Séance 1

Architecture Web

Application de gestion de location de véhicules

Exercice 1 : Posons les bases...

Pour commencer ce projet, lancez Eclipse pour le développement d'applications Java EE.

1. Générez un nouveau projet maven avec la commande :
   mvn archetype:generate \
   -DarchetypeGroupId=org.apache.maven.archetypes \
   -DarchetypeArtifactId=maven-archetype-simple \
   -DgroupId=epf \
   -DartifactId=rentmanager
2. Extrayez le contenu du zip rentmanager.zip dans le dossier rentmanager/src/main (enlevez le contenu existant)
3. Dans les properties du projet sélectionnez « Java Compiler » puis cochez "Enable project specific settings" puis dans « Compiler compliance level » sélectionnez 1.8 ou 9. Puis appliquez et fermez.
4. ajouter dans le pom.xml:

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>exec-maven-plugin</artifactId>

<version>1.4.0</version>

<configuration>

<mainClass>com.epf.rentmanager.persistence.FillDatabase</mainClass>

</configuration>

</plugin>

dans la balise <plugins> qui est dans la balise <pluginManagement> qui vous permettra d'utiliser le plugin exec et :

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>1.4.200</version>

</dependency>

Dans la balise <dependencies> pour résoudre les problèmes de dépendances lié

à h2.

Vous pouvez régler les problèmes de dépendances manquantes en créant les classes comme indiqué dans la question 2, ou en supprimant temporairement les classes dans les packages autre que dans le package persistance.

Puis dans le dossier qui contient votre pom.xml, vous pouvez maintenant taper dans un terminal :

```
mvn clean install exec:java
```

Ce qui va initialiser votre base de données (si votre projet compile, ce qui n'est pas le cas avec les fichiers initiaux).

Puis vous pourrez exécuter le main de votre choix en remplaçant dans le plugin exec la mainClass par celle de votre choix.

Vous trouverez dans le répertoire `src`, situé dans « Java Resources », le package `com.excilys.rentmanager`. Il s'agira de notre package « racine ». Tout le code Java que vous serez amenés à écrire devra se situer dans des sous-packages de ce package racine.

Le package `persistence`, situé à l'intérieur du package racine, contient le fichier ConnectionManager.java. Il s'agit de la classe qui vous permettra d'interagir avec la base de données `H2`.

Exercice 2 : La représentation des données

À présent que notre application est initialisée, nous allons pouvoir y ajouter les premiers éléments de notre application web Java : les classes de représentation des données.

1. Créez *dans un endroit approprié* les trois classes permettant de représenter les objets stockés dans les tables `Client`, `Véhicule` de la base de données.

Indications :

- Vous pourrez retrouver en annexe A de ce sujet le schéma des tables de la base de données.
- Pour représenter les éléments de type DATETIME vous utiliserez la classe `java.time.LocalDate` en Java. Vous trouverez la JavaDoc à ce sujet ici.
- Afin de tester vos classes de modèle, il peut être pertinent de redéfinir leur méthode `toString()`. La méthode générée automatiquement par Eclipse est une bonne base et pourra vous faire gagner du temps.

Exercice 3 : L'accès aux données

Ajoutons maintenant la couche d'accès aux données de notre application web Java.

1. Créez, *dans un endroit approprié*, la classe `DaoException` qui hérite de la

classe `Exception`.

2. Complétez les deux classes fournies en respectant la signature des méthodes ainsi que la gestion des exceptions.

   Les méthodes de ces classes sont en principe suffisamment bien nommées pour comprendre ce que chacune est censée faire. Nous allons juste mettre en lumière le fait que les méthodes `create()` de toutes les DAOs ont pour type de retour `int`. En effet, ces deux méthodes retournent l'identifiant du nouvel élément créé en base de données. Pour récupérer l'identifiant de l'élément inséré, vous devrez ajouter un paramètre lors de la création de votre Statement. Un exemple vous est fourni en annexe C de ce sujet.

Indications :

- Pour récupérer au format Java `Date` des données stockées dans un champ DATETIME de la base de données, vous pouvez utiliser la méthode `getDate()` du `ResultSet`. Il est ensuite possible de convertir un objet de type `Date` en objet de type `LocalDate` à l'aide de la méthode `toLocalDate()`. Il est également possible de convertir un objet de type LocalDate en un objet de type Date avec la méthode static `Date.valueOf()`.

