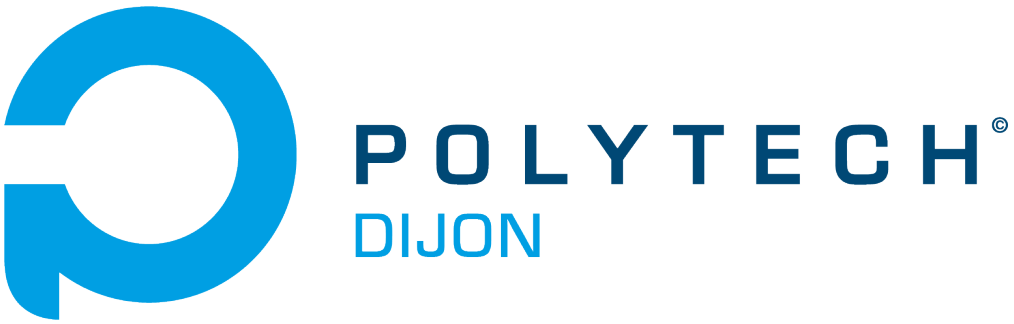
Polytech Dijon 3A options : Informatique & Réseaux

Module : ITC315

TD JAVA BDD

Auteur :
HUBERT Matéo

Professeur référent :
MEUNIER Charles



2023-2024

Table des matières

Table des figures

1 PREPARATION

L'entièreté du code est disponible sur ce repo

1.1 MYSQL

Pour commencer, il faut installer xampp pour pouvoir lancer les différents services nécessaires au TP.

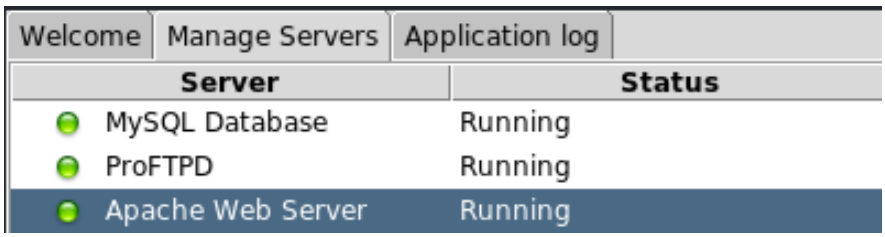


FIGURE 1 – XAMPP Panel Control

Ensuite on créer la base de données pour pouvoir faire les requêtes SQL.

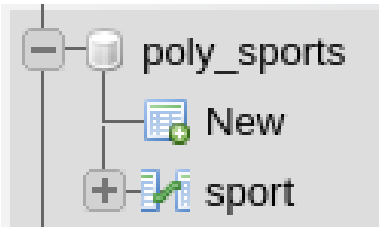


FIGURE 2 – Base de données poly_sport

1.2 DRIVER MYSQL

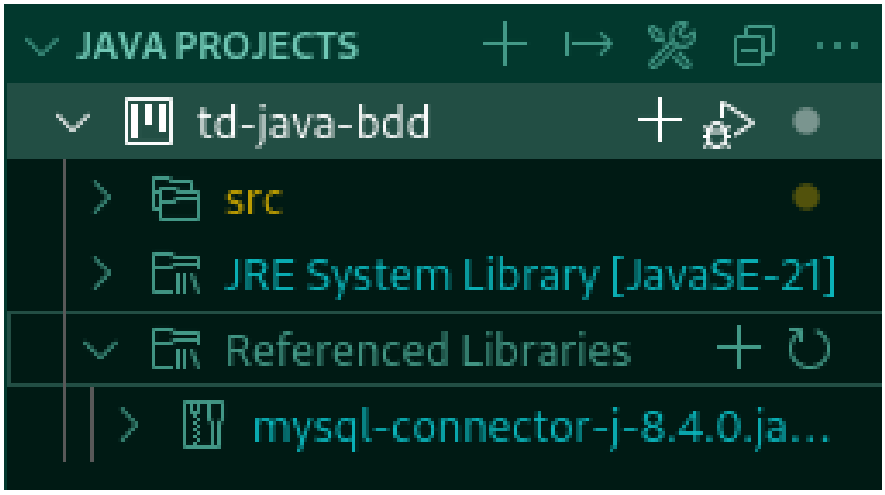


FIGURE 3 – Bibliothèque externe

On importe le driver JDBC dans le projet java afin d'avoir accès aux différentes méthodes nécessaires pour réaliser les connexions et commandes sur la base de données.

