TP - Outil de gestion des étudiants

Introduction

Dans ce TP, vous allez travailler sur une application suivant une **architecture en couches** que nous l'avons abordé dans le cours de qualité de développement. Chaque couche à un rôle bien précis :

- La couche **ihm** qui permet de gérer les différentes fenêtres graphiques et surtout l'interaction avec l'utilisateur.
- La couche **métier** qui contient les différentes **entités** manipulées qui proviennent initialement de la phase d'analyse et correspondent aux tables de la base.
- La couche **application** qui permet de faire le lien entre la couche **ihm** et la couche **métier**. Elle contient différents **controllers** dont le rôle est de gérer les évènements qui surviennent sur l'interface et d'envoyer des requêtes auprès de la sous-couche '**service** qui contient des classes permettant de manipuler les entités métiers et d'implémenter la partie **logique** de votre application. Les controllers se chargent aussi de transmettre les résultats obtenus à l'ihm.
- Enfin, la couche **stockage** qui permet de gérer la **persistance des données** à travers une forme de stockage configurée (base de données, fichier...). Son rôle va donc être de sauvegarder et charger les données des différentes entités de la couche **métier**. Cette couche est généralement utilisée par les différents classes de services.

Dans le contexte du TP, la partie **IHM**, **métier** et une partie de la couche **service** ont déjà été développés. Actuellement, les données de l'application **ne sont pas stockées** et disparaissent lors de la fermeture du programme. L'objectif du TP va donc être de développer la **couche stockage** afin de permettre la sauvegarde des informations dans une **base de données**.

Installation

1 - Pour commencer, téléchargez le fichier **TpOGEBase.zip** sur **Moodle** et extrayez-le sur votre machine.

2 - Démarrez **IntelliJ** puis ouvrez le dossier **TPOGEBase** contenant les sources du projet. Laissez lui un peu de temps pour se configurer.

Exercice 1 : Découverte

L'interface de l'application propose de gérer trois aspects : Les **ressources** (différentes matières), les **étudiants** (qui ont chacun une ressource favorite) et des **notes** (chaque étudiant possède des notes dans diverses ressources).

Le chargement des ressources par les différents **services** suivent une stratégie dite de **lazy loading**. Cela signifie que les données de l'application ne sont pas conservées dans la mémoire du programme. Chaque fois qu'on veut y accéder, on interagit avec l'entité de stockage (la base de données, dans notre cas).

Les applications web PHP adoptent une stratégie identique. Cela permet une meilleure synchronisation, en cas d'accès depuis diverses sources (un logiciel bureau, une application web, une application mobile...), mais demande beaucoup plus d'interaction avec la base via le réseau.

1 - Explorez le projet mis à votre disposition. Prenez un peu de temps de regarder les différentes classes et fichiers à votre disposition.

2 - Lancez l'application et interagissez avec l'interface. Pour l'instant, rien ne devrait fonctionner (dans le sens où les données ne sont pas visibles et que vos actions ne produisent aucun effet).

Exercice 2 : Connexion entre la couche service et stockage

Pour l'instant, rien ne s'affiche dans l'interface quand on tente de créer des ressources ou des étudiants car la partie de la couche **application** correspondant aux **services** n'est pas reliée à la couche **stockage** (qui n'existe pas vraiment pour le moment, en soi). Pour le moment, les trois services **RessourceService**, **EtudiantService** et **NoteService** ne font donc rien de particulier.

Pour s'assurer que tout fonctionne bien, on a créé une "fausse" couche de stockage chargeant des données volatiles en local (dans la mémoire du programme) et simulant ainsi un stockage. Ce genre d'artifice est généralement utilisé dans le cadre de tests. Cela permet de s'assurer que l'**IHM** et la couche **service** fonctionne sans pour autant avoir besoin d'accéder à la base. On appelle cela un **stub** (bouchon).

Pour l'instant, nous ne nous intéresserons que aux **ressources** et aux **étudiants**.

1 - Dans le package **stockage/stub**, ouvrez les classes **StockageRessourceStub** et **StockageEtudiantStub** et essayez d'en comprendre le fonctionnement.

2 - Dans le package **application/services** ouvrez la classe **RessourceService**. Comme vous pouvez le constater, toutes les méthodes de la classe sont vides. Ajoutez un **attribut** de type StockageRessourceStub et instanciez-le directement avec un **StockageRessourceStub**.

