**Objectives**

**Explain in detail about HTTP Request and Response**

HTTP (Hypertext Transfer Protocol) is the foundation of communication on the web, where a client (like a browser) sends an HTTP request, and the server responds with an HTTP response. An HTTP request includes a request line with the method (e.g., GET), the URL path (e.g., /index.html), and the HTTP version. It also contains headers like Content-Type, which specifies the format of the data, and User-Agent, which identifies the browser or client making the request. The HTTP response from the server includes a status line with the status code (e.g., 200 OK), response headers (such as Content-Type), and the actual content (HTML, JSON, or image). The request URL indicates the specific resource being requested, and the method defines the type of action (GET, POST, etc.) the client wants to perform. Understanding this structure helps developers analyze and debug web interactions effectively.

**Explain the need and benefits of RESTful Web Services**

RESTful Web Services are essential in modern web development because they allow systems to communicate over the internet using standard HTTP methods like GET, POST, PUT, and DELETE. The need for REST arises from the demand for scalable, stateless, and easily maintainable architectures that can support multiple platforms including web, mobile, and cloud-based applications. One of the key benefits of REST is its simplicity—it uses standard web protocols and data formats like JSON or XML, making it lightweight and easy to integrate. RESTful services are stateless, meaning each request is independent, which improves performance and scalability. Additionally, REST supports separation of concerns, allowing client and server to evolve independently, and enabling easier testing, versioning, and deployment. Its language-independent nature makes it ideal for building APIs that can be consumed by any technology.

**Demonstrate implementation of RESTful Web Service using GET method**

#### 1. ****Controller Class****

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

@RestController@RequestMapping("/api")public class GreetingController {

@GetMapping("/greeting")

public String getGreeting() {

return "Hello, welcome to RESTful Web Service!";

}

}

#### 2. ****Main Application Class****

import org.springframework.boot.SpringApplication;import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplicationpublic class RestDemoApplication {

public static void main(String[] args) {

SpringApplication.run(RestDemoApplication.class, args);

}

}

**How It Works:**

The @RestController annotation makes this class a REST API controller.

@RequestMapping("/api") sets the base URL path.

@GetMapping("/greeting") maps HTTP GET requests to /api/greeting.

When you run the application and go to http://localhost:8080/api/greeting, it responds with:  
➤ "Hello, welcome to RESTful Web Service!"

**Demonstrate implementation of end to end testing of RESTful Web Service using MockMVC**

## 1. REST Controller – GreetingController.java

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

@RestController@RequestMapping("/api")public class GreetingController {

@GetMapping("/greeting")

public String getGreeting() {

return "Hello from RESTful Web Service!";

}

}

2. Spring Boot Application Class – RestApp.java

import org.springframework.boot.SpringApplication;import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplicationpublic class RestApp {

public static void main(String[] args) {

SpringApplication.run(RestApp.class, args);

}

}

3. Test Class Using MockMvc – GreetingControllerTest.java

import org.junit.jupiter.api.Test;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

@WebMvcTest(GreetingController.class)public class GreetingControllerTest {

@Autowired

private MockMvc mockMvc;

@Test

public void testGreetingEndpoint() throws Exception {

mockMvc.perform(get("/api/greeting"))

.andExpect(status().isOk())

.andExpect(content().string("Hello from RESTful Web Service!"));

}

}

Explanation:

@WebMvcTest(GreetingController.class) is used to load only the **web layer** (controller) for testing.

MockMvc is injected to **simulate HTTP requests** without starting a real server.

perform(get("/api/greeting")) simulates a **GET request** to the endpoint.

.andExpect(status().isOk()) checks the HTTP response status is 200 OK.

.andExpect(content().string(...)) checks that the returned content matches the expected response.

**HTTP Request Response**   
HTTP (HyperText Transfer Protocol) request and response are fundamental to how communication happens between a client, such as a web browser, and a web server. When a user enters a URL or interacts with a web application, the browser constructs and sends an HTTP request to the server, which in turn sends back an HTTP response.

A typical HTTP request consists of several lines. The first line includes three important components: the method type, which is GET in this case, indicating that the client wants to retrieve data; the resource, which is /hello.txt, specifying the file being requested; and the HTTP version being used, which is HTTP/1.1. The second line of the request includes the User-Agent header, which provides information about the client software or browser making the request. The third line contains the Host header, identifying the server to which the request is directed—in this case, [www.example.com](http://www.example.com" \t "_new). Additionally, the Accept-Language header tells the server which languages the client prefers in the response. When a user enters a URL in the browser like https://www.example.com/hello.txt, it is split into Host as [www.example.com](http://www.example.com" \t "_new) and Resource as /hello.txt in the actual HTTP request.

