### Performing CRUD Operations with `JdbcTemplate`

`JdbcTemplate` in Spring provides a convenient and flexible way to interact with relational databases using SQL. Here’s how you can use `JdbcTemplate` to perform basic CRUD operations (Create, Read, Update, and Delete) in a Spring Boot application.

---

### 1. \*\*Setup: Create a Table and Model Class\*\*

For this example, let's assume we are working with a simple `users` table in a database, with the following structure:

```sql

CREATE TABLE users (

id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(100),

email VARCHAR(100)

);

```

We'll also define a `User` model class to map the table’s data to a Java object:

```java

public class User {

private int id;

private String name;

private String email;

public User() {}

public User(int id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

// Getters and Setters

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

}

```

---

### 2. \*\*Setting up `JdbcTemplate` in Spring Boot\*\*

To use `JdbcTemplate`, you need to add the required dependencies in `pom.xml`:

```xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

```

Next, configure the database connection in `application.properties` or `application.yml`:

```properties

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

```

---

### 3. \*\*Create a `UserRepository` with `JdbcTemplate`\*\*

Now, let's implement the `UserRepository` using `JdbcTemplate`. This repository will handle CRUD operations.

#### \*\*Repository Class:\*\*

```java

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.RowMapper;

import org.springframework.stereotype.Repository;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.List;

@Repository

public class UserRepository {

private final JdbcTemplate jdbcTemplate;

public UserRepository(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

// RowMapper to convert ResultSet to User object

private RowMapper<User> userRowMapper = (rs, rowNum) -> new User(

rs.getInt("id"),

rs.getString("name"),

rs.getString("email")

);

// CREATE: Insert a new user

public int save(User user) {

String sql = "INSERT INTO users (name, email) VALUES (?, ?)";

return jdbcTemplate.update(sql, user.getName(), user.getEmail());

}

// READ: Get all users

public List<User> findAll() {

String sql = "SELECT \* FROM users";

return jdbcTemplate.query(sql, userRowMapper);

}

// READ: Find a user by ID

public User findById(int id) {

String sql = "SELECT \* FROM users WHERE id = ?";

return jdbcTemplate.queryForObject(sql, userRowMapper, id);

}

// UPDATE: Update a user

public int update(User user) {

String sql = "UPDATE users SET name = ?, email = ? WHERE id = ?";

return jdbcTemplate.update(sql, user.getName(), user.getEmail(), user.getId());

}

// DELETE: Delete a user by ID

public int deleteById(int id) {

String sql = "DELETE FROM users WHERE id = ?";

return jdbcTemplate.update(sql, id);

}

}

```

### Explanation:

1. \*\*`save(User user)`\*\*: Inserts a new record into the `users` table using the `update()` method.

2. \*\*`findAll()`\*\*: Retrieves all users from the database using the `query()` method and maps each row to a `User` object using a `RowMapper`.

3. \*\*`findById(int id)`\*\*: Fetches a specific user by their ID using `queryForObject()`.

4. \*\*`update(User user)`\*\*: Updates an existing user's information in the `users` table.

5. \*\*`deleteById(int id)`\*\*: Deletes a user by their ID from the `users` table.

---

### 4. \*\*Performing CRUD Operations\*\*

You can now perform the CRUD operations via a service or directly in a controller.

#### \*\*Example Service Class:\*\*

```java

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class UserService {

private final UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

public int createUser(User user) {

return userRepository.save(user);

}

public List<User> getAllUsers() {

return userRepository.findAll();

}

public User getUserById(int id) {

return userRepository.findById(id);

}

public int updateUser(User user) {

return userRepository.update(user);

}

public int deleteUserById(int id) {

return userRepository.deleteById(id);

}

}

```

---

### 5. \*\*Example REST Controller\*\*

If you're exposing the CRUD operations via REST APIs using Spring Boot, you can create a REST controller that interacts with the service.

```java

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/api/users")

public class UserController {

private final UserService userService;

public UserController(UserService userService) {

this.userService = userService;

}

// CREATE: Add a new user

@PostMapping

public ResponseEntity<String> createUser(@RequestBody User user) {

int result = userService.createUser(user);

return ResponseEntity.ok("User created with ID: " + result);

}

// READ: Get all users

@GetMapping

public List<User> getAllUsers() {

return userService.getAllUsers();

}

// READ: Get user by ID

@GetMapping("/{id}")

public User getUserById(@PathVariable int id) {

return userService.getUserById(id);

}

// UPDATE: Update a user

@PutMapping("/{id}")

public ResponseEntity<String> updateUser(@PathVariable int id, @RequestBody User user) {

user.setId(id);

userService.updateUser(user);

return ResponseEntity.ok("User updated successfully");

}

// DELETE: Delete a user

@DeleteMapping("/{id}")

public ResponseEntity<String> deleteUser(@PathVariable int id) {

userService.deleteUserById(id);

return ResponseEntity.ok("User deleted successfully");

}

}

```

---

### 6. \*\*Testing CRUD Operations\*\*

You can now test the CRUD operations by sending HTTP requests to the REST endpoints. Here's an overview of how the URLs map to operations:

- \*\*Create User\*\*: `POST /api/users`

- Body: `{"name": "John", "email": "john@example.com"}`

- \*\*Get All Users\*\*: `GET /api/users`

- \*\*Get User by ID\*\*: `GET /api/users/{id}`

- \*\*Update User\*\*: `PUT /api/users/{id}`

- Body: `{"name": "John Doe", "email": "john.doe@example.com"}`

- \*\*Delete User\*\*: `DELETE /api/users/{id}`

---

### Conclusion

Using `JdbcTemplate` for performing CRUD operations in Spring Boot simplifies the code required for interacting with a relational database. With its powerful features like automatic resource management, exception handling, and the ability to map rows of a `ResultSet` to Java objects, `JdbcTemplate` provides a straightforward and efficient approach for database interaction.