**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.

### **1. Steps to Create a Spring Web Project Using Maven**

#### **Step 1: Go to https://start.spring.io/**

* Visit Spring Initializr, an online tool to bootstrap Spring applications.

#### **Step 2: Change Group as “com.cognizant”**

* In the **Group** field, enter com.cognizant. This will define the base package for your project.

#### **Step 3: Change Artifact Id as “spring-learn”**

* In the **Artifact Id** field, enter spring-learn. This defines the name of the project or artifact generated.

#### **Step 4: Select Spring Boot DevTools and Spring Web**

* Under **Dependencies**, search for and select **Spring Boot DevTools** and **Spring Web**. Spring Boot DevTools provides features for automatic restarts and other developer conveniences, while Spring Web allows you to build web applications (e.g., REST APIs, web pages).

#### **Step 5: Create and Download the Project as a ZIP**

* Once the options are configured, click **Generate**, and a zip file containing your Spring Boot project will be downloaded.

#### **Step 6: Extract the ZIP in Root Folder to Eclipse Workspace**

* Extract the downloaded ZIP file into the root folder of your Eclipse workspace. The project structure will be ready for import.

#### **Step 7: Build the Project Using Maven Command**

In the terminal or command prompt, navigate to the project directory and run the Maven command to build the project. You may need to adjust the proxy settings based on your network configuration.  
  
 mvn clean package -Dhttp.proxyHost=proxy.cognizant.com -Dhttp.proxyPort=6050 -Dhttps.proxyHost=proxy.cognizant.com -Dhttps.proxyPort=6050 -Dhttp.proxyUser=123456

* This command will clean the project, compile the code, and package it as a runnable jar file.

#### **Step 8: Import the Project into Eclipse**

* Open Eclipse IDE, go to **File > Import > Maven > Existing Maven Projects**.
* Click **Browse** and select the extracted folder. Click **Finish** to import the project into Eclipse.

#### **Step 9: Include Logs to Verify the main() Method**

Open the SpringLearnApplication.java file, and include some logging to verify that the main() method is being executed. Add the following to the main method:  
  
 import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

@SpringBootApplication

public class SpringLearnApplication {

private static final Logger logger = LoggerFactory.getLogger(SpringLearnApplication.class);

public static void main(String[] args) {

logger.info("Spring Boot Application Started.");

SpringApplication.run(SpringLearnApplication.class, args);

}

}

* You will need the **SLF4J** logging framework to handle logging in the application.

#### **Step 10: Run the SpringLearnApplication Class**

* Right-click on SpringLearnApplication.java and select **Run As > Java Application**. The Spring Boot application will start, and you should see logs indicating that the application is running.

### **2. Walkthrough the Spring Boot Project Structure**

#### **a. src/main/java - Folder with Application Code**

**src/main/java** contains the Java source code for your application. It includes the main Spring Boot application class (e.g., SpringLearnApplication.java), as well as any controllers, services, or repositories you might create.  
  
 For instance, you might create a controller like this:  
  
 @RestController

@RequestMapping("/hello")

public class HelloController {

@GetMapping

public String sayHello() {

return "Hello, Spring Boot!";

}

}

* This code snippet defines a simple REST endpoint (/hello) that returns a string when accessed.

#### **b. src/main/resources - Folder for Application Configuration**

* **src/main/resources** holds configuration files and other resources used by the application.  
  + **application.properties**: This is the main configuration file for Spring Boot applications. You can specify database connection details, server settings, and other application-level properties.

Example application.properties:  
  
 server.port=8081

spring.application.name=spring-learn

* + **static/**: This folder contains static resources like HTML, CSS, and JavaScript for web applications.
  + **templates/**: If you're using Spring MVC with Thymeleaf or other templating engines, the HTML templates go here.

#### **c. src/test/java - Folder for Testing the Application**

**src/test/java** contains unit tests for your application. Typically, Spring Boot applications will use **JUnit** and **Mockito** for testing.  
  
