TP Spring & Hibernate

Partie 0 : Récupérer l’application et la lancer

Récupérer l’application Java EE / JSP / JDBC fournie par l’encadrant.

Cette application est une application Java EE standard, utilisant :

- des servlets pour gérer les interactions avec l’utilisateur (requêtes HTTP)
- des vues JSP générées par les servlets
- Des données (stockées par défaut dans une base de données h2 en mémoire, recréée à chaque démarrage de l’application) accédées et mises à jour depuis les servlets au travers d’une abstraction JDBC.

L’utilisation utilise gradle comme gestionnaire de dépendances, outil de build et lanceur d’application.

Pour démarrer l’application, il faut :

1. Se placer dans le répertoire de l’application
2. lancer la commande : **gradlew appRun**

Cette commande va lancer un container Tomcat embarqué et déployer automatiquement l’application dans celui-ci.

L’application est alors accessible à l’URL : http://localhost:8080/spring_hibernate_tp/

L’application possède plusieurs fonctionalités rendues disponibles à l’utilisateur à travers la couche de présentation en utilisant des servlets :

- La création, l’édition, la suppression et le listing de produits à travers les servlets associées (createProductServlet, …)
- Le login d’un utilisateur (LoginServlet)
- L’affichage de l’utilisateur courant (UserInfoServlet)

L’utilisateur courant est stockée à travers un cookie (CookieFilter)

Toutes ces fonctionalités sont regroupées dans le package org.epsi.b3.simplewebapp.web

Cette couche de présentation permet une interaction avec la couche métier proposant 2 fonctionalités :

- La gestion de comptes utilisateurs (package org.epsi.b3.simplewebapp.users)
- La gestion de produits (package org.epsi.b3.simplewebapp.products)

La persistance des donnérs de la couche métier en base de données est assurée dans le package org.epsi.b3.simplewebapp.db

Partie I : Hibernate

La première partie du TP consiste à faire évoluer la partie existante de la couche de persistance vers Hibernate.

Pour cela, il va falloir :

1. Définir les entités métier à persister
2. Définir et implémenter un mapping objet-relationnel à appliquer
3. Définir et implémenter le cycle de vie des sessions
4. Configurer Hibernate pour gérer les connections avec la base de données et pour instancier les mappings
5. Réaliser les appels à la base de données au travers de Hibernate

I.1. Choix des entités métier à persister

Analyser l'application actuelle pour trouver quelles sont les entités à persister en base de donnée, et à quel moment cela est réalisé.

I.2. Mapping objet-relationnel

Une fois les entitées à persister définies, il faut maintenant réaliser un design Objet-Relationnel pour définir quels sont les attributs à persister et comment. Implémenter ensuite ce schéma, **soit** par l'utilisation d'un fichier xml **soit** par l'utilisation d'annotations (les 2 méthodes sont équivalentes).
Avant toute chose, décommenter dans le fichier build.gradle la ligne correspondant à hibernate afin de rajouter les dépendances dans le projet.

Attention : dans le cas d'un fichier XML, celui-ci devra être placé dans le répertoire src/main/resources de l'application pour être lu. A cause de cette contrainte, il faudra donc spécifier le nom complet de la classe PRECEDE DU PACKAGE dans le champ <class name=...>

Exemple d'un fichier XML MyEntity.hbm.xml situé dans le même package que la classe MyEntity :

```
<hibernate-mapping>

    <class name="org.myPackage.MyEntity"
table=”MY_ENTITY”>

        <id name="id">

            <generator class="native"/>

         </id>

         <property name="myString" type="string"
not-null=”true” />

    </class>

</hibernate-mapping>
```

Exemple d’une classe MyEntity annotée :

```
@Entity
@Table(name = "MY_ENTITY")
public class MyEntity {
    private Integer myId;
    private String myString;

    @Id
    public Integer getMyId() {          return myId;     }
    @Basic(optional = false)
    public String getMyString() {    return myString;     }
```

I.3. Cycle de vie des sessions

Rappel : tout dialogue avec la base de données doit être réalisé dans une transaction !
Avant de déléguer le contrôle de la base de données à Hibernate, il faut donc architecturer le cycle de vie d’une Session (quand la créer, quand la libérer) ainsi que celui de la SessionFactory.

