Accessing data with MySQL

This guide walks you through the process of creating a Spring application connected to a MySQL Database (as opposed to an in-memory, embedded database, which most of the other guides and many sample applications use). It uses Spring Data JPA to access the database, but this is only one of many possible choices (for example, you could use plain Spring JDBC).

Get the Code

Go To Repo

What You Will Build

You will create a MySQL database, build a Spring application, and connect it to the newly created database.

MySQL is licensed with the GPL, so any program binary that you distribute with it must use the GPL, too. See the GNU General Public Licence.

What You Need

- About 15 minutes
- A favorite text editor or IDE
- Java 17 or later

How to Complete This Guide

Like most Spring Getting Started guides you can start from scratch and complete each step, or you can jump straight to the solution, by viewing the code in this repository.

To **see the end result in your local environment**, you can do one of the following:

- Download and unzip the source repository for this guide
- Clone the repository using Git: `git clone https://github.com/spring-guides/gs-accessing-data-mysql.git`
- Fork the repository which lets you request changes to this guide through submission of a pull request

Setting up the MySQL Database

Before you can build your application, you first need to configure a MySQL database. This guide assumes that you use Spring Boot Docker Compose support. A prerequisite of this approach is that your development machine has a Docker environment, such as Docker Desktop, available. Add a dependency `spring-boot-docker-compose` that does the following:

- Search for a `compose.yml` and other common compose filenames in your working directory
- Call `docker compose up` with the discovered `compose.yml`
- Create service connection beans for each supported container
- Call `docker compose stop` when the application is shutdown

To use Docker Compose support, you need only follow this guide. Based on the dependencies you pull in, Spring Boot finds the correct `compose.yml` file and start your Docker container when you run your application.

Starting with Spring Initializr

You can use this pre-initialized project 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. 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 **Spring Web**, **Spring Data JPA**, **MySQL Driver**, **Docker Compose Support**, and **Testcontainers**.
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.

Create the `@Entity` Model

You need to create the entity model, as the following listing (in `src/main/java/com/example/accessingdatamysql/User.java`) shows:

```java
package com.example.accessingdatamysql;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity // This tells Hibernate to make a table out of this class
public class User {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Integer id;

  private String name;

  private String email;

  public Integer getId() {
    return id;
  }

  public void setId(Integer 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;
  }
}
```

Hibernate automatically translates the entity into a table.

Create the Repository

You need to create the repository that holds user records, as the following listing (in `src/main/java/com/example/accessingdatamysql/UserRepository.java`) shows:

```java
package com.example.accessingdatamysql;

import org.springframework.data.repository.CrudRepository;

import com.example.accessingdatamysql.User;

// This will be AUTO IMPLEMENTED by Spring into a Bean called userRepository
// CRUD refers Create, Read, Update, Delete

public interface UserRepository extends CrudRepository<User, Integer> {

}
```

Spring automatically implements this repository interface in a bean that has the same name (with a change in the case — it is called `userRepository`).

Create a Controller

You need to create a controller to handle HTTP requests to your application, as the following listing (in `src/main/java/com/example/accessingdatamysql/MainController.java`) shows:

```java
package com.example.accessingdatamysql;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller // This means that this class is a Controller
@RequestMapping(path="/demo") // This means URL's start with /demo (after Application path)
public class MainController {
  @Autowired // This means to get the bean called userRepository
         // Which is auto-generated by Spring, we will use it to handle the data
  private UserRepository userRepository;

  @PostMapping(path="/add") // Map ONLY POST Requests
  public @ResponseBody String addNewUser (@RequestParam String name
      , @RequestParam String email) {
    // @ResponseBody means the returned String is the response, not a view name
    // @RequestParam means it is a parameter from the GET or POST request

    User n = new User();
    n.setName(name);
    n.setEmail(email);
    userRepository.save(n);
    return "Saved";
  }

  @GetMapping(path="/all")
  public @ResponseBody Iterable<User> getAllUsers() {
    // This returns a JSON or XML with the users
    return userRepository.findAll();
  }
}
```

The preceding example explicitly specifies `POST` and `GET` for the two endpoints. By default, `@RequestMapping` maps all HTTP operations.

Create an Application Class

Spring Initializr creates a simple class for the application. The following listing shows the class that Initializr created for this example (in `src/main/java/com/example/accessingdatamysql/AccessingDataMysqlApplication.java`):

```java
package com.example.accessingdatamysql;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AccessingDataMysqlApplication {

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

}
```

For this example, you need not modify the `AccessingDataMysqlApplication` class.

Spring Initializr adds the `@SpringBootApplication` annotation to our main class. `@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`: Spring Boot attempts to automatically configure your Spring application based on the

dependencies that you have added.

- `@ComponentScan`: Tells Spring to look for other components, configurations, and services. If specific packages are not defined, recursive scanning begins with the package of the class that declares the annotation.

Run the Application

At this point, you can now run the application to see your code in action. You can run the main method through your IDE or from the command line. Note that, if you have cloned the project from the solution repository, your IDE may look in the wrong place for the `compose.yaml` file. You can configure your IDE to look in the correct place or you could use the command line to run the application. The `./gradlew bootRun` and `./mvnw spring-boot:run` commands launch the application and automatically find the compose.yaml file.

Test the Application

Now that the application is running, you can test it by using `curl` or some similar tool. You have two HTTP endpoints that you can test:

`GET localhost:8080/demo/all`: Gets all data. `POST localhost:8080/demo/add`: Adds one user to the data.