 For example, to test a controller:  
  
 @RunWith(SpringRunner.class)

@SpringBootTest

public class HelloControllerTests {

@Autowired

private MockMvc mockMvc;

@Test

public void testHelloEndpoint() throws Exception {

mockMvc.perform(get("/hello"))

.andExpect(status().isOk())

.andExpect(content().string("Hello, Spring Boot!"));

}

}

#### **d. SpringLearnApplication.java - Walkthrough of the main() Method**

This is the main entry point of the application. When you run the application, Spring Boot will start up and perform classpath scanning, bean initialization, and application context creation.  
  
 @SpringBootApplication

public class SpringLearnApplication {

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

}

}

* The @SpringBootApplication annotation is a convenience annotation that combines three important annotations:  
  + @Configuration: Marks the class as a source of bean definitions for the application context.
  + @EnableAutoConfiguration: Tells Spring Boot to automatically configure beans based on the project dependencies.
  + @ComponentScan: Tells Spring to scan for other components, configurations, and services in the package where the application is located.

#### **e. Purpose of @SpringBootApplication Annotation**

* The @SpringBootApplication annotation is crucial in a Spring Boot application because it configures your project automatically and simplifies the setup. It eliminates the need for verbose configuration in Java classes.

### **3. pom.xml - Walkthrough of the Configuration**

The **pom.xml** file is the configuration file for Maven, a build automation tool. It contains project dependencies, plugin configurations, and other settings.

Here's a basic breakdown of the contents:

<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 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.cognizant</groupId>

<artifactId>spring-learn</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<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>

</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>

* **groupId** and **artifactId** define the project coordinates.
* The **dependencies** section lists the libraries the project requires (e.g., Spring Boot Web, DevTools, Testing).
* The **build** section defines the plugin that helps with packaging the application as a Spring Boot executable JAR.

#### **4. Dependency Hierarchy**

* In Eclipse, you can view the dependency hierarchy to understand how your dependencies are structured:  
  + Right-click on the project, go to **Maven > Show Dependency Hierarchy**.
  + This will show the entire tree of dependencies, including transitive dependencies (dependencies of dependencies).

**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.

### **Hands-on 2: Spring Core – Load SimpleDateFormat from Spring Configuration XML**

### **Introduction**

In this exercise, you will learn how to use **Spring Core** to load a SimpleDateFormat bean from a Spring XML configuration file and retrieve it in your Java code. This approach helps to centralize configuration, avoid redundancy, and enhance maintainability by eliminating the need to create the SimpleDateFormat object in multiple places across the application.

The steps involve:

* Creating a Spring configuration XML file (date-format.xml).
* Defining the SimpleDateFormat bean in the XML.
* Accessing the SimpleDateFormat bean from the Spring container.
* Using the bean to format and parse dates in your application.

### **Step 1: Create the Spring Configuration File (date-format.xml)**

To start, you need to define the Spring configuration in XML format. Spring allows you to configure beans using XML configuration files. This approach is powerful and flexible, letting you define beans, their properties, and dependencies without writing much Java code.

1. **Navigate to src/main/resources folder** in your project (the folder where Spring configuration files are stored).
2. **Create a new file called date-format.xml**.

In the date-format.xml, you will define the SimpleDateFormat bean, which will be used throughout the application. Below is the XML configuration that defines the SimpleDateFormat bean with the pattern dd/MM/yyyy:

<?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">

<!-- Define SimpleDateFormat bean -->

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

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

</bean>

</beans>

### **Explanation:**

* **XML Declaration**: The first line defines the XML version and encoding. This is a standard header for XML files.
* **beans element**: This is the root element, indicating the start of the Spring configuration.
* **xmlns and xsi:schemaLocation**: These attributes define the namespace and schema location for Spring's XML configuration format.
* **bean element**: The <bean> tag defines the SimpleDateFormat bean. The id attribute is used to uniquely identify this bean in the Spring container (in this case, "dateFormat").  
  + The class attribute specifies the class type of the bean, which is SimpleDateFormat in this case.
  + The <constructor-arg> element specifies the constructor argument for the SimpleDateFormat. The value "dd/MM/yyyy" sets the date format pattern.

