**REST Services –**

**Spring Data JPA (Overview)**

Spring-boot provides support for Sprins Let’s create a new project *learn-jpa*. We will add below starter dependencies at the time of project creation:

\*\* This is brief a introduction to the Spring JDBC and Spring Data JPA. \*\*

* Spring boot Web – For REST APIs
* Spring boot JDBC
* Spring Boot JPA
* H2 – In-memory database for the project.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jdbc</artifactId>

</dependency>

<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>com.h2database</groupId>

<artifactId>h2</artifactId>

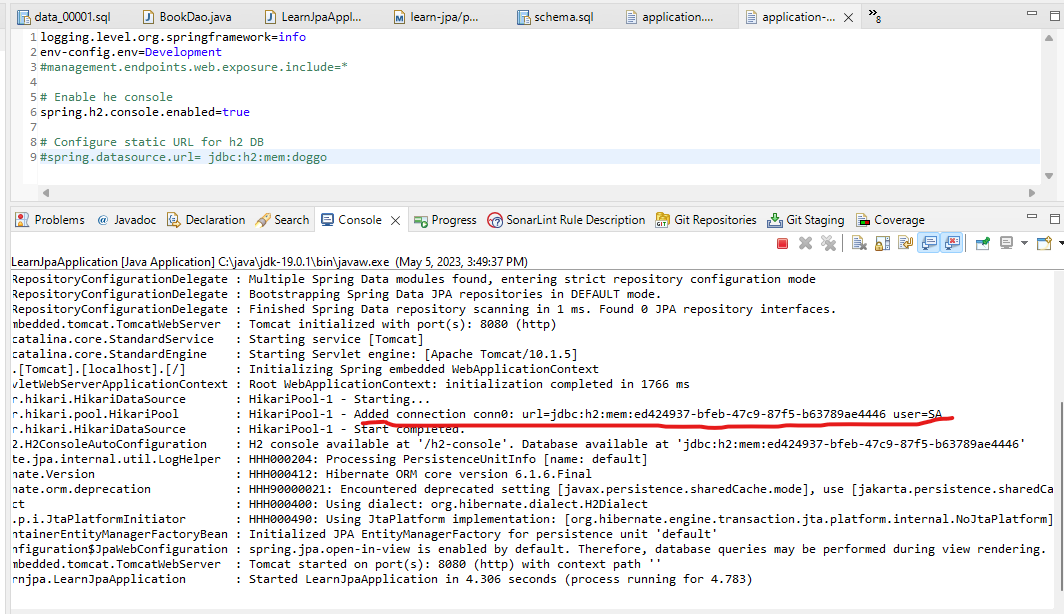
<scope>runtime</scope>

</dependency>

**\*\* Set up a new project on start.spring.io or you can add these dependencies to an existing project. I have created a new project for this guide. The Github link is** [**https://github.com/vivekbirdi/spring-boot-jpa.git\*\***](https://github.com/vivekbirdi/spring-boot-jpa.git**)

**H2 DB Settings (In Memory):**

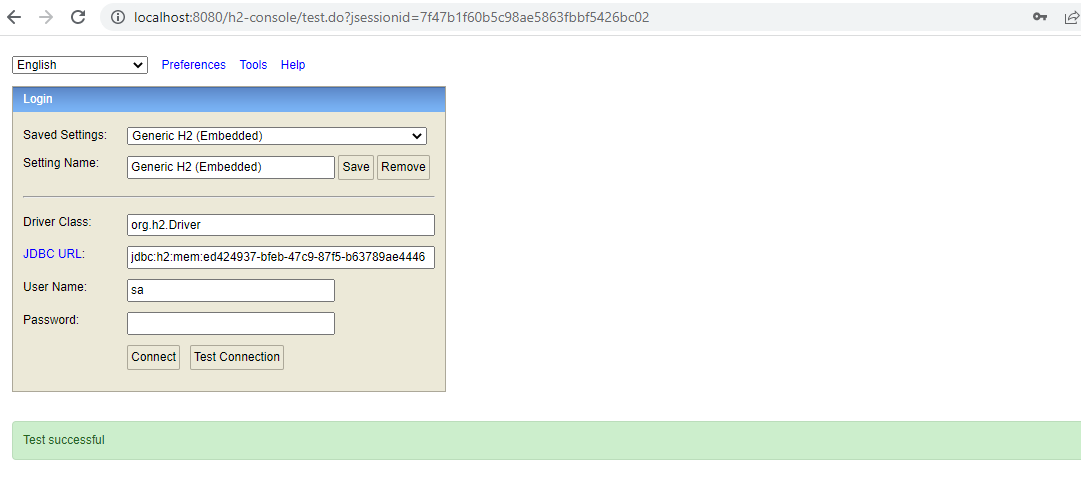
**Connecting H2 DB Connection:** You can see the default H2 DB URL in the logs when we start the Spring-boot application.



