T6 - Advanced Java

T-JAV-600

Server

Design pattern

1.0

{EPITECH.}

Server

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

SERVER-JAVA

DESCRIPTION

Bonjour à tous, dans ce workshop nous allons créer un serveur java en utilisant trois design pattern différent.
1) Singleton pour la socket.
2) Builder pour la création d'un objet.
3) Dao pour notre base de donnée.
La raison de ce workshop est vraiment pour vous faire découvrir les designs patterns. Il en existe beaucoup qui sont très différents.
Les trois que nous allons voir sont ceux que j'utilise le plus.
Vous pouvez utiliser ces patterns avec la plupart des langages orienté objet.

DÉMARRAGE

Tout d'abord, télécharger ce repo —> git@github.com:Puigs/server-java.git
Vous pouvez trouver dans ce dossier un client c permettant d'envoyer des messages à votre futur serveur.
Maintenant nous allons télécharger les outils que nous allons avoir besoin pour ce workshop.

```
Terminal
~/T-JAV-600> sudo apt-get update && sudo apt-get upgrade
```

```
Terminal
~/T-JAV-600> sudo apt install maven
```

```
Terminal
~/T-JAV-600> mvn -version
```

```
Terminal
~/T-JAV-600> sudo apt install default-jdk
```

```
Terminal
~/T-JAV-600> mvn archetype:generate -DgroupId=com.mycompany.app
-DartifactId=server -DarchetypeArtifactId=maven-archetype-quickstart
-DarchetypeVersion=1.4 -DinteractiveMode=false
```

Avec cette dernière commande on a créé notre projet.
Pour build notre projet utiliser la commance :

```
Terminal
~/T-JAV-600> mvn package
```

et pour lancer notre projet :

```
Terminal
~/T-JAV-600> mvn exec:java -Dexec.args="9000"
```

Alors si mvn exec:java ne fonctionne pas aller dans le pom.xml et ajouter :

```
<plugin> <groupId>org.codehaus.mojo</groupId> <artifactId>exec-maven-plugin</artifactId> <version>1.2.1</
version> <executions> <execution> <goals> <goal>java</goal> </goals> </execution> </executions> <configuration
> <mainClass>com.mycompany.app.App</mainClass> </configuration> </plugin>
```

SINGLETON

Alors qu'es qu'un Singleton? C'est un patron de conception dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet.
Pourquoi faire ça? Parce que.
Je vais vous demander de créer un dossier dans com.mycompany.app que vous allez appeler src.
Dans ce dossier vous allez créer une classe que vous allez appeler Server.
Votre fichier Server.java doit ressembler à ça :

```
package com.mycompany.app.src;

import java.io.*;
import java.net.*;
public class Server {}
```

Part one : Singleton

Nous allons créer les variables suivantes :

```
private static final Server instance = new Server();
private int my_port;
private Socket socket;
```

Je vous laisse créer votre constructeur qui doit être en private.

Une fois le constructeur créer, vous devez ajouter cette méthode à votre classe :

```
public static final Server getInstance(){
return instance;
}
```

Notre classe est prête (presque du moins parce que le côté serveur n'est pas encore là).
Maintenant pour créer notre objet on va retourner dans le fichier App.java.
On va devoir importer notre classe dans notre ficher :

```
import com.mycompany.app.src.Server;
```

Puis dans le main on va instancier notre objet :

```
Server server = Server.getInstance();
```

Part two : Go faire une socket

Tout d'abord, notre programme va devoir récupérer un argument. Maintenant quand vous executer votre projet ajouter à la fin le nombre 9000 qui sera notre port.

Dans le main, je vous laisse vérifier que vous avez bien reçu le port et je vous laisse transformer votre String en int. Nous allons en avoir besoin pour notre serveur.

On va retourner dans notre classe Server et on va ajouter une méthode :

```
public void init(int port);
```

Dans cette méthode nous allons récupérer notre port en faisant :

```
this.my_port = port;
```

Dans notre fonction on va ajouter un try and catch :

```
try (ServerSocket serverSocket = new ServerSocket(port)){
} catch (IOException ex){
System.out.println("Server exception: " + ex.getMessage());
ex.printStackTrace();
}
```

Dans notre try on va faire un while (true) où on va attendre que notre client se connecte à notre socket.
Pour que le client puisse se connecter nous devons utiliser la méthode accept de notre objet ServerSocket, on va récupérer le retour dans la variable Socket que nous avons créer plus tôt.
Une fois cette variable rempli et que donc notre client s'est connecté nous allons quitter notre méthode init.
Maintenant on va créer notre méthode qui va lire les messages envoyés de notre client.

```
public String do_read(){}
```

Voici à quoi ressemble notre méthode :

```
public String do_read() {
String str = "";
try {
/
Ici vous devez lire les messages reçu par votre serveur
Insérez ici votre code
/
OutputStream output = this.socket.getOutputStream();
PrintWriter writer = new PrintWriter(output, true);
writer.println("Votre message (" + str + ") a bien été reçu");
} catch (IOException ex) {
System.out.println("Server exception:" + ex.getMessage());
ex.printStackTrace();
}
return str;
}
```

Maintenant il ne reste plus qu'à appeler cette méthode dans une boucle while dans main et notre socket est prête!

