**WEEK-04**

**HANDS-ON SOLUTIONS**

**1. SPRING-REST-HANDSON**

**EXERCISE-01:**

**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:**As part of Exercise 1, I created a new Spring Boot web project using Maven. I started by navigating to Spring Initializr, where I configured the project with the group name `com.cognizant` and the artifact ID `spring-learn`.

I selected Spring Web and Spring Boot DevTools as dependencies, since the goal was to set up a basic web application that supports rapid development with automatic restarts.

Once the project was generated and downloaded as a `.zip` file, I extracted it to my local directory and imported it into Eclipse using the Maven > Existing Maven Projects option. Eclipse automatically recognized the `pom.xml` and configured the project with the appropriate Maven structure. At this point, I attempted to build the project using the `mvn clean package` command.

Initially, I encountered a build failure due to an incompatible Java version.

The default version on my system was Java 1.8, but the Spring Boot version (3.5.3) required Java 17.

To resolve this, I installed Eclipse Temurin JDK 17 and properly set the `JAVA\_HOME` environment variable (excluding the `\bin` directory).

I also updated the `Path` variable to include `%JAVA\_HOME%\bin`.

After restarting the command prompt, I verified the setup using `java -version` and `mvn -version`.

With the environment correctly configured, I ran `mvn clean package` again inside the project directory.

This time, the build completed successfully, producing a `BUILD SUCCESS` message and creating the `.jar` file in the `target` folder.

This confirmed that the project was properly set up and that the development environment was ready for further exercises.

I also explored the structure of the project in Eclipse. The `src/main/java` folder contains the application source code, while `src/main/resources` holds configuration files like `application.properties`.

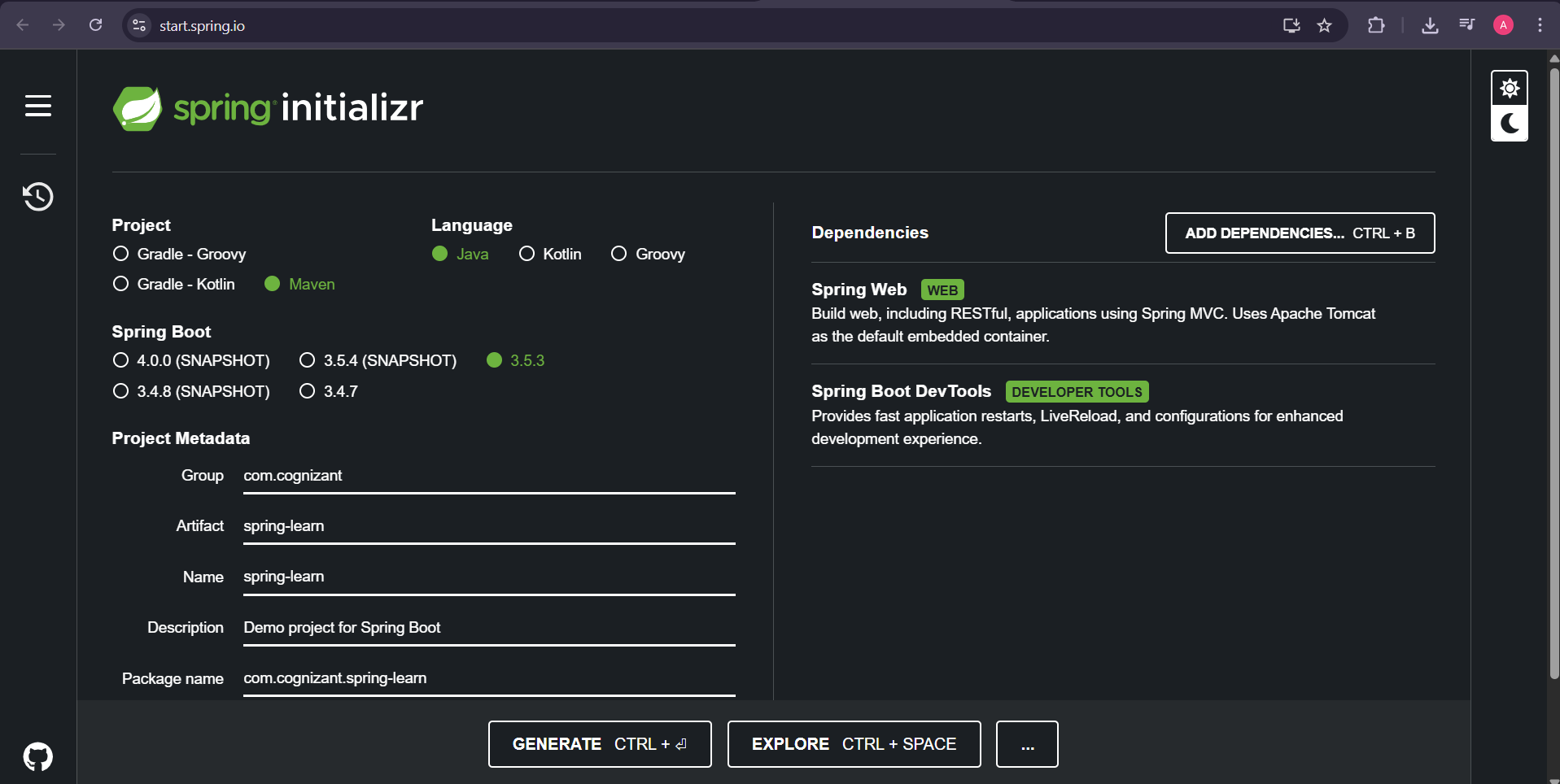
The `SpringLearnApplication.java` class serves as the entry point and is annotated with `@SpringBootApplication`, which combines configuration, component scanning, and auto-configuration.

