

Table of Contents

Step 1: Create Spring Boot Project	2
Step 2: Understand Spring Boot as opinionated.....	3
Step 3: Understand Spring object creation	3
Step 4: Understand DI (@Autowired) , @RestController and @RequestMapping	6
Step 5: @PathVariable	8
Step 6: @RequestParam	10
Step 7: @PostMapping	11
Step 8: @PutMapping.....	13
Step 9: @DeleteMapping	14
Modify the code to fetch data from database	16
Step 10: Update properties file for Db connection parameters.....	16
Step 11: Map java class to database table so that it becomes “DATABASE MANAGED ENTITY”.....	16
Step 12: JpaRepository.....	17
Step 13: service layer	17
Step 14: Appendix.....	18

Step 1: Create Spring Boot Project

1. Bitbucket URL: <https://bitbucket.org/shalini-ws/springbootrestrepo/src/main/>
2. Please download **POSTMAN** on your VM.
<https://www.postman.com/downloads/>
3. Open below url on the browser:
<https://start.spring.io/>
4. Create spring boot project as shown on the screen below:

The screenshot shows the Spring Boot project generator interface. Red boxes highlight the following elements:

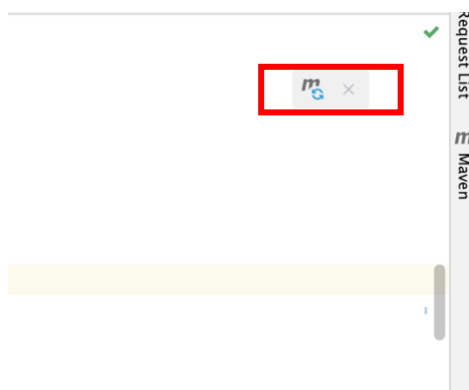
- Project:** Maven
- Language:** Java
- Spring Boot:** 3.3.2
- Dependencies:** Spring Web
- Project Metadata:**
 - Group: com.training
 - Artifact: SpringBootRETRapo
 - Name: SpringBootRETRapo
 - Description: Demo project for Spring Boot
 - Package name: com.training.SpringBootRETRapo
- Packaging:** Jar
- Java:** 17

Buttons at the bottom: GENERATE, EXPLORE, SHARE...

5. Click on Generate, it will download the zip. Extract and open project using IntelliJ
6. Once opened on INTELLIJ, please add below dependency in pom.xml: What this is for will discuss at later stage.

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.0.3</version>
</dependency>
```

After adding the dependency, you will get an option to reload in pom.xml as follows: Please click on that:



7. The embedded tomcat with spring boot web includes a light weight server which is the tomcat core and is capable of processing HTTP requests and send JSON as a response
8. **Following the package structure is very important with spring boot for it to follow the default configurations.**

Step 2: Understand Spring Boot as opinionated

1. View the pom.xml file that has spring boot starter parent.

It is a special starter project that provides default configurations for our application and a complete dependency tree to quickly build our Spring Boot project.

It also provides default configurations for Maven plugins, such as maven-failsafe-plugin, maven-jar-plugin, maven-surefire-plugin, and maven-war-plugin.

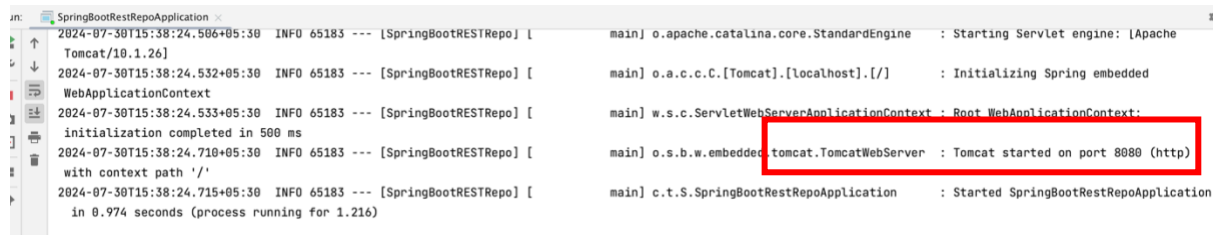
Beyond that, it also inherits dependency management from spring-boot-dependencies, which is the parent to the spring-boot-starter-parent.

2. @SpringBootApplication on the class with the main method:

This annotation is used to enable three features, that is:

- a. @EnableAutoConfiguration: enable Spring Boot's auto-configuration mechanism
- b. @ComponentScan: enable @Component scan on the package where the application is located.
- c. @Configuration: allow to register extra beans in the context or import additional configuration classes

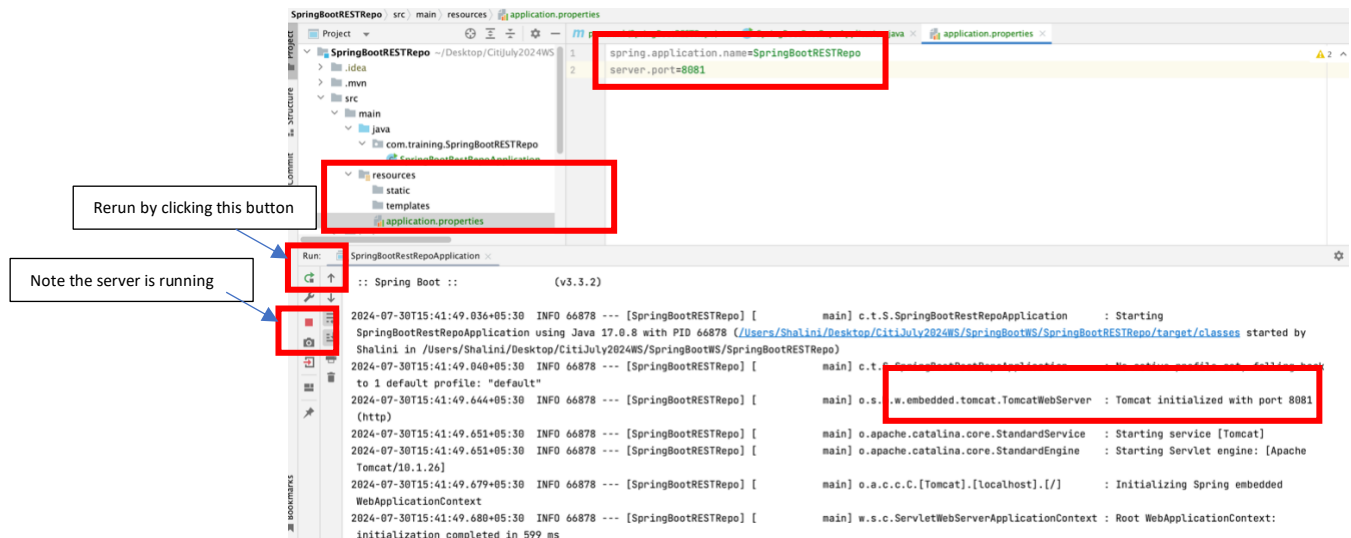
3. Run the main method and observe the console.



