


JAVA ACADEMY - XIDERAL MONTERREY, N.L

A large, light green, semi-transparent watermark of the Spring logo is centered in the background. It consists of a stylized 'S' shape with a circular arrow around it, all within a hexagonal frame.

EXERCISE - WEEK 3 SPRING DATA JPA Example

Made by:
Luis Miguel Sánchez Flores

INTRODUCTION

With modern applications thriving on massive amounts of data, whether for storing user information, tracking transactions or executing analytics, the need of a database connection inside the source code is paramount, allowing it to store, access and manage data in an efficient, dynamic and seamless manner.

Over the years, various technologies and APIs have been developed over time that establish the connection between the source code and the database. For Java, database connectivity began with the introduction of **JDBC** (Java Database Connectivity), an API that provides a standard interface to interact directly with databases of different types, which allowed developers to execute SQL queries and process the retrieved results.

However, JDBC required developers to write low-level, boilerplate code in order to achieve the desired operation onto the database, making it a tedious and time-consuming process that often led to verbose and error-prone code. This impelled the search for more robust and efficient solutions that allowed developers to interact with the databases more intuitively. This ultimately led to the creation of the **JPA** (Java Persistence API, or now known as the Jakarta Persistence API).

JPA is a higher-level abstraction framework built on top of the JDBC that introduces the concept of Object-Relational Mapping (ORM) that allows it to map persisting Java objects to a relational database. As such, Java developers are able to work with objects instead of raw SQL queries, making the process of performing CRUD operations much easier, improves the readability of the code and enhances the portability between different databases.

JPA can be integrated with the Spring Boot framework, the most popular for building enterprise Java applications that allows developers to further streamline the database operations thanks to its dependency injection and configuration capabilities.


The use of Spring Boot along with JPA represents a modern and effective solution to connect and interact with a database from the Java source code, being able to create robust and feature-rich applications while leaving out the complexities of the database interaction.


Getting Started

We can easily integrate the Spring Boot framework and its JPA dependency by importing a generated Maven or Gradle project, created with the **spring initializer website** that sets up all the dependencies needed for the application such as JPA.

Popular IDEs such as Eclipse have plugins that create Spring Boot projects directly.

A maven project was generated for this case, adding the dependency of Spring Data JPA and MySQL Driver that will allow us to connect with the database:





