**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>

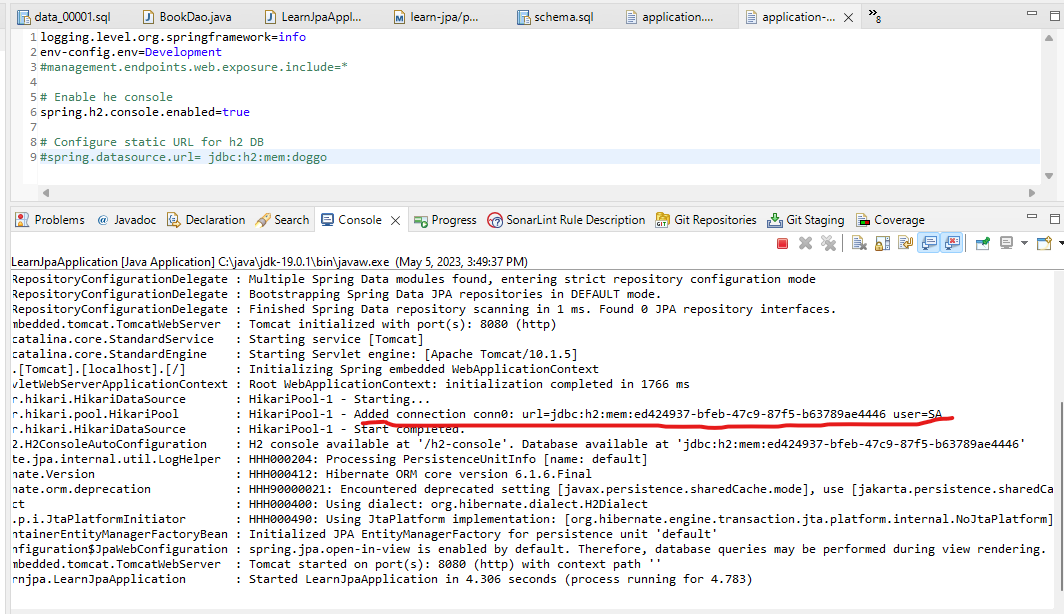
<scope>runtime</scope>

</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):**

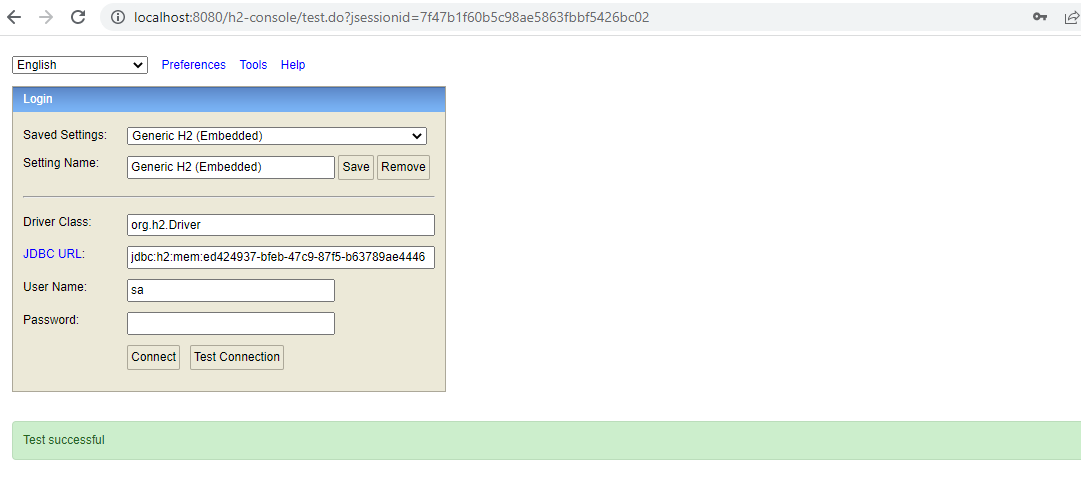
**Connecting H2 DB Connection:** You can see the default H2 DB URL in the logs when we start the Spring-boot application.