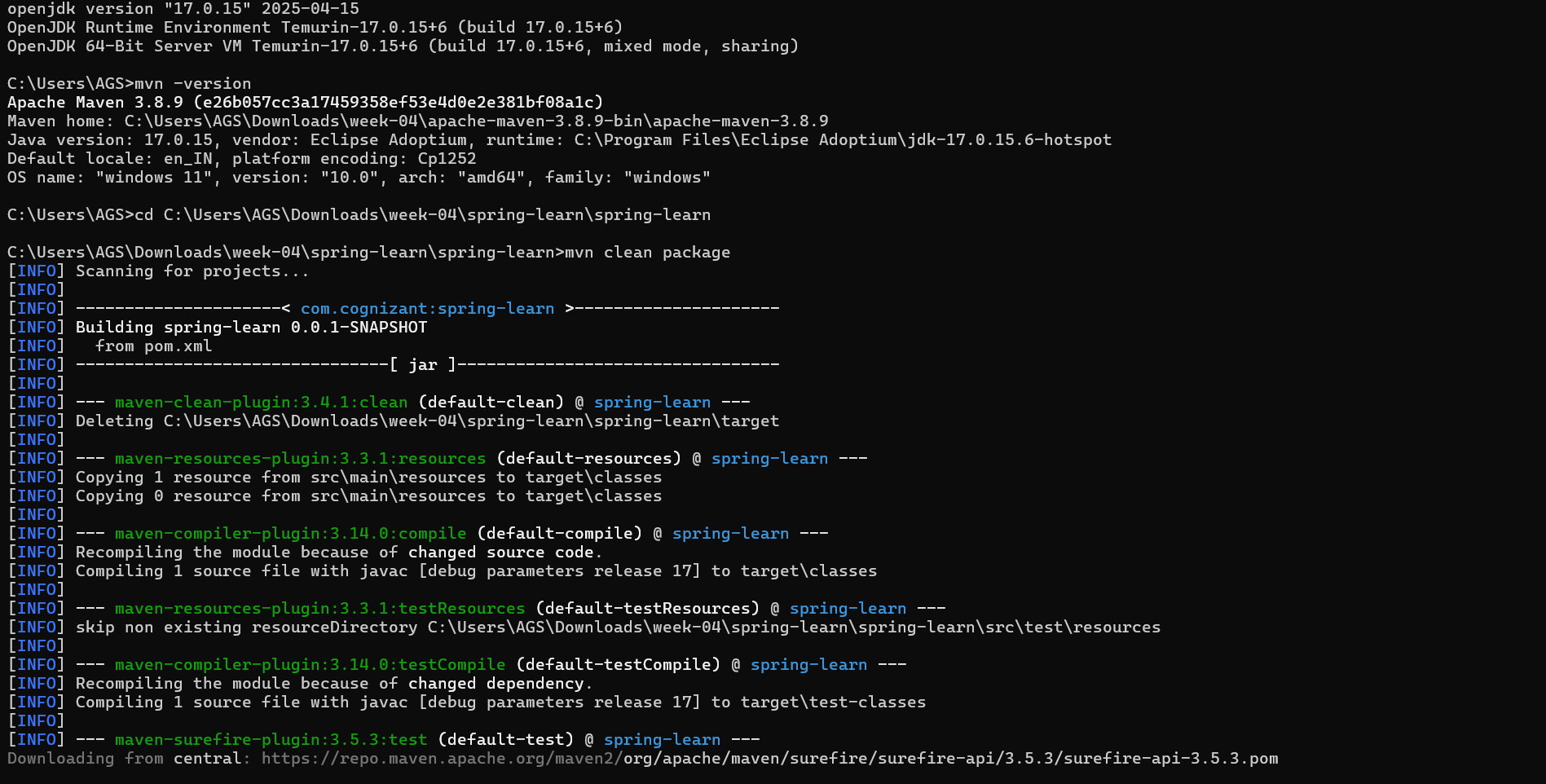
I reviewed the `pom.xml` file to understand how dependencies are managed and viewed the dependency hierarchy in Eclipse to see all the transitive dependencies included with Spring Boot