To access the H2 DB, we need to enable H2 console. This can be done by adding a property in application.properties files.

# Enable the console

spring.h2.console.enabled=true

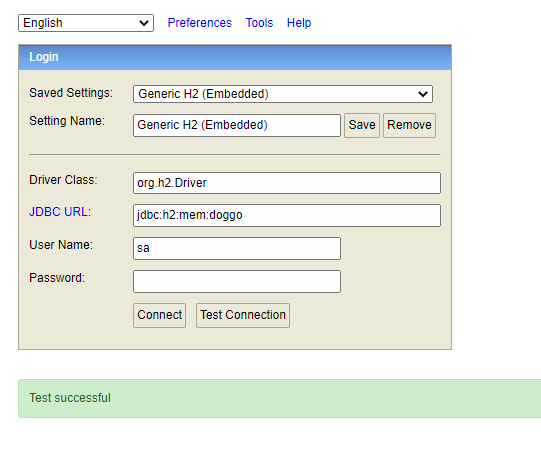
Now after starting the application, you can go to the browser and open <http://localhost:8080/h2-console>. It will open the xH2 DB connection parameter dialogue box. Enter the DB URL from the logs and test the connection (Please refer to below screenshot).

**Setting up static DB URL:** The URL in logs will always change when you restart the application. We can make it static by adding an entry in the application.properties file.

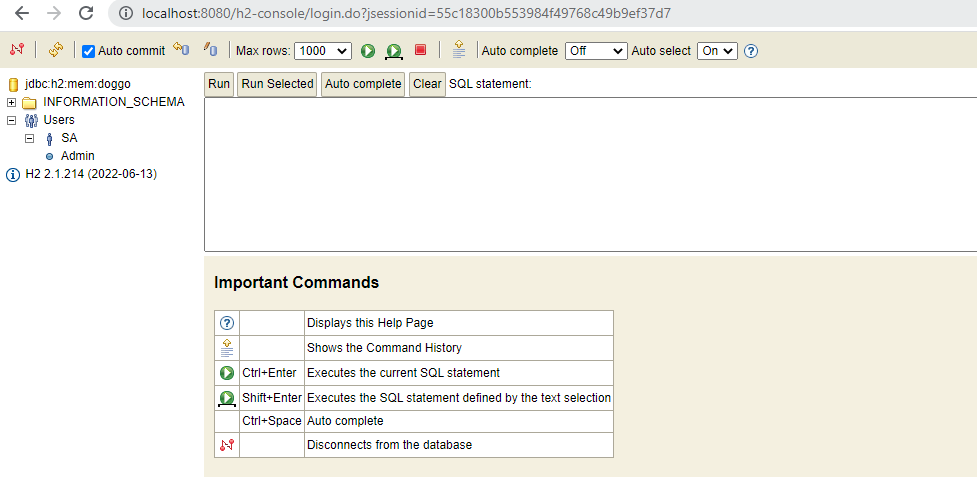
# Configure static URL for h2 DB

spring.datasource.url= jdbc:h2:mem:doggo

Now restart the application and you can use jdbc:h2:mem:doggo as the DB URL.

