Spring Boot is built on the top of the spring and contains all the features of spring. And is becoming a favorite of developers these days because of its rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the configuration and setup. Spring Boot is a microservice-based framework and making a production-ready application in it takes very little time. In this article, we will discuss Hello World Example using the spring boot. We will discuss two ways to print Hello World using Spring Boot.

- With the help of the CommandRunner interface of SpringBoot
- With the controller class in the SpringBoot

First, initialize the project on our machine. Spring Initializr is a web-based tool using which we can easily generate the structure of the Spring Boot project. It also provides various different features for the projects expressed in a metadata model. This model allows us to configure the list of dependencies that are supported by JVM. Here, we will create the structure of an application using a spring initializer and then use an IDE to create a sample GET route. Therefore, to do this, the following steps are followed sequentially as follows.

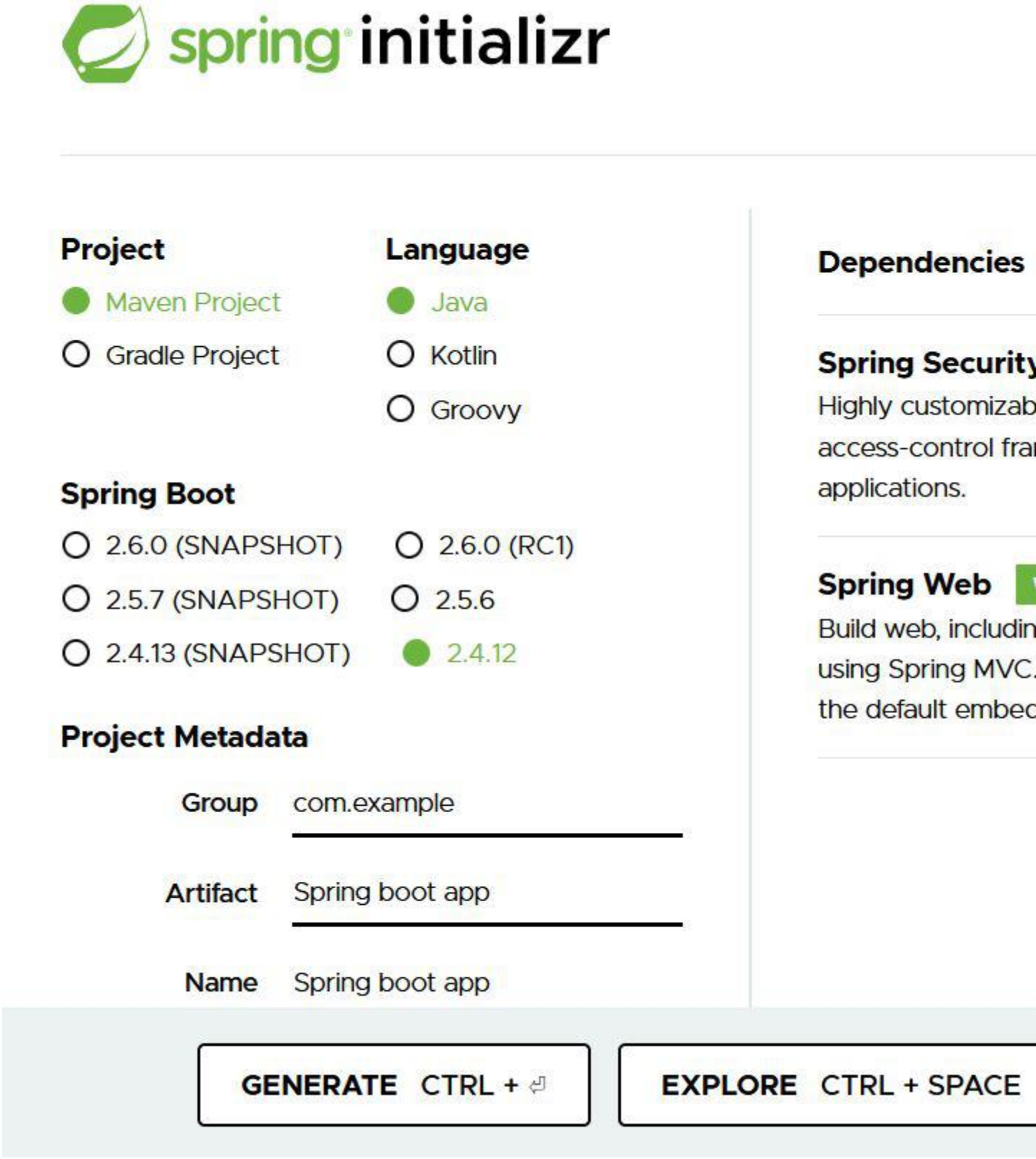
Step by Step Implementation

Step 1: Go to Spring Initializr

Fill in the details as per the requirements. For this application:

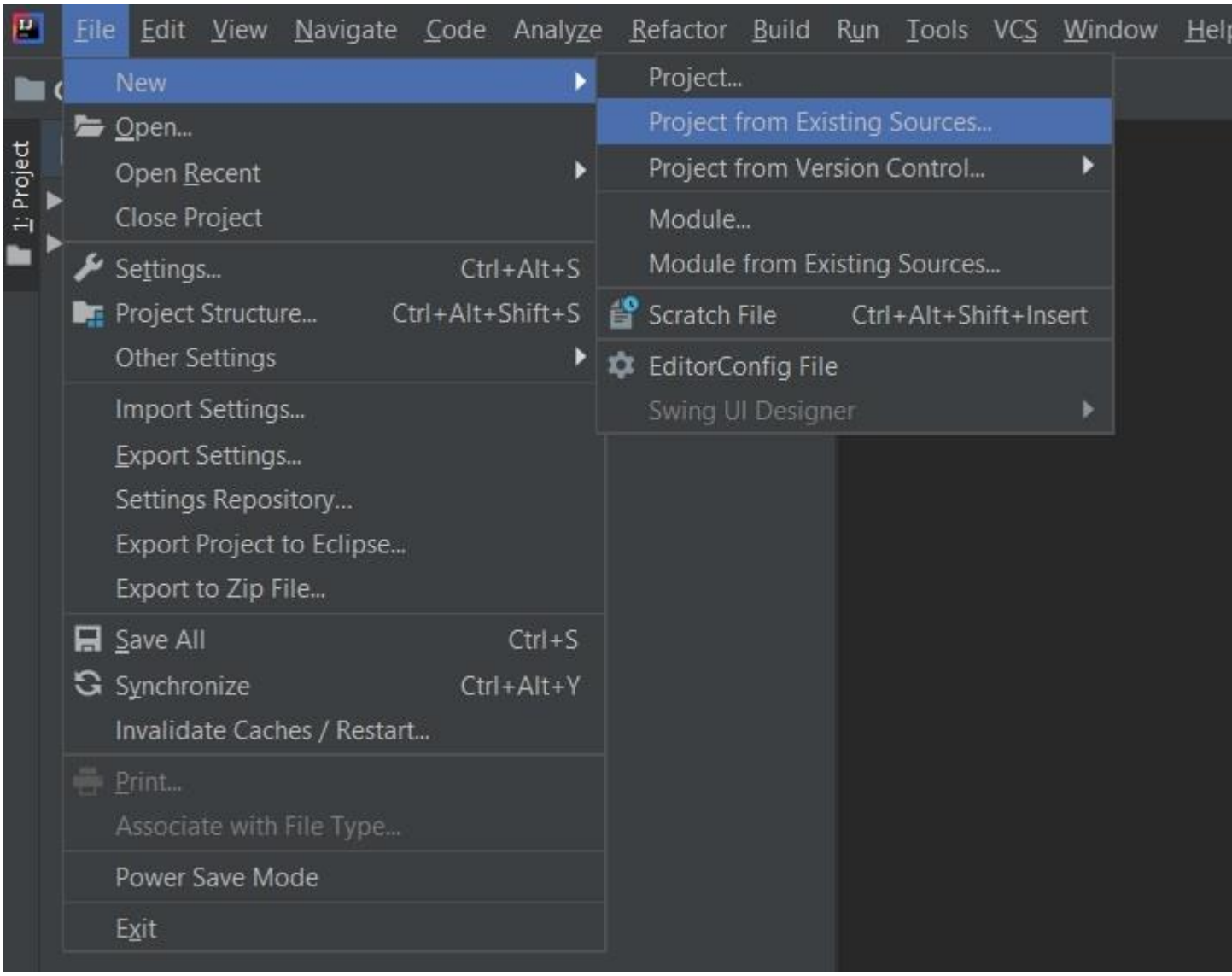
```
Project: Maven

Language: Java

Spring Boot: 2.2.8

Packaging: JAR

Java: 8

Dependencies: Spring Web
```

