

Table of Contents

Step 1: Create Spring Boot Project	2
Step 2: Understand Spring Boot as opinionated.....	3
Step 3: Understand Spring object creation	4
Step 4: Understand DI (@Autowired) , @RestController and @RequestMapping	7
Step 5: @PathVariable	9
Step 6: @RequestParam	11
Step 7: @PostMapping	12
Step 8: @PutMapping.....	15
Step 9: @DeleteMapping	16
Step 10: XML Response.....	17
Step 11: Exception Handling.....	18
Step 12: Bean Validation API	20
Step 13: Logging in Spring Boot	22
Step 14: Actuator and Health Metrics.....	24
Step 15: Spring Boot Caching	30

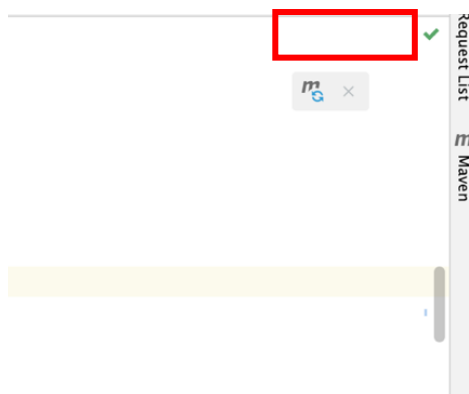
Step 1: Create Spring Boot Project

1. Please download **POSTMAN** on your VM.
<https://www.postman.com/downloads/>
2. Open below url on the browser:
<https://start.spring.io/>
3. Create spring boot project as shown on the screen below: **DO NOT FORGET TO click Add Dependencies**

4. Click on Generate, it will download the zip. Extract and open project using IntelliJ
5. 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:



6. 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
7. **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. Since boot is opinionated framework, it looks for database configuration as it found dependency in the build path. Add below in application.properties:

```
spring.data.mongodb.uri=mongodb+srv://<username>:<password>@democluster.6c2vj.mongodb.net/?retryWrites=true&w=majority&appName=DemoCluster
```

for local mongodb

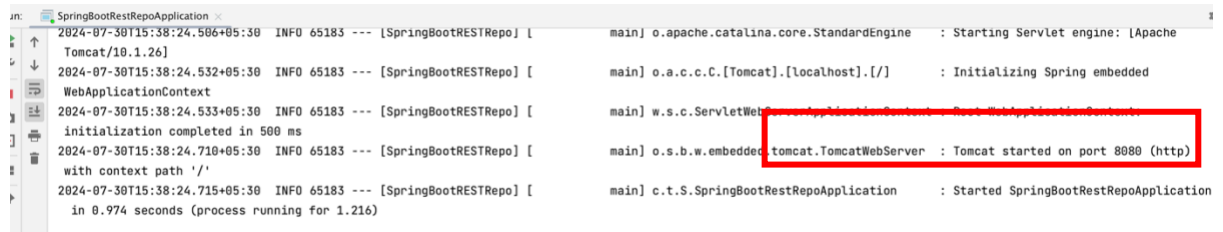
spring.data.mongodb.uri=mongodb://localhost:27017/telecomdb

```
logging.level.org.springframework.data.mongodb.core.MongoTemplate=DEBUG
```

```
logging.level.org.mongodb.driver.protocol.command=DEBUG
```

```
spring.data.mongodb.database=telecomdb
```

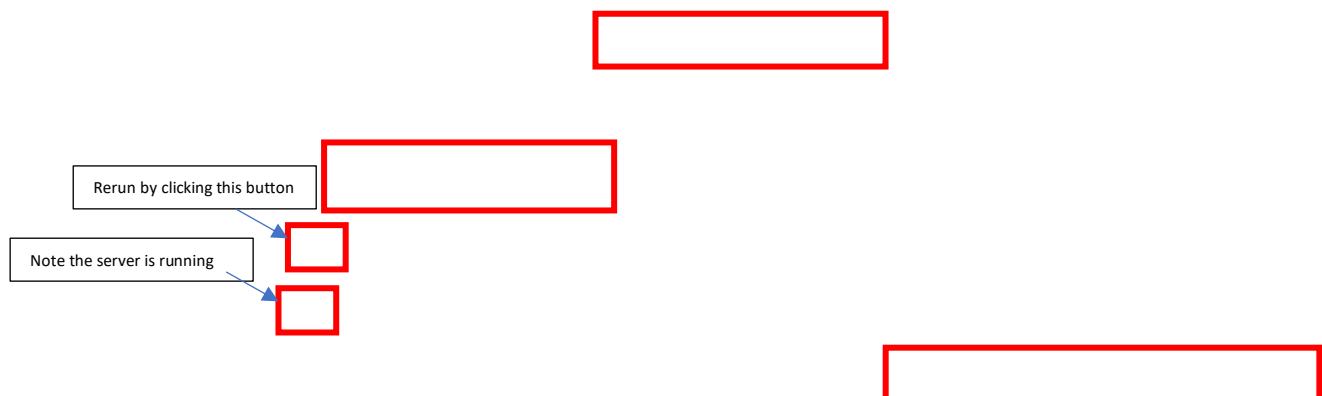
4. Run the main method and observe the console.

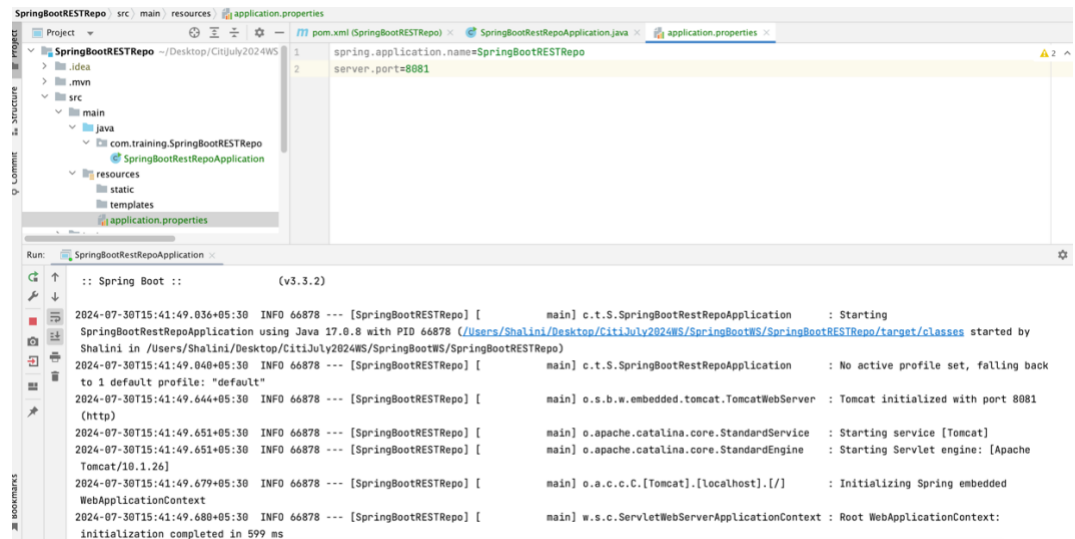


```
2024-07-30T15:38:24.506+05:30 INFO 65183 --- [SpringBootRESTRepo] [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.26]
2024-07-30T15:38:24.532+05:30 INFO 65183 --- [SpringBootRESTRepo] [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded 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





Step 3: Understand Spring object creation

1. Create a class Book as follows:

package com.boot.demo.springbootdemo.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 + "\n" +
            ", author=" + author + "\n" +
            ", desc=" + desc + "\n" +

```

```

        ", 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;
    }
}

```

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

```

package com.boot.demo.springbootdemo.service;

import com.boot.demo.springbootdemo.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
}

```

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

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

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

```
package com.boot.demo.springbootdemo.rest;
```

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

```
@RestController
```

```
public class BookRestController {
```

```
    public BookRestController() {
```

```
        System.out.println("Book Rest Controller default constructor");
```

```
    }
```

```
}
```