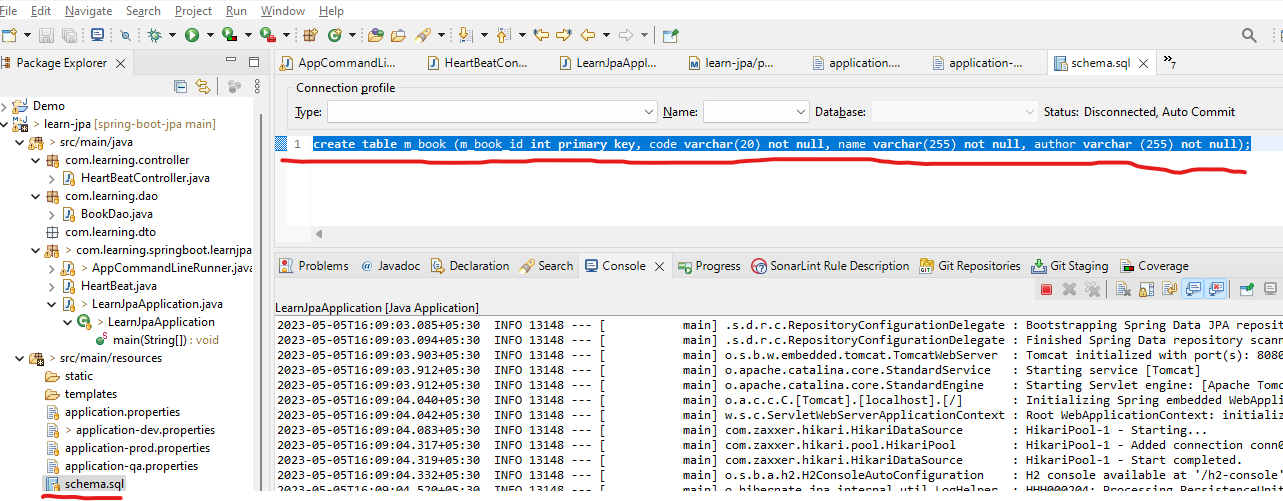


Now click on Connect button to connect to the H2 DB. You will be able to see the empty database.

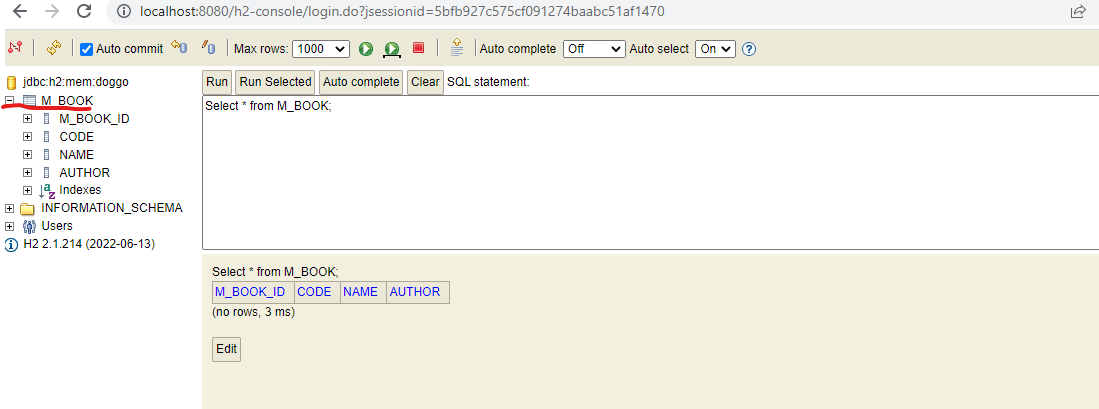


**Create Table in H2 DB:** To create the tables in H2 DB, we need to place schema.sql in the resources folder. When the application starts, the tables defined in schema.sql will be created in the DB. Below is the Table definition, which I have added in the schema.sql.

