Fil rouge : séance 2
Module FIP INF 211

Gestion de la persistance des données (JPA)

Déroulement

À ce stade du fil rouge, le modèle relationnel pour le stockage des données de l'application au sein du SGBD PostgreSQL est prêt : du schéma logique a été déduit le script SQL de création des tables, ce script a été exécuté et quelques données de test sont présentes dans la base.

Le développement de l'application *cabinet de recrutement* qui manipulera les données stockées dans cette base peut commencer. La première tâche à réaliser concerne la gestion de la persistance des données par l'application. Celle-ci est développée en langage Java, langage objet, dont le modèle est entièrement différent du modèle relationnel utilisé par le SGBD PostgreSQL. Cette gestion de la persistance doit donc assurer la correspondance entre ces deux modèles dont le nom le plus courant, en bon français, est le ***mapping objet/relationnel***.

L'application est déployée dans un serveur GlassFish qui implémente la spécification EJB 3. Cette spécification comprend une partie liée à la gestion de la persistance : JPA (*Java Persistence API*). Cette API sera au centre du travail de la séance constitué de l'essentiel de la couche basse de l'application : ***la couche données***. Il peut être séparé en deux parties principales :

- La création des classes qui « encapsuleront » les données : les ***Entity Beans*** (appelées aussi EJB Entités ou classes entités, etc.).
- La création de classes qui assureront les fonctionnalités minimales autour des classes entités : création, récupération, mise à jour, effacement appelées aussi fonctionnalités CRUD (*Create*, *Retrieve ou Read*, *Update* et *Delete*). Ces classes, sous-parties de la couche données, sont appelées conventionnellement (ce n'est pas obligatoire…) des ***DAO*** (*Data Access Object*).

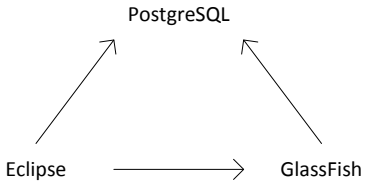
Pour aller plus loin…

- Tutorial sur le pattern DAO : http://cyrille-herby.developpez.com/tutoriels/java/mapper-sa-base-donnees-avec-pattern-dao/
- Résumé de la technologie JPA à l'URL http://igm.univ-mlv.fr/~dr/XPOSE2007/acollign_ORM-JPA/jpa-tech2.html.
- Support de cours sur JPA par un enseignant de l'université de Nice :
  - Partie 1 : http://deptinfo.unice.fr/~grin/mescours/minfo/modpersobj/supports/jpa2-1-6.pdf
  - Partie 2 : http://deptinfo.unice.fr/~grin/mescours/minfo/modpersobj/supports/jpa2-2-6.pdf

Bien évidemment, d'autres ressources sur le sujet sont disponibles sur le Web ou en librairie.

Informations préliminaires

Ces quelques informations permettent de comprendre comment les différents outils qui sont liés entre eux fonctionnent. Ces informations ne sont pas fondamentales pour la réalisation du travail mais elles fournissent une aide à la compréhension des mécanismes mis en œuvre. Elles peuvent être aussi utiles si vous voulez dupliquer l'environnement de travail du fil rouge ou si vous voulez mettre en place un environnement similaire.



Configuration d'Eclipse pour l'accès à PostgreSQL

L'IDE Eclipse installé dans la machine virtuelle est configuré à la fois pour un accès au serveur GlassFish (pilotage du serveur via l'onglet *Servers*) et pour une connexion à la base CABINET_RECRUTEMENT_BD. Cette deuxième configuration est nécessaire aux outils Eclipse qui seront utilisés dans la séance. Le paramétrage de cette configuration est assuré par plusieurs choses :