Exercice 4 : Manipulation des données par les services

Maintenant que nous avons mis en place les Daos.

1. Créez la classe `ServiceException`, qui hérite de la classe `Exception`.

2. Créez deux classes représentant les services respectifs en rapport aux DAOs précédemment créés. Tout comme pour les DAOs, vous prendrez garde à nommer vos classes de façon cohérente. Vous mettrez en place dans ces classes de service un système de traitement et de vérification des données. En particulier :

   a. On empêchera la création ou la mise à jour d'un `Client` si son nom/prenom est vide. Si une telle opération est tentée, on enverra une `ServiceException`.
   b. On empêchera la création ou la mise à jour d'un `Véhicule` si son constructeur (manufacturer) est vide. On s'assurera également que le nombres de places est supérieure à 1. Si de telles opérations sont tentés, on enverra une `ServiceException`.
   c. Lors de la création ou la mise à jour d'un `Client`, on fera en sorte que son nom de famille soit enregistré en MAJUSCULES dans la base de données.

Indications :

- Les services ont besoin des objets DAOs pour effectuer les appels à la base de données.

Exercice 5 : Interface utilisateur

Nous arrivons quasiment à la dernière étape de notre application web Java : la

mise en place de l'interface en ligne de commande.

1. Créez un package `cli` dans le package `ui`, dans lequel vous placerez vos classes gérant l'interface en ligne de commandes.

2. Les fonctionnalités minimales auxquelles l'interface devra répondre sont les suivantes :

    a. Avoir la possibilité de créer un `Client`.
    b. Pouvoir lister tous les `Clients`.
    c. Avoir la possibilité de créer un `Véhicule`.
    d. Pouvoir lister tous les `Véhicules`.
    e. (Bonus) Pouvoir supprimer un Client.
    f. (Bonus) Pouvoir supprimer un `Véhicule`.

Indications :

- Lors de la création d'un `Client`, plusieurs champs sont obligatoires : nom, prénom, email et date de naissance
- Vous devez effectuer une validation des informations, en particulier le format des dates et la syntaxe de l'email.
- Vous avez à votre disposition une classe IOUtils dans le package `utils`, qui fournit des méthodes vous permettant de lire des entrées en ligne de commande.

Exercice 6 : La Réservation

Une fois que les fonctionnalités de base sont implémentées, il vous sera possible d'ajouter la table `Réservation`.

1. Créez la classe modèle correspondante.
2. Implémentez la couche d'accès au données pour les `Réservations`.
3. Créez le service associé.
4. Ajoutez à l'interface les fonctionnalités suivantes :
    a. Créer une `Réservation`.
    b. Lister toutes les `Réservations`.
    c. Lister toutes les `Réservations` associées à un `Client` donné.
    d. Lister toutes les `Réservations` associées à un `Véhicule` donné.
    e. Supprimer une `Réservation`.

Indications :

- La table `Réservation` contient des clés secondaires faisant le lien avec les tables `Client` et `Véhicule`. Cela peut rendre la suppression des `Clients` ou des `Véhicules` un peu plus compliqué. De plus, dans notre classe de modèle correspondante, nous préférerions stocker directement des objets de type `Client` et `Véhicule`.

Annexes

A. Schéma des tables de la base de données

Client
+ INT **id** PRIMARY KEY AUTOINCREMENT + VARCHAR **nom** + VARCHAR **prenom** + VARCHAR **email** + DATETIME **naissance**

Vehicule
+ INT **id** PRIMARY KEY AUTOINCREMENT + VARCHAR **constructeur** + VARCHAR **modele** + TINYINT **nb_places**

Reservation
+ INT **id** PRIMARY KEY AUTOINCREMENT + INT **client_id** + INT **vehicle_id** + DATETIME **debut** + DATETIME **fin**

B. Requêtes SQL proposées

Client

Lister tous les clients

```
SELECT id, nom, prenom, email, naissance FROM Client;
```

Récupérer un client par son identifiant

```
SELECT id, nom, prenom, email, naissance FROM Client WHERE id = ?;
```

Créer un nouveau client

```
INSERT INTO Client(nom, prenom, email, naissance) VALUES (?, ?, ?, ?);
```