After receiving the request, the server responds with an HTTP response. A standard HTTP response also consists of multiple lines. The first line contains the HTTP version (HTTP/1.1), a status code (200), and a status message (OK), indicating that the request was successfully processed. The second line includes the date and time when the response was generated. The following lines contain various headers, such as Server, which reveals the server software used (Apache in this case), Last-Modified, which indicates when the resource was last changed, and Content-Length, which specifies the size of the response content. The Content-Type header, found in one of the last lines before the response body, tells the browser how to interpret the content. For example, text/plain means the response is plain text, text/html indicates HTML content, application/json refers to JSON data, and image/png indicates that an image in PNG format is being returned. The final line of the response is the body itself, which contains the actual content the client requested—here it is a text message saying, “Hello World! My payload includes a trailing CRLF.”

To observe these HTTP request and response details in a real browser, one can use Chrome Developer Tools. By pressing F12 and going to the Network tab, the user can load a website and click on any request listed under the Name column. This opens a panel on the right side showing detailed information under three main sections: General, which shows the method, URL, and status; Request Headers, which displays all headers sent by the browser; and Response Headers, which show what the server returned. This feature is especially useful for debugging and understanding how web communication works under the hood.

**Hello World RESTful Web Service**

### 1. ****Create the Controller Class****

**File:** HelloController.java  
**Package:** com.cognizant.spring\_learn.controller

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;

@RestControllerpublic class HelloController {

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

@GetMapping("/hello")

public String sayHello() {

LOGGER.info("START - sayHello()");

String message = "Hello World!!";

LOGGER.info("END - sayHello()");

return message;

}

}

2. **Main Application Class**

**File:** SpringLearnApplication.java  
(Already created in your Spring Learn project)

package com.cognizant.spring\_learn;

import org.springframework.boot.SpringApplication;import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplicationpublic class SpringLearnApplication {

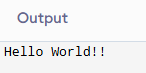
public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

}

}

**OUTPUT:**



**REST - Country Web Service**

### 1. ****Create**** Country ****Model****

#### Country.java

package com.cognizant.springlearn.model;

public class Country {

private String code;

private String name;

// Constructors

public Country() {}

public Country(String code, String name) {

this.code = code;

this.name = name;

}

// Getters and Setters

public String getCode() {

return code;

}

public void setCode(String code) {

this.code = code;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

2. **Create** CountryController

#### CountryController.java

package com.cognizant.springlearn.controller;

import java.util.List;import org.springframework.context.ApplicationContext;import org.springframework.context.support.ClassPathXmlApplicationContext;import org.springframework.web.bind.annotation.GetMapping;import org.springframework.web.bind.annotation.RestController;

import com.cognizant.springlearn.model.Country;

@RestControllerpublic class CountryController {

@GetMapping("/countries")

public List<Country> getAllCountries() {

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

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

return countries;

}

}

3. **Create** country.xml **in** src/main/resources

#### 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.model.Country">

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

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

</bean>

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

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

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

</bean>

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

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

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

</bean>

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

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

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

</bean>

</list>

</constructor-arg>

</bean>

</beans>

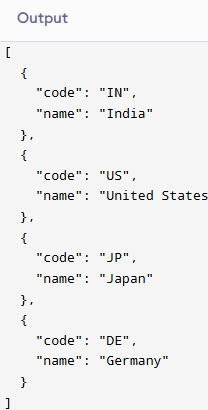
Make sure this file is in the resources folder so that it’s on the classpath.

4. **Change Port to 8083**

In src/main/resources/application.properties:

server.port=8083

**OUTPUT:**



**REST - Get country based on country code**

### 1. Country.java (Already Exists)

Make sure your model class is present in:  
 com.cognizant.springlearn.model.Country

public class Country {

private String code;

private String name;

// constructors, getters, setters

}

2. country.xml

Make sure it's in:  
src/main/resources/country.xml

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

<constructor-arg>

<list>

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

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

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

</bean>

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

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

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

</bean>

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

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

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

</bean>

</list>

</constructor-arg></bean>

3. Create the Service Class

CountryService.java

package com.cognizant.springlearn.service;

import java.util.List;import org.springframework.context.ApplicationContext;import org.springframework.context.support.ClassPathXmlApplicationContext;import org.springframework.stereotype.Service;

import com.cognizant.springlearn.model.Country;

@Servicepublic class CountryService {

public Country getCountry(String code) {

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

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

// Case-insensitive search using lambda

return countries.stream()

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

.findFirst()

.orElse(null); // or throw exception if not found

}

}

4. Update Controller

CountryController.java

package com.cognizant.springlearn.controller;

import org.springframework.beans.factory.annotation.Autowired;import org.springframework.web.bind.annotation.\*;

import com.cognizant.springlearn.model.Country;import com.cognizant.springlearn.service.CountryService;

@RestControllerpublic class CountryController {

@Autowired

private CountryService countryService;

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

public Country getCountry(@PathVariable String code) {

return countryService.getCountry(code);

}

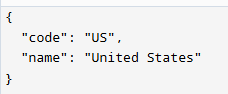
}

5. Update application.properties (if needed)

src/main/resources/application.properties

server.port=8083

**OUTPUT:**



**REST - Get country exceptional scenario**

## Create the Custom Exception

Create a new exception class in the package:  
com.cognizant.springlearn.service.exception.CountryNotFoundException.java

package com.cognizant.springlearn.service.exception;

import org.springframework.http.HttpStatus;import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value = HttpStatus.NOT\_FOUND, reason = "Country not found")public class CountryNotFoundException extends RuntimeException {

// You can include additional constructors if needed

public CountryNotFoundException() {

super();

}

public CountryNotFoundException(String message) {

super(message);

}

}

2. Update the Service Method to Throw the Exception

Update CountryService.getCountry(String code) so that if no matching country is found, the custom exception is thrown.