Project

☒ Gradle - Groovy ☐ Gradle - Kotlin ☒ Java ☐ Kotlin ☐ Groovy

☐ Maven

Spring Boot

☐ 3.4.0 (SNAPSHOT) ☐ 3.4.0 (M2) ☐ 3.3.4 (SNAPSHOT) ☒ 3.3.3 ☐ 3.2.10 (SNAPSHOT) ☐ 3.2.9

Project Metadata

Group

com.example

Artifact

demo

Name

demo

Description

Demo project for Spring Boot

Package name

com.example.demo

Packaging

☒ Jar ☐ War


Java

☐ 22 ☐ 21 ☒ 17

Dependencies

ADD DEPENDENCIES... CTRL + B

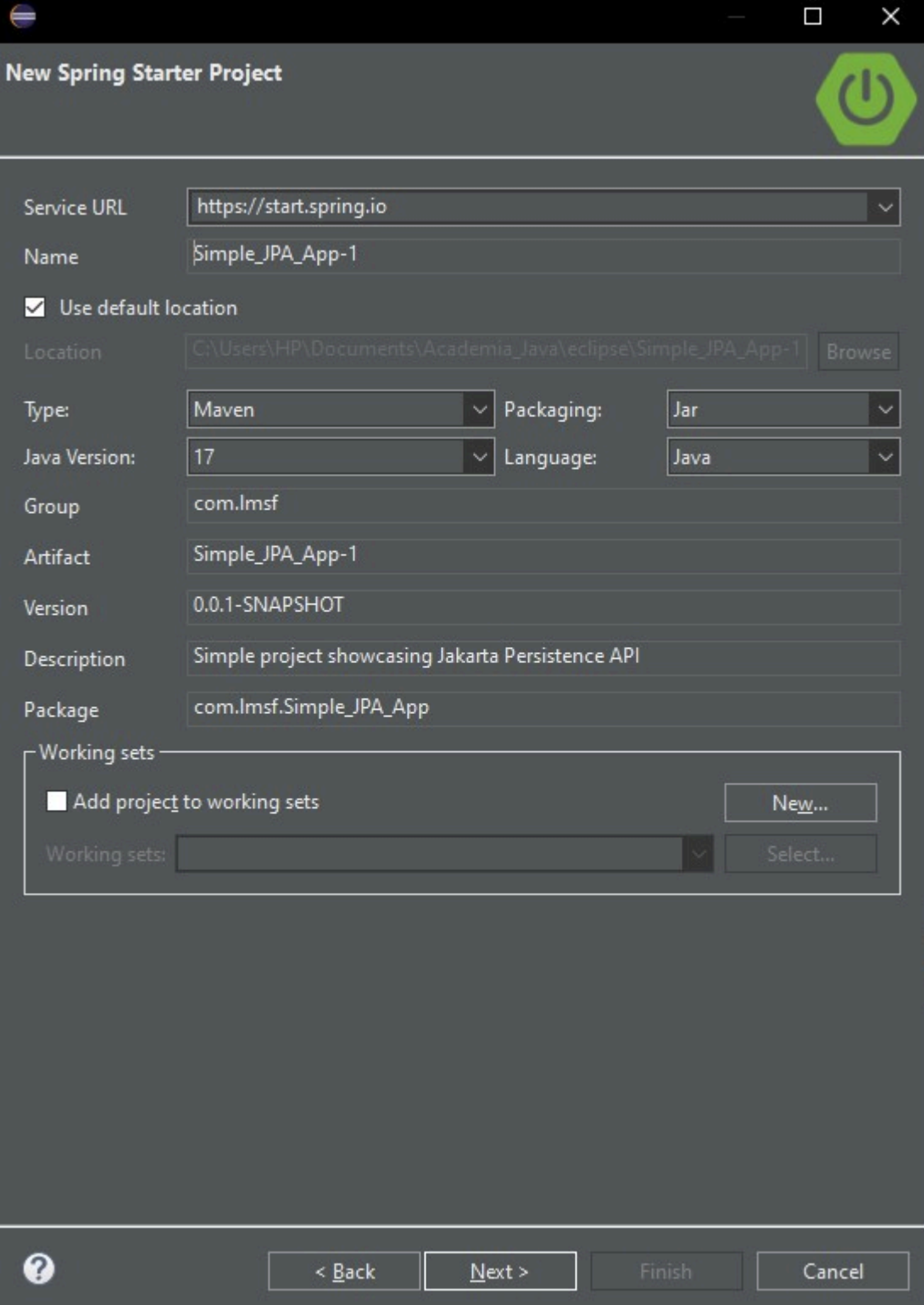
No dependency selected



GENERATE CTRL + G

EXPLORE CTRL + SPACE

SHARE...



The image shows a 'New Spring Starter Project' dialog box with a dark theme. At the top right is a green power button icon. The dialog contains several input fields and checkboxes for configuring a new project. The 'Service URL' is set to 'https://start.spring.io'. The 'Name' is 'Simple_JPA_App-1'. The 