**Hands on 1**

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

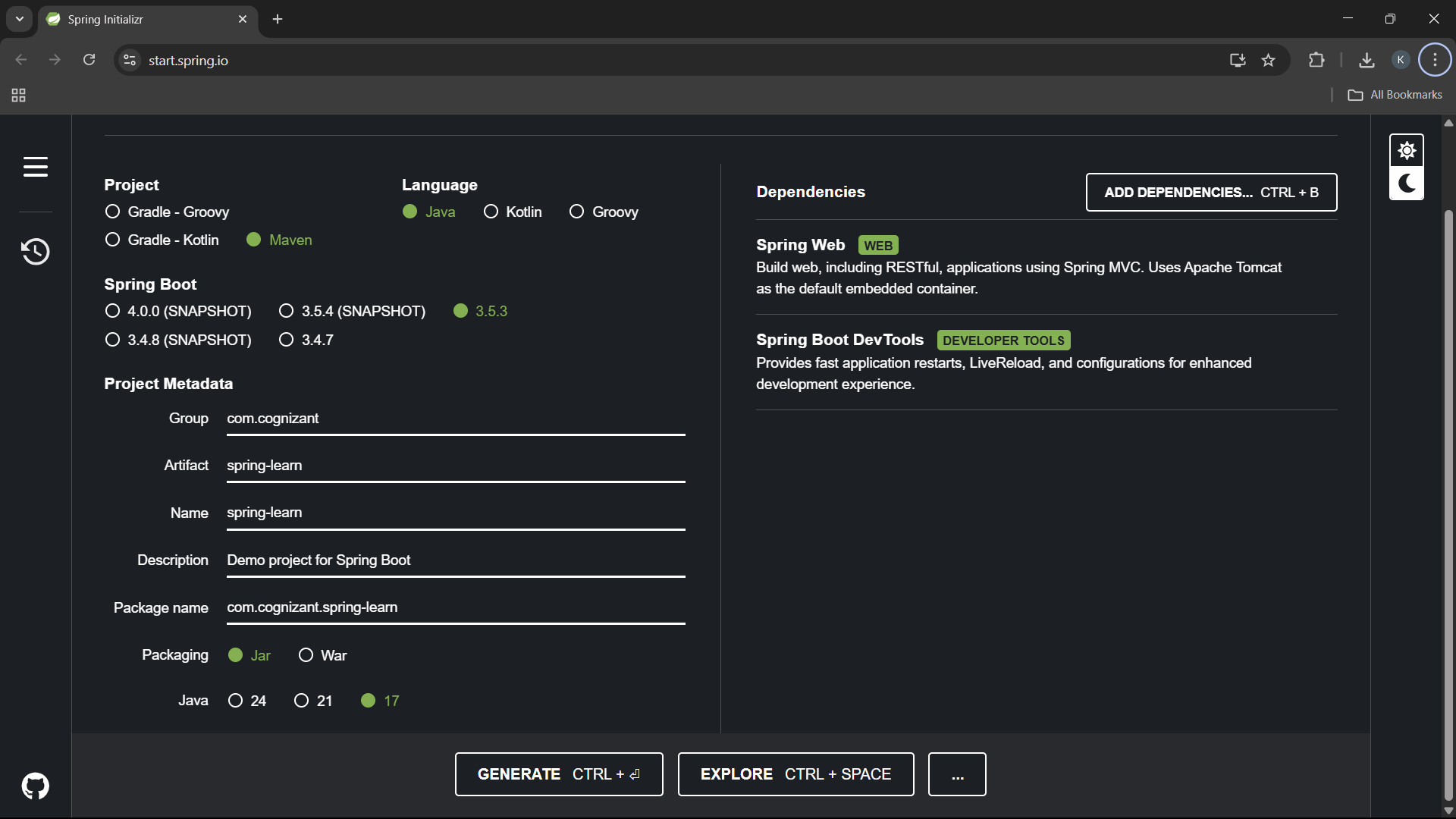
1. Go to <https://start.spring.io/>
2. Change Group as “com.cognizant”
3. Change Artifact Id as “spring-learn”
4. Select Spring Boot DevTools and Spring Web
5. Create and download the project as zip
6. Extract the zip in root folder to Eclipse Workspace
7. Build the project using ‘mvn clean package -Dhttp.proxyHost=proxy.cognizant.com -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:

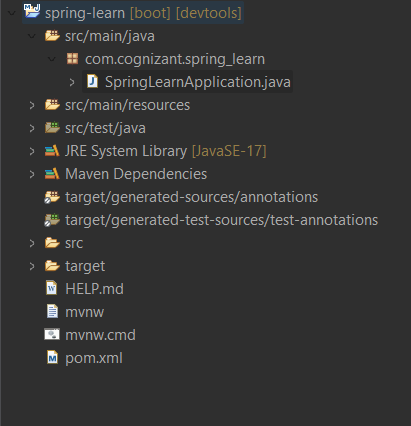
1. src/main/java - Folder with application code
2. src/main/resources - Folder for application configuration
3. src/test/java - Folder with code for testing the application
4. SpringLearnApplication.java - Walkthrough the main() method.
5. Purpose of @SpringBootApplication annotation
6. pom.xml
   1.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

**Solution**

**Spring Initializer :**



**Imported Project after extraction :**

****

**Main Application :**

package com.cognizant.spring\_learn;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringLearnApplication {

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

System.out.println("SpringLearnApplication started...");

}

}

**Pom.xml –**

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.5.3</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.cognizant</groupId>

<artifactId>spring-learn</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-learn</name>

<description>Demo project for Spring Boot</description>

<url/>

<licenses>

<license/>

</licenses>

<developers>

<developer/>

</developers>

<scm>

<connection/>

<developerConnection/>

<tag/>

<url/>

</scm>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

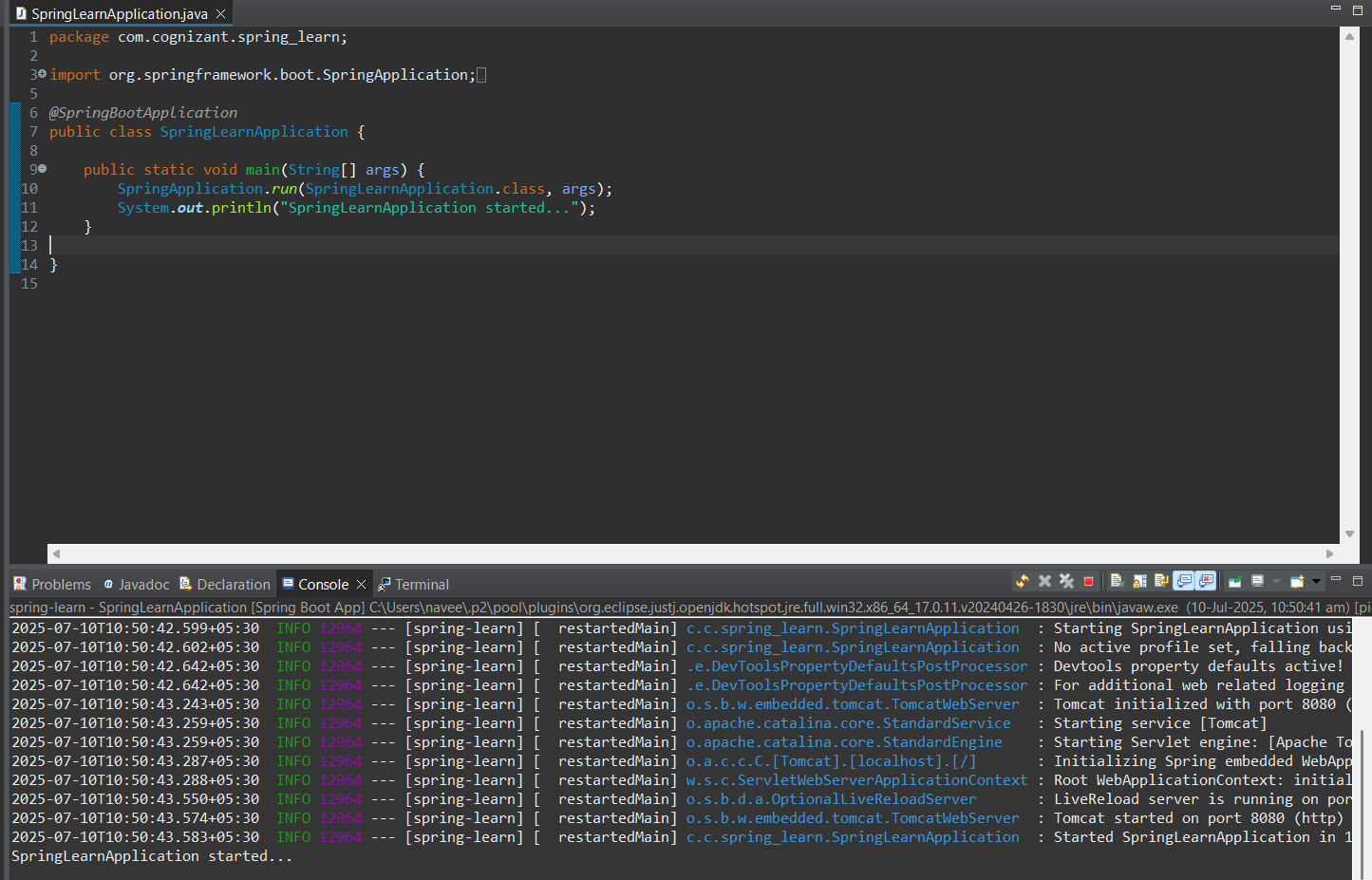
</plugin>

</plugins>

</build>

</project>

**Output :**

****

**SME :**

1. **src/main/java - Folder with application code**

**Answer** - This directory contains all the **main Java source files** of the application. This is the core folder where we write the business logic, controllers, services, and other application classes.

