API to API Communication: HttpClient  
================================

**Using RestTemplate**

RestTemplate is a synchronous client that simplifies communication with HTTP services and supports RESTful interactions.

1. **Create a Configuration Class** to define your RestTemplate bean:

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.web.client.RestTemplate;

@Configuration

**public** **class** RestTemplateConfig {

@Bean

**public** RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

}

1. **Use RestTemplate in your Service Layer**:

Java

**package** com.k7it.service;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.http.HttpStatus;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.stereotype.Service;

**import** org.springframework.web.client.RestTemplate;

**import** com.k7it.controller.StudentResponse;

**import** com.k7it.model.Student;

**import** com.k7it.model.Tutorial;

**import** com.k7it.repo.StudentRepo;

@Service

**public** **class** StudentService {

@Autowired

**private** StudentRepo studentRepo;

@Autowired

**private** RestTemplate restTemplate;

**public** ResponseEntity<StudentResponse> getStudentsWithTheirTutorials(String id) {

// here we need to get student details by id and get tutorials for that student

//1 . get student details by id

Student student = studentRepo.findById(id).orElse(**null**);

StudentResponse studentResponse = **new** StudentResponse();

**if**(student != **null**) {

studentResponse.setStudent(student);

//2. get the tutorials for given student id from tutorial service

List<Tutorial> tutorials = **null**;

//1. restTemplate

//RestTemplate restTemplate = new RestTemplate();//direct use

String tutorial\_url = "http://localhost:9090/tutorial/student/"+id;

tutorials = restTemplate.getForObject(tutorial\_url, List.**class**);

**if**(tutorials != **null**) {

studentResponse.setTutorials(tutorials);

}

}**else** {

**throw** **new** RuntimeException("No student is found with given ID");

}

**return** **new** ResponseEntity<StudentResponse>(studentResponse, HttpStatus.***OK***);

}

**public** ResponseEntity<Student> createStudent(Student entity) {

**return** ResponseEntity.*ok*(studentRepo.save(entity));

}

}

**Using HttpClient**

HttpClient provides more flexibility and supports both synchronous and asynchronous operations.

1. **Include the dependency** in your pom.xml if you're using Maven:

Xml

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

<version>4.5.13</version>

</dependency>

1. **Create a Configuration Class** for CloseableHttpClient:

Java

**import** org.apache.http.impl.client.CloseableHttpClient;

**import** org.apache.http.impl.client.HttpClients;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

@Configuration

**public** **class** HttpClientConfig {

@Bean

**public** CloseableHttpClient httpClient() {

**return** HttpClients.createDefault();

}

}

1. **Use HttpClient in your Service Layer**:

java

**package** com.k7it.service;

**import** java.util.List;

**import** org.apache.http.client.methods.CloseableHttpResponse;

**import** org.apache.http.client.methods.HttpGet;

**import** org.apache.http.impl.client.CloseableHttpClient;

**import** org.apache.http.impl.client.HttpClients;

**import** org.apache.http.util.EntityUtils;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.http.HttpStatus;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.stereotype.Service;

**import** org.springframework.web.client.RestTemplate;

**import** com.fasterxml.jackson.core.type.TypeReference;

**import** com.fasterxml.jackson.databind.ObjectMapper;

**import** com.k7it.controller.StudentResponse;

**import** com.k7it.model.Student;

**import** com.k7it.model.Tutorial;

**import** com.k7it.repo.StudentRepo;

@Service

**public** **class** StudentService {

@Autowired

**private** StudentRepo studentRepo;

@Autowired

**private** RestTemplate restTemplate;

@Autowired

**private** CloseableHttpClient httpClient;

**public** ResponseEntity<StudentResponse> getStudentsWithTheirTutorials(String id) **throws** Exception {

// here we need to get student details by id and get tutorials for that student

//1 . get student details by id

Student student = studentRepo.findById(id).orElse(**null**);