'Use default location' checkbox is checked, and the 'Location' is 'C:\Users\HP\Documents\Academia_Java\eclipse\Simple_JPA_App-1'. The 'Type' is 'Maven', 'Packaging' is 'Jar', 'Java Version' is '17', and 'Language' is 'Java'. The 'Group' is 'com.lmsf', 'Artifact' is 'Simple_JPA_App-1', 'Version' is '0.0.1-SNAPSHOT', 'Description' is 'Simple project showcasing Jakarta Persistence API', and 'Package' is 'com.lmsf.Simple_JPA_App'. At the bottom, there is a 'Working sets' section with an unchecked 'Add project to working sets' checkbox and a 'Working sets' dropdown menu. The bottom of the dialog has a question mark icon and four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

New Spring Starter Project

Service URL:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

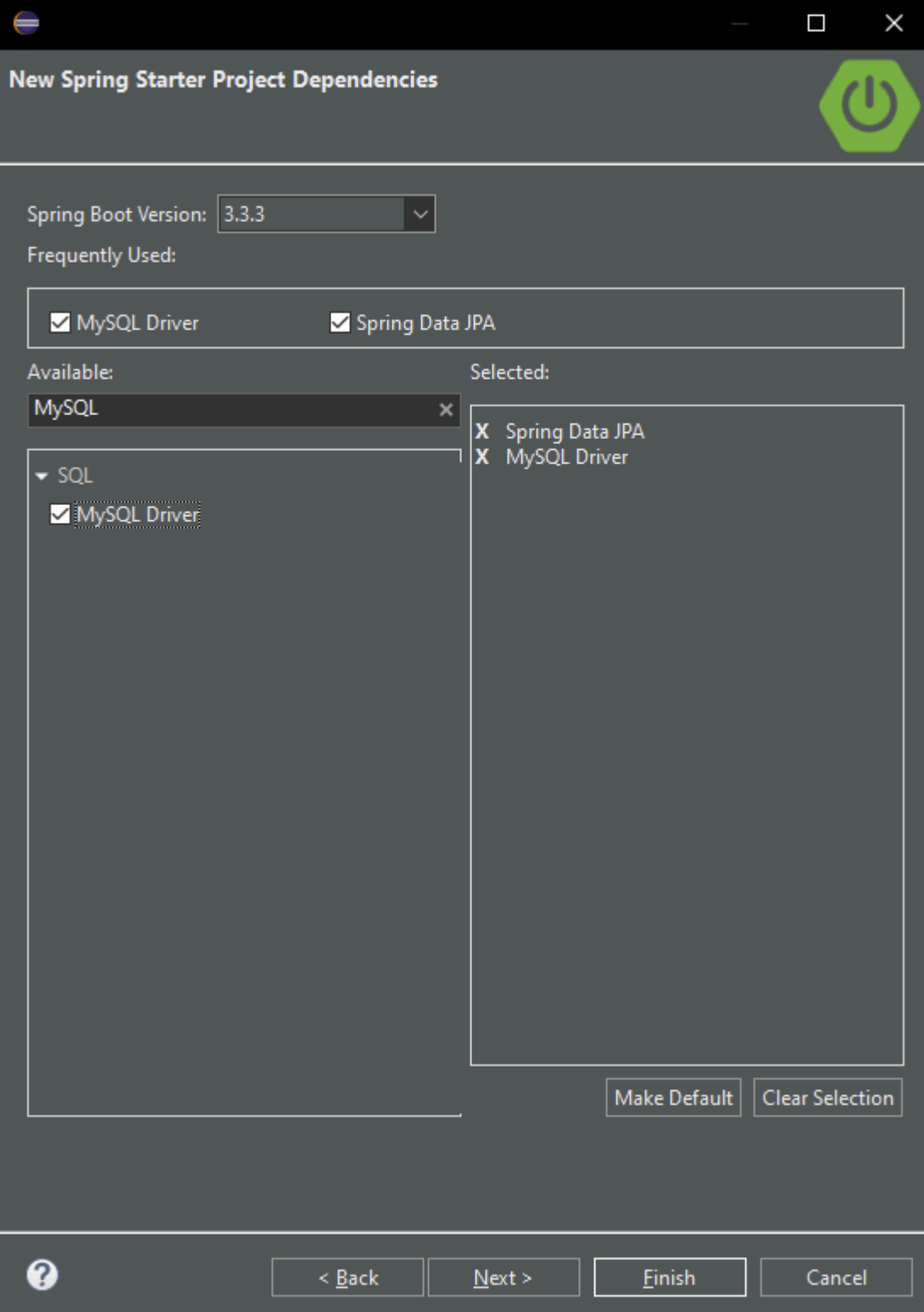
Description:

Package:

Working sets

☐ Add project to working sets

Working sets:



The image shows a 'New Spring Starter Project Dependencies' dialog box. At the top, it has a title bar with a green power button icon. Below the title bar, there's a section for 'Spring Boot Version' with a dropdown menu set to '3.3.3'. Underneath, a 'Frequently Used:' section contains two checked checkboxes: 'MySQL Driver' and 'Spring Data JPA'. The main area is divided into two columns: 'Available:' and 'Selected:'. The 'Available:' column shows a search bar with 'MySQL' entered, a dropdown arrow, and a list with 'SQL' expanded, showing 'MySQL Driver' checked. The 'Selected:' column shows 'Spring Data JPA' and 'MySQL Driver' with 'X' icons. At the bottom right of the main area are 'Make Default' and 'Clear Selection' buttons. The footer contains a help icon, '< Back', 'Next >', 'Finish', and 'Cancel' buttons.

New Spring Starter Project Dependencies

Spring Boot Version: 3.3.3

Frequently Used:

☒ MySQL Driver ☒ Spring Data JPA

Available: Selected:

MySQL

SQL

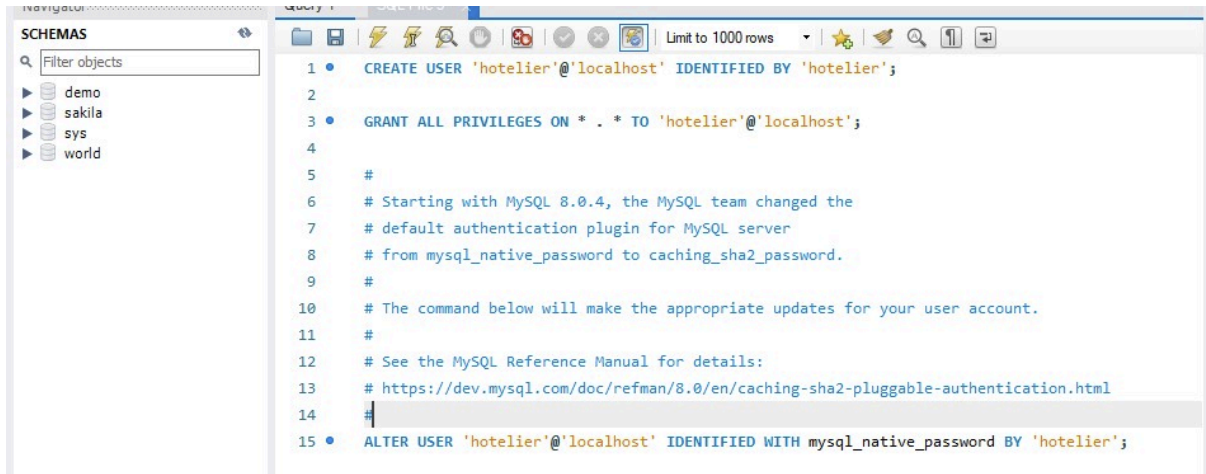
☒ MySQL Driver

X Spring Data JPA
X MySQL Driver

Make Default Clear Selection

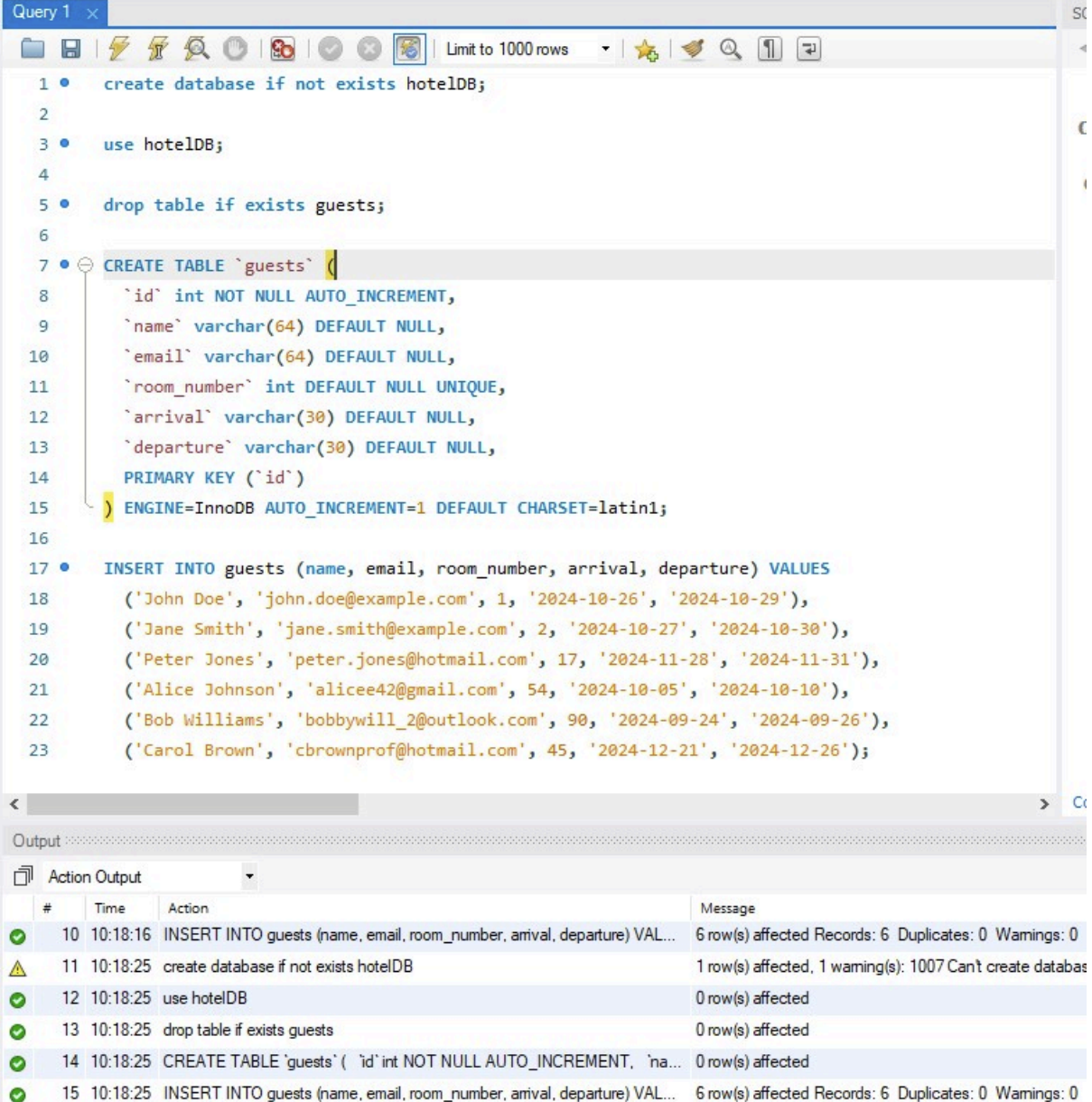
? < Back Next > Finish Cancel

For this Spring JPA example, a mock hotel database was created, generating an admin user of the hotelier along with a table to store guest data such as name, email, room number, etc., executing the following SQL queries:



The screenshot shows a MySQL command-line client interface. On the left, the 'SCHEMAS' panel lists 'demo', 'sakila', 'sys', and 'world'. The main window displays a list of SQL queries numbered 1 through 15. The queries are as follows:

```
1 • CREATE USER 'hotelier'@'localhost' IDENTIFIED BY 'hotelier';
2
3 • GRANT ALL PRIVILEGES ON * . * TO 'hotelier'@'localhost';
4
5 #
6 # Starting with MySQL 8.0.4, the MySQL team changed the
7 # default authentication plugin for MySQL server
8 # from mysql_native_password to caching_sha2_password.
9 #
10 # The command below will make the appropriate updates for your user account.
11 #
12 # See the MySQL Reference Manual for details:
13 # https://dev.mysql.com/doc/refman/8.0/en/caching-sha2-pluggable-authentication.html
14 #
15 • ALTER USER 'hotelier'@'localhost' IDENTIFIED WITH mysql_native_password BY 'hotelier';
```