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

4. 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();
    }
}
```

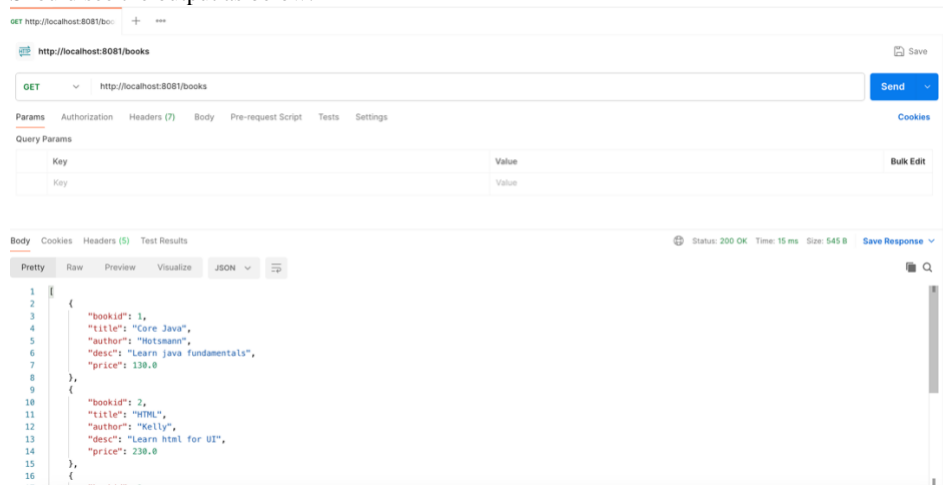
MAKE SURE TO RESTART THE SERVER

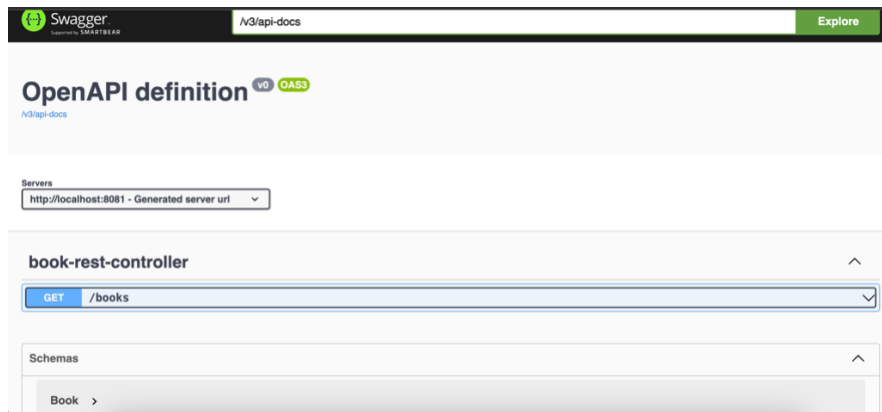
It will be available at <http://localhost:8081/books>.

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:

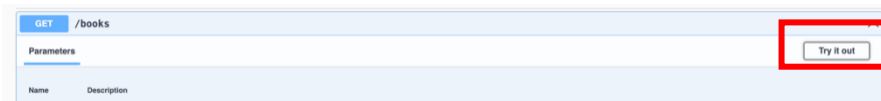
<http://localhost:8081/swagger-ui/index.html#/>

Should see the output as below:

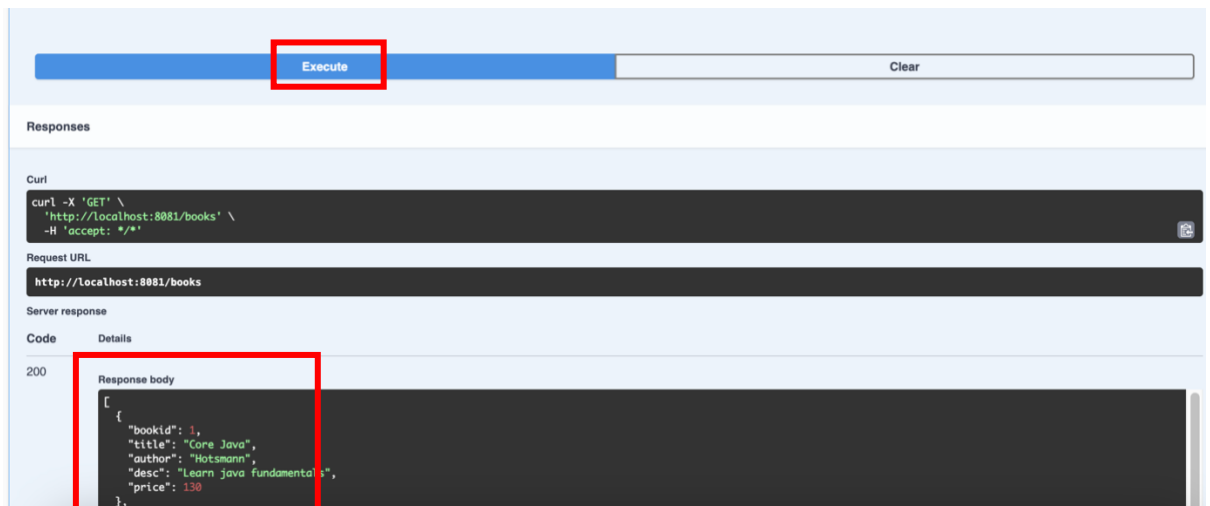




When you click on the arrow, you get Try it out. Click on that will have an option to Execute.



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);
}
```

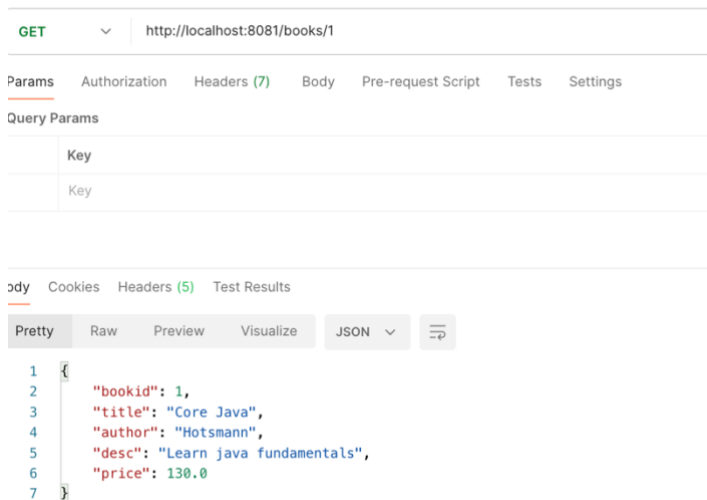
2. To make this method available at REST API endpoint and return book for a specific id, update the method as follows:

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

MAKE SURE TO RESTART THE SERVER

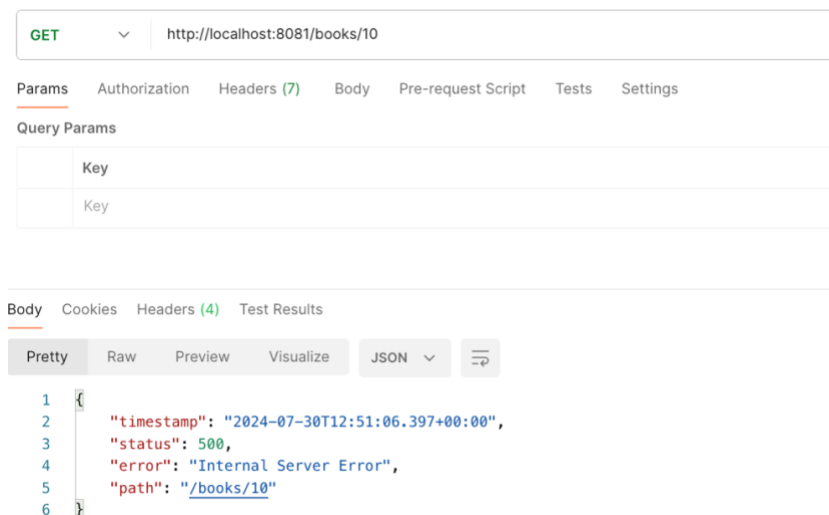
To access type in postman url : <http://localhost:8081/books/1>

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

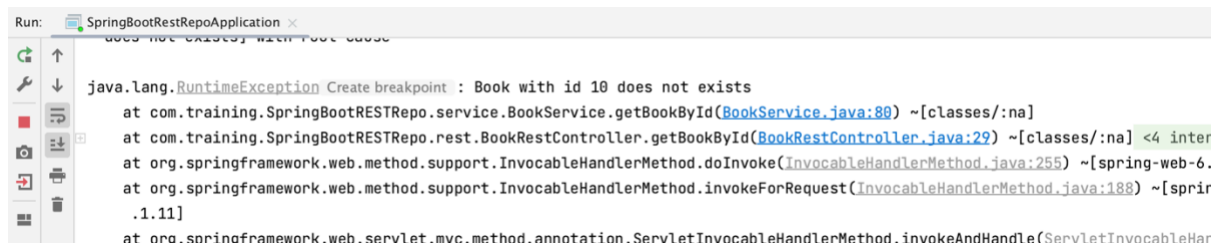


{ } -> is the placeholder for the value [1] passed in the url.
{id} is mapped to method parameter id using @PathVariable annotation.

