**Spring Core Maven**

**Exercise 1: Configuring a Basic Spring Application**

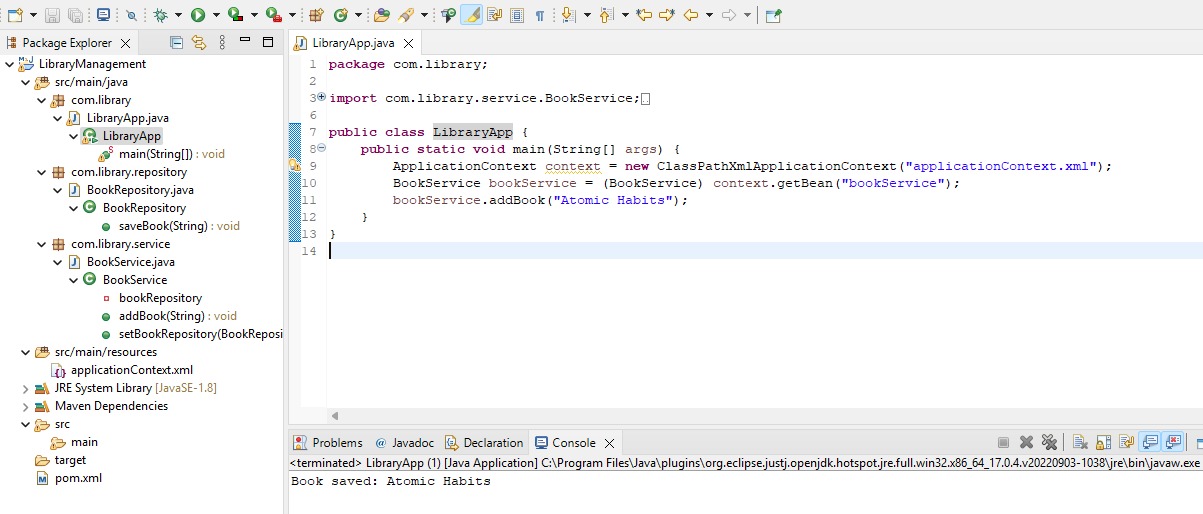
**Scenario:**

Your company is developing a web application for managing a library. You need to use the Spring Framework to handle the backend operations.

**Steps:**

1. **Set Up a Spring Project:**
   * Create a Maven project named **LibraryManagement**.
   * Add Spring Core dependencies in the **pom.xml** file.
2. **Configure the Application Context:**
   * Create an XML configuration file named **applicationContext.xml** in the **src/main/resources** directory.
   * Define beans for **BookService** and **BookRepository** in the XML file.
3. **Define Service and Repository Classes:**
   * Create a package **com.library.service** and add a class **BookService**.
   * Create a package **com.library.repository** and add a class **BookRepository**.
4. **Run the Application:**
   * Create a main class to load the Spring context and test the configuration.

**Output:**



**Exercise 2: Implementing Dependency Injection**

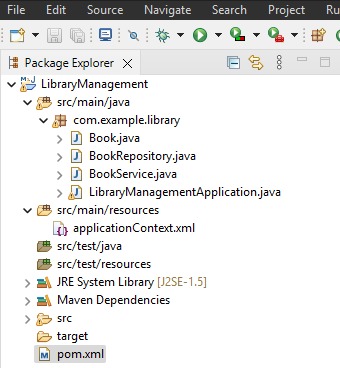
**Scenario:**

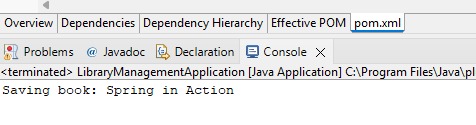
In the library management application, you need to manage the dependencies between the BookService and BookRepository classes using Spring's IoC and DI.

**Steps:**

1. **Modify the XML Configuration:**
   * Update **applicationContext.xml** to wire **BookRepository** into **BookService**.
2. **Update the BookService Class:**
   * Ensure that **BookService** class has a setter method for **BookRepository**.
3. **Test the Configuration:**
   * Run the **LibraryManagementApplication** main class to verify the dependency injection.

**Output:**





**Exercise 3: Implementing Logging with Spring AOP**

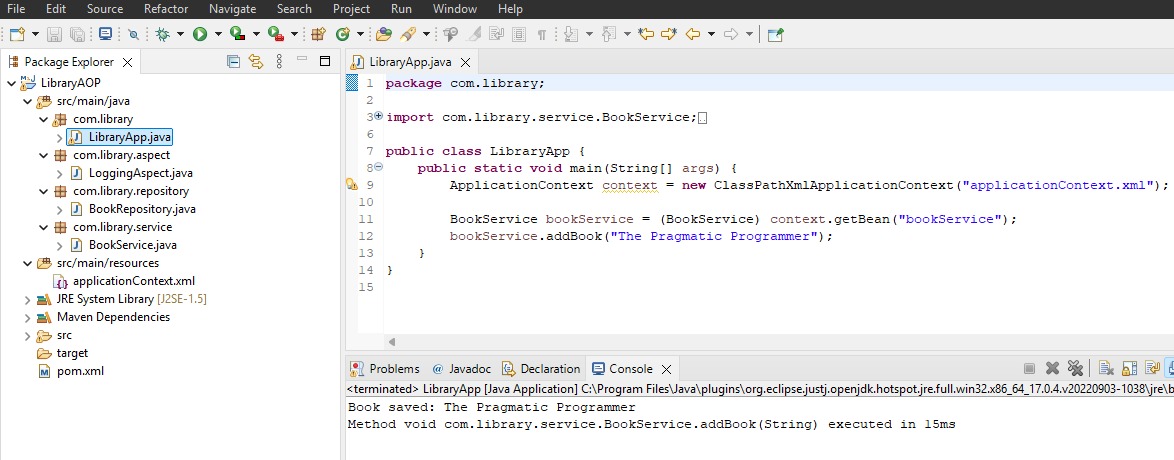
**Scenario:**

The library management application requires logging capabilities to track method execution times.

**Steps:**

1. **Add Spring AOP Dependency:**
   * Update **pom.xml** to include Spring AOP dependency.
2. **Create an Aspect for Logging:**
   * Create a package **com.library.aspect** and add a class **LoggingAspect** with a method to log execution times.
3. **Enable AspectJ Support:**
   * Update **applicationContext.xml** to enable **AspectJ** support and register the aspect.
4. **Test the Aspect:**
   * Run the **LibraryManagementApplication** main class and observe the console for log messages indicating method execution times.

**Output:**



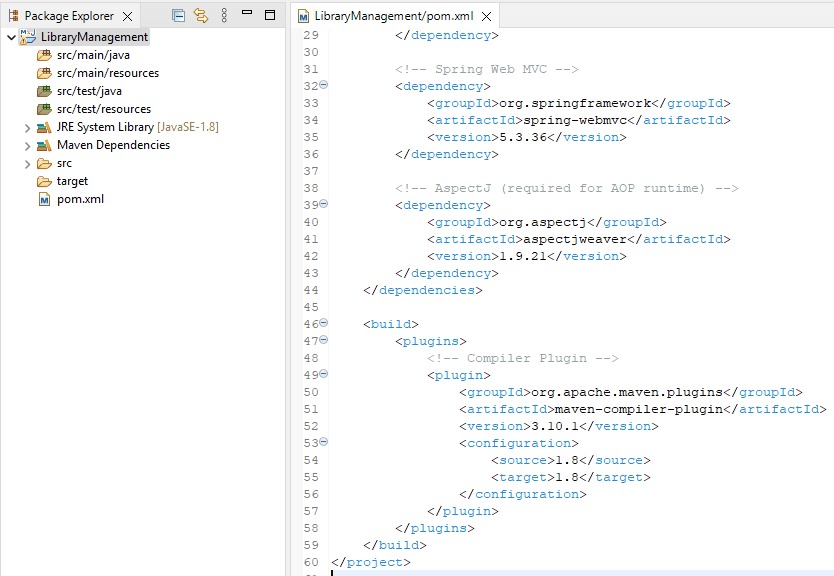
**Exercise 4: Creating and Configuring a Maven Project**

**Scenario:**

You need to set up a new Maven project for the library management application and add Spring dependencies.

**Steps:**

1. **Create a New Maven Project:**
   * Create a new Maven project named **LibraryManagement**.
2. **Add Spring Dependencies in pom.xml:**
   * Include dependencies for Spring Context, Spring AOP, and Spring WebMVC.
3. **Configure Maven Plugins:**
   * Configure the Maven Compiler Plugin for Java version 1.8 in the pom.xml file.

**Output:**

**Exercise 5: Configuring the Spring IoC Container**

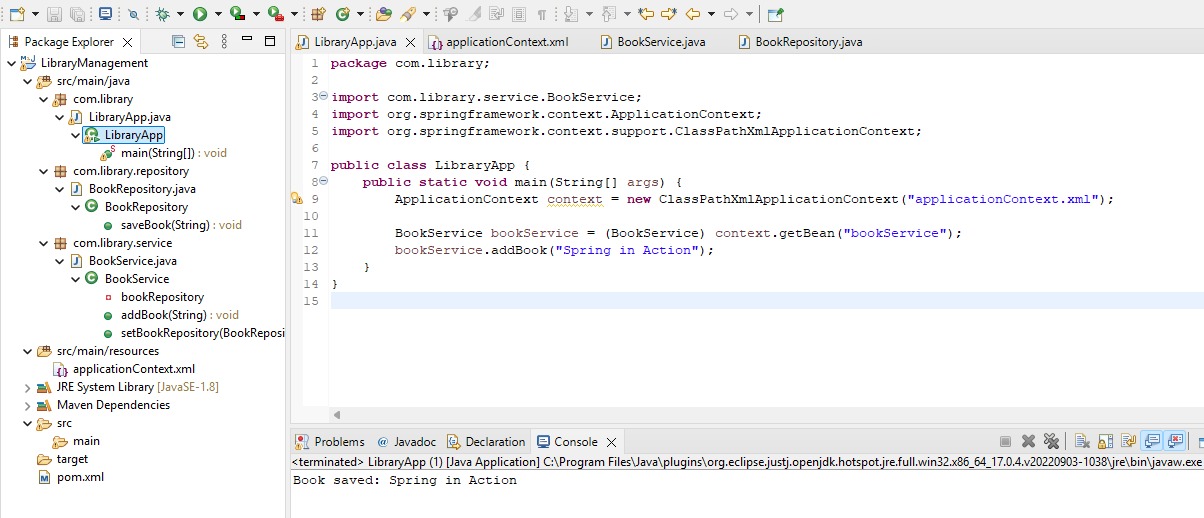
**Scenario:**

The library management application requires a central configuration for beans and dependencies.

**Steps:**

1. **Create Spring Configuration File:**
   * Create an XML configuration file named **applicationContext.xml** in the **src/main/resources** directory.
   * Define beans for **BookService** and **BookRepository** in the XML file.
2. **Update the BookService Class:**
   * Ensure that the **BookService** class has a setter method for **BookRepository**.
3. **Run the Application:**
   * Create a main class to load the Spring context and test the configuration.