3 - En vous appuyant sur la documentation de la classe, **complétez chaque méthode** de **RessourceService** afin de rendre cette classe fonctionnelle. Il faudra bien sûr s'aider du **repository** de la classe. Vous pouvez aussi aller consulter les différents **constructeurs** des différentes entités (Ressource, Etudiant, Note...).

4 - Lancez l'application. Vous devriez être capables de créer, éditer et supprimer des **ressources**.

5 - De la même manière, rendez la classe **EtudiantService** fonctionnelle. Attention, un **étudiant** est relié à une **ressource** (sa favorite). Il faudra donc utiliser **RessourceService** lors de la création et la mise à jour afin de récupérer le bon objet. Vous pouvez directement accéder à une instance de **RessourceService** via **RessourceService.getInstance()** et ainsi utiliser les méthodes du **service** (il s'agit du design pattern appelé **singleton**, que vous aurez l'occasion d'étudier dans le cours de **qualité de développement**).

6 - Lancez l'application. Vous devriez être capables de créer, éditer et supprimer des **étudiants**.

7 - Que se passe-t-il si vous supprimez une ressource qui est affectée à un étudiant ? Nous voulons changer ce comportement en faisant en sorte que l'étudiant soit supprimé quand la ressource en question est supprimée. Nos classes **stubs** ont atteint leur limite, nous allons donc passer au développement d'une **couche de stockage** utilisant une **base de données Oracle**.

Exercice 3 : Utilisation de JDBC avec Oracle

Nous allons maintenant utiliser **l'API JDBC** pour stocker les informations de l'application. Tout au long des exercices suivants, il vous faudra donc utiliser **le support de cours** ainsi que **la documentation JDBC** disponibles sur Moodle. Mais, avant toute chose, il faut que vous mettiez en place une **base de données** !

Voici le schéma relationnel de la base que nous allons utiliser :

RessourcesOGE(idRessource, nom)

EtudiantsOGE(idEtudiant, nom, prenom, idRessourceFavorite#)

NotesOGE(idNote, idRessource#, idEtudiant#, note)

1 - Sur **Oracle**, importez cette base avec le script **OGE.sql** disponible sur **Moodle**.

2 - Dans le fichier **pom.xml** à la racine du projet, importez le **driver Oracle** en rajoutant cet import dans le bloc **dependencies** :

```
<dependencies>
    ...
    <dependency>
        <groupId>com.oracle.database.jdbc</groupId>
        <artifactId>ojdbc8</artifactId>
        <version>21.9.0.0</version>
    </dependency>
    ...
</dependencies>
```

Un bouton vous permettant de reconstruire le projet (pour télécharger les nouvelles librairies) devrait apparaître sur la droite, appuyez dessus. S'il n'apparait pas, effectue un clic-droit sur le fichier **pom.xml** puis **Maven** et **Reload project**.

Ensuite, dans le fichier **module-infos.java** dans le dossier **src/main/java** ajoutez la ligne suivante :

```
open module tp.oge {
    //...
    requires java.sql;
}
```

3 - Créez un package **sql** à l'intérieur du package **stockage** et ajoutez-y une classe **SQLUtils**. Cette classe devra implémenter le pattern **singleton** afin de pouvoir gérer un objet de type **Connection** permettant d'interagir avec notre base (consulter la partie **"Exemple de paramétrage pour Oracle"** du support de cours). Le but est de rendre accessible cet objet à travers l'instance unique de cette classe.

Le squelette de cette classe est le suivant :

```
class SQLUtils {

    private static SQLUtils instance = null;

    private Connection connection;

    private SQLUtils() {
        /*
        * Initialisation de l'attribut connexion
        */
    }

    public Connection getConnection() {
        return this.connection;
    }

    public static SQLUtils getInstance() {
        if(instance == null) {
            instance = new SQLUtils();
        }

        return instance;
    }

}
```

Quand on aura besoin de l'objet **Connection** dans le code, on fera donc :

```
SQLUtils utils = SQLUtils.getInstance();
Connection connection = utils.getConnection();
```

Pour faciliter l'exercice, un seul objet **Connection** sera géré dans le programme et il ne sera pas explicitement fermé. Dans un contexte réel, il faudrait fermer la connexion quand on n'en a plus besoin ou que l'application est fermée. Il existe également un mécanisme de *pools* permettant de réutiliser une connexion.

