Persistance des objets

Dans l'architecture en couches des applications distribuées, la couche de persistance permet de faire le lien avec la sauvegarde physique des données.

Nous avons des objets qui représentent des entités avec des états :

- une personne avec son nom, son âge, sa date de naissance... ;
- un compte bancaire avec son numéro, son solde... ;
- une adresse ;
- etc.

Ces objets ont des états en mémoire et si l'application est arrêtée, cet état n'est pas sauvegardé. Le développeur doit donc se soucier du maintien de l'état dans une base de données : la persistance. Le nec plus ultra pour le développeur serait qu'il n'ait pas à se soucier des mécanismes sous-jacents qui permettent aux objets d'être "persistés". Ceci arrivera peut-être un jour...

Cette sauvegarde est principalement effectuée dans une base de données, en général, relationnelle : Oracle, MySql, PosgreSQL, HSQLDB. Elle peut aussi être effectuée par sérialisation utilisation de fichiers, de XML, etc.

Cette couche va encapsuler les mécanismes qui auront la responsabilité de sauvegarder, modifier ou restaurer l'état de certains des objets métiers de l'application.

Il ne doit pas y avoir de requêtes SQL, ou autres, qui parsèment l'ensemble du code des autres couches applicatives. Idéalement, le développeur ne devrait pas avoir à se soucier des mécanismes sous-jacents utilisés pour permettre la persistance de ses objets. Cette couche doit aussi permettre de garder une certaine indépendance vis-à-vis de l'architecture de la base de données utilisée. Les couches de persistance s'appuient sur l'API JDBC.

De très nombreuses implémentations de cette couche peuvent être envisagées :

- codage de la couche par le développeur en utilisant l'API JDBC ;
- utilisation du framework Hibernate qui est inclus dans la distribution JBoss ;
- utilisation des EJB entités ;
- utilisation d'autres frameworks...

1. Codage du pattern DAO

Si le codage de la couche est effectué par le développeur, celui-ci mettra certainement en pratique le design pattern DAO *(Data Acces Object)*. Ce design pattern de conception montre comment encapsuler tous les accès aux sources de données. En général, à chaque objet métier est associé un objet DAO qui contient le code SQL nécessaire à la sauvegarde, modification, suppression en base de données.

Les sources présentés ici, sont issues du projet "Chap4 - JDBC". Par exemple, une classe métier `Ville` est associée à une classe `VilleDAO` qui contiendra les méthodes permettant l'ajout, les recherches en base de données, etc. Les méthodes de cette classe DAO renvoient un objet, ou des collections d'objets, de type `Ville`. Les classes de type `Ville` sont des classes en général, très simples, constituées par des méthodes de type setter/getter, appelées aussi POJO *(Plain Old Java Object)*.

Le codage de ces classes DAO n'est pas complexe, mais s'avère très vite fastidieux, voire ennuyeux car répétitif. En général, des méthodes nommées `getQuelqueChoseByAutreChose(…)` encapsulent des requêtes SQL du type `"SELECT quelqueChose FROM table WHERE colonne=autreChose;"`.

La classe `Ville` :

```
public class Ville implements Serializable
{
    private String codePostal;
    private String nom;
```

```
    public Ville()
    {}

    public Ville(String nom, String cp)
    {
        this.nom = nom;
        this.codePostal = cp;
    }

    public String getCodePostal()
    {
        return codePostal;
    }

    public void setCodePostal(String codePostal)
    {
        this.codePostal = codePostal;
    }

    public String getNom()
    {
        return nom;
    }

    public void setNom(String nom)
    {
        this.nom = nom;
    }

    public String toString()
    {
        return this.nom + " - " + this.codePostal;
    }
}
```

Voici quelques extraits de la classe `VilleDAO` :

```
public class VilleDAO
{
    private DAO dao = null;

    public VilleDAO(DAO dao)
    {
        this.dao = dao;
    }

    public Collection<Ville> getVillesByCodePostal(String cp)
throws ApplicationDAOException
    {
        Collection<Ville> villes = new ArrayList<Ville>();

        String sql = "SELECT * FROM codes_postaux WHERE cp=?";
        Connection con = null;
        try
        {
            con = dao.getConnection();
            PreparedStatement st = con.prepareStatement(sql);
            st.setString(1, cp);
            ResultSet rs = st.executeQuery();
            while (rs.next())
            {
                villes.add(this.construireVille(rs));
            }
        }
        catch (SQLException e)
        {
            throw new ApplicationDAOException(e);
        }
```

```
        finally
        {
            try
            {
               dao.releaseConnection(con);
            }
            catch (SQLException e)
            {
            }
        }
        return villes;
    }

...
}
```

Vous remarquerez que cette classe est construite sur une classe DAO, que nous présenterons dans la section suivante.