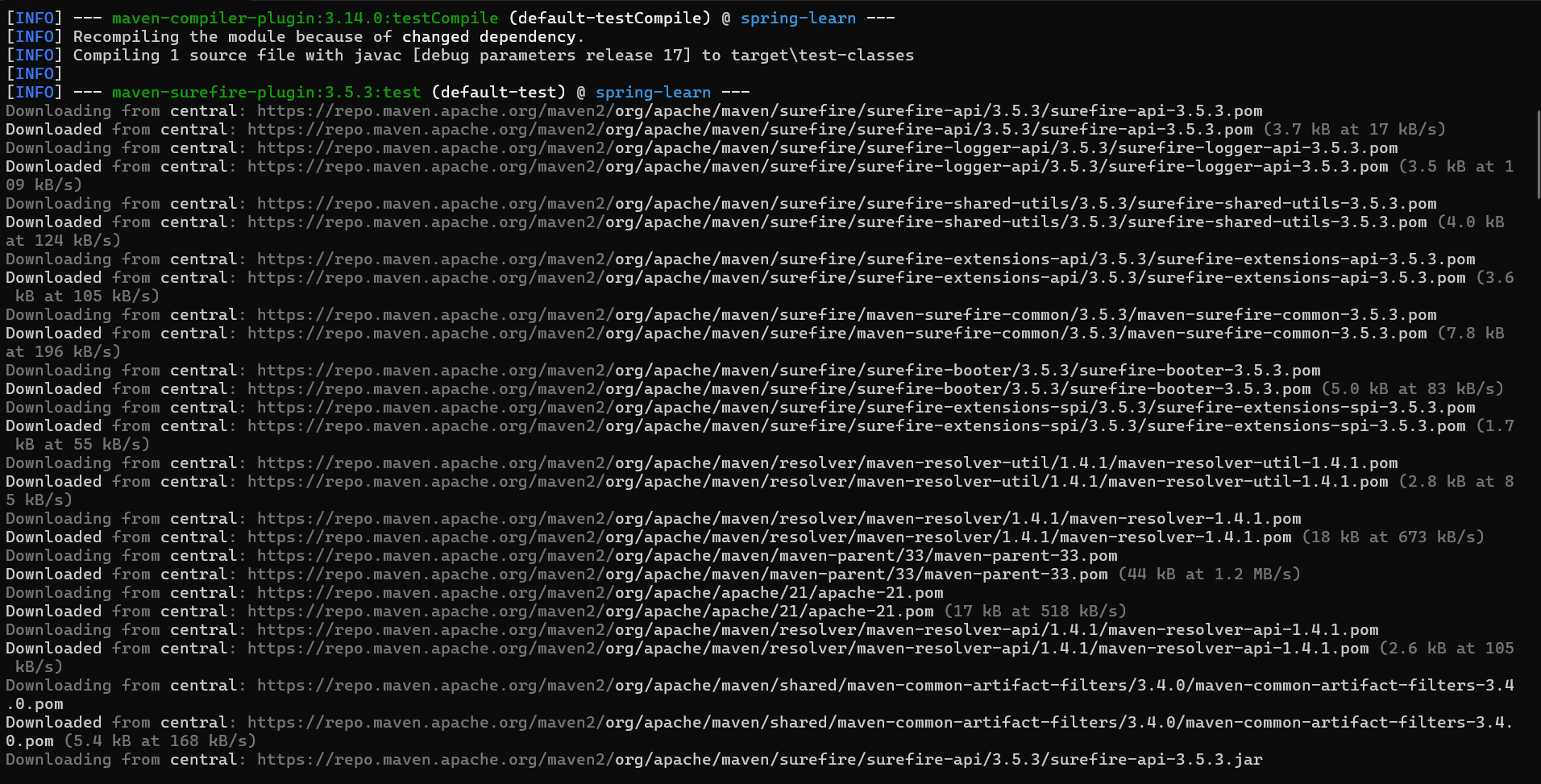
. In the end, I ran the `SpringLearnApplication` class inside Eclipse, which successfully started the embedded Tomcat server and printed the startup logs in the console.

