Une introduction à JDBC

1 Mise en place

1.1 Utiliser HyperSQL

Pour nos exemples, nous allons utiliser la base de données embarquée HyperSQL [1]. N'oubliez pas de lire la documentation HyperSQL [2]. Elle se présente comme une librairie à ajouter à vos projet.

1.2 Spring Boot pour JDBC

- Reprenez le projet Eclipse du TP précédent.
- Ajoutez à votre fichier `pom.xml` les dépendances ci-dessous. Elles permettent de charger le *Starter JDBC* de **Spring Boot** et ainsi d'inclure les outils dont nous avons besoin.

À ajouter au fichier pom.xml

```
...
    <!-- pour utiliser jdbc Spring -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <!-- pour utiliser HyperSQL -->
    <dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
    </dependency>
...
```

- Créez le package `myapp.jdbc` dans les sources et dans les tests.

2 Premiers programmes JDBC

- Commencez par créer la classe ci-dessous. C'est un exemple de classe DAO (*Data Access Object*) qui réalise les actions CRUD (*Create Read Update Delete*) sur des données très simples.

1. http://hsqldb.org/
2. ref:doc-hsqldb

```java
package myapp.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Collection;
import java.util.LinkedList;

import javax.annotation.PostConstruct;

import org.springframework.stereotype.Service;

@Service
public class NameDao {

    final String url = "jdbc:hsqldb:file:databases/test-jdbc";
    final String user = "SA";
    final String password = "";

    private Connection newConnection() throws SQLException {
        return DriverManager.getConnection(url, user, password);
    }

    @PostConstruct
    public void initSchema() throws SQLException {
        var query = "create table if not exists NAME (" //
                + " id integer not null, " //
                + " name varchar(50) not null, " //
                + " primary key (id) )";
        try (var conn = newConnection()) {
            conn.createStatement().execute(query);
        }
    }

    public void addName(int id, String name) throws SQLException {
        var query = "insert into NAME values (?,?)";
        try (var conn = newConnection()) {
            var st = conn.prepareStatement(query);
            st.setInt(1, id);
            st.setString(2, name);
            st.execute();
        }
    }

    public void deleteName(int id) throws SQLException {
        var query = "Delete From NAME where (id = ?)";
        try (var conn = newConnection()) {
            var st = conn.prepareStatement(query);
            st.setInt(1, id);
            st.execute();
        }
    }

    public String findName(int id) throws SQLException {
        var query = "Select * From NAME where (id = ?)";
        try (var conn = newConnection()) {
            var st = conn.prepareStatement(query);
            st.setInt(1, id);
            var rs = st.executeQuery();
            if (rs.next()) {
                return rs.getString("name");
            }
        }
        return null;
    }

    public Collection<String> findNames() throws SQLException {
```

- **Remarque** : chaque méthode a la même structure (connexion, utilisation, fermeture). Cette classe est donc sûre dans une environnement multi-threads.
- Vous pouvez ensuite créer la classe de test et vérifier son bon fonctionnement :

```java
package myapp.jdbc;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

import java.sql.SQLException;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
public class TestNameDao {

    @Autowired
    NameDao dao;

    @Test
    public void testNames() throws SQLException {
        dao.deleteName(100);
        dao.deleteName(200);
        dao.addName(100, "Hello");
        dao.addName(200, "Salut");
        assertEquals("Hello", dao.findName(100));
        assertEquals("Salut", dao.findName(200));
        dao.findNames().forEach(System.out::println);
    }

    @Test
    public void testErrors() throws SQLException {
        dao.deleteName(300);
        assertThrows(SQLException.class, () -> {
            dao.addName(300, "Bye");
            dao.addName(300, "Au␣revoir");
        });
        assertEquals("Bye", dao.findName(300));
    }

}
```

Travail à faire :

- Étudiez le **try-with-resources** mise en oeuvre dans ces méthodes (explications [3]).
- Ajoutez à la classe `NameDao` une méthode de mise à jour (`updateName`) et testez-là.
- Vérifiez par un test unitaire que les injections SQL ne sont pas possibles pour cette méthode.
- Les données sont stockés dans des fichiers. Consultez, dans votre projet, les fichiers `databases/*.script` et `databases/*.log` .

3. http ://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

3 Utiliser une DataSource

3.1 Mise en place

Une *DataSource* permet de centraliser la phase de paramétrage et ainsi de simplifier le code JDBC. Ces objets permettent également de gérer un ensemble de connexions ouvertes prêtes à être utilisées. Nous évitons ainsi l'ouverture de nouvelles connexions à chaque transaction (opération très coûteuse pour la BD).