Step 2: Click on Generate which will download the starter project



Step 3: Extract the zip file. Now open a suitable IDE and then go to File > New > Project from existing sources > Spring-boot-app and select pom.xml. Click on

import changes on prompt and wait for the project to sync as pictorially depicted below as follows:



***Note**: In the Import Project for Maven window, make sure you choose the same version of JDK which you selected while creating the project.*

Method 1: With the help of the CommandRunner interface of SpringBoot

Step 4: Go to src > main > java > com.gfg.Spring.boot.app, Below is the code for the **SpringBootAppApplication.java** file.

- Java

```java
@SpringBootApplication

// Main class

// Implementing CommandLineRunner interface
public class SpringBootAppApplication implements CommandLineRunner
{
    // Method 1

    // run() method for springBootApplication to execute
    @Override
    public void run(String args[]) throws Exception
    {
        // Print statement when method is called
        System.out.println("HEllo world");
    }

    // Method 2

    // Main driver method
    public static void main(String[] args)
    {
        // Calling run() method to execute
        // SpringBootApplication by

        // invoking run() inside main() method
        SpringApplication.run(SpringBootAppApplication.class, args);
    }
}
```

This application is now ready to run.

Step 6: Run the SpringBootAppApplication class and wait for the Tomcat server to start where the default port is already set.
Tip*: The default port of the Tomcat server is 8080 and can be changed in the application.properties file.*

Output: Generated on terminal/CMD

```
2021-10-26 10:21:31.027  INFO 24312 --- [           main] JPA.demo.DemoA
HEllo world
```

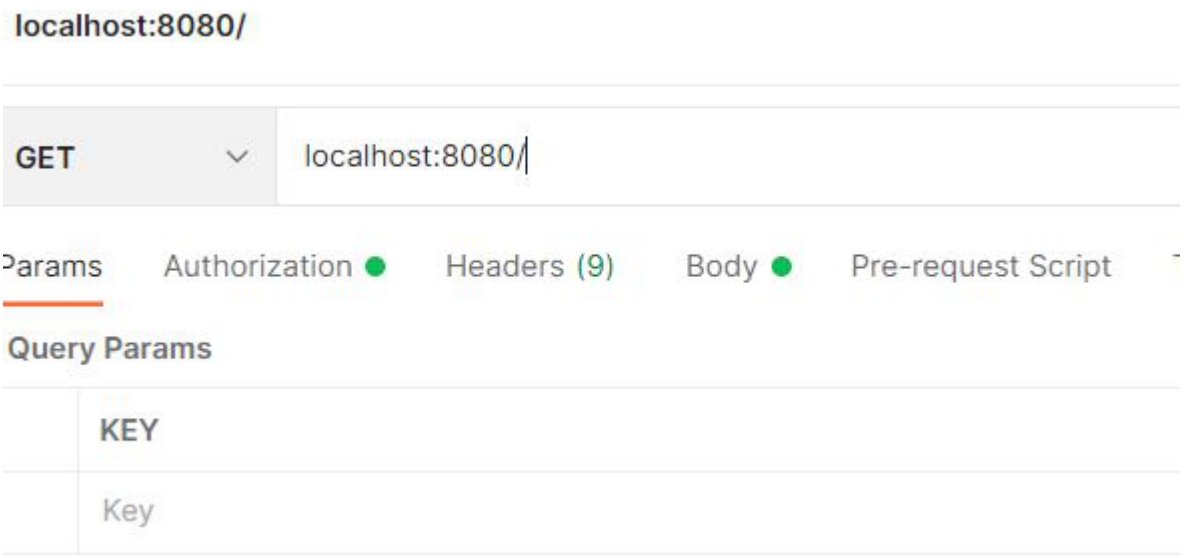
Method 2: With the controller class in the SpringBoot