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

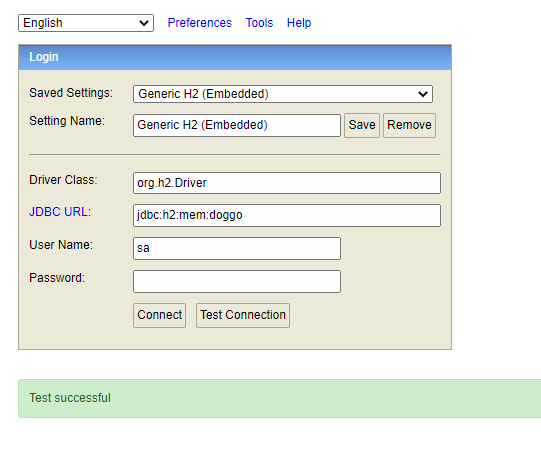
Now after starting the application, you can go to the browser and open <http://localhost:8080/h2-console>. It will open the xH2 DB connection parameter dialogue box. Enter the DB URL from the logs and test the connection (Please refer to below screenshot).

**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.

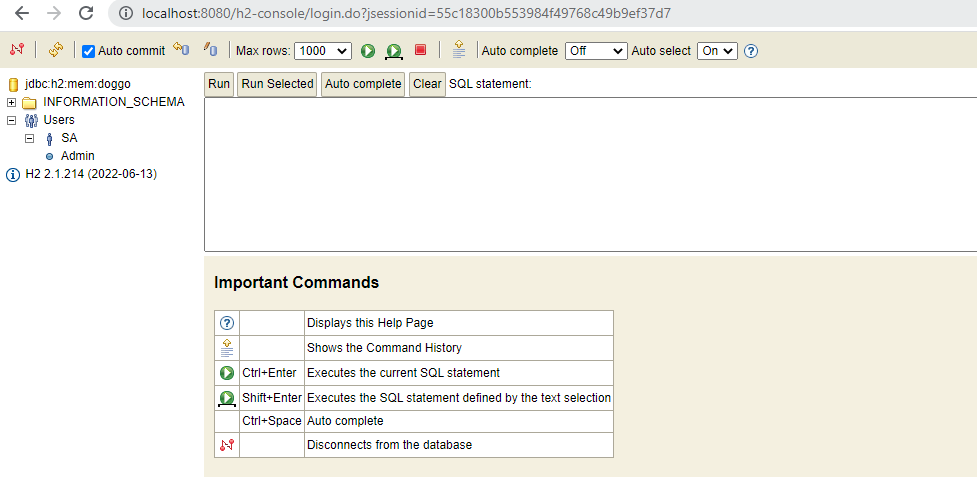
# Configure static URL for h2 DB

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

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

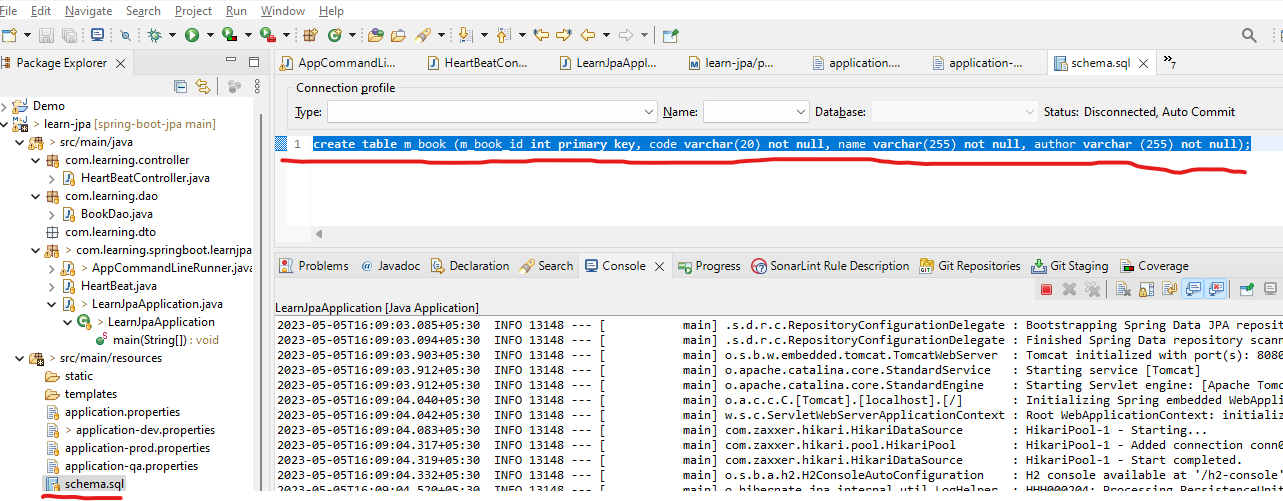