Remarquez aussi que les exceptions levées sont d'un type propre à l'application ce qui permet de ne pas "polluer" les autres couches de l'application par des Exceptions trop spécifiques qui exposent les APIs utilisées. En effet, la couche DAO gère la persistance, si une exception d'une API de bas niveau remonte vers la couche métier ou la couche de présentation, on supprime le couplage faible qui doit exister entre les couches de l'application. La classe `ApplicationDAOException` vient encapsuler le type réel de l'exception. Cette classe est très simple et permet de changer la couche DAO, qui peut générer d'autres types d'exceptions, sans impacter les couches utilisatrices.

```
public class ApplicationDAOException extends Exception
{
    private static final long serialVersionUID = 1L;

    public ApplicationDAOException()
    {}

    public ApplicationDAOException(String cause)
    {
        super(cause);
    }

    public ApplicationDAOException(Exception e)
    {
        super(e);
    }

    public ApplicationDAOException(String cause,Exception e)
    {
       super(cause,e);
    }
}
```

Le modèle général de codage utilisant JDBC est le suivant :

- obtention d'une connexion au serveur de base de données ;
- création d'une requête SQL ;
- exécution de la requête SQL ;
- traitement du jeu de résultat ;
- fermeture de la connexion.

Le codage des classes DAO est toujours basé sur le même schéma, symbolisé par l'acronyme CRUD *(Create, Read, Update and Delete)* pour la création, la lecture, la mise à jour et la suppression des enregistrements en base.

2. Source de données et pool de connexion

Si le codage lui-même n'est pas complexe, il se pose malgré tout, très vite, dans les environnements distribués, un problème important : comment gérer les connexions et déconnexions à la base de données. De manière classique, la connexion et la déconnexion sont effectuées via la classe `java.sql.DriverManager` de JDBC.

```
connexion = DriverManager.getConnection(url,user,pswd)
```

C'est donc le développeur qui gère les connexions. Mais, par exemple pour un site web, les cycles de connexion/déconnexion peuvent être très fréquents et le nombre de connexions nécessaires à un instant donné, très important. Ce mode de fonctionnement est donc inadapté.

Pour pallier à ce problème, les serveurs d'applications fournissent un service de source de données qui utilise un mécanisme de pool de connexion. Le développeur n'utilise plus le `DriverManager`, mais une classe `javax.sql.DataSource` qui est fournie par le serveur. Attention, cette classe n'est utilisable que dans un environnement qui peut la fournir, elle ne l'est pas dans une application autonome.

Ce n'est plus l'application qui gère les connexions, c'est le serveur. Le serveur maintient un certain nombre de connexions qui sont physiquement connectées à la base de données, et qui sont distribuées au fur et à mesure des besoins de l'application. L'appel de la méthode `getConnection()` permet de récupérer une connexion dans le pool. Une connexion est remise dans le pool après un appel à la méthode `close()`, ou après un timeout.

Les sources de données sont montées dans le nommage JNDI par le serveur. Pour utiliser une source de données, il faut donc :

- effectuer une recherche JNDI ;
- demander une connexion.

Une classe spécifique peut gérer les connexions et déconnexions à la base. C'est ici, la classe appelée DAO qui permet d'encapsuler le type réel utilisé pour l'accès aux objets java.sql.Connection. En effet, une application non distribuée utilisera le `DriverManager`, tandis qu'une application exécutée au sein d'un conteneur utilisera de préférence une `DataSource`.

```
public class DAO
{
    String driver;
    String url;
    String user;
    String pswd;
    DataSource dataSource=null;

    public DAO(String driver, String url,
            String user, String pswd)
        throws ClassNotFoundException
    {
        this.driver = driver;
        this.url = url;
        this.user = user;
        this.pswd = pswd;
        Class.forName(driver);
    }

    public DAO(DataSource dataSource)
    {
        this.dataSource = dataSource;
    }

    public Connection getConnection() throws SQLException
    {
        Connection connexion = null;
        if(this.dataSource!=null)
        {
            connexion = this.dataSource.getConnection();
        }
        else
        {
            connexion =
                DriverManager.getConnection(url,user,pswd);
        }
```

```
        return connexion;
    }

    public void releaseConnection(Connection con)
        throws SQLException
    {
        con.close();
    }
}
```

Les classes du modèle de `VilleDAO` vont demander à la classe `DAO` une connexion à la base. La classe DAO, en fonction de son mode de construction, par une `DataSource`, ou par les paramètres de connexion, renverra la connexion adéquate.