2 EXPLICATIONS DU CODE

```java
import java.util.ArrayList;
import java.util.Scanner;

public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        PolySportsDatabase poly_sport = PolySportsDatabase.getInstance();
        poly_sport.connect();
        SportsDAO dao = new SportsDAO(poly_sport);
        ArrayList<Sport> sports = dao.findAll();
        Sport sport = dao.findById(sport:1);
        Sport sport_inexistant = dao.findById(sport:15);
        Scanner myScanner = new Scanner(System.in);
        String input = myScanner.nextLine();
        ArrayList<Sport> sports_name = dao.findByName(input);
        for(int i = 0; i < sports.size(); i++){
            System.out.println("Pour le sport: " + sports.get(i).getName() + " il faut: " + spo
        }
        System.out.println("Pour le sport: " + sport.getName() + " il faut: " + sport.getRequir

        try {
            System.out.println("Pour le sport: " + sport_inexistant.getName() + " il faut: " +
        } catch (Exception e) {
            System.err.println("Le sport demandé n'existe pas");
        }
        for(int i = 0; i < sports_name.size(); i++){
            System.out.println("Pour le sport: " + sports_name.get(i).getName() + " il faut: "
        }
    }
}
```

FIGURE 4 – APP.JAVA

Le but de App.java est de créer toutes les instances de classes nécessaires au fonctionnement de l'application. Dans un premier temps, on créer une instance de PolySportsDatabase qui est un singleton que l'on utilise pour se connecter à la base de données. On créer ensuite un objet dao qui servira à récupérer soit un sport ciblé par un nom ou un id ou tous les sports de la base de données. On appelle la méthode findAll, findById, findByName les unes à la suite des autres et on affiche le résultat.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;


public class MYSQLDatabase {
    private String host;
    private int port;
    private String databaseName;
    private String user;
    private String password;
    private Connection connection;
    private static boolean driverLoaded;
    MYSQLDatabase(String host, int port, String databaseName, String user, String password){
        this.host = host;
        this.port = port;
        this.databaseName = databaseName;
        this.user = user;
        this.password = password;
        connection = null;
        driverLoaded = false;
        loadDriver();
    }
    public void connect(){
        try {
            connection = DriverManager.getConnection(
                "jdbc:mysql://"+host+":"+port+"/" + databaseName+"?allowMultiQueries=true",
                user ,
                password
                );
            //connection.createStatement();
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
```

FIGURE 5 – MYSQLDATABASE.JAVA partie 1

Dans cette première partie de code, on crée les différents attributs privés que l'on utilisera dans la classe. Le constructeur qui reçoit les informations pour la connexion initialise les attributs avec les valeurs données en paramètres et appel la fonction loadDriver qui sert à charger le driver JDBC si celui-ci ne l'est pas déjà et ce dans le but de ne pas le charger à chaque fois que l'on créer une connexion. La méthode connect permet de se connecter à la base de données en utilisant une url spécifique ainsi qu'un nom d'utilisateur et d'un mot de passe.