**create** **table** m\_book (m\_book\_id **int** **primary** **key**, code **varchar**(20) **not** **null**, name **varchar**(255) **not** **null**, author **varchar** (255) **not** **null**);

****

In H2 console, you can see M\_BOOK table is created automatically after the application is started.

****

**Note: This is in-memory database, so when the application is restarted all the data added to tables will be deleted. If we want to add some data in the table at the start of the application, there is a way for that, which we will discuss later.x**

**Spring JDBC:** Spring JDBC removes a lot of boilerplate code for JDBC connection and provides a simple solution to communicate with Database.We will work on the simple insertion/ query examples using Spring JDBC.

**Inserting Data in the table using Spring JDBC:** To communicate with the database in Spring we first need to create a class using annotation @*Repository*. In this Class, we Autowire the object of *JdbcTemplate* Class. Below is the sample code for JDBC Insert (You can find all this code in Git Repo):

**package** com.learning.dao;

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

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.stereotype.Repository;

/\*\*

\* BookDao for CRUD operations on M\_Book table

\* **@author** Vivek Birdi

\*

\*/

@Repository

**public** **class** BookDao {

@Autowired

**private** JdbcTemplate jdbcTemplate;

**private** String initQuery = """

INSERT INTO M\_BOOK (M\_BOOK\_ID, CODE, NAME, AUTHOR)

values (1,'1001','Spring Boot','Doggo');

""";

**public** **void** initInsert() {

jdbcTemplate.update(initQuery);

}

}

For insertion, update, and deletion JDBC Template has *update* method with different versions. You can use it according to the requirements.

Now if we want to execute this query at the start of the application, you can use the *CommanLineRunner* interface of the Spring framework.

**package** com.learning.springboot.learnjpa;

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

**import** org.springframework.boot.CommandLineRunner;

**import** org.springframework.stereotype.Component;

**import** com.learning.dao.BookDao;

/\*\*

\* CommandLineRunner for executing code as the server gets started.

\* **@author** Vivek Birdi

\*

\*/

@Component

**public** **class** AppCommandLineRunner **implements** CommandLineRunner{

@Autowired

**private** BookDao bookDao;

@Override

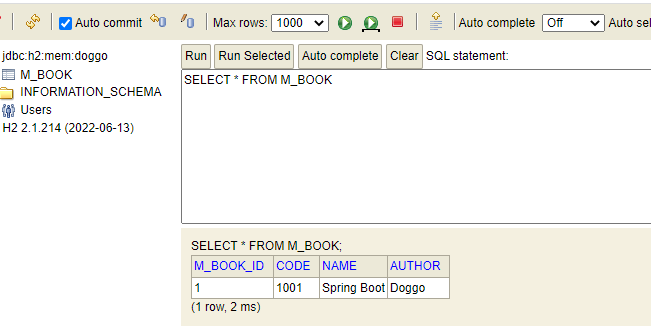
**public** **void** run(String... args) **throws** Exception {

bookDao.initInsert();

}

}

When you will start the application, the insert query written in BookDao will be executed by AppCommandLineRunner. Now if you run the select query on the M\_BOOK table in the h2 console, you will see that 1 record is returned as a result.



**This is how you can populate the data in H2 Tables when an application is started.**

**Populating data in the DB Table dynamically:** The insert query written in BookDao will populate static data in Book Table. If you need to insert dynamically, you can write a parameterized query. Below is an example.

**private** String dynamicInsertQuery = """

INSERT INTO M\_BOOK (M\_BOOK\_ID, CODE, NAME, AUTHOR)

values (?,?,?,?);

""";

Refer to the below method to make insertion dynamic:

**public** **void** insert(Book book) {

jdbcTemplate.update(dynamicInsertQuery, book.getBookId(), book.getCode(), book.getName(), book.getAuthor());

}

You can use this template to update and delete queries too according to your need.

Now let’s call the *insert* method from the command line runner to populate the data in M\_Book Table.