```
2024-07-30T15:38:24.506+05:30 INFO 65183 --- [SpringBootREStRepo] [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.26]
2024-07-30T15:38:24.532+05:30 INFO 65183 --- [SpringBootREStRepo] [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-07-30T15:38:24.533+05:30 INFO 65183 --- [SpringBootREStRepo] [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 500 ms
2024-07-30T15:38:24.710+05:30 INFO 65183 --- [SpringBootREStRepo] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http)
2024-07-30T15:38:24.715+05:30 INFO 65183 --- [SpringBootREStRepo] [main] c.t.S.SpringBootRestRepoApplication : Started SpringBootRestRepoApplication in 0.974 seconds (process running for 1.216)
```

- a. Notice tomcat is running on port 8080
- b. Tomcat server by default runs on port 8080. To change the port add below in application.properties file
server.port=8081

Rerun the project and you will see the output as below



```
spring.application.name=SpringBootREStRepo
server.port=8081
```

```
2024-07-30T15:41:49.036+05:30 INFO 66878 --- [SpringBootREStRepo] [main] c.t.S.SpringBootRestRepoApplication : Starting SpringBootRestRepoApplication using Java 17.0.8 with PID 66878 (/Users/Shalini/Desktop/CitiJuly2024WS/SpringBootWS/SpringBootREStRepo/target/classes started by ShaLin in /Users/Shalini/Desktop/CitiJuly2024WS/SpringBootWS/SpringBootREStRepo)
2024-07-30T15:41:49.040+05:30 INFO 66878 --- [SpringBootREStRepo] [main] c.t.S.SpringBootRestRepoApplication : No active profile found, using default 1 to 1 default profile: "default"
2024-07-30T15:41:49.644+05:30 INFO 66878 --- [SpringBootREStRepo] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8081 (http)
2024-07-30T15:41:49.651+05:30 INFO 66878 --- [SpringBootREStRepo] [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-07-30T15:41:49.651+05:30 INFO 66878 --- [SpringBootREStRepo] [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.26]
2024-07-30T15:41:49.679+05:30 INFO 66878 --- [SpringBootREStRepo] [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-07-30T15:41:49.680+05:30 INFO 66878 --- [SpringBootREStRepo] [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 599 ms
```

Step 3: Understand Spring object creation

1. Create a class Book as follows:

```
package com.training.SpringBootREStRepo.entity;
```

```
public class Book {
```

```
    private int bookid;
```

```
    private String title;
```

```
    private String author;
```

```
    private String desc;
```

```
    private double price;
```

```

public Book() {

}

public Book(int bookid, String title, String author, String desc, double price) {
    this.bookid = bookid;
    this.title = title;
    this.author = author;
    this.desc = desc;
    this.price = price;
}

public Book(String title, String author, String desc, double price) {
    this.title = title;
    this.author = author;
    this.desc = desc;
    this.price = price;
}

@Override
public String toString() {
    return "Book{" +
        "bookid=" + bookid +
        ", title=" + title + "\" +
        ", author=" + author + "\" +
        ", desc=" + desc + "\" +
        ", price=" + price +
        '}'';
}

public int getBookid() {
    return bookid;
}

public void setBookid(int bookid) {
    this.bookid = bookid;
}

public String getTitle() {
    return title;
}

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

public String getAuthor() {
    return author;
}

public void setAuthor(String author) {
    this.author = author;
}

public String getDesc() {
    return desc;
}

public void setDesc(String desc) {
    this.desc = desc;
}

public double getPrice() {
    return price;
}

public void setPrice(double price) {
    this.price = price;
}
}

```

- 2 . Create service class as follows that will provide with book details:

```

package com.training.SpringBootRESTRRepo.service;

```

```

import com.training.SpringBootRESTRepo.entity.Book;
import java.util.ArrayList;
import java.util.List;

public class BookService {

    private List<Book> bookList;

    public BookService() {
        System.out.println("Book service default constructor");
        bookList = new ArrayList<>();
        bookList.add(
            new Book(1, "Core Java", "Hotsmann", "Learn java fundamentals", 130.0));
        bookList.add(
            new Book(2, "HTML", "Kelly", "Learn html for UI", 230.0));
        bookList.add(
            new Book(3, "python", "ryan", "Learn python fundamentals", 130.0));
        bookList.add(
            new Book(4, "css", "kelly", "Learn css for designing webpage", 130.0));
    }

    public long getTotalBookCount(){
        return bookList.size();
    }
    public List<Book> getAllBooks(){
        return bookList;
    }
    public Book addNewBook(Book book){

        for (Book ob : bookList){
            if(ob.getBookid() == book.getBookid())
                throw new RuntimeException("Book with id "+book.getBookid()+" already exists");
        }
        Book lastBook = bookList.get(bookList.size()-1);
        int id = lastBook.getBookid() + 1;
        book.setBookid(id);
        bookList.add(book);
        return book;
    }

    public Book updateBook(Book book){
        for (int i=0;i<bookList.size();i++){
            if(bookList.get(i).getBookid() == book.getBookid()) {
                bookList.set(i, book);
                return book;
            }
        }
        throw new RuntimeException("Book with id "+book.getBookid()+" does not exist");
    }
    public boolean deleteBook(int id){
        for (int i=0;i<bookList.size();i++){
            if(bookList.get(i).getBookid() == id) {
                bookList.remove(i);
                return true;
            }
        }
        throw new RuntimeException("Book with id "+id+" does not exist");
    }
    public List<Book> getBooksByAuthor(String author){
        List<Book> booksByAuthor = new ArrayList<>();
        for (Book ob : bookList){
            if(ob.getAuthor().equalsIgnoreCase(author))
                booksByAuthor.add(ob);
        }
        return booksByAuthor;
    }
    public Book getBookById(int id){

        for (Book ob : bookList) {
            if (ob.getBookid() == id)
                return ob;
        }
        throw new RuntimeException("Book with id "+id+" does not exists");
    }
}

```

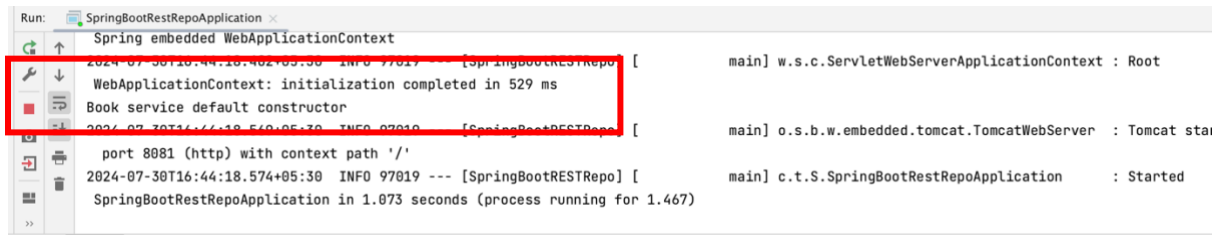
```
}
}
```

- Using spring specific annotations will automatically load and instantiate the classes. Since we need BookService class object just annotate as follows:

```
@Service
public class BookService {

    private List<Book> bookList;
    // other parts are same
}
```

- The classes that are loaded and instantiated by spring are called as “**SPRING MANAGED BEANS**”
- Just Rerun the server and should see output from the constructor on the console



Step 4: Understand DI (@Autowired) , @RestController and @RequestMapping

- To inform spring that a class is exposing data over HTTP protocol, we need to use RestController annotation on the class.
- Create a class BookRestController as follows:

```
package com.training.SpringBootRESTRepo.rest;

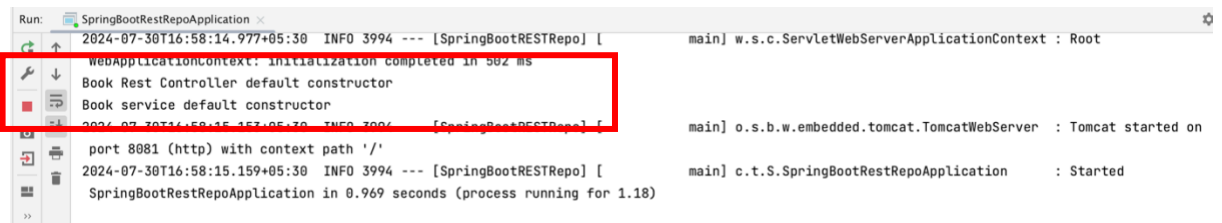
import org.springframework.web.bind.annotation.RestController;

@RestController
public class BookRestController {

    public BookRestController() {
        System.out.println("Book Rest Controller default constructor");
    }

}
```

- Rerun the application and should see the output from Rest Controller class as follows:



- This class needs reference of BookService class to get the data. Update BookRestController as follows:

```
@RestController
public class BookRestController {

    private BookService bookService;
    // other parts are same

    public List<Book> getBooks(){
        return bookService.getAllBooks();
    }

}
```

```
}
}
```

5. Normally we provide dependencies as follows:
 BookService bs = new BookService();
 BookRestController ob = new BookRestController(bs);
6. Since BookService is a spring managed bean, we need to tell spring to inject this dependency. Update the code as follows:

```
@RestController
public class BookRestController {

    @Autowired
    private BookService bookService;
    // other parts are same
}
```

@Autowired annotation tells spring about the dependency and it looks for the bean within its context and if found inject it.

7. To expose data annotate the class with @RequestMapping to specify the exposed endpoint URI and update method with @GetMapping annotation to fetch data.

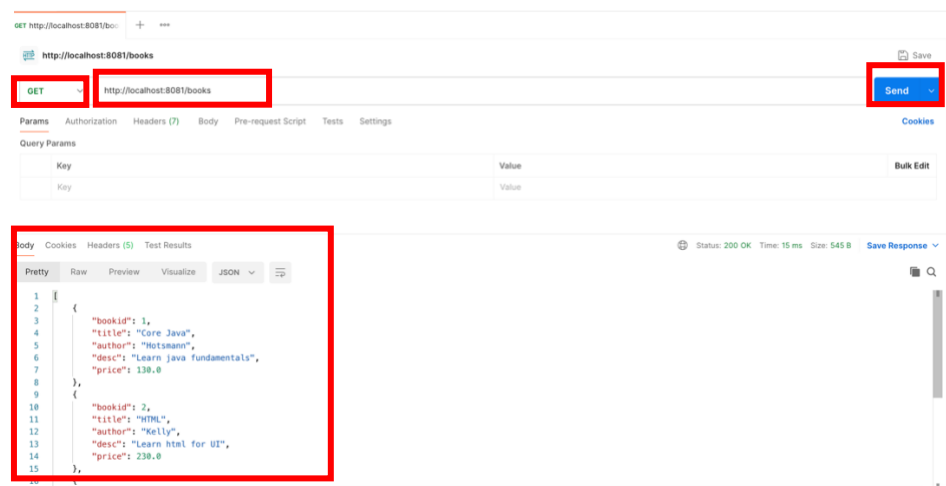
```
@RestController
@RequestMapping("/books")
public class BookRestController {
    // other parts are same

    @GetMapping
    public List<Book> getBooks(){
        return bookService.getAllBooks();
    }
}
```

MAKE SURE TO RESTART THE SERVER

It will be available at <http://localhost:8080/books>.

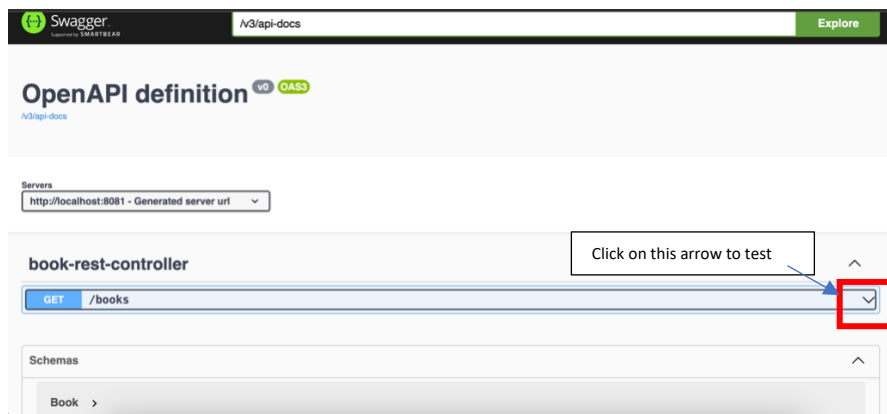
8. To check if it is working open POSTMAN and check if REST endpoint is working as follows:
 - a. Type url in the address bar
 - b. Make sure to select GET from dropdown
 - c. Click on Send
 - d. Should see the output as follows:



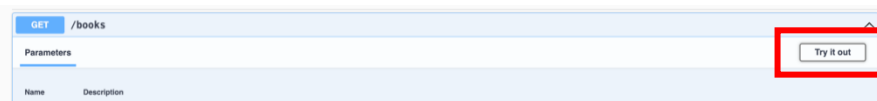
9. Alternatively it can be checked using swagger. Swagger is used for REST endpoints documentation and testing. Go on to browser and type in the below url:

<http://localhost:8081/swagger-ui/index.html#/>

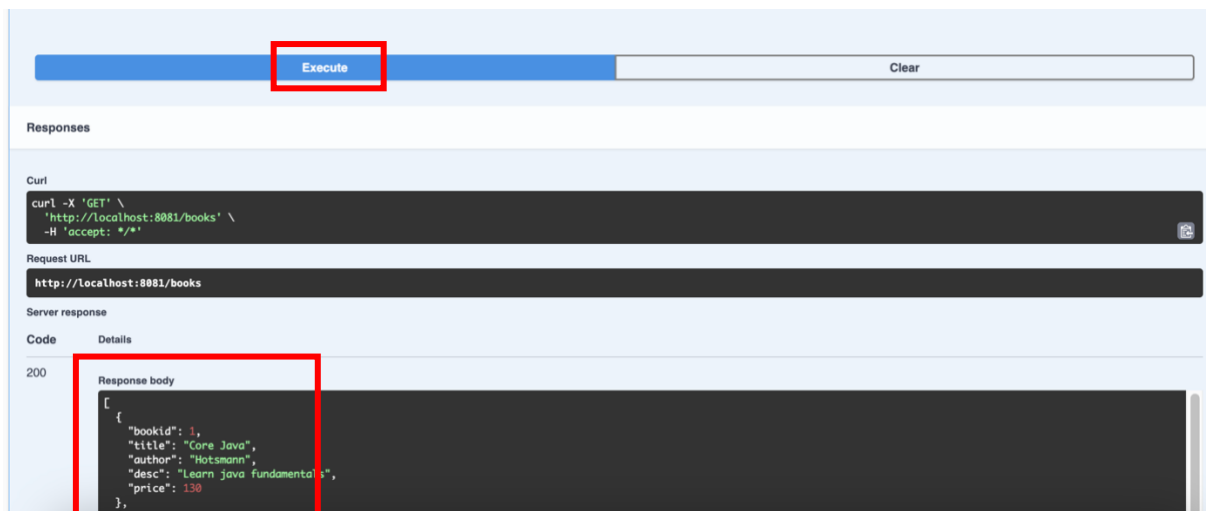
Should see the output as below:



When you click on the arrow, you get Try it out. Click on that will have an option to Execute.



Click on Execute and should see the JSON response as shown below:



Step 5: @PathVariable

1. Update Controller and add below method to return book by id as follows:

```
public Book getBookById(int id){
    return bookService.getBookById(id);
}
```

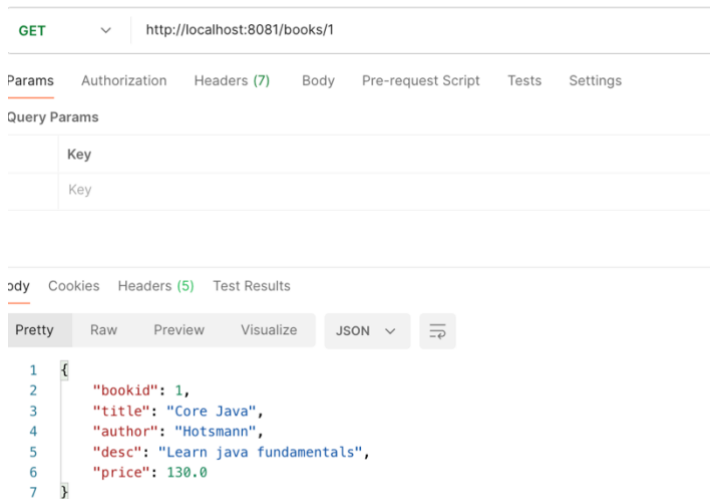
2. To make this method available at REST API endpoint and return book for a specific id, update the method as follows:

```
@GetMapping("/{id}")
public Book getBookById(@PathVariable int id){
    return bookService.getBookById(id);
}
```

MAKE SURE TO RESTART THE SERVER

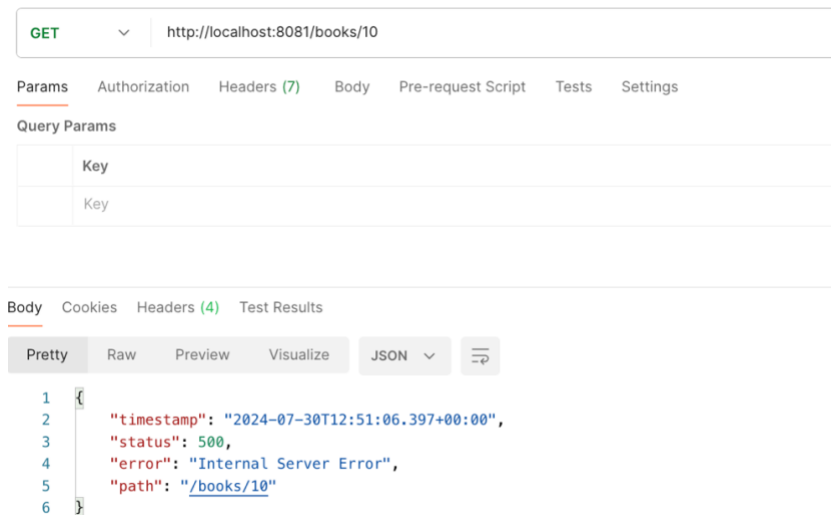
To access type in postman url : <http://localhost:8080/books/1>

Make sure GET is selected in the dropdown and click on send. Should see the details of book by id 1.

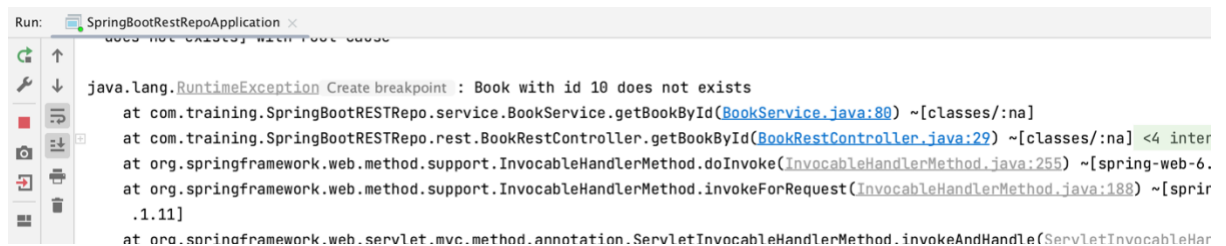


{ } -> is the placeholder for the value [1] passed in the url.
{id} is mapped to method parameter id using @PathVariable annotation.

- Now try to access a book that does not exists: You should get following screen on postman:



And on IDE console, you will see the exception:



- Displaying internal server error is not a good practice. Let's modify the code to handle the exception and return an appropriate response along with respective status code. Spring provides with ResponseEntity class to wrap the data and any extra information to be returned .

