Accessing data with MySQL

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

What you'll build

You'll create a MySQL database, build a Spring application and connect it with the newly created database.

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

What you'll 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 2.3+ or Maven 3.0+
- 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 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-accessing-data-mysql.git`
- cd into `gs-accessing-data-mysql/initial`
- Jump ahead to Create the database.

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

Build with Gradle

Build with Maven

First you set up a basic build script. You can use any build system you like when building apps with Spring, but the code you need to work with Maven is included here. If you're not familiar with Maven, refer to Building Java Projects with Maven.

Create the directory structure

In a project directory of your choosing, create the following subdirectory structure; for example, with `mkdir -p src/main/java/hello` on *nix systems:

```
└── src
    └── main
        └── java
            └── hello
```

`pom.xml`

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>org.springframework</groupId>
    <artifactId>gs-mysql-data</artifactId>
    <version>0.1.0</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.2.RELEASE</version>
    </parent>

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

        <!-- JPA Data (We are going to use Repositories, Entities,
Hibernate, etc...) -->

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

```
    <!-- Use MySQL Connector-J -->

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

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

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

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
```

```
                <artifactId>spring-boot-maven-plugin</artifactId>

            </plugin>

        </plugins>

    </build>



</project>
```

The Spring Boot Maven plugin provides many convenient features:

- It collects all the jars on the classpath and builds a single, runnable "über-jar", which makes it more convenient to execute and transport your service.
- It searches for the `public static void main()` method to flag as a runnable class.
- It provides a built-in dependency resolver that sets the version number to match Spring Boot dependencies. You can override any version you wish, but it will default to Boot's chosen set of versions.

Build with your IDE

Create the database

Go to the terminal (command Prompt `cmd` in Microsoft Windows). Open MySQL client with a user that can create new users.

For example: On a Linux, use the command

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

This connects to MySQL as a root, this is **not the recommended way** for a production server.

Create a new database

```
mysql> create database db_example; -- Create the new database

mysql> create user 'springuser'@'localhost' identified by 'ThePassword';
-- Creates the user
```

```
mysql> grant all on db_example.* to 'springuser'@'localhost'; -- Gives
all the privileges to the new user on the newly created database
```

Create the `application.properties` file

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

In the sources folder, you create a resource file `src/main/resources/application.properties`

```
spring.jpa.hibernate.ddl-auto=create

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

spring.datasource.username=springuser

spring.datasource.password=ThePassword
```

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

- `none` This is the default for `MySQL`, no change to the database structure.
- `update` Hibernate changes the database according to the given Entity structures.
- `create` Creates the database every time, but don't drop it when close.
- `create-drop` Creates the database then drops it when the `SessionFactory` closes.

We here begin with `create` because we don't have the database structure yet. After the first run, we could 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`, but for others like `MySQL` is `none`

It is good security practice that after your database is in production state, you make this `none` and revoke all privileges from the MySQL user connected to the Spring application, then give him only SELECT, UPDATE, INSERT, DELETE.

This is coming in details in the end of this guide.

Create the `@Entity` model

`src/main/java/hello/User.java`

```
package hello;



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;
	}

}
```

This is the entity class which Hibernate will automatically translate into a table.

Create the repository

`src/main/java/hello/UserRepository.java`

```
package hello;



import org.springframework.data.repository.CrudRepository;



import hello.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, Long> {



}
```

This is the repository interface, this will be automatically implemented by Spring in a bean with the same name with changing case The bean name will be `userRepository`

Create a new controller for your Spring application

`src/main/java/hello/MainController.java`

```
package hello;



import org.springframework.beans.factory.annotation.Autowired;
```

```java
import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.ResponseBody;



import hello.User;

import hello.UserRepository;



@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;



        @GetMapping(path="/add") // Map ONLY GET 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 above example does not specify `GET` vs. `PUT`, `POST`, and so forth, because `@RequestMapping` maps all HTTP operations by default. Use `@RequestMapping(method=GET)` to narrow this mapping.

Make the application executable

Although it is possible to package this service as a traditional WAR file for deployment to an external application server, the simpler approach demonstrated below creates a standalone application. You package everything in a single, executable JAR file, driven by a good old Java `main()` method. Along the way, you use Spring's support for embedding the Tomcat servlet container as the HTTP runtime, instead of deploying to an external instance.

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

```
package hello;
```

```
import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;



@SpringBootApplication

public class Application {



    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);

    }

}
```

Build an executable JAR

You can run the application from the command line with Gradle or Maven. Or you can build a single executable JAR file that contains all the necessary dependencies, classes, and resources, and run that. This 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 are using Gradle, you can run the application using `./gradlew bootRun`. Or you can build the JAR file using `./gradlew build`. Then you can run the JAR file:

```
java -jar build/libs/gs-accessing-data-mysql-0.1.0.jar
```

If you are using Maven, you can run the application using `./mvnw spring-boot:run`. Or you can build the JAR file with `./mvnw clean package`. Then you can run the JAR file:

```
java -jar target/gs-accessing-data-mysql-0.1.0.jar
```

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

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.

Use `curl` for example. Now you have 2 REST Web Services you can test

`localhost:8080/demo/all` This gets all data `localhost:8080/demo/add` This adds one user to the data

```
$ curl
'localhost:8080/demo/add?name=First&email=someemail@someemailprovider.com
'
```

The reply should be

```
Saved
```

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

The reply should be

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

Make some security changes

Now when you are on 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, make those changes to your database before you expose the application to users.

```
mysql> revoke all on db_example.* from 'springuser'@'localhost';
```

This revokes ALL the priviliges from the user associated with the Spring application. Now the Spring application **cannot do** anything in the database. We don't want that, so

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

This gives your Spring application only the privileges necessary to make changes to **only** the data of the database and not the structure (schema).

Now make this change to your `src/main/resources/application.properties`

```
spring.jpa.hibernate.ddl-auto=none
```

This is instead of `create` which was on the first run for Hibernate to create the tables from your entities.

When you want to make changes on the database, regrant the permissions, change the `spring.jpa.hibernate.ddl-auto` to `update`, then re-run your applications, then repeat. Or, better, use a dedicated migration tool such as Flyway or Liquibase.

Summary

Congratulations! You've just developed a Spring application which is bound to a MySQL database, Ready for production!

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