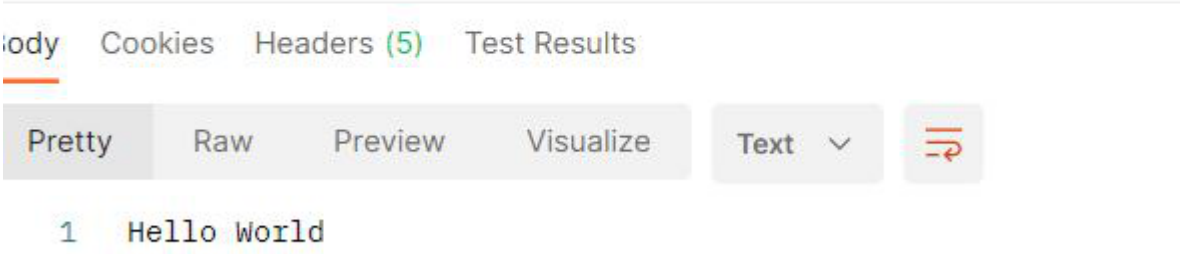
Go to src > main > java > com.gfg.Spring.boot.app and create a controller class. Below is the code for the **controller.java** file.

- Java

```java
@RestController
public class controller {
    @GetMapping("/")
    String return1(){
        return "Hello World";
    }
}
```

This controller helps to handle all the incoming requests from the client-side. Now we will use the PostMan and call the get API of the Spring boot application.





Spring Boot – REST Example

We all know in today's world, most web app follows the client-server architecture. The app itself is the client or frontend part under the hood it needs to call the server or the backend to get or save the data this communication happens using HTTP protocol the same protocol is the power of the web. On the server, we expose the bunch of services that are accessible via the HTTP protocol. The client can then directly call the services by sending the HTTP request.

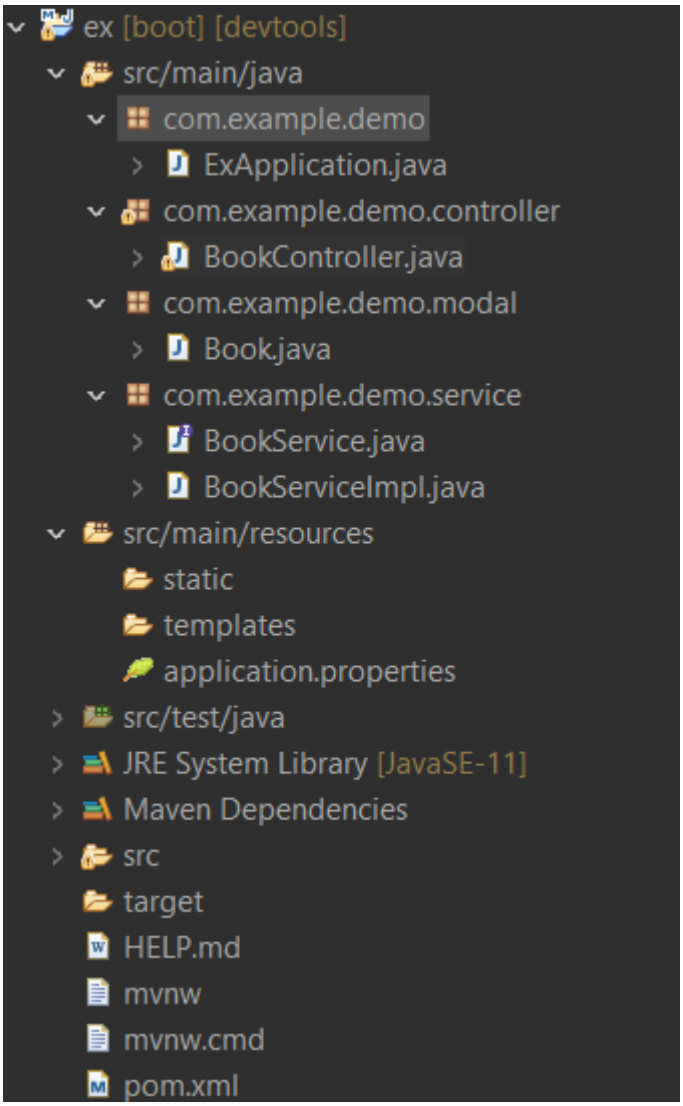
Now, this is where **REST** comes into the picture. Rest stands for **R**epresentation **S**tate **T**ransfer it is basically a convention to building these HTTP services. So we use a simple HTTP protocol principle to provide support to **CREATE**, **READ**, **UPDATE** & **DELETE** data. We refer to these operations all together called **CRUD** operations. In short, It is a set of rules that developers follow when they create their API. One of these rules states that you should be able to get a piece of data (called a resource) when you link to a specific URL. Let's implement an RSET application and understand the REST approach by creating an example where we simply return the Book data in the form of **JSON**.