StudentResponse studentResponse = **new** StudentResponse();

**if**(student != **null**) {

studentResponse.setStudent(student);

//2. get the tutorials for given student id from tutorial service

List<Tutorial> tutorials = **null**;

String tutorial\_url = "http://localhost:9090/tutorial/student/"+id;

//1. restTemplate

//RestTemplate restTemplate = new RestTemplate();

// tutorials = restTemplate.getForObject(tutorial\_url, List.class);

// if(tutorials != null) {

// studentResponse.setTutorials(tutorials);

// }

//

//2. using HttpClient

// CloseableHttpClient httpClient = HttpClients.createDefault();// direct use without configuration class

HttpGet getRequest = **new** HttpGet(tutorial\_url);

CloseableHttpResponse response = httpClient.execute(getRequest);

**if**(response != **null**) {

String jsonStr = EntityUtils.*toString*(response.getEntity());

ObjectMapper mapper = **new** ObjectMapper();

tutorials = mapper.readValue(jsonStr, **new** TypeReference<List<Tutorial>>() {});

studentResponse.setTutorials(tutorials);

}

}**else** {

**throw** **new** RuntimeException("No student is found with given ID");

}

**return** **new** ResponseEntity<StudentResponse>(studentResponse, HttpStatus.***OK***);

}

**public** ResponseEntity<Student> createStudent(Student entity) {

**return** ResponseEntity.*ok*(studentRepo.save(entity));

}

}

RestTemplate and HttpClient without configuration classes. Here’s how you can do it directly within your service classes:

### Using RestTemplate Without Configuration Class

You can directly instantiate RestTemplate within your service class without a separate configuration.

java

**import** org.springframework.stereotype.Service;

**import** org.springframework.web.client.RestTemplate;

@Service

**public** **class** SimpleEmployeeService {

**private** **final** RestTemplate restTemplate;

**public** SimpleEmployeeService() {

**this**.restTemplate = **new** RestTemplate();

}

**public** String getStudentInfo(String studentId) {

String url = "http://student-service/api/students/" + studentId;

**return** restTemplate.getForObject(url, String.**class**);

}

}

### Using HttpClient Without Configuration Class

Similarly, you can instantiate CloseableHttpClient directly in your service class.

java

**import** org.apache.http.client.methods.CloseableHttpResponse;

**import** org.apache.http.client.methods.HttpGet;

**import** org.apache.http.impl.client.CloseableHttpClient;

**import** org.apache.http.impl.client.HttpClients;

**import** org.apache.http.util.EntityUtils;

**import** org.springframework.stereotype.Service;

@Service

**public** **class** SimpleEmployeeService {

**private** **final** CloseableHttpClient httpClient;

**public** SimpleEmployeeService() {

**this**.httpClient = HttpClients.createDefault();

}

**public** String getStudentInfo(String studentId) **throws** Exception {

String url = "http://student-service/api/students/" + studentId;

HttpGet request = **new** HttpGet(url);

**try** (CloseableHttpResponse response = httpClient.execute(request)) {

**return** EntityUtils.toString(response.getEntity());

}

}

}

These approaches allow you to keep everything self-contained within your service class without the need for separate configuration classes.

**Using WebClient**

WebClient is part of Spring WebFlux and supports both synchronous and asynchronous operations.

is it required any dependencies for using Webclient.builder in my springboot project for api to api comuncation ?