```java
public Statement createStatement(){
    Statement statement = null;
    try {
        statement = connection.createStatement();
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
    return statement;
}
private static void loadDriver(){
    if(driverLoaded != true){
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println(e.getMessage());
        }
        driverLoaded = true;
    }
}
public PreparedStatement prepareStatement(String commande){
    PreparedStatement mypreparedSatement = null;
    try {
        mypreparedSatement = this.connection.prepareStatement(commande);
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
    return mypreparedSatement;
}
```

Figure 6 – MYSQLDATABASE.JAVA partie 2

Dans cette seconde partie, on définit la méthode createStatement qui permet de créer un statement sur la connexion précédemment effectuée avec connect. La méthode loadDriver permet de vérifier si le driver a déjà été chargé ou non et sinon de le charger. La méthode prepareStatement permet de créer des statements, mais de spécifier où et quels "types" les paramètres auront dans le but d'empêcher les injections SQL.

```java
public class PolySportsDatabase extends MYSQLDatabase{
    private static PolySportsDatabase instance;
    private PolySportsDatabase(){
        super(host:"localhost", port:3307, databaseName:"poly_sports", user:"mateo", password:"esirem");
        instance = null;
    }
    public static PolySportsDatabase getInstance(){
        if(instance == null){
            instance = new PolySportsDatabase();
        }
        return instance;
    }
}
```

Figure 7 – POLYSPORTSDATABASE.JAVA

La classe PolySportsDatabase a pour objectif de gérer la connexion à la base de données, mais aussi de limiter le nombre de connexion à 1 pour ne pas créer des connexions à chaque requête. Cela est un avantage sur les petits systèmes, mais peut s'avérer être un problème de performance sur de plus grosses applications. Comme on peut le voir, le constructeur de la classe est privé et c'est grâce à ce principe que l'on fait de l'objet instance un singleton.

```java
public class Sport {
    private int id;
    private String name;
    private int requiredParticipants;

    public Sport(int id, String name, int requiredParticipants){
        this.id = id;
        this.name = name;
        this.requiredParticipants = requiredParticipants;
    }
    public int getId(){
        return this.id;
    }
    public String getName(){
        return this.name;
    }
    public int getRequiredParticipants(){
        return this.requiredParticipants;
    }
}
```

FIGURE 8 – SPORT.JAVA

La classe Sport permet de gérer un sport en retournant son id, son nom et le nombre de participants requis. Pour cela, on créer les 3 attributs privés qui conserveront les informations du sport en question et qui seront instanciés dans le constructeur. La classe possède 3 getters pour obtenir l'id, le nom et le nombre de joueurs requis d'un sport.

```java
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;

public class SportsDAO {
    private MYSQLDatabase database;

    public SportsDAO(MYSQLDatabase database){
        this.database = database;
    }
    public ArrayList<Sport> findAll(){
        ArrayList<Sport> sports = new ArrayList<Sport>();
        Statement myStatement = database.createStatement();
        try {
            ResultSet myResults = myStatement.executeQuery("SELECT `name`, `id`, `required_participants` FROM `sport`;");
            while(myResults.next()){
                int id = myResults.getInt("id");
                String name = myResults.getString("name");
                int required_participants = myResults.getInt("required_participants");
                Sport sport = new Sport(id, name, required_participants);
                sports.add(sport);
            }
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
        return sports;
    }
```

FIGURE 9 – SPORTDAO.JAVA partie 1

La classe SportDAO possède l'essentiel des méthodes permettant de faire les requêtes SQL. Dans un premier temps, il y a la méthode findAll qui retourne un tableau de tous les sports présent dans la BDD. Pour faire cela, on créer un tableau vide. On créer ensuite un statement sur la base de données qui a été fourni lors de la création de l'objet. Ensuite, on exécute notre requête SQL. Pour chaque ligne dans la réponse, on récupère l'id, le nom et le nombre de participants requis et on créer un sport avec ces informations que l'on ajoute à notre tableau de sports. Une fois l'opération effectuée pour toutes les lignes de la réponse, on retourne le tableau pour l'utiliser dans App.java.