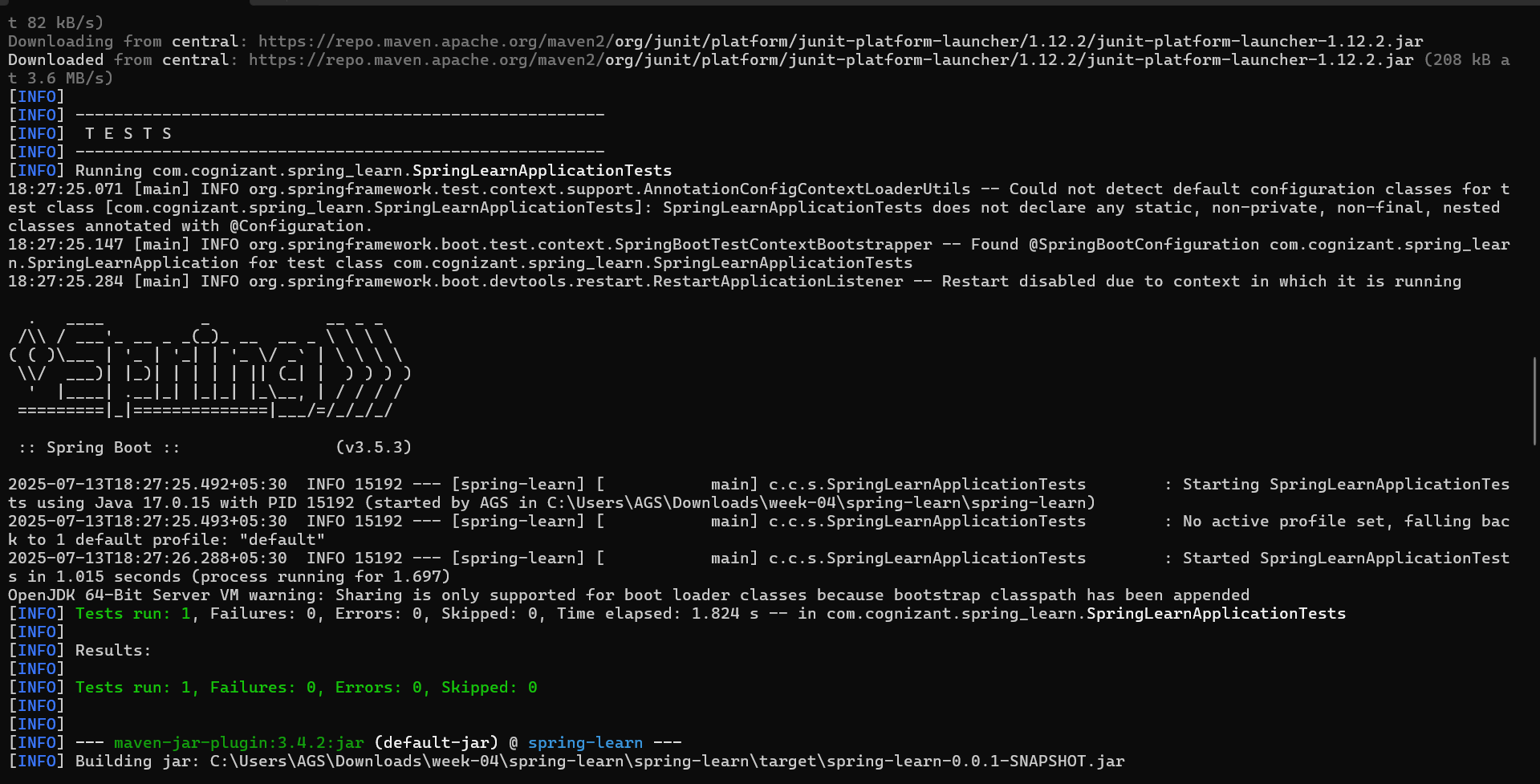
This confirmed that the project was correctly configured and running.