1. Le driver JDBC pour PostgreSQL est référencé dans Eclipse. Voir : menu *Window* > *Preferences*, dans l'arborescence à gauche ouvrir *Data Management* > *Connectivity* > *Driver Definitions*. Sélectionnez *PostgreSQL JDBC Driver* puis le bouton *Edit…* pour accéder au paramétrage. Bien évidemment, ne rien modifier dans ce paramétrage pour que la connexion fonctionne…
   Note : un driver JDBC est un outil logiciel qui assure les fonctionnalités de connexion à un SGBD. Il existe des drivers JDBC spécifiques pour Oracle, PostgreSQL, MySQL, etc.
2. Une connexion à la base CABINET_RECRUTEMENT_BD est paramétrée (elle utilise la définition du driver JDBC ci-dessus). Il est possible, via ce référencement, de manipuler les données dans la base. Pour y accéder il faut ouvrir l'onglet Data Source Explorer. Dans cet onglet, les données de la base sont accessibles à partir de l'arborescence *Database Connections* > *PostgreSQL (localhost) - CABINET_RECRUTEMENT_BD*. L'accès à la base de cette manière n'est pas le plus ergonomique qui soit mais ce référencement est indispensable pour d'autres outils Eclipse (outils de retro-conception pour la génération des classes entités, voir plus loin).

Configuration d'Eclipse pour l'accès à GlassFish

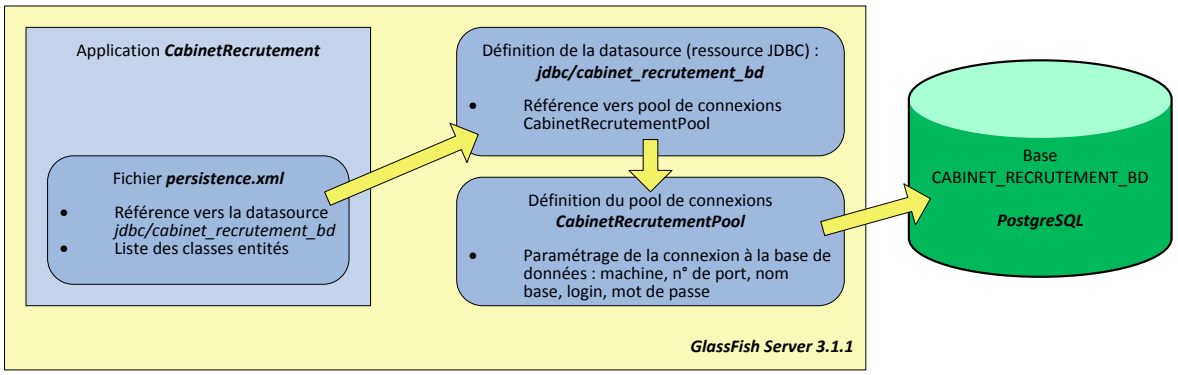
Le paramétrage de l'accès au serveur GlasshFish, outre son pilotage (lancement/arrêt), permet un déploiement automatique de l'application dès qu'une modification est apportée. Lors de la modification d'un des fichiers source, Eclipse le compile automatiquement. GlassFish détecte la modification de la(des) classe(s) compilée(s) et re-déploie l'application qui peut alors être testée. Pour configurer ce paramétrage :

1. Création d'un *Runtime Environment* : menu *Window* > *Preferences*, ouvrir *Server* > *Runtime Environments*. L'entrée GlassFish 3.1 permet de configurer l'accès dans le système de fichiers vers le serveur en lui-même.
2. La référence vers le serveur est ensuite ajoutée à l'onglet *Servers*. Cette référence se sert du *Runtime Environment* créé au-dessus.

Configuration de GlassFish pour l'accès à PostgreSQL

Eclipse nécessite une connexion à la base pour le bon fonctionnement de certains outils mais l'application, une fois déployée, en a aussi besoin. L'application étant déployée dans GlassFish, une partie de cette configuration y est paramétrée, l'autre partie se faisant au sein de l'application elle-même. Cette manière de faire permet à l'application

d'abstraire les paramètres de connexion (ils sont totalement indépendants de l'application), le paramétrage devient ainsi une tâche d'administration système.