Rest with Example

Set up the spring project:
So first we will set up the spring project in STS(Spring tool suite) IDE. Whose instructions have been given below

- Click File -> New -> Project -> Select Spring Starter Project -> Click Next.
- A New Dialog box will open where you will provide the project-related information like project name, Java version, Maven version, and so on.
- After that select required maven dependencies like **Spring Web**, **Spring Boot DevTools** (Provides fast application restarts, LiveReload, and configurations for enhanced development experience.)
- Click Finish.

Project Structure:



pom.xml:

- 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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.1</version>
        <relativePath/> <!-- lookup parent from repository -->
```

```
    </parent>
    <groupId>com.example</groupId>
    <artifactId>ex</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>ex</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>

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

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>

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

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

</project>
```

Note: *No need to add anything to the* ***application.properties*** *because we didn't use a database.*

POJO (Plain old java object) class:

- Java

```java
package com.example.demo.modal;

public class Book {

    private long id;
    private String name;
    private String title;

    public Book() {
        super();
    }
    public Book(long id, String name, String title) {
        super();
        this.id = id;
        this.name = name;
        this.title = title;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
```

```
    }

}
```

Service interface and Service implementation class

Here we have created an interface called **BookService** which contains all the service methods that our application is going to provide to the user.
And **BookServiceImpl** class that implements the BookService interface.

- Java
- Java

```
package com.example.demo.service;

import java.util.HashSet;
import com.example.demo.modal.Book;

public interface BookService {
     HashSet<Book> findAllBook();
     Book findBookByID(long id);
     void addBook(Book b);
     void deleteAllData();
}
```

Rest Controller:

Here is the **ExController** class where we are exposing all the APIs which we have created.
API's list

- **http://localhost:8080/**
  - To save the data
- **http://localhost:8080/findbyid/2**
  - Find Book by id
- **http://localhost:8080/findall**
  - Find all books
- **http://localhost:8080/delete**
  - Delete all books

- Java

```
package com.example.demo.controller;

import java.util.ArrayList;
```

```java
import java.util.HashSet;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.modal.Book;

import com.example.demo.service.BookServiceImpl;

@RestController
public class BookController {

    @Autowired
    BookServiceImpl bookServiceImpl;

    @PostMapping("/")
    public void addBook(@RequestBody Book book) {
        bookServiceImpl.addBook(book);
    }

    @GetMapping("/findall")
    public HashSet<Book> getAllBook() {
        return bookServiceImpl.findAllBook();
    }

    @GetMapping("/findbyid/{id}")
    public Book geBookById(@PathVariable long id) {
        return bookServiceImpl.findBookByID(id);
    }

    @DeleteMapping("/delete")
    public void deleteBook() {
        bookServiceImpl.deleteAllData();
    }

}
```

ExApplication.java

To run the application.

- Java

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ExApplication {

    public static void main(String[] args) {
        SpringApplication.run(ExApplication.class, args);
    }

}
```

Spring Data JPA

Spring Data JPA or JPA stands for **Java Persistence API,** so before looking into that, we must know about **ORM (Object Relation Mapping)**. So Object relation mapping is simply the process of persisting any java object directly into a database table. Usually, the name of the object being persisted becomes the name of the table, and each field within that object becomes a column. With the table setup set up, each row corresponds to a record in the application. Hibernate is one example of ORM. In short, JPA is the interface while hibernate is the implementation.
The java persistence API provides a specification for persisting, reading, and managing data from your java object to your relational tables in the database. JPA specifies the set of rules and guidelines for developing interfaces that follow standards. Straight to the point: JPA is just guidelines to implement ORM and there is no underlying code for the implementation. Spring Data JPA is part of the spring framework. The goal of spring data repository abstraction is to

significantly reduce the amount of boilerplate code required to implement a data access layer for various persistence stores. Spring Data JPA is not a JPA provider, it is a library/framework that adds an extra layer of abstraction on the top of our JPA provider line Hibernate.

Configure Spring Data JPA in Spring Application with Example

Requirements: STS IDE, MySQL workbench, Java 8+
Create a spring boot project in STS. Give project name & select add required dependencies(Spring JPA, MySQL driver, Spring web) shown in attached **pom.xml**.