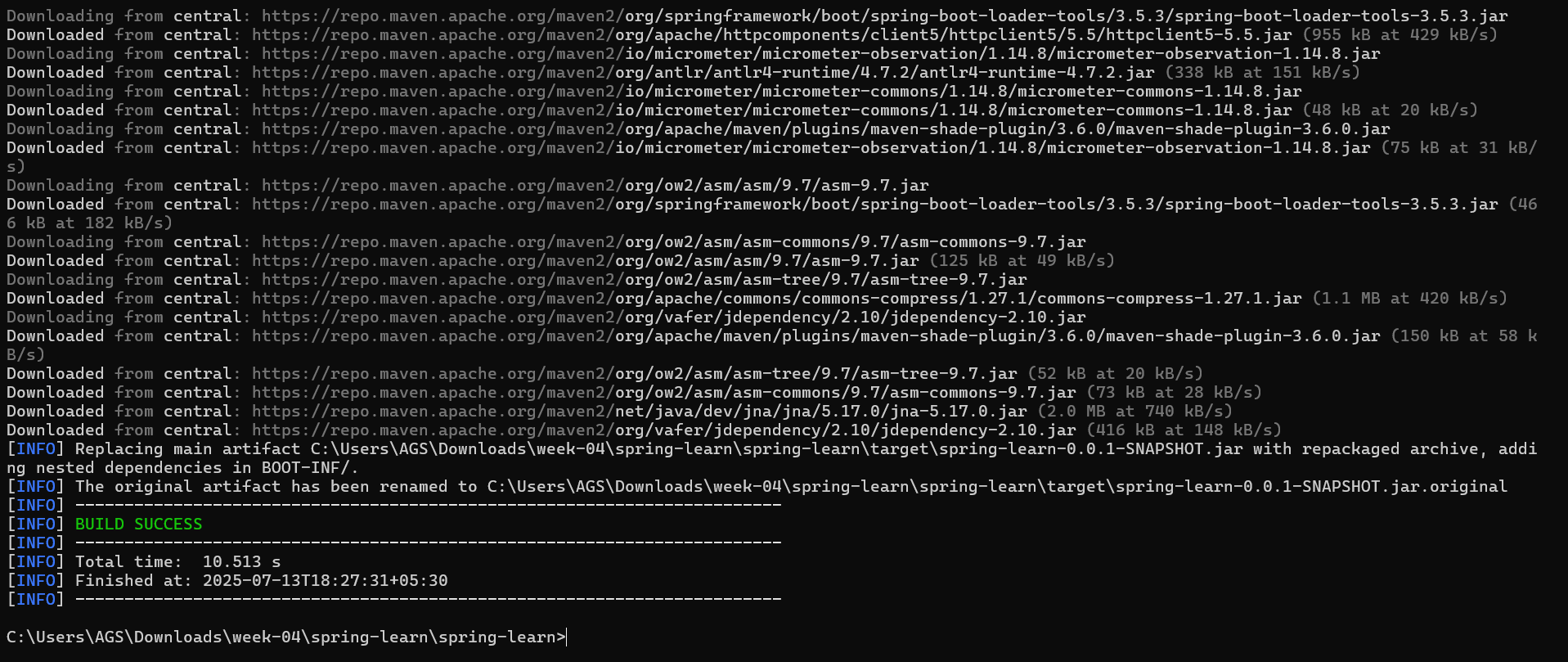
The successful build output, absence of errors, and the running application server together indicated that Exercise 1 was completed as expected.