Note : le commit ou rollback d’une transaction ferment en général la session, sauf dans des architectures où la session a un cycle de vie un peu spécial.
Pour plus d’information :
https://docs.jboss.org/hibernate/core/3.3/reference/en/html/transactions.html#transactions-basics-uow

Demander à l’encadrant de valider le choix d’architecture retenu pour le cycle de vie des sessions.

I.4. Configuration Hibernate

I.4.1 Fichier de configuration

Hibernate se configure à l’aide d’un fichier hibernate.cfg.xml à placer dans src/main/resources

Ce fichier contient à la fois la configuration de la connexion à la BD, le cycle de vie des sessions, et les options d’hibernate (cache, …) mais également les références vers les différents mappings à importer !

Un exemple de configuration est fourni dans le répertoire resourcesEtudiants à la racine du projet.

Pour chaque mapping utilisant des annotations, ajouter une ligne de la forme :

```
<!-- Names the annotated entity class -->
<mapping class="org.epsi.b3.MyEntity"/>
```

Pour chaque mapping utilisant un fichier XML, ajouter une ligne de la forme :

```
<!-- Names the XML mappings -->
<mapping resource="MyEntity.hbm.xml"/>
```

I.4.2 Génération de la SessionFactory

Nous allons créer une instance de SessionFactory, qui va nous servir de point d’entrée pour utiliser Hibernate.
On pourra utiliser le code suivant :

```
new Configuration().configure().buildSessionFactory()
```

Attention à bien respecter le cycle de vie de la SessionFactory ! Utiliser un design pattern adéquat pour partager la SessionFactory dans les classes de votre code lorsque nécessaire.

I.5. Transaction BD déléguée à Hibernate

Il est maintenant temps de supprimer le code JDBC faisant les requêtes en base de donnée pour utiliser des méthodes ORM (Object-Relational Mapping) directement.
On pourra utiliser les méthodes save(), update(), delete() de la classe Session pour effectuer des opérations de modification sur les données persistées et les queries de la classe Session pour lister les données persistées.

Par exemple, pour persister un objet myEntity en base, on pourra utiliser :

```
final Transaction transaction =
currentSession.beginTransaction();
try {
    currentSession.save(myEntity);
    transaction.commit();
} catch (RuntimeException e) {
    transaction.rollback();
}
```

ATTENTION : FAIRE UN PREMIER RENDU DU PROJET APRES LA PARTIE I

Partie II : Spring Framework et application multi-couche

Afin de faciliter l'architecture et la configuration de l'application, nous allons déléguer à Spring un certain nombre de comportements.
Nous allons également réécrire une partie de l'application pour la rendre multi-couches

II.1 Génération d'un ApplicationContext

Spring stocke l'ensemble de ses configurations, ses composants, … sous forme de Bean (par défaut, une instance unique de chaque classe). Tous ces beans sont rattachés à un singleton particulier appelé ApplicationContext.

Il existe de nombreuses manières de générer un ApplicationContext, nous allons nous contenter de l'instance par défaut qui gère une application par annotations :

```
ApplicationContext myApplicationContext = new
AnnotationConfigApplicationContext("org.epsi");
```

Attention : cet objet doit donc être un singleton dans votre application, et être disponible à chaque fois que nécessaire !

II.2 Injection de dépendances - notions

Spring utilise massivement l'injection de dépendances pour fournir une implémentation d'une interface lorsqu'elle est nécessaire.