Pour **initialiser la connexion**, les différents paramètres dont vous aurez besoin sont :

— L'ip : **162.38.222.149** (SGBD Oracle de l'iut)

— Le port : **1521**
— Le nom de la base : **iut**
— Le login : votre **login iut**
— Le mot de passe : votre **mot de passe iut**
— Le driver (pour Oracle) **oracle.jdbc.driver.OracleDriver**

Exercice 4 : La classe StockageRessourceDatabase

1 - Dans le package **sql**, créez une classe **StockageRessourceDatabase** et faites la implémenter l'interface **Stockage** en la paramétrant avec **Ressource** (si vous êtes confus sur cette instruction, vous pouvez aller voir **StockageRessourceStub**)

2 - A l'aide d'**IntelliJ**, générez le squelette des méthodes à implémenter. Complétez ensuite chacune des méthodes de manière à interagir avec la base de données à l'aide de requêtes SQL pour exécuter les opérations :

— **create** : Fait une requête d'insertion sur la base en utilisant les attributs de l'entité passée en paramètre vous n'aurez pas à gérer l'identifiant de l'entité car celui-ci s'incrémente automatiquement (géré du côté de la base). Pour une ressource, il faut donc seulement insérer un nom.
— **update** : Fait une requête de mise à jour en utilisant les attributs de l'entité passée en paramètre.
— **deleteById** : Fait une requête de suppression de l'entité ciblée par l'identifiant (clé primaire) passée en paramètre.
— **getById** : Effectue une requête pour récupérer le tuple correspondant à l'identifiant passé en paramètre (clé primaire) puis instancie l'entité correspondante et affecte ses attributs avec les données récupérées par la requête. Cette entité est alors renvoyée.
— **getAll** : Effectue une requête pour récupérer toutes les entités de la table. Comme pour **getById**, les données des tuples sont utilisées pour instancier des entités. L'ensemble des entités sont stockées dans une liste qui est renvoyée à la fin du traitement.

Pour vous aider, servez-vous du **support de cours** et de ses exemples de code, et, si nécessaire, de la **documentation JDBC**. L'objet **Connection** devrait être accessible depuis la méthode que vous avez créé dans**SQLUtils**. N'oubliez pas d'utiliser le **try-with-ressource** afin que les différentes ressources manipulées soient automatiquement fermées à la fin du traitement.

3 - Modifiez votre classe **RessourceService** afin d'utiliser votre classe **StockageRessource-Database** au lieu de **StockageRessourceStub** pour gérer le stockage des données.

4 - Lancez votre application, et vérifiez que l'ajout, l'édition et la suppression de ressources fonctionnent toujours bien. Normalement, si vous relancez le programme, les ressources que vous avez créé devraient toujours être affichées !

5 - Sur **Oracle**, affichez le contenu de la table **ressource**. Mettez à jour une **ressource** (changez son nom), et appuyez sur le bouton "rafraîchir" de l'interface du programme. Vous devriez constater que la mise à jour est bien prise en compte.

Exercice 5 : La classe StockageEtudiantDatabase

1 - Dans le package **sql**, créez une classe **StockageEtudiantDatabase** et faites la implémenter l'interface **Stockage** en la paramétrant avec **Etudiant**.

2 - Comme vous l'avez fait précédemment pour **StockageRessourceDatabase**, implémentez les différentes méthodes de manière à interagir avec la base de données à l'aide de requêtes SQL pour exécuter les opérations. Comme pour les ressources, les identifiants (clé primaires) des étudiants s'auto-incrémentent dans la base. **Attention**, un étudiant est lié à une **ressource**. Nous souhaitons que les données de la ressource favorite soient aussi chargées lors du chargement d'un étudiant ! Il faudra donc utiliser une jointure...

3 - Modifiez votre classe **EtudiantService** afin d'utiliser votre classe **StockageEtudiantDatabase** au lieu de **StockageEtudiantStub** pour gérer le stockage des données.

4 - Lancez votre application, et vérifiez que l'ajout, l'édition et la suppression d'étudiants fonctionnent toujours bien. Normalement, si vous relancez le programme, les étudiants que vous avez créé devraient toujours être affichés !

5 - Dorénavant, que se passe-t-il si vous supprimez une ressource qui est affectée à un étudiant ?

6 - Dans votre base de données sur **Oracle**, affichez le contenu de la table **etudiant**. Modifiez un **étudiant** sur la base puis appuyez sur le bouton "rafraîchir" de l'interface pour vérifier que la mise à jour est bien prise en compte.

Exercice 6 : Utilisation de Hibernate et Hibernate Repositories

Pour la suite, et notamment la **gestion des notes** nous allons utiliser l'ORM **Hibernate** ainsi qu'une librairie développée dans le cadre de ce TP : **Hibernate Repositories**. Cette librairie permettra de faciliter l'utilisation de Hibernate.