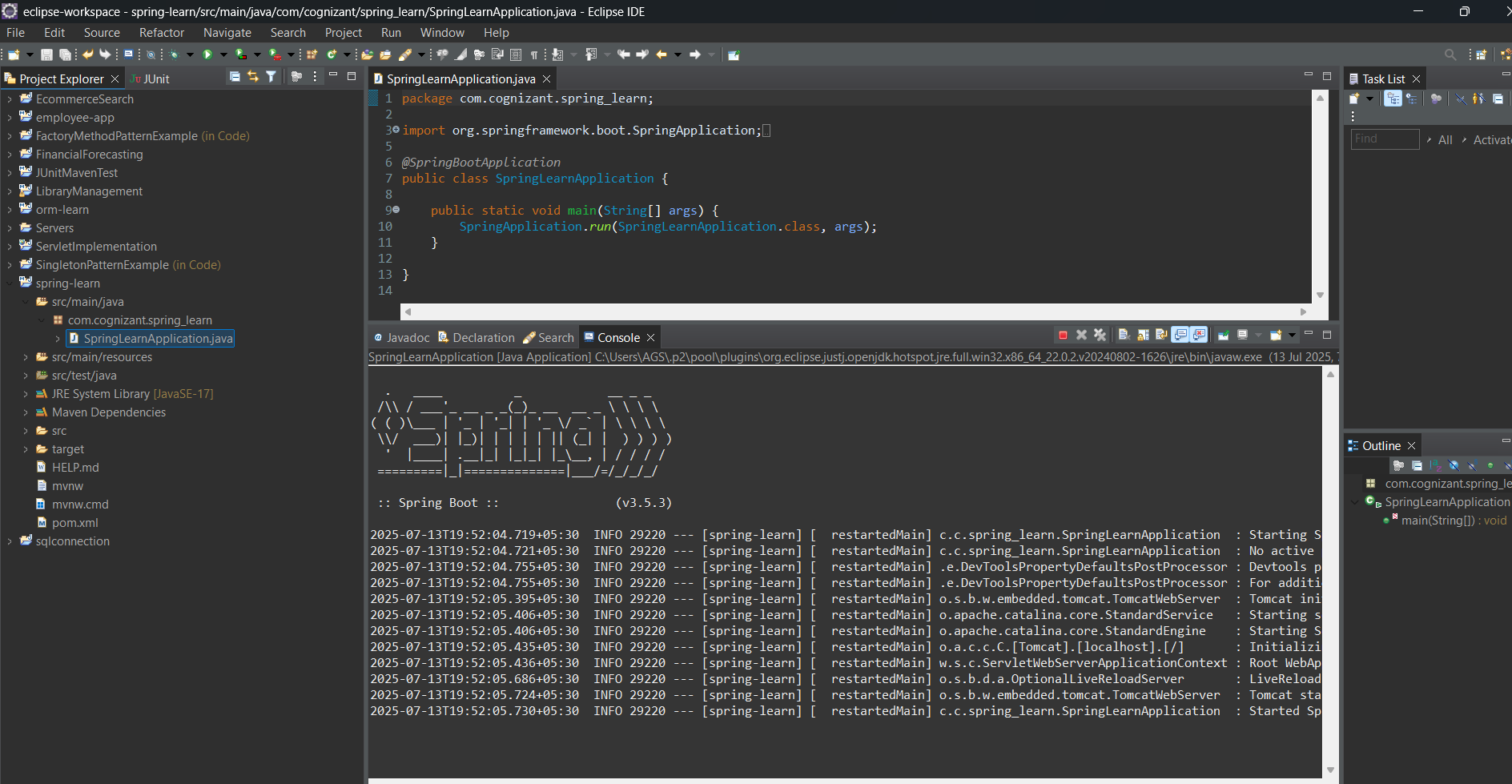
**OUTPUT:** ****

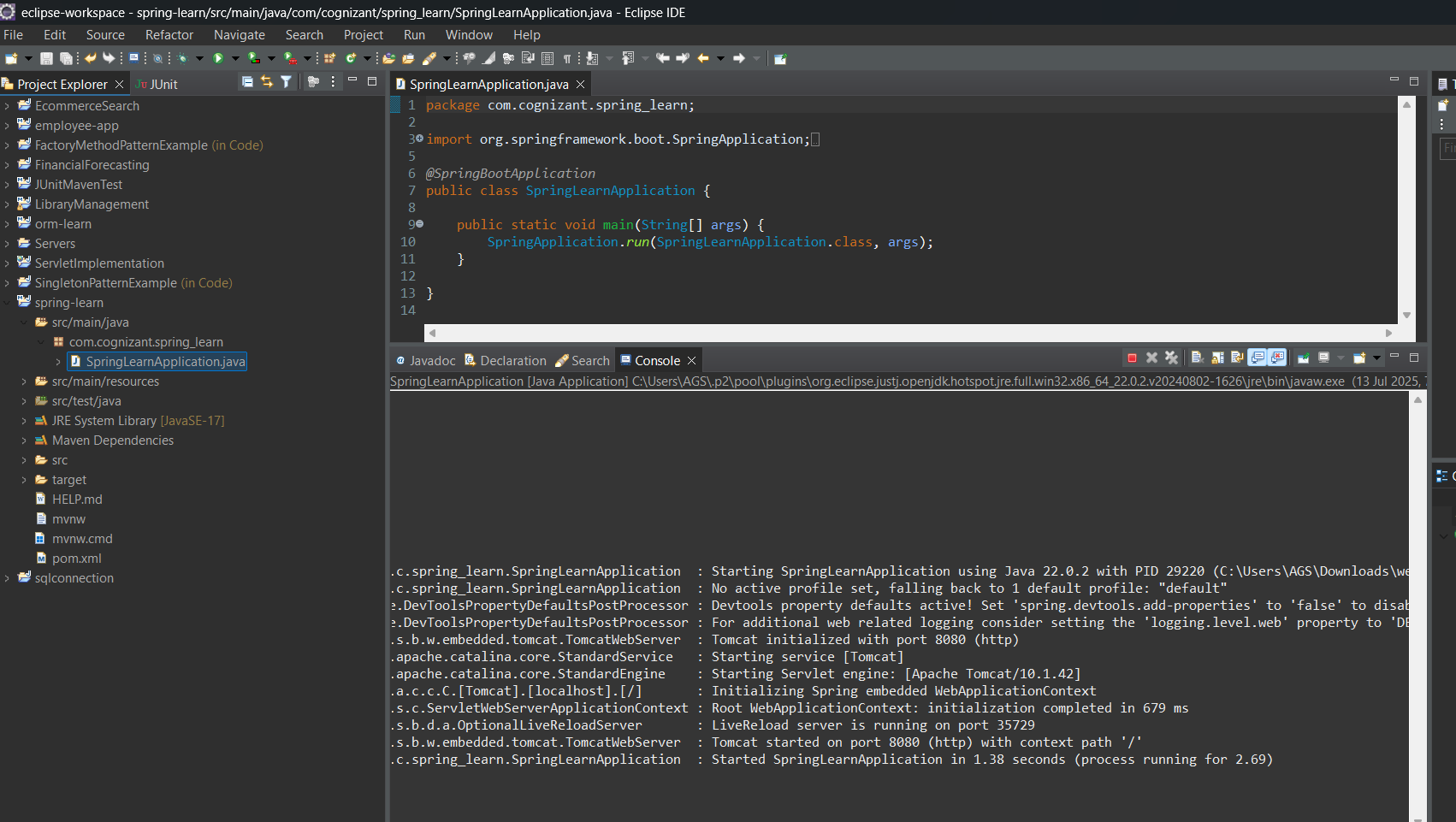
****

****

****

****

****

****

**EXERCISE-02:**

**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  
<bean id="country" class="com.cognizant.springlearn.Country">  
<property name="code" value="IN" />  
<property name="name" value="India" />  
</bean>  
• Create Country class with following aspects:  
o Instance variables for code and name  
o Implement empty parameter constructor with inclusion of debug  
log within the constructor with log message as “Inside Country  
Constructor.”  
o Generate getters and setters with inclusion of debug with relevant  
message within each setter and getter method.  
o 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 conte  
xt.getBean(“beanId”, Country.class). Refer sample code for  
displayCountry() method below.  
ApplicationContext context = new ClassPathXmlApplicationContext("country.xml");  
Country country = (Country) context.getBean("country", Country.class);  
LOGGER.debug("Country : {}", country.toString());  
• 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.

**SOLUTION:**

As part of Exercise-02, I was tasked with creating a Spring Core application that loads and displays country details from a Spring XML configuration.

This exercise simulates a use case where an airline website supports bookings from multiple countries and needs to display country information, such as a two-character ISO code and the full country name, on a dropdown menu.

To begin, I created a Spring configuration file named country.xml, where I defined multiple <bean> entries—one for each country:

India (IN), United States (US), Germany (DE), and Japan (JP).