```

@GetMapping("/{id}")
public ResponseEntity<Object> getBookById(@PathVariable int id){
    Map<String, Object> map = new HashMap<>();
    try {
        map.put(AppConstants.STATUS, Status.SUCCESS);
        map.put("book",bookService.getBookById(id) );
        return ResponseEntity.ok(map);
    }
}
```

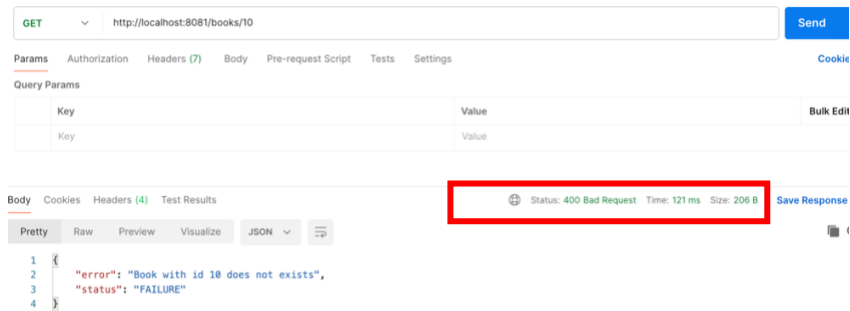
```

catch (RuntimeException e){
    map.put(AppConstants.STATUS, Status.FAILURE);
    map.put("error",e.getMessage());
    return ResponseEntity.badRequest().body(map);
}
}

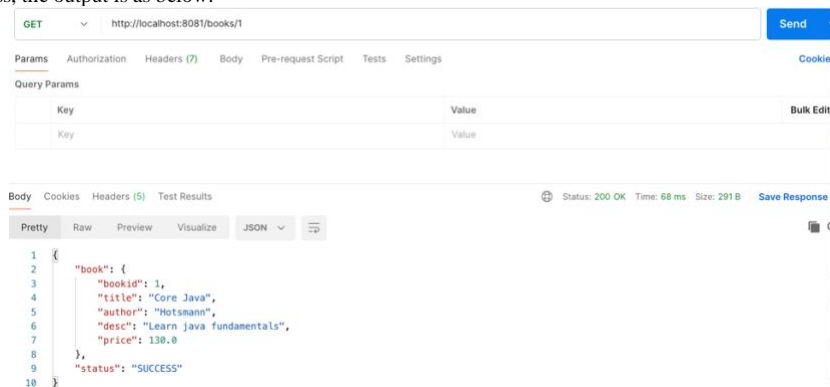
```

DO CHECK THE UTILITY PACKAGE FOR AppConstants and Status used here.

MAKE SURE TO RESTART THE SERVER



For success, the output is as below:



Step 6: @RequestParam

1. Get books method returns all the books. How about we need to give users choice to get books filtered by author? This has to be optional if no filter provided then return all books. Use RequestParam annotation for the same: Update the method as follows:

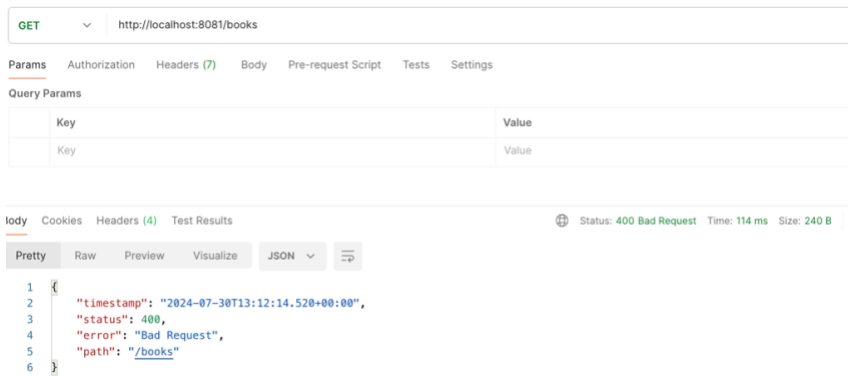
```

@GetMapping
public List<Book> getBooks(@RequestParam String author){
    if(author==null)
        return bookService.getAllBooks();
    return bookService.getBooksByAuthor(author);
}

```

MAKE SURE TO RESTART THE SERVER

To access type in postman url : <http://localhost:8080/books>
You will get below error as value for author was not provided.



Now access with this url : <http://localhost:8081/books?author=kelly>

- But the problem is providing value for author is mandatory. Update the method to make author as required false.

```
@GetMapping
public List<Book> getBooks(@RequestParam(required = false) String author){
    if(author==null)
        return bookService.getAllBooks();
    return bookService.getBooksByAuthor(author);
}
```

MAKE SURE TO RESTART THE SERVER

Now this url works just fine without providing the value for author : <http://localhost:8080/books>

Step 7: @PostMapping

- To add new book we use @PostMapping annotation. Add below method in controller:

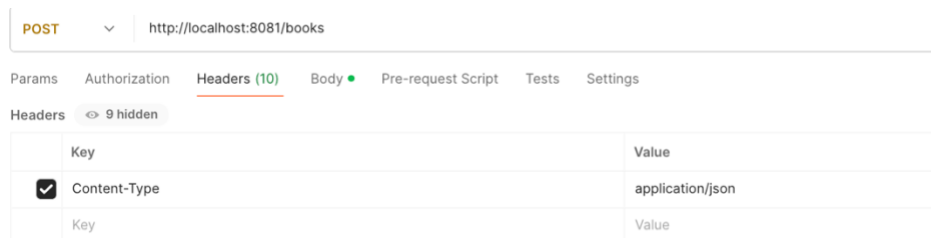
```
@PostMapping
public ResponseEntity<Object> addBook(Book book){
    System.out.println("Book "+book);
    Map<String, Object> map = new HashMap<>();
    try {
        map.put(AppConstants.STATUS, Status.SUCCESS);
        map.put("book",bookService.addNewBook(book) );
        return ResponseEntity.ok(map);
    }
    catch (RuntimeException e){
        map.put(AppConstants.STATUS, Status.FAILURE);
        map.put("error",e.getMessage());
        return ResponseEntity.badRequest().body(map);
    }
}
```

MAKE SURE TO RESTART THE SERVER

Try this url in postman. Make sure to select **POST** from dropdown of POSTMAN :

<http://localhost:8080/books>

ALSO PLEASE UPDATE HEADER AS FOLLOWS:



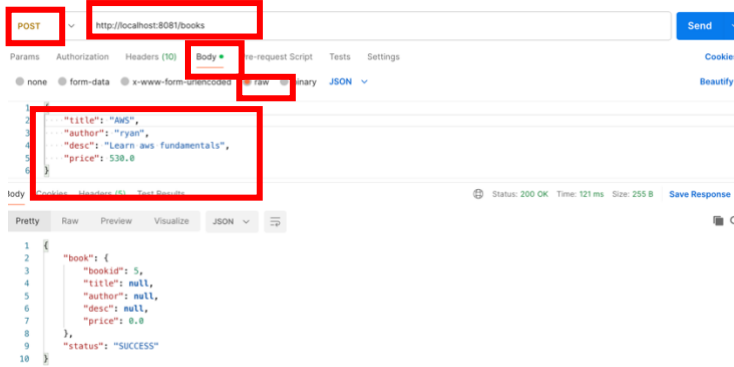
Add below JSON in body
{