- Now try to access a book that does not exist: 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) );
    }
}
```

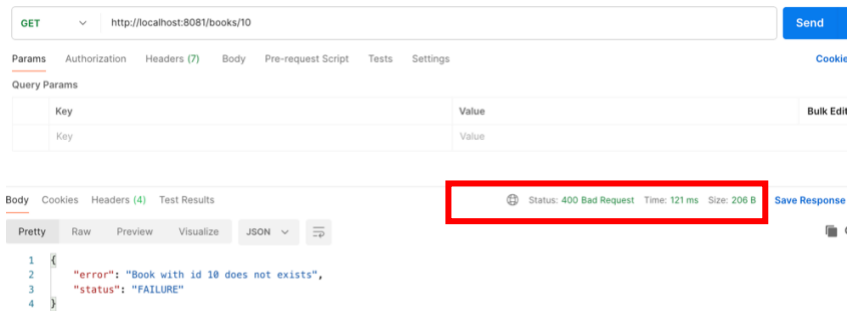
```

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

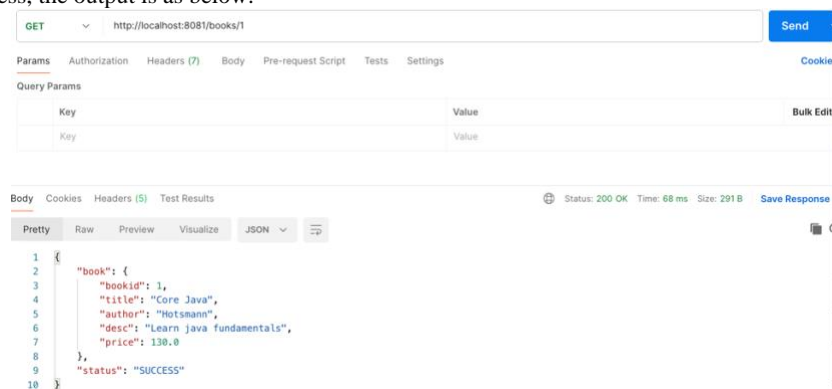
```

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:

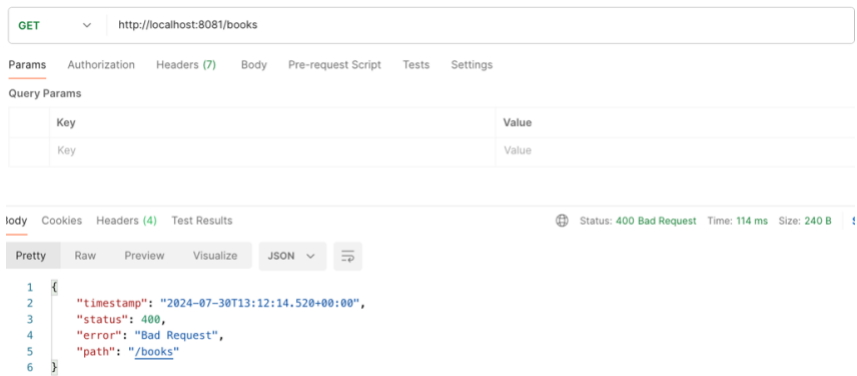
```

@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:8081/books>
 You will get below error as value for author was not provided.



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

2. 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:8081/books>

Step 7: @PostMapping

1. 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:8081/books>

ALSO PLEASE UPDATE HEADER AS FOLLOWS:

POST http://localhost:8081/books

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

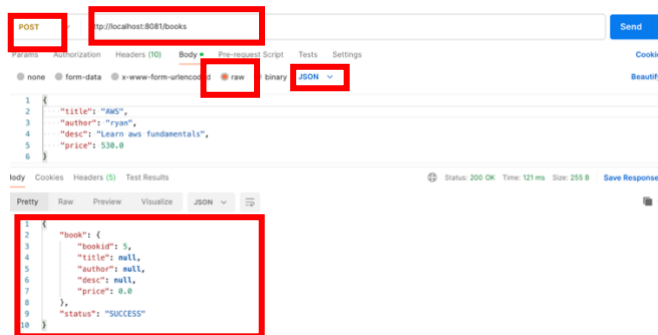
Headers 9 hidden

Key	Value
<input checked="" type="checkbox"/> Content-Type	application/json
Key	Value

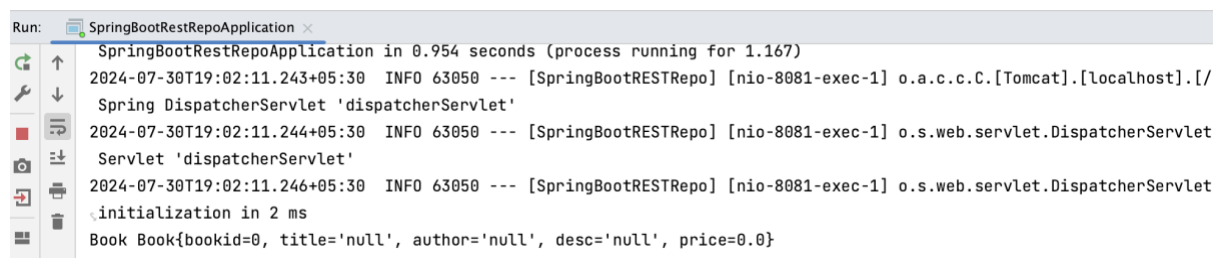
Add below JSON in body

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

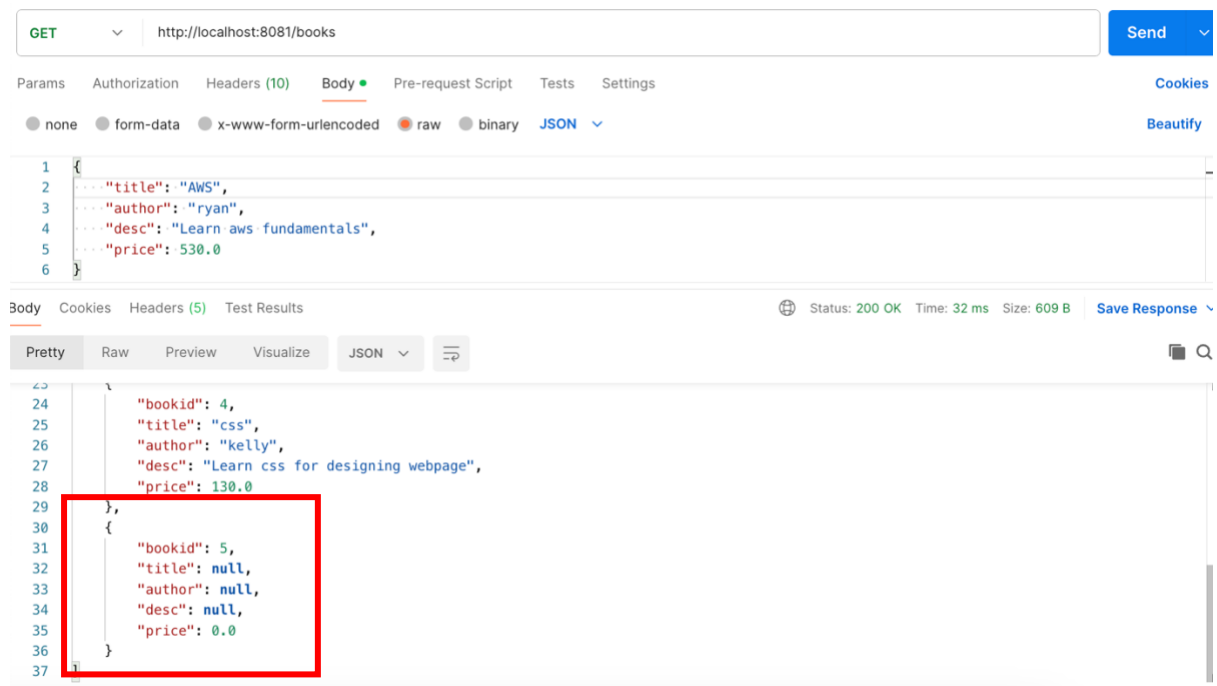
You will get below output on postman: HMMMM??? 🤔