**Output:**



**Exercise 6: Configuring Beans with Annotations**

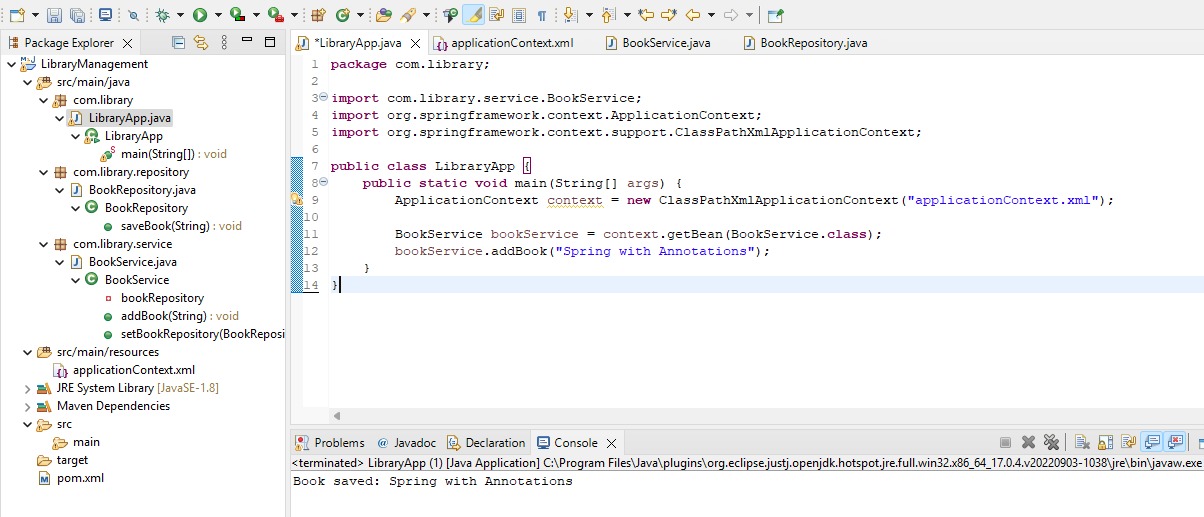
**Scenario:**

You need to simplify the configuration of beans in the library management application using annotations.

**Steps:**

1. **Enable Component Scanning:**
   * Update **applicationContext.xml** to include component scanning for the **com.library** package.
2. **Annotate Classes:**
   * Use **@Service** annotation for the **BookService** class.
   * Use **@Repository** annotation for the **BookRepository** class.
3. **Test the Configuration:**
   * Run the **LibraryManagementApplication** main class to verify the annotation-based configuration.

**Output:**



**Exercise 7: Implementing Constructor and Setter Injection**

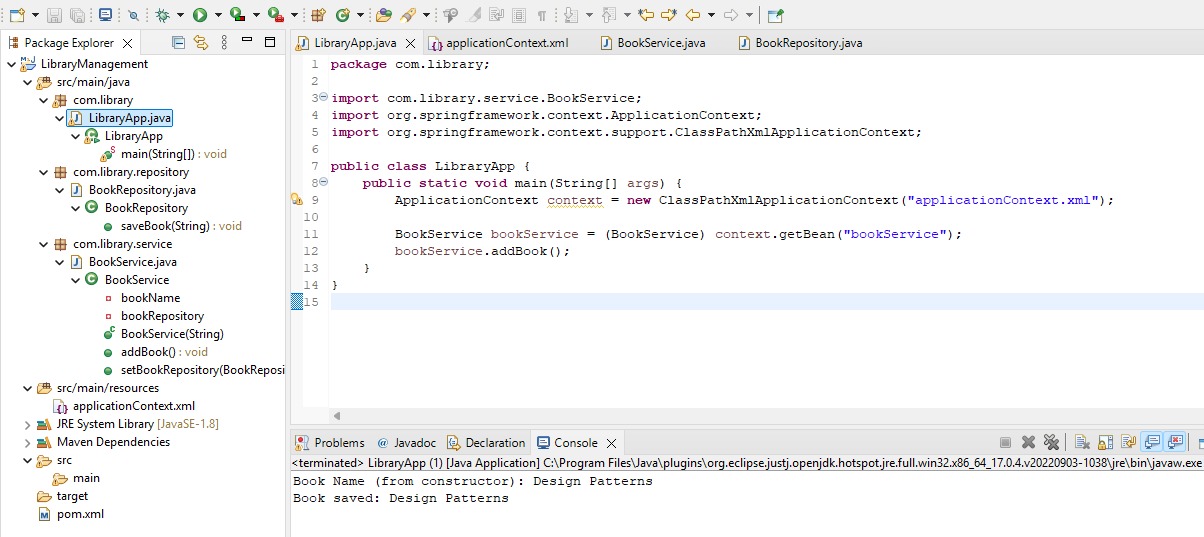
**Scenario:**

The library management application requires both constructor and setter injection for better control over bean initialization.

**Steps:**

1. **Configure Constructor Injection:**
   * Update applicationContext.**xml** to configure constructor injection for **BookService**.
2. **Configure Setter Injection:**
   * Ensure that the **BookService** class has a setter method for **BookRepository** and configure it in **applicationContext.xml**.
3. **Test the Injection:**
   * Run the **LibraryManagementApplication** main class to verify both constructor and setter injection.

**Output:**



**Exercise 8: Implementing Basic AOP with Spring**

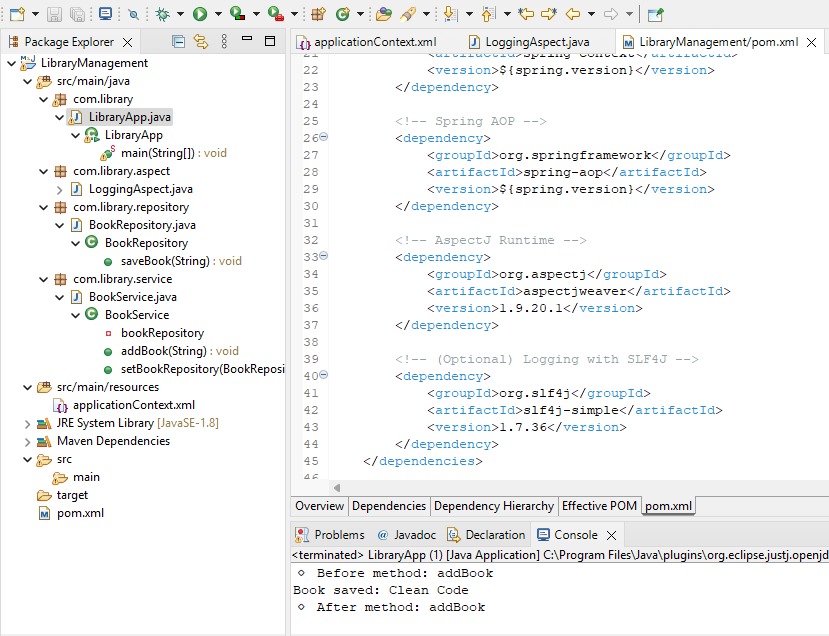
**Scenario:**

The library management application requires basic AOP functionality to separate cross-cutting concerns like logging and transaction management.

**Steps:**

1. **Define an Aspect:**
   * Create a package **com.library.aspect** and add a class **LoggingAspect**.
2. **Create Advice Methods:**
   * Define advice methods in **LoggingAspect** for logging before and after method execution.
3. **Configure the Aspect:**
   * Update **applicationContext.xml** to register the aspect and enable **AspectJ** auto-proxying.
4. **Test the Aspect:**
   * Run the **LibraryManagementApplication** main class to verify the AOP functionality.

**Output:**



**Exercise 9: Creating a Spring Boot Application**

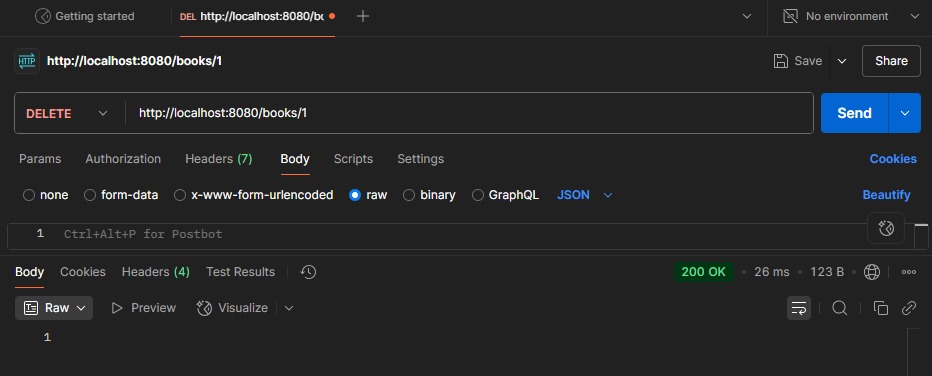
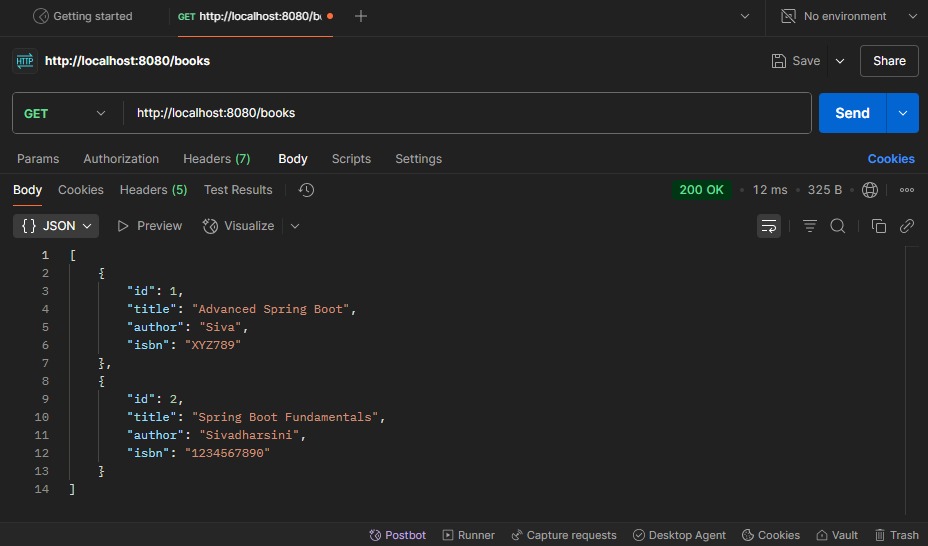
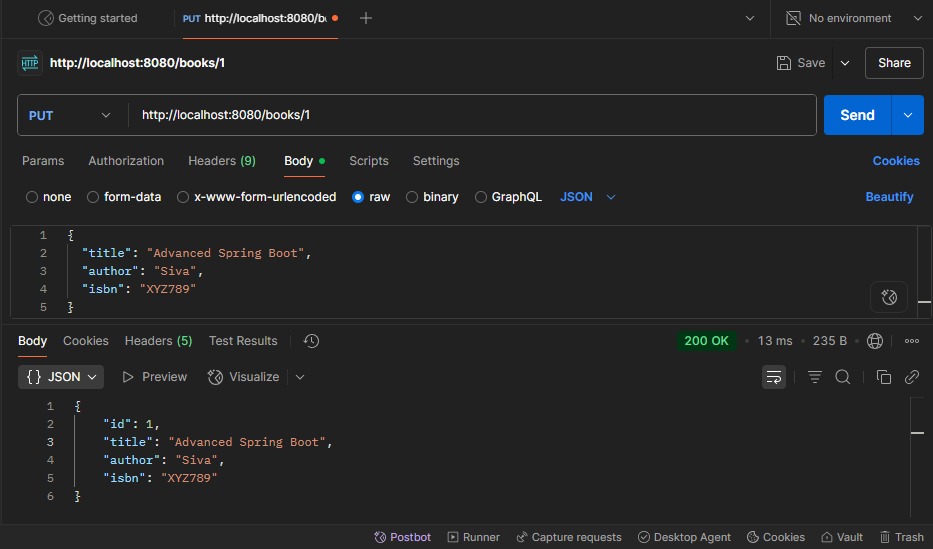
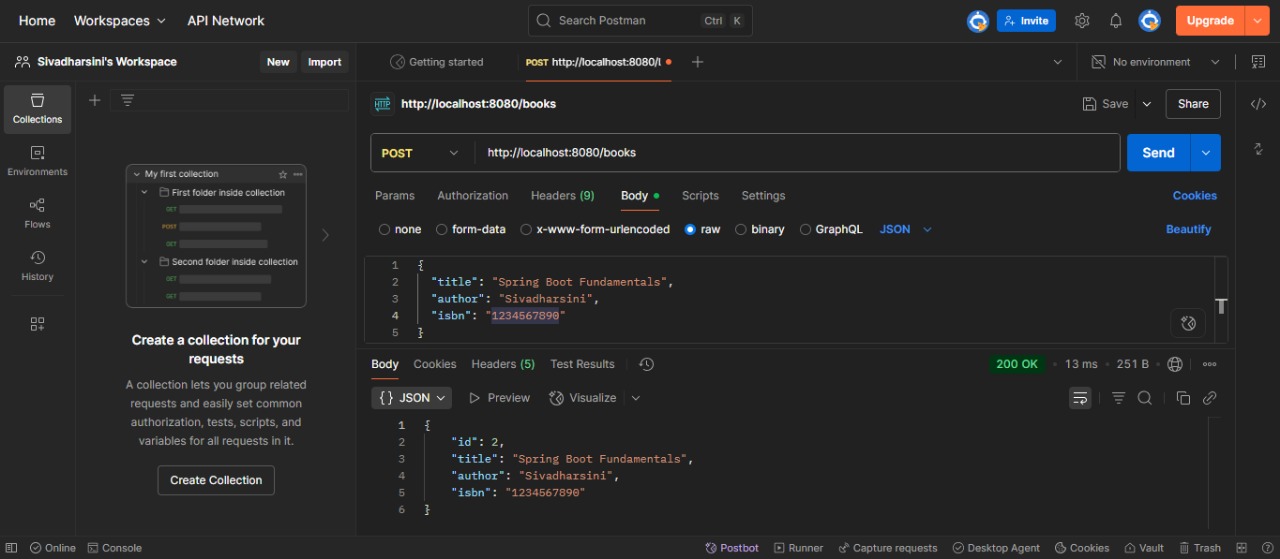
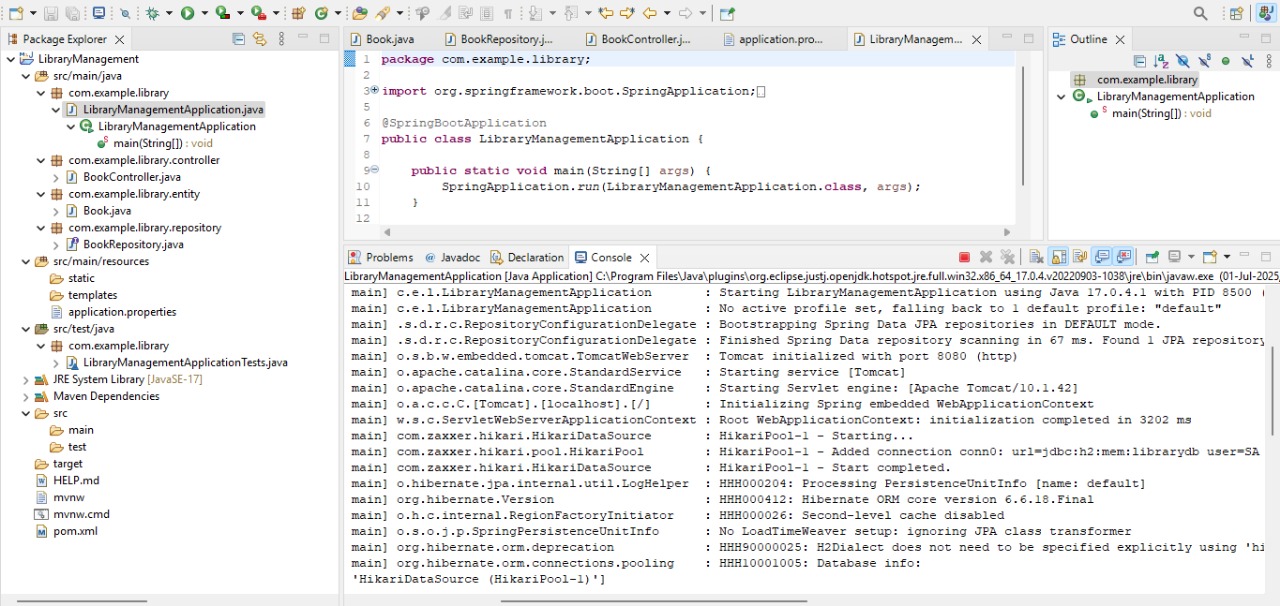
**Scenario:**