The screenshot displays a SQL IDE interface. The top pane, titled 'Query 1', contains the following SQL code:

```
1 • create database if not exists hotelDB;
2
3 • use hotelDB;
4
5 • drop table if exists guests;
6
7 • CREATE TABLE `guests` (
8     `id` int NOT NULL AUTO_INCREMENT,
9     `name` varchar(64) DEFAULT NULL,
10    `email` varchar(64) DEFAULT NULL,
11    `room_number` int DEFAULT NULL UNIQUE,
12    `arrival` varchar(30) DEFAULT NULL,
13    `departure` varchar(30) DEFAULT NULL,
14    PRIMARY KEY (`id`)
15 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1;
16
17 • INSERT INTO guests (name, email, room_number, arrival, departure) VALUES
18     ('John Doe', 'john.doe@example.com', 1, '2024-10-26', '2024-10-29'),
19     ('Jane Smith', 'jane.smith@example.com', 2, '2024-10-27', '2024-10-30'),
20     ('Peter Jones', 'peter.jones@hotmail.com', 17, '2024-11-28', '2024-11-31'),
21     ('Alice Johnson', 'alicee42@gmail.com', 54, '2024-10-05', '2024-10-10'),
22     ('Bob Williams', 'bobbywill_2@outlook.com', 90, '2024-09-24', '2024-09-26'),
23     ('Carol Brown', 'cbrownprof@hotmail.com', 45, '2024-12-21', '2024-12-26');
```

The bottom pane, titled 'Output', shows the execution results in a table format:

#	Time	Action	Message
✓ 10	10:18:16	INSERT INTO guests (name, email, room_number, arrival, departure) VAL...	6 row(s) affected Records: 6 Duplicates: 0 Warnings: 0
⚠ 11	10:18:25	create database if not exists hotelDB	1 row(s) affected, 1 warning(s): 1007 Can't create databas
✓ 12	10:18:25	use hotelDB	0 row(s) affected
✓ 13	10:18:25	drop table if exists guests	0 row(s) affected
✓ 14	10:18:25	CREATE TABLE `guests` (`id` int NOT NULL AUTO_INCREMENT, `na...	0 row(s) affected
✓ 15	10:18:25	INSERT INTO guests (name, email, room_number, arrival, departure) VAL...	6 row(s) affected Records: 6 Duplicates: 0 Warnings: 0

We executed a SELECT operation to ensure that the mock guest data was performed successfully:

Result Grid						
		Filter Rows:		Edit: Export/Import:		
	id	name	email	room_number	arrival	departure
▶	1	John Doe	john.doe@example.com	1	2024-10-26	2024-10-29
	2	Jane Smith	jane.smith@example.com	2	2024-10-27	2024-10-30
	3	Peter Jones	peter.jones@hotmail.com	17	2024-11-28	2024-11-31
	4	Alice Johnson	alicee42@gmail.com	54	2024-10-05	2024-10-10
	5	Bob Williams	bobbywill_2@outlook.com	90	2024-09-24	2024-09-26
	6	Carol Brown	cbrownprof@hotmail.com	45	2024-12-21	2024-12-26
*	NULL	NULL	NULL	NULL	NULL	NULL

In our imported Maven project, the **SimpleJpaAppApplication** was assigned by Spring as the main entry point of the application by adding the `@SpringBootApplication` annotation. Since this project will be a simple command line program, a **CommandLineRunner** method is created that Spring executes after the application context is initialized. In this case, the following message is printed out:

```
1 package com.lmsf.Simple_JPA_App;
2
3 import org.springframework.boot.CommandLineRunner;
4 import org.springframework.boot.SpringApplication;
5 import org.springframework.boot.autoconfigure.SpringBootApplication;
6 import org.springframework.context.annotation.Bean;
7
8 @SpringBootApplication
9 public class SimpleJpaAppApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(SimpleJpaAppApplication.class, args);
13     }
14
15     @Bean
16     public CommandLineRunner commandLineRunner(String[] args) {
17         return runner -> {
18             System.out.println("Simple JPA Project");
19         };
20     }
21
22 }
```

```
default]
2024-08-30T10:24:19.507-06:00 INFO 9376 --- [Simple_JPA_App] [          main]
org.hibernate.Version
: HHH000412: Hibernate ORM core version 6.5.2.Final
2024-08-30T10:24:19.546-06:00 INFO 9376 --- [Simple_JPA_App] [          main]
o.h.c.internal.RegionFactoryInitiator
: HHH000026: Second-level cache disabled
2024-08-30T10:24:19.871-06:00 INFO 9376 --- [Simple_JPA_App] [          main]
o.s.o.j.p.SpringPersistenceUnitInfo
: No LoadTimeWeaver setup: ignoring JPA class transformer
2024-08-30T10:24:20.248-06:00 INFO 9376 --- [Simple_JPA_App] [          main]
o.h.e.t.j.p.i.JtaPlatformInitiator
: HHH000489: No JTA platform available (set
'hibernate.transaction.jta.platform' to enable JTA platform integration)
2024-08-30T10:24:20.253-06:00 INFO 9376 --- [Simple_JPA_App] [          main]
j.LocalContainerEntityManagerFactoryBean
: Initialized JPA EntityManagerFactory for persistence
unit 'default'
2024-08-30T10:24:20.418-06:00 INFO 9376 --- [Simple_JPA_App] [          main]
c.l.S.SimpleJpaAppApplication
: Started SimpleJpaAppApplication in 2.416 seconds
(process running for 4.94)
Simple JPA Project
2024-08-30T10:24:20.427-06:00 INFO 9376 --- [Simple_JPA_App] [ionShutdownHook]
j.LocalContainerEntityManagerFactoryBean
: Closing JPA EntityManagerFactory for persistence unit
'default'
2024-08-30T10:24:20.428-06:00 INFO 9376 --- [Simple_JPA_App] [ionShutdownHook]
com.zaxxer.hikari.HikariDataSource
: HikariPool-1 - Shutdown initiated...
2024-08-30T10:24:20.435-06:00 INFO 9376 --- [Simple_JPA_App] [ionShutdownHook]
com.zaxxer.hikari.HikariDataSource
: HikariPool-1 - Shutdown completed.
```

