Accessing Relational Data using JDBC with Spring

This guide walks you through the process of accessing relational data with Spring.

What you'll build

You'll build an application using Spring's `JdbcTemplate` to access data stored in a relational database.

What you'll need

- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 or later
- Gradle 2.3+ or Maven 3.0+
- You can also import the code from this guide as well as view the web page directly into Spring Tool Suite (STS) and work your way through it from there.

Table of contents

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step, or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Build with Gradle.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git:
  ```
  git clone https://github.com/spring-guides/gs-relational-data-access.git
  ```
- cd into `gs-relational-data-access/initial`
- Jump ahead to Create a Customer object.

When you're finished, you can check your results against the code in `gs-relational-data-access/complete`.

Build with Gradle

Build with Maven

Tags

spring-data jdbc

Projects

Spring Framework

Build with your IDE

Create a Customer object

The simple data access logic you will work with below below manages first and last names of customers. To represent this data at the application level, create a `Customer` class.

`src/main/java/hello/Customer.java`

```
package hello;

public class Customer {
    private long id;
    private String firstName, lastName;

    public Customer(long id, String firstName, String lastName)
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return String.format(
                "Customer[id=%d, firstName='%s', lastName='%s']"
```

Concepts and technologies

Git

```
                id, firstName, lastName);
    }

    // getters & setters omitted for brevity
}
```

Store and retrieve data

Spring provides a template class called `JdbcTemplate` that makes it easy to work with SQL relational databases and JDBC. Most JDBC code is mired in resource acquisition, connection management, exception handling, and general error checking that is wholly unrelated to what the code is meant to achieve. The `JdbcTemplate` takes care of all of that for you. All you have to do is focus on the task at hand.

`src/main/java/hello/Application.java`

```
package hello;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplicat
import org.springframework.jdbc.core.JdbcTemplate;
```

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

@SpringBootApplication
public class Application implements CommandLineRunner {

    private static final Logger log = LoggerFactory.getLogger(Ap

    public static void main(String args[]) {
        SpringApplication.run(Application.class, args);
    }

    @Autowired
    JdbcTemplate jdbcTemplate;

    @Override
    public void run(String... strings) throws Exception {

        log.info("Creating tables");

        jdbcTemplate.execute("DROP TABLE customers IF EXISTS");
        jdbcTemplate.execute("CREATE TABLE customers(" +
                "id SERIAL, first_name VARCHAR(255), last_name V

        // Split up the array of whole names into an array of fi
        List<Object[]> splitUpNames = Arrays.asList("John Woo",
                .map(name -> name.split(" "))
```

```
                .collect(Collectors.toList());

        // Use a Java 8 stream to print out each tuple of the li
        splitUpNames.forEach(name -> log.info(String.format("Ins

        // Uses JdbcTemplate's batchUpdate operation to bulk loa
        jdbcTemplate.batchUpdate("INSERT INTO customers(first_na

        log.info("Querying for customer records where first_name
        jdbcTemplate.query(
                "SELECT id, first_name, last_name FROM customers
                (rs, rowNum) -> new Customer(rs.getLong("id"), r
        ).forEach(customer -> log.info(customer.toString()));
    }
}
```

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration` tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration` tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
- `@ComponentScan` tells Spring to look for other components,

configurations, and services in the the `hello` package. In this case, there aren't any.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there wasn't a single line of XML? No **web.xml** file either. This web application is 100% pure Java and you didn't have to deal with configuring any plumbing or infrastructure.

Spring Boot spots **H2**, an in-memory relational database engine, and automatically creates a connection. Because we are using **spring-jdbc**, Spring Boot automatically creates a `JdbcTemplate`. The `@Autowired JdbcTemplate` field automatically loads it and makes it available.

This `Application` class implements Spring Boot's `CommandLineRunner`, which means it will execute the `run()` method after the application context is loaded up.

First, you install some DDL using `JdbcTemplate's `execute` method.

Second, you take a list of strings and using Java 8 streams, split them into firstname/lastname pairs in a Java array.

Then you install some records in your newly created table using `JdbcTemplate's `batchUpdate` method. The first argument to the method call is the query string, the last argument (the array of `Object` s) holds the variables to be substituted into the query where the "?" characters are.

> **Note:** For single insert statements, `JdbcTemplate's `insert` method is good. But for multiple inserts, it's better to use `batchUpdate`.

> **Note:** Use `?` for arguments to avoid SQL injection attacks by instructing JDBC to bind variables.

Finally you use the `query` method to search your table for records matching the criteria. You again use the "?" arguments to create parameters for the query, passing in the actual values when you make the call. The last argument is a Java 8 lambda used to convert each result row into a new `Customer` object.

> **Note:** Java 8 lambdas map nicely onto single method interfaces, like Spring's `RowMapper`. If you are using Java 7 or earlier, you can easily plug in an anonymous interface implementation and have the same method body as the lambda expresion's body contains, and it will work with no fuss from Spring.

Build an executable JAR

If you are using Gradle, you can run the application using `./gradlew bootRun`.

You can build a single executable JAR file that contains all the necessary dependencies, classes, and resources. This makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

```
./gradlew build
```

Then you can run the JAR file:

```
java -jar build/libs/gs-relational-data-access-0.1.0.jar
```

If you are using Maven, you can run the application using

`mvn spring-boot:run`. Or you can build the JAR file with `mvn clean package` and run the JAR by typing:

```
java -jar target/gs-relational-data-access-0.1.0.jar
```

> **Note:** The procedure above will create a runnable JAR. You can also opt to build a classic WAR file instead.

You should see the following output:

```
2015-06-19 10:58:31.152  INFO 67731 --- [           main] hello.
2015-06-19 10:58:31.219  INFO 67731 --- [           main] hello.
2015-06-19 10:58:31.220  INFO 67731 --- [           main] hello.
2015-06-19 10:58:31.220  INFO 67731 --- [           main] hello.
2015-06-19 10:58:31.220  INFO 67731 --- [           main] hello.
2015-06-19 10:58:31.230  INFO 67731 --- [           main] hello.
2015-06-19 10:58:31.242  INFO 67731 --- [           main] hello.
2015-06-19 10:58:31.242  INFO 67731 --- [           main] hello.
2015-06-19 10:58:31.244  INFO 67731 --- [           main] hello.
```

Summary

Congratulations! You've just used Spring to develop a simple JDBC client.

TEAM SERVICES TOOLS

SUBSCRIBE TO OUR NEWSLETTER

Email Address