```

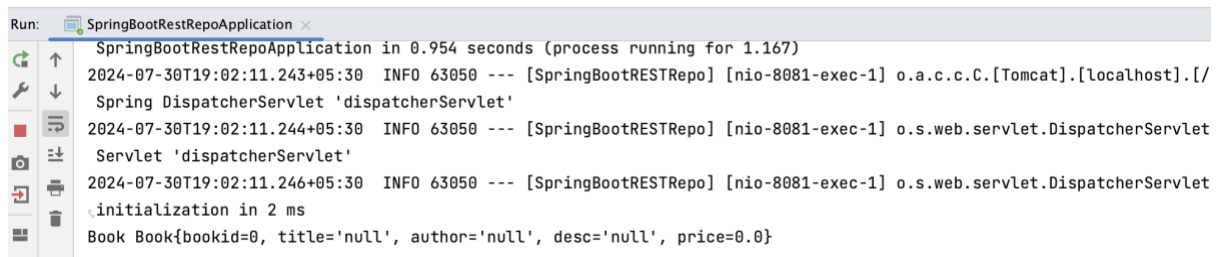
"title": "AWS",
"author": "ryan",
"desc": "Learn aws fundamentals",
"price": 530.0
}

```

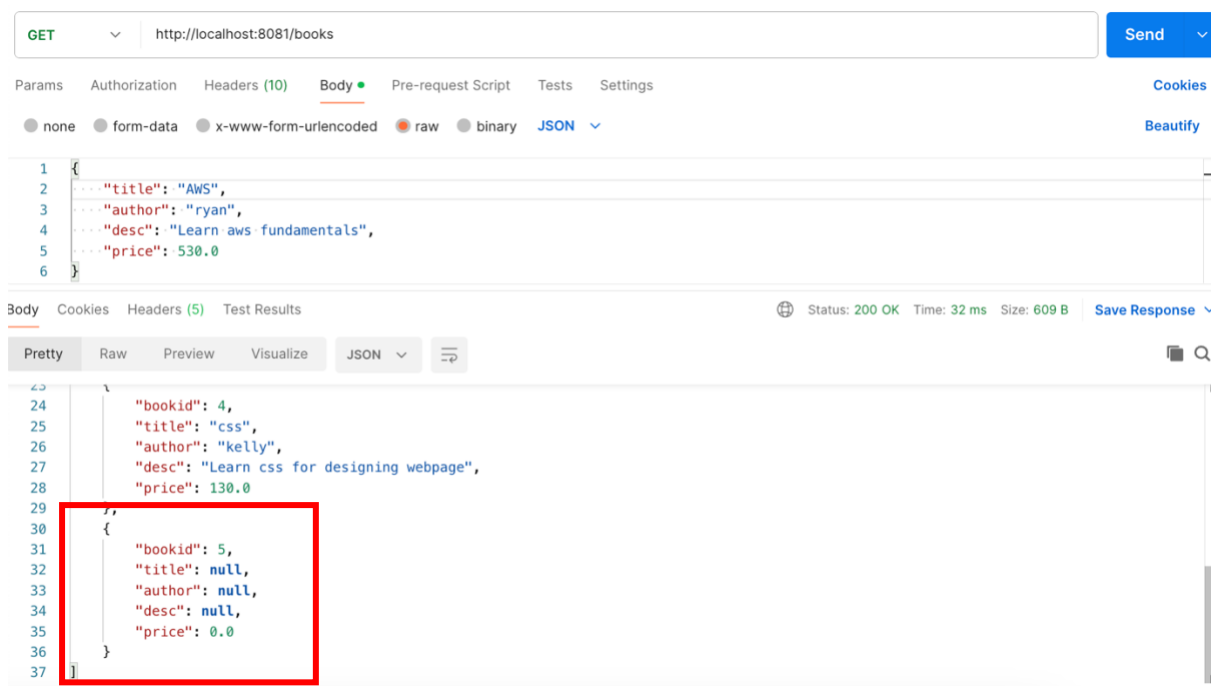
You will get below output on postman: HMMMM???? 😊



Check the IDE console. WHAT??? Book data is null 😊



Check fetching records : Make a GET request to <http://localhost:8080/books>

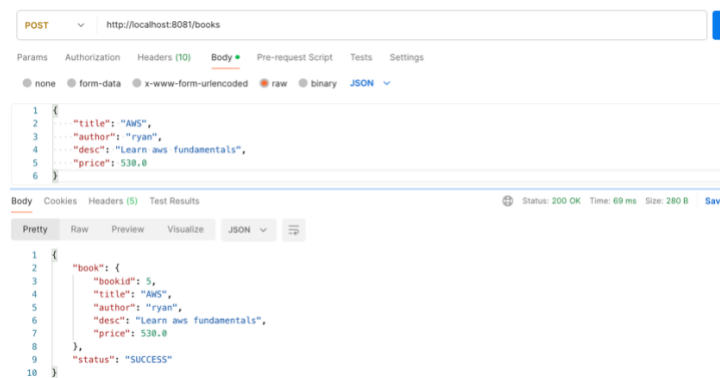


- Looks like spring was not able to map the data coming in the request to java class. We need to add `@RequestBody` in the method parameter for spring to know to do the mapping of JSON data to java class.

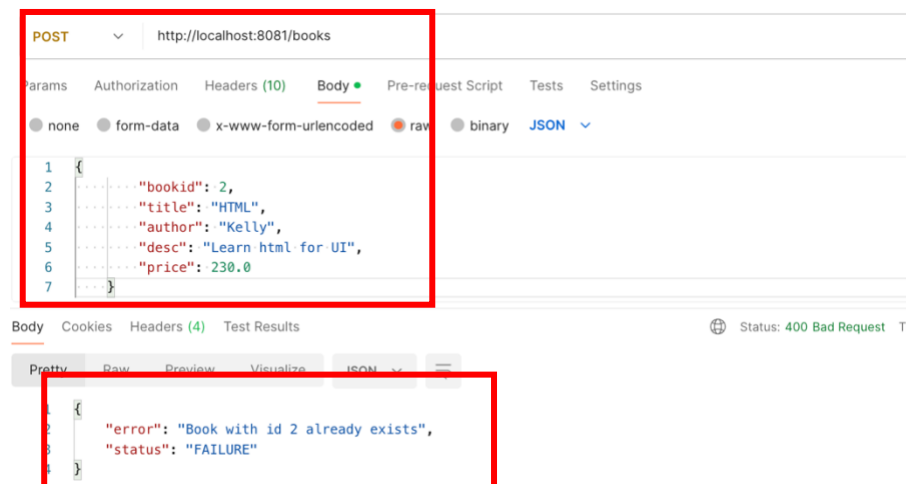
```
@PostMapping
public ResponseEntity<Object> addBook(@RequestBody Book book){
    System.out.println("Book "+book);
    Map<String, Object> map = new HashMap<>();
    try {
        map.put(AppConstants.STATUS, Status.SUCCESS);
        map.put("book",bookService.addNewBook(book) );
        return ResponseEntity.ok(map);
    }
    catch (RuntimeException e){
        map.put(AppConstants.STATUS, Status.FAILURE);
        map.put("error",e.getMessage());
        return ResponseEntity.badRequest().body(map);
    }
}
```

MAKE SURE TO RESTART THE SERVER

Now checking the above url's for POST will work as expected: DO NOT FORGET THE **HEADER**



Also do check for adding an already existing book. Should get output as follows: DO NOT FORGET THE **Header**



Step 8: @PutMapping

- To update a book add below method:

```
@PutMapping
public ResponseEntity<Object> updateBook(@RequestBody Book book){
    System.out.println("Book "+book);
    Map<String, Object> map = new HashMap<>();
    try {
        map.put(AppConstants.STATUS, Status.SUCCESS);
        map.put("book",bookService.updateBook(book) );
    }
```

```

    return ResponseEntity.ok(map);
}
catch (RuntimeException e){
    map.put(AppConstants.STATUS, Status.FAILURE);
    map.put("error",e.getMessage());
    return ResponseEntity.badRequest().body(map);
}
}

```

MAKE SURE TO RESTART THE SERVER

ALSO PLEASE UPDATE HEADER :

Content-Type: application/json

PUT http://localhost:8081/books

Headers (10) Body ● Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary JSON

```

{
  "bookid": 1,
  "title": "Core Java and Functional Programming",
  "author": "Cay Hotsmann",
  "desc": "Learn java fundamentals",
  "price": 230.0
}

```

body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```

1 {
2   "book": {
3     "bookid": 1,
4     "title": "Core Java and Functional Programming",
5     "author": "Cay Hotsmann",
6     "desc": "Learn java fundamentals",
7     "price": 230.0
8   },
9   "status": "SUCCESS"
10 }

```

Try to update a book that does not exist:

PUT http://localhost:8081/books

Params Authorization Headers (10) Body ● Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary JSON

```

1 {
2   "bookid": 20,
3   "title": "HTML",
4   "author": "Kelly",
5   "desc": "Learn html for UI",
6   "price": 230.0
7 }

```

body Cookies Headers (4) Test Results

Status: 400 Bad Req

Pretty Raw Preview Visualize JSON

```

1 {
2   "error": "Book with id 20 does not exist",
3   "status": "FAILURE"
4 }

```

Step 9: @DeleteMapping

1. To delete a book add below method:

```

@DeleteMapping("/{id}")
public ResponseEntity<Object> deleteBook(@PathVariable int id){

```

```

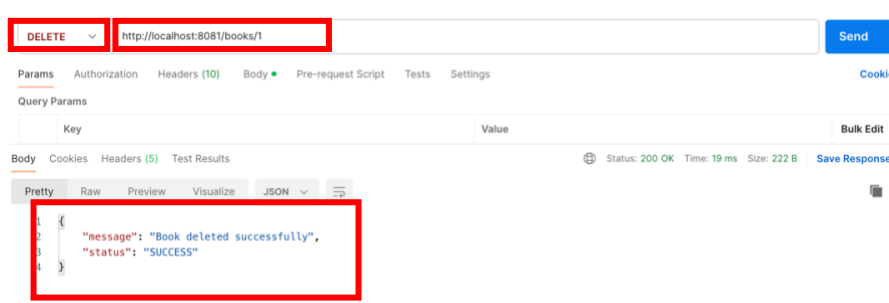
Map<String, Object> map = new HashMap<>();
try {
    map.put(AppConstants.STATUS, Status.SUCCESS);
    if(bookService.deleteBook(id)) {
        map.put("message", "Book deleted successfully");
        return ResponseEntity.ok(map);
    }
}
catch (RuntimeException e){
    map.put(AppConstants.STATUS, Status.FAILURE);
    map.put("error",e.getMessage());

}
return ResponseEntity.badRequest().body(map);
}

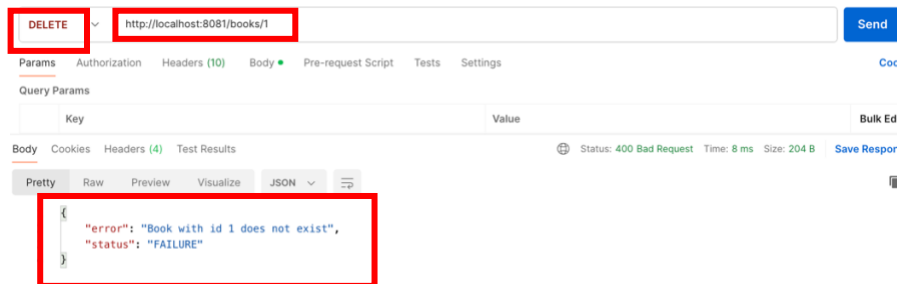
```

MAKE SURE TO RESTART THE SERVER

Try for success deletion for a book that exists with the id:



Try for failure deletion for a book that does not exists with the id:



Modify the code to fetch data from database

Step 10: Update properties file for Db connection parameters

1. **Open database and make sure to create database named "neueda"**
2. Add below dependency in pom.xml . **DO NOT FORGET TO RELOAD MAVEN**

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <scope>runtime</scope>
</dependency>
```

3. Open application.properties file within the resources folder and add the database-related connection parameters:

MAKE SURE CREDENTIALS ARE AS PER YOUR DATABASE

```
spring.datasource.url=jdbc:mysql://localhost:3306/neueda
spring.datasource.username=root
spring.datasource.password=c0nygre
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver

# below property if used will perform DDL operations to create table
# create will always drop and recreate
# update will create only if none exists
spring.jpa.hibernate.ddl-auto=create

# below property if used will show the sql queries generated by hibernate
spring.jpa.show-sql=true

// will decorate the console output
spring.output.ansi.enabled=ALWAYS
```

Step 11: Map java class to database table so that it becomes **"DATABASE MANAGED ENTITY"**

1. To let hibernate [which is part of spring boot data jpa] know how to map this class to a database table and create the table accordingly, update the Book class for respective annotations:

```
// tells to treat this class as database table.
@Entity

//[Optional ] if used will create the table with the name as book_table else maps with classname
//@Table(name = "book_table")

public class Book {
  // Used for primary key identifier
  @Id
  // to generate auto-increment values use Identity strategy
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int bookid;
  // rest remains the same
}
```

2. Run the main method and you should get below error:


```

at org.springframework.boot.SpringApplication.run(SpringApplication.java:1352) ~[spring-boot-3.3.2.jar:3.3.2]
at com.training.SpringBootDatabaseRepo.SpringBootDatabaseRepoApplication.main(SpringBootDatabaseRepoApplication.java:10) ~[classes/:na]
Caused by: java.sql.SQLException: Create breakpoint : You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version fo
the right syntax to use near 'desc varchar(255), title varchar(255), primary key (bookid)) engine=InnoDB' at line 1
at com.mysql.cj.jdbc.exceptions.SQLExceptionor.createSQLException(SQLException.java:111) ~[mysql-connector-j-8.3.0.jar:8.3.0]
at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:122) ~[mysql-connector-j-8.3.0.jar:8.3.0]
at com.mysql.cj.jdbc.StatementImpl.executeInternal(StatementImpl.java:779) ~[mysql-connector-j-8.3.0.jar:8.3.0]
at com.mysql.cj.jdbc.StatementImpl.execute(StatementImpl.java:653) ~[mysql-connector-j-8.3.0.jar:8.3.0]
at com.zaxxer.hikari.pool.ProxyStatement.execute(ProxyStatement.java:96) ~[HikariCP-5.1.0.jar:na]
at com.zaxxer.hikari.pool.HikariProxyStatement.execute(HikariProxyStatement.java) ~[HikariCP-5.1.0.jar:na]
at org.hibernate.tool.schema.internal.exec.GenerationTargetToDatabase.accept(GenerationTargetToDatabase.java:80) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]
... 38 common frames omitted

```

3. The reason is desc is a keyword in database. Hence use @Column to specify the name of column to be created in database table. Update class as follows:

```

// default it creates column with length 255 and allows null. Modify using attributes as below
@Column(length = 100, nullable = false)
private String title;
@Column(length = 100, nullable = false)
private String author;
@Column(name = "description")
private String desc;
// Other parts remain the same
}

```

4. Now if you run main method again, you should see table created in database. Check description : default 255 and null

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	bookid	int(11)			No	None		AUTO_INCREMENT	Change Drop More
2	price	double			No	None			Change Drop More
3	author	varchar(100)	utf8_general_ci		No	None			Change Drop More
4	title	varchar(100)	utf8_general_ci		No	None			Change Drop More
5	description	varchar(255)	utf8_general_ci		Yes	NULL			Change Drop More

5. Once the entities are created, run the main class, see the console where it shows the queries generated by spring boot since we used show-sql as true and see the database where tables are created

```

Hibernate: drop table if exists book
Hibernate: create table book (bookid integer not null auto_increment, price float(53) not null, author varchar(100) not null, title varchar(100) not null,
description varchar(255), primary key (bookid)) engine=InnoDB
2024-07-30T12:23:50.772+05:30 INFO 48765 --- [SpringBootDatabaseRepo] [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA
EntityManagerFactory for persistence unit 'default'

```

Step 12: JpaRepository

1. Once the tables are created you need to perform CRUD operations. Spring boot provides an interface for the same. **[GOOD NEWS!!! – NO NEED TO WRITE SQL QUERIES 😊]**
2. Just extend the interface and we get the most relevant CRUD methods for standard data access available in a standard DAO.

```

public interface BookRepo extends JpaRepository<Book, Integer> {
}

```

3. JpaRepository<T,ID> is generic interface where
T : represents the database managed entity name
ID: represents the type of identifier.

Step 13: service layer

1. Create BookServiceRepo class to perform business operations on the book table as follows:

```

package com.training.SpringBootDatabaseRepo.service;

```

```

import com.training.SpringBootDatabaseRepo.entity.Book;
import com.training.SpringBootDatabaseRepo.repo.BookRepo;
import jakarta.persistence.EntityExistsException;
import jakarta.persistence.EntityNotFoundException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class BookServiceRepo {
    @Autowired
    private BookRepo bookRepo;

    public long getTotalBookCount(){
        return bookRepo.count();
    }
    public List<Book> getAllBooks(){
        return bookRepo.findAll();
    }
    public Book addNewBook(Book book){
        if(!bookRepo.existsById(book.getBookid()))
            throw new EntityExistsException("Cannot add "+book.getBookid()+" already exists");
        return bookRepo.save(book);
    }

    public Book updateBook(Book book){
        if(!bookRepo.existsById(book.getBookid()))
            throw new EntityNotFoundException("cannot update "+book.getBookid()+" does not exist");
        return bookRepo.save(book);
    }
    public boolean deleteBook(int id){
        if(!bookRepo.existsById(id))
            throw new EntityNotFoundException("cannot delete "+id+" does not exist");
        bookRepo.deleteById(id);
        return true;
    }
    public List<Book> getBooksByAuthor(String author){
        return bookRepo.findByAuthor(author);
    }
    public Book getBookById(int id){
        if(!bookRepo.existsById(id))
            throw new EntityNotFoundException(id+" not found");
        return bookRepo.findById(id).get();
    }
}

```

Step 14: Appendix

JPA Annotations:

1. **@Entity** : used at the class level and marks the class as a persistent entity. It signals to the JPA provider that the class should be treated as a table in the database
2. **@Id**: indicating the member field below is the primary key of the current entity.
3. **@GeneratedValue** : This annotation is generally used in conjunction with **@Id** annotation to automatically generate unique values for primary key columns within our database tables.
4. The **@GeneratedValue** annotation provides us with different strategies for the generation of primary keys which are as follows :
 - a. **GenerationType.IDENTITY**: This strategy will help us to generate the primary key value by the database itself using the auto-increment column option. It relies on the database's native support for generating unique values.

- b. `GenerationType.AUTO`: This is a default strategy and the persistence provider which automatically selects an appropriate generation strategy based on the database usage.
 - c. `GenerationType.TABLE`: This strategy uses a separate database table to generate primary key values. The persistence provider manages this table and uses it to allocate unique values for primary keys.
 - d. `GenerationType.SEQUENCE`: This generation-type strategy uses a database sequence to generate primary key values. It requires the usage of database sequence objects, which varies depending on the database which is being used.
- 5. `@Column` : used to customize the mapping of a specific field to a database column. While JPA automatically maps most fields based on naming conventions, the `@Column` annotation provides developers with greater control over the mapping process.
 - 6. `@OneToOne`: is used to associate one JAVA object with another JAVA object.
 - 7. `@JoinColumn` annotation is used to define a foreign key with the specified name
 - 8. `@JoinTable` annotation in JPA is used to customize the association table that holds the relationships between two entities in a many-to-many relationship. This annotation is often used in conjunction with the `@ManyToMany` annotation to define the structure of the join table.