**Hands on 1**

**Create a Spring Web Project using Maven:**

**Solution:**

SpringLearnApplication.java :-

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);

}

}

ApplicationTest:

package com.cognizant.spring\_learn;

import org.junit.jupiter.api.Test;

import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest

class SpringLearnApplicationTests {

@Test

void contextLoads() {

}

}

APPLICATION PROPERTIES:

spring.application.name=spring-learn

server.port = 8112

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>

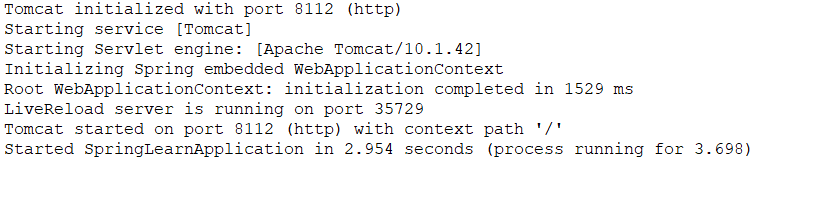
</plugin>

</plugins>

</build>

</project>

OUTPUT:



**Hands on 4**

**Spring Core – Load Country from Spring Configuration XML**

**Solution:**

Country.java:

**package** com.cognizant.spring\_learn;

**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 + "]";

}

}

SpringLearnApplication.java:

**package** com.cognizant.spring\_learn;

**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** displayCountry() {

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

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

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

}

**public** **static** **void** main(String[] args) {

***LOGGER***.debug("START");

*displayCountry*();

***LOGGER***.debug("END");

}

}

The <**bean**> tag in Spring XML is used to define an object that the Spring container will manage, which is referred to as a Spring bean. The id attribute within the bean tag provides a unique name for that bean so that it can be referenced later in the application using methods like context.getBean("id"). The class attribute specifies the fully qualified name of the Java class that Spring will instantiate and manage as a bean. Within the bean tag, we use the <property> tag to inject values into the properties of the Java class, enabling dependency injection through XML. The name attribute inside the property tag indicates the name of the property in the Java class (matching the JavaBean setter), while the value attribute provides the specific value that will be injected into that property during the bean initialization process. For example, using <property name="code" value="IN"/> will result in Spring internally calling the setCode("IN") method on the created bean object.

The **ApplicationContext** in Spring is an advanced container interface responsible for managing the complete lifecycle of beans, dependency injection, event propagation, message resource handling for internationalization, and loading resources. It is a more feature-rich interface compared to BeanFactory and is commonly used in all Spring applications for bean management. ClassPathXmlApplicationContext is a concrete implementation of ApplicationContext that loads the Spring configuration XML file from the classpath, typically placed under src/main/resources. When we create an object using new ClassPathXmlApplicationContext("country.xml"), Spring parses the XML configuration, creates all beans defined within it, manages their lifecycle, injects the required dependencies, and makes these beans ready for use throughout the application.

When context.**getBean**() is invoked, Spring first looks up the bean definition using the provided id within its internal container. It then ensures the bean matches the expected type provided in the method call. If the bean has not been instantiated yet, Spring will instantiate it using reflection, calling the appropriate constructor of the class specified in the XML configuration. It then injects all the dependencies specified using property tags or constructor arguments into the bean and finally returns the fully initialized bean instance to the caller. This allows developers to retrieve and use fully managed and dependency-injected objects throughout their application without explicitly managing their creation and dependencies, promoting loose coupling and clean architecture within Spring-based projects.