minibank.core.entity;

import javax.annotation.Generated;
import javax.persistence.metamodel.ListAttribute;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;

@Generated(value = "org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Client.class)
public abstract class Client_ {

	public static volatile SingularAttribute<Client, String> password;
	public static volatile SingularAttribute<Client, Long> numero;
	public static volatile ListAttribute<Client, Compte> comptes;
	public static volatile SingularAttribute<Client, Adresse> adresse;
	public static volatile SingularAttribute<Client, String> telephone;
	public static volatile SingularAttribute<Client, String> nom;
	public static volatile SingularAttribute<Client, String> prenom;
	public static volatile SingularAttribute<Client, String> email;
}
```

NB : La classe modèle _Client a été générée via un plugin maven "hibernate-jpamodelgen" (ou autre) qui peut se déclencher au moment de la compilation des classes ordinaires "Client" , …

…/...

```
<dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-jpamodelgen</artifactId>
      <version>4.3.1.Final</version>
      <scope>provided</scope>
</dependency>

...

<plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
            <source>1.7</source>
            <target>1.7</target>
            <compilerArguments>
                  <processor>
                  org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor
                  </processor>
            </compilerArguments>
        </configuration>
</plugin>
```

NB : il existe une autre façon de déclencher **`org.hibernate.jpamodelgen (plus paramétrable) .`**

Les classes modèles "XXX_" et les packages java sont par défaut générés dans le répertoire "**`target/generated-sources/annotations`**"

Il faut ensuite paramétrer le projet eclipse pour qu'il voit les classes générées :

1) activer le menu contextuel "**properties / compiler / annotation-processing**"
depuis le projet eclipse "….-ejb"
2) **enable specific** / "**target/generated-sources/annotations**"
3) ajouter une utilisation de XXX_.yyy dans le code du DAO