**Dependency for WebClient**

Add the following dependency to your pom.xml:

xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

1. **Create a Configuration Class** for WebClient:

Java

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.web.reactive.function.client.WebClient;

@Configuration

**public** **class** WebClientConfig {

@Bean

**public** WebClient.Builder webClientBuilder() {

**return** WebClient.builder();

}

}

1. **Use WebClient in your Service Layer**:

java

**package** com.k7it.service;

**import** java.util.List;

**import** org.apache.http.impl.client.CloseableHttpClient;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.http.HttpStatus;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.stereotype.Service;

**import** org.springframework.web.client.RestTemplate;

**import** org.springframework.web.reactive.function.client.WebClient;

**import** com.k7it.controller.StudentResponse;

**import** com.k7it.model.Student;

**import** com.k7it.model.Tutorial;

**import** com.k7it.repo.StudentRepo;

**import** reactor.core.publisher.Mono;

@Service

**public** **class** StudentService {

@Autowired

**private** StudentRepo studentRepo;

@Autowired

**private** RestTemplate restTemplate;

@Autowired

**private** CloseableHttpClient httpClient;

@Autowired

**private** WebClient.Builder webClientBuilder;

**public** ResponseEntity<StudentResponse> getStudentsWithTheirTutorials(String id) **throws** Exception {

// here we need to get student details by id and get tutorials for that student

// 1 . get student details by id

Student student = studentRepo.findById(id).orElse(**null**);

StudentResponse studentResponse = **new** StudentResponse();

**if** (student != **null**) {

studentResponse.setStudent(student);

// 2. get the tutorials for given student id from tutorial service

List<Tutorial> tutorials = **null**;

String tutorial\_url = "http://localhost:9090/tutorial/student/" + id;

// 1. restTemplate

// RestTemplate restTemplate = new RestTemplate();

// tutorials = restTemplate.getForObject(tutorial\_url, List.class);

// if(tutorials != null) {

// studentResponse.setTutorials(tutorials);

// }

//

// 2. using HttpClient

// CloseableHttpClient httpClient = HttpClients.createDefault();// direct use

// without configuration class

/\*

\* HttpGet getRequest = new HttpGet(tutorial\_url);

\*

\* CloseableHttpResponse response = httpClient.execute(getRequest);

\*

\* if (response != null) { String jsonStr =

\* EntityUtils.toString(response.getEntity()); ObjectMapper mapper = new

\* ObjectMapper(); tutorials = mapper.readValue(jsonStr, new

\* TypeReference<List<Tutorial>>() { });

\* studentResponse.setTutorials(tutorials); }

\*/

// 3. webclient

// WebClient.Builder webClientBuilder = WebClient.builder();// direct usage

Mono<List<Tutorial>> tutorialMono = webClientBuilder.build()

.get()

.uri(tutorial\_url)

.retrieve()

.bodyToFlux(Tutorial.**class**)

.collectList();

tutorials = tutorialMono.block();

**if**(tutorials != **null**) {

studentResponse.setTutorials(tutorials);

}

} **else** {

**throw** **new** RuntimeException("No student is found with given ID");

}

**return** **new** ResponseEntity<StudentResponse>(studentResponse, HttpStatus.***OK***);

}

**public** ResponseEntity<Student> createStudent(Student entity) {

**return** ResponseEntity.*ok*(studentRepo.save(entity));

}

}

**Feign client**

Feign Client is a powerful and declarative REST client that simplifies the process of making HTTP requests in your Spring Boot applications. It integrates seamlessly with Spring Boot and allows you to define API clients with minimal boilerplate code.

### Key Features of Feign Client

1. **Declarative Syntax**: Feign uses a declarative approach, which means you define an interface and annotate it. The Feign library takes care of the rest.
2. **Easy Integration**: Feign integrates well with other Spring Cloud components like Eureka for service discovery, Hystrix for fault tolerance, and Ribbon for load balancing.
3. **Readability and Maintainability**: Code written with Feign Client is more readable and maintainable because it abstracts away much of the boilerplate code associated with making HTTP requests.
4. **Custom Configuration**: You can customize Feign clients with configuration classes to set timeouts, interceptors, and error handling.

