< ALL GUIDES

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).

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

- MySQL version 5.6 or better. If you have Docker installed, it might be useful to run the database as a container.
- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 or later
- Gradle 4+ or Maven 3.2+
- You can also import the code straight into your IDE:
    - Spring Tool Suite (STS)
    - IntelliJ IDEA

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 Starting with Spring Initializr.

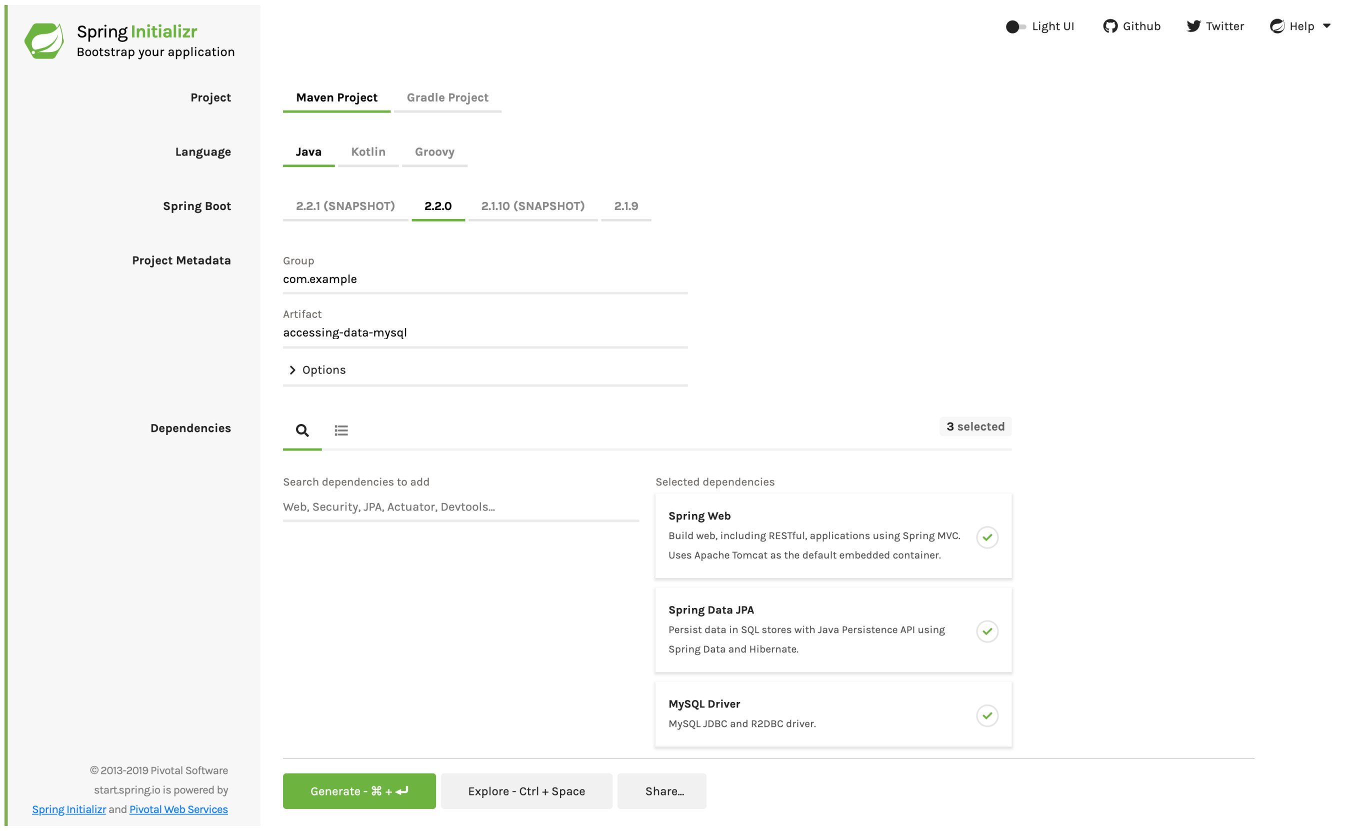
To **skip the basics**, do the following:

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

When you finish, you can check your results against the code in `gs-accessing-data-mysql/complete`.