Configuration de l'application pour la connexion à la base de données

La configuration de GlassFish pour la connexion à la base de données se fait grâce à la console d'administration Web accessible à l'URL http://localhost:8080/CabinetRecrutement_WEB/ Elle se fait à deux niveaux :

1. Définition d'un pool de connexions JDBC.
   Note : pour que GlassFish se connecte à une base PostgreSQL il lui faut, comme Eclipse, avoir accès à un driver JDBC. Ce driver, un fichier JAR nommé *postgresql-9.0-801.jdbc4.jar*, est déposé dans le répertoire */opt/glassfish-3.1/glassfish/domains/domain1/lib/ext*
   Dans l'arborescence de la partie gauche de l'interface, ouvrir *Ressources* > *JDBC* > *Pools de connexions JDBC*. À cet endroit se retrouvent paramétrés plusieurs pools (ceux des TP architecture) dont celui de l'application du fil rouge : *CabinetRecrutementPool*. C'est ici que sont notés les paramètres de connexions à la base (onglet *Autres propriétés*) : nom de la machine, n° de port, nom de la base, login et mot de passe de l'utilisateur de la base de données.
2. Définition d'une datasource (appelée ici ressource JDBC).
   GlassFish gère l'accès à un certain nombre de ressources (dont les datasources) grâce à un système d'annuaire, les applications hébergées peuvent ainsi y accéder. La ressource JDBC n'est qu'une entrée d'annuaire qui pointe vers le pool de connexions ci-dessus. Pour accéder à ce paramétrage, ouvrir *Ressources* > *JDBC* > *Ressources JDBC*, vous y trouverez la datasource de l'application : *jdbc/cabinet_recrutement_bd*.

Pour se connecter à la base de données, l'application n'a plus qu'à appeler la « référence » de la connexion (la ressource JDBC) disponible au sein de l'annuaire disponible dans GlassFish. Cette référence est notée dans un fichier (au format XML) disponible dans l'application : le fichier *persistence.xml* situé dans le projet *CabinetRecrutement_EJB*, répertoire *ejbModule/META-INF*. Ce fichier comprend plusieurs choses : la liste des classes entités à prendre en compte pour le mapping objet/relationnel et le nom de la datasource dans l'élément *<jta-data-source>*.

Génération des classes entités.

Le propos de cette partie des développements est identique à ce qui a été vu dans le TP *JPA pour le mapping objet-relationnel*. En revanche, l'approche utilisée sera une approche contraire: *bottom-up* à la place de *top-down*. Quelques explications ont été données page 7 du document de la séance 1.

Dans l'approche *bottom-up*, la base de données préexiste au modèle objet. Deux solutions sont alors possibles : écrire manuellement la représentation objet des données (les classes …) ainsi que la correspondance (mapping) avec les structures de la base (les tables…) ou utiliser un outil qui fera automatiquement ce travail (☺). C'est ce second choix qu'il vous est demandé de réaliser.

Cette version d'Eclipse (Eclipse pour *Java Enterprise Edition*) dispose d'un outil pour faire cette retro-conception. Pour y accéder :