Mettre à jour un client

```
UPDATE Client SET nom = ?, prenom = ?, email = ?, naissance = ? WHERE id = ?;
```

Supprimer un client

```
DELETE FROM Client WHERE id = ?;
```

Compter le nombre de clients total

```
SELECT COUNT(id) AS count FROM Client;
```

Véhicule

Lister tous les véhicules

```
SELECT id, constructeur, modele, nb_places FROM Vehicule BY constructeur;
```

Récupérer un véhicule par son identifiant

```
SELECT id, constructeur, modele, nb_places FROM Vehicule WHERE id = ?;
```

Créer un nouveau véhicule

```
INSERT INTO Vehicule(constructeur, modele, nb_places) VALUES (?, ?);
```

Mettre à jour un véhicule

```
UPDATE Vehicule SET constructeur = ?, modele = ?, nb_places = ? WHERE id = ?;
```

Supprimer un véhicule

```
DELETE FROM Vehicule WHERE id = ?;
```

Compter le nombre de véhicules total

```
SELECT COUNT(id) AS count FROM Vehicule;
```

Réservation

Lister toutes les réservations

```
SELECT id, client_id, vehicule_id, debut, fin FROM Reservation;
```

Récupérer une réservation par l'identifiant du client

```
SELECT id, vehicule_id, debut, fin FROM Reservation WHERE client_id = ?;
```

Récupérer une réservation par son identifiant

```
SELECT id, client_id, debut, fin FROM Reservation WHERE vehicule_id = ?;
```

Créer une nouvelle réservation

```
INSERT INTO Reservation(client_id, vehicule_id, debut, fin)
VALUES (?, ?, ?, ?);
```

Compter le nombre de réservations totales

```
SELECT COUNT(id) AS count FROM Reservation;
```

C. Récupérer l'identifiant de l'élément inséré avec JDBC

Pour récupérer l'identifiant de l'élément inséré dans le cadre d'une table possédant un champ `id` autoincrement, vous devez ajouter l'attribut `Statement.RETURN_GENERATED_KEYS` lors de la création de votre objet Statement :

```
PreparedStatement stmt =
connection.prepareStatement("INSERT ...",
statement.RETURN_GENERATED_KEYS);
```

Vous pouvez ensuite utiliser l'objet `stmt` de façon normale. Une fois que vous aurez exécuté la requête à l'aide de `stmt.executeUpdate()`, vous serez en mesure de récupérer les éléments générés de la façon suivante :

```
ResultSet resultSet = stmt.getGeneratedKeys();
```

Manipulez ensuite votre ResultSet comme dans le cas d'une requête SELECT : parcourez les enregistrements qu'il contient (si tout s'est bien passé il ne doit y avoir qu'un seul élément dans votre ResultSet), et utilisez les méthodes `getType(element)`.

Exemple :

```
if (resultSet.next()) {
    int id = resultSet.getInt(1);
}
```

D. Le design pattern Singleton

Le singleton est un design pattern dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet. Il est utilisé lorsqu'on a besoin exactement d'un objet pour coordonner des opérations dans un système. Le modèle est parfois utilisé pour son efficacité, lorsque le système est plus rapide ou occupe moins de mémoire avec peu d'objets qu'avec beaucoup d'objets similaires. (https://fr.wikipedia.org/wiki/Singleton_(patron_de_conception))

Il existe plusieurs façons de mettre en place le design pattern Singleton. Vous trouverez ci-dessous la façon que nous vous proposons d'utiliser. Si vous souhaitez vous renseigner sur

les autres possibilités, vous pouvez jeter un oeil à l'article suivant :
https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples

Prenons pour exemple une classe que nous nommerons `Singleton`. Nous allons utiliser la « lazy instanciation ». Cela consiste à créer un attribut statique `instance` dont le type est `Singleton`. La classe `Singleton` doit disposer d'exactement un constructeur **privé**. On ajoute à cette classe une méthode statique `getInstance()` ne prenant pas d'argument, et renvoyant l'instance stockée dans l'attribut du même nom. Cette méthode `getInstance()` teste d'abord si l'attribut `instance` est null ou non. Si oui, alors elle appelle le constructeur de la classe pour créer une instance de celle-ci, et la stocke dans l'attribut. Elle renvoie ensuite l'attribut.