```java
public Sport findById(int sport){
    //Statement myStatement = database.createStatement();
    String commande = "SELECT `name`, `id`, `required_participants` FROM `sport` WHERE `id` = ?;";
    PreparedStatement mypreparedStatement = database.prepareStatement(commande);
    try {
        mypreparedStatement.setInt(1, sport);
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
    Sport sport_voulu = null;
    try {
        //ResultSet myResults = myStatement.executeQuery("SELECT `name`, `id`, `required_participar
        ResultSet myResults = mypreparedStatement.executeQuery();
        while(myResults.next()){
            int id = myResults.getInt("id");
            String name = myResults.getString("name");
            int required_participants = myResults.getInt("required_participants");
            sport_voulu = new Sport(id, name, required_participants);
        }
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
    return sport_voulu;
}
```

FIGURE 10 – SPORTDAO.JAVA partie 2

Dans cette seconde partie de code, on créer la méthode findById qui a pour but de retourner le sport ayant l'id fournit en paramètre. Pour empêcher les injections SQL, on n'utilise non pas des statements, mais des preparedStatements. On définit alors une requête type et l'on remplace l'id par un point d'interrogation. On prépare ensuite notre statement en donnant en paramètre la commande. Il faut par la suite indiquer que l'on veut remplacer le premier point d'interrogation par le contenu de la variable sport. Une fois, cela fait, on peut exécuter notre requête et traiter le résultat et le retourner.

```java
public ArrayList<Sport> findByName(String nom){
    //Statement myStatement = database.createStatement();
    String commande = "SELECT * FROM `sport` WHERE `name` LIKE ? ORDER BY `name`;";
    PreparedStatement mypreparedStatement = database.prepareStatement(commande);
    try {
        mypreparedStatement.setString(1, "%"+nom+"%");
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
    ArrayList<Sport> sports = new ArrayList<Sport>();
    try {
        //ResultSet myResults = myStatement.executeQuery("SELECT * FROM `sport` WHERE `
        ResultSet myResults = mypreparedStatement.executeQuery();
        while(myResults.next()){
            int id = myResults.getInt("id");
            String name = myResults.getString("name");
            int required_participants = myResults.getInt("required_participants");
            Sport temp_sport = new Sport(id, name, required_participants);
            sports.add(temp_sport);
        }
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
    return sports;
}
```

FIGURE 11 – SPORTDAO.JAVA partie 3

Dans cette dernière partie, on créer la méthode findByName qui doit retourner tous les sports où le nom donné en paramètre figure. C'est-à-dire que si on donne en paramètre "bad", il faut que tous les sports dont le nom contient le mot "bad" soient retourner. Pour cela, une fois de plus, on utilise des preparedStatements. On indique que l'on remplace le point d'interrogation par la variable nom qui est donné en paramètre et qui est saisie dans le terminal par l'utilisateur lors de l'exécution du programme. La méthode retourne alors tous les sports ayant dans leur nom le mot saisie par l'utilisateur tout en empêchant les injections SQL.

3 PREMIER PAS AVEC JDBC

3.1 PREMIERE REQUETE

```
┌──(azymut㉿hp)-[~/…/Meunier/Java/Java_BDD/td-java-bdd]
└─$ cd /home/azymut/Desktop/3A/S2/Meunier/Java/Java_BDD/td-java-bdd ; /usr/bi
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Pour le sport: Badminton (simple), il faut: 2 participants
Pour le sport: Badminton (double), il faut: 4 participants
Pour le sport: Basket, il faut: 10 participants
```

FIGURE 12 – Première requête

Comme on peut le voir sur la capture ci-dessus, on obtient bien la liste de tous les sports avec toutes les informations de chaque sport.

4 STRUCTURONS TOUT CELA

4.1 MYSQLDATABASE

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class MYSQLDatabase {
    private String host;
    private int port;
    private String databaseName;
    private String user;
    private String password;
    private Connection connection;
    private static boolean driverLoaded;
    MYSQLDatabase(String host, int port, String databaseName, String user, String password){
        this.host = host;
        this.port = port;
        this.databaseName = databaseName;
        this.user = user;
        this.password = password;
        connection = null;
        driverLoaded = false;
        loadDriver();
    }
    public void connect(){
        try {
            connection = DriverManager.getConnection(
                "jdbc:mysql://"+host+":"+port+"/" + databaseName,
                user ,
                password
                );
            connection.createStatement();
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
    public Statement createStatement(){
        Statement statement = null;
        try {
            statement = connection.createStatement();
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
        return statement;
    }
    private static void loadDriver(){
        if(driverLoaded != true){
            try {
                Class.forName("com.mysql.cj.jdbc.Driver");
            } catch (ClassNotFoundException e) {
                System.err.println(e.getMessage());
            }
            driverLoaded = true;
        }
    }
}
```