1. **src/main/resources - Folder for application configuration**

**Answer -** This is where we keep all configuration and static resource files. These resources are automatically picked up by Spring Boot at runtime. Ex. Application.properties.

1. **src/test/java - Folder with code for testing the application**

**Answer -**  Contains unit and integration test cases written using JUnit or Spring Boot Test. It mirrors the structure of src/main/java so each class can have a corresponding test class. Example: CountryTest.java would test the Country.java class.

1. **SpringLearnApplication.java - Walkthrough the main() method.**

**Answer -** This class contains the main() method which is the entry point of the application.

**Code –**

@SpringBootApplication

public class SpringlearnApplication {

public static void main(String[] args) {

SpringApplication.run(SpringlearnApplication.class, args);

}

}

**Inside main():**

* SpringApplication.run(...) bootstraps the Spring Boot application.
* It automatically scans all classes in the package and subpackages to register beans and components.

1. **Purpose of @SpringBootApplication annotation**

**Answer -** This is a composite annotation that includes:

* @Configuration: Allows the class to define beans using @Bean methods.
* @EnableAutoConfiguration: Automatically configures Spring components based on dependencies on the classpath.
* @ComponentScan: Scans for components like @RestController, @Service, @Component in the current package and subpackages.

This annotation is essential to enable Spring Boot’s auto-configuration and component scanning features.

1. **pom.xml**
   1. **Walkthrough all the configuration defined in XML file**

**Answer –**

*Parent Tag –*

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.5.3</version>

</parent>

* Inherits Spring Boot’s default dependency versions, plugin configuration, and dependency management.
* Ensures compatibility with Spring Boot 3.5.3 and simplifies version control for dependencies.

*Project Coordinates –*

<groupId>com.cognizant</groupId>

<artifactId>spring-learn</artifactId>

<version>0.0.1-SNAPSHOT</version>

These values uniquely identify your project in a Maven repository.

* groupId: Organization or company name
* artifactId: Project or module name
* version: Current version of the project (SNAPSHOT indicates development version)

*Java Version –*

<properties>

<java.version>17</java.version>

</properties>

* Specifies that the project should be compiled with **Java 17**.
* Used by Spring Boot to generate correct bytecode and avoid compatibility issues.

*Dependencies Section –*

This section tells Maven what libraries your project depends on.

1. Spring Boot Web Starter

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

Brings in everything required for building a RESTful web application:

* Spring MVC, Tomcat (embedded), Jackson (for JSON), logging, etc.

1. Spring Boot DevTools

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

* Adds auto-restart and hot-reloading support.
* Makes development faster by automatically restarting the app when files are changed.

1. Spring Boot Starter Test

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

Provides libraries like:

* **JUnit 5** for writing unit tests
* **Mockito** for mocking
* **Spring Test** for integration testing

*Build Section – Plugin –*

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

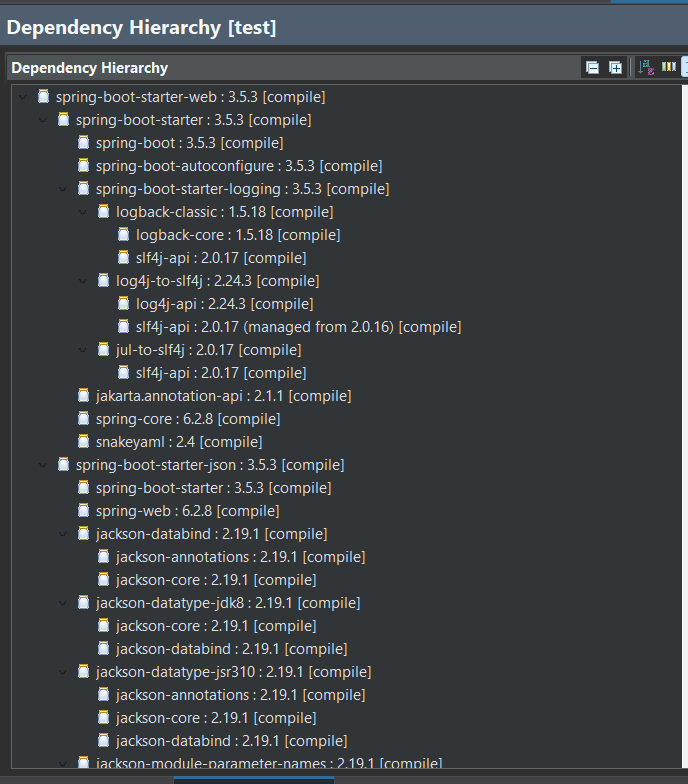
</build>

Allows you to **build and run** the application using:

* mvn spring-boot:run
* mvn package (generates an executable .jar)

Embeds the application server (Tomcat) into the JAR, so no external server is needed.

**Open 'Dependency Hierarchy' and show the dependency tree.**

****

**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. Refer code below.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://www.springframework.org/schema/beans

        https://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="dateFormat" class="java.text.SimpleDateFormat">

<constructor-arg value="dd/MM/yyyy" />

</bean>

</beans>

* Create new method displayDate() in SpringLearnApplication.java
* In displayDate() method create the ApplicationContext. Refer code below:

ApplicationContext context = new ClassPathXmlApplicationContext("date-format.xml");

* Get the dateFormat using getBean() method. Refer code below.

SimpleDateFormat format = context.getBean("dateFormat", SimpleDateFormat.class);

* 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.

**Troubleshooting Tips**   
  
If the tomcat port has a conflict and the server is not starting include the below property in application.properties file in src/main/resources folder.

**Solution:**

**Code :**

**Country.java :**

package com.cognizant.springlearn;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class Country {

private static final Logger LOGGER = LoggerFactory.getLogger(Country.class);

private String code;

private String name;

public Country() {

LOGGER.debug("Inside Country Constructor.");

}

public String getCode() {

LOGGER.debug("Inside getCode()");

return code;

}

public void setCode(String code) {

LOGGER.debug("Inside setCode()");

this.code = code;

}

public String getName() {

LOGGER.debug("Inside getName()");

return name;

}

public void setName(String name) {

LOGGER.debug("Inside setName()");

this.name = name;

}

@Override

public String toString() {

return "Country [code=" + code + ", name=" + name + "]";

}

}

**Country.xml :**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="country" class="com.cognizant.springlearn.Country">

<property name="code" value="IN" />

<property name="name" value="India" />

</bean>

</beans>

**Logback.xml :**

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger**{36}** - %msg%n</pattern>

</encoder>

</appender>

<root level="debug">

<appender-ref ref="STDOUT"/>

</root>

</configuration>

**SpringlearnApplication.java :**

package com.cognizant.springlearn;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringlearnApplication {

private static final Logger LOGGER = LoggerFactory.getLogger(SpringlearnApplication.class);

public static void main(String[] args) {

LOGGER.debug("START");

displayCountry();

LOGGER.debug("END");

}

public static void displayCountry() {

ApplicationContext context = new ClassPathXmlApplicationContext("country.xml");

Country country = (Country) context.getBean("country", Country.class);

LOGGER.debug("Country : {}", country.toString());

}

}

**Pom.xml :**

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.5.3</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.cognizant</groupId>

<artifactId>springlearn</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>springlearn</name>

<description>Demo project for Spring Boot</description>

<url/>

<licenses>

<license/>

</licenses>

<developers>

<developer/>

</developers>

<scm>

<connection/>

<developerConnection/>

<tag/>

<url/>

</scm>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<!-- Spring Core -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<!-- SLF4J + Logback for logging -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