In order to set up the database to which Spring must be connected to, we can simply specify the URL, username and password to use in the project's **application.properties** file, and Spring will automatically apply the appropriate driver to connect to the database successfully:

```
application.... 02-student... SimpleJpaAp... application.... x table-setup.sql >>4
1 spring.datasource.url=jdbc:mysql://localhost:3306/hotelldb
2 spring.datasource.username=hotelier
3 spring.datasource.password=hotelier
4
5 spring.application.name=Simple_JPA_App
6 |
```

Mapping Java objects to MySQL database

To begin mapping the Java objects with the connected database, we created the Guest class that is labeled as an **Entity** class thanks to the **@Entity** annotation, along with a **@Table** annotation that helps out the JPA into mapping the Java class with the corresponding “guest” table of the database. JPA requires entities to have a no-argument constructor for the mapping to work.

The Optional `@Column` annotation is used to align the Java fields with the matching database columns. If it's left out, Spring assumes that both properties share the same names.

The `@Id` annotation designates the variable that acts as the primary key of the entity, while the `@GeneratedValue` points out that the variable's value is generated automatically.

```
5 @Entity // <- Used to indicate that the class is mapped to a database table
6 @Table(name="guests") // <- Map to the corresponding table
7 public class Guest {
8
9     @Id // <- PRIMARY KEY
10    @GeneratedValue(strategy=GenerationType.IDENTITY) // <- Specify how the primary keys for new objects are assigned
11    @Column(name="id") // <- Map to the corresponding column of the table
12    private int id;
13
14    @Column(name="name")
15    private String name;
16
17    @Column(name="email")
18    private String email;
19
20    @Column(name="room_number")
21    private int roomNum;
22
23    @Column(name="arrival")
24    private String arrivalDate;
25
26    @Column(name="departure")
27    private String departureDate;
28
29    // DEFAULT CONSTRUCTOR REQUIRED BY JPA
30    public Guest() {}
```

In order to interact with the database, a **Data Access Object (DAO)** pattern is applied, creating an interface that encapsulates all access to the data source and will enable us to choose different kinds of implementations in the future that better suit our needs, improving the maintainability of the code and ensuring a safe and consistent interaction with the database.

In this case, a GuestDAO interface is created that establishes some common CRUD operations that future implementations must define, such as adding guests, removing a guest by their ID, or searching guests by a specific name:

```
3+ import java.util.List;
6
7 // Data-Access-Object (DAO) to communicate with the database
8
9 public interface GuestDAO {
10
11     Guest getGuestByID(int id);
12     List<Guest> getGuestsByName(String name);
13     List<Guest> getAllGuests();
14     void addGuest(Guest guest);
15     void updateGuest(Guest guest);
16     void removeGuest(int id);
17 }
18
```

Within one of the DAO implementations (**GuestImplementation**), an `EntityManager` is initialized.

The **EntityManager** is an essential component that manages the lifecycle of the entity's instances introduced during the program's execution, and it is responsible for the execution of the CRUD operations. The `EntityManager` requires a `Data Source` that contains information about the database connection. Both the `EntityManager` and the `Data Source` are managed automatically by Spring Boot.

```
4
5 import org.springframework.stereotype.Repository;
6
7 import com.lmsf.Simple_JPA_App.entities.Guest;
8
9 import jakarta.persistence.EntityManager;
10 import jakarta.persistence.TypedQuery;
11 import jakarta.transaction.Transactional;
12
13 @Repository
14 public class GuestImplementation implements GuestDAO {
15
16     private EntityManager entityManager;
17
18
19     public GuestImplementation(EntityManager entityManager) {
20         // Set up the main component that is the EntityManager
21         this.entityManager = entityManager;
22     }
23 }
```

The EntityManager offers built-in methods that encapsulates the CRUD operations and allows developers to work with domain Java objects instead of direct SQL queries, making them easier to use programmatically. Such methods are:

- **persist()** for creating new records
- **find()** for reading existing records
- **merge()** for updating records
- **remove()** for deleting records

```
23
24● @Override
25 @Transactional
26 public void addGuest(Guest guest) {
27     entityManager.persist(guest); // <- Simple command to save the guest into the database...
28 }
29
30
31● @Override
32 @Transactional
33 public void updateGuest(Guest guest) {
34     entityManager.merge(guest); // <- Updates the entity
35 }
36
37● @Override
38 @Transactional
39 public void removeGuest(int id) {
40     // retrieve the student
41     Guest retrievedGuest = entityManager.find(Guest.class, id);
42     entityManager.remove(retrievedGuest); // <- Removes the guest from the database...
43 }
44 }
45
```

Spring can automatically start and finish transaction processes through the use of the `@Transactional` annotation, ensuring data consistency across various database operations.