You need to create a Spring Boot application for the library management system to simplify configuration and deployment.

**Steps:**

1. **Create a Spring Boot Project:**
   * Use **Spring Initializr** to create a new Spring Boot project named **LibraryManagement**.
2. **Add Dependencies:**
   * Include dependencies for **Spring Web, Spring Data JPA, and H2 Database**.
3. **Create Application Properties:**
   * Configure database connection properties in **application.properties**.
4. **Define Entities and Repositories:**
   * Create **Book** entity and **BookRepository** interface.
5. **Create a REST Controller:**
   * Create a **BookController** class to handle CRUD operations.
6. **Run the Application:**
   * Run the Spring Boot application and test the REST endpoints.

**Output:**



**Spring REST Using Spring Boot**

1.Spring-rest-handson

**Hands on 1**

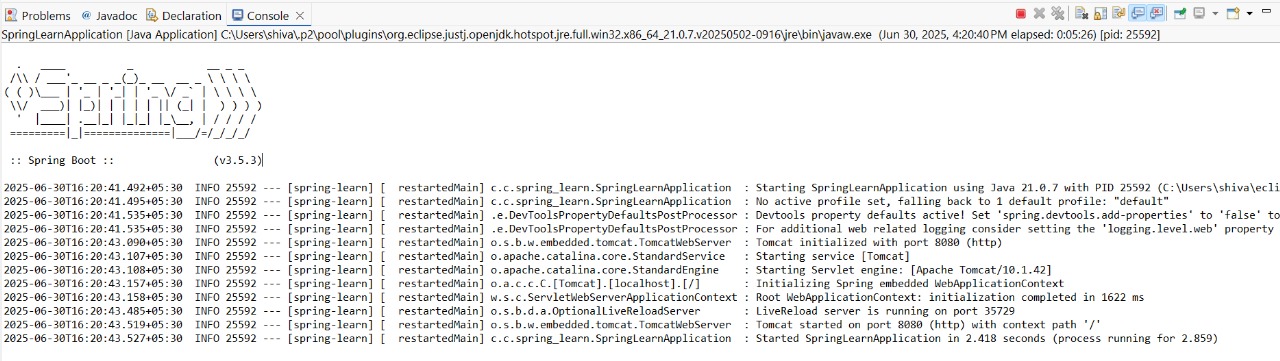
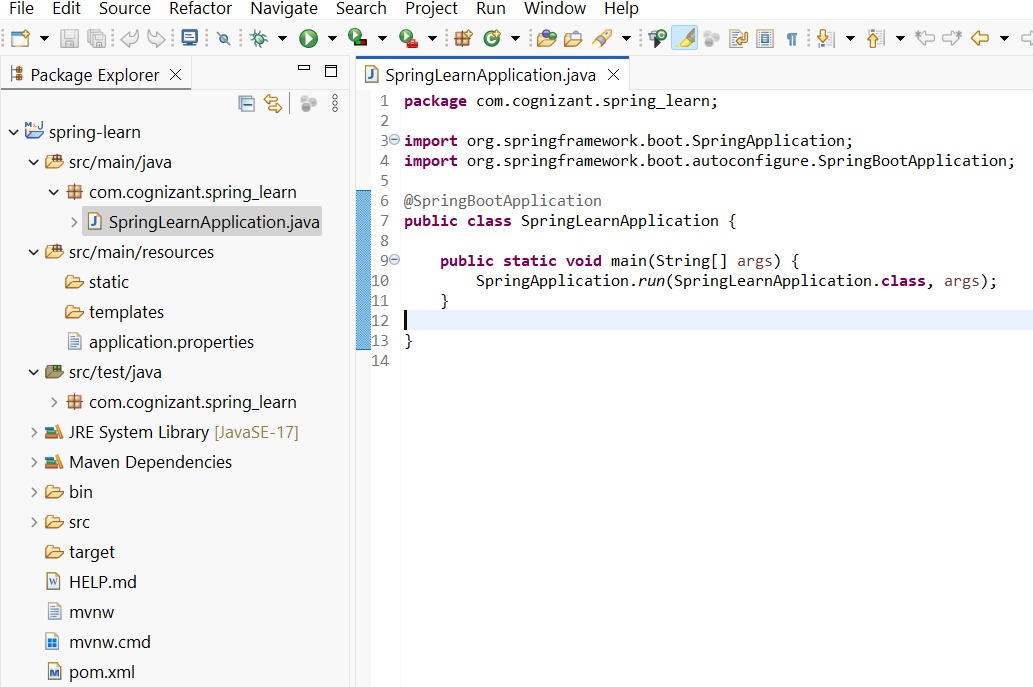
**Create a Spring Web Project using Maven**   
   
Follow steps below to create a project:

1. Go to <https://start.spring.io/>
2. Change Group as “com.cognizant”
3. Change Artifact Id as “spring-learn”
4. Select Spring Boot DevTools and Spring Web
5. Create and download the project as zip
6. Extract the zip in root folder to Eclipse Workspace
7. Build the project using ‘mvn clean package -Dhttp.proxyHost=proxy.cognizant.com -Dhttp.proxyPort=6050 -Dhttps.proxyHost=proxy.cognizant.com -Dhttps.proxyPort=6050 -Dhttp.proxyUser=123456’ command in command line
8. Import the project in Eclipse "File > Import > Maven > Existing Maven Projects > Click Browse and select extracted folder > Finish"
9. Include logs to verify if main() method of SpringLearnApplication.
10. Run the SpringLearnApplication class.

SME to walk through the following aspects related to the project created:

1. src/main/java - Folder with application code
2. src/main/resources - Folder for application configuration
3. src/test/java - Folder with code for testing the application
4. SpringLearnApplication.java - Walkthrough the main() method.
5. Purpose of @SpringBootApplication annotation
6. pom.xml
   1. Walkthrough all the configuration defined in XML file
   2. Open 'Dependency Hierarchy' and show the dependency tree.

**Output:**

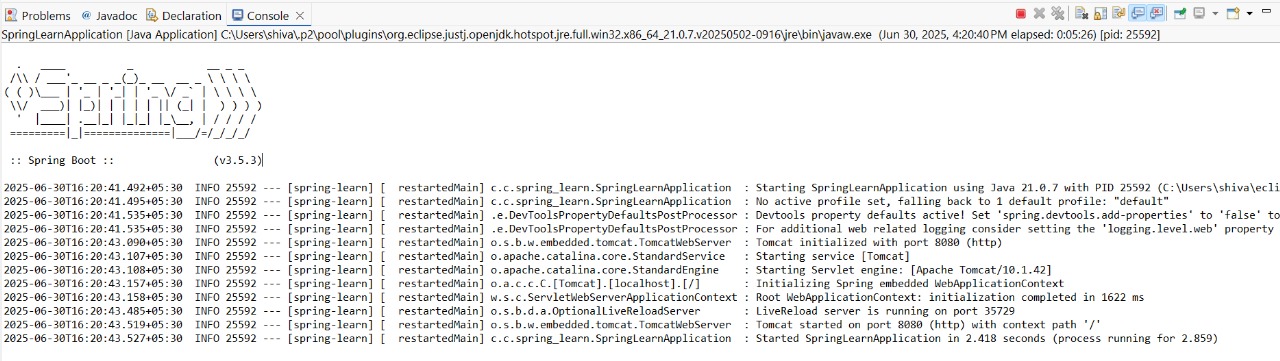


**Hands on 2**

**Spring Core – Load SimpleDateFormat from Spring Configuration XML**   
   
SimpleDateFormat with the pattern ‘dd/MM/yyyy’ is created in multiple places of an application. To avoid creation of SimpleDateFormat in multiple places, define a bean in Spring XML Configuration file and retrieve the date.  
   