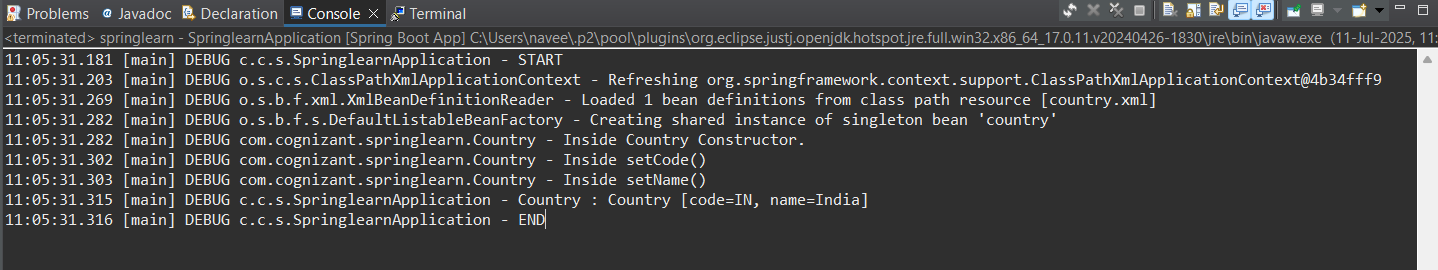
</plugin>

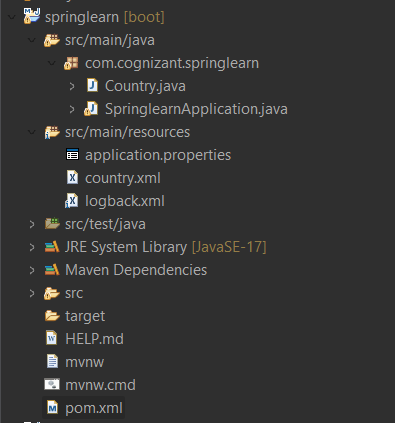
</plugins>

</build>

</project>

**Output : (Screenshots)**

****

****

**SME :**

* **bean tag, id attribute, class attribute, property tag, name attribute, value attribute**

**Answer -** The <bean> tag is used in Spring XML configuration to define a bean (an object) that will be managed by the Spring IoC (Inversion of Control) container.

**Attributes:**

* **id:** A unique identifier for the bean. You use this ID to retrieve the bean from the Spring container.
* **class:** Fully qualified name of the class that the Spring container will instantiate.

**Ex: <bean id="country" class="com.cognizant.springlearn.Country">**

This tells Spring to create an object of Country class and name it as "country" within the container.

The <property> tag is used to inject values into the bean’s fields (using setter injection).

**Attributes:**

* **name**: Name of the Java property (must match the setter method in the class).
* **value**: Literal value to be set for that property.

**Ex: <property name="code" value="IN" />**

**<property name="name" value="India" />**

This will invoke the setCode("IN") and setName("India") methods of the Country class when the bean is initialized.

* **ApplicationContext, ClassPathXmlApplicationContext**

**Answer –**

* ApplicationContext is the central interface for accessing Spring’s IoC container.
* ClassPathXmlApplicationContext is a concrete implementation that loads the Spring configuration from an XML file in the classpath**.**

**Ex : ApplicationContext context = new ClassPathXmlApplicationContext("country.xml");**

This line creates and initializes the Spring container by reading the country.xml file.

* **What exactly happens when context.getBean() is invoked**

**Answer –**

**Country country = (Country) context.getBean("country", Country.class);**

Here’s what happens:

1. Spring looks up the bean with ID "country" in the XML config.
2. It finds the class name com.cognizant.springlearn.Country.
3. It creates an instance of that class (using the default constructor).
4. It injects the properties using the setters (setCode(), setName()).
5. The fully initialized bean is returned.

This is the essence of Dependency Injection in Spring.

**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!!   
  
**IMPORTANT NOTE**: Don't forget to include start and end log in the sayHello() method.  
  
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

**Solution:**

**Code –**

**HelloController.java :**

package com.cognizant.spring\_learn.controller;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

@RestController

public class HelloController {

private static final Logger LOGGER = LoggerFactory.getLogger(HelloController.class);

@GetMapping("/hello")

public String sayHello() {

LOGGER.debug("START");

String msg = "Hello World!";

LOGGER.debug("END");

return msg;

}

}

SpringlearnApplication.java \u2013

package com.cognizant.spring\_learn;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringLearnApplication {

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

System.out.println("SpringLearnApplication started...");

}

}

**Application.properties :**

spring.application.name=spring-learn

server.port=8083

**pom.xml :**

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.5.3</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.cognizant</groupId>

<artifactId>spring-learn</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-learn</name>