Now click on Connect button to connect to the H2 DB. You will be able to see the empty database.

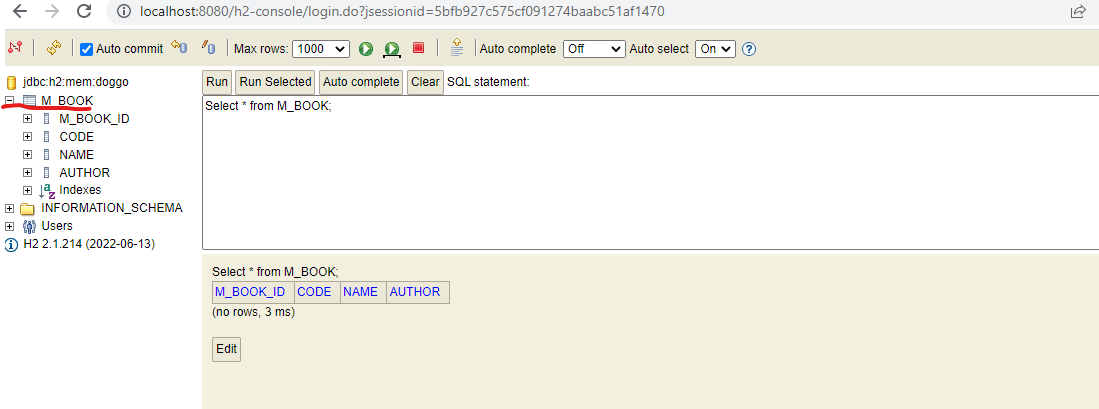


**Create Table in H2 DB:** To create the tables in H2 DB, we need to place schema.sql in the resources folder. When the application starts, the tables defined in schema.sql will be created in the DB. Below is the Table definition, which I have added in the schema.sql.

**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

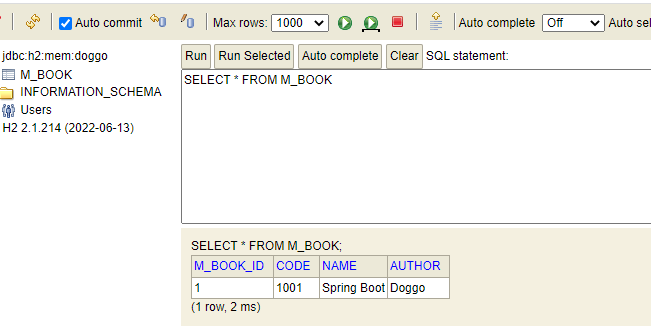
**public** **void** run(String... args) **throws** Exception {

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();