### **Step 2: Referencing the Spring Core Documentation**

Before proceeding further, you can review the official Spring Core documentation for more insights on Spring beans and XML configuration. Here’s the link to the Spring Framework Reference documentation:

Spring Framework Reference - Core

This section explains the various bean definitions, configuration metadata, and different ways to configure beans in Spring. While not essential for this particular example, it's a valuable resource if you wish to dive deeper into the Spring Core concepts.

### **Step 3: Define a Method (displayDate()) in SpringLearnApplication.java**

In the next step, you need to modify the main Spring Boot application class (SpringLearnApplication.java) to access the SimpleDateFormat bean from the Spring container and use it to parse a date.

1. **Open the SpringLearnApplication.java file** located in src/main/java/com/cognizant/springlearn.
2. **Add a new method called displayDate()** to retrieve and use the SimpleDateFormat bean from the Spring container.

Here is how the updated SpringLearnApplication.java class would look:

package com.cognizant.springlearn;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.text.SimpleDateFormat;

import java.util.Date;

@SpringBootApplication

public class SpringLearnApplication {

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

// Call displayDate() to display the parsed date

displayDate();

}

public static void displayDate() {

// Load the Spring configuration file

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

// Retrieve the SimpleDateFormat bean from the Spring context

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

// Parse the date string using SimpleDateFormat

try {

String dateString = "31/12/2018";

Date date = dateFormat.parse(dateString);

// Output the result

System.out.println("Parsed Date: " + date);

} catch (Exception e) {

e.printStackTrace();

}

}

}

### **Explanation of Code:**

* **ApplicationContext context**: This object loads the Spring configuration and provides access to the beans defined in the date-format.xml.
* **getBean()**: This method retrieves the bean with the ID "dateFormat" from the application context. The SimpleDateFormat.class argument ensures that the returned bean is of type SimpleDateFormat.
* **dateFormat.parse()**: This method parses the string "31/12/2018" into a Date object using the SimpleDateFormat bean.
* **Exception Handling**: We wrap the parsing code in a try-catch block to handle potential parsing errors.

### **Step 4: Run the Application**

Now that everything is set up, you can run the application and check the output in the console.

1. **Right-click** on the SpringLearnApplication.java file in Eclipse.
2. **Select "Run As" > "Java Application"**.

You should see output similar to the following:

sql

Parsed Date: Mon Dec 31 00:00:00 IST 2018

The SimpleDateFormat bean correctly parses the date string "31/12/2018" and converts it to a Date object.

### **Step 5: Troubleshooting Tips**

#### **If Tomcat Port Conflicts with Existing Services**

If the Tomcat server fails to start due to port conflicts, you can change the default port in the application.properties file. Here's how:

1. Open the **application.properties** file in the src/main/resources folder.
2. Add the following property to change the default port:

server.port=8081

This will configure the application to run on port 8081 instead of the default port 8080.

**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 createdin src/main/resources folder
* Add below lines in application.properties

logging.level.org.springframework=info

logging.level.com.cognizant.springlearn=debug

logging.pattern.console=%d{yyMMdd}|%d{HH:mm:ss.SSS}|%-20.20thread|%5p|%-25.25logger{25}|%25M|%m%n

* In SpringLearnApplication.java include the following imports:

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

* Include the below static variable in SpringLearnApplication.java:

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

* Include info log on start and end of method. Debug log for displaying the date (refer code below)

public void displayDate() {

    LOGGER.info(“START”);

    //..

    LOGGER.debug(date);

    //..

    LOGGER.info(“END”);

}

**IMPORTANT NOTE:** Going forward all methods should incorporate logging as specified above. **Never** use System.out.println().

### **Hands-on 3: Spring Core - Incorporate Logging**

In this task, we will enhance the Spring Boot project by incorporating logging functionality. Proper logging allows developers to track the application's behavior, especially when debugging and troubleshooting. We will use **SLF4J** (Simple Logging Facade for Java) with **Logback** (the default logging framework in Spring Boot) to log messages in different levels.

### **Step 1: Create application.properties if Not Already Created**

First, we need to create the application.properties file in the src/main/resources folder if it does not already exist. This file will be used to configure the logging levels and formats for different packages within the application.

