Activity 8.1: Spring Remoting “HTTP-based Communication with RestTemplate”

This activity will guide you through implementing HTTP-based communication using Spring's RestTemplate. You will create two separate Spring Boot applications: a **Server** that exposes a REST API, and a **Client** that consumes it using RestTemplate.

In this activity, we will focus on using RestTemplate to demonstrate client-server communication over HTTP.

STEP 1: Project Setup (Server Application - REST API)

First, let's set up the Spring Boot application that will expose our REST endpoints.

1. **Go to Spring Initializr:** Open your web browser and navigate to https://start.spring.io/.
2. **Configure Your Project:**
   - **Project:** Maven Project
   - **Language:** Java
   - **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
   - **Group:** com.example.restcomm
   - **Artifact:** rest-server
   - **Name:** rest-server
   - **Description:** Spring REST Server Demo
   - **Package Name:** com.example.restcomm.server
   - **Packaging:** Jar
   - **Java:** Choose Java 17 or higher.
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
   - **Spring Web:** Essential for building REST controllers.
   - **Lombok:** (Optional but recommended) Reduces boilerplate code.
4. **Generate and Download:** Click the "Generate" button. Download the .zip file.
5. **Import into IDE:** Unzip the downloaded file and import the project into your IDE (IntelliJ IDEA, Eclipse, VS Code).

STEP 2: Define Data Transfer Objects (DTOs - Shared)

We'll define simple DTOs (Data Transfer Objects) that will be exchanged between the client and server. These classes represent the structure of the data.

- Create a new package com.example.restcomm.shared in src/main/java/ of your rest-server project.
- Create a Java class named GreetingRequest.java inside this package:

```
// src/main/java/com/example/restcomm/shared/GreetingRequest.java
package com.example.restcomm.shared;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

/**
 * DTO for sending a greeting request.
 */
@Data // Lombok: Generates getters, setters, toString, equals, hashCode
@NoArgsConstructor // Lombok: Generates no-argument constructor
@AllArgsConstructor // Lombok: Generates constructor with all fields
public class GreetingRequest {
    private String name;
}
```

- Create a Java class named GreetingResponse.java inside the same com.example.restcomm.shared package:

```
// src/main/java/com/example/restcomm/shared/GreetingResponse.java
package com.example.restcomm.shared;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

/**
```

```
 * DTO for receiving a greeting response.
 */
@Data
@NoArgsConstructor
@AllArgsConstructor
public class GreetingResponse {
    private String message;
    private String timestamp;
}
```

STEP 3: Create REST Controller on Server-Side

Now, let's create the REST controller that will expose endpoints for our client to consume.

- Create a new package com.example.restcomm.controller in src/main/java/ of your rest-server project.
- Create a Java class named GreetingController.java inside this package:

```
// src/main/java/com/example/restcomm/server/controller/GreetingController.java
package com.example.restcomm.controller;

import com.example.restcomm.shared.GreetingRequest;
import com.example.restcomm.shared.GreetingResponse;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

/**
 * REST Controller exposing greeting endpoints.
 */
```

```
@RestController // Marks this as a REST controller
public class GreetingController {

    private static final DateTimeFormatter FORMATTER =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

    /**
     * Handles GET requests to /api/greet.
     * Example: GET http://localhost:8080/api/greet?name=World
     * @param name The name to greet, from request parameter.
     * @return A GreetingResponse object.
     */
    @GetMapping("/api/greet")
    public GreetingResponse getGreeting(@RequestParam(defaultValue = "Guest")
String name) {
        System.out.println("Server: Received GET request for greeting for: " + name);
        String message = "Hello, " + name + " from REST server (GET)!";
        return new GreetingResponse(message,
LocalDateTime.now().format(FORMATTER));
    }

    /**
     * Handles POST requests to /api/greet.
     * Expects a JSON request body with a 'name' field.
     * Example: POST http://localhost:8080/api/greet
     * Body: {"name": "Alice"}
     * @param request The GreetingRequest object from the JSON request body.
     * @return A GreetingResponse object.
     */
    @PostMapping("/api/greet")
    public GreetingResponse postGreeting(@RequestBody GreetingRequest
```

```
request) {
        System.out.println("Server: Received POST request for greeting for: " +
request.getName());
        String message = "Hello, " + request.getName() + " from REST server
(POST)!";
        return new GreetingResponse(message,
LocalDateTime.now().format(FORMATTER));
    }
}
```

STEP 4: Main Server Application Class

This is the standard Spring Boot application entry point for the server.

