TP JEE (2)
Entreprise Java Beans

L'objectif de ce TP est d'utiliser une base de données pour persister les Contacts d'un utilisateur. Nous utiliserons ici deux bases de données différentes : une pour l'application et une pour les tests d'intégration. Nous utiliserons la spécification JEE JPA (Java Persistence API). Chaque base de données est de type H2, une base de données intégrée à JBOSS.

Dans ce TP, nous nous baserons sur le projet créé au TP précédent. Ainsi, après avoir nettoyé le projet (mvn clean), copiez-coller le dossier tp1-web/ContactWeb vers tp2/ContactWeb et ajoutez ce nouveau dossier au dépôt mercurial (attention aux fichiers autogénérés). Générez un projet Eclipse et ouvrez-le.

Table des matières

1 Les Datasources

Pour JPA, un datasource représente une base de données. Ainsi, il est nécessaire de créer un datasource pour chaque base de données que nous utiliserons. Une application spécifiera alors quel datasource JPA doit utiliser à son déploiement. Si le datasource n'est pas déclaré, l'application ne sera pas déployée.

Nous utiliserons deux méthodes différentes pour créer un datasource. La première consiste à le déclarer au sein des fichiers de configuration de JBOSS. La seconde permet de déclarer un datasource qui sera déployé en même temps que l'application.

1.1 La base de données utilisateur

La base de données utilisateur sera déclarée par la définition d'un datasource directement au sein de JBOSS :

1. ouvrez le fichier $JBOSS_HOME/standalone/configuration/standalone.xml
2. trouvez la section datasources
3. ajoutez un nouveau datasource en vous inspirant de l'extrait de code suivant

```
<datasource jndi-name="java:jboss/datasources/ContactDS"
    pool-name="ExampleDS" enabled="true" use-java-context="
    true">
        <connection-url>jdbc:h2:mem:contact;DB_CLOSE_DELAY
            =-1</connection-url>
     <driver>h2</driver>
     <security>
        <user-name>sa</user-name>
        <password>sa</password>
     </security>
</datasource>
```

4. Re-démarez JBoss et assurez vous que le nouveau datasource est bien déclaré.

Il est maintenant nécessaire de spécifier quel datasource nous allons utiliser dans l'application. Pour cela il est nécessaire de créer un fichier /src/main/resources/META-INF/persistence.xml avec le contenu suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http
      ://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
        http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.
           xsd">
   <persistence-unit name="primary">
            <jta-data-source>java:jboss/datasources/ContactDS</
               jta-data-source>
                <properties>
                        <property name="hibernate.hbm2ddl.auto"
                           value="update"/>
                        <property name="hibernate.show_sql" value
                           ="true" />
```

```
            </properties>
        </persistence-unit>
</persistence>
```

La propriété hibernate.hbm2ddl.auto permet de dire à JBoss si le schéma SQL de la base de données sera créé, supprimé ou modifié au déploiement de l'application. Ici nous utilisons update : le schéma sera créé une fois puis modifié si besoin.

1. Créez le fichier /src/java/main/resources/META-INF/persistence.xml.

1.2 La base de données de test

Le datasource correspondant à la base de données de test qui sera déclarée dans l'application elle-même. Pour cela, nous allons créer un fichier src/test/resources/test-ds.xml contenant la définition du datasource :

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources xmlns="http://www.jboss.org/ironjacamar/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jboss.org/ironjacamar/schema
        http://docs.jboss.org/ironjacamar/schema/datasources_1_0.
        xsd">
    <datasource jndi-name="java:jboss/datasources/ContactTestDS"
        pool-name="contact-test" enabled="true"
        use-java-context="true">
        <connection-url>jdbc:h2:mem:contact-test;DB_CLOSE_DELAY
            =-1</connection-url>
        <driver>h2</driver>
        <security>
            <user-name>sa</user-name>
            <password>sa</password>
        </security>
    </datasource>
</datasources>
```

Il est ensuite de nécessaire de créer un fichier test-persistence.xml permettant de spécifier que les tests utiliseront ce datasource. Nous allons créer ce fichier dans : src/test/resources/META-INF, avec le contenu suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http
        ://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
         http://java.sun.com/xml/ns/persistence
         http://java.sun.com/xml/ns/persistence/persistence_2_0.
            xsd">
```