1. **Navigate to src/main/resources**.
2. **Create or open application.properties**.
3. **Add the following logging configuration** to control the logging levels and pattern:

# Set the logging level for Spring Framework

logging.level.org.springframework=info

# Set the logging level for the application package

logging.level.com.cognizant.springlearn=debug

# Set the logging pattern for console output

logging.pattern.console=%d{yyMMdd}|%d{HH:mm:ss.SSS}|%-20.20thread|%5p|%-25.25logger{25}|%25M|%m%n

### **Explanation of the Configuration:**

* **logging.level.org.springframework=info**: This configures the Spring Framework to log messages at the info level or higher.
* **logging.level.com.cognizant.springlearn=debug**: This sets the logging level for your own application (com.cognizant.springlearn) to debug, meaning all debug-level messages and above will be logged.
* **logging.pattern.console**: This defines the format of log messages in the console. The pattern specifies the layout of the log entries:  
  + %d{yyMMdd}: Date in yyMMdd format.
  + %d{HH:mm:ss.SSS}: Time in HH:mm:ss.SSS format.
  + %-20.20thread: Thread name, limited to 20 characters.
  + %5p: Log level (INFO, DEBUG, ERROR, etc.).
  + %-25.25logger{25}: Logger name, truncated to 25 characters.
  + %25M: Method name.
  + %m%n: Log message followed by a newline.

### **Step 2: Include Logger Imports in SpringLearnApplication.java**

Next, we need to modify the SpringLearnApplication.java to incorporate logging.

1. **Open SpringLearnApplication.java**.
2. **Add the following imports** at the top of the class:

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

These imports allow us to use the SLF4J Logger interface to create logger instances in our application.

### **Step 3: Define Logger Variable**

In the SpringLearnApplication.java file, we will define a LOGGER static variable that will be used to log messages in the class.

1. **Add the following static logger variable** inside the SpringLearnApplication class:

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

This will create a logger named after the SpringLearnApplication class, which will be used to log messages.

### **Step 4: Modify displayDate() Method to Include Logging**

Now, let's modify the displayDate() method to add logging at the start and end of the method, and also log the parsed date at the debug level.

1. **Modify the displayDate() method** to incorporate logging as shown below:

public void displayDate() {

// Log the start of the method

LOGGER.info("START");

try {

// Parsing date

String dateString = "31/12/2018";

Date date = dateFormat.parse(dateString);

// Log the parsed date (debug level)

LOGGER.debug("Parsed Date: {}", date);

} catch (Exception e) {

// Log the error (error level)

LOGGER.error("Error parsing date", e);

}

// Log the end of the method

LOGGER.info("END");

}

### **Explanation of the Logging in displayDate():**

* **LOGGER.info("START")**: This logs the start of the displayDate() method at the INFO level.
* **LOGGER.debug("Parsed Date: {}", date)**: This logs the parsed date at the DEBUG level. The {} placeholder is used to insert the value of the date object.
* **LOGGER.error("Error parsing date", e)**: This logs any error that occurs during the date parsing process at the ERROR level. The exception e is passed as a second argument to log the stack trace.
* **LOGGER.info("END")**: This logs the end of the method execution at the INFO level.

### **Step 5: Verify Logging Behavior**

Now, let's test the logging functionality by running the application.

1. **Run the Spring Boot Application** as a **Java Application** in your IDE (Eclipse or IntelliJ).
2. **Check the console output** for log messages. You should see log messages with different log levels (INFO, DEBUG, ERROR) based on the configuration in application.properties.

Example console output:

pgsql

18/07/22|10:20:05.123|main |INFO |SpringLearnApplication | displayDate |START

18/07/22|10:20:05.125|main |DEBUG|SpringLearnApplication | displayDate |Parsed Date: Mon Dec 31 00:00:00 IST 2018

18/07/22|10:20:05.128|main |INFO |SpringLearnApplication | displayDate |END

### **Step 6: Apply Logging in Other Methods**

The logging format introduced above should be incorporated into all methods in your project. For example, if you have another method like someOtherMethod(), you would implement logging as follows:

public void someOtherMethod() {

LOGGER.info("START");

// Do some work here

LOGGER.debug("Some debug information");

// Handle exceptions

try {

// Some logic here

} catch (Exception e) {

LOGGER.error("Error occurred in someOtherMethod", e);

}

LOGGER.info("END");

}

### **Step 7: Avoid Using System.out.println()**

In production code, **never use System.out.println()** for logging or debugging, as it is not efficient and does not provide the flexibility of a logging framework. Always use SLF4J logging as shown above for consistent, configurable, and performant logging.