Nous allons utiliser les dataSources Hikari fournies par Spring que nous allons configurer **en ajoutant** ces lignes dans notre fichier `src/main/resources/application.properties` :

```
spring.datasource.url=jdbc:hsqldb:file:databases/myBase
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.driver-class-name=org.hsqldb.jdbc.JDBCDriver
```

Nous pouvons ainsi simplifier notre classe `NameDao` :

```
...

    @Autowired
    DataSource dataSource;

    // Nouvelle version
    private Connection newConnection() throws SQLException {
        return dataSource.getConnection();
    }

...
```

Nous pouvons également construire une classe de configuration afin de maitriser la création de nos ressources :

```java
package myapp.jdbc;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

@Configuration
public class SpringJdbcConfig {

    @Bean
    public DataSource myDataSource(//
            @Value("${spring.datasource.url}") String url, //
            @Value("${spring.datasource.username}") String user, //
            @Value("${spring.datasource.password}") String password//
    ) {
        System.out.println("---␣my␣datasource");
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl(url);
        config.setUsername(user);
        config.setPassword(password);
        // le pool de connexion doit gérer
        // entre 5 et 10 connections prêtes.
        config.setMinimumIdle(5);
        config.setMaximumPoolSize(10);
        return new HikariDataSource(config);
    }

}
```

3.2 Pool de connexions

Nous allons maintenant illustrer le pool de connexions géré par la `DataSource` :

- Préparez dans `NameDao` un travail long (une seconde) :

```java
public void longWork() {
    try (var c = newConnection()) {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
    } catch (SQLException e1) {
    }
}
```

- Préparez un test qui va créer cinq threads pour tester l'allocation des connexions :

```java
@Test
public void testWorks() throws Exception {
    long debut = System.currentTimeMillis();

    // exécution des threads
    ExecutorService executor = Executors.newFixedThreadPool(10);
    for (int i = 1; (i < 5); i++) {
        executor.execute(dao::longWork);
    }

    // attente de la fin des threads
    executor.shutdown();
    executor.awaitTermination(10, TimeUnit.HOURS);

    // calcul du temps de réponse
    long fin = System.currentTimeMillis();
    System.out.println("duree␣=␣" + (fin - debut) + "ms");
}
```

- Faites varier les deux derniers paramètres de la `DataSource` et étudiez l'impact sur le temps de réponse.

4 Spring et JDBC

Préalable. Jusqu'à maintenant nous avons utilisé Spring pour définir nos services et nos injections (de la DataSource dans la Dao et de la DAO dans les classses de test). Notre code JDBC n'utilise pas les fonctions de Spring.

4.1 Faciliter l'utilisation de JDBC

Nous allons maintenant utiliser les outils de Spring pour simplifier l'utilisation de JDBC.

- Dupliquez la classe `NameDao` en `SpringNameDao`.
- Ajoutez à votre classe `SpringNameDao` une injection automatique d'une instance de `JdbcTemplate` (boite à outils JDBC de Spring) :

```java
package myapp.jdbc;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Collection;
import java.util.List;

import javax.annotation.PostConstruct;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
@Transactional
public class SpringNameDao {

    @Autowired
    JdbcTemplate jt;

    ...

}
```

- Vous pouvez ensuite dupliquer votre classe de test (afin de tester la version Spring) et, bien entendu, lui injecter la Dao version Spring.

Travail à faire :

- Avec l'aide de `JdbcTemplate` et des méthodes proposées (consultez la Javadoc[4]), reprogrammez les méthodes de la classe `SpringNameDao`. Vérifiez à chaque fois que le test unitaire est positif.
- Grace à l'annotation `@Transactional`, l'exécution d'une méthode est associée à une transaction (et donc une connexion). Pour bien comprendre ce principe, ajoutez à votre Dao la méthode de création double ci-dessous. Cette méthode va forcément échouer (à cause de la clé primaire). Vérifiez dans un test unitaire que la première insertion n'est pas conservée (un *rollback* est effectué).

```java
    public void addNameTwoTimes(int id, String name) {
        ...
    }
```

4.2 Prévoir un script d'initialisation

Nous pouvons améliorer la phase de création du schéma relationnel en délocalisation le script dans un fichier `schema.sql` placé dans la package `myapp.jdbc` :

```java
@PostConstruct
public void initSchema() throws SQLException {
    try (var c = jt.getDataSource().getConnection()) {
        var res = new ClassPathResource("schema.sql", SpringNameDao.class);
        ScriptUtils.executeSqlScript(c, res);
    }
}
```

4. https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html

4.3 Utiliser un Javabean

Commencez par préparer le Javabean de représentation des noms :

```java
package myapp.jdbc;

import java.io.Serializable;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Name implements Serializable {

    private static final long serialVersionUID = 1L;

    private int id;
    private String name;

}
```

Travail à faire :

- Utilisez ce Javabean pour revoir les signatures des méthodes de la classe `SpringNameDao` :

```java
public List<Name> findNames() {
    ...
}
```

  Vous aurez besoin d'utiliser un **mapper** c'est-à-dire une méthode qui va construire une instance de la classe `Name` à partir d'une ligne de la base de données représentée par un `ResultSet` :

```java
private static Name nameMapper(ResultSet rs, int i) throws SQLException {
    var n = new Name();
    n.setId(rs.getInt("id"));
    n.setName(rs.getString("name"));
    return n;
}
```

- Faites de même pour la méthode d'ajout que devient `void addName(Name n)` ainsi que pour la méthode de lecture.
- Vous pouvez ajouter la méthode `int countNames(String pattern)` basée sur la méthode `queryForObject` de `JdbcTemplate`.
- Vous pouvez également remplacer `JdbcTemplate` par `NamedParameterJdbcTemplate` une version orientée vers l'utilisation des propriétés des Javabeans qui codent les données (plus d'information dans cette documentation [5] section 3.2).
- Testez l'utilisation des `RowSet` en vous basant sur le cours [6].

5. https://www.baeldung.com/spring-jdbc-jdbctemplate
6. jdbc.html#rowset