- Open your main application class (e.g., RestServerApplication.java in com.example.restcomm).

```
// src/main/java/com/example/restcomm/server/RestServerApplication.java
package com.example.restcomm;

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

@SpringBootApplication
public class RestServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(RestServerApplication.class, args);
    }
}
```

STEP 5: Project Setup (Client Application)

Now, let's set up the Spring Boot application that will consume the REST API.

1. **Create a New Project:** Repeat STEP 1, but with the following configurations:
    - **Group:** com.example.restcomm
    - **Artifact:** rest-client
    - **Name:** rest-client

- **Description:** Spring REST Client Demo
- **Package Name:** com.example.restcomm.client
- **Packaging:** Jar
- **Java:** Choose Java 17 or higher.
- **Dependencies: Spring Web** (this includes RestTemplate), **Lombok**.

2. **Import into IDE:** Unzip and import the rest-client project.
3. **Copy the Shared DTOs:**
   - Copy GreetingRequest.java and GreetingResponse.java from rest-server/src/main/java/com/example/restcomm/shared/ to rest-client/src/main/java/com/example/restcomm/client/shared/.
   - **Important:** The package structure (com.example.restcomm.shared) must be identical in both projects.

STEP 6: Configure RestTemplate on Client-Side

We'll configure RestTemplate as a Spring @Bean so it can be easily injected and used.

- Create a new package com.example.restcomm.client.config in src/main/java/ of your rest-client project.
- Create a Java class named ClientConfig.java inside this package:

```
// src/main/java/com/example/restcomm/client/config/ClientConfig.java
package com.example.restcomm.client.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate; // The key class for HTTP client

/**
 * Configuration for creating and customizing RestTemplate.
 */
@Configuration
public class ClientConfig {
```

```
    /**
     * Creates a RestTemplate bean.
     * @return A configured RestTemplate instance.
     */
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

STEP 7: Create a Client Component to Consume REST API

Let's create a simple component that injects RestTemplate and makes calls to the server's REST API.

- Create a new package com.example.restcomm.client.component in src/main/java/ of your rest-client project.
- Create a Java class named RestClientRunner.java inside this package:
  // src/main/java/com/example/restcomm/client/component/RestClientRunner.java
  package com.example.restcomm.client.component;

  import com.example.restcomm.client.shared.GreetingRequest;
  import com.example.restcomm.client.shared.GreetingResponse;
  import org.springframework.boot.CommandLineRunner; // Interface to run code after Spring context is loaded
  import org.springframework.stereotype.Component;
  import org.springframework.web.client.RestTemplate; // For making HTTP requests

  /**
   * A client component that runs after the Spring Boot application starts.
   * It uses RestTemplate to call the remote REST API.
   */
  @Component
  public class RestClientRunner implements CommandLineRunner {

      private final RestTemplate restTemplate; // Spring will inject the RestTemplate bean here
      private static final String SERVER_BASE_URL = "http://localhost:8080/api";

      public RestClientRunner(RestTemplate restTemplate) {
          this.restTemplate = restTemplate;
      }

      @Override
      public void run(String... args) throws Exception {
          System.out.println("\n--- Client Application Started ---");
          System.out.println("Attempting to call remote REST API using RestTemplate...");

          // --- 1. Perform a GET request ---
          try {

```
            System.out.println("\n--- Calling GET endpoint: " + SERVER_BASE_URL +
"/greet?name=John ---");
            String getName = "John";
            // Use getForObject for simple GET requests expecting an object
            GreetingResponse getResponse = restTemplate.getForObject(
                SERVER_BASE_URL + "/greet?name={name}",
                GreetingResponse.class,
                getName
            );
            System.out.println("Client received GET response: " + getResponse);
        } catch (Exception e) {
            System.err.println("Client encountered an error calling GET endpoint: " +
e.getMessage());
        }

        // --- 2. Perform a POST request ---
        try {
            System.out.println("\n--- Calling POST endpoint: " + SERVER_BASE_URL
+ "/greet ---");
            GreetingRequest postRequest = new GreetingRequest("Jane");
            // Use postForObject for POST requests sending an object and expecting
an object
            GreetingResponse postResponse = restTemplate.postForObject(
                SERVER_BASE_URL + "/greet",
                postRequest,
                GreetingResponse.class
            );
            System.out.println("Client received POST response: " + postResponse);
        } catch (Exception e) {
            System.err.println("Client encountered an error calling POST endpoint: " +
e.getMessage());
        }