**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

    <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:
  + 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 countrydetails. ClassPathXmlApplicationContext, ApplicationContext and context.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

### **Hands-on 4: Spring Core – Load Country from Spring Configuration XML**

In this task, we will build a simple application that loads country data (like country code and name) from a Spring XML configuration file and displays it. The configuration will be done using a Spring Bean, which will allow us to inject the country details into the Java application. We will also incorporate debug-level logging to trace the execution of constructors and methods.

Let's break this down into clear, detailed steps.

### **Step 1: Create the Spring XML Configuration File (country.xml)**

The first step is to create the Spring XML configuration file, which will define a Country bean with the required properties (country code and name).

1. **Navigate to src/main/resources**.
2. **Create a new file named country.xml**.

In the country.xml file, we will define a bean with two properties: code (for the ISO country code) and name (for the full name of the country).

Here is the configuration file content:

<?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">

<!-- Define the Country bean for India -->

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

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

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

</bean>

</beans>

### **Explanation of the XML Configuration:**

* **<bean> tag**: Defines the Country bean. It has an id of country, meaning you will refer to this bean as "country" in the Spring container.
* **class="com.cognizant.springlearn.Country"**: Specifies the class of the bean, which is Country in the com.cognizant.springlearn package.
* **<property> tag**: Sets the values for the code and name properties of the Country class. The name attribute is used to specify the property name in the class, and the value attribute is used to specify the value to inject into that property.

### **Step 2: Create the Country Class**

Now, let's create the Country class, which will contain the code and name properties. This class will have the following methods:

* **A constructor** that logs a debug message when invoked.
* **Getters and setters** that log messages when called.
* **A toString() method** to display the country details.

Here is the Country.java class:

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;

// Empty constructor

public Country() {

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

}

// Getter for code

public String getCode() {

LOGGER.debug("Getting Code: {}", code);

return code;

}

// Setter for code

public void setCode(String code) {

LOGGER.debug("Setting Code: {}", code);

this.code = code;

}

// Getter for name

public String getName() {

LOGGER.debug("Getting Name: {}", name);

return name;

}

// Setter for name

public void setName(String name) {

LOGGER.debug("Setting Name: {}", name);

this.name = name;

}

// toString method to display the country details

@Override

public String toString() {

return "Country{" +

"code='" + code + '\'' +

", name='" + name + '\'' +

'}';

}

}

### **Explanation of the Country Class:**

* **Logger (LOGGER)**: We use SLF4J's Logger interface to log messages. This helps in tracking the execution flow (e.g., constructor calls, setter and getter invocations).
* **Constructor (Country())**: The constructor logs a debug message "Inside Country Constructor." to indicate that the Country object is being created.
* **Getters and Setters**: Each getter and setter logs a debug message indicating the value being fetched or set.
* **toString() method**: This method returns a string representation of the Country object, which includes the code and name properties.

### **Step 3: Create the displayCountry() Method in SpringLearnApplication.java**

Now, we will modify the SpringLearnApplication.java file to load the Country bean from the Spring context and display its details.

1. **Open SpringLearnApplication.java** in the src/main/java/com/cognizant/springlearn directory.
2. **Add the displayCountry() method** to load the Country bean and display its details using toString().

Here is the updated 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;

@SpringBootApplication

public class SpringLearnApplication {

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

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

// Call displayCountry() to display the country details

displayCountry();

}

public static void displayCountry() {

// Load the Spring configuration file and retrieve the country bean

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

// Retrieve the Country bean from the context

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

// Log and display the country details

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

}

}

### **Explanation of displayCountry():**

* **ApplicationContext context = new ClassPathXmlApplicationContext("country.xml");**: This line loads the Spring context using the country.xml configuration file.
* **context.getBean("country", Country.class)**: This retrieves the Country bean with the ID "country". The second argument specifies that we expect a Country object.
* **LOGGER.debug("Country : {}", country.toString());**: This logs the string representation of the Country object, displaying the details of the country (code and name).