- Sélectionnez le projet *CabinetRecrutement_EJB* dans l'onglet *Project Explorer* ou *Navigator*, clic droit > menu *JPA Tools* > *Generate Entities from Tables…* Note, l'outil pour l'approche inverse (*top-down*) existe aussi : *Generate Tables from Entites*.
  Un assistant composé de plusieurs pages apparaît (en cas de doute, il est possible de revenir en arrière).
- Page 1 : ***Select Tables***
  - Sélectionnez la connexion : *PostgreSQL (localhost) – CABINET_RECRUTEMENT_BD*.
  - Activez la connexion (bouton *Connect*).
  - Sélectionnez le schéma *public*. Les tables que vous avez créées doivent apparaître, sélectionnez-les. Vérifiez que la case *Update class list in persistence.xml* est bien cochée.
  - Bouton *Next >*.
- Page 2 : ***Table Associations***
  La liste des associations entre les tables apparaît, il est possible de sélectionner les associations une à une et d'accéder au paramétrage qui peut être adapté (on ne le fera pas ici) :
  - L'attribut de jointure.
  - La navigabilité entre les classes qu'il est possible de (dé)sélectionner. Essayez de comprendre ce qu'implique cette notion de navigabilité.
  - Le nom des attributs qui représente les associations.
  - Bouton *Next >*.
- Page 3 : ***Customize Default Entity Generation***
  Différents réglages liés à la génération des classes entités sont opérés ici.
  - Méthode de génération des valeurs de clés primaires, *Key generator*. PostgreSQL utilise des séquences, valeur qu'il faut choisir dans la liste déroulante.
  - *Sequence name* : saisir le pattern $table_$pk_seq qui correspond au nom des séquences générées automatiquement par PostgreSQL.
  - *Entity access* : *Field*
  - *Associations fetch* : *Default*
  - *Collection properties type* : conservez *java.util.Set*.
  - *Package* : choisir le package *eu.telecom_bretagne.cabinet_recrutement.data.model*. La classe *Entreprise* est déjà présente dans ce package.
- Page 4 : ***Customize Individual Entities***
  - Page récapitulative par classe entité des choix de la page précédente. Rien à modifier ici.
- Bouton *Finish*.

Vérifiez que les classes soient générées dans le package indiqué et que les références de celles-ci existent bien dans le fichier *persistence.xml*.

Examinez les classes entités et le mapping. Au besoin l'assistant de création peut être rejoué si on veut modifier le paramétrage : celui-ci est mémorisé, il suffit alors d'indiquer les modifications. De même, il peut être intéressant de comparer le modèle généré avec celui présent dans le diagramme de classes UML présenté dans le document de la séance 1.

Petite modification à apporter aux classes générées : il est nécessaire de compléter l'annotation *@SequenceGenerator* de l'attribut correspondant à la clé primaire de chacune des différentes entités en ajoutant le paramètre *allocationSize=1* qui fixe l'incrément de la séquence. Par défaut, la valeur de cet incrément est de 50, ce qui provoque une erreur à l'exécution. Par exemple, pour la classe *Entreprise*, il faut remplacer…

```
@Id
@SequenceGenerator(name="ENTREPRISE_ID_GENERATOR", sequenceName="ENTREPRISE_ID_SEQ")
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="ENTREPRISE_ID_GENERATOR")
private Integer id;
```

…par…

```
@Id
@SequenceGenerator(name="ENTREPRISE_ID_GENERATOR", sequenceName="ENTREPRISE_ID_SEQ", allocationSize=1)
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="ENTREPRISE_ID_GENERATOR")
private Integer id;
```

Création des DAO

Une fois les classes entités créées, afin de pouvoir les manipuler ensuite au sein de l'application, il est nécessaire de créer les classes utilitaires qui vont assurer le lien entre la couche métier (celle qui assure les fonctionnalités de l'application) et nos toutes nouvelles entités

Ces classes assureront les fonctionnalités CRUD pour les classes entités : il y aura donc un DAO pour la classe *Entreprise* (*EntrepriseDAO* dont l'ébauche existe déjà), un DAO pour la classe *Candidature* (*CandidatureDAO*), etc. Les DAO sont une partie de la couche données, ils seront donc localisés dans le package *eu.telecom_bretagne.cabinet_recrutement.data.dao*. Par convention ils portent le nom de la classe entité suffixé par DAO.

Implémentation complète du DAO de la classe entité Entreprise : EntrepriseDAO

L'ébauche de cette classe est déjà fonctionnelle mais incomplète, une fois complétée, elle pourra servir de modèle pour les autres DAO. Cette classe est un composant EJB, l'annotation *@stateless* au-dessus de la déclaration de la classe l'indique. Elle dispose d'un attribut, de type *EntityManager*, qui sert dans chacune des méthodes et dont le but est d'assurer le lien avec la base de données. À noter, cet attribut n'est pas instancié dans la classe (il n'existe pas de *new EntityManager(…)*). À l'exécution, la référence vers cet objet sera assuré par le conteneur d'EJB à l'aide du mécanisme appelé *injection de dépendance*.