<description>Demo project for Spring Boot</description>

<url/>

<licenses>

<license/>

</licenses>

<developers>

<developer/>

</developers>

<scm>

<connection/>

<developerConnection/>

<tag/>

<url/>

</scm>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

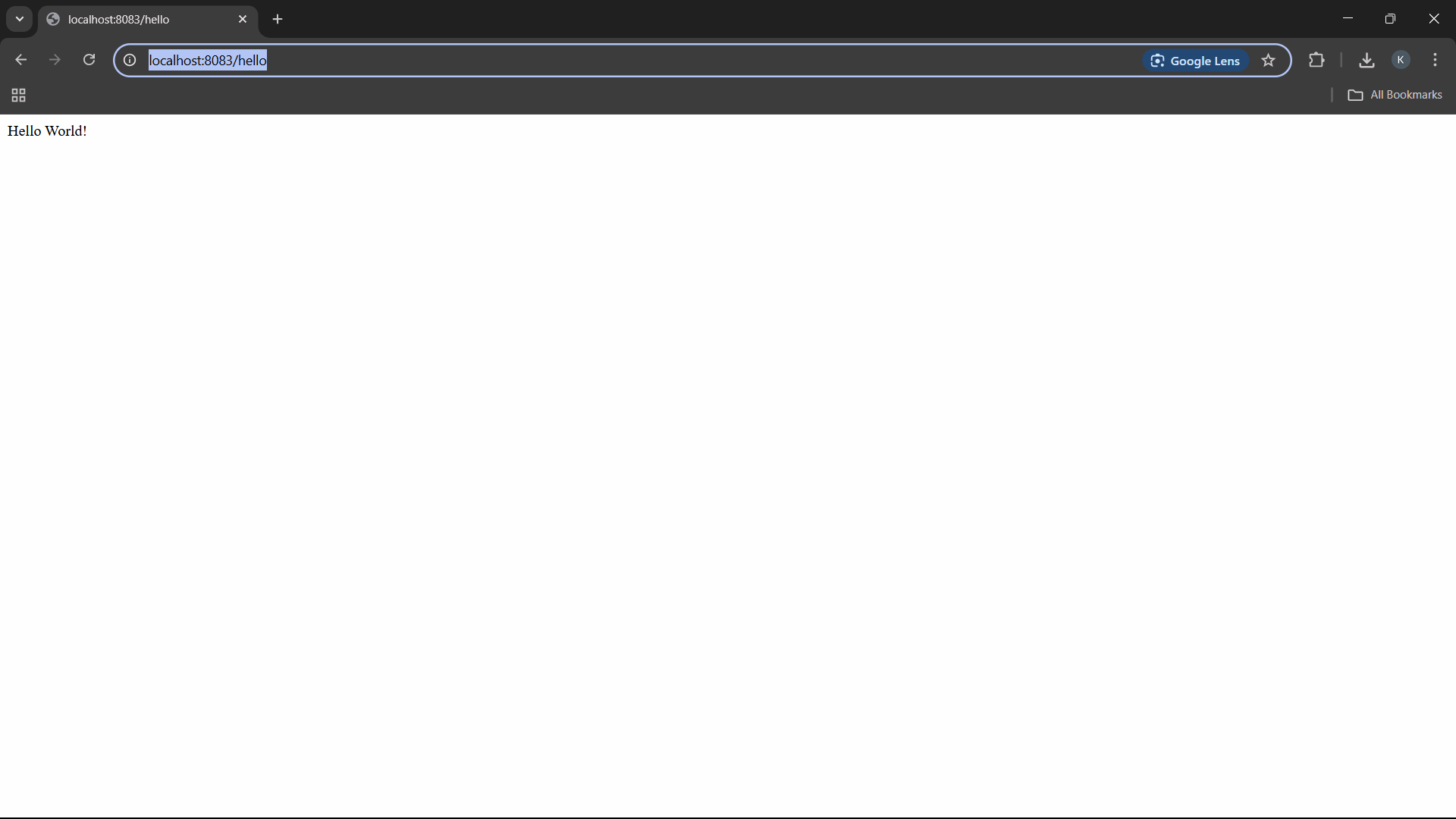
</plugin>

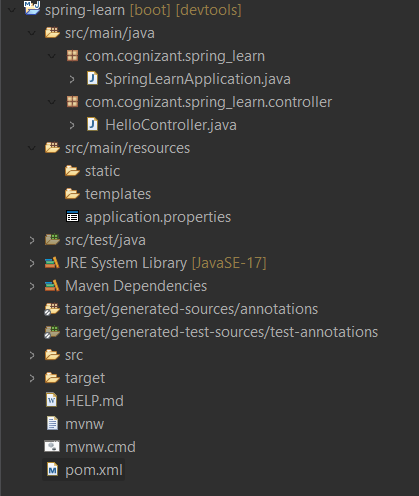
</plugins>

</build>

</project>

**Output – (Screenshots)**



****

**SME :**

**In network tab of developer tools show the HTTP header details received**

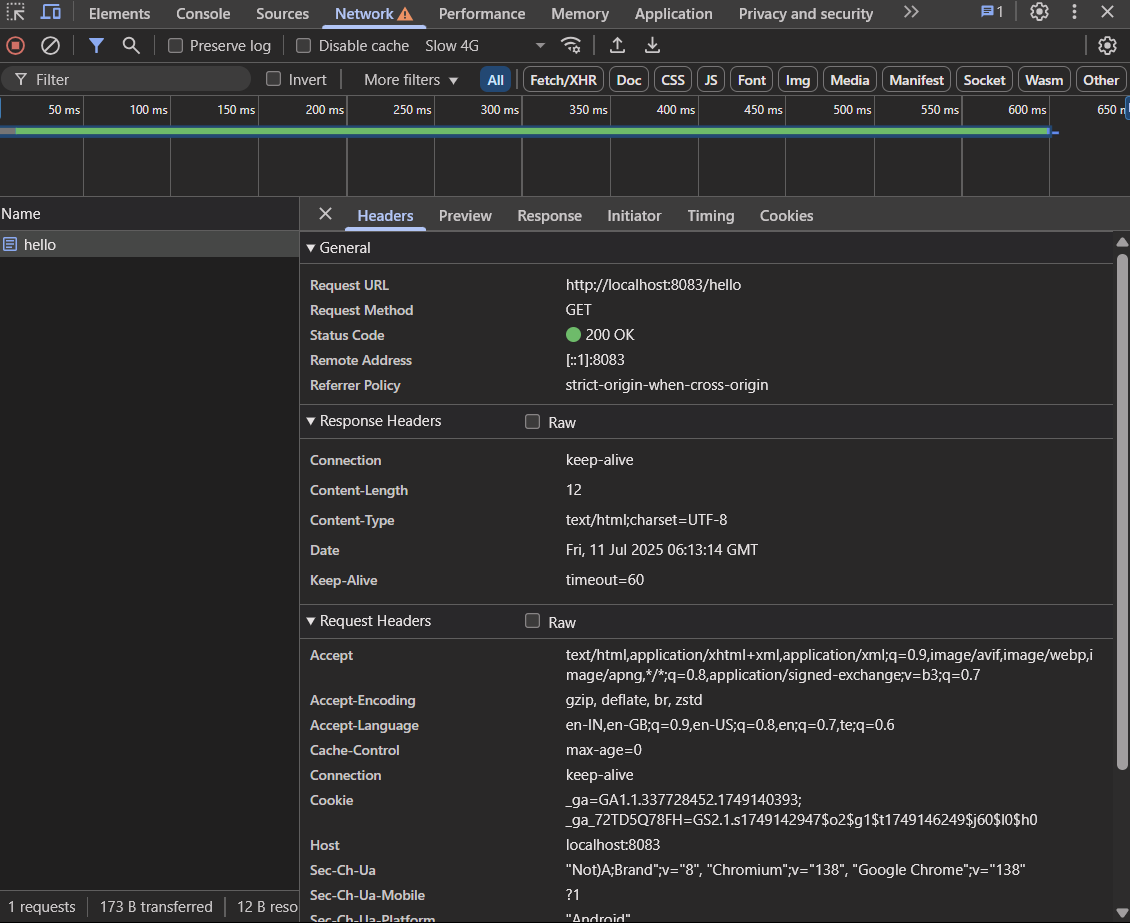
When you open <http://localhost:8083/hello> in Chrome:

1. Open **Dev Tools** → Go to the **Network** tab
2. Refresh the page (F5) → Click on the /hello request

Headers You’ll See (under “Response Headers” and “General”):

HTTP headers give **metadata** about the request and response:

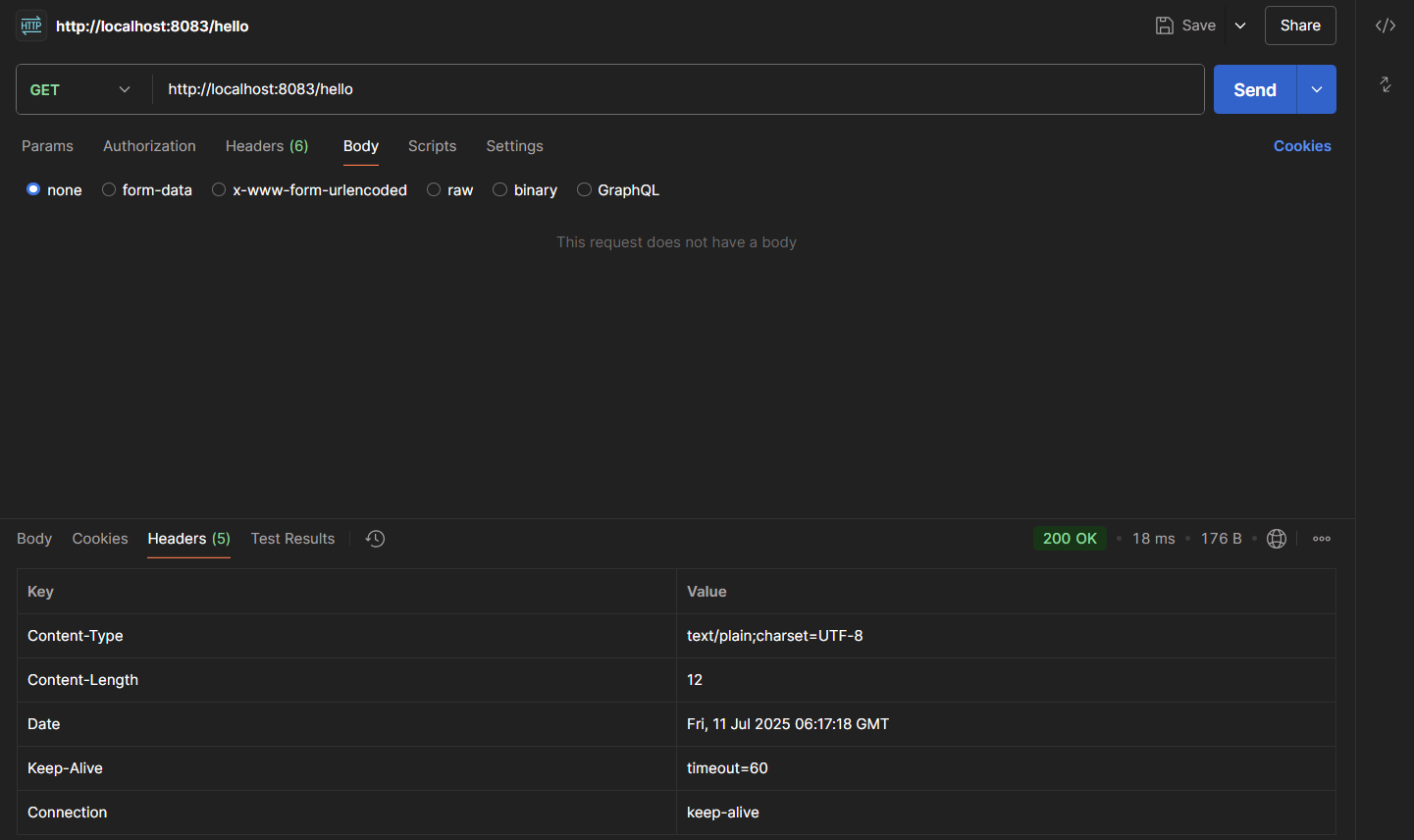
* They tell clients **how to interpret** the response
* Help with **debugging, caching, and security**
* Provide insights on server behavior and content type



**In postman click on "Headers" tab to view the HTTP header details received**

When you send a GET request to <http://localhost:8083/hello> in **Postman**:

1. Click **"Send"**
2. Go to the **"Headers" tab** under the response section



**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

**Solution :**

**Code –**

**CountryController.java :**

package com.cognizant.springlearn.controller;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

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

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

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

import com.cognizant.springlearn.Country;

@RestController

public class CountryController {

private static final Logger LOGGER = LoggerFactory.getLogger(CountryController.class);

@RequestMapping(value = "/country", method = RequestMethod.GET)

public Country getCountryIndia() {

LOGGER.debug("Start - getCountryIndia()");

ApplicationContext context = new ClassPathXmlApplicationContext("country.xml");

Country country = context.getBean("country", Country.class);

LOGGER.debug("Country: {}", country);

return country;

}

}

**SpringlearnApplication.java :**

package com.cognizant.springlearn;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringlearnApplication {

public static void main(String[] args) {

SpringApplication.run(SpringlearnApplication.class, args);