Each bean was created using the <bean> tag, specifying the id and the fully qualified class attribute for the Country class.

The property tags inside each bean were used to set the values for code and name attributes.

For example, the bean for India was defined with code IN and name India.

I repeated this process for the other countries using appropriate ISO codes and names.

This approach allowed Spring to manage the creation and injection of country objects using configuration, following the Inversion of Control (IoC) principle.

Next, I implemented the Country class in the com.cognizant.spring\_learn package.

This class included two instance variables: code and name, along with an empty constructor that logs the message "Inside Country Constructor" using SLF4J.

For each setter and getter method, I added debug-level log statements to track when they are accessed.

Additionally, I overrode the toString() method to return a readable format of the country object, which helps in logging and displaying the country details.

In the SpringLearnApplication.java class, I created a static method displayCountry() to read the Spring context and retrieve the configured country beans.

I used ApplicationContext and loaded the country.xml using ClassPathXmlApplicationContext.

The method retrieved each country bean by its ID using context.getBean("beanId", Country.class) and printed its details using the logger.

This method was invoked from the main() method to ensure execution on startup.

Upon executing the application, I observed the constructor, setter, and getter logs in the console, confirming that Spring instantiated and injected the beans as expected.

The log output showed the debug statements from each lifecycle step, and the country details were correctly printed, satisfying the requirement of the exercise.

**The key Spring concepts demonstrated in this exercise are as follows:**

**Bean Configuration:** The <bean> tag in the country.xml file is used to define individual objects (beans) that Spring should manage.

The id attribute uniquely identifies each bean within the context, while the class attribute specifies the fully qualified name of the Java class that the bean represents.

In this case, each bean corresponds to a Country object.

**Property Injection**: Inside each <bean> definition, <property> tags are used to set values into the bean’s fields.

The name attribute refers to the name of the Java property (e.g., code, name), and the value attribute assigns the value to that property.

This approach demonstrates how Spring injects dependencies using setter methods during bean initialization.

**ApplicationContext & ClassPathXmlApplicationContext:** The ApplicationContext interface represents the Spring IoC container.

Its implementation ClassPathXmlApplicationContext loads the bean definitions from an XML file located in the classpath.

This allows the application to bootstrap all beans and configurations defined in XML.

**Working of context.getBean():** When the context.getBean("beanId", Class.class) method is invoked, Spring either creates a new bean instance (if it's a prototype) or returns a cached singleton (default behavior).

Spring resolves the bean definition from the XML configuration, performs dependency injection, and returns the fully initialized object to the application.

**OUTPUT:**