The `EntityManager` supports the use of **JPQL** (JPA Query Language), which allows developers to write database queries using Java syntax based on the name and field of an entity, eliminating the need to write raw SQL queries and improve portability and maintainability of the code.

In this case, the JPQL is used to develop queries that retrieve all the guests information from the table, or retrieve guests that contain a specific string in their name, as shown below:

```
53 • @Override
54 public List<Guest> getGuestsByName(String name) {
55     // Create a query that retrieves the guests that contains a specific name:
56     TypedQuery<Guest> searchQuery = entityManager.createQuery(
57         "FROM Guest WHERE name LIKE :specificName", Guest.class
58     );
59
60     // Set up the parameter query to that of the function's parameter
61     searchQuery.setParameter("specificName", "%" + name + "%");
62
63     // Return all the records that meets the condition (if any):
64     return searchQuery.getResultList();
65 }
66
67 • @Override
68 public List<Guest> getAllGuests() {
69     // Retrieve ALL the guest records from the table.
70     TypedQuery<Guest> searchQuery = entityManager.createQuery(
71         "FROM Guest", Guest.class
72     );
73
74     // Return said records:
75     return searchQuery.getResultList();
76 }
77
78 }
```

Testing it out

With our code all setup to interact with the mock hotel database and guest table, multiple functions were developed in the main class SimpleJpaAppApplication, from which the methods of the DAO implementation passed as a parameter will be tested, such as adding the guest to the table, retrieving a guest based on its ID, updating the room number of a guest, deleting a guest's information from the table, among others:


```
51
52 private void addGuest(GuestDAO guestDAO, Guest guest) {
53     System.out.println("\nAdding the new guest to the database.... \n" + guest);
54
55     // Add the provided guest onto the database...
56     guestDAO.addGuest(guest);
57
58     System.out.println("Guest was successfully added!");
59     System.out.println("NEW ID : " + guest.getId());
60 }
61
62
63 private void searchGuest(GuestDAO guestDAO, int id) {
64     System.out.println("\nRetrieving guest info by ID "+id+"...");
65
66     // Pass the ID argument into the DAO function to retrieve the guest:
67     Guest retrievedGuest = guestDAO.getGuestByID(id);
68
69
70     // If there is no guest in the table with the specified ID, print out the following message:
71     if(retrievedGuest == null)
72         System.out.println("No guest with the ID of " + id + " was found in the database...");
73
74     // Otherwise, display the retrieved guest
75     else
76         System.out.println("RETRIEVED GUEST: " + retrievedGuest);
77 }
```

```
private void getAllGuests(GuestDAO guestDAO) {
    System.out.println("\nRetrieving all guests from the database...");

    // Get all the guests from the table:
    List<Guest> retrievedGuests = guestDAO.getAllGuests();

    // Print out each guest's info:
    System.out.println("ALL GUESTS :");
    printGuests(retrievedGuests);
}

private void searchGuestsByName(GuestDAO guestDAO, String name) {
    System.out.println("\nRetrieving all guests that contain the string : "+name+"...");

    // Get the list of guests that contains the specified name argument:
    List<Guest> retrievedGuests = guestDAO.getGuestsByName(name);

    // Display all the guests that met the condition:
    System.out.println("RETRIEVED GUESTS: ");
    printGuests(retrievedGuests);
}
```



```
private void updateGuestRoomNumber(GuestDAO guestDAO, int id) {
    System.out.println("\nUpdate the room number for guest with ID #" + id + "...");

    // Search the guest by ID:
    Guest retrievedGuest = guestDAO.getGuestByID(id);

    // Update the guest's room number :
    retrievedGuest.setRoomNum(999);

    // Update the guest in the database:
    guestDAO.updateGuest(retrievedGuest);

    System.out.println("UPDATED GUEST : " + retrievedGuest);
}

private void removeGuest(GuestDAO guestDAO, int id) {
    System.out.println("\nRemove the guest with ID #" + id + "...");

    // Search the guest by ID:
    Guest retrievedGuest = guestDAO.getGuestByID(id);

    // Print out the guest to remove:
    System.out.println("GUEST TO REMOVE : " + retrievedGuest);

    // Execute the delete command on the database:
    guestDAO.removeGuest(id);

    // Print if the action was successfully performed:
    System.out.println("The guest was successfully deleted.");
}
```

To test out each of the CRUD operations in the commandLineRunner function, the following test Guest object was created:

```
@Bean
public CommandLineRunner commandLineRunner(GuestDAO guestDAO) {
    return runner -> {

        // Guest object to test out the CRUD operations:
        Guest testGuest = new Guest("Troy Baker", "troy_bake90@gmail.com", 12, "2024-11-03", "2024-11-07");

        // Add the new guest into the database:
```

The results obtained for each of the developed functions are presented below:

addGuest():

```
// Add the new guest into the database:  
addGuest(guestDAO, testGuest);|
```

Adding the new guest to the database....