Builder

Nous arrivons maitenant à la partie deux de notre serveur.
Alors à quoi sert le design pattern Builder?
Si je fais la célébre commande man google j'obtiens ceci :
« Le pattern Builder est utilisé pour la création d'objets complexes dont les différentes parties doivent être créées suivant un certain ordre ou algorithme spécifique. Une classe externe contrôle l'algorithme de construction »
Typiquement, ce design pattern sera préconisé dans les situations où au moins :
-l'objet final est imposant, et sa création complexe ;
-beaucoup d'arguments doivent être passés à la construction de l'objet, afin d'avoir un design lisible ;
-certains de ces arguments sont optionnels (par ex. : un bateau n'a pas forcément de capitaine, mais toujours une taille), ou ont plusieurs variations (la taille peut par exemple être passée en mètres ou en pieds).

Ce design pattern est du coup tout désigné pour notre deuxième partie où nous allons créer un objet complexe.
En effet notre objet devra contenir… Juste deux string… Une string nom et une prénom…
Bon d'accord l'utilisation de design pattern n'est pas pertinante dans cette exercice mais bon… Faute de mieux on part sur ça.

Alors, pour réaliser ce builder vous allez créer deux classes :
–> Personne.java
–> PersonneBuilder.java

Dans Personne.java je vous laisse créer deux variables String, et les célèbres Getteur et Setteurs de ces variables.
Pour l'instant rien de compliqué, maintenant passons à notre classe PersonneBuilder.java :
Nous allons trouver dans notre classe :

—> De nouveau nos deux String
—> Des méthodes Setteur, voici un exemple :

```
public PersonneBuilder set_nom(String _nom){
}
```

—> Notre méthode :

```
public Personne build(){}
```

Voilà à quoi ressemble la création de nos objets dans le main :

```
Personne obj = new PersonneBuilder().set_nom(mots[0]).set_prenom(mots[1]).build();
```

DAO

Un dao (ou Data access object) propose de regrouper les accès aux données persistantes dans des classes à part, plutôt que de les disperser. Il s'agit surtout de ne pas écrire ces accès dans les classes "métier", qui ne seront modifiées que si les règles de gestion métier changent.
L'utilisation de DAO permet de s'abstraire de la façon dont les données sont stockées au niveau des objets métier. Ainsi, le changement du mode de stockage ne remet pas en cause le reste de l'application. En effet, seules ces classes dites "techniques" seront à modifier (et donc à re-tester). Cette souplesse implique cependant un coût additionnel, dû à une plus grande complexité de mise en œuvre.

SQLite

Ajouter à votre pom.xml

```
<dependency> <groupId>org.xerial</groupId> <artifactId>sqlite-jdbc</artifactId> <version>3.32.3.2</version
> </dependency>
```

DAO.java

Première classe à créer pour cette partie —>

package com.mycompany.app.src;

```
import java.util.List;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public abstract class Dao<T> {
public Connection connection = null;
public Statement statement = null;

public abstract void init();
public abstract List<T> getAllPersonnes();
public abstract void addPersonne(T personne);
public abstract void deletePersonne(T personne);
}
```

PersonneDAO.java

Et pour finir notre dernière classe à faire et notre PersonneDAO.java
Tout d'abord —>

```
package com.mycompany.app.src;

import com.mycompany.app.src.Dao;
import com.mycompany.app.src.Personne;
import java.util.*;
import java.util.List;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

Je vais vous "donner" les fonctions importante du dao, à vous de remplir la dernière méthode (deletePersonne).

```
public class PersonneDao extends Dao<Personne> {

    private List<Personne> personnes;

    public void init() {
        try {
            this.connection = DriverManager.getConnection("jdbc:sqlite:sample.db");
            this.statement = this.connection.createStatement();
            this.statement.setQueryTimeout(30);
            this.statement.executeUpdate("drop table if exists person");
            this.statement.executeUpdate("create table person (nom string, prenom
                string)");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public List<Personne> getAllPersonnes() {
        try {
            ResultSet rs = statement.executeQuery("select * from person");
            while(rs.next())
            {
                System.out.println("nom = " + rs.getString("nom"));
                System.out.println("prenom = " + rs.getString("prenom"));
                Personne obj = new PersonneBuilder().set_nom(rs.getString("nom")).
                    set_prenom(rs.getString("prenom")).build();
                this.personnes.add(obj);
            }
            return personnes;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return personnes;
    }

    public void addPersonne(Personne personne) {
        try {
            statement.executeUpdate("insert into person values('"+personne.get_nom()
                +"', '"+personne.get_prenom()+"')");
        } catch (SQLException e) {
            e.printStackTrace();
```

```
            }
        }

        public void deletePersonne(Personne personne) {

        }
    }
```