Starting with Spring Initializr

For all Spring applications, you should start with the Spring Initializr. The Initializr offers a fast way to pull in all the dependencies you need for an application and does a lot of the set up for you. This example needs the Spring Web Starter, Spring Data JPA, and MySQL Driver dependencies. The following image shows the Initializr set up for this sample project:



Project: Maven Project | Gradle Project
Language: Java | Kotlin | Groovy
Spring Boot: 2.2.1 (SNAPSHOT) | 2.2.0 | 2.1.10 (SNAPSHOT) | 2.1.9
Project Metadata
Group: com.example
Artifact: accessing-data-mysql
Options
Dependencies: 3 selected
Search dependencies to add: Web, Security, JPA, Actuator, Devtools...
Selected dependencies
Spring Web: Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
Spring Data JPA: Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
MySQL Driver: MySQL JDBC and R2DBC driver.

© 2013-2019 Pivotal Software
start.spring.io is powered by Spring Initializr and Pivotal Web Services

The preceding image shows the Initializr with Maven chosen as the build tool. You can also use Gradle. It also shows values of `com.example` and `accessing-data-mysql` as the Group and Artifact, respectively. You will use those values throughout the rest of this sample.

The following listing shows the `pom.xml` file created when you choose Maven:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apach
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.2.2.RELEASE</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>accessing-data-mysql</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>accessing-data-mysql</name>
        <description>Demo project for Spring Boot</description>

        <properties>
                <java.version>1.8</java.version>
        </properties>
```

```
	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
			<exclusions>
				<exclusion>
					<groupId>org.junit.vintage</groupId>
					<artifactId>junit-vintage-engine</artifac
				</exclusion>
			</exclusions>
		</dependency>
	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```

The following listing shows the `build.gradle` file created when you choose Gradle:

```
plugins {
	id 'org.springframework.boot' version '2.2.2.RELEASE'
	id 'io.spring.dependency-management' version '1.0.8.RELEASE'
	id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
	mavenCentral()
}
```

```
dependencies {
	implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
	implementation 'org.springframework.boot:spring-boot-starter-web'
	runtimeOnly 'mysql:mysql-connector-java'
	testImplementation('org.springframework.boot:spring-boot-starter-test') {
		exclude group: 'org.junit.vintage', module: 'junit-vintage-engine
	}
}

test {
	useJUnitPlatform()
}
```

Create the Database

Open a terminal (command prompt in Microsoft Windows) and open a MySQL client as a user who can create new users.

For example, on a Linux system, use the following command;

COPY

```
$ sudo mysql --password
```

This connects to MySQL as `root` and allows access to the user from all hosts. This is **not the recommended way** for a production server.

To create a new database, run the following commands at the `mysql` prompt:

COPY

```
mysql> create database db_example; -- Creates the new database
mysql> create user 'springuser'@'%' identified by 'ThePassword'; -- Creates the user
mysql> grant all on db_example.* to 'springuser'@'%'; -- Gives all privileges to the new user on the newly created database
```

Create the `application.properties` File

Spring Boot gives you defaults on all things. For example, the default database is `H2`. Consequently, when you want to use any other database, you must define the connection attributes in the `application.properties` file.

Create a resource file called `src/main/resources/application.properties`, as the following listing shows:

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/db_example
spring.datasource.username=springuser
spring.datasource.password=ThePassword
```

Here, `spring.jpa.hibernate.ddl-auto` can be `none`, `update`, `create`, or `create-drop`. See the Hibernate documentation for details.

- `none`: The default for `MySQL`. No change is made to the database structure.
- `update`: Hibernate changes the database according to the given entity structures.
- `create`: Creates the database every time but does not drop it on close.
- `create-drop`: Creates the database and drops it when `SessionFactory` closes.

You must begin with either `create` or `update`, because you do not yet have the database structure. After the first run, you can switch it to `update` or `none`, according to program requirements. Use `update` when you want to make some change to the database structure.

The default for `H2` and other embedded databases is `create-drop`. For other databases, such as `MySQL`, the default is `none`.

It is a good security practice to, after your database is in a production state, set this to `none`, revoke all privileges from the MySQL user connected to the Spring application, and give the MySQL user only `SELECT`, `UPDATE`, `INSERT`, and `DELETE`. You can read more about this at the end of this guide.

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:

```
package com.example.accessingdatamysql;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.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:

COPY

```
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:

COPY

```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`):

```
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.

`@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. For example, if `spring-webmvc` is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a `DispatcherServlet`.
- `@ComponentScan`: Tells Spring to look for other components, configurations, and services in the `com/example` package, letting it find the controllers.

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

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-accessing-data-mysql-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-accessing-data-mysql-0.1.0.jar
```

The steps described here create a runnable JAR. You can also build a classic WAR file.

When you run the application, logging output is displayed. The service should be up and running within a few seconds.

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

COPY

The reply should be as follows:

```
Saved
```

The following command shows all the users:

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

The reply should be as follows:

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

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 'springuser'@'%';
```

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
'springuser'@'%';
```

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.