Follow steps below to implement:

* Create spring configuration file date-format.xml in src/main/resources folder of 'spring-learn' project
* Open <https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-factory-metadata>
* Copy the XML defined in the section of previous step URL and paste it into date-format.xml
* Define bean tag in the XML with for date format.
* Create new method displayDate() in SpringLearnApplication.java
* In displayDate() method create the ApplicationContext.
* Get the dateFormat using getBean() method.
* Using the format variable try to parse string '31/12/2018' to Date class and display the result using System.out.println.
* Run the application as 'Java Application' and check the result in console log output.

**Output:**

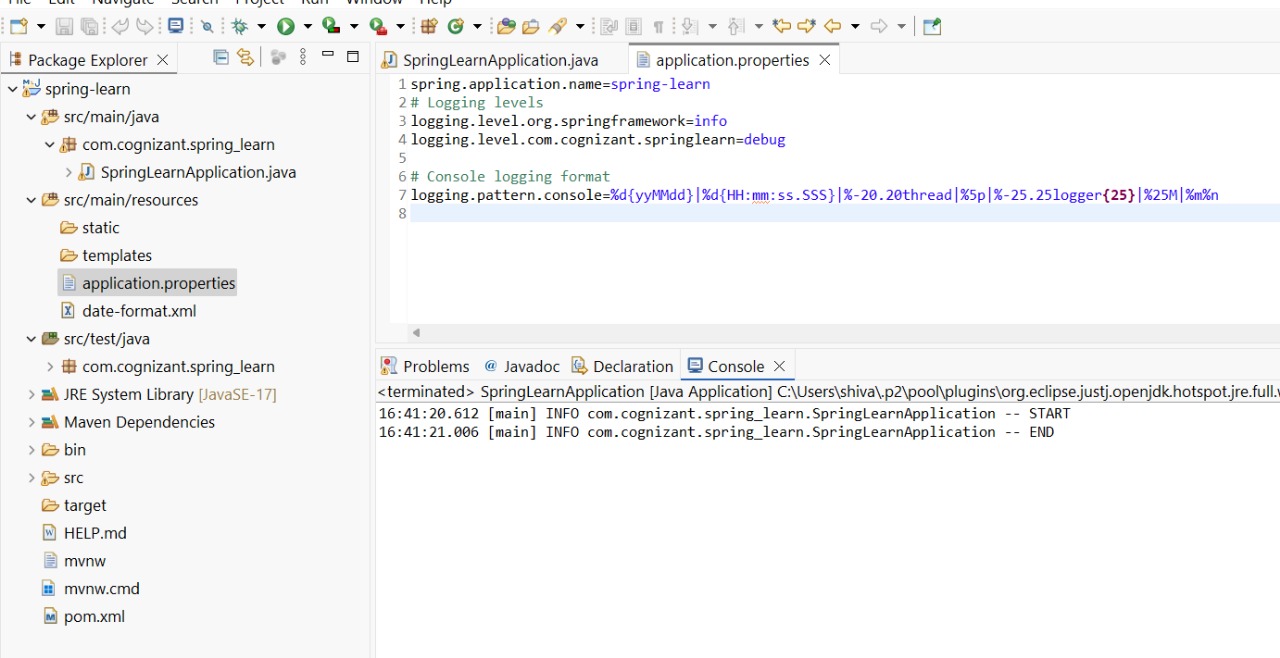


**Hands on 3**

**Spring Core - Incorporate Logging**   
   
Incorporate logging in the Spring Boot project created in previous hands on. Refer steps below:

* Create application.properties if not yet created in src/main/resources folder
* Add below lines in application.properties
* In SpringLearnApplication.java include the following imports:
* Include the below static variable in SpringLearnApplication.java:
* Include info log on start and end of method. Debug log for displaying the date (refer code below)

**Output:**



**Hands on 4**

**Spring Core – Load Country from Spring Configuration XML**   
   
An airlines website is going to support booking on four countries. There will be a drop down on the home page of this website to select the respective country. It is also important to store the two-character ISO code of each country.

|  |  |
| --- | --- |
| **Code** | **Name** |
| US | United States |
| DE | Germany |
| IN | India |
| JP | Japan |

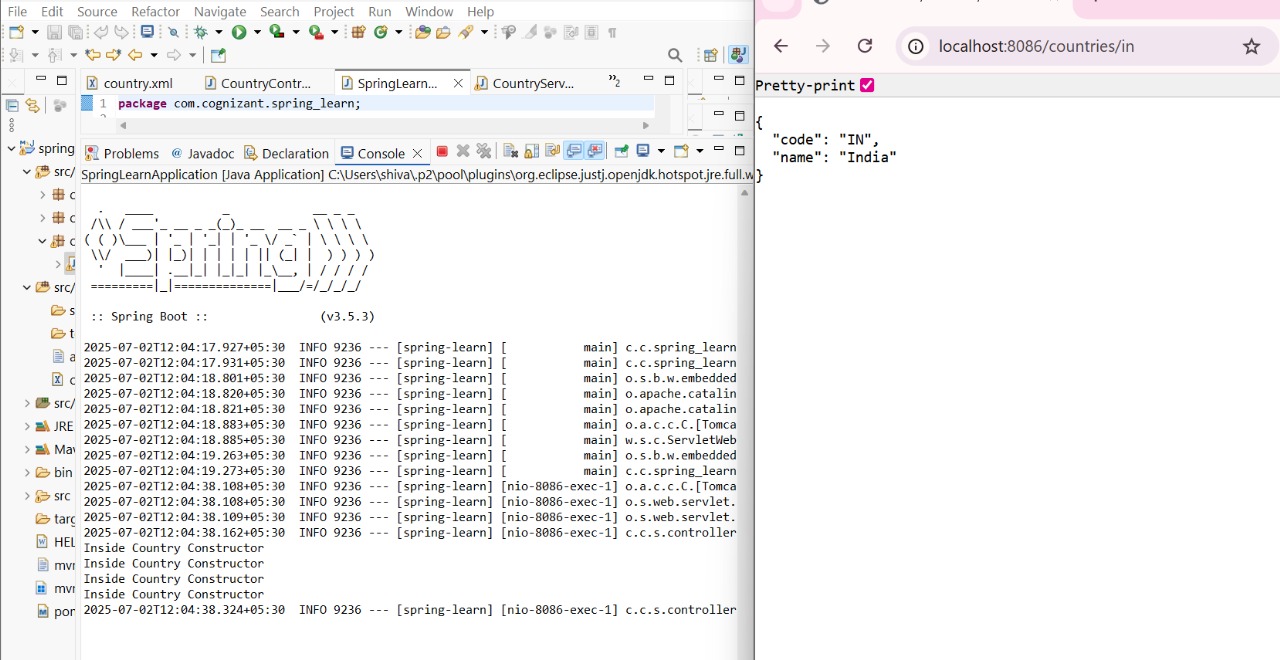
Above data has to be stored in spring configuration file. Write a program to read this configuration file and display the details.  
   
Steps to implement

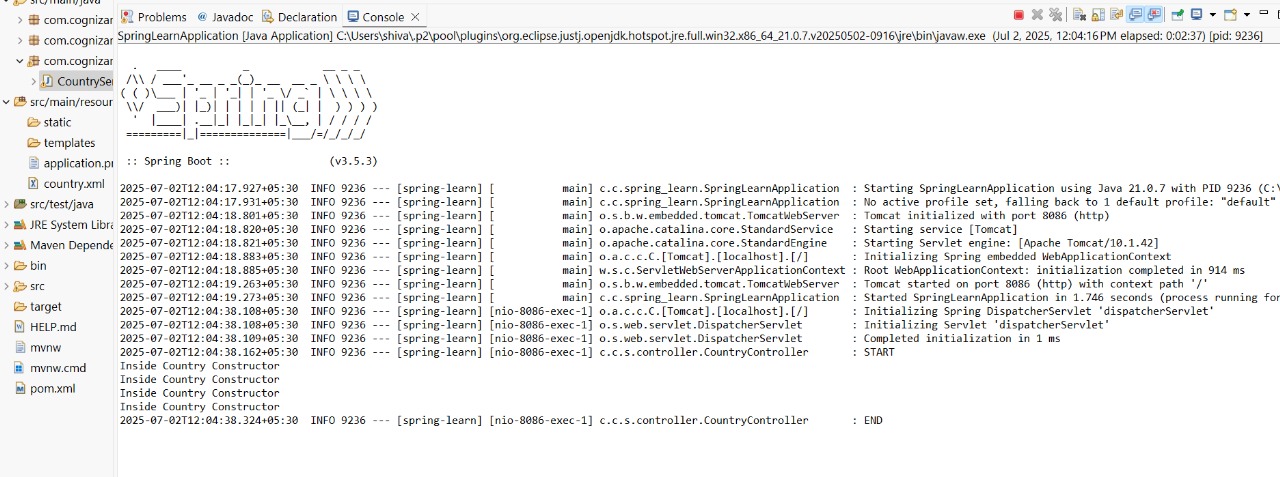
* Pick any one of your choice country to configure in Spring XML configuration named country.xml.
* Create a bean tag in spring configuration for country and set the property and values.
* Create Country class with following aspects:
  + Instance variables for code and name
  + Implement empty parameter constructor with inclusion of debug log within the constructor with log message as “Inside Country Constructor.”
  + Generate getters and setters with inclusion of debug with relevant message within each setter and getter method.
  + Generate toString() method
* Create a method displayCountry() in SpringLearnApplication.java, which will read the country bean from spring configuration file and display the country details. ClassPathXmlApplicationContext, ApplicationContext and context.getBean(“beanId”, Country.class). Refer sample code for displayCountry() method below.
* Invoke displayCountry() method in main() method of SpringLearnApplication.java.
* Execute main() method and check the logs to find out which constructors and methods were invoked.

SME to provide more detailing about the following aspects:

* bean tag, id attribute, class attribute, property tag, name attribute, value attribute
* ApplicationContext, ClassPathXmlApplicationContext
* What exactly happens when context.getBean() is invoked

**Output:**





**Hands on 5**

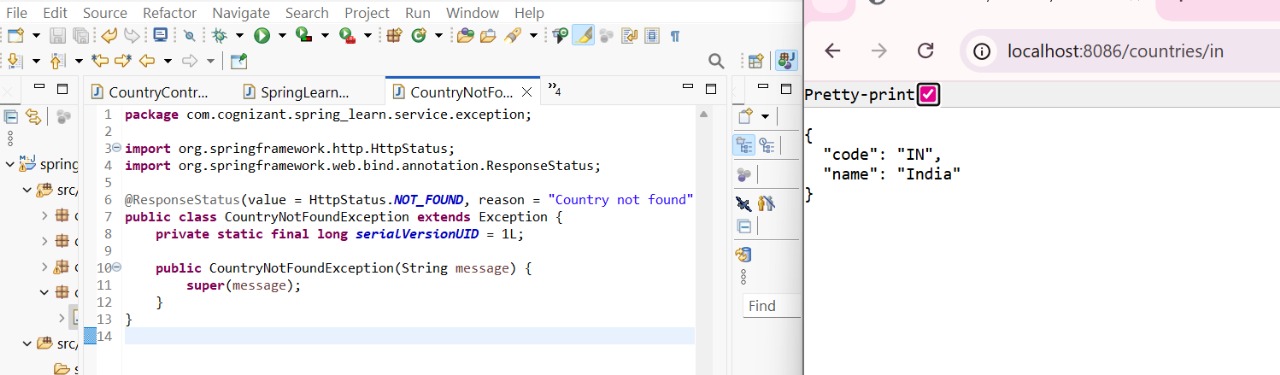
**Spring Core – Demonstration of Singleton Scope and Prototype Scope**   
   
The Country bean done in the previous hands on will be used to demonstrate the scopes in Spring. Implement the steps below.  
   