GUEST INFO:
[id=0, name=Troy Baker, email=troy_bake90@gmail.com, roomNum=12, arrivalDate=2024-11-03, departureDate=2024-11-07]
Guest was successfully added!
NEW ID : 14

Result Grid						
		Filter Rows:		Edit:		Export/Import:
	id	name	email	room_number	arrival	departure
▶	1	John Doe	john.doe@example.com	1	2024-10-26	2024-10-29
	2	Jane Smith	jane.smith@example.com	2	2024-10-27	2024-10-30
	3	Peter Jones	peter.jones@hotmail.com	17	2024-11-28	2024-11-31
	4	Alice Johnson	alicee42@gmail.com	54	2024-10-05	2024-10-10
	5	Bob Williams	bobbywill_2@outlook.com	90	2024-09-24	2024-09-26
	6	Carol Brown	cbrownprof@hotmail.com	45	2024-12-21	2024-12-26
	14	Troy Baker	troy_bake90@gmail.com	12	2024-11-03	2024-11-07
✱	NULL	NULL	NULL	NULL	NULL	NULL

updateGuestRoomNumber():

```
// Let's update its room number  
updateGuestRoomNumber(guestDAO, testGuest.getId());
```

Update the room number for guest with ID #14...

UPDATED GUEST :

GUEST INFO:
[id=14, name=Troy Baker, email=troy_bake90@gmail.com, roomNum=999, arrivalDate=2024-11-03, departureDate=2024-11-07]

	14	Troy Baker	troy_bake90@gmail.com	999	2024-11-03	2024-11-07
✱	NULL	NULL	NULL	NULL	NULL	NULL

removeGuest():

```
// Now remove it!  
removeGuest(guestDAO, testGuest.getId());
```

```
Remove the guest with ID #14...  
GUEST TO REMOVE :  
GUEST INFO:  
[id=14, name=Troy Baker, email=troy_bake90@gmail.com, roomNum=999, arrivalDate=2024-11-03, departureDate=2024-11-07]  
The guest was successfully deleted.
```

	id	name	email	room_number	arrival	departure
▶	1	John Doe	john.doe@example.com	1	2024-10-26	2024-10-29
	2	Jane Smith	jane.smith@example.com	2	2024-10-27	2024-10-30
	3	Peter Jones	peter.jones@hotmail.com	17	2024-11-28	2024-11-31
	4	Alice Johnson	alicee42@gmail.com	54	2024-10-05	2024-10-10
	5	Bob Williams	bobbywill_2@outlook.com	90	2024-09-24	2024-09-26
	6	Carol Brown	cbrownprof@hotmail.com	45	2024-12-21	2024-12-26
⌵	NULL	NULL	NULL	NULL	NULL	NULL

searchGuest() [Unavailable]:

```
// try searching it...  
searchGuest(guestDAO, testGuest.getId());
```

```
Retrieving guest info by ID 14...  
No guest with the ID of 14 was found in the database...
```

searchGuest() [Available]:

```
// What about the first guest...  
searchGuest(guestDAO, 1);
```

```
Retrieving guest info by ID 1...  
RETRIEVED GUEST:  
GUEST INFO:  
[id=1, name=John Doe, email=john.doe@example.com, roomNum=1, arrivalDate=2024-10-26, departureDate=2024-10-29]
```

searchGuestsByName():

```
// Search the hotel guests that contain the string "John"  
searchGuestsByName(guestDAO, "John");
```

```
Retrieving all guests that contain the string : John...  
RETRIEVED GUESTS:  
  
GUEST INFO:  
[id=1, name=John Doe, email=john.doe@example.com, roomNum=1, arrivalDate=2024-10-26, departureDate=2024-10-29]  
  
GUEST INFO:  
[id=4, name=Alice Johnson, email=alicee42@gmail.com, roomNum=54, arrivalDate=2024-10-05, departureDate=2024-10-10]
```

getAllGuests():

```
// Retrieve all guests of the hotel!  
getAllGuests(guestDAO);
```

```
Retrieving all guests from the database...  
ALL GUESTS :  
  
GUEST INFO:  
[id=1, name=John Doe, email=john.doe@example.com, roomNum=1, arrivalDate=2024-10-26, departureDate=2024-10-29]  
  
GUEST INFO:  
[id=2, name=Jane Smith, email=jane.smith@example.com, roomNum=2, arrivalDate=2024-10-27, departureDate=2024-10-30]  
  
GUEST INFO:  
[id=3, name=Peter Jones, email=peter.jones@hotmail.com, roomNum=17, arrivalDate=2024-11-28, departureDate=2024-11-31]  
  
GUEST INFO:  
[id=4, name=Alice Johnson, email=alicee42@gmail.com, roomNum=54, arrivalDate=2024-10-05, departureDate=2024-10-10]  
  
GUEST INFO:  
[id=5, name=Bob Williams, email=bobbywill_2@outlook.com, roomNum=90, arrivalDate=2024-09-24, departureDate=2024-09-26]  
  
GUEST INFO:  
[id=6, name=Carol Brown, email=cbrownprof@hotmail.com, roomNum=45, arrivalDate=2024-12-21, departureDate=2024-12-26]
```