Cette classe ne possède pour le moment que deux méthodes (fonctionnalité R de l'acronyme CRUD) :

- *public Entreprise **findById**(Integer id)*
  La méthode retourne une instance de la classe *Entreprise* pour une valeur d'*id* donné.
- *public List<Entreprise> **findAll**()*
  La méthode retourne la liste (au sein d'une instance de *java.util.List*) de toutes les entreprises référencées dans la base.

Afin de compléter cette classe, il sera nécessaire d'ajouter les méthodes suivantes (les noms donnés sont optionnels, il est possible de les modifier) :

- *public Entreprise **persist**(Entreprise entreprise)*
  Fonctionnalité C de l'acronyme CRUD : cette méthode stockera dans la base de données les informations d'une entreprise (présent sous la forme d'une instance d'*Entreprise*) .
- *public Entreprise **update**(Entreprise entreprise)*
  Fonctionnalité U de l'acronyme CRUD : cette méthode rend persistant les mises à jour opérées sur une instance d'*Entreprise* passée en paramètre.
- *public void **remove**(Entreprise entreprise)*
  Fonctionnalité D de l'acronyme CRUD : cette méthode supprime les données d'une entreprise passée en paramètre.

Les méthodes de mise à jour et de suppression sont liées à des fonctionnalités marquées comme mineures (donc optionnelles) dans le cahier des charges. Cependant pour une meilleure compréhension des mécanismes il est intéressant d'implémenter complètement la classe.

Méthodes principales de la classe *EntityManager*

Comme indiqué plus haut, la liaison avec la base de données est assurée par l'intermédiaire de l'*EntityManager*. Les principales méthodes de cette classe (c'est en fait une interface) sont :

- *void* ***persist****(java.lang.Object entity)*
  Rend persistant l'instance de la classe entité passée en paramètre.
- *<instance d'entité>* ***find****(classe de l'entité, valeur de la clé primaire)*
  Retourne l'instance de l'entité dont on connait la clé primaire (peut être une chaîne de caractères, un entier, etc.). Classe de l'entité : par exemple, pour la classe *Entreprise*, *Entreprise.class*.
- *Query* ***createQuery****(java.lang.String qlString)*
  Permet d'exprimer une requête exprimée en JPQL (attention, ce n'est pas du SQL !). Sur l'objet *Query* retourné, il est possible d'obtenir le résultat de la requête par l'appel de la méthode *getResultList()*.
- *void* ***merge****(java.lang.Object entity)*
  Met à jour dans la base les modifications apportée à une instance de classe entité.
- *void* ***remove****(java.lang.Object entity)*
  Supprime de la base une instance de classe entité.

Éléments de syntaxe du langage de requête *JPQL*

Le langage JPQL (*Java Persistence Query Language*) est une des possibilités permettant d'exprimer des requêtes sur une base de données dans la spécification JPA. Sa syntaxe la rend proche du langage SQL (*select*, *update*, *delete*, etc.), ce qui facilite son apprentissage mais qui peut rendre aussi sa lecture ambigüe…

Exemple 1 : requête JPQL de la méthode *findAll()* :

- *select entreprise from Entreprise entreprise order by entreprise.id*

Le langage JPQL travaille avec le modèle objet et non le modèle relationnel. D'un point de vue pratique, cela veut dire que dans la requête ci-dessus, derrière la clause *from* le terme *Entreprise* désigne la classe entité et NON LA TABLE *entreprise*. De même, derrière *order by*, *entreprise.id* désigne l'attribut *id* de la classe *Entreprise* et non la colonne *id* de la table *entreprise*.