### **Step 4: Run the Application**

Now that everything is set up, let's run the application and check the logs.

1. **Run the Spring Boot Application** by right-clicking on the SpringLearnApplication.java file and selecting **Run As > Java Application**.
2. **Check the console output** for log messages. You should see something like this:

pgsql

18/07/22|10:20:05.123|main |DEBUG|SpringLearnApplication | displayCountry |Country : Country{code='IN', name='India'}

This indicates that the Country bean was successfully loaded and its details were displayed.

### **Step 5: Explanation of Key Concepts**

#### **Bean Tag (<bean>) in Spring XML Configuration**

* **id attribute**: Uniquely identifies the bean in the Spring container. In this case, the id="country" is used to refer to the Country bean.
* **class attribute**: Specifies the fully qualified class name of the bean. In this case, com.cognizant.springlearn.Country is the class that Spring will instantiate.
* **<property> tag**: Used to inject values into the bean's properties. The name attribute corresponds to the property name in the Java class (like code and name), and the value attribute holds the value to be injected.

#### **ApplicationContext and ClassPathXmlApplicationContext**

* **ApplicationContext**: This is the Spring container that holds all the beans in your application. It manages the lifecycle and configuration of beans.
* **ClassPathXmlApplicationContext**: This is a concrete implementation of ApplicationContext that loads beans from an XML configuration file, such as country.xml.

#### **What Happens When context.getBean() is Invoked**

* **context.getBean()** retrieves the bean with the specified ID from the Spring container.  
  + If the bean is not already instantiated, Spring will create it using the configuration in country.xml.
  + Spring will then inject the properties (code and name) into the Country bean.
  + Once the bean is created, it is returned to the calling method, where you can use it (in this case, logging its details).

**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.

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

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

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

* 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.

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

* Run the application
* Constructor will be called twice, which means that two instances of country is created.

### **Hands-on 5: Spring Core – Demonstration of Singleton Scope and Prototype Scope**

In this task, we will demonstrate the concepts of **Singleton Scope** and **Prototype Scope** in Spring using the Country bean that we created in the previous hands-on. Spring allows us to define the scope of a bean, and understanding these scopes is important to controlling the lifecycle and visibility of beans within the application.

### **Concepts:**

* **Singleton Scope**:  
  + This is the default scope in Spring. When a bean is defined with a singleton scope, the same instance of the bean is returned every time it is requested from the application context. This means the bean is created once, and its instance is shared throughout the application.
* **Prototype Scope**:  
  + When a bean is defined with a prototype scope, a new instance of the bean is created each time it is requested from the application context. Therefore, every time you request a prototype-scoped bean, a new object is instantiated.

### **Step 1: Demonstrate Singleton Scope**

To demonstrate the Singleton scope, you will retrieve the same Country bean multiple times from the application context, and you will see that the constructor is only called once, meaning only one instance of the Country bean is created.

#### **Modify country.xml to Use Singleton Scope (default scope)**

By default, Spring uses the Singleton scope, so we don't need to explicitly specify the scope for a Singleton-scoped bean. However, for demonstration purposes, let's ensure that the Country bean is configured as a singleton.

Here is the country.xml configuration file (no need to change the bean scope for Singleton, since it's the default):

<?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">

<!-- Define the Country bean for India (Singleton Scope) -->

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

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

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

</bean>

</beans>

#### **Modify the SpringLearnApplication.java to Demonstrate Singleton Scope**

Now, modify the displayCountry() method in SpringLearnApplication.java to get the Country bean twice from the application context. Since the bean is singleton, you should get the same instance both times.