Afin d'enregistrer une classe comme composant Spring et comme candidat à être injecté dans une autre classe, on utilisera :

- l'annotation @Component (ou les annotations qui en dérivent comme @Service, …) sur le prototype d'une classe

- ou l’annotation @Bean sur une méthode générant une instance de cette classe

Afin d’injecter une instance de classe ou une implémentation d’interface, on utilisera l’annotation @Autowired sur l’objet à instancier, par exemple :

```
@Component
public class MaDependance {
 [...]
}
```

```
@Component
public class MaClasseAvecDependance {

    @Autowired
    private MaDependance maDependance;

}
```

Si MaClasseAvecDependance est récupérée depuis le Spring contexte (voir note ci-dessous), alors maDependance sera automatiquement instanciée et injecté dans cette classe – il n’est donc plus nécessaire de setter maDependance

Il existe également de nombreuses autres méthodes permettant de définir et d’injecter les dépendances : fichier XML, constructeur, setter, ...

Note sur l’injection de dépendances

Attention : Pour profiter de l'injection de dépendances, il faut laisser Spring "prendre la main" sur ses composants. Cela veut dire qu'il ne faut jamais instantier soi-même un composant Spring !!

Or, les points d'entrée de notre application sont les différentes servlets Java EE, qui NE SONT PAS des composants Spring ! (les servlet sont gérées par le container application, Tomcat ou Jetty par exemple).

Une conséquence est que tant qu'un composant Spring n'a pas été récupéré depuis l'ApplicationContext, il n'est pas possible d'injecter des dépendances (par exemple avec l'annotation Autowired) dans le contexte de la classe courante (et donc il n'est pas possible de faire un Autowired dans la servlet !)

Pour récupérer l'instance d'un composant Spring générée par Spring, si appContext est l'ApplicationContext, il est possible d'utiliser :
```
appContext.getBean(ProductDAO.class);
```

Ainsi, pour obtenir une instance de ProductDAO avec injection de dépendances, il ne faut pas utiliser :
~~`new ProductDAO();`~~
mais plutôt :
```
appContext.getBean(ProductDAO.class);
```

ProductDAO peut donc maintenant profiter de l'injection de dépendance, ainsi que tous les objets encapsulés dans cette classe.

II.3 Configuration d'Hibernate depuis un bean Spring

Le module spring-orm va nous permettre d'injecter la configuration d'Hibernate directement depuis un bean de configuration de Spring.
Pour cela, nous allons créer une nouvelle classe annotée @Configuration. Cette annotation indique à Spring de créer une instance unique de cette classe, utilisée comme configuration et rattachée à l'ApplicationContext.

Un exemple de configuration pour hibernate est disponible dans le répertoire resourcesEtudiants/SpringHibernateConfig.java

On remarque notamment qu'une méthode sessionFactory() est annotée @Bean. Cette annotation indique à Spring de générer un singleton ayant le même nom que la méthode (sessionFactory) et de le rattacher à l'ApplicationContext.
Spring peut donc gérer pour nous la création et le cycle de vie de la SessionFactory, qui est désormais disponible comme Bean dans l'ApplicationContext.

Nous pouvons donc maintenant supprimer le fichier de configuration d'hibernate : src/main/resources/hibernate.cfg.xml

II.4 Déléguer la gestion des transactions à Spring

Spring intègre la gestion et la propagation des transactions au travers de ses composants (les transactions sont en principe définies dans la couche "Service"). Il existe notamment une implémentation pour les transactions Hibernate :
org.springframework.orm.hibernate3.HibernateTransactionManager

Une transaction est définie grâce à l'annotation @Transaction sur une méthode ou sur un composant (dans ce cas, toutes les méthodes sont par défaut transactionnelles).

Utiliser les transactions Spring pour simplifier la gestion des transactions dans le code.

II.5 Application multi-couches

Utiliser l'injection de dépendances de Spring pour faciliter la séparation en différentes couches de l'application.

En essaiera notamment d’utiliser une couche service pour la gestion des transactions et/ou la gestion de la persistance.

ATTENTION : FAIRE UN DEUXIEME RENDU DU PROJET APRES LA PARTIE II