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



Check fetching records : Make a GET request to <http://localhost:8081/books>



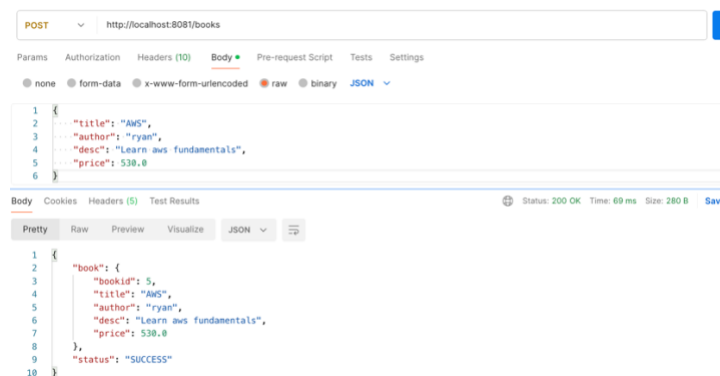
- Looks like spring was not able to map the data coming in the request to java class. We need to add `@RequestBody` in the method parameter for spring to know to do the mapping of JSON data to java class.

@PostMapping

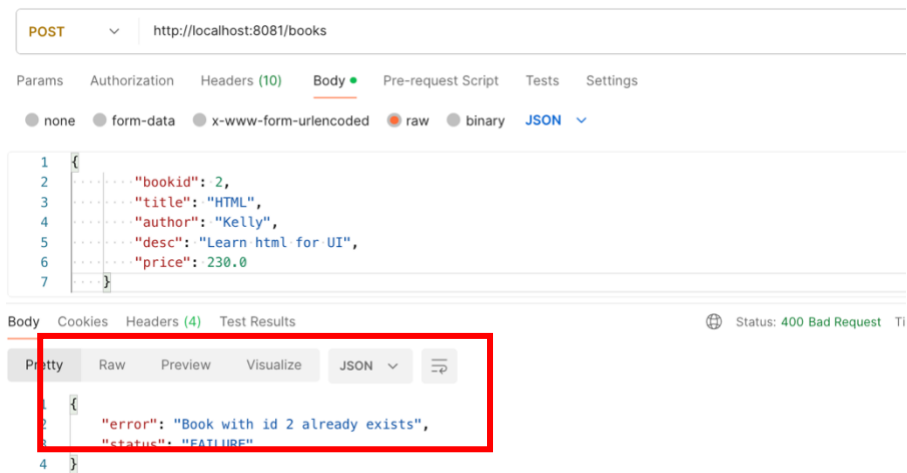
```
public ResponseEntity<Object> addBook(@RequestBody Book book){
    System.out.println("Book "+book);
    Map<String, Object> map = new HashMap<>();
    try {
        map.put(AppConstants.STATUS, Status.SUCCESS);
        map.put("book",bookService.addNewBook(book) );
        return ResponseEntity.ok(map);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

1. 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

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

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

```

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

```

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);
    }
}

```



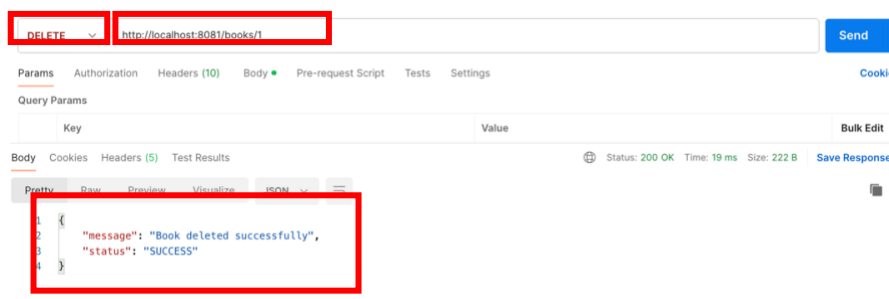
```

        map.put("error",e.getMessage());
    }
    return ResponseEntity.badRequest().body(map);
}

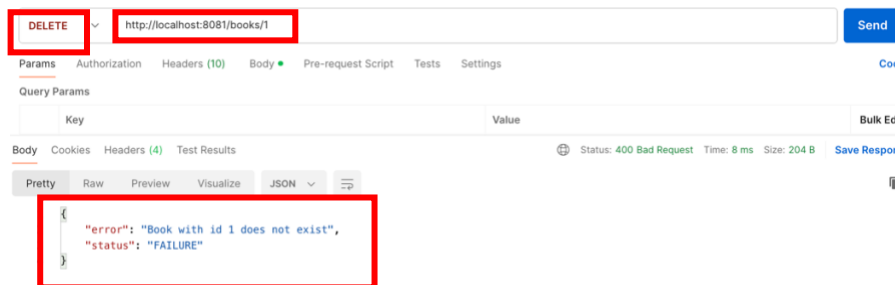
```

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 exist with the id:



Step 10: XML Response

1. Add below dependency in pom.xml

```

<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>

```

2. Modify the `getBooks` method as follows to produce data of type JSON and XML both:

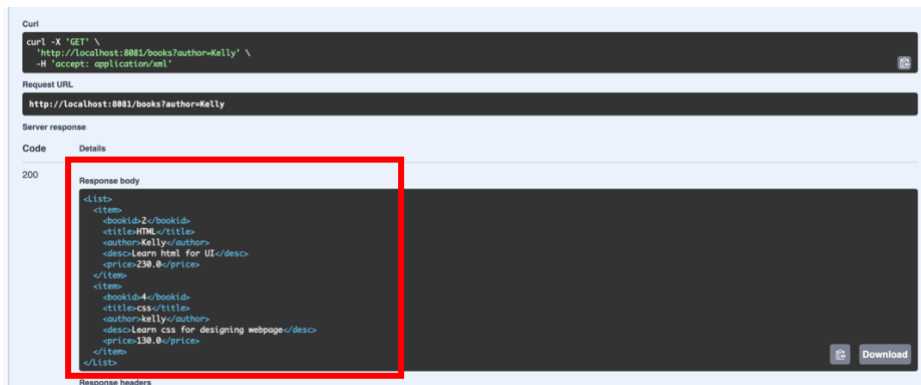
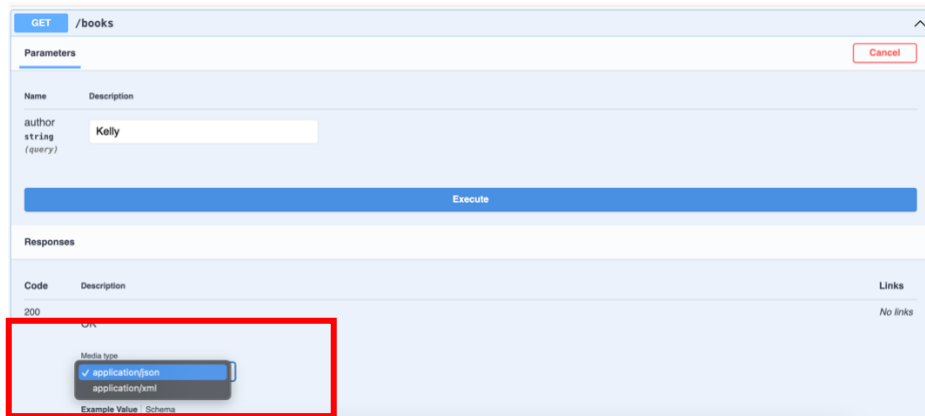
```

@GetMapping(produces = { "application/json", "application/xml" })
public List<Book> getBooks (@RequestParam(required = false, defaultValue = "Kelly") String author){
    if(author.equalsIgnoreCase("all"))
        return this.bookService.getAllBooks();
    return this.bookService.getBooksByAuthor(author);
}

```

3. Restart the server and try on postman or Swagger and you should get XML data as follows:

Swagger provides you with the option to specify the response type as follows: Then when you choose xml or json and click on Execute button, the response will be of the type specified.