Here is the updated 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;

@SpringBootApplication

public class SpringLearnApplication {

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

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

// Call displayCountry() to demonstrate Singleton scope

displayCountry();

}

public static void displayCountry() {

// Load the Spring configuration file and retrieve the country bean

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

// Get the Country bean for the first time

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

// Get the Country bean for the second time

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

// Log both instances

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

LOGGER.debug("Country (second instance) : {}", anotherCountry.toString());

}

}

### **What Happens in Singleton Scope?**

* When you run this, the constructor of the Country class will be called only once, indicating that Spring created only one instance of the Country bean, and both country and anotherCountry refer to the same instance.

### **Expected Logs:**

pgsql

18/07/22|10:20:05.123|main |DEBUG|SpringLearnApplication | displayCountry |Country (first instance) : Country{code='IN', name='India'}

18/07/22|10:20:05.125|main |DEBUG|SpringLearnApplication | displayCountry |Country (second instance) : Country{code='IN', name='India'}

You will notice that both instances of country and anotherCountry are the same, and no new instance was created.

### **Step 2: Demonstrate Prototype Scope**

To demonstrate the **Prototype scope**, we need to change the scope of the Country bean in the country.xml configuration file. When you set the scope to prototype, Spring will create a new instance of the Country bean each time it is requested.

#### **Modify country.xml for Prototype Scope**

Now, let's modify the country.xml to use the **Prototype scope**:

<?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">

<!-- Define the Country bean for India (Prototype Scope) -->

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

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

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

</bean>

</beans>

The key change here is the **scope="prototype"** attribute, which tells Spring to create a new instance of the Country bean every time it is requested.

#### **Run the Application Again to Demonstrate Prototype Scope**

Now, run the application again, but this time, since the Country bean is defined with the prototype scope, Spring will create a new instance each time we call getBean().

Here’s the code again, which remains the same as in the singleton scope demonstration:

public static void displayCountry() {

// Load the Spring configuration file and retrieve the country bean

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

// Get the Country bean for the first time

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

// Get the Country bean for the second time

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

// Log both instances

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

LOGGER.debug("Country (second instance) : {}", anotherCountry.toString());

}

### **What Happens in Prototype Scope?**

* The constructor of the Country class will be called twice, indicating that Spring is creating two different instances of the Country bean, one for each getBean() call.

### **Expected Logs:**

pgsql

18/07/22|10:20:05.123|main |DEBUG|SpringLearnApplication | displayCountry |Country (first instance) : Country{code='IN', name='India'}

18/07/22|10:20:05.125|main |DEBUG|SpringLearnApplication | displayCountry |Country (second instance) : Country{code='IN', name='India'}

In this case, you will notice that even though the code and name are the same for both instances, the instances are different because Spring created a new Country object each time.

### **Step 3: Run the Application and Verify**

When you run the application with **Prototype scope**, you will see in the logs that the constructor is called twice. This is because, with the **Prototype scope**, each getBean() call results in a new instance of the Country bean being created.

**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.

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

        <constructor-arg>

            <list>

                <ref bean="in"></ref>

                <ref bean="us"></ref>

                <ref bean="de"></ref>

                <ref bean="jp"></ref>

            </list>

        </constructor-arg>

    </bean>

* 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

### **Hands-on 6: Spring Core – Load List of Countries from Spring Configuration XML**

In this task, we will configure Spring to load a list of countries from an XML configuration and then display the list using a debug log. The countries are represented by Country beans, and we will store these beans in a list that will be injected into the application context.

### **Step 1: Modify country.xml to Define Multiple Country Beans**

You need to define individual beans for each of the four countries (United States, Germany, India, and Japan) in the Spring XML configuration file (country.xml). Each bean will represent a country with its ISO code and name. You will also define an ArrayList to hold all these Country beans.

#### **Step 1.1: Define Country Beans in country.xml**

Here is how you can define the four Country beans (in, us, de, jp) in the country.xml configuration file:

<?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">

<!-- Define individual Country beans -->

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

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

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

</bean>

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

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

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

</bean>

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

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

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

</bean>

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

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

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

</bean>

<!-- Define a list of countries -->

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

<constructor-arg>

<list>

<ref bean="in" />

<ref bean="us" />

<ref bean="de" />

<ref bean="jp" />

</list>

</constructor-arg>

</bean>

</beans>

### **Explanation of the Configuration:**

* **<bean> for each country**: We define beans for each country (in, us, de, jp) with their respective properties code (ISO code) and name (full country name).
* **<list> tag**: This tag is used to define a list of beans, which will hold the references to the individual country beans. This list will be injected into the countryList bean.
* **<ref> tag**: This tag is used to refer to other beans. Each <ref> element is a reference to a bean defined earlier in the configuration file. The bean attribute specifies the ID of the bean you want to reference.