3. Autre modèles de conception de la couche de persistance

Lors du codage des couches DAO, il est très vite visible que le code est très répétitif et pourrait presque être généré automatiquement. L'objectif des solutions présentées ici est donc d'éviter le codage des requêtes SQL avec l'utilisation de JDBC.

D'autres solutions sont envisageables :

- la sérialisation des instances ;
- utilisation des EJB (EJB 2 ou EJB 3) dont le déploiement est détaillé dans cet ouvrage ;
- utilisation de JDO *(Java Data Object)* qui va enrichir le pseudo-code de la classe à persister ;
- utilisation de framework de mapping entre les classes et leur représentation en base, comme Hibernate (abordé plus loin), ou iBatis.

Paramétrage des sources de données

JBoss fournit un moyen très simple et très souple de configurer une source de données. La description de la configuration de connexion à la base de données se trouve dans un fichier qui doit être nommé *yyy-ds.xml* où yyy est le nom (arbitraire) de la configuration de votre source de données et `ds` pour data source. Ainsi, une configuration pour une base de données MySql pourrait être *contacts-ds.xml*.

Ce fichier est simplement mis dans un répertoire de déploiement pour être pris en compte par JBoss. La prise en compte de cette configuration est effectuée par le service JCA de JBoss. Le modèle de nommage du fichier est nécessaire pour qu'il soit pris en compte par la classe `XSLSubDeployer`.

Plusieurs sources de données peuvent être configurées dans un même fichier, mais une bonne habitude est de mettre en place un fichier par source de données, ou par base de données. Le fichier *<installation_jboss>/docs/dtd/jboss-ds_1_5.dtd* décrit la DTD complète de la structure de ce fichier de configuration. Nous allons décrire ici les principaux éléments.

1. Principaux éléments du fichier de configuration

a. Éléments de base

`<datasources>` : élément racine du fichier de configuration *yyy-ds.xml*.

```
<!ELEMENT datasources (mbean | local-tx-datasource
| xa-datasource | no-tx-datasource | ha-local-tx-datasource |
ha-xa-datasource)*>
```

- `<mbean>` : des MBeans peuvent être spécifiés pour qu'ils puissent être configurés avant leur utilisation par la source de données ;
- `<no-tx-datasource>` : utilisé pour des sources de données sans le support de gestion des transactions ;
- `<local-tx-datasource>`, `<xa-datasource>` : sources de données avec support transactionnel. Lorsqu'une connexion doit être utilisée, ou qu'une transaction démarre, un contrôle de l'existence d'une connexion déjà engagée dans la transaction est effectué, si cette connexion existe, elle est utilisée. `<xa-datasource>` permet la gestion des transactions distribuées ;
- `<ha-local-tx-datasource>`, `<ha-xa-datasource>` : éléments identiques à `<local-tx-datasource>` et `<xa-datasource>` avec ajout d'un module expérimental de gestion de pannes sur un cluster de bases de données. Ce module ne doit donc pas être utilisé sur un serveur en production.

b. Principaux éléments de configuration des sources de données

Les éléments suivants doivent être mis dans un des cinq éléments précédents qui définissent une source de données.

- `<jndi-name>` : nom JNDI dans le contexte avec lequel la source de données sera liée. Le nom JNDI est créé par défaut dans le contexte *java:* qui n'est pas partagé en dehors de la machine virtuelle du serveur.
- `<user-java-context>` : si cet élément contient `false`, alors la source de données est liée au contexte global JNDI, donc accessible en dehors du serveur. Si cet élément est absent, il est par défaut positionné à `true`.
- `<user-name>` : identifiant de connexion à la base de données.
- `<password>` : mot de passe de connexion à la base de données. Cet élément, comme l'élément `<user-name>`, peut être redéfini par l'application lors de l'invocation de `getConnection(…)`, ou par un contexte sécurité JAAS.
- `<min-pool-size>` : minimum de connexions que doit contenir le pool de connexions.
- `<max-pool-size>` : nombre maximum de connexions dans le pool de connexions.