**Follow steps below to demonstrate Singleton Scope**

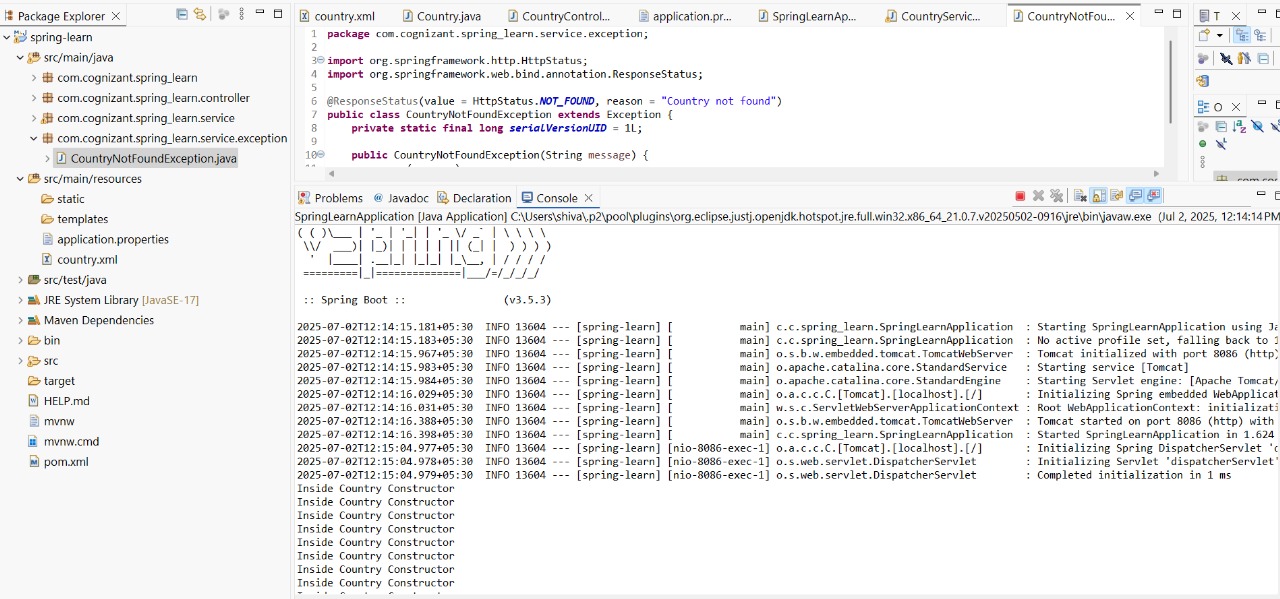
* Include a line in displayCountry() to get country bean reference one more time from the same application context. Only the third line of the below code snippet should be copied and pasted.
* The constructor will be called only once, which means that only one instance of Country bean is created

**Follow steps below to demonstrate Prototype Scope**

* Include scope="prototype" attribute in bean definition xml.
* Run the application
* Constructor will be called twice, which means that two instances of country is created.

**Output:**





**Hands on 6**

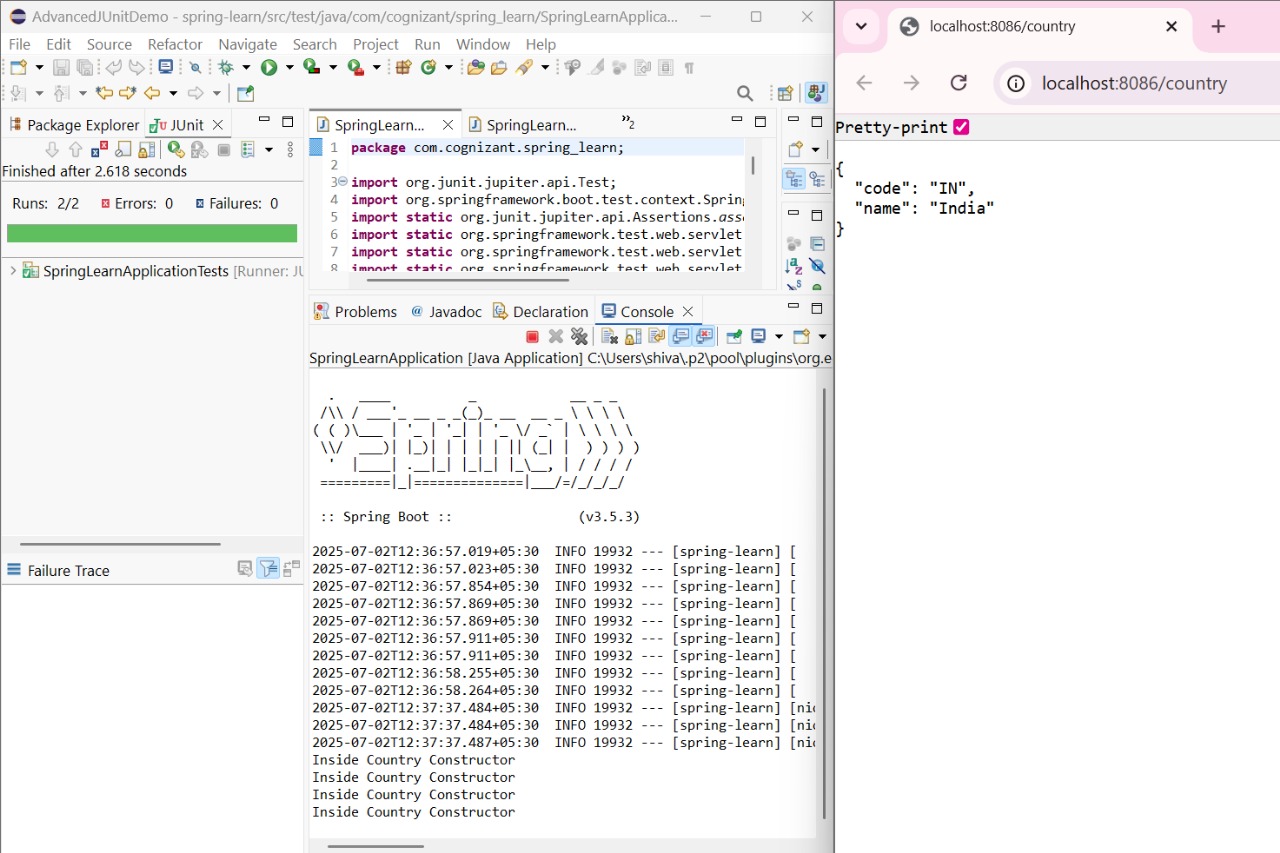
**Spring Core – Load list of countries from Spring Configuration XML**   
   
Our main objective was to retrieve the list of four countries for the airlines website. Refer steps below to get this incorporated.

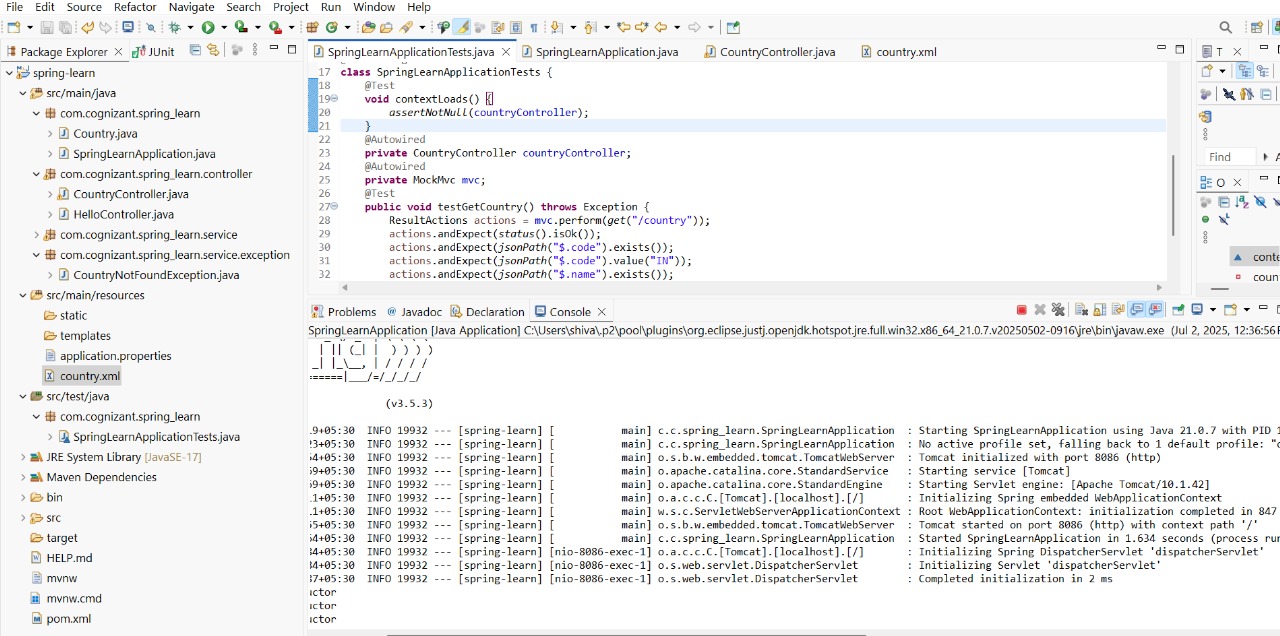
* Create a separate bean for each of the four country in country.xml.
* Create an ArrayList of Country in country.xml. Refer code below.
* Include new method displayCountries() in SpringLearnApplication.java
* In displayCountries() read the country list created above
* Display the list of countries as debug log.

SME to provide detailing on below aspects:

* <list>
* <ref>
* bean attribute

**Output:**



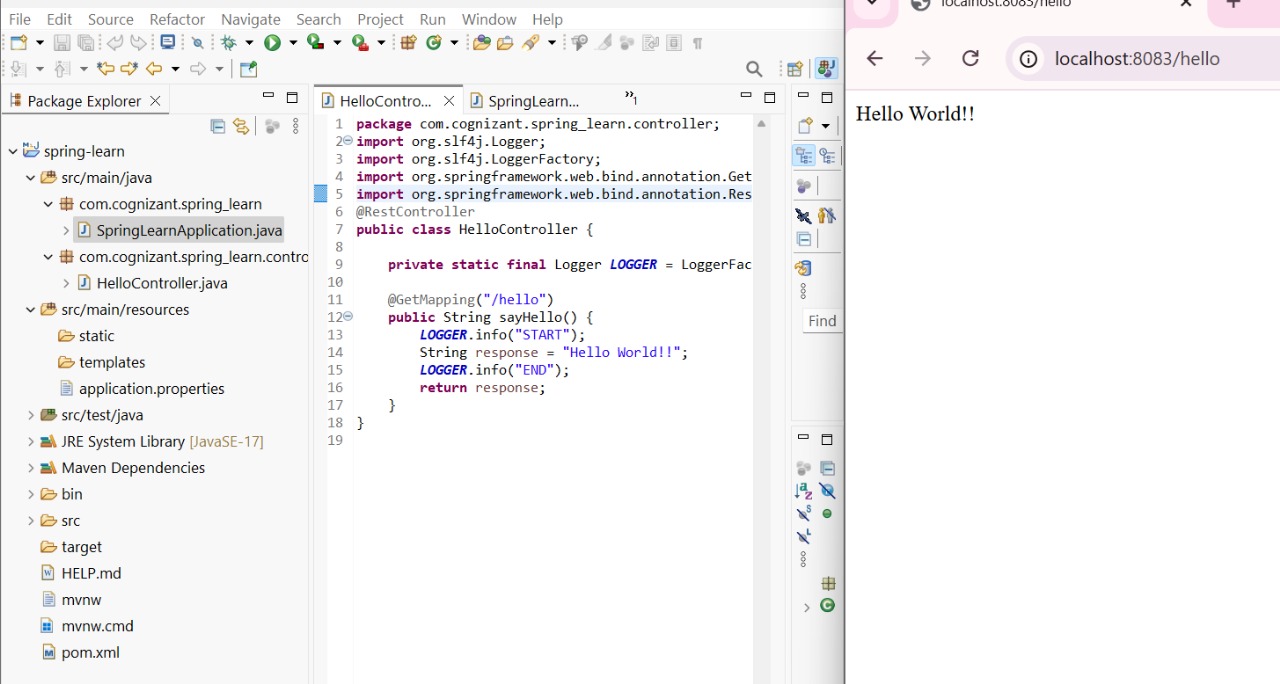


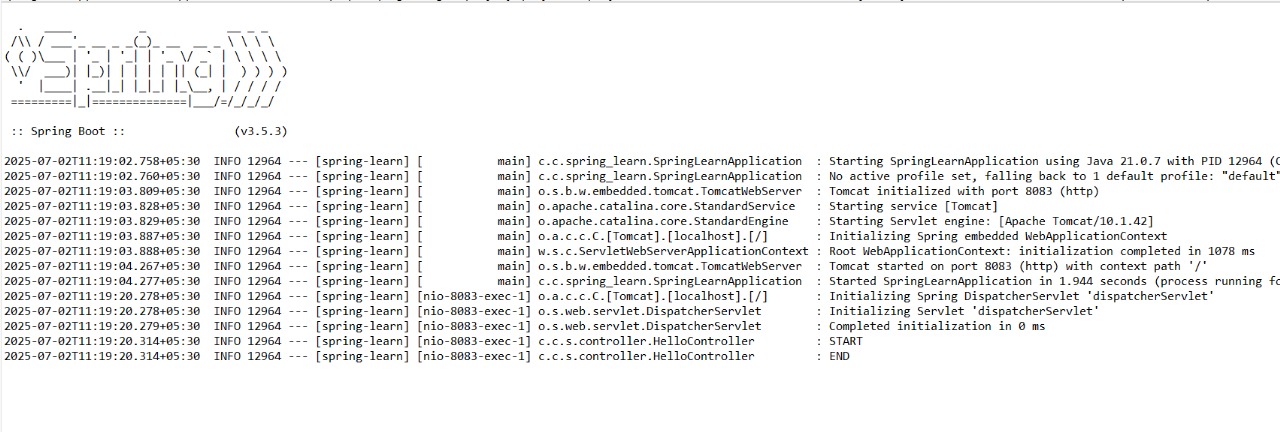
2.Spring-rest-handson  
   
**Hello World RESTful Web Service**   
   
Write a REST service in the spring learn application created earlier, that returns the text "Hello World!!" using Spring Web Framework. Refer details below:  
   
**Method:** GET  
**URL:** /hello  
**Controller:** com.cognizant.spring-learn.controller.HelloController  
**Method Signature:** public String sayHello()  
**Method Implementation:** return hard coded string "Hello World!!"  
**Sample Request**: http://localhost:8083/hello  
**Sample Response:** Hello World!!

Try the URL <http://localhost:8083/hello> in both chrome browser and postman.  
   
SME to explain the following aspects:

* In network tab of developer tools show the HTTP header details received
* In postman click on "Headers" tab to view the HTTP header details received

**Output:**





**REST - Country Web Service**   
   
Write a REST service that returns India country details in the earlier created spring learn application.  
   
**URL**: /country  
**Controller**: com.cognizant.spring-learn.controller.CountryController  
**Method Annotation**: @RequestMapping  
**Method Name**: getCountryIndia()  
**Method Implementation**: Load India bean from spring xml configuration and return  
**Sample Request**: <http://localhost:8083/country>**Sample Response**:

{

"code": "IN",

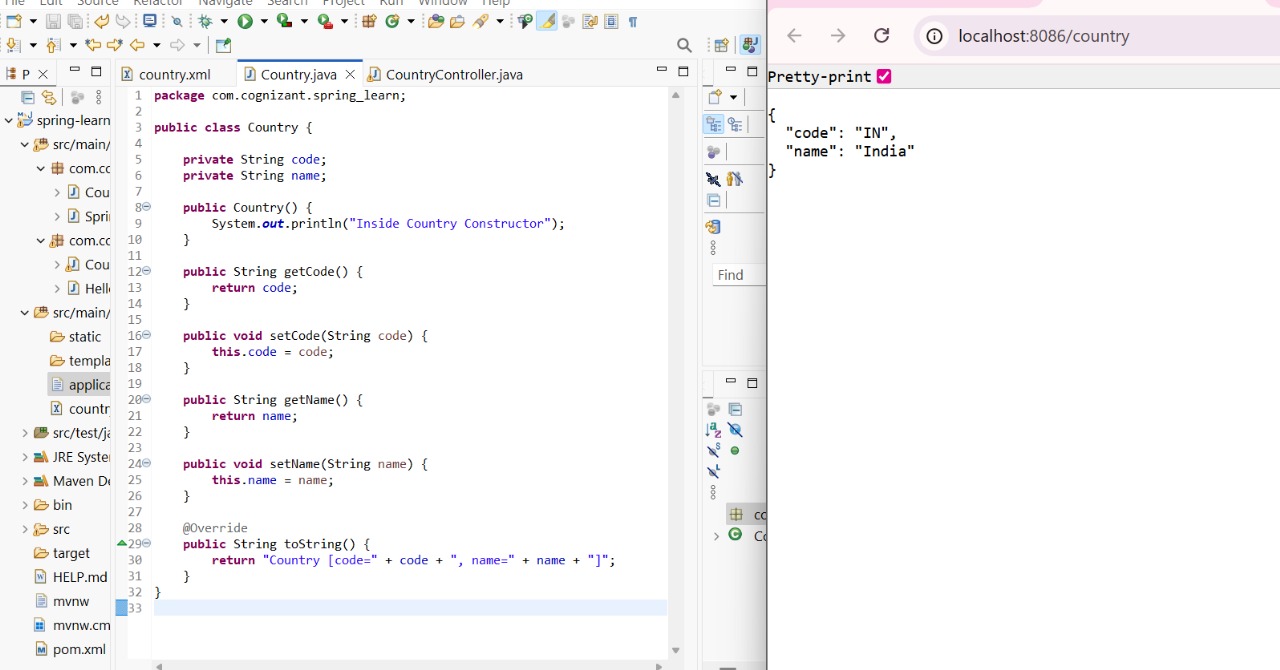
"name": "India"

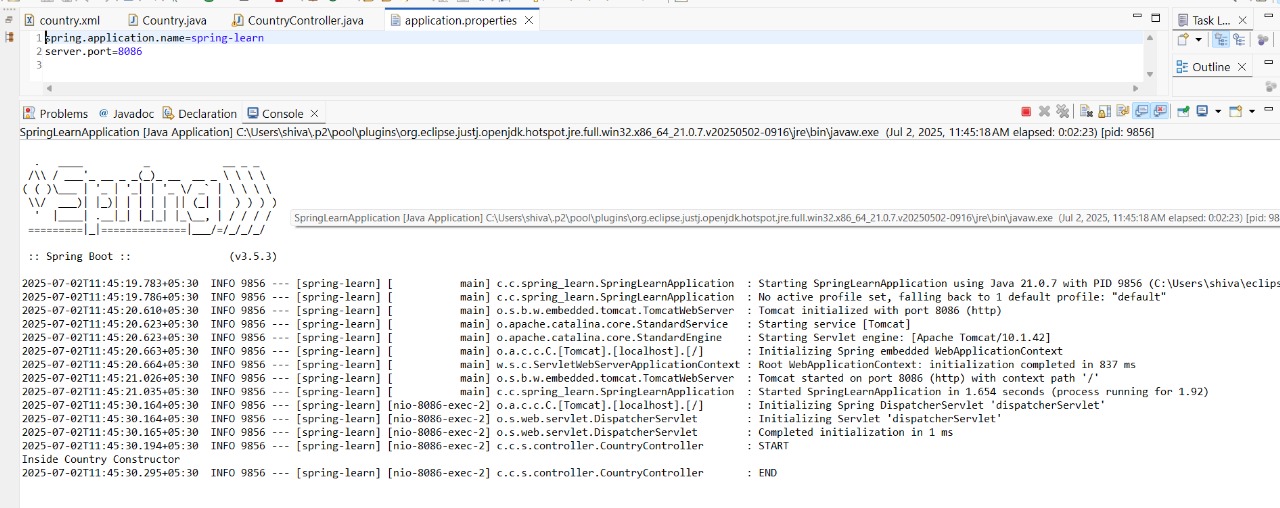
}

SME to explain the following aspects:

* What happens in the controller method?
* How the bean is converted into JSON reponse?
* In network tab of developer tools show the HTTP header details received
* In postman click on "Headers" tab to view the HTTP header details received

**Output:**





**REST - Get all countries**   
   
Write a REST service that returns all the countries.  
   
**Controller**: com.cognizant.spring-learn.controller.CountryController  
**Method Annotation**: @GetMapping("/countries")  
**Method Name**: getAllCountries()  
**Method Implementation**: Load country list from country.xml and return  
**Sample Request**: <http://localhost:8083/countries>**Sample Response**:

[

{ "code": "IN", "name": "India"},

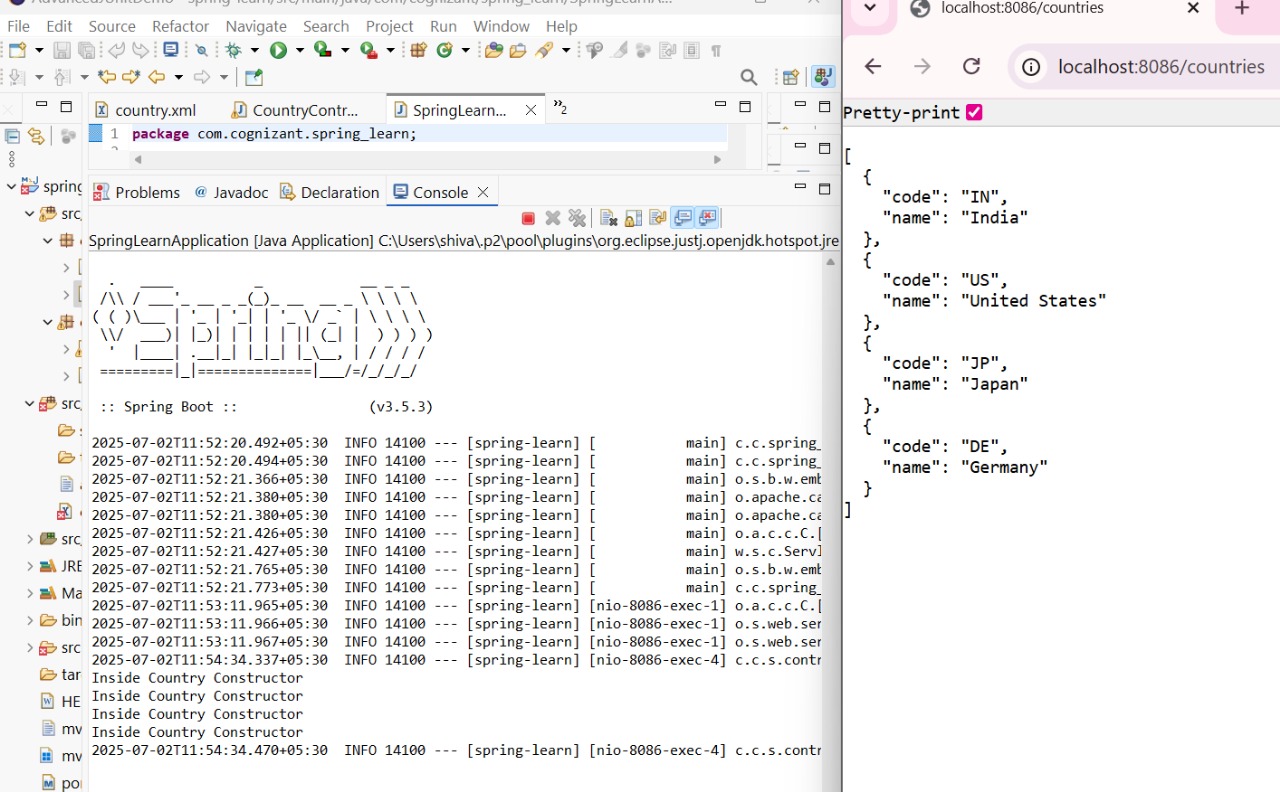
{ "code": "US", "name": "United States"},