```
    <persistence-unit name="primary">
        <jta-data-source>java:jboss/datasources/ContactTestDS</jta-
            data-source>
        <properties>
            <!-- Properties for Hibernate -->
            <property name="hibernate.hbm2ddl.auto" value="create-
                drop" />
            <property name="hibernate.show_sql" value="true" />
        </properties>
    </persistence-unit>
</persistence>
```

Remarquez que cette fois-ci, le schéma sera créé et supprimée à chaque déploiement des tests.

Il ne nous reste plus qu'à spécifier à Arquillian (le framework de tests d'intégration) qu'il est nécessaire de déployer ces deux fichiers lors des tests. Pour cela nous utiliserons les méthodes addAsResource("CheminDuFichier","CheminDuFichierDansArchive") et addAsWebInfResource("leDataSource").

1. En vous inspirant du code suivant, modifiez l'ensemble des tests d'intégration qui necessecite une base de donnée :

```
.....
return ShrinkWrap
        .create(WebArchive.class, "test.war")
        .addClasses(Contact.class, ContactService.class,
            Resources.class)
        ........
        .addAsResource("META-INF/test-persistence.xml","META
            -INF/persistence.xml")
        .....
        .addAsWebInfResource("test-ds.xml")
        .....
```

2 Développement de l'EJB Entity : "Contact"

2.1 Introduction

Les EJB Entity : il s'agit de classes qui représentent les objets persistés dans une base de données. Un Entity Bean est une classe annotée par *@Entity* avec un identifiant annoté par *@Id*. L'ensemble des attributs à persister en base sont annotés par des annotations spécifiques : @Temporal, @Basic, @OneToMany, @ManyToOne, etc. À travers ces annotations, une classe Java et appariée à une ou plusieurs tables. Par exemple la classe suivante sera associée à une table Personne [id, email, prenom].

```
[...]
@Entity
@Table( name = "Personne" )
public class Personne implements Serializable {

        @Id
        @GeneratedValue (strategy=GenerationType.AUTO)
        private Integer id;

        @Basic
        private String email;

        @Basic
        private String prenom;
}
```

Questions

1. Modifiez la classe Contact pour la "transformer" en entity bean.
2. Expliquer chacune des annotations Java présentes dans la classe Contact.

3 Développement de l'EJB Session : ContactService

3.1 Les EJBs Session

Les EJBs session sont des classes Java (POJO) annotées et managées par le conteneur. Un EJB Session propose, de base, un ensemble de fonctionnalités : thread-safe, transactions avec la base de données, etc. Il existe plusieurs types d'EJB Session : les EJB Stateful, les EJB Stateless et les EJB Singleton. Dans ce TP, nous utiliserons des EJB Stateless.

1. Qu'est ce qu'un EJB Stateless ?
2. Qu'est ce qu'un EJB Stateful ?
3. Quels sont les cas d'usages de ces deux types d'EJBs ?

Pour transformer une classe en EJB Stateless il suffit (JEE6) d'ajouter l'annotation @Stateless sur une classe.

1. Transformez la classe ContactService en EJB Stateless.
2. Pourquoi est ce que les tests ne passent pas ? Corrigez.

3.2 EntityManager et JPAQL

Les accès (lecture, écriture, modification, requêtes SQL) avec JPA se font via une classe appelée EntityManager. Pour cela, la classe propose un ensemble de méthodes, comme par exemple :

- em.persist(o), pour enregistrer un nouvel objet o.
- em.find(UneClasse.class, id), pour récupérer un objet par son identifiant.
- em.merge(o), pour mettre à jour un objet o.
- em.remove(o), pour supprimer un objet o. Attention : l'objet passé en paramètre doit être managé (récupéré par la méthode find ou createQuery).
- em.createQuery(query, UneClasse.class), pour exécuter une requêtes JPAQL sur les entités d'une classe.

La classe EntityManager doit être injectés par le conteneur d'EJB de la façon suivante :

```
@PersistenceContext
EntityManager em;
```

JPAQL est un langage de requêtes orienté objet indépendant de la base de données (MySQL, H2, etc.). Voici quelques exemples de requêtes :

```
//Selection de tous les personnes d'une base.
String query = "select p from Personne p";
Collection<Personne> contacts = em.createQuery(query, Personne.
    class).getResultList();


/// Selection de tout les auteurs ayant pour nom Adam Bien.
String query = "select a from Author a, where a.name = 'Adam Bien
    '";
Author adam = em.createQuery(query, Author.class).getSingleResult
    ();
```

Il est aussi possible de passer des paramètres à une requête :

```
// Selection de tous les auteurs ayant un nom dans la liste names
    en parametre
List<String> names = ...
String query = "select a from Author where a.name in :names";
List<Contact> authors = em.createQuery(query, Author.class).
    setParameter("names", names).getResultList();
```

Exercice : Modifiez votre application afin d'utiliser la base de données pour la plupart des services proposés par la classe ContactService. **Attention :** n'oubliez

pas que maintenant l'identifiant de la classe Contact est automatiquement généré par JPA. L'ajout d'un contact en base avec un identifiant qui serait différent de null générerait des exceptions.