### **Step 2: Modify SpringLearnApplication.java to Display the List of Countries**

Now, let's add a method displayCountries() in the SpringLearnApplication.java file to retrieve the list of countries from the Spring context and log the details.

#### **Step 2.1: Define the displayCountries() Method**

Here is the updated SpringLearnApplication.java file:

package com.cognizant.springlearn;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.List;

@SpringBootApplication

public class SpringLearnApplication {

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

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

// Call displayCountries() to display the list of countries

displayCountries();

}

public static void displayCountries() {

// Log start of the method

LOGGER.info("START - Display Countries");

// Load the Spring context and get the countryList bean

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

// Retrieve the list of countries

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

// Log each country's details

for (Country country : countries) {

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

}

// Log end of the method

LOGGER.info("END - Display Countries");

}

}

### **Explanation of displayCountries() Method:**

* **Log Start and End**: We add LOGGER.info() statements to log the start and end of the displayCountries() method.
* **ApplicationContext and ClassPathXmlApplicationContext**: We load the Spring context using the country.xml file, which contains the configuration for our Country beans and the countryList.
* **context.getBean("countryList")**: We retrieve the countryList bean from the Spring context, which contains a list of Country objects.
* **for loop**: We iterate over the List<Country> and log the details of each country using country.toString(). This will print the country code and name.

### **Expected Log Output:**

When you run the application, you should see logs similar to the following:

pgsql

18/07/22|10:20:05.123|main |INFO |SpringLearnApplication | displayCountries |START - Display Countries

18/07/22|10:20:05.124|main |DEBUG|SpringLearnApplication | displayCountries |Country : Country{code='IN', name='India'}

18/07/22|10:20:05.125|main |DEBUG|SpringLearnApplication | displayCountries |Country : Country{code='US', name='United States'}

18/07/22|10:20:05.126|main |DEBUG|SpringLearnApplication | displayCountries |Country : Country{code='DE', name='Germany'}

18/07/22|10:20:05.127|main |DEBUG|SpringLearnApplication | displayCountries |Country : Country{code='JP', name='Japan'}

18/07/22|10:20:05.128|main |INFO |SpringLearnApplication | displayCountries |END - Display Countries

You will see debug logs for each country, including its code and name, as well as info-level logs indicating the start and end of the method execution.

### **Step 3: Understanding Key Aspects of the Configuration**

Let's break down the Spring XML configuration and understand some important tags:

#### **<list> Tag**

* The <list> tag is used to define a collection of items. In this case, it's used to define a list of beans that we want to inject into another bean (in this case, the countryList bean).
* Each item inside the <list> is a <ref> tag, which refers to a bean that was previously defined in the Spring configuration file.

#### **<ref> Tag**

* The <ref> tag is used to reference another bean defined in the Spring context. The bean attribute specifies the ID of the bean that we want to refer to.
* In our case, <ref bean="in" /> refers to the Country bean with the ID in.

#### **bean Attribute**

* The bean attribute in both the <bean> and <ref> tags is used to identify the bean. Each bean in Spring needs a unique identifier (the id attribute) that can be referenced later, either in another bean configuration (via <ref>) or programmatically in the application code.

### **Conclusion**

In this task, we:

1. Created individual beans for each of the four countries in the country.xml configuration.
2. Defined an ArrayList bean (countryList) that holds references to these country beans.
3. Implemented the displayCountries() method to read the list of countries from the Spring context and log their details.
4. Explained the usage of the <list>, <ref>, and bean attributes in Spring XML configuration.

This approach allows us to manage a collection of related beans (like the list of countries) in a clean and efficient way using Spring's dependency injection and XML configuration.