Step 11: Exception Handling

1.1. Add below dependency in pom.xml

In previous BookRestController, every method is responsible for handling the exceptions. There is lots of repetition for the exception handler. Spring boot provides a central exception handler in 2 ways as follows:

1.1.1. Use @ExceptionHandler

Spring Boot provides the @ExceptionHandler annotation to handle exceptions thrown by a specific controller method. This annotation can be used to provide customized error responses for specific exceptions.

1.1.2. Create a new Rest Controller as follows and look at the method with @ExceptionHandler.

DO NOT FORGET TO CHANGE THE REQUEST MAPPING AS HIGHLIGHTED BELOW

```
package com.boot.demo.springbootdemo.rest;
import com.boot.demo.springbootdemo.entity.Book;
import com.boot.demo.springbootdemo.service.BookServiceRepo;
import com.boot.demo.springbootdemo.utility.AppConstants;
import com.boot.demo.springbootdemo.utility.Status;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
```

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
@RestController
```

```
@RequestMapping("/books/ex")
```

```

public class BookRestControllerExceptionHandler {

    @Autowired
    private BookServiceRepo bookService;
    public BookRestControllerExceptionHandler() {
        System.out.println("Book Rest Controller default constructor");
    }

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<Object> handleResourceNotFoundException(RuntimeException ex) {
        Map<String, Object> body = new HashMap<>();
        body.put("error", ex.getMessage());
        return new ResponseEntity<>(body, HttpStatus.NOT_FOUND);
    }

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

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

    @PostMapping
    public ResponseEntity<Object> addBook(@RequestBody Book book){
        System.out.println("Book "+book);
        Map<String, Object> map = new HashMap<>();
        map.put(AppConstants.STATUS, Status.SUCCESS);
        map.put("book",bookService.addNewBook(book) );
        return ResponseEntity.ok(map);
    }

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

    @DeleteMapping("/{id}")
    public ResponseEntity<Object> deleteBook(@PathVariable int id){
        Map<String, Object> map = new HashMap<>();

        map.put(AppConstants.STATUS, Status.SUCCESS);
        if(bookService.deleteBook(id)) {
            map.put("message", "Book deleted successfully");
            return ResponseEntity.ok(map);
        }
        return ResponseEntity.badRequest().body(map);
    }
}

```

1.1.3. Use @ControllerAdvice

Spring Boot provides the @ControllerAdvice annotation to handle exceptions globally across all controllers. This annotation can be used to provide a centralized error handling mechanism for an entire application.

1.1.4. Create a class as follows to handle global exception

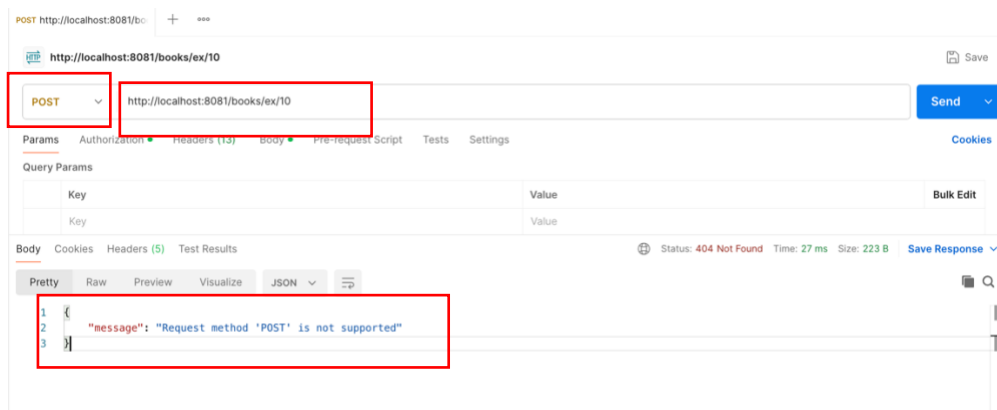
```
package com.boot.demo.springbootdemo.exception;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import java.util.HashMap;
import java.util.Map;

@ControllerAdvice
public class MyGlobalHandler {

    MyGlobalHandler(){
        System.out.println("Global handler");
    }
    @ExceptionHandler(Exception.class)
    public ResponseEntity<Object> handleResourceNotFoundException(Exception ex) {
        Map<String, Object> body = new HashMap<>();
        body.put("message", ex.getMessage());

        return new ResponseEntity<>(body, HttpStatus.NOT_FOUND);
    }
}
```

1.1.5. To test the global handler, make a request as follows and see the global handler in action:



Step 12: Bean Validation API

1. Add below dependency

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

2. Add below annotations on the entity class:

```
@NotNull(message = "Title must not be empty")
private String author;
```

```
@Positive(message = "Price must be positive")
private double price;
```

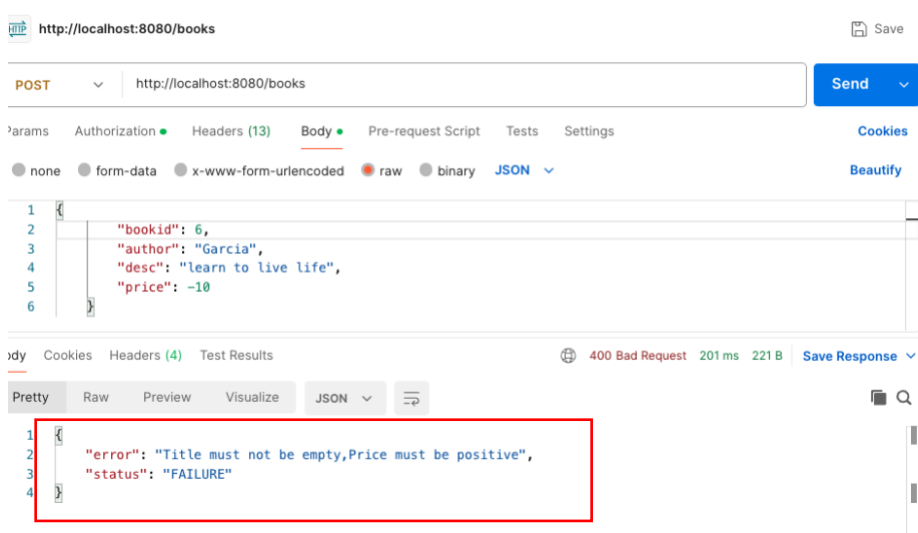
3. Update the controller as follows:

```
public ResponseEntity<Object> addBook(@Valid @RequestBody Book book)
```

4. Now run the code and post below json:

```
{
  "bookid": 0,
  "title": "string",
  "desc": "string",
  "price": -100
}
```

Should get an exception as follows:



5. To provide custom exception handler, add below code:

```
@ExceptionHandler({Exception.class, MethodArgumentNotValidException.class})
public ResponseEntity<Object> handleException(Exception ex){
    Map<String, Object> map = new HashMap<>();
    map.put(AppConstants.STATUS, Status.FAILURE);