### How to Use Feign Client

Here's a step-by-step guide to setting up and using Feign Client in a Spring Boot application:

#### Step 1: Add Dependencies

Add the Spring Cloud Feign dependency to your pom.xml:

<properties>

<java.version>21</java.version>

<spring-cloud.version>2023.0.2</spring-cloud.version>

</properties>

And above <Dependencies> add this dependency management details  
<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<!-- Feign Client Dependency -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

### Step 2: Enable Feign Clients

Enable Feign clients in your Spring Boot application by adding the @EnableFeignClients annotation to your main application class.

java

**package** com.k7it;

**import** org.springframework.boot.SpringApplication;

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

@SpringBootApplication

@EnableFeignClients

**public** **class** SchoolApplication {

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

SpringApplication.*run*(SchoolApplication.**class**, args);

}

}

### Step 3: Create the Feign Client Interface

Define an interface for your Feign client and annotate it with @FeignClient. This interface will contain the methods to call the student service endpoints.

java

**package** com.k7it.config;

**import** java.util.List;

**import** org.springframework.cloud.openfeign.FeignClient;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** com.k7it.model.Tutorial;

@FeignClient(name = "tutorial-service", url = "http://localhost:9090")

**public** **interface** TutorialFeignClient {

@GetMapping("/tutorial/student/{studentId}")

**public** List<Tutorial> getTutorialsByStudentId(@PathVariable String studentId);

}

### Step 4: Use the Feign Client in Your Service

Inject the Feign client into your service class and use it to make API calls to the student service.

java

**package** com.k7it.service;

**import** java.util.List;

**import** org.apache.http.impl.client.CloseableHttpClient;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.http.HttpStatus;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.stereotype.Service;

**import** org.springframework.web.client.RestTemplate;

**import** org.springframework.web.reactive.function.client.WebClient;

**import** com.k7it.config.TutorialFeignClient;

**import** com.k7it.controller.StudentResponse;

**import** com.k7it.model.Student;

**import** com.k7it.model.Tutorial;

**import** com.k7it.repo.StudentRepo;

**import** reactor.core.publisher.Mono;

@Service

**public** **class** StudentService {

@Autowired

**private** StudentRepo studentRepo;

@Autowired

**private** RestTemplate restTemplate;

@Autowired

**private** CloseableHttpClient httpClient;

@Autowired

**private** WebClient.Builder webClientBuilder;

@Autowired

**private** TutorialFeignClient tutorialFeignClient;

**public** ResponseEntity<StudentResponse> getStudentsWithTheirTutorials(String id) **throws** Exception {

// here we need to get student details by id and get tutorials for that student

// 1 . get student details by id

Student student = studentRepo.findById(id).orElse(**null**);

StudentResponse studentResponse = **new** StudentResponse();

**if** (student != **null**) {

studentResponse.setStudent(student);

// 2. get the tutorials for given student id from tutorial service

List<Tutorial> tutorials = **null**;

String tutorial\_url = "http://localhost:9090/tutorial/student/" + id;

// 1. restTemplate

// RestTemplate restTemplate = new RestTemplate();

// tutorials = restTemplate.getForObject(tutorial\_url, List.class);

// if(tutorials != null) {

// studentResponse.setTutorials(tutorials);

// }

//

// 2. using HttpClient

// CloseableHttpClient httpClient = HttpClients.createDefault();// direct use

// without configuration class

/\*

\* HttpGet getRequest = new HttpGet(tutorial\_url);

\*

\* CloseableHttpResponse response = httpClient.execute(getRequest);

\*

\* if (response != null) { String jsonStr =

\* EntityUtils.toString(response.getEntity()); ObjectMapper mapper = new

\* ObjectMapper(); tutorials = mapper.readValue(jsonStr, new

\* TypeReference<List<Tutorial>>() { });

\* studentResponse.setTutorials(tutorials); }

\*/

// 3. webclient

// WebClient.Builder webClientBuilder = WebClient.builder();// direct usage

/\* Mono<List<Tutorial>> tutorialMono = webClientBuilder.build()

.get()

.uri(tutorial\_url)

.retrieve()

.bodyToFlux(Tutorial.class)