FIGURE 13 – MySQLDatabase.java

```java
public class App {
    Run|Debug
    public static void main(String[] args) throws Exception {
        MYSQLDatabase mysql = new MYSQLDatabase(host:"localhost", port:3307, databaseName:"poly_sports", user:"mateo", password:"esirem");
        mysql.connect();
    }
}
```

FIGURE 14 – App.java

```
(azymut㉿hp)-[~/Desktop/temp/Java_BDD/td-java-bdd]
└─$ cd /home/azymut/Desktop/temp/Java_BDD/td-java-bdd ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-amd64/bin/java @/tmp/cp_2ftzymh0z5162xj01m3eu596v.argfile App
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
```

FIGURE 15 – Test de fonctionnement MySQLDatabase

Comme on peut le voir sur la capture ci-dessus, la gestion de la connexion à la base de données via la création d'un statement avec la classe MySQLDatabase ne pose pas de problème.

4.2 POLYSPORTDATABASE

```java
public class PolySportsDatabase extends MYSQLDatabase{
    private static PolySportsDatabase instance;
    private PolySportsDatabase(){
        super(host:"localhost", port:3307, databaseName:"poly_sports", user:"mateo", password:"esirem");
        instance = null;
    }
    public static PolySportsDatabase getInstance(){
        if(instance == null){
            instance = new PolySportsDatabase();
        }
        return instance;
    }
}
```

FIGURE 16 – PolySportsDatabase.java

```java
public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        PolySportsDatabase poly_sport = PolySportsDatabase.getInstance();
        poly_sport.connect();
    }
}
```

FIGURE 17 – App.java

```
┌──(azymut㉿hp)-[~/Desktop/temp/Java_BDD/td-java-bdd]
└─$ cd /home/azymut/Desktop/temp/Java_BDD/td-java-bdd ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-amd64/bin/java @/tmp/cp_2ftzymh0z5162xj01m3eu596v.argfile App
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true

┌──(azymut㉿hp)-[~/Desktop/temp/Java_BDD/td-java-bdd]
└─$
```

FIGURE 18 – Test de fonctionnement PolySportsDatabase

Comme on peut le voir encore une fois, la gestion de la connexion via la classe PolySportsDatabase ne pose pas de problème.

4.3 SPORT

```java
public class Sport {
    private int id;
    private String name;
    private int requiredParticipants;

    public Sport(int id, String name, int requiredParticipants){
        this.id = id;
        this.name = name;
        this.requiredParticipants = requiredParticipants;
    }
    public int getId(){
        return this.id;
    }
    public String getName(){
        return this.name;
    }
    public int getRequiredParticipants(){
        return this.requiredParticipants;
    }
}
```

FIGURE 19 – Sport.java

```java
public class App {
    Run|Debug
    public static void main(String[] args) throws Exception {
        PolySportsDatabase poly_sport = PolySportsDatabase.getInstance();
        poly_sport.connect();
        Sport sport = new Sport(id:2, name:"natation", requiredParticipants:1);
        System.out.println("Pour faire le sport: " + sport.getName()+ ", il faut: " + sport.getRequiredParticipants() + " participant et c'est le sport: " + sport.getId());
    }
}
```

FIGURE 20 – App.java

```
(azymut㉿hp)-[~/Desktop/temp/Java_BDD/td-java-bdd]
$ cd /home/azymut/Desktop/temp/Java_BDD/td-java-bdd ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-amd64/bin/java @/tmp/cp_2ftzymh0z5162xj01m3eu596v.argfile App
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Pour faire le sport: natation, il faut: 1 participant et c'est le sport: 2
```

FIGURE 21 – Test de fonctionnement Sport

Comme on peut le voir, on arrive bien à récupérer les informations d'un sport.