### File: CountryService.java

package com.cognizant.springlearn.service;

import java.util.List;import org.springframework.context.ApplicationContext;import org.springframework.context.support.ClassPathXmlApplicationContext;import org.springframework.stereotype.Service;import com.cognizant.springlearn.model.Country;import com.cognizant.springlearn.service.exception.CountryNotFoundException;

@Servicepublic class CountryService {

public Country getCountry(String code) {

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

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

// Use a lambda expression for case-insensitive search

Country country = countries.stream()

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

.findFirst()

.orElse(null);

if (country == null) {

// Throw the custom exception if not found

throw new CountryNotFoundException();

}

return country;

}

}

3. Update the Controller

Modify CountryController so that the getCountry method declares it throws the exception.

### File: CountryController.java

package com.cognizant.springlearn.controller;

import org.springframework.beans.factory.annotation.Autowired;import org.springframework.web.bind.annotation.\*;import com.cognizant.springlearn.model.Country;import com.cognizant.springlearn.service.CountryService;import com.cognizant.springlearn.service.exception.CountryNotFoundException;

@RestControllerpublic class CountryController {

@Autowired

private CountryService countryService;

// Get country by code; throws CountryNotFoundException if not found.

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

public Country getCountry(@PathVariable String code) throws CountryNotFoundException {

return countryService.getCountry(code);

}

}

4. Testing the Exceptional Scenario

### Using a Browser

**Sample Request URL:**  
http://localhost:8083/countries/az

If you send a GET request to this URL and no country with code "az" exists, Spring Boot will return a response like:

{

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

"status": 404,

"error": "Not Found",

"message": "Country not found",

"path": "/countries/az"}

### Using cURL Command

Open Git Bash or any terminal and execute:

bash

CopyEdit

curl -i http://localhost:8083/countries/az

**OUTPUT:**



**MockMVC - Test get country service**

### 1. ****Add JUnit & MockMvc Annotations****

File: SpringLearnApplicationTests.java

package com.cognizant.springlearn;

import static org.junit.Assert.assertNotNull;import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;

import org.junit.Test;import org.junit.runner.RunWith;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;import org.springframework.boot.test.context.SpringBootTest;import org.springframework.test.context.junit4.SpringRunner;import org.springframework.test.web.servlet.MockMvc;import org.springframework.test.web.servlet.ResultActions;

import com.cognizant.springlearn.controller.CountryController;

@RunWith(SpringRunner.class)@SpringBootTest@AutoConfigureMockMvcpublic class SpringLearnApplicationTests {

// Check if controller is loaded

@Autowired

private CountryController countryController;

// Inject MockMvc

@Autowired

private MockMvc mvc;

// Test if the controller is loaded into context

@Test

public void contextLoads() {

assertNotNull(countryController);

}

// Test the GET /country service

@Test

public void testGetCountry() throws Exception {

ResultActions actions = mvc.perform(get("/country")); // Adjust path if needed

actions.andExpect(status().isOk());

actions.andExpect(jsonPath("$.code").exists());

actions.andExpect(jsonPath("$.code").value("IN"));

actions.andExpect(jsonPath("$.name").exists());

actions.andExpect(jsonPath("$.name").value("India"));

}

}

2. Important Notes

Make sure your /country endpoint exists and returns

{

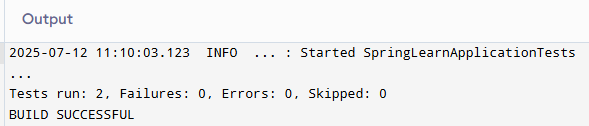
"code": "IN",

"name": "India"}

If you’re using /countries/IN instead, change:

mvc.perform(get("/countries/IN"));

**OUTPUT:**



**MockMVC - Test get country service for exceptional scenario**

### Update SpringLearnApplicationTests.java:

Add this test method:

@Testpublic void testGetCountryException() throws Exception {

ResultActions actions = mvc.perform(get("/countries/xyz")); // Non-existent code

actions.andExpect(status().isNotFound());

actions.andExpect(status().reason("Country not found"));

}

Explanation:

get("/countries/xyz") – hits the controller with a country code that does **not exist** in country.xml.

status().isNotFound() – expects HTTP 404.

status().reason("Country not found") – matches the custom reason from @ResponseStatus.

**OUTPUT:**