- 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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>ex</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>ex</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>11</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>
```

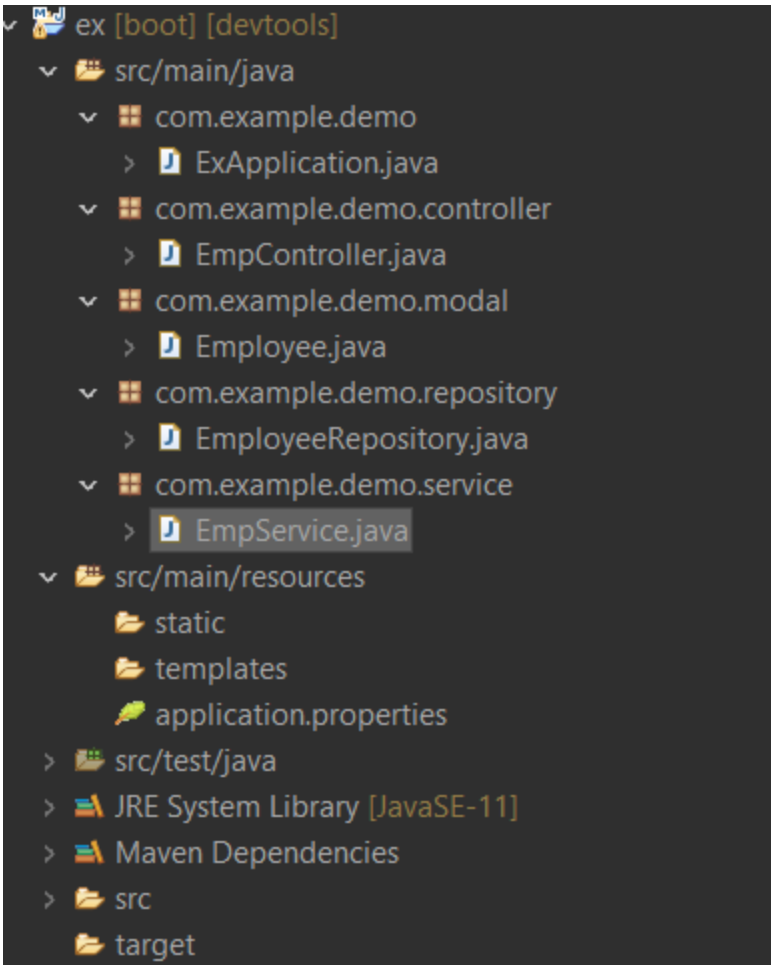
```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

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

OR you can also add these dependencies while creating the project.

application.properties file:

```
spring.datasource.url=jdbc:mysql://localhost:3306/emp

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.database-platform = org.hibernate.dialect.MySQL5Dialect

spring.jpa.generate-ddl=true

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

Project Structure:



Modal layer:
Create a simple POJO(Plain old java class) with some JPA annotation.

- @**Entity**: This annotation defines that a class can be mapped to a table
- @**Id:** This annotation specifies the primary key of the entity.
- @**GeneratedValue:** This annotation is used to specify the primary key generation strategy to use. i.e. Instructs database to generate a value for this field automatically. If the strategy is not specified by default AUTO will be used.

- Java

```java
package com.example.demo.modal;

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

```java
import javax.persistence.Id;

// @Entity annotation defines that a
// class can be mapped to a table
@Entity
public class Employee {

    // @ID This annotation specifies
    // the primary key of the entity.
    @Id

    // @GeneratedValue This annotation
    // is used to specify the primary
    // key generation strategy to use
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    private String name;
    private String city;

    public Employee() {
        super();
    }
    public Employee(String name, String city) {
        super();
        this.name = name;
        this.city = city;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
```

```
}
```

DAO(Data access object) layer:

- **@Repository:** The @Repository annotation is a marker for any class that fulfills the role or stereotype of a repository (also known as Data Access Object or DAO).
- **JpaRepository<Employee, Long>** JpaRepository is a JPA-specific extension of the Repository. It contains the full API of CrudRepository and PagingAndSortingRepository. So it contains API for basic CRUD operations and also API for pagination and sorting. Here we enable database operations for Employees.

