Spring

semifir

Spring Data JDBC

Spring Data JDBC

Spring Data JDBC est un autre module de Spring Data qui permet d'accéder à une base de données relationnelle.

Elle est basée sur JDBC et permet de faire des requêtes SQL.

Dependency

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

Configuration

La configuration de Spring Data JDBC se fait dans le fichier `application.yml`.

```yaml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/spring_data
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver
```

Différence avec Spring Data JPA

Spring Data JDBC ne définie pas:

- d'interface pour les repositories (JpaCrudRepository).
- d'Entité (@Entity).
- de génération de clé primaire (@GeneratedValue).

A la place, il faut définir les requêtes SQL dans les repositories et définir comment sérialiser les données.

JDBCTemplate

Pour accéder à la base de données, il faut utiliser la classe `JdbcTemplate`.

```
@Autowired
private JdbcTemplate jdbcTemplate;
```

Elle est disponible dans le context de Spring si le module `spring-boot-starter-jdbc` est présent.

Utilisation de JDBCTemplate

Il y a deux méthodes principales:

- `update` pour executer une requête SQL qui modifie la base de données `INSERT`, `UPDATE`, `DELETE`, ...
- `query` pour executer une requête SQL de type `SELECT`.

La méthode update

Pour executer une requête SQL qui modifie la base de données, il faut utiliser la méthode `update`.

```
jdbcTemplate.update(
    "INSERT INTO user (name, email, date) VALUES (?, ?, ?)",
    user.getName(),
    user.getEmail(),
    user.getDate()
);
```

Les paramètres de la requête

Pour ne pas avoir de problème de sécurité, il est préférable d'utiliser des paramètres pour les requêtes SQL.

```
jdbcTemplate.update(
    "INSERT INTO user (name, email, date) VALUES (?, ?, ?)",
    user.getName(),
    user.getEmail(),
    user.getDate()
);
```

et non

```
jdbcTemplate.update(
    "INSERT INTO user (name, email, date) VALUES ('"
    + user.getName() + "', '"
    + user.getEmail() + "', '"
    + user.getDate() + "')"
);
```

Query

Pour executer une requête SQL, il faut utiliser la méthode `query`.

Contrairement à JPA, JDBC ne désérialise pas les données automatiquement.

Il faut donc définir un `RowMapper` qui permet de mapper les données.

```
jdbcTemplate.query("SELECT * FROM user", new UserRowMapper());
```

Utilisation de JDBCTemplate

La méthode `query` prend en paramètre:

- la requête SQL.
- Un RowMapper qui permet de mapper les résultats de la requête.
- [Optionnel] Les paramètres de la requête.

```
jdbcTemplate.query(
    "SELECT * FROM user WHERE name = ?",
    new UserRowMapper(),
    "John"
);
```

RowMapper

Le RowMapper est une interface fonctionnelle qui prend en paramètre:

- le résultat de la requête de type `ResultSet`.
- le numéro de la ligne.

```
@FunctionalInterface
public interface RowMapper<T> {
    T mapRow(ResultSet rs, int rowNum) throws SQLException;
}
```

ResultSet

Le ResultSet est un objet qui contient les résultats de la requête.

Il est semblable à une Map ou Dictionary.

Il permet de récupérer les données par nom de colonne ou par numéro de colonne.

```
rs.getString("name"); // retourne la valeur de la colonne name au format String
rs.getString(1); // retourne la valeur de la colonne 1 au format String
rs.getInt("age"); // retourne la valeur de la colonne age au format int
```

Exercice 5: Spring Data JDBC

Exercice 5

Profiles

Dans certains cas, il est nécessaire de définir des configurations spécifiques pour un environnement donné.

C'est dans ces moments que les profiles sont utiles.

Configuration

Pour définir un profile, il faut ajouter un fichier `application-{profile}.yml`.

```
spring:
    profiles:
    active: "dev"
```

```
spring:
  profiles: dev
  datasource:
    url: jdbc:mysql://localhost:3306/spring_data
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver
```

```yaml
spring:
  profiles: prod
  datasource:
    url: jdbc:mysql://prod:3306/spring_data
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver
```

Utilisation

Pour activer un profile, il faut ajouter l'option `--spring.profiles.active={profile}`.

```
java -jar target/demo-0.0.1-SNAPSHOT.jar --spring.profiles.active=prod
```

Connexion multiple

- Dans des cas plus complexes, il peut être necessaire de définir plusieurs pool de connexion
- Pour cela nous allons avoir besoin de définir plusieurs configurations dans le fichier `application.yml`

```yaml
spring:
  profiles: dev
  user-datasource:
    url: jdbc:mysql://localhost:3306/user
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver

  client-datasource:
    url: jdbc:mysql://localhost:3307/client
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver
```

Utilisation

- Il faut dans un premier temps définir leurs classes de configuration

```java
@Configuration
public class UserDataSourceConfig {
    @Bean
    @ConfigurationProperties("spring.user-datasource")
    public DataSource userDataSource() {
```

```
        return DataSourceBuilder.create().build();
    }
}
```

```
@Configuration
public class ClientDataSourceConfig {
    @Bean
    @ConfigurationProperties("spring.client-datasource")
    public DataSource clientDataSource() {
        return DataSourceBuilder.create().build();
    }
}
```

Utilisation

- Ensuite il faut injecter les beans dans les classes qui en ont besoin

```
@Bean
public DataSource userDataSource() {
    return userDataSourceProperties()
        .initializeDataSourceBuilder()
        .build();
}
```

```
@Bean
public DataSource clientDataSource() {
    return clientDataSourceProperties()
        .initializeDataSourceBuilder()
        .build();
}
```

Utilisation JDBC

- Dans le cas de JDBC nous devons configurer le `JdbcTemplate` dans les classes qui en ont besoin

```
public JdbcTemplate userJdbcTemplate(
    @Qualifier("userDataSource") DataSource userDataSource) {
    return new JdbcTemplate(userDataSource);
}
```

- Ensuite nous pouvons injecter le `JdbcTemplate` dans les classes qui en ont besoin

```
@Autowired
@Qualifier("userJdbcTemplate")
private JdbcTemplate userJdbcTemplate;
```