- `<blocking-timeout-millis>` : délai d'attente (en millisecondes) d'une connexion avant le déclenchement d'une exception, par défaut : 500 ms.
- `<idle-timeout-minutes>` : délai maximum (en minutes) avant qu'une connexion inutilisée soit fermée.
- `<new-connection-sql>` : ordre SQL devant être exécuté lorsqu'une connexion est créée.
- `<exception-sorter-class>` : nom complet de la classe implémentant l'interface `org.jboss.resource.adapter.jdbc.ExceptionSorter`, qui permet l'encapsulation des exceptions de connexion.

  Les éléments suivants sont communs aux sources de données locale `<no-tx-datasource>` et `<local-tx-datasource>`.

- `<connection-url>` : URL de connexion au driver JDBC, de la forme *jdbc:mysql://localhost:3306/contacts*.
- `<driver-class>` : nom complet de la classe driver JDBC utilisée, par exemple : `com.mysql.jdbc.Driver`.

Le répertoire *<installation_jboss>/docs/examples/jca* contient des fichiers type de configuration vers différentes sources de données.

Exemple de fichier de base pour une connexion vers le serveur MySql :

```
<datasources>
   <local-tx-datasource>
    <jndi-name>mysqlContacts</jndi-name>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <connection-url>jdbc:mysql:///contacts</connection-url>
    <user-name>eni</user-name>
    <password>password</password>

    <min-pool-size>5</min-pool-size>

    <max-pool-size>20</max-pool-size>

    </local-tx-datasource>
</datasources>
```

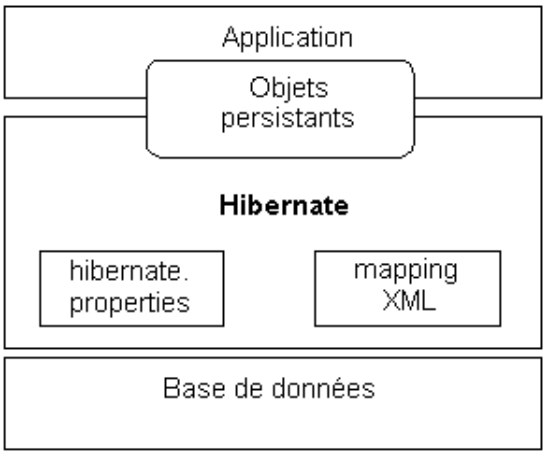
Attention, bien que le déploiement des sources de données par le ficher yyy-ds.xml fonctionne, il arrive que des changements sur le fichier déployé ne soient pas pris en compte. Redémarrez JBoss, si cela persiste, c'est certainement dû au fait que votre fichier est mal renseigné. Vérifiez la console du serveur pour y voir d'éventuels problèmes. Vérifiez bien que votre fichier suive le modèle yyy-ds.xml.

Hibernate

Hibernate est un framework de persistance d'objets java. C'est un ORM *(Object Relationnal Mapping)*, c'est-à-dire qu'il prend en charge le mapping entre un objet java de type JavaBean et la base de données.

JBoss intègre Hibernate depuis la version 4.0.

Pour éviter les confusions entre les JavaBeans et les EJB, le terme POJO *(Plain Old Java Object)* est utilisé pour désigner les classes java très simples. Ces classes obéissent aux spécifications de base des JavaBean. Elles sont sérialisables, possèdent un constructeur par défaut et des getters/setters pour accéder aux propriétés.



Le schéma précédent montre une vue très simplifiée du framework. L'architecture d'Hibernate est très flexible et peut gérer les transactions, les sessions, etc.

Nous allons illustrer ici la manière de configurer Hibernate en tant que service JBoss, nous ne présenterons pas le framework Hibernate et son utilisation. Le lecteur intéressé pourra consulter le site de référence d'Hibernate : http://www.hibernate.org/ où il pourra trouver une documentation très fournie, claire... et en français.

Hibernate utilise une base de données et des propriétés de configuration, pour fournir un service de persistance d'objets.

Les fichiers XML de configuration d'Hibernate (fichiers HBM) vont associer les propriétés des objets avec les champs des tables de la base de données. Nous utiliserons des objets de type POJO qui vont traverser toutes les couches applicatives, ce sont les TO, pour *Transfer Object*. Le fichier d'association Hibernate assure la liaison entre l'objet et la base de données.

L'exemple présenté dans ce chapitre reprend les projets "Chap 4 - Hibernate" et "Chap 4 - Web", ces deux projets devant être déployés sous JBoss. Dans cet exemple, nous afficherons une liste de destinations de voyage accessible par un site web, à l'adresse : http://localhost:8080/chap4/.