4.4 SPORTSDAO

```java
public class SportsDAO {
    private MYSQLDatabase database;

    public SportsDAO(MYSQLDatabase database){
        this.database = database;
    }
    public ArrayList<Sport> findAll(){
        ArrayList<Sport> sports = new ArrayList<>();
        Statement myStatement = database.createStatement();
        try {
            ResultSet myResults = myStatement.executeQuery("SELECT `name`, `id`, `required_participants` FROM `sport`;");
            while(myResults.next()){
                int id = myResults.getInt("id");
                String name = myResults.getString("name");
                int required_participants = myResults.getInt("required_participants");
                Sport sport = new Sport(id, name, required_participants);
                sports.add(sport);
            }
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
        return sports;
    }
}
```

FIGURE 22 – SportDAO.java

```java
public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        PolySportsDatabase poly_sport = PolySportsDatabase.getInstance();
        poly_sport.connect();
        SportsDAO dao = new SportsDAO(poly_sport);
        ArrayList<Sport> sports = dao.findAll();
        for(int i = 0; i < sports.size(); i++){
            System.out.println("Pour le sport: " + sports.get(i).getName() + " il faut: " + sports.get(i).getRequiredParticipants() + " participants et c'est le sport ayant pour id: " + sports.get(i).getId());
        }
    }
}
```

FIGURE 23 – App.java

```
┌──(azymut㉿hp)-[~/Desktop/temp/Java_BDD/td-java-bdd]
└─$  cd /home/azymut/Desktop/temp/Java_BDD/td-java-bdd ; /usr/bin/env /usr/lib/jvm/j
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Pour le sport: Rugby il faut: 22 participants et c'est le sport ayant pour id: 1
Pour le sport: natation il faut: 1 participants et c'est le sport ayant pour id: 2
```

FIGURE 24 – Test de fonctionnement SportDAO

Comme on peut le voir sur la capture ci-dessus, on récupère bien tous les sports de la base de données (ici différents car recrés après la suppression avec la partie injections SQL).

4.5 FINDBYID

```java
public Sport findById(int sport){
    Statement myStatement = database.createStatement();
    Sport sport_voulu = null;
    try {
        ResultSet myResults = myStatement.executeQuery("SELECT `name`, `id`, `required_participants` FROM `sport` WHERE `id` = "+sport+";");
        while(myResults.next()){
            int id = myResults.getInt("id");
            String name = myResults.getString("name");
            int required_participants = myResults.getInt("required_participants");
            sport_voulu = new Sport(id, name, required_participants);
        }
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
    return sport_voulu;
}
```

FIGURE 25 – Méthode findById

```java
Sport sport = dao.findById(sport:1);
Sport sport_inexistant = dao.findById(sport:15);
```

FIGURE 26 – App.java

```
┌──(azymut㉿hp)-[~/Desktop/temp/Java_BDD/td-java-bdd]
└─$  cd /home/azymut/Desktop/temp/Java_BDD/td-java-bdd ; /usr/bin/env /usr/lib/jvm/
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Pour le sport: Rugby il faut: 22 participants et c'est le sport ayant pour id: 1
Pour le sport: natation il faut: 1 participants et c'est le sport ayant pour id: 2
Pour le sport: Rugby il faut: 22 participants et c'est le sport ayant pour id: 1
Le sport demandé n'existe pas
```

FIGURE 27 – Test de fonctionnement findById

Comme on peut le voir, on a toujours l'affichage de tous les sports de la base de données qui sont les deux premières lignes puis on affiche le sport avec l'id 1 et enfin quand un sport qui n'existe pas est demandé, on capture l'exception et on affiche un message pour dire que le sport n'existe pas.