Avant toute chose, rendez-vous dans la partie consacrée à **Hibernate** dans **le support de cours**. Prenez le temps de relire les explications et les exemples donnés.

1 - Téléchargez le fichier **HibernateRepositories-1.0.jar** depuis **Moodle**. Placez-le dans un dossier **src/main/resources/libs** de votre projet.

2 - Dans le fichier **pom.xml**, importez **Hibernate** ainsi que **Hibernate Repositories** au niveau du bloc **dependencies** :

```
<dependencies>
    ...
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.3.0.Final</version>
    </dependency>
    <dependency>
        <groupId>com.gasquet</groupId>
        <artifactId>hrepositories</artifactId>
        <version>1.0</version>
        <scope>system</scope>
        <systemPath>
            ${project.basedir}/src/main/resources/libs/HibernateRepositories-1.0.jar
```

```
        </systemPath>
    </dependency>
    ...
</dependencies>
```

Appuyez sur le bouton vous permettant de reconstruire le projet.

Ensuite, dans le fichier **module-infos.java** dans le dossier **src/main/java** ajoutez la ligne suivante :

```
open module tp.oge {
    //...
    requires org.hibernate.orm.core;
    requires jakarta.persistence;
    requires HibernateRepositories;
}
```

3 - Téléchargez le fichier de configuration **hibernate.cfg.xml** sur **Moodle** et placez-le à la racine du dossier **resources**. Configurez-le en remplaçant les diverses données pour pouvoir établir la connexion avec votre base de données (globalement, ce sont les mêmes informations de connexion que précédemment).

4 - Sur **Oracle**, supprimez les 3 tables précédemment importées via la script en exécutant le code ci-dessous. Hibernate permet de les régénérer automatiquement.

```
DROP TABLE RessourcesOGE CASCADE CONSTRAINT;
DROP TABLE EtudiantsOGE CASCADE CONSTRAINT;
DROP TABLE NotesOGE CASCADE CONSTRAINT;

DROP SEQUENCE ressources_oge_autoincrement;
DROP SEQUENCE etudiants_oge_autoincrement;
DROP SEQUENCE notes_oge_autoincrement;
```

5 - Référencez les trois classes **Ressource**, **Etudiant** et **Note** dans le fichier de configuration **hibernate.cfg.xml**

Exercice 7 : Configuration des entités

1 - Configurez l'entité **Ressource** en utilisant les différentes **annotations** nécessaires. On souhaite que l'identifiant s'auto-incrémente.

2 - Configurez l'entité **Etudiant** en utilisant les différentes **annotations** nécessaires. On souhaite que l'identifiant s'auto-incrémente. Attention, il y a une association entre l'étudiant et une **ressource** au niveau de la **ressource favorite**. Choisissez judicieusement l'annotation à placer sur cet attribut. Précisez l'action **CASCADE** pour la suppression. Il y a également une autre association spéciale à configurer avec les **notes** (le chargement doit utiliser la stratégie **EAGER**).

3 - Configurez l'entité **Note** en utilisant les différentes **annotations** nécessaires. On souhaite que l'identifiant s'auto-incrémente. Il y a une association avec un **étudiant** et une autre avec une **ressource**. Pour ces deux associations, précisez également l'action **CASCADE** pour la suppression.

Exercice 8 : Modification des services

Maintenant que vos entités sont prêtes, il n'y a plus qu'à modifier vos **services** pour qu'ils utilisent des **repositories** issus de **Hibernate Repositories** au lieu de vos classes du package **sql** !

1 - Modifiez **RessourceService** et **EtudiantService** afin qu'ils utilisent chacun un **repository** adéquat afin de réaliser les différentes opérations de stockage (en remplacement de vos objets StockageRessourceDatabase et StockageEtudiantDatabase). Pour cela, vous vous servirez de la méthode **RepositoryService.getRepository** présentée dans le **support de cours**.

2 - En initialisant un **repository** adéquat comme pour les autres services, complétez les différentes méthodes de la classe **NoteService** afin de la rendre fonctionnelle. Pour rappel, **RessourceService** et **EtudiantService** sont accessibles en utilisant **RessourceService.getInstance()** et **EtudiantService.getInstance()**

3 - Lancez votre application, vérifiez que tout fonctionne bien comme avant. Dorénavant, la visualisation, l'ajout, la modification et la suppression des notes d'un étudiant devrait fonctionner !

4 - Allez observer le contenu de votre base sur Oracle. Comme vous pouvez le constater, les tables ont été générées automatiquement sans que vous ayez à toucher quoi que ce soit !