La classe POJO qui nous sert d'objet de transfert est la suivante :

```
public class DestinationTO implements Serializable
{
    private long id;
    private String pays;
    private String description;

    public DestinationTO()
    {}

    public long getId()
    {
        return id;
    }

    public void setId(long id)
    {
         this.id = id;
    }
```

```
    public String getPays()
    {
         return pays;
    }

    public void setPays(String pays)
    {
        this.pays = pays;
    }

    public String getDescription()
    {
        return description;
    }

    public void setDescription(String description)
    {
        this.description = description;
    }

    public String toString()
    {
        return this.pays+" : "+this.description;
    }
}
```

Cette classe possède trois propriétés :

- `id` : est le reflet de l'identifiant unique en base de données. Cet identifiant est indispensable pour que Hibernate puisse mapper l'objet avec l'enregistrement en base de données, et le synchroniser, si nécessaire.
- `pays` : correspond au pays de la destination.
- `description` : correspond à une description de la destination.

Pour chacune de ces propriétés, nous avons une méthode getter et une méthode setter.

Le fichier XML qui associe la classe aux champs en base de données est le fichier **Destination.hbm.xml** :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="fr.editions.eni.chap4.TO">
    <class name="DestinationTO" table="destinations" >
        <id name="id" column="kp_destination">
            <generator class="native" />
        </id>
        <property name="pays" />
        <property name="description"/>
    </class>
</hibernate-mapping>
```

Nous retrouvons ici les éléments suivants :

- `<hibernate-mapping>` : définit les associations entre les propriétés des classes et la base de données. L'attribut `package` définit le package dans lequel se trouvent les classes, décrites dans les associations.