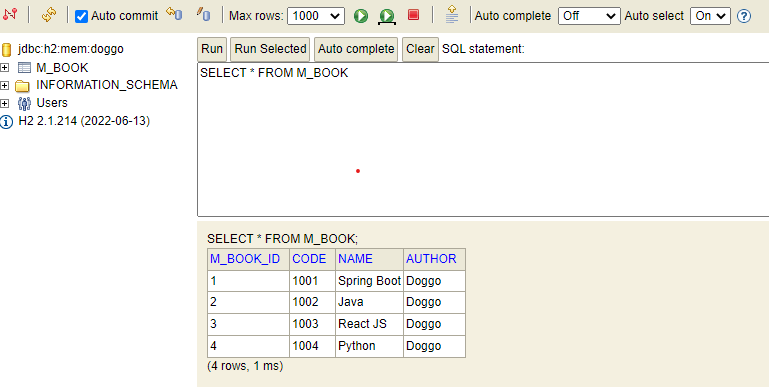
bookDao.insert(**new** Book(2, "1002", "Java", "Doggo"));

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.

* **Return multiple records:**

Let’s write the query to return all the rows in the M\_Book Table in the *BookDao* Class.

**private** String selectAll = "Select \* from M\_Book";

We will use the same POJO *Book* and the same RowMapper as used in the example for selecting the single record. To select multiple records with a custom RowMapper, Spring JDBCTempalate provides an overloaded version of the *queryForStream* method.

**public** List<Book> findAll() {

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

**return** jdbcTemplate.queryForStream(selectAll, rowMapper).toList();

}

Now you can use this method of *BookDao* in the CommandLineRunner for testing.

//2. Select all records example

List<Book> books = bookDao.findAll();

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

//2. end

When you will start the application, you will be able to see the below output in the console (It will print all the records of M\_Book Table):

[Book [bookId=1, code=1001, name=Spring Boot, author=Doggo], Book [bookId=2, code=1002, name=Java, author=Doggo], Book [bookId=3, code=1003, name=React JS, author=Doggo], Book [bookId=4, code=1004, name=Python, author=Doggo]]

**\*\* The above examples are for basic operations using Spring JDBC and are a small introduction. Spring JDBC is capable of supporting a wide range of database operations required in the application. \*\***

**JPA- Java Persistence API:** It is a set of specifications that define how Java objects can be stored, retrieved, and managed in a relational database. JPA provides a high-level object-relational mapping (ORM) framework that allows developers to interact with databases using Java objects, without having to write SQL queries directly.

JPA defines a set of standard interfaces and annotations that Java developers can use to map Java classes to database tables, define relationships between classes, and query the database. JPA implementations provide runtime support for these interfaces and annotations, allowing developers to interact with databases using standard Java APIs.

Some popular implementations of the JPA are Hibernate, EclipseLink, and OpenJPA.

Let’s take the simple example of M\_Subject Table. We need to create the table named M\_Subject, insert some data, delete a row, and retrieve the rows. We can do all these operations using JPA without using SQL queries.

SpringBoot provide provides spring-starter to use JPA in Spring project.

<dependency>

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

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

</dependency>

**Entity Class:**

An entity class is a Java class that represents a persistent object in a database. An entity class is annotated with JPA annotations, which define the mapping between the object's properties and the columns of a database table.

Entity classes are typically used in JPA applications to represent domain objects, such as customers, orders, or in our example, Subjects. Each entity class corresponds to a single database table, and each instance of the entity class represents a single row in that table.

To be considered an entity class, a Java class must meet certain criteria:

* It must be annotated with the @Entity annotation.
* It must have a no-argument constructor that is either public or protected.
* It must have a unique identifier field or property, annotated with the @Id annotation.
* It must have appropriate annotations for each of its fields or properties, defining how they are mapped to columns in the database table.

Suppose we have a table to store the list of Subjects (M\_Subject with columns : m\_subject\_id, code, name, instructor).

Below is the example entity (Subject) for the M\_subject table.