    if(ex instanceof MethodArgumentNotValidException){
        String msg = ((MethodArgumentNotValidException) ex).getAllErrors()
            .stream().map(ObjectError::getDefaultMessage)
            .collect(Collectors.joining(", "));
        map.put("error", msg);
        return ResponseEntity.badRequest().body(map);
    }
    System.out.println("general exception");
    System.out.println(ex.getMessage());
    map.put("error", ex.getMessage());
    return ResponseEntity.badRequest().body(map);
}
```

Reference:

In above example, we used only few annotations such as @NotEmpty and @Positive. There are more such annotations to validate request data. Check them out when needed.

@AssertFalse

The annotated element must be false.

@AssertTrue

The annotated element must be true.

@DecimalMax

The annotated element must be a number whose value must be lower or equal to the specified maximum.

@DecimalMin

The annotated element must be a number whose value must be higher or equal to the specified minimum.

@Future

The annotated element must be an instant, date or time in the future.

@Max

The annotated element must be a number whose value must be lower or equal to the specified maximum.

@Min

The annotated element must be a number whose value must be higher or equal to the specified minimum.

@Negative

The annotated element must be a strictly negative number.

@NotBlank

The annotated element must not be null and must contain at least one non-whitespace character.

@NotEmpty

The annotated element must not be null nor empty.

@NotNull

The annotated element must not be null.

@Null

The annotated element must be null.

@Pattern

The annotated CharSequence must match the specified regular expression.

@Positive

The annotated element must be a strictly positive number.

@Size

The annotated element size must be between the specified boundaries (included).

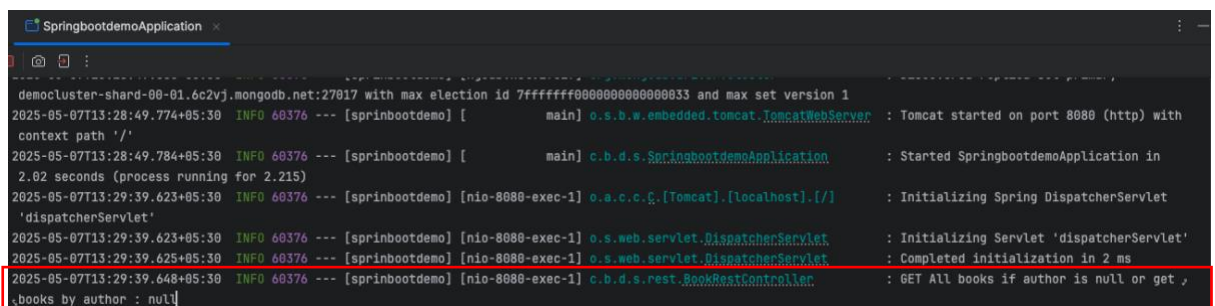
Step 13: Logging in Spring Boot

1. In Spring Boot, the dependency spring-boot-starter-logging includes the logging frameworks. Since many Spring Boot starters include the spring-boot-starter-logging automatically, we unlikely need to add it manually.

For example, adding the dependency web spring-boot-starter-web will automatically include the spring-boot-starter-logging.

2. By default, Spring Boot uses Logback for the logging, and the loggers are pre-configured to use console output with optional file output.
3. To use logging, perform below steps in **BookRestController** class:
 - a. Add below line just above the constructor:
Logger logger = LoggerFactory.getLogger(**BookRestController.class**);
 - b. Add below line within the getBooks() method
logger.info("GET All books if author is null or get books by author : "+ author);

4. Restart the server and you should see the output in INTELLIJ terminal as follows when you make a get request to <http://localhost:8081/books>



```
SpringbootdemoApplication
democluster-shard-00-01.6c2vj.mongodb.net:27017 with max election id 7fffffff0000000000000033 and max set version 1
2025-05-07T13:28:49.774+05:30 INFO 60376 --- [springbootdemo] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with
context path '/'
2025-05-07T13:28:49.784+05:30 INFO 60376 --- [springbootdemo] [main] c.b.d.s.SpringbootdemoApplication : Started SpringbootdemoApplication in
2.02 seconds (process running for 2.21S)
2025-05-07T13:29:39.623+05:30 INFO 60376 --- [springbootdemo] [nio-8080-exec-1] o.a.c.c.f.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet
'dispatcherServlet'
2025-05-07T13:29:39.623+05:30 INFO 60376 --- [springbootdemo] [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2025-05-07T13:29:39.625+05:30 INFO 60376 --- [springbootdemo] [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms
2025-05-07T13:29:39.648+05:30 INFO 60376 --- [springbootdemo] [nio-8080-exec-1] c.b.d.s.rest.BookRestController : GET All books if author is null or get ,
books by author : null
```

5. To send logs to a file. Add below in application.properties as follows:
application.properties
logging.file.name=logs/app.log
Log files rotate when they reach 10 MB

If you restart the server and make a request to <http://localhost:8081/books>
Should see a logs folder and a file app.log within the logs folder

6. Log Levels
TRACE
DEBUG
INFO
WARN
ERROR
OFF

When we set the log level to INFO (default), it logs the INFO, WARN, and ERROR log events.
When we set the log level to DEBUG, it logs the DEBUG, INFO, WARN, and ERROR log events.
When we set the log level to ERROR, it logs only the ERROR log events.

7. We can define the log levels in the application.properties.
root level
logging.level.root=error

package level logging
logging.level.org.springframework.web=debug
logging.level.org.hibernate=error
logging.level.com.mkyong=error

8. Read more here:
<https://mkyong.com/spring-boot/spring-boot-logging-example/>
9. In details:
<https://docs.spring.io/spring-boot/reference/features/logging.html>

Step 14: Actuator and Health Metrics

1. In essence, Actuator brings production-ready features to our application. Monitoring our app, gathering metrics, and understanding traffic or the state of our database becomes trivial with this dependency.

The main benefit of this library is that we can get production-grade tools without actually having to implement these features ourselves.

The actuator mainly exposes operational information about the running application — health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to enable us to interact with it.

Once this dependency is on the classpath, several endpoints are available for us out of the box. As with most Spring modules, we can easily configure or extend it in many ways.

2. Add below dependency in pom.xml file:

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

3. Once you add the dependency, restart the server and should see the actuator endpoint expose as follows:

```

SpringbootdemoApplication x
demoCluster-shard-00-01.6c2vj.mongodb.net:27017, demoCluster-shard-00-01.6c2vj.mongodb.net:27017, passives=[], arbiters=[], primary= demoCluster-shard-00-01.6c2vj.mongodb.net:27017, tagSet=TagSet([Tag(name='availabilityZone', value='aps1-az1'), Tag(name='diskState', value='READY'), Tag(name='nodeType', value='ELECTABLE'), Tag(name='provider', value='AWS'), Tag(name='region', value='AP_SOUTH_1'), Tag(name='workLoadType', value='OPERATIONAL')]), electionId=null, setVersion=1, topologyVersion=TopologyVersion([processId=68176bf5371616a0b68482bf, counter=4], lastWriteDate=Wed May 07 13:53:48 IST 2025, lastUpdateTimeNanos=22816883986708)
2025-05-07T13:53:49.339+05:30 INFO 11443 --- [springbootdemo] [ngodb.net:27017] org.mongodb.driver.cluster : Discovered replica set primary demoCluster-shard-00-01.6c2vj.mongodb.net:27017 with max election id 7fffffff0000000000000003 and max set version 1
2025-05-07T13:53:49.459+05:30 INFO 11443 --- [springbootdemo] [ main] o.s.b.a.e.w.eb.EndpointLinksResolver : Exposing 1 endpoint beneath base path '/actuator'
2025-05-07T13:53:49.493+05:30 INFO 11443 --- [springbootdemo] [ main] o.s.b.w.e.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2025-05-07T13:53:49.505+05:30 INFO 11443 --- [springbootdemo] [ main] c.b.d.s.SpringbootdemoApplication : Started SpringbootdemoApplication in 2.239 seconds (process running for 2.418)

```

4. Type below url in the browser:

<http://localhost:8080/actuator>

Should get below output:

```
localhost:8081/actuator
{
  "root": { "1 item"
    "_links": { "3 items"
      "self": { "2 items"
        "href": "http://localhost:8081/actuator"
        "templated": false
      }
      "health": { "2 items"
        "href": "http://localhost:8081/actuator/health"
        "templated": false
      }
      "health-path": { "2 items"
        "href": "http://localhost:8081/actuator/health/{*path}"
        "templated": true
      }
    }
  }
}
```

5. Actuator comes with most endpoints disabled. Thus, the only two available by default are /health and /info.
6. If we want to enable all of them, we could set below in application.properties file . Alternatively, we can list endpoints that should be enabled.
`management.endpoints.web.exposure.include=*`
7. Now hitting the same url <http://localhost:8080/actuator> should expose more endpoints.

8. All Actuator endpoints are now placed under the /actuator path by default. To tweak this path add below in application.properties file:

management.endpoints.web.base-path=/manage

Now the url changes from /actuator to /manage

NOTE: Quite interestingly, when the management base path is set to /, the discovery endpoint is disabled to prevent the possibility of a clash with other mappings.

9. 3.3. Predefined Endpoints – Reference

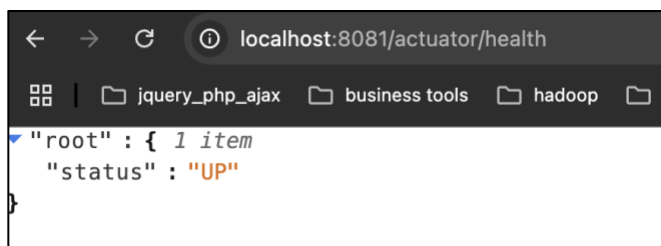
1. /auditevents lists security audit-related events such as user login/logout. Also, we can filter by principal or type among other fields.
2. /beans returns all available beans in our BeanFactory. Unlike /auditevents, it doesn't support filtering.
3. /conditions, formerly known as /autoconfig, builds a report of conditions around autoconfiguration.
4. /configprops allows us to fetch all @ConfigurationProperties beans.
5. /env returns the current environment properties. Additionally, we can retrieve single properties.
6. /flyway provides details about our Flyway database migrations.
7. /health summarizes the health status of our application.
8. /heapdump builds and returns a heap dump from the JVM used by our application.
9. /info returns general information. It might be custom data, build information or details about the latest commit.
10. /liquibase behaves like /flyway but for Liquibase.
11. /logfile returns ordinary application logs.
12. /loggers enables us to query and modify the logging level of our application.
13. /metrics details metrics of our application. This might include generic metrics as well as custom ones.
14. /prometheus returns metrics like the previous one, but formatted to work with a Prometheus server.
15. /scheduledtasks provides details about every scheduled task within our application.
16. /sessions lists HTTP sessions, given we are using Spring Session.
17. /shutdown performs a graceful shutdown of the application.
18. /threaddump dumps the thread information of the underlying JVM.

10. Health Indicators:

- 10.1. To check the health status, go to the browser and hit the below url:

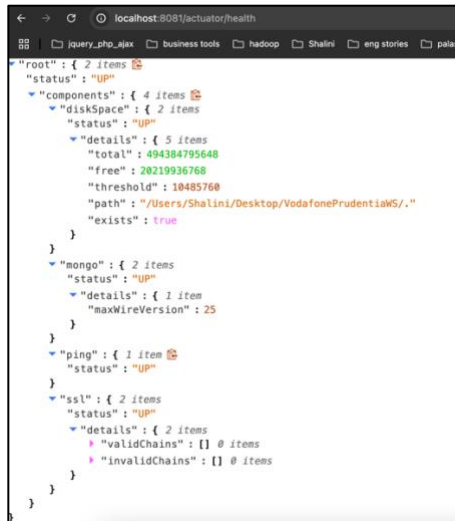
<http://localhost:8081/actuator/health>

You will see only basic status as follows:



- 10.2. Add below in application.properties file to see the details of health endpoint:
management.endpoint.health.show-details=ALWAYS

Now should see the complete status as follows. Your output may differ than mine and its ok.



10.3. We can add custom indicators easily as follows. Let's have a look at a simple custom health check:

```

package com.boot.demo.springbootdemo.actuator;

import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;
import java.net.URL;
import java.net.URLConnection;

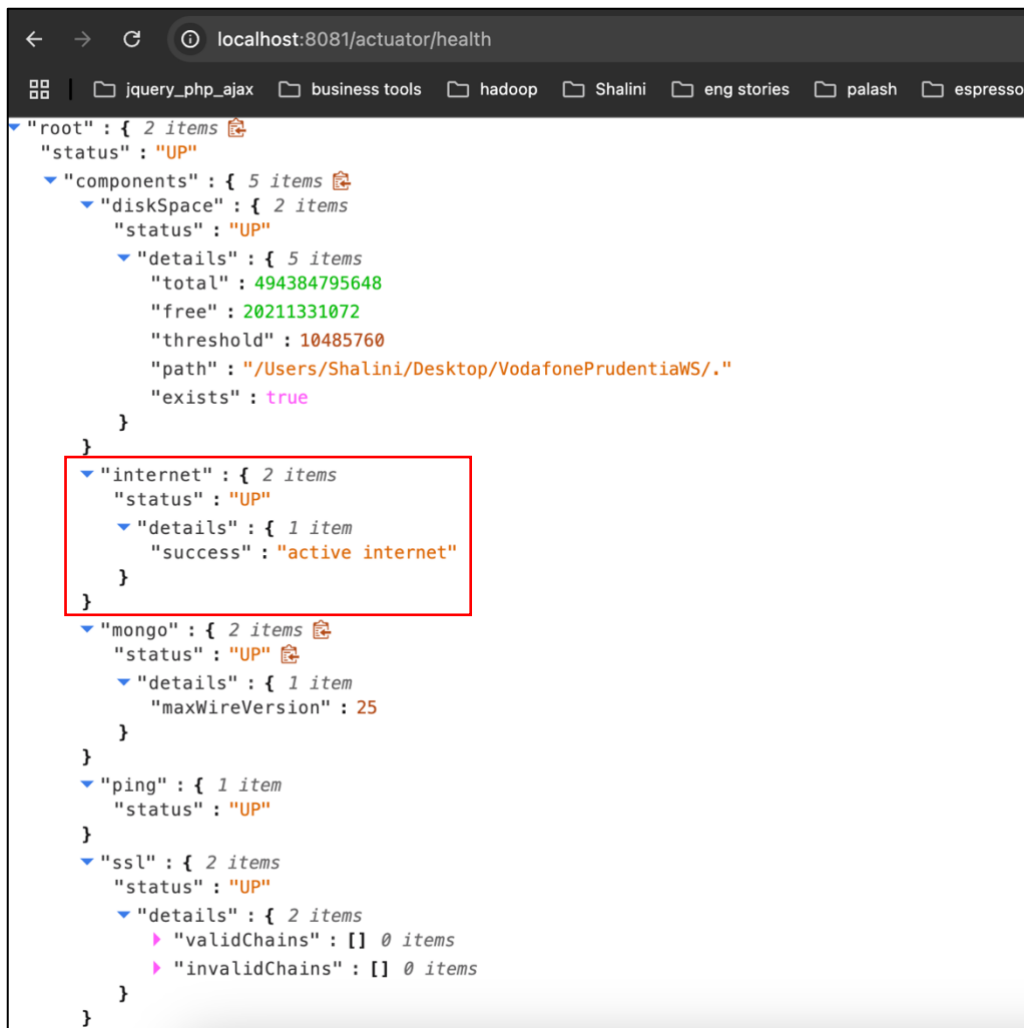
@Component
public class InternetHealthIndicator implements HealthIndicator {
    @Override
    public Health health() {
        Health health = null;
        health = check()==true?Health.up().withDetail("success", "active internet").build():Health.down().withDetail("success", "internet down").build();
        return health;
    }
    private boolean check() {
        boolean flag = false;
        try {
            URL url = new URL("http://www.google.com");
            URLConnection con = url.openConnection();
            con.connect();
            flag = true;
        } catch (Exception e) {
            return flag;
        }
        return flag;
    }
}

```

Now go to the browser and hit the below url:

<http://localhost:8081/actuator/health>

Should see the entire details as follows along with you custom indicator information as well.



11. Metrics in Spring Boot

11.1. Metrics information is available at the <http://localhost:8081/metrics> endpoint has been entirely revamped:

```

{
  "names": [
    "jvm.gc.pause",
    "jvm.buffer.memory.used",
    "jvm.memory.used",
    "jvm.buffer.count",
    // ...
  ]
}

```

12. As we can see, the metrics list provides just overview of the values. To access specific metric values, we can now navigate to the desired metric, such as <http://localhost:8081/actuator/metrics/jvm.gc.pause>, to retrieve a detailed response:

```

{
  "name": "jvm.gc.pause",
  "measurements": [
    {
      "statistic": "Count",
      "value": 3.0
    },
    {
      "statistic": "TotalTime",
      "value": 7.9E7
    }
  ]
}

```

```

    },
    {
        "statistic": "Max",
        "value": 7.9E7
    }
],
"availableTags": [
    {
        "tag": "cause",
        "values": [
            "Metadata GC Threshold",
            "Allocation Failure"
        ]
    },
    {
        "tag": "action",
        "values": [
            "end of minor GC",
            "end of major GC"
        ]
    }
]
}
}

```

13. Creating a Custom Endpoint

We can create custom endpoints.

13.1. Let's create an Actuator endpoint in our application. For this create below classes within actuator package:

13.1.1. Create ReleaseNote class