- `<class>` : définit quelle classe est associée à quelle table, au travers des attributs `name` et `table`. Le package n'est pas précisé car il a été défini dans l'attribut `package` de l'élément parent.
- `<id>` : décrit l'association sur l'identifiant unique. Le nom de l'identifiant dans la classe est précisé par l'attribut `name`, l'attribut `column` définissant le nom de la colonne dans la table.

- `<generator>` définit dans l'attribut `class`, la stratégie de génération de l'identifiant unique en base, qui correspond ici, à une génération par la base de données. Les stratégies peuvent être :
  - `increment` : généré par Hibernate (attention au mode cluster) ;
  - `native` : généré par la base de données ;
  - `assigned` : assigné, car il s'agit d'une clé primaire naturelle.
- `<property>` définit l'association entre une propriété de la classe dans l'attribut `name`, et une colonne, en base de données, dans l'attribut `column`. Si l'attribut `column` n'est pas présent, Hibernate considère que le nom de la colonne est le même que celui de la propriété.

Nous devons ensuite créer le servie Hibernate. Pour cela, nous allons créer une MBean dont le fichier de configuration *jboss-service.xml* est le suivant :

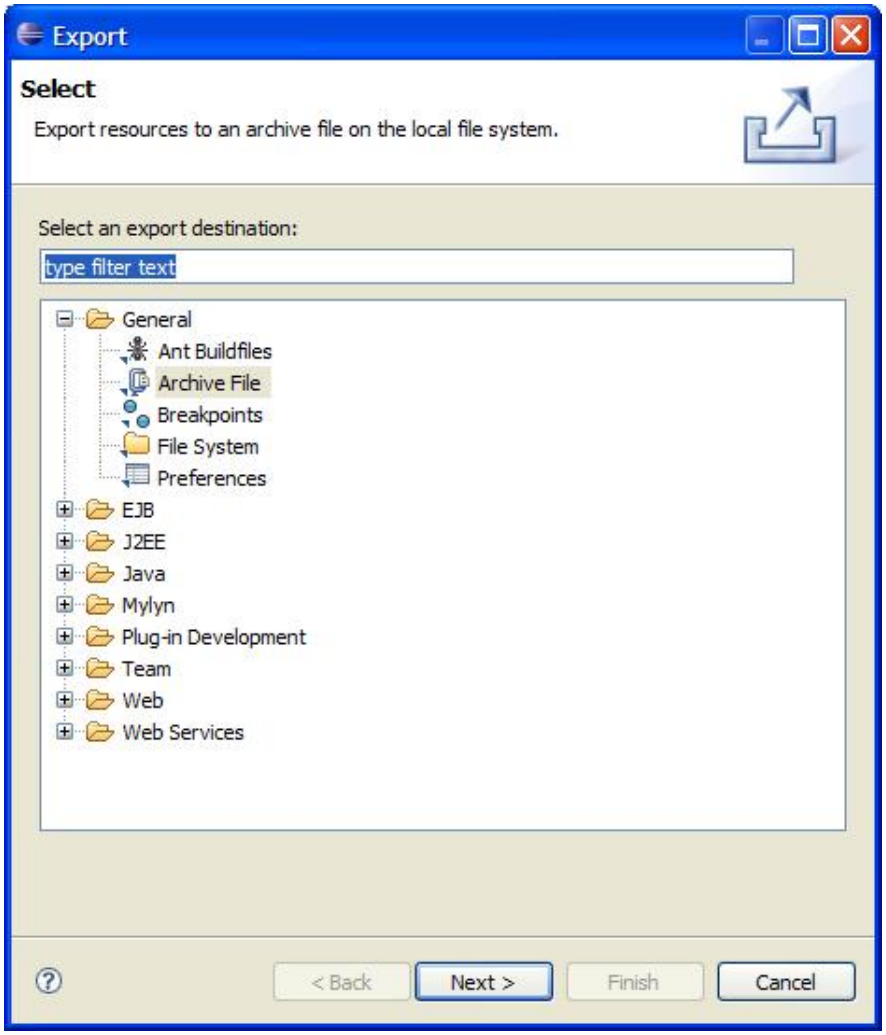
```
<mbean code="org.jboss.hibernate.jmx.Hibernate"
       name="jboss.har:service=Hibernate">
    <attribute
        name="DatasourceName">
            java:/jdbc/BovoyageDS
    </attribute>
    <attribute
        name="Dialect">
            org.hibernate.dialect.MySQLDialect
    </attribute>
    <attribute
        name="SessionFactoryName">
            java:/hibernate/SessionFactory
    </attribute>
    <attribute name="ShowSqlEnabled">true</attribute>
    <attribute
        name="CacheProviderClass">
            org.hibernate.cache.HashtableCacheProvider
    </attribute>
</mbean>
```

Il comprend les éléments suivants :

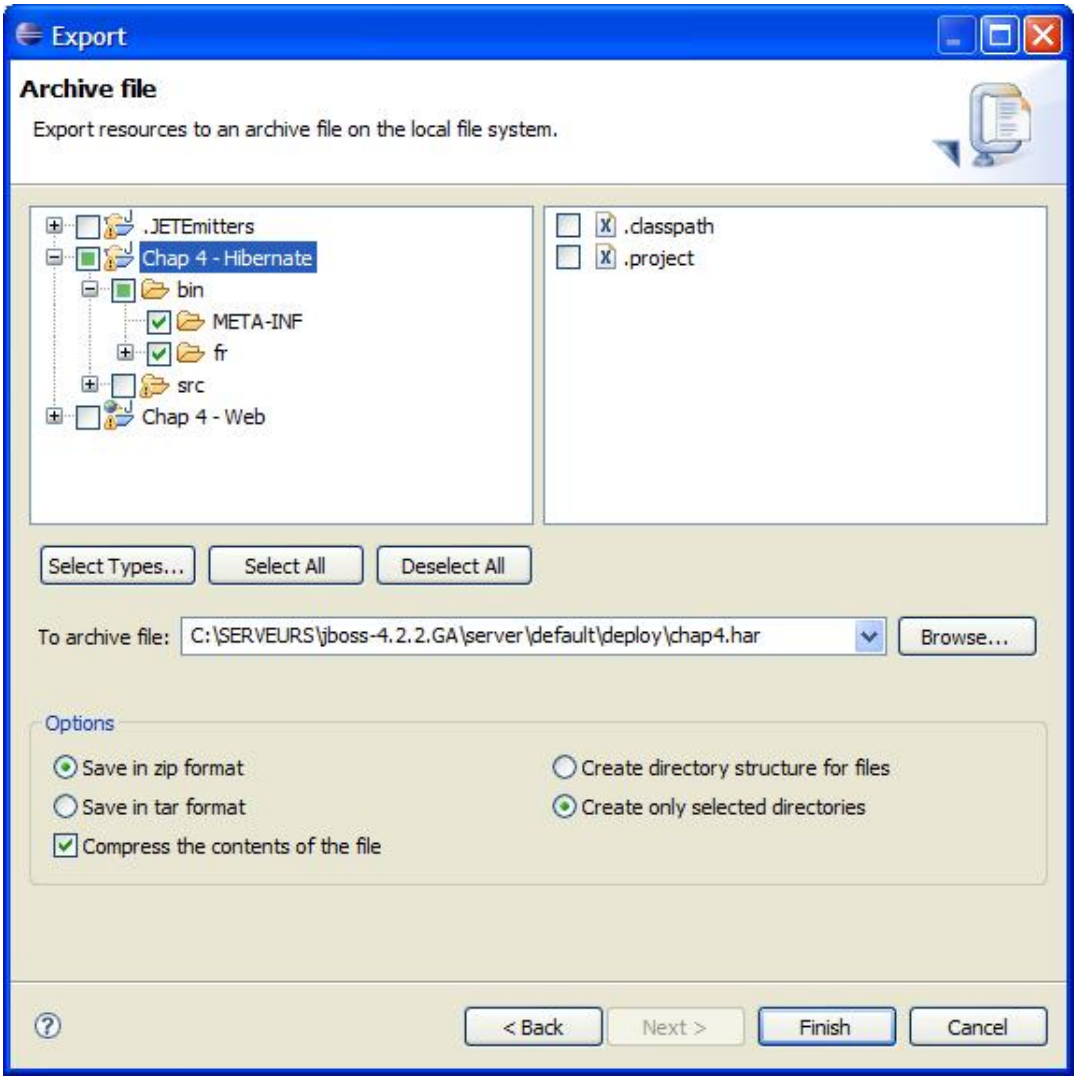
- `<mbean>` : spécifie le nom du service et la classe d'implémentation ;