        System.out.println("\n--- Client Application Finished ---");
    }
}
```

- **Add application.properties to the Client Application:**

1. In your remoting-client project, navigate to src/main/resources/.
2. Create a new file named application.properties (if it doesn't already exist).

3. Add the following line to this application.properties file:

   ```
   # src/main/resources/application.properties

   server.port=8081
   ```

4. Now, when you start the client application, it will attempt to use port 8081 instead of 8080, resolving the port conflict.

STEP 8: Main Client Application Class

This is the standard Spring Boot application entry point for the client.

- Open your main application class (e.g., RestClientApplication.java in com.example.restcomm.client).

```
// src/main/java/com/example/restcomm/client/RestClientApplication.java
package com.example.restcomm.client;

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

@SpringBootApplication
public class RestClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(RestClientApplication.class, args);
    }
}
```

STEP 9: Run Both Applications and Test

To see the communication in action, you need to run both the server and client applications simultaneously.

1. **Start the Server Application:**
   - In your IDE, navigate to RestServerApplication.java (in the rest-server project).
   - Run its main method.
   - You should see Spring Boot starting the server on port 8080 (default). Keep this running.
2. **Start the Client Application:**
   - In a separate instance of your IDE (or a new run configuration), navigate to

RestClientApplication.java (in the rest-client project).

- Run its main method.
- **Observe the console output of both applications.**

Expected Output:

- **Server Console (rest-server):** You should see messages indicating it received GET and POST requests, e.g.:
  Server: Received GET request for greeting for: John
  Server: Received POST request for greeting for: Jane

- **Client Console (rest-client):** You should see messages indicating the responses received from the server, e.g.:
  --- Client Application Started ---
  Attempting to call remote REST API using RestTemplate...

  --- Calling GET endpoint: http://localhost:8080/api/greet?name=John ---
  Client received GET response: GreetingResponse(message=Hello, John from REST server (GET)!, timestamp=YYYY-MM-DD HH:MM:SS)

  --- Calling POST endpoint: http://localhost:8080/api/greet ---
  Client received POST response: GreetingResponse(message=Hello, Jane from REST server (POST)!, timestamp=YYYY-MM-DD HH:MM:SS)

  --- Client Application Finished ---

This confirms that the client successfully communicated with the REST API on the server using RestTemplate.

You have successfully implemented HTTP-based communication using Spring's RestTemplate! You've learned how to:

- Create a simple Spring Boot REST API on the server.
- Define DTOs for data exchange.
- Configure and use RestTemplate on the client-side to make GET and POST

requests.
- Observe the interaction between a client and a server via standard HTTP.

RestTemplate is a versatile tool for consuming various web services and is a fundamental part of building microservices that communicate over HTTP.

Activity 9.1: Spring with JMS “Messaging with ActiveMQ”

This activity will guide you through implementing messaging with Spring and JMS (Java Message Service), using Apache ActiveMQ as the message broker. You will learn how to send messages, receive messages, and implement a request-reply messaging pattern.

STEP 1: Set Up Apache ActiveMQ Broker

Before starting our Spring Boot applications, we need a running JMS broker. Apache ActiveMQ is a popular open-source choice.

1. **Download ActiveMQ:**
   - Go to the Apache ActiveMQ download page: https://activemq.apache.org/components/classic/download/activemq-5-18-3-release-notes (or find the latest stable release).
   - Download the binary distribution (e.g., apache-activemq-5.18.3-bin.zip).
2. **Extract and Start ActiveMQ:**
   - Unzip the downloaded file to a convenient location (e.g., C:\activemq or ~/activemq).
   - Open a command prompt or terminal.
   - Navigate to the bin directory inside the extracted ActiveMQ folder:
     - **Windows:** cd C:\activemq\apache-activemq-5.18.3\bin\win64 (or win32)
     - **Linux/macOS:** cd ~/activemq/apache-activemq-5.18.3/bin
   - Run the ActiveMQ start command:
     - **Windows:** activemq start
     - **Linux/macOS:** ./activemq start
   - You should see output indicating that ActiveMQ has started.
   - **Verify:** Open your web browser and go to ActiveMQ's web console: http://localhost:8161/. The default username/password is admin/admin. You should see the ActiveMQ dashboard. Keep this running throughout the activity.

STEP 2: Project Setup (Spring Boot JMS Application)

We'll create a single Spring Boot application that acts as both a message sender and