**package** com.learning.entity;

**import** jakarta.persistence.Column;

**import** jakarta.persistence.Entity;

**import** jakarta.persistence.Id;

**import** jakarta.persistence.Table;

@Entity

@Table(name = "m\_subject")

**public** **class** MSubject {

@Id

@Column(name = "m\_subject\_id")

**private** **long** subjectId;

@Column(name = "code")

**private** String code;

@Column(name = "name")

**private** String name;

@Column(name = "instructor")

**private** String instructor;

**public** MSubject() {

// Empty

}

**public** MSubject(**long** subjectId, String code, String name, String instructor) {

**super**();

**this**.subjectId = subjectId;

**this**.code = code;

**this**.name = name;

**this**.instructor = instructor;

}

**public** **long** getSubjectId() {

**return** subjectId;

}

**public** **void** setSubjectId(**long** subjectId) {

**this**.subjectId = subjectId;

}

**public** String getCode() {

**return** code;

}

**public** **void** setCode(String code) {

**this**.code = code;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getInstructor() {

**return** instructor;

}

**public** **void** setInstructor(String instructor) {

**this**.instructor = instructor;

}

@Override

**public** String toString() {

**return** "MSubject [subjectId=" + subjectId + ", code=" + code + ", name=" + name + ", instructor=" + instructor + "]";

}

}

**EntityManager:** An entity manager in JPA is an object that provides a way to interact with the database. It is used to create, read, update, and delete entities. The entity manager also manages the lifecycle of entities, which means that it tracks when an entity is created, when it is modified, and when it is deleted.

The entity manager is an important part of JPA because it provides a consistent way to interact with the database. It also makes it easier to write code that is portable to different databases.

**Below are the uses of the EntityManager:**

* Create new entities
* Find existing entities
* Update existing entities
* Delete existing entities
* Manage the lifecycle of entities
* Persist entities to the database
* Retrieve entities from the database
* Query the database

To use EntityManager, we need to create a class annotated with @Repository annotation (Spring framework annotation).

We can auto-wire the EntityManager into our Repository using @PresistenceContext annotation. Below is the example for the CRUD operations repository on the M\_Subject Table.

**package** com.learning.dao;

**import** java.util.List;

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

**import** com.learning.entity.MSubject;

**import** jakarta.persistence.EntityManager;

**import** jakarta.persistence.PersistenceContext;

**import** jakarta.persistence.Query;

**import** jakarta.transaction.Transactional;

@Repository

@Transactional

**public** **class** SubjectJPADao {

@PersistenceContext

**private** EntityManager entityManager;

// Insert a Record

**public** **void** insert (MSubject subject) {

entityManager.merge(subject);

}

// Select a row

**public** MSubject findById (**long** subjectId) {

**return** entityManager.find(MSubject.**class**, subjectId);

}

// Select All

**public** List<MSubject> findAll(){

String jpql = "SELECT e FROM MSubject e";

Query query = entityManager.createQuery(jpql);

**return** query.getResultList();

}

//Delete a row

**public** **void** deleteById (**long** subjectId) {

MSubject subject = findById(subjectId);

entityManager.remove(subject);

}

}

To execute the example code, we will use the Spring Frameworks’ CommandLineRunner. Below is the example code:

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

**import** java.util.List;

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

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

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

**import** com.learning.dao.SubjectJPADao;

**import** com.learning.entity.MSubject;

@Component

**public** **class** JPACommandLineRunner **implements** CommandLineRunner {

@Autowired

**private** SubjectJPADao subjectJPADao;

@Override

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

// JPA Insert - Start

subjectJPADao.insert(**new** MSubject(1, "1001", "Relational Database", "Mr. Doland Trump"));

subjectJPADao.insert(**new** MSubject(2, "1002", "Programming Languages", "Mr. John Whick"));

subjectJPADao.insert(**new** MSubject(3, "1003", "Operating System", "Ms. Alia Bhatt"));

subjectJPADao.insert(**new** MSubject(4, "1004", "Data Structure", "Ms. Shakira"));

// JPA Insert - End

// JPA Select - Start

MSubject subject = subjectJPADao.findById(1);

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

// JPA Select - End

// JPA Select All- Start

List<MSubject> subjects = subjectJPADao.findAll();

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

// JPA Select All - End

// JPA Delete Single - Start

subjectJPADao.deleteById(4);

// JPA Delete Single - End

// JPA Select All- Start

List<MSubject> remainingBooks = subjectJPADao.findAll();

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

// JPA Select All - End

}

}

The *JPACommandLineRunner* will be executed when the application is started and operations written in the Repository class will be executed in the order mentioned in the ***run*** method of the *JPACommandLine*.