- `<attribute name="DatasourceName">` : reprend la valeur d'une source de données liée au contexte JNDI, comme nous l'avons vu dans la section Paramétrage des sources de données de ce chapitre ;
- `<attribute name="Dialect">` : permet de préciser quel type de SQL est utilisé. Ici, nous utilisons le SQL de MySQL ;
- `<attribute name="SessionFactoryName">` : définit le nom JNDI de la classe de fabrication des sessions Hibernate. Cette classe sera utilisée par notre application pour récupérer une session Hibernate ;
- `<attribute name="cacheProviderClass">` : définit la classe qui implémente la stratégie de cache, utilisée par JBoss ;
- `<attribute name="ShowSqlEnabled">` : permet de visualiser le code SQL généré par Hibernate dans la console de JBoss, ou dans les fichiers log.

Le projet "Chap 4 - Hibernate" doit être déployé sous forme d'un fichier HAR dans le serveur JBoss. Sauf plugin spécifique, ou définition d'une tâche Ant, Eclipse ne gère pas les archives Hibernate. Une archive HAR est constituée des classes et fichiers de mapping, ainsi que le fichier de configuration du service qui se trouve dans le *META-INF* de l'archive. Nous allons donc voir comment déployer notre projet.

- Une fois le projet importé sous Eclipse, ouvrez-le, puis sélectionnez-le, et effectuez un clic droit dessus. Choisissez l'option **Export** du menu contextuel. Un assistant d'exportation s'ouvre alors.



- Dans le menu **General**, choisissez l'item **Archive File**, puis cliquez sur le bouton **Next**.



- Ouvrez le projet **Chap 4 - Hibernate** situé dans la partie gauche. Désélectionnez la case à cocher du projet et sélectionnez les cases à cocher situées en face de "META-INF" et "fr" (début du package) dans le répertoire bin.

- Sélectionnez le répertoire de déploiement sur le serveur et donnez un nom à votre archive, en prenant garde de bien lui donner l'extension *har*.

- Vérifiez que l'option **Create only selected directories** est cochée, puis cliquez sur le bouton **Finish**.

Vous devez suivre dans la console le déploiement du service Hibernate.