- Java

```java
package com.example.demo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.example.demo.modal.Employee;

// @Repository is a Spring annotation that
// indicates that the decorated class is a repository.
@Repository
public interface EmployeeRepository extends JpaRepository<Employee,
Long>{
    ArrayList<Employee> findAllEmployee();
}
```

Service layer:

- Java

```java
package com.example.demo.service;
import java.util.ArrayList;
import com.example.demo.modal.Employee;
public interface EmpService {
    ArrayList<Employee> findAllEmployee();
    Employee findAllEmployeeByID(long id);
    void addEmployee();
    void deleteAllData();
}
```

@Service: This annotation is used with classes that provide some business functionalities. Spring context will autodetect these classes when annotation-based configuration and classpath scanning is used. Here JPA repository has lots of predefined generic methods to perform the database operation some are used in the below code.

- Java

```java
package com.example.demo.service;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.example.demo.modal.Employee;
import com.example.demo.repository.EmployeeRepository;


//  @Service marks a Java class that performs some service,
//  such as executing business logic, performing
//  calculations, and calling external APIs.
@Service
public class EmpServiceImpl implements EmpService {
    @Autowired
    EmployeeRepository employeeRepository;

    @Override
    public ArrayList<Employee> findAllEmployee() {
        return (ArrayList<Employee>) employeeRepository.findAll();
    }

    @Override
    public Employee findAllEmployeeByID(long id) {
        Optional<Employee> opt = employeeRepository.findById(id);
        if (opt.isPresent())
            return opt.get();
        else
            return null;
    }

    @Override
    public void addEmployee() {
        ArrayList<Employee> emp = new ArrayList<Employee>();
        emp.add(new Employee("Lucknow", "Shubham"));
```

```
        emp.add(new Employee("Delhi", "Puneet"));
        emp.add(new Employee("Pune", "Abhay"));
        emp.add(new Employee("Noida", "Anurag"));
        for (Employee employee : emp) {
            employeeRepository.save(employee);
        }
    }

    @Override
    public void deleteAllData() {
        employeeRepository.deleteAll();
    }
}
```

Controller layer:

- **@RestController:** This is a Spring annotation that is used to build REST API in a declarative way. RestController annotation is applied to a class to mark it as a request handler, and Spring will do the building and provide the RESTful web service at runtime.
- **@Autowired:** This annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property, or methods with arbitrary names and/or multiple arguments.
- **@PostMapping:** This annotation maps HTTP POST requests onto specific handler methods. It is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod. POST)
- **@GetMapping:** This annotation is a specialized version of @RequestMapping annotation that acts as a shortcut for @RequestMapping(method = RequestMethod. GET). The @GetMapping annotated methods in the @Controller annotated classes handle the HTTP GET requests matched with the given URI expression.
- **@DeleteMapping:** This annotation maps HTTP DELETE requests onto specific handler methods. It is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod. DELETE)

- Java

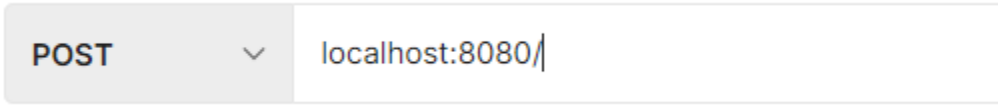
```
package com.example.demo.controller;
import java.util.ArrayList;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
```

```java
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;
import com.example.demo.modal.Employee;
import com.example.demo.service.EmpServiceImpl;

@RestController
public class EmpController {

    @Autowired
    EmpServiceImpl empServiceImpl;

    @PostMapping("/")
    public void add() {
        empServiceImpl.addEmployee();
    }

    @GetMapping("/findall")
    public ArrayList<Employee> getAllEmployee() {
        return empServiceImpl.findAllEmployee();
    }

    @GetMapping("/findbyid/{id}")
    public Employee getEmployeeUsingId(@PathVariable long id) {
        return empServiceImpl.findAllEmployeeByID(id);
    }

    @DeleteMapping("/delete")
    public void delete() {
        empServiceImpl.deleteAllData();
    }
}
```

Display output using JPA in postman:

Open postman and hit the listed API one by one.

Save employee data



Find all employee list

GET localhost:8080/findall


```
[
    {
        "id": 11,
        "name": "Pune",
        "city": "Abhay"
    },
    {
        "id": 10,
        "name": "Delhi",
        "city": "Puneet"
    },
    {
        "id": 9,
        "name": "Lucknow",
        "city": "Shubham"
    },
    {
        "id": 12,
        "name": "Noida",
        "city": "Anurag"
    }
]
```


Find an employee by id