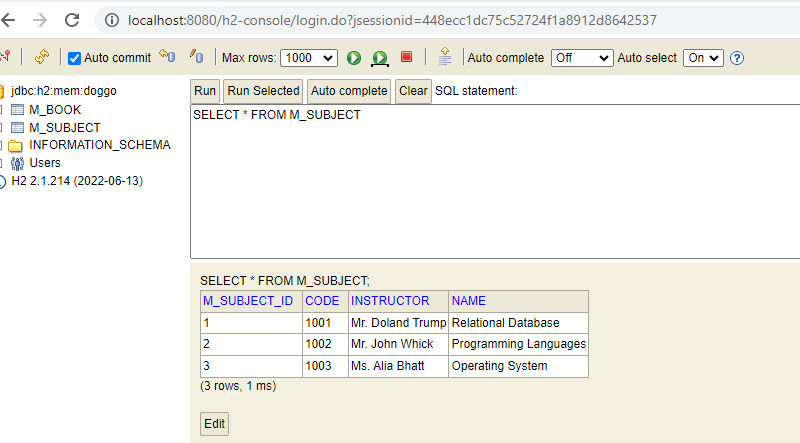
The run() method is the entry point for the CommandLineRunner interface. This method is called when the application starts up.

The insert() method of the SubjectJPADao class is used to insert a new subject into the database. The findById() method is used to select a subject by ID. The findAll() method is used to select all subjects from the database. The deleteById() method is used to delete a subject by ID.

After starting the application, you can check the H2Console:

* M\_Subject table will be automatically created.
* Data inserted at application startup can be queried from the M\_Subject Table.

Please refer to the below screenshot of the H2Console



You can refer to the code from GIT Repo: <https://github.com/vivekbirdi/spring-boot-jpa.git>

**Spring Data JPA:**

With Springboot and JPA, we use EntityManager in the Repository class to perform Database operations. Spring Data JPA further simplifies the database operations and eliminates the need to EntityManager in our code.

Spring Data JPA is a library that provides abstractions over the Java Persistence API (JPA). It makes it easier to develop applications that store data in a relational database. Spring Data JPA provides a number of features, including:

* Repository interfaces: Spring Data JPA provides repository interfaces that can be used to interact with the database. These interfaces are automatically implemented by Spring Data JPA, so you don't need to write any code to access the database.x
* Query methods: Spring Data JPA provides a number of query methods that can be used to retrieve data from the database. These methods are automatically generated based on the entity class, so you don't need to write any code to create queries.
* Transaction management: Spring Data JPA automatically manages transactions. This means that you don't need to worry about manually starting and committing transactions.

Overall, Spring Data JPA is a powerful library that can help you to develop applications that store data in a relational database.

Let’s take the example of M\_Department table with three fields M\_Department\_ID, Code, and Name. We will use Spring Data JPA for the curd operations at M\_Deparment Table.

First, we will create *Department* Entity (Note that if the name of the column and class variable are the same, we do not require to map as JPA will use the variable name as the column name).