The following curl command adds a user:

```
$ curl http://localhost:8080/demo/add -d name=First -d email=someemail@someemailprovider.com
```

The reply should be as follows:

```
Saved
```

The following command shows all the users:

```
$ curl http://localhost:8080/demo/all
```

The reply should be as follows:

```
[{"id":1,"name":"First","email":"someemail@someemailprovider.com"}]
```

Preparing to Build the Application

To package and run the application, we need to provide an external MySQL database rather than using Spring Boot Docker Compose Support. For this task, we can reuse the provided `compose.yaml` file with a few modifications: First, modify the `ports` entry in `compose.yaml` to be `3306:3306`. Second, add a `container_name` of `guide-mysql`.

After these steps, the `compose.yaml` file should be:

```
services:
  mysql:
    container_name: 'guide-mysql'
    image: 'mysql:latest'
    environment:
      - 'MYSQL_DATABASE=mydatabase'
      - 'MYSQL_PASSWORD=secret'
      - 'MYSQL_ROOT_PASSWORD=verysecret'
      - 'MYSQL_USER=myuser'
    ports:
      - '3306:3306'
```

You can now run `docker compose up` to start this MySQL container.

Third, we need to tell our application how to connect to the database. This step was previously handled automatically with Spring Boot Docker Compose support. To do so, modify the `application.properties` file so that it is now:

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase
spring.datasource.username=myuser
spring.datasource.password=secret
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.show-sql: true
```

Building the Application

This section describes different ways to run this guide:

1. Building and executing a JAR file

2. Building and executing a Docker container using Cloud Native Buildpacks
3. Building and executing a native image
4. Building and executing a native image container using Cloud Native Buildpacks

Regardless of how you choose to run the application, the output should be the same.

To run the application, you can package the application as an executable jar. The `./gradlew clean build` command compiles the application to an executable jar. You can then run the jar with the `java -jar build/libs/accessing-data-mysql-0.0.1-SNAPSHOT.jar` command.

Alternatively, if you have a Docker environment available, you could create a Docker image directly from your Maven or Gradle plugin, using buildpacks. With Cloud Native Buildpacks, you can create Docker compatible images that you can run anywhere. Spring Boot includes buildpack support directly for both Maven and Gradle. This means you can type a single command and quickly get a sensible image into a locally running Docker daemon. To create a Docker image using Cloud Native Buildpacks, run the `./gradlew bootBuildImage` command. With a Docker environment enabled, you can run the application with the `docker run --network container:guide-mysql docker.io/library/accessing-data-mysql:0.0.1-SNAPSHOT` command.

The `--network` flag tells Docker to attach our guide container to the existing network that our external container is using. You can find more information in the Docker documentation.

Native Image Support

Spring Boot also supports compilation to a native image, provided you have a GraalVM distribution on your machine. To create a native image with Gradle using Native Build Tools, first make sure that your Gradle build contains a `plugins` block that includes `org.graalvm.buildtools.native`.

```
plugins {
	id 'org.graalvm.buildtools.native' version '0.9.28'
...
```

You can then run the `./gradlew nativeCompile` command to generate a native image. When the build completes, you will be able to run the code with a near-instantaneous start up time by executing the `build/native/nativeCompile/accessing-data-mysql` command.

You can also create a Native Image using Buildpacks. You can generate a native image by running the `./gradlew bootBuildImage` command. Once the build completes, you can start your application with the `docker run --network container:guide-mysql docker.io/library/accessing-data-mysql:0.0.1-SNAPSHOT` command.

Test the Application in Docker

If you ran the application using a Docker instruction above, a simple curl command from a terminal or command line will no longer work. This is because we are running our containers in a Docker network that is not accessible from the terminal or command line. To run curl commands, we can start a third container to run our curl commands and attach it to the same network.

First, obtain an interactive shell to a new container that is running on the same network as the MySQL database and the application:

```
docker run --rm --network container:guide-mysql -it alpine
```

COPY

Next, from the shell inside of the container, install curl:

```
apk add curl
```

COPY

Finally, you can run the curl commands as described in Test the Application.

Make Some Security Changes

When you are on a production environment, you may be exposed to SQL injection attacks. A hacker may inject `DROP TABLE` or any other destructive SQL commands. So, as a security practice, you should make some changes to your database before you expose the application to your users.

The following command revokes all the privileges from the user associated with the Spring application:

```
mysql> revoke all on db_example.* from 'myuser'@'%';
```

COPY

Now the Spring application cannot do anything in the database.

The application must have some privileges, so use the following command to grant the minimum privileges the application needs:

```
mysql> grant select, insert, delete, update on db_example.* to 'myuser'@'%';
```

COPY

Removing all privileges and granting some privileges gives your Spring application the privileges necessary to make changes to only the data of the database and not the structure (schema).

When you want to make changes to the database:

1. Regrant permissions.
2. Change the `spring.jpa.hibernate.ddl-auto` to `update`.
3. Re-run your applications.

Then repeat the two commands shown here to make your application safe for production use again. Better still, use a dedicated migration tool, such as Flyway or Liquibase.

Summary

Congratulations! You have just developed a Spring application that is bound to a MySQL database and is ready for production!

See Also

The following guides may also be helpful:

- Accessing Data with JPA
- Accessing Data with MongoDB
- Accessing data with Gemfire

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.