4.6 FINDBYNAME

```java
public ArrayList<Sport> findByName(String nom){
    Statement myStatement = database.createStatement();
    ArrayList<Sport> sports = new ArrayList<Sport>();
    try {
        ResultSet myResults = myStatement.executeQuery("SELECT * FROM `sport` WHERE `name` LIKE '%"+nom+"%';");
        while(myResults.next()){
            int id = myResults.getInt("id");
            String name = myResults.getString("name");
            int required_participants = myResults.getInt("required_participants");
            Sport temp_sport = new Sport(id, name, required_participants);
            sports.add(temp_sport);
        }
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
    return sports;
}
```

FIGURE 28 – Méthode findByName

```java
Scanner myScanner = new Scanner(System.in);
String input = myScanner.nextLine();
ArrayList<Sport> sports_name = dao.findByName(input);
```

FIGURE 29 – App.java

```
└─$  cd /home/azymut/Desktop/temp/Java_BDD/td-java-bdd ; /usr/bin/env /usr/lib/jvm/java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
natation
Pour le sport: Rugby il faut: 22 participants et c'est le sport ayant pour id: 1
Pour le sport: natation il faut: 1 participants et c'est le sport ayant pour id: 2
Pour le sport: Rugby il faut: 22 participants et c'est le sport ayant pour id: 1
Le sport demandé n'existe pas
Pour le sport: natation il faut: 1 participants et c'est le sport ayant pour id: 2
```

FIGURE 30 – Test de fonctionnement findByName

Comme on peut le voir à la dernière ligne, si l'on demande le sport natation, celui-ci est affiché sans erreur.

5 INJECTION SQL

5.1 ALLOWMULTIQUERIES

```
'; DELETE FROM sport;
```

FIGURE 31 – Injection SQL

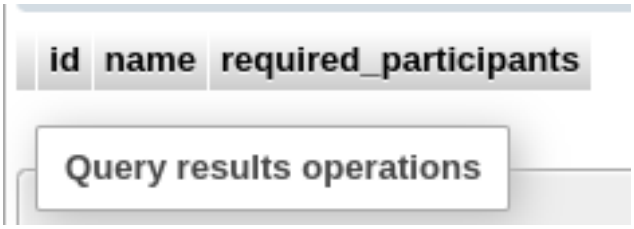


FIGURE 32 – BDD après injection

Comme on peut le voir après exécution de la commande, l'entièreté de la BDD est supprimée.

5.2 PREPAREDSTATEMENT

```
public PreparedStatement prepareStatement(String commande){
    PreparedStatement mypreparedSatement = null;
    try {
        mypreparedSatement = this.connection.prepareStatement(commande);
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
    return mypreparedSatement;
}
```

FIGURE 33 – Méthode prepareStatement

```
public Sport findById(int sport){
    //Statement myStatement = database.createStatement();
    String commande = "SELECT `name`, `id`, `required_participants` FROM `sport` WHERE `id` = ?;";
    PreparedStatement mypreparedStatement = database.prepareStatement(commande);
    try {
        mypreparedStatement.setInt(1, sport);
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
```

FIGURE 34 – Modification de la méthode findById

```
public ArrayList<Sport> findByName(String nom){
    //Statement myStatement = database.createStatement();
    String commande = "SELECT * FROM `sport` WHERE `name` LIKE ? ORDER BY `name`;";
    PreparedStatement mypreparedStatement = database.prepareStatement(commande);
    try {
        mypreparedStatement.setString(1, "%"+nom+"%");
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
```

FIGURE 35 – Modification de la méthode findByName

Après ces modifications, les requêtes sont toujours possibles, mais il n'est plus possible de faire des injections SQL.