Exemple 2 : expression d'une jointure. Dans le cas d'un employé appartenant à un département (présence d'un attribut *departement* de type *Departement* dans la classe *Employe*), la requête récupérant la liste des employés appartenant à un département donné (trié par nom d'employé descendant) s'exprime de la manière suivante :

- *select emp from Employe emp join emp.departement dept where emp.nom="nom_du_dept_recherché" order by emp.nom desc*

Création des autres DAO

Le DAO *EntrepriseDAO* servira de modèle pour le développement des autres DAO. Certains d'entre-eux nécessiteront d'être adaptés par rapport au DAO existant :

- *CandidatureDAO*
  - Ajout d'une méthode pour l'obtention de la liste des candidatures qui correspondent à un secteur d'activité et un niveau de qualification donnés.
- *OffreEmploiDAO*
  - Ajout d'une méthode pour l'obtention de la liste des offres d'emploi pour une entreprise donnée.
  - Ajout d'une méthode pour l'obtention de la liste des offres d'emploi qui correspond à un secteur d'activité et un niveau de qualification donnés.
- *SecteurActiviteDAO*
  - La liste des secteurs d'activité n'évolue pas, elle reste fixe. Il n'est donc pas nécessaire d'en supprimer, la méthode *remove(…)* est donc inutile.
  - Suivant les choix de navigabilité faits lors de la création des classes entités, une instance de la classe *SecteurActivite* peut référencer un ensemble d'offres d'emploi et un ensemble de candidatures. Si c'est le cas (c'est le choix par défaut), la méthode *update(…)* est alors nécessaire.

- *NiveauQualificationDAO*
    - Mêmes remarques que pour le DAO *SecteurActiviteDAO*.

Un DAO est une classe de type *bean session* (un EJB). Pour créer ce type de classes, Eclipse fournit un assistant :

- Sélectionnez le projet *CabinetRecrutement_EJB* dans l'onglet *Project Explorer* ou *Navigator*, clic droit > menu *New… > Session Bean (EJB 3.x)…*
- Page 1 : ***Create EJB 3.x Session Bean***
    - Saisir le nom de package de la classe : *eu.telecom_bretagne.cabinet_recrutement.data.dao*.
    - Saisir le nom de la classe.
    - Laisser les choix par défaut pour *State type* (*Stateless*) et *Create business interface*.
- Page 2, rien à modifier. Bouton *Finish*.

L'ébauche de la classe est créée mais la seule méthode existante est le constructeur par défaut. Les annotations *@Stateless* et *@LocalBean* indiquent qu'il s'agit bien d'un EJB.

Procédure de contrôle du bon fonctionnement des DAO

Afin de contrôler que les DAO fonctionnent correctement, il n'est pas possible de créer une classe (avec une méthode *main*) qui instancierait un DAO sur lequel il serait possible de tester les appels de méthodes. En effet pour fonctionner, les DAO nécessitent d'être déployés sur GlassFish afin de bénéficier de toute la « mécanique » offerte par l'environnement d'exécution (connexion à la base, mécanisme d'injection de dépendance, etc.). Les tests doivent donc se faire sur le serveur.

Cette manipulation, un peu plus complexe qu'une simple classe exécutable, se fait au sein du projet *CabinetRecrutement_WEB*. Une classe particulière qui génère du contenu Web (un classe de type *Servlet*) est accessible à l'URL : http://localhost:8080/CabinetRecrutement_WEB/ControlesDAO (le lien est présent dans les marque-pages de Firefox). Cette page affiche le résultat du test. Le code est accessible dans la classe *ControlesDAOServlet* (package *eu.telecom_bretagne.cabinet_recrutement.front.controlesDAO*).

La référence vers le DAO à tester est faite via la classe *ServiceLocator* (package *eu.telecom_bretagne.cabinet_recrutement.front.utils*). Une fois cette référence récupérée, les méthodes du DAO peuvent être appelées et les résultats sont affichés grâce aux appels *out.println(contenu_à_afficher)*.

Un exemple de contrôle pour les deux méthodes fournies dans la classe *EntrepriseDAO* est donné, l'adaptation pour les autres DAO devrait se faire facilement.

Livrables

Le code des classes développées, classes entités et classes des DAO, sera à déposer sur Moodle avant la séance 3 du fil rouge (dépôt avant le 26 septembre). Faire une archive contenant uniquement les fichiers sources (au format zip, 7z, tar, bz2, etc.). Ce fichier devra porter impérativement les noms des deux membres des binômes.