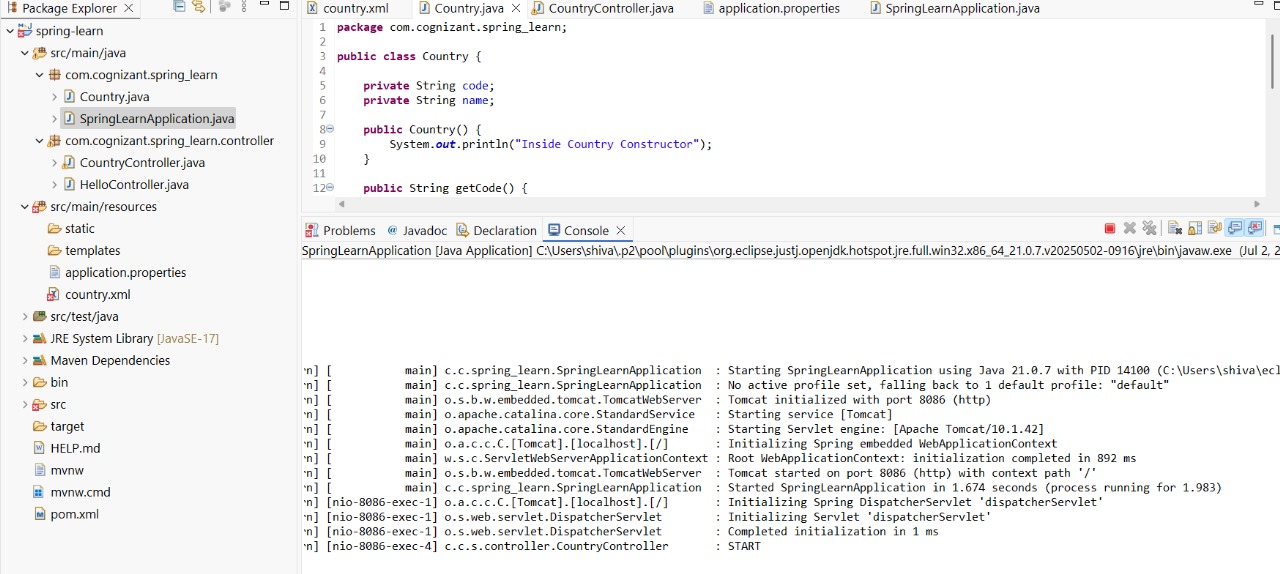
{ "code": "JP", "name": "Japan"},

{ "code": "DE", "name": "Germany"}

]

**Output:**





**REST - Get country based on country code**   
   
Write a REST service that returns a specific country based on country code. The country code should be case insensitive.  
   
**Controller**: com.cognizant.spring-learn.controller.CountryController  
 **Method Annotation:** @GetMapping("/countries/{code}")  
 **Method Name**: getCountry(String code)  
 **Method Implemetation**: Invoke countryService.getCountry(code)   
 **Service Method:** com.cognizant.spring-learn.service.CountryService.getCountry(String code)  
   
**Service Method Implementation**:

* Get the country code using @PathVariable
* Get country list from country.xml
* Iterate through the country list
* Make a case insensitive matching of country code and return the country.
* Lambda expression can also be used instead of iterating the country list

**Sample Request**: <http://localhost:8083/country/in>   
**Sample Response**:

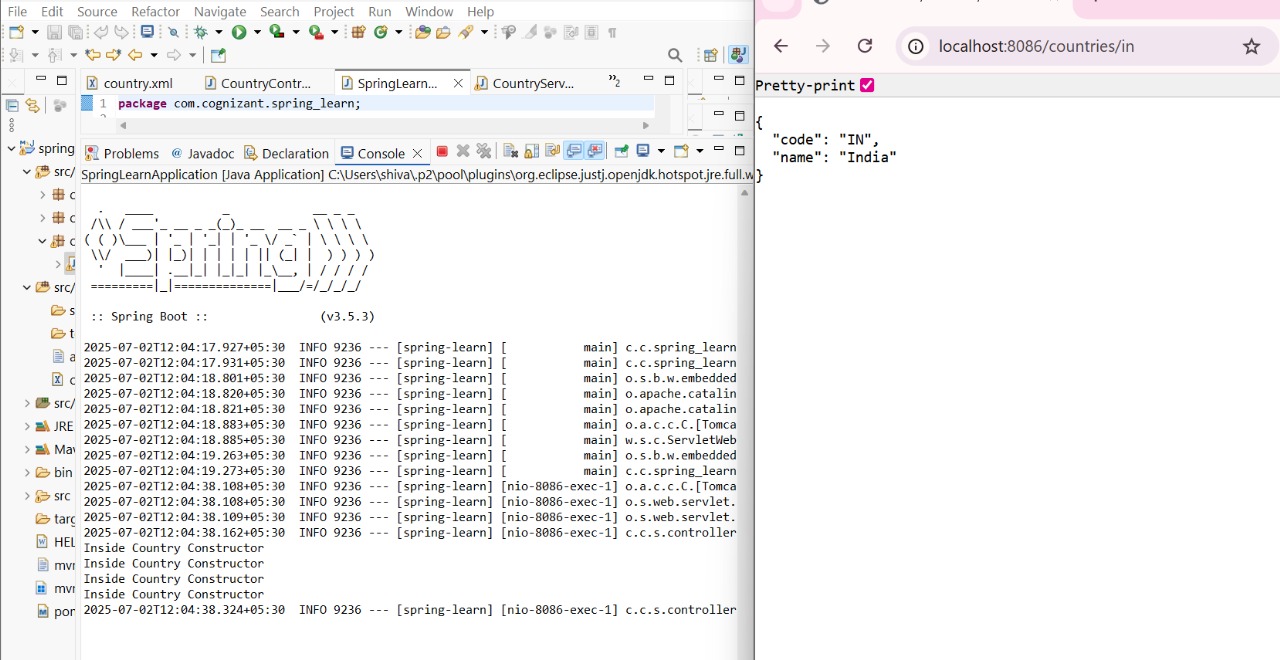
{

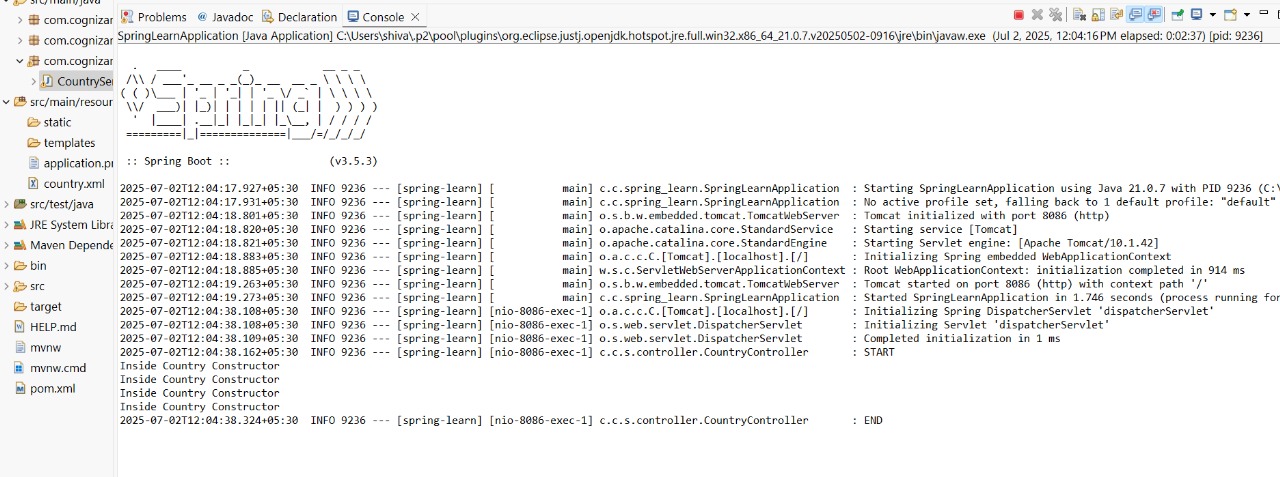
"code": "IN",

"name": "India"

}

**Output:**





**REST - Get country exceptional scenario**   
   
In the previous hands on where we implemented getting country based on country code, what happens if the country code provided in the URL is not present.  
   
**Refer steps below to implement**

* Create a new exception class com.cognizant.springlearn.service.exception.CountryNotFoundException
* Include below specified annotation at the class level in CountryNotFoundException class

@ResponseStatus(value = HttpStatus.NOT\_FOUND, reason = "Country not found")

* In CountryService.getCountry() method include the logic to throw CountryNotFoundException if the country code does not exists in the list.
* In CountryController.getCountry() method include throws clause in method signature. This will respond to the caller of the web service with appropriate error message in JSON format.
* Test the service in postman and using curl command. Refer below for executing curl command.

**Steps to invoke RESTful Web Service using curl command**

* Open Git Bash
* Execute the below command

curl -i <http://localhost:8090/country/az>

**Sample Request**: <http://localhost:8083/country/az>   
**Sample Response**:

{

"timestamp": "2019-10-02T03:27:54.521+0000",

"status": 404,

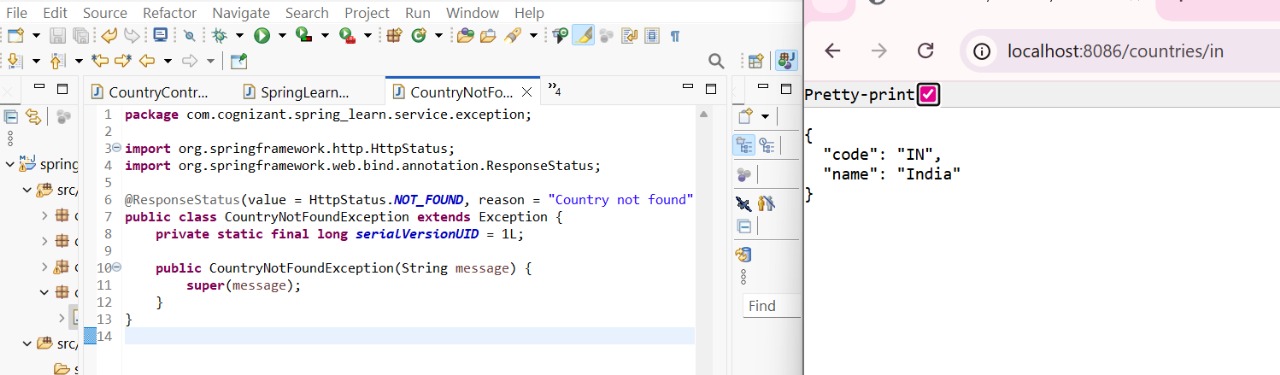
"error": "Not Found",

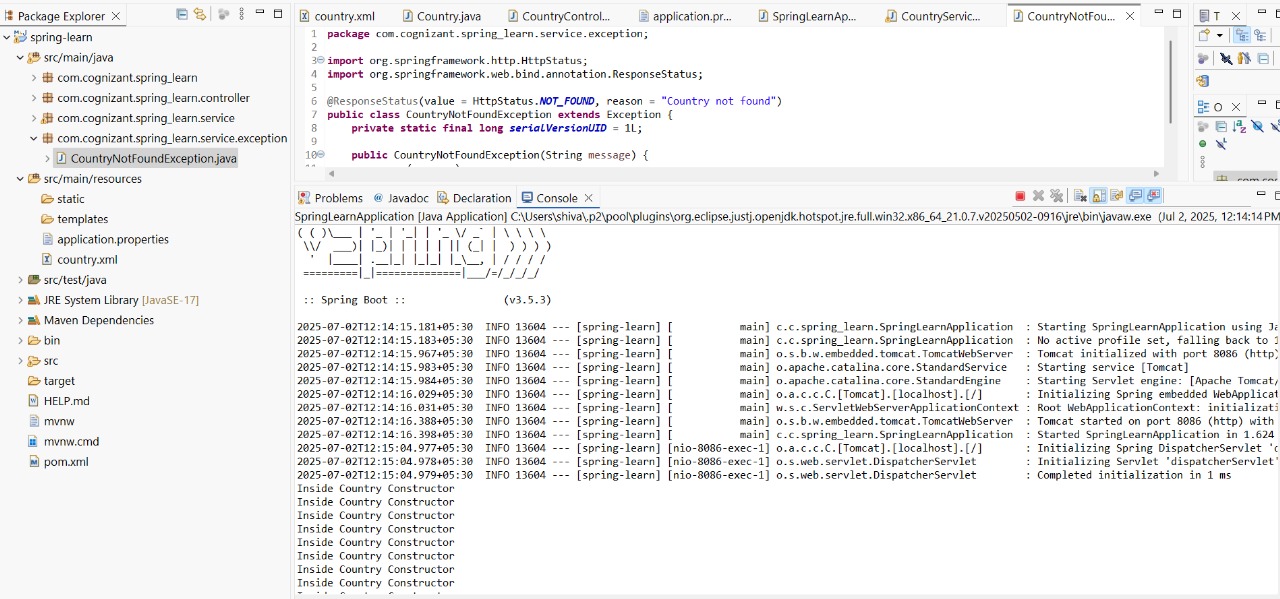
"message": "Country not found",

"path": "/country/az"