**package** com.learning.entity;

**import** jakarta.persistence.Column;

**import** jakarta.persistence.Entity;

**import** jakarta.persistence.Id;

**import** jakarta.persistence.Table;

@Entity

@Table(name = "m\_department")

**public** **class** Department {

@Id

@Column(name = "m\_department\_id")

**private** **long** id;

**private** String code;

**private** String name;

**public** Department() {

// Empty constructor

}

**public** Department(**long** id, String code, String name) {

**super**();

**this**.id = id;

**this**.code = code;

**this**.name = name;

}

**public** **long** getId() {

**return** id;

}

**public** **void** setId(**long** id) {

**this**.id = id;

}

**public** String getCode() {

**return** code;

}

**public** **void** setCode(String code) {

**this**.code = code;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

@Override

**public** String toString() {

**return** "Department [id=" + id + ", code=" + code + ", name=" + name + "]";

}

}

In Spring Data JPA, we do not need to create a Repository class. It provides an interface JpaRepository to. We need to create another interface extending this interface and query the database. Below is an example:

**package** com.learning.repo;

**import** org.springframework.data.jpa.repository.JpaRepository;

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

**import** com.learning.entity.Department;

@Repository

**public** **interface** DepartmentRepository **extends** JpaRepository<Department, Long> {

}

To showcase the use of DepartmentRepository, I will create another command runner to write the CRUD operations.

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

**import** java.util.List;

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

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

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

**import** com.learning.entity.Department;

**import** com.learning.repo.DepartmentRepository;

@Component

**public** **class** SpringDataJPACommandLineRunner **implements** CommandLineRunner{

@Autowired

**private** DepartmentRepository repository;

@Override

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

// Insert records in Department Table

repository.save(**new** Department(1l, "1001", "Computer"));

repository.save(**new** Department(2l, "1002", "Electrical"));

repository.save(**new** Department(3l, "1003", "Electronics"));

repository.save(**new** Department(4l, "1004", "Mechanical"));

// Delete records from Department Table

repository.deleteById(4l);

// Select from Department Table

List<Department> departments = repository.findAll();

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

}

}

The above code is sufficient for CRUD operations. We do not need to write a Repository class to write the basic database operations using EntityManager. Here, in this example, we just Autowired the DepartmentRepository. The JpaRepository interface provides a lot of methods for database operation. You can refer to the official documentation.

**Note**: In order to Autowire the DepartmentRepository, we need to add JpaRepositoryScan on the Springboot Startup class. If the package where you are writing your Repository interfaces is already under the default package-scan, then you can avoid this step.

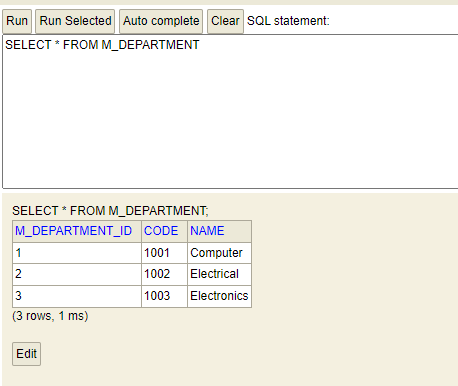
@EnableJpaRepositories ("com.learning.repo")

**public** **class** LearnJpaApplication {

After starting the application, you can check the H2Console:

* M\_Department table will be automatically created.
* Data inserted at application startup can be queried from the M\_ Department Table.

Below is the screen shot of H2Console:



You can refer to the code from GIT Repo: <https://github.com/vivekbirdi/spring-boot-jpa.git>

**Adding custom search in Repository:** To search by any other column than the ID column, we can need to add methods in the Repository Interface. We will just declare the method without any definition. To make it work, we need to follow the nomenclature:

* Method should start with *findBy* word.
* Should be followed by the name of the column by which you want to search. e.g. if we need to search by the *code* column, then the method name will be *findByCode*.
* We can add **AND/OR** conditions, but here we will go with a simple example.

So, to search records by code, you just need to add to declare a method in DepartmentRepository Interface as below:

**public** List<Department> findByCode(String code);

Now, you can directly use this method in the command line runner and get the output:

System.***out***.println("\*\*\*\*\*\*\*\*\* Find by code \*\*\*\*\*\*");

List<Department> departmentsByCode = repository.findAll();

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

Upon starting application, below logs will be printed in console:

\*\*\*\*\*\*\*\*\* Find by code \*\*\*\*\*\*

[Department [id=1, code=1001, name=Computer]].