```
package com.boot.demo.springbootdemo.actuator;
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.time.LocalDateTime;
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ReleaseNote {
    private String version;
    private LocalDateTime date;
    private String changeLogData;
    // Constructors // Getters/Setters
}
```

13.1.2. Create class ReleaseNotesDataRepository as follows:

```
package com.boot.demo.springbootdemo.actuator;
```

```
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
```

```
@Component
public class ReleaseNotesDataRepository {

    private final List<ReleaseNote> releaseNoteList;

    public ReleaseNotesDataRepository() {
        releaseNoteList = new ArrayList<>();
        releaseNoteList.add(new ReleaseNote("RN1", LocalDateTime.of(2023, 11, 13, 12, 12, 12), "Added by admin"));
        releaseNoteList.add(new ReleaseNote("RN2", LocalDateTime.of(2022, 12, 1, 12, 12, 12), "Added by user"));
    }
}
```

```

    }

    public List<ReleaseNote> getReleaseNoteList() {
        return releaseNoteList;
    }

    public ReleaseNote addReleaseNote(ReleaseNote releaseNote) {
        releaseNoteList.add(releaseNote);
        return releaseNote;
    }

    public void deleteReleaseNote(String version) {
        for (int i = 0; i < releaseNoteList.size(); i++) {
            if(releaseNoteList.get(i).getVersion().equals(version)){
                releaseNoteList.remove(releaseNoteList.get(i));
                break;
            }
        }
    }
}

```

13.1.3. Create custom endpoint as follows:

```

@Component
@Endpoint(id="releaseNotes")
public class ReleaseNotesEndpoint {
    @Autowired private ReleaseNotesDataRepository releaseNotesDataRepository;
    @ReadOperation
    public List<ReleaseNote> getReleaseNotes() {
        return releaseNotesDataRepository.getReleaseNoteList();
    }
    @WriteOperation
    public ReleaseNote addReleaseNote(String version, String changeLogData) {
        ReleaseNote releaseNote = new ReleaseNote(version, LocalDateTime.now(), changeLogData);
        return releaseNotesDataRepository.addReleaseNote(releaseNote);
    }
    @DeleteOperation
    public void deleteReleaseNote(@Selector String version) {
        releaseNotesDataRepository.deleteReleaseNote(version);
    }
}

```

To get the endpoint, we need a bean. In our example, we're using `@Component` for this. Also, we need to decorate this bean with `@Endpoint`.

14. The path of our endpoint is determined by the id parameter of `@Endpoint`. In our case, it'll route requests to: `/actuator/ releaseNotes`.
15. Once ready, we can start defining operations using:
 - 15.1. `@ReadOperation`: It'll map to HTTP GET.
 - 15.2. `@WriteOperation`: It'll map to HTTP POST.
 - 15.3. `@DeleteOperation`: It'll map to HTTP DELETE.

When we run the application with the previous endpoint in our application, Spring Boot will register it.

Step 15: Spring Boot Caching

1. What is Caching:

Caching is crucial for building high-performance, scalable applications. It helps store frequently accessed data in the cache, reducing the need to access slower underlying storage systems like a database. Redis is a popular in-memory data structure store used as a cache, database, and message broker. Spring Boot seamlessly integrates with Redis for caching through its Spring Cache abstraction.

2. Introduction to Caching and Redis:

- a. **Caching:** Caching involves storing frequently accessed data in memory so that future requests for that data can be served faster, without fetching it from the primary data source (e.g., database, API).
- b. **Redis:** Redis (Remote Dictionary Server) is an open-source, in-memory data store that supports various data structures such as strings, hashes, lists, sets, and sorted sets. Redis is highly performant and commonly used for caching due to its low latency and high throughput.

3. Spring Cache Abstraction:

The Spring Framework provides a caching abstraction that allows you to define caching logic without binding the application to a specific caching solution. This abstraction can be easily integrated with various caching providers, including Redis.

The Spring Cache abstraction uses annotations to define cache behavior:

- `@EnableCaching`: Enables Spring's annotation-driven cache management.
- `@Cacheable`: Indicates that the method's return value should be cached.
- `@CachePut`: Updates the cache without interfering with the method execution.
- `@CacheEvict`: Removes data from the cache.

4. How Caching Works with Redis in the Spring Boot:

- Cacheable: When `@Cacheable` is used, Spring first checks if the value is present in the Redis cache. If present, it returns the cached value without executing the method. If not, it executes the method, caches the result in Redis, and returns the result.
- CachePut: When `@CachePut` is used, it updates the cache with new data, regardless of whether the method was executed or not.
- CacheEvict: When `@CacheEvict` is used, it removes data from the cache, which is useful for maintaining cache consistency when the underlying data changes.

5. Implement caching in spring boot:

- a. Install redis :
 - i. Locally
https://redis.io/docs/latest/operate/oss_and_stack/install/
 - ii. Using docker :
<https://www.geeksforgeeks.org/installation-and-starting-the-servers-of-redis-stack-using-docker/>
 - iii. Once installation is done running the command redis-server should start the server:
KEEP THIS TERMINAL OPEN

```

MacOS-MacBook-Pro:~$ ssh root@157.140.2.255 redis-server
15185C:~ # May 2025 15:57:56.479 # oOoOoOoOoOoOoOo Redis is starting oOoOoOoOoOoOo
15185C:~ # May 2025 15:57:56.479 # Redis version=6.0.9, bits=64, commit=00000000, modified=
    15185C:~ # May 2025 15:57:56.479 # Redis started
15185C:~ # May 2025 15:57:56.479 # Warning: no config file specified, using the default con
    go in order to specify a config file use redis-server --path=/path/to/redis.conf
15185M:~ # May 2025 15:57:56.472 # Increased maximum number of open files to 10032 (it was
    originally set to 256).
15185M:~ # May 2025 15:57:56.471 # monotonic clock: POSIX clock_gettime

Redis Open Source
6.0.9 (00000000/1) 64 bit
Running in standalone mode
Port: 6379
PID: 31585

https://redis.io

15185M:~ # May 2025 15:57:56.472 # WARNING: The TCP backlog setting of 511 cannot be enforced
    because kern.ipc.somaxconn is set to the lower value of 128.
15185M:~ # May 2025 15:57:56.472 # Server initialized
15185M:~ # May 2025 15:57:56.474 # Ready to accept connections tcp

```

- b. Configure spring boot project to cache data in redis:

- i. Add below dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

- ii. Add below annotation in the class with main method just below the @SpringBootApplication annotation:

```
@EnableCaching
```

- iii. Update the BookService class methods as follows:

```
@CachePut(value = "books", key = "#book.bookid")
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");
}

@CacheEvict(value = "books", key = "#id")
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");
}

@Cacheable(value = "books", key = "#id")
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");

}
```

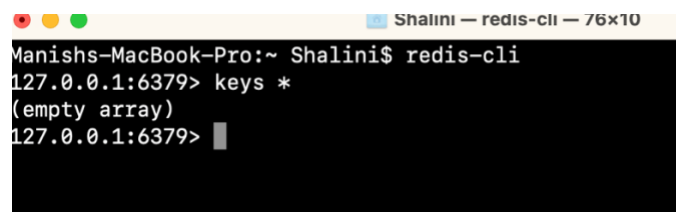
- c. Open another terminal or command prompt and type in below commands as shown:

redis-cli

Once inside redis shell:

keys *

It shows empty array as nothing is cached



```
Manishs-MacBook-Pro:~ Shalini$ redis-cli
127.0.0.1:6379> keys *
(empty array)
127.0.0.1:6379>
```

- d. Restart the server and make a get book by id request as follows:

<http://localhost:8081/books/1> -> GET
<http://localhost:8081/books/2> -> GET

- e. Again type keys * and should see the ids being cached:

```
[127.0.0.1:6379> keys *  
1) "books::2"  
2) "books::1"  
127.0.0.1:6379> █
```

- f. Now make a DELETE HTTP request to <http://localhost:8081/books/2>

Check redis-cli keys and should see the id 2 evicted

```
[127.0.0.1:6379> keys *  
1) "books::1"  
127.0.0.1:6379> █
```
