**Accessing Relational Data using JDBC with Spring**

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

What You Will build

You will build an application that uses Spring’s JdbcTemplate to access data stored in a relational database.

What You Need

* About 15 minutes
* A favorite text editor or IDE
* [Java 1.8](https://www.oracle.com/java/technologies/downloads/) or later
* [Gradle 7.5+](https://gradle.org/install/) or [Maven 3.5+](https://maven.apache.org/download.cgi)
* You can also import the code straight into your IDE:
  + [Spring Tool Suite (STS)](https://spring.io/guides/gs/sts)
  + [IntelliJ IDEA](https://spring.io/guides/gs/intellij-idea/)
  + [VSCode](https://spring.io/guides/gs/guides-with-vscode/)

How to complete this guide

Clone / Fork the rep and follow the instructions.

Starting with Spring Initializr (Skip this step)

You can use this [pre-initialized project](https://start.spring.io/#!type=maven-project&language=java&platformVersion=2.5.5&packaging=jar&jvmVersion=11&groupId=com.example&artifactId=relational-data-access&name=relational-data-access&description=Demo%20project%20for%20Spring%20Boot&packageName=com.example.relational-data-access&dependencies=jdbc,h2) and click Generate to download a ZIP file. This project is configured to fit the examples in this tutorial.

To manually initialize the project:

1. Navigate to [https://start.spring.io](https://start.spring.io/). This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or Maven and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **JDBC API** and **H2 Database**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

|  |  |
| --- | --- |
|  | If your IDE has the Spring Initializr integration, you can complete this process from your IDE. |
|  | You can also fork the project from Github and open it in your IDE or other editor. |

Create a Customer Object

The simple data access logic you will work with manages the first and last names of customers. To represent this data at the application level, create a Customer class, as the following listing (from src/main/java/com/example/relationaldataaccess/Customer.java) shows:

package com.example.relationaldataaccess;

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']",

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. The following listing (from src/main/java/com/example/relationaldataaccess/RelationalDataAccessApplication.java) shows a class that can store and retrieve data over JDBC:

package com.example.relationaldataaccess;

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.SpringBootApplication;

import org.springframework.jdbc.core.JdbcTemplate;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

@SpringBootApplication

public class RelationalDataAccessApplication implements CommandLineRunner {

private static final Logger log = LoggerFactory.getLogger(RelationalDataAccessApplication.class);

public static void main(String args[]) {

SpringApplication.run(RelationalDataAccessApplication.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 VARCHAR(255))");

// Split up the array of whole names into an array of first/last names

List<Object[]> splitUpNames = Arrays.asList("John Woo", "Jeff Dean", "Josh Bloch", "Josh Long").stream()

.map(name -> name.split(" "))

.collect(Collectors.toList());

// Use a Java 8 stream to print out each tuple of the list

splitUpNames.forEach(name -> log.info(String.format("Inserting customer record for %s %s", name[0], name[1])));

// Uses JdbcTemplate's batchUpdate operation to bulk load data

jdbcTemplate.batchUpdate("INSERT INTO customers(first\_name, last\_name) VALUES (?,?)", splitUpNames);

log.info("Querying for customer records where first\_name = 'Josh':");

jdbcTemplate.query(

"SELECT id, first\_name, last\_name FROM customers WHERE first\_name = ?", new Object[] { "Josh" },

(rs, rowNum) -> new Customer(rs.getLong("id"), rs.getString("first\_name"), rs.getString("last\_name"))

).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 com.example.relationaldataaccess package. In this case, there are none.

The main() method uses Spring Boot’s SpringApplication.run() method to launch an application.

Spring Boot supports H2 (an in-memory relational database engine) and automatically creates a connection. Because we use 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.

First, install some DDL by using the execute method of JdbcTemplate.

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

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

|  |  |  |
| --- | --- | --- |
|  | For single insert statements, the insert method of JdbcTemplate is good. However, for multiple inserts, it is better to use batchUpdate. | |
|  | | Use ? for arguments to avoid [SQL injection attacks](https://en.wikipedia.org/wiki/SQL_injection) by instructing JDBC to bind variables. |

Finally, use the query method to search your table for records that match 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 that is used to convert each result row into a new Customer object.

|  |  |
| --- | --- |
|  | Java 8 lambdas map nicely onto single method interfaces, such as Spring’s RowMapper. If you use Java 7 or earlier, you can plug in an anonymous interface implementation and have the method body be the same as the lambda expression’s body. |

Build an executable JAR

You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Gradle, you can run the application by using ./gradlew bootRun. Alternatively, you can build the JAR file by using ./gradlew build and then run the JAR file, as follows:

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

If you use Maven, you can run the application by using ./mvnw spring-boot:run. Alternatively, you can build the JAR file with ./mvnw clean package and then run the JAR file, as follows:

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

|  |  |
| --- | --- |
|  | The steps described here create a runnable JAR. You can also [build a classic WAR file](https://spring.io/guides/gs/convert-jar-to-war/). |

You should see the following output:

2019-09-26 13:46:58.561 INFO 47569 --- [ main] c.e.r.RelationalDataAccessApplication : Creating tables

2019-09-26 13:46:58.564 INFO 47569 --- [ main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...

2019-09-26 13:46:58.708 INFO 47569 --- [ main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.

2019-09-26 13:46:58.809 INFO 47569 --- [ main] c.e.r.RelationalDataAccessApplication : Inserting customer record for John Woo

2019-09-26 13:46:58.810 INFO 47569 --- [ main] c.e.r.RelationalDataAccessApplication : Inserting customer record for Jeff Dean

2019-09-26 13:46:58.810 INFO 47569 --- [ main] c.e.r.RelationalDataAccessApplication : Inserting customer record for Josh Bloch

2019-09-26 13:46:58.810 INFO 47569 --- [ main] c.e.r.RelationalDataAccessApplication : Inserting customer record for Josh Long

2019-09-26 13:46:58.825 INFO 47569 --- [ main] c.e.r.RelationalDataAccessApplication : Querying for customer records where first\_name = 'Josh':

2019-09-26 13:46:58.835 INFO 47569 --- [ main] c.e.r.RelationalDataAccessApplication : Customer[id=3, firstName='Josh', lastName='Bloch']

2019-09-26 13:46:58.835 INFO 47569 --- [ main] c.e.r.RelationalDataAccessApplication : Customer[id=4, firstName='Josh', lastName='Long']

Summary

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

|  |  |
| --- | --- |
|  | Spring Boot has many features for configuring and customizing the connection pool — for instance, to connect to an external database instead of an in-memory one. See the [Reference Guide](http://docs.spring.io/spring-boot/docs/2.2.1.RELEASE/reference/htmlsingle/#boot-features-configure-datasource) for more detail. |