```
13:59:18,390 INFO  [SettingsFactory] Default batch fetch size: 1
13:59:18,390 INFO  [SettingsFactory] Generate SQL with comments: disabled
13:59:18,390 INFO  [SettingsFactory] Order SQL updates by primary key:
disabled
13:59:18,390 INFO  [SettingsFactory] Order SQL inserts for batching: disabled
13:59:18,390 INFO  [SettingsFactory] Query translator: org.hibernate.hql.ast.
ASTQueryTranslatorFactory
13:59:18,390 INFO  [ASTQueryTranslatorFactory] Using ASTQueryTranslator
Factory
13:59:18,390 INFO  [SettingsFactory] Query language substitutions: {}
13:59:18,390 INFO  [SettingsFactory] JPA-QL strict compliance: disabled
13:59:18,390 INFO  [SettingsFactory] Second-level cache: enabled
13:59:18,390 INFO  [SettingsFactory] Query cache: disabled
13:59:18,390 INFO  [SettingsFactory] Cache provider: org.hibernate.cache.
HashtableCacheProvider
13:59:18,390 INFO  [SettingsFactory] Optimize cache for minimal puts:
disabled
13:59:18,390 INFO  [SettingsFactory] Structured second-level cache entries:
```

```
disabled
13:59:18,406 INFO  [SettingsFactory] Echoing all SQL to stdout
13:59:18,406 INFO  [SettingsFactory] Statistics: disabled
13:59:18,406 INFO  [SettingsFactory] Deleted entity synthetic identifier
rollback: disabled
13:59:18,406 INFO  [SettingsFactory] Default entity-mode: pojo
13:59:18,406 INFO  [SettingsFactory] Named query checking : enabled
13:59:18,437 INFO  [SessionFactoryImpl] building session factory
13:59:18,687 INFO  [SessionFactoryObjectFactory] Not binding factory to JNDI,
no JNDI name configured
13:59:18,687 INFO  [NamingHelper] JNDI InitialContext properties:{}
13:59:18,687 INFO  [Hibernate] SessionFactory successfully built and bound
into JNDI [java:/hibernate/SessionFactory]
```

- Dans la vue des noms JNDI déployés sous JBoss, vous devez aussi retrouver le fabricant de session, sous le contexte `java:`.

```
  +- hibernate (class: org.jnp.interfaces.NamingContext)
  |   +- SessionFactory (class: org.hibernate.impl.SessionFactoryImpl)
  +- timedCacheFactory (class: javax.naming.Context)
```

La partie client est constituée d'une application Web "Chap 4 - Web" qui affiche les destinations disponibles. La recherche de la session Hibernate est encapsulée dans une classe de localisation `ServiceLocator`.

```
public class ServiceLocator
{
    private static final String HIBERNATE_SESSION_FACTORY =
"java:/hibernate/SessionFactory";

    public static SessionFactory getHibernateSessionFactory()
    {
        SessionFactory sessionFactory = null;
        try
        {
            Context ctx = new InitialContext();
            sessionFactory = (SessionFactory)
ctx.lookup(HIBERNATE_SESSION_FACTORY);
        }
        catch (Exception e)
        {
           e.printStackTrace();
        }
        return sessionFactory;
        }

    public static Session getHibernateSession()
    {
        Session session = null;

        session = getHibernateSessionFactory().openSession();

        return session;
    }
}
```

Le nommage JNDI utilisé ici est le nommage global. Nous aurions pu utiliser un nom local en liant le nommage global au sein des fichiers jboss-web.xml et web.xml.

La servlet qui utilise la session Hibernate, invoque la méthode `getHibernateSession()` du `ServiceLocator`.

```
public class DestinationsServlet extends javax.servlet.http.HttpServlet
implements javax.servlet.Servlet {
  static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
         List destinations = new ArrayList();
```

```
        Session hibernate = null;

        hibernate = ServiceLocator.getHibernateSession();
        Criteria criteria = hibernate.createCriteria(DestinationTO.class);
        destinations = criteria.list();

        request.setAttribute("destinations", destinations);
        RequestDispatcher rd =
this.getServletContext().getRequestDispatcher("/destinations.jsp");
        rd.forward(request, response);
    }

    protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        doGet(request,response);
    }
}
```

La servlet récupère ensuite la liste des destinations et met cette liste dans un contexte de requête pour que la page *destinations.jsp* puisse l'afficher. La récupération de la liste des destinations passe par la création d'un objet Criteria, qui équivaudra à un `"SELECT * FROM destinations"`.

Nous avons pu interroger notre base de données sans une seule ligne de code JDBC. Maintenant, nous pourrions ajouter, supprimer ou modifier des destinations sans rien ajouter, en utilisant la session Hibernate.

Pour résumer l'utilisation que nous avons faite d'Hibernate :

1. création de la source de données, fichier **-ds.xml** ;
2. création des POJO ;
3. création du fichier de mapping, fichier **-hbm.xml** ;
4. création du fichier décrivant le service Hibernate ;
5. déploiement du fichier HAR contenant les éléments précédents ;
6. codage d'un `ServiceLocator`, qui sera réutilisé... ;
7. création de l'application cliente.