}

**Output:**

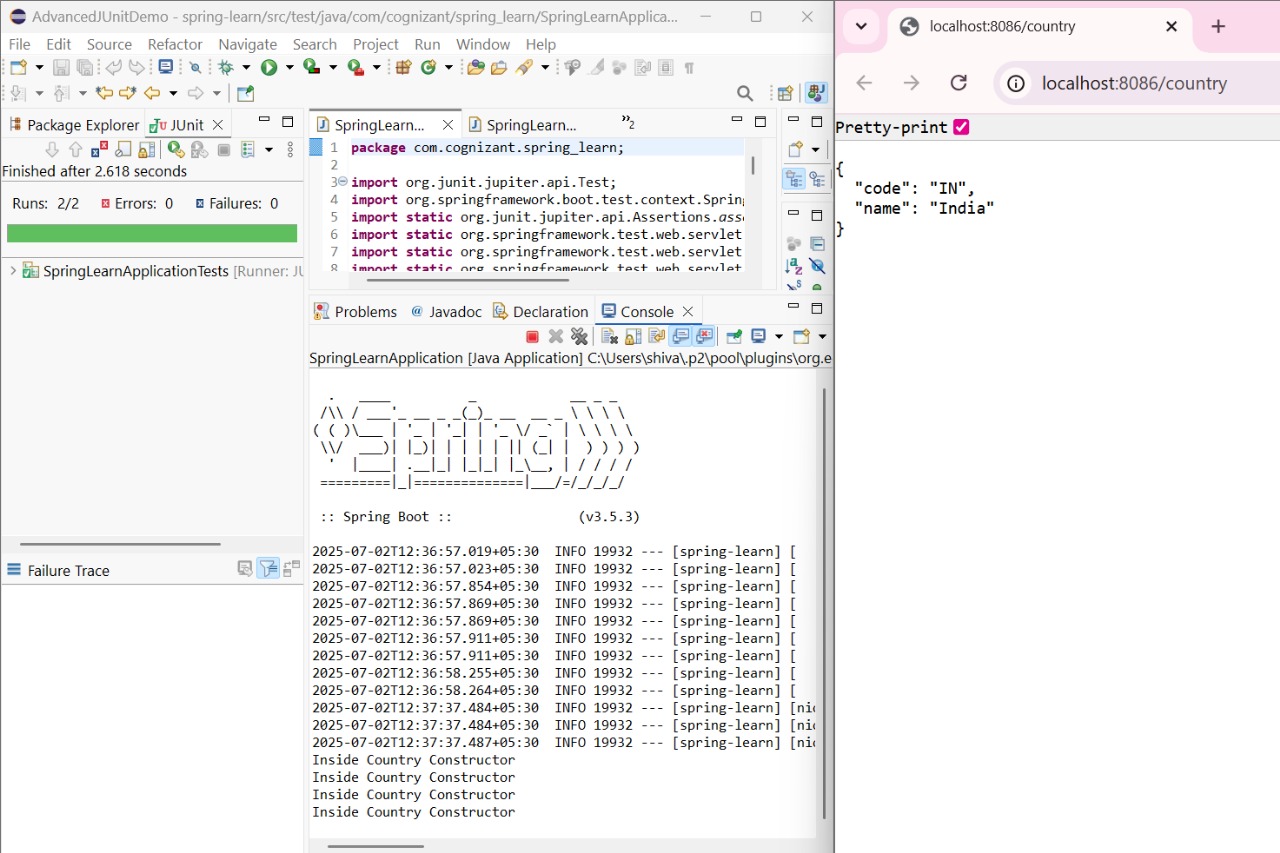


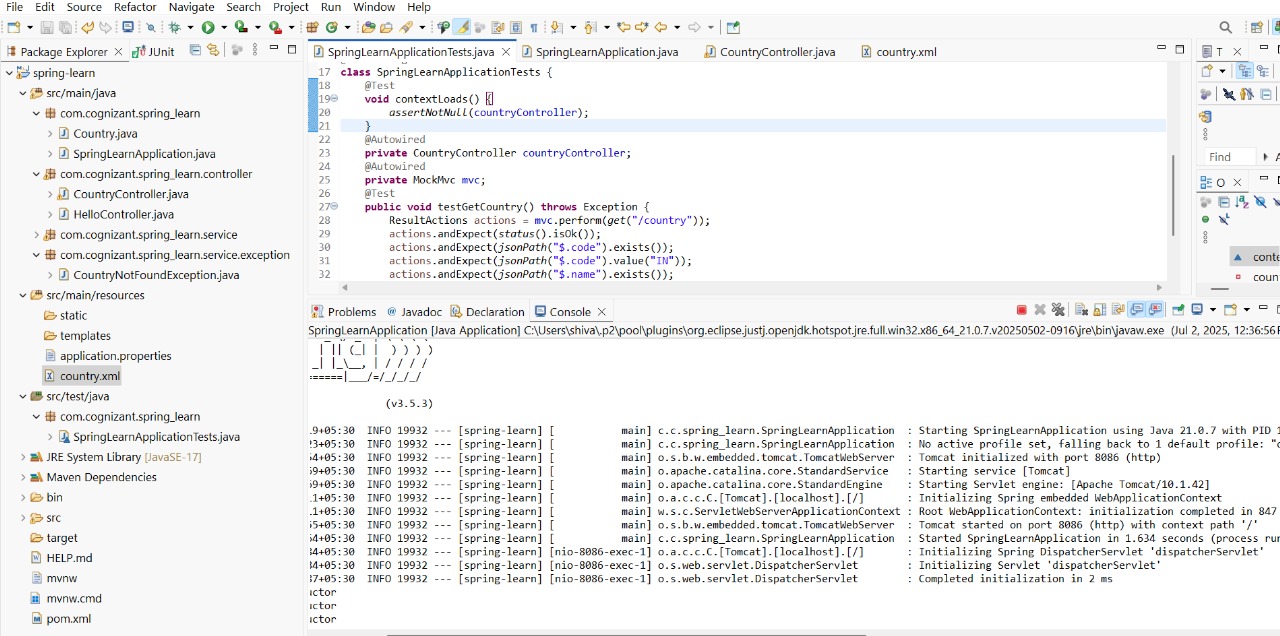


**MockMVC - Test get country service**   
   
Using MockMVC test the get country service.  
   
Create a test cases to test the following aspects:

* Test is the CountryController is loaded
* Invoke the service to get country and check in the response if it contains code as "IN" and name as "India"

**Output:**





**Create RESTful Web Service to handle POST request of Country**

**A new RESTful Web Service method to handle POST request of Country. Follow steps below to incorporate the same:**

Create new method in CountryController based on the following details:

Annotation - @PostMapping()

Method Signature - public void addCountry()

Within this method  include "Start" logger.

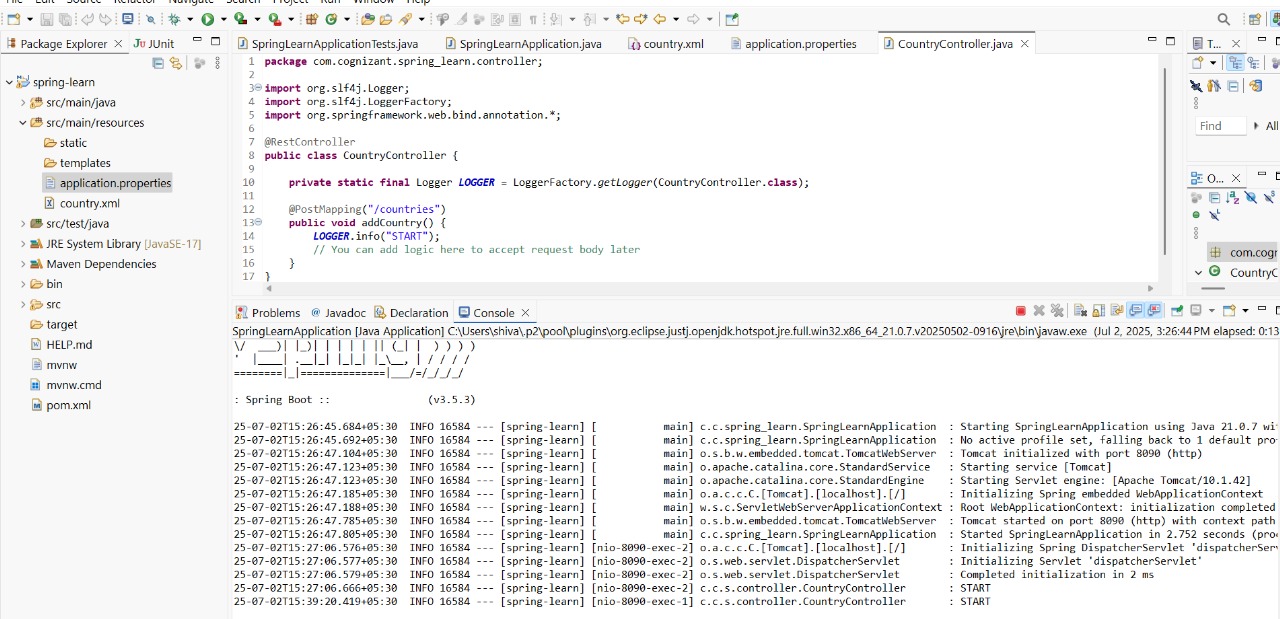
Start the web application

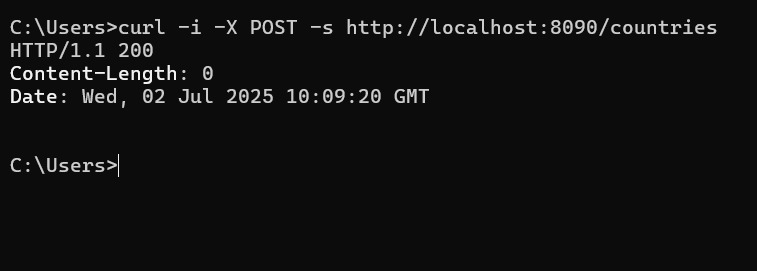
Open Git Bash

Execute the following curl command, to invoke the web service:

-i to display the

**Output:**





**Read country data as a bean in RESTful Web Service**

**The country data should be included in the request payload, which should be read by the controller method.**

Follow steps below to incorporate the same:

Include country as parameter to addCountry() method with @RequestBody annotation and country as parameter. Refer method signature below.

public Country addCountry(@RequestBody Country country)

Include log to display country details

Return the country. This is to check if country details are populated correctly

Invoke the service using the following curl command. This can also be tried for execution from Postman.

-H denotes inclusion of header. This denotes that we are sending content type in the request header and it mentions that the request payload is of type JSON

-d denotes the data payload sent in the request. This represents the country to be added

curl -i -H 'Content-Type: application/json' -X POST -s -d '{"code":"IN","name":"India"}' http://localhost:8090/countries

**Output:**