}

}

**Country.java :**

package com.cognizant.springlearn;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class Country {

private static final Logger LOGGER = LoggerFactory.getLogger(Country.class);

private String code;

private String name;

public Country() {

LOGGER.debug("Inside Country Constructor.");

}

public String getCode() {

LOGGER.debug("Inside getCode()");

return code;

}

public void setCode(String code) {

LOGGER.debug("Inside setCode()");

this.code = code;

}

public String getName() {

LOGGER.debug("Inside getName()");

return name;

}

public void setName(String name) {

LOGGER.debug("Inside setName()");

this.name = name;

}

@Override

public String toString() {

return "Country [code=" + code + ", name=" + name + "]";

}

}

**Country.xml :**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="country" class="com.cognizant.springlearn.Country">

<property name="code" value="IN" />

<property name="name" value="India" />

</bean>

</beans>

**Logback.xml :**

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger**{36}** - %msg%n</pattern>

</encoder>

</appender>

<root level="debug">

<appender-ref ref="STDOUT"/>

</root>

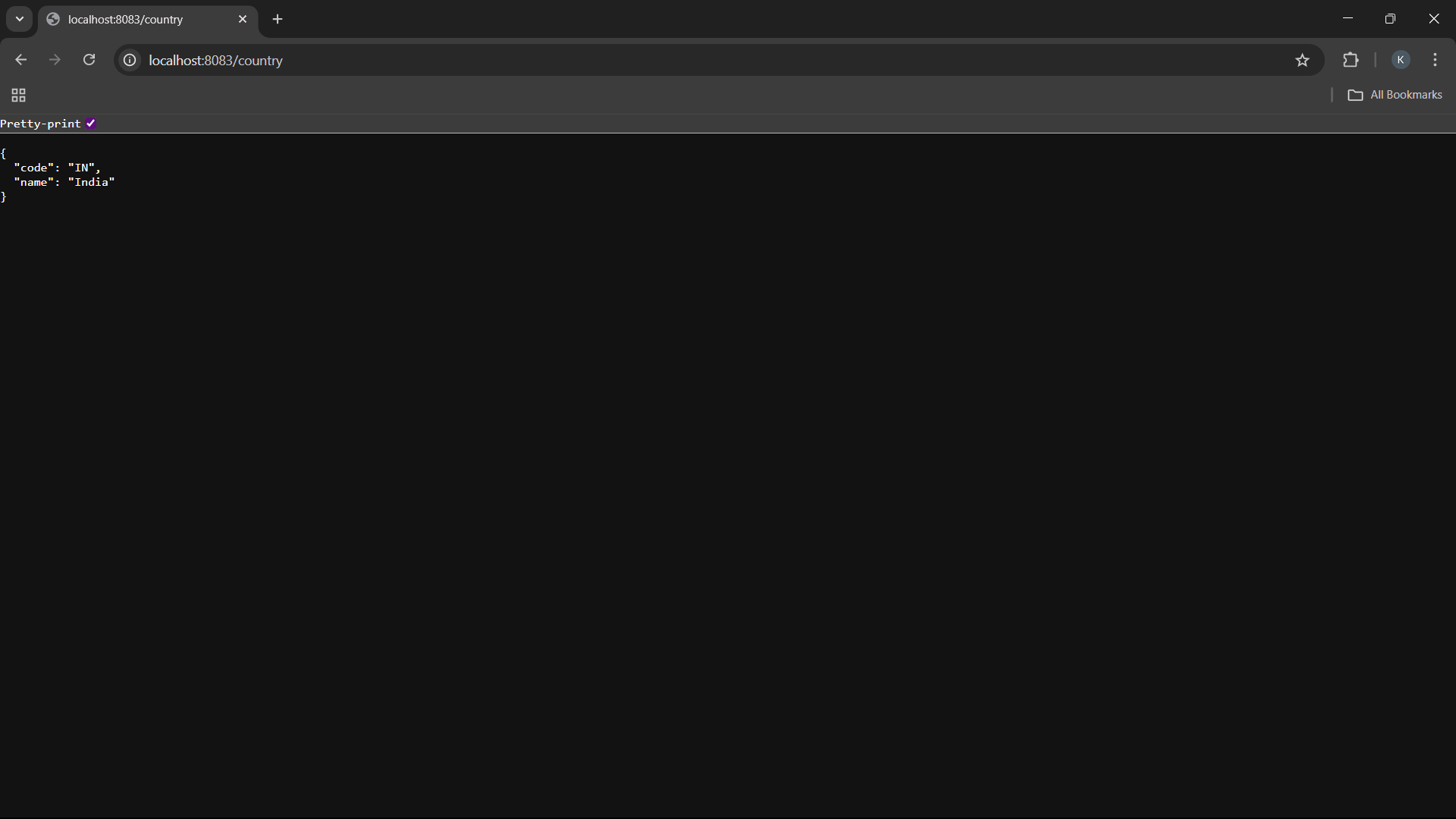
</configuration>

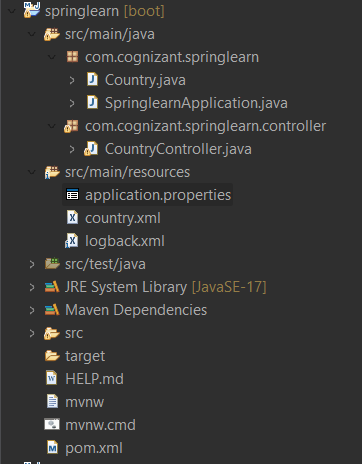
**Application.properties :**

spring.application.name=springlearn

server.port=8083

**Output – (Screenshots)**



****

**SME :**

* **What happens in the controller method?**

**Answer -** When the user hits the URL http://localhost:8083/country, the following happens:

Lifecycle:

1. Spring Boot handles the incoming HTTP GET request to /country.
2. It matches the request to this controller method:

@RequestMapping(value = "/country", method = RequestMethod.GET)

public Country getCountryIndia() { ... }

1. The Country bean (either autowired or loaded from XML) is returned from the method.
2. Spring uses **HttpMessageConverter** (Jackson by default) to convert the Country object into **JSON**.
3. The JSON is sent back in the **HTTP Response Body**.

* **How the bean is converted into JSON reponse?**

**Answer -** Spring Boot uses Jackson (a JSON library) which is included automatically via:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

**Behind the scenes:**

* @RestController tells Spring to skip view rendering and instead return **data** (usually as JSON).
* Jackson looks at your Country class (using getters) and serializes it into JSON.
* For example: Country country = new Country("IN", "India");

is converted to:

{

"code": "IN",

"name": "India"

}

This happens automatically because of **Spring's HttpMessageConverter chain**.

* **In network tab of developer tools show the HTTP header details received**

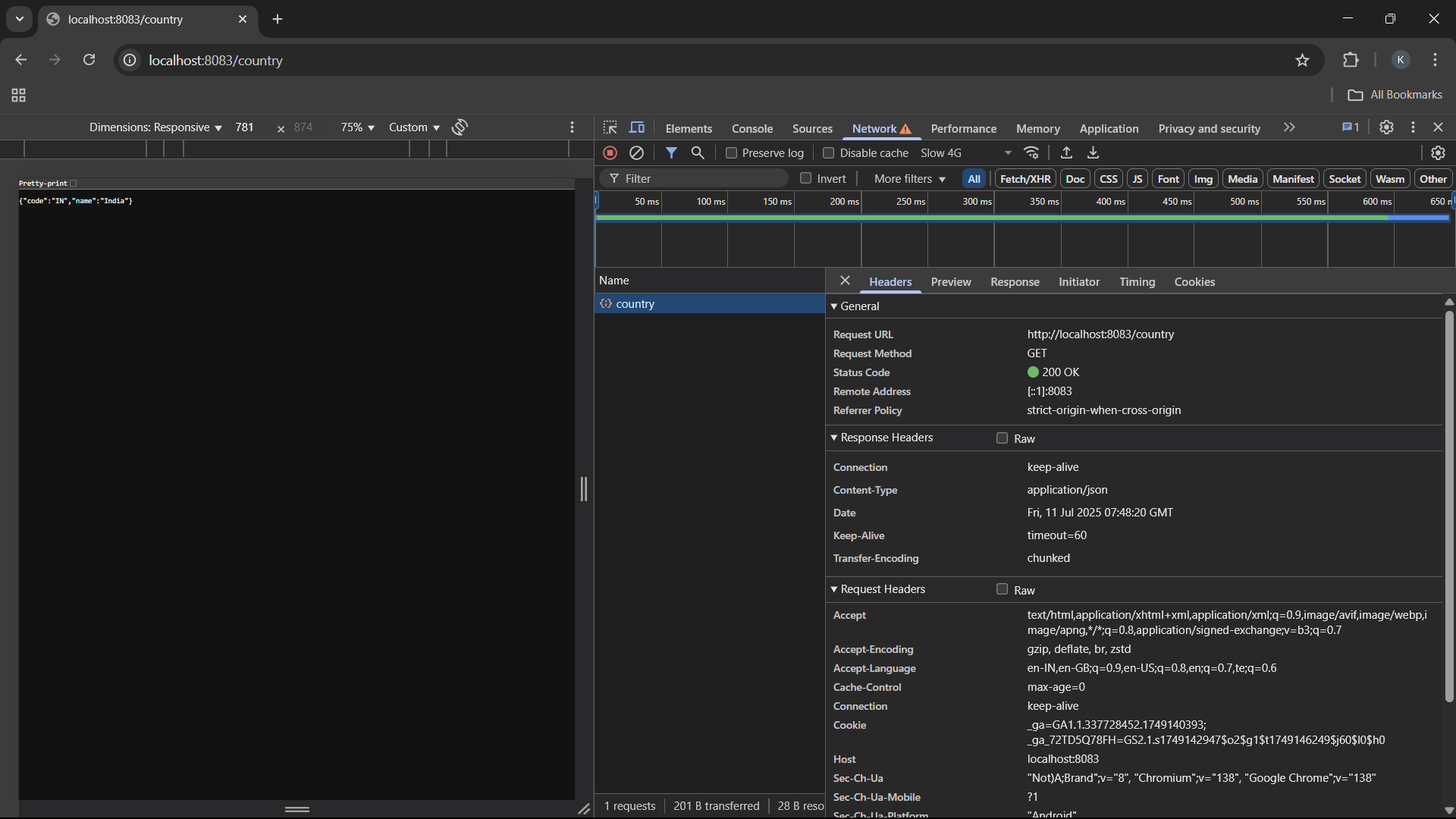
**Answer** - You can inspect the response headers using:

**Steps (in Chrome):**

1. Right-click > Inspect → Open Developer Tools.
2. Go to the Network tab.
3. Hit your endpoint in browser: http://localhost:8083/country
4. Click on the network request (usually labeled country).
5. View the Headers tab:
   * Request Headers: Info sent from browser (like User-Agent, Accept).
   * Response Headers: Info sent from your Spring Boot app, such as:

Content-Type: application/json

Content-Length: 34



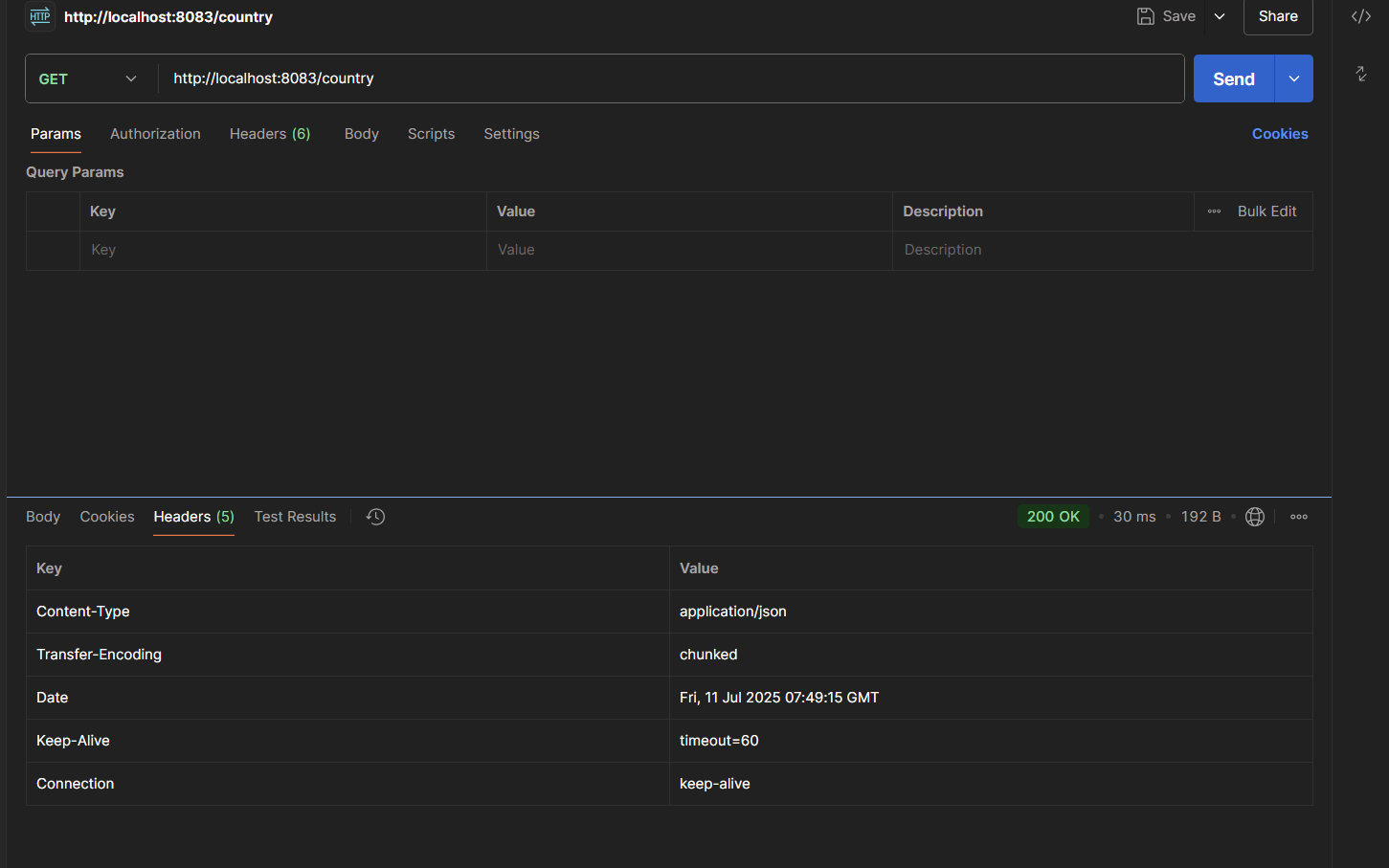
* **In postman click on "Headers" tab to view the HTTP header details received**

**Answer –**

**Steps:**

1. Open Postman.
2. Send a GET request to: http://localhost:8083/country
3. Go to the Headers tab (right next to "Body" and "Cookies").
4. You'll see:
   * Content-Type: application/json (returned from Spring)
   * Content-Length
   * Date
   * Server: Apache Tomcat

These headers confirm that the response is valid JSON, sent over HTTP correctly.



**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"

}

**Solution:**

**Code :**

**CountryController.java :**

package com.cognizant.springlearn.controller;

import java.util.List;

import com.cognizant.springlearn.Country;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.ApplicationContext;

import org.springframework.web.bind.annotation.\*;

@RestController

public class CountryController {

private static final Logger LOGGER = LoggerFactory.getLogger(CountryController.class);

@Autowired

private ApplicationContext context;

@RequestMapping(value = "/country", method = RequestMethod.GET)

public Country getCountryIndia() {

LOGGER.debug("Start - getCountryIndia()");

Country country = context.getBean("country", Country.class);

LOGGER.debug("Country: {}", country);

return country;

}

@GetMapping("/countries/{code}")

public Country getCountry(@PathVariable String code) {

LOGGER.debug("Start - getCountry() for code: {}", code);

List<Country> countries = (List<Country>) context.getBean("countryList");

return countries.stream()

.filter(c -> c.getCode().equalsIgnoreCase(code))

.findFirst()

.orElse(null);

}

}

**SpringlearnApplication.java :**

package com.cognizant.springlearn;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.ImportResource;

@SpringBootApplication

@ImportResource("classpath:country.xml")

public class SpringlearnApplication {

public static void main(String[] args) {

SpringApplication.run(SpringlearnApplication.class, args);

}

}

**Country.java :**

package com.cognizant.springlearn;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class Country {

private static final Logger LOGGER = LoggerFactory.getLogger(Country.class);

private String code;

private String name;

public Country() {

LOGGER.debug("Inside Country Constructor.");

}

public String getCode() {

LOGGER.debug("Inside getCode()");

return code;

}

public void setCode(String code) {

LOGGER.debug("Inside setCode()");

this.code = code;

}

public String getName() {

LOGGER.debug("Inside getName()");

return name;

}

public void setName(String name) {

LOGGER.debug("Inside setName()");

this.name = name;

}

@Override

public String toString() {

return "Country [code=" + code + ", name=" + name + "]";

}

}

**Country.xml :**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="countryList" class="java.util.ArrayList">

<constructor-arg>

<list>

<bean class="com.cognizant.springlearn.Country">

<property name="code" value="IN"/>

<property name="name" value="India"/>

</bean>

<bean class="com.cognizant.springlearn.Country">

<property name="code" value="US"/>

<property name="name" value="United States"/>

</bean>

<bean class="com.cognizant.springlearn.Country">

<property name="code" value="DE"/>

<property name="name" value="Germany"/>

</bean>

</list>

</constructor-arg>

</bean>

</beans>

**Logback.xml :**

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger**{36}** - %msg%n</pattern>