@Override

**public** **void** run(String... args) **throws** Exception {

bookDao.initInsert();

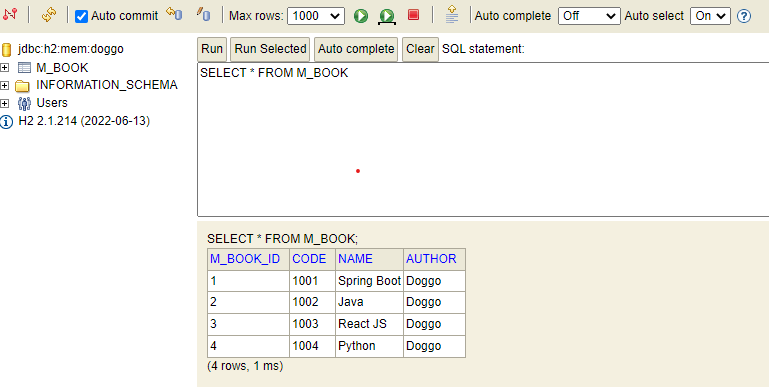
bookDao.insert(**new** Book(2, "1002", "Java", "Doggo"));

bookDao.insert(**new** Book(3, "1003", "React JS", "Doggo"));

bookDao.insert(**new** Book(4, "1004", "Python", "Doggo"));

}

When the application will be started, you will be able to see these records in the M\_Book table from H2 Console.



**SELECT Query using Spring JDBC:** To select row/ rows from the database, Spring JDBC provides multiple versions of the *executeQuery* method. Below are examples:

* **Return Single Record:**

Let’s write a select query in BookDao, which will return a single record based on m\_book\_id.

**private** String selectSingle = "Select \* from M\_Book WHERE M\_Book\_ID = ?";

Now we require to execute this query using Spring JDBC and will convert the result of the query into the object of the class Book. Below are the properties of the class Book.

**public** **class** Book {

**private** **long** bookId;

**private** String code;

**private** String name;

**private** String author;

**public** Book() {

// Empty

}

…

}

The names of the columns of the M\_Book table are:

M\_Book\_ID, code, name and author.

The *executeQuery* method of the Spring JDBC accepts the RowMapper object, which maps the columns of the table to the properties of the POJO class. If the names of the columns of the table are matching with the POJO class, then we can use the object of the *BeanPropertyRowMapper* class of the Spring Framework. It will automatically map the matching columns of the table to the matching properties.

In our case, the property bookId is different from the column m\_book\_id. So *BeanPropertyRowMapper* will map other matching properties except for bookId. In this case, we can use a custom RowMapper. Below is an example:

**public** Book findById(**long** bookId) {

RowMapper<Book> rowMapper = **this**::mapBook; // Custom Row Mapper

**return** jdbcTemplate.queryForObject(selectSingle, rowMapper, bookId);

}

**private** Book mapBook (ResultSet rs, **int** row) {

Book book = **new** Book();

**try** {

book.setBookId(rs.getInt("m\_book\_id"));

book.setCode(rs.getString("code"));

book.setName(rs.getString("name"));

book.setAuthor(rs.getString("author"));

}**catch** (Exception e) {

e.printStackTrace();

}

**return** book;

}

Let’s call the code from *CommandLineRunner* to see the result (Insert the below code in CommandLineRunner and start the application).

// Select a single example

Book book= bookDao.findById(2);

System.***out***.println(book);

// end

In the console, you will see the below output:

Book [bookId=2, code=1002, name=Java, author=Doggo]

If the name of the columns would be matching with the name of the properties, then you would not be required to write a custom Rowmapper. Instead, you could use BeanPropertyRowMapper. Below is the example code.

**return** jdbcTemplate.queryForObject(selectSingle, **new** BeanPropertyRowMapper<>(Book.**class**), bookId);

If we use this code in our example, it will give the below output in the console:

Book [bookId=0, code=1002, name=Java, author=Doggo]

Note that the value of the bookId field is 0 as it is not matching with the name of the column. The rest of the fields are matching with the column names, so values are mapped properly.