</encoder>

</appender>

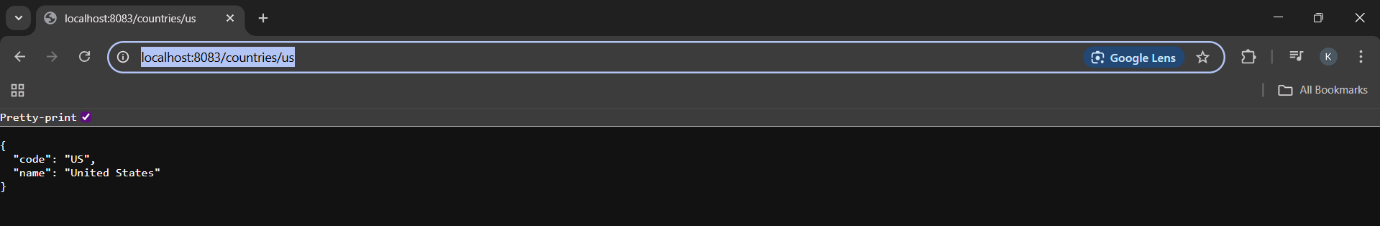
<root level="debug">

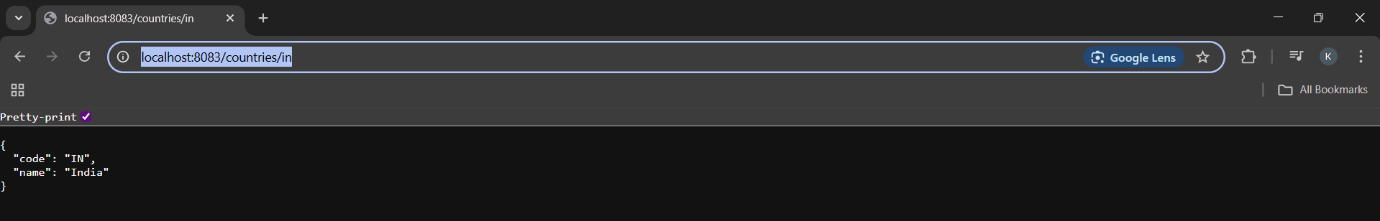
<appender-ref ref="STDOUT"/>

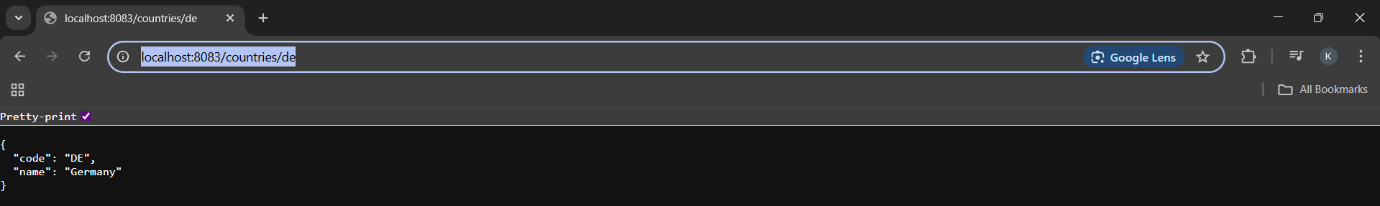
</root>

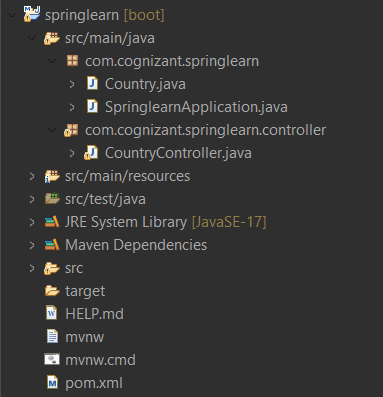
</configuration>

Output – (Screenshots)









**Create authentication service that returns JWT**   
  
As part of first step of JWT process, the user credentials needs to be sent to authentication service request that generates and returns the JWT.  
  
Ideally when the below curl command is executed that calls the new authentication service, the token should be responded. Kindly note that the credentials are passed using -u option.  
  
**Request**

curl -s -u user:pwd http://localhost:8090/authenticate

**Response**

{"token":"eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyIiwiaWF0IjoxNTcwMzc5NDc0LCJleHAiOjE1NzAzODA2NzR9.t3LRvlCV-hwKfoqZYlaVQqEUiBloWcWn0ft3tgv0dL0"}

This can be incorporated as three major steps:

* Create authentication controller and configure it in SecurityConfig
* Read Authorization header and decode the username and password
* Generate token based on the user retrieved in the previous step

Let incorporate the above as separate hands on exercises.

**Solution:**

**1. Create authentication controller and configure it in SecurityConfig**

**Code –**

**AuthController.java :**

package com.cognizant.jwttest.controller;

import org.springframework.web.bind.annotation.\*;

@RestController

public class AuthController {

@PostMapping("/authenticate")

public String authenticate() {

return "Authentication endpoint hit";

}

}

**SecurityConfig.java :**

package com.cognizant.jwttest.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http.csrf().disable()

.authorizeHttpRequests()

.requestMatchers("/authenticate").permitAll() // Allow this endpoint

.anyRequest().authenticated() // Secure all others

.and()

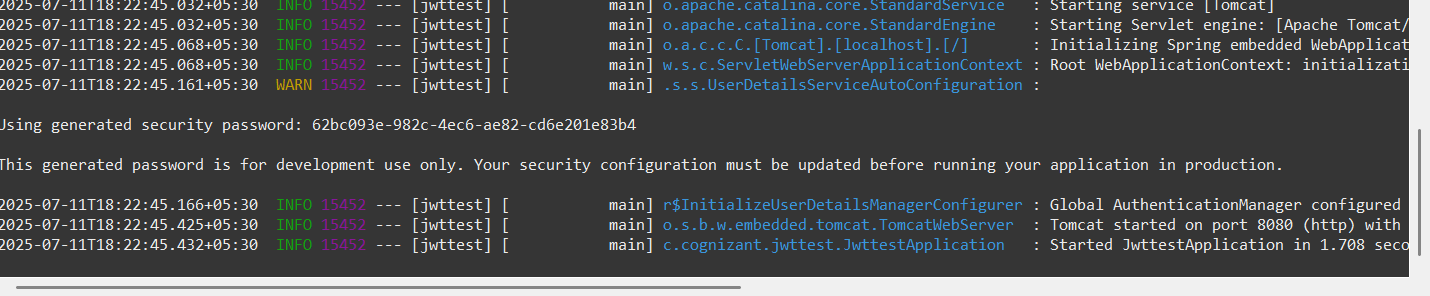
.httpBasic(); // Enable basic auth for username/password

return http.build();

}

}

**Output –**

****

**2. Read Authorization header and decode the username and password**

**Code –**

**AuthController.java –**

package com.cognizant.jwttest.controller;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.nio.charset.StandardCharsets;

import java.util.Base64;

@RestController

public class AuthController {

@PostMapping("/authenticate")

public ResponseEntity<?> authenticate(@RequestHeader("Authorization") String authHeader) {

if (authHeader == null || !authHeader.startsWith("Basic ")) {

return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Missing or invalid Authorization header");

}

// Decode Base64 credentials

String base64Credentials = authHeader.substring("Basic ".length());

byte[] decodedBytes = Base64.getDecoder().decode(base64Credentials);

String decodedCredentials = new String(decodedBytes, StandardCharsets.UTF\_8);

// Split into username and password

String[] values = decodedCredentials.split(":", 2);

if (values.length != 2) {

return ResponseEntity.status(HttpStatus.BAD\_REQUEST).body("Invalid credentials format");

}

String username = values[0];

String password = values[1];

// TEMP: Print to verify

return ResponseEntity.ok("Username: " + username + ", Password: " + password);

}

}

**SecurityConfig.java :**

package com.cognizant.jwttest.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.Customizer;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

public class SecurityConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.csrf(csrf -> csrf.disable())

.authorizeHttpRequests(auth -> auth

.requestMatchers("/authenticate").permitAll()

.anyRequest().authenticated()

)

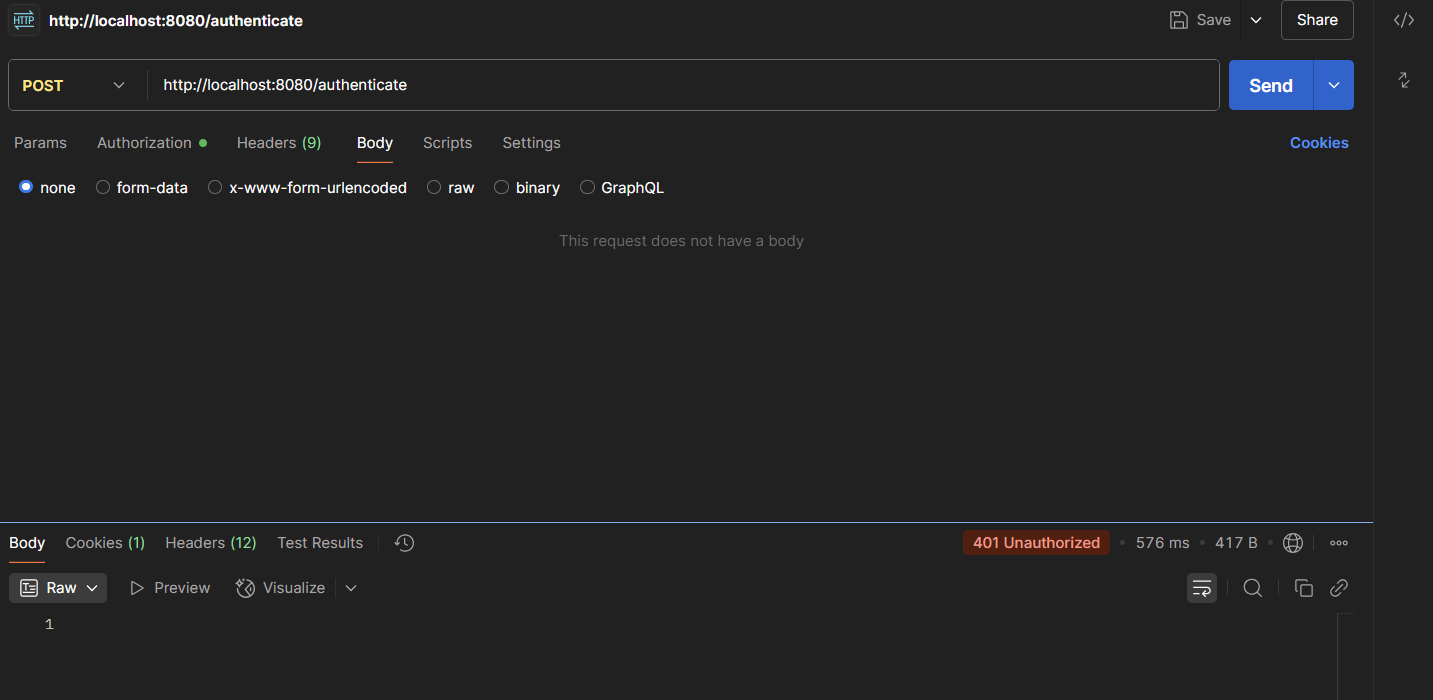
.httpBasic(Customizer.withDefaults()); // Enables Basic Auth

return http.build();

}

}

**Output –**

****

**3. Generate token based on the user retrieved in the previous step**

**Code –**

**JwtUtil.java :**

package com.cognizant.jwttest.util;

import java.util.Date;

import org.springframework.stereotype.Component;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

@Component

public class JwtUtil {

private final String SECRET\_KEY = "secret123"; // In real apps, use env vars

public String generateToken(String username) {

long nowMillis = System.currentTimeMillis();

long expMillis = nowMillis + 1000 \* 60 \* 10; // 10 mins expiry

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date(nowMillis))

.setExpiration(new Date(expMillis))

.signWith(SignatureAlgorithm.HS256, SECRET\_KEY.getBytes())

.compact();

}

}

**AuthController.java :**

package com.cognizant.jwttest.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import com.cognizant.jwttest.util.JwtUtil;

import java.nio.charset.StandardCharsets;

import java.util.Base64;

import java.util.HashMap;

import java.util.Map;

@RestController

public class AuthController {

@Autowired

private JwtUtil jwtUtil;

@PostMapping("/authenticate")

public ResponseEntity<?> authenticate(@RequestHeader("Authorization") String authHeader) {

if (authHeader == null || !authHeader.startsWith("Basic ")) {

return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Missing or invalid Authorization header");

}

String base64Credentials = authHeader.substring("Basic ".length());

byte[] decodedBytes = Base64.getDecoder().decode(base64Credentials);

String decodedCredentials = new String(decodedBytes, StandardCharsets.UTF\_8);

String[] values = decodedCredentials.split(":", 2);

if (values.length != 2) {

return ResponseEntity.status(HttpStatus.BAD\_REQUEST).body("Invalid credentials format");

}

String username = values[0];

String password = values[1];

// Hardcoded check \u2014 later replace with DB or user service

if ("user".equals(username) && "pwd".equals(password)) {

String token = jwtUtil.generateToken(username);

Map<String, String> response = new HashMap<>();

response.put("token", token);

return ResponseEntity.ok(response);

} else {

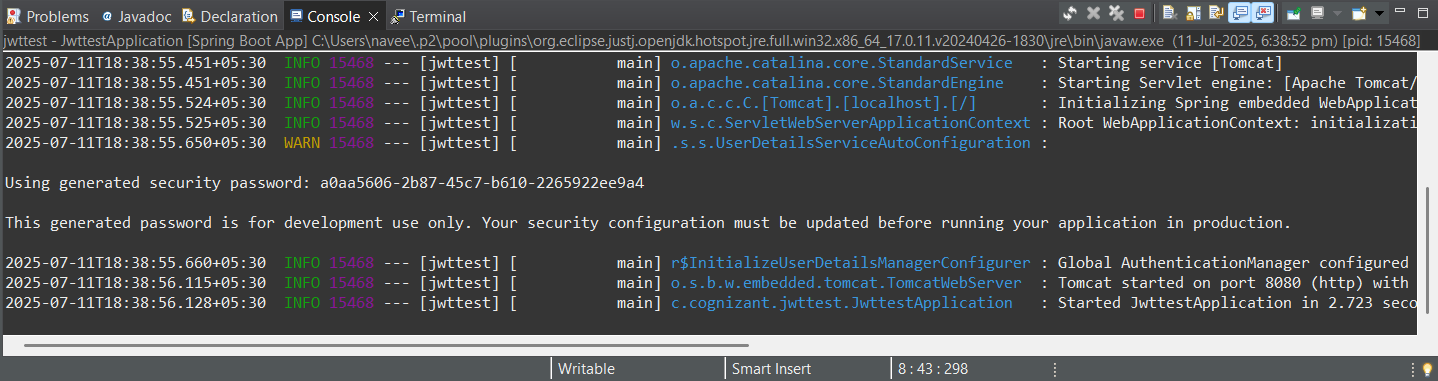
return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid username or password");

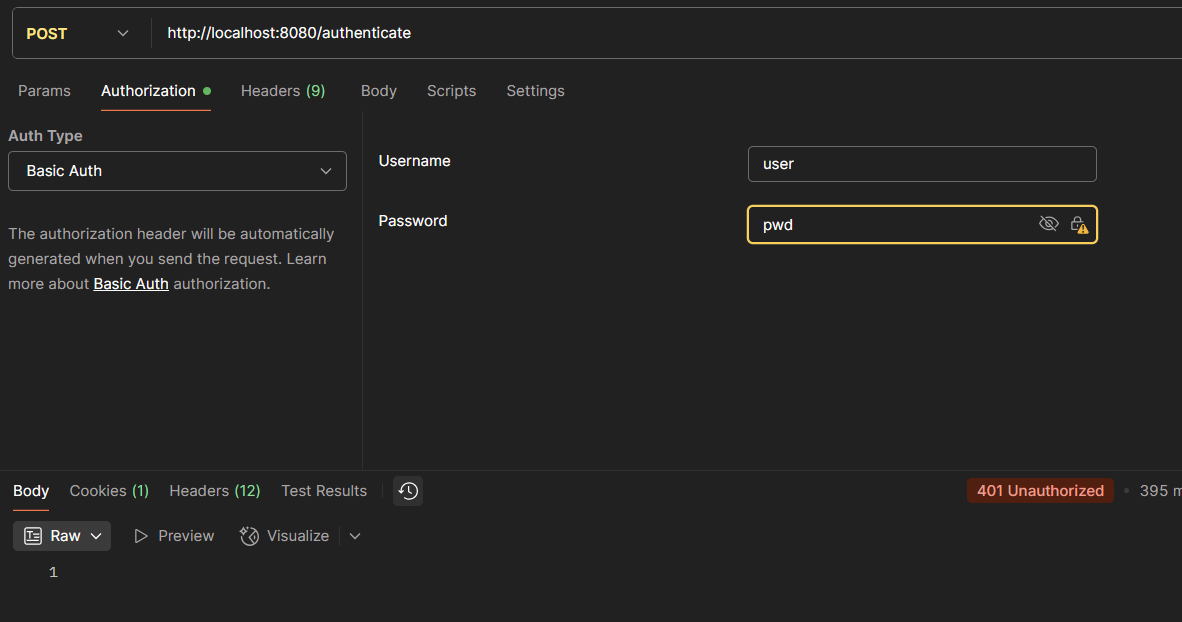
}

}

}

**Output :**

****

****