.collectList();

tutorials = tutorialMono.block();

if(tutorials != null) {

studentResponse.setTutorials(tutorials);

}\*/

//4. Feign Client

tutorials = tutorialFeignClient.getTutorialsByStudentId(id);

**if**(tutorials != **null**) {

studentResponse.setTutorials(tutorials);

}

} **else** {

**throw** **new** RuntimeException("No student is found with given ID");

}

**return** **new** ResponseEntity<StudentResponse>(studentResponse, HttpStatus.***OK***);

}

**public** ResponseEntity<Student> createStudent(Student entity) {

**return** ResponseEntity.*ok*(studentRepo.save(entity));

}

}

### Step 5: Configuration (Optional)

If you need to customize the Feign client (e.g., adding interceptors, custom configuration), you can create a separate configuration class.

java

**import** feign.Logger;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

@Configuration

**public** **class** FeignConfig {

@Bean

Logger.Level feignLoggerLevel() {

**return** Logger.Level.FULL;

}

}

Complete pom.xml File:  
========================

<?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>3.4.1</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.k7it</groupId>

<artifactId>school</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>school</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>21</java.version>

<spring-cloud.version>2023.0.2</spring-cloud.version>

</properties>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

<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.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<!-- HttpClient Dependency -->

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

<version>4.5.13</version>

</dependency>

<!-- WebClient Dependency -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

<!-- Feign Client Dependency -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<annotationProcessorPaths>

<path>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</path>

</annotationProcessorPaths>

</configuration>

</plugin>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<excludes>

<exclude>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</exclude>

</excludes>

</configuration>

</plugin>

</plugins>

</build>

</project>

Comparison of **HttpClient**, **WebClient**, and **FeignClient**:

| **Feature** | **RestTemplate** | **HttpClient** | **WebClient** | **FeignClient** |
| --- | --- | --- | --- | --- |
| **Type** |  | Synchronous | Non-blocking (Reactive) | Declarative REST Client |
| **Library** |  | Apache HttpClient | Spring WebFlux | Spring Cloud |
| **Programming Model** |  | Blocking | Asynchronous | Synchronous |
| **Use Case** |  | Simple and advanced HTTP requests | Handling multiple requests concurrently | Simplifying REST client creation |
| **Performance** |  | Can be resource-intensive with many requests | Efficient with fewer resources | Can create multiple threads for each request |
| **Ease of Use** |  | Requires manual handling of requests and responses | Simplifies HTTP requests with reactive programming | Simplifies REST client creation with annotations |
| **Example** | RestTemplate restTemplate = new RestTemplate();  // tutorials = restTemplate.getForObject(tutorial\_url, List.class);  // if(tutorials != null) {  // studentResponse.setTutorials(tutorials);  // } | // CloseableHttpClient httpClient = HttpClients.createDefault();// direct use  // without configuration class  HttpGet getRequest = **new** HttpGet(tutorial\_url);  CloseableHttpResponse response = httpClient.execute(getRequest);  **if** (response != **null**) {  String jsonStr = EntityUtils.*toString*(response.getEntity());  ObjectMapper mapper = **new** ObjectMapper();  tutorials = mapper.readValue(jsonStr, **new** TypeReference<List<Tutorial>>() {  });  studentResponse.setTutorials(tutorials);  } | // WebClient.Builder webClientBuilder = WebClient.builder();// direct usage  Mono<List<Tutorial>> tutorialMono = webClientBuilder.build().get().uri(tutorial\_url).retrieve()  .bodyToFlux(Tutorial.**class**).collectList();  tutorials = tutorialMono.block(); | // WebClient.Builder webClientBuilder = WebClient.builder();// direct usage  Mono<List<Tutorial>> tutorialMono = webClientBuilder.build().get().uri(tutorial\_url).retrieve()  .bodyToFlux(Tutorial.**class**).collectList();  tutorials = tutorialMono.block();  //useage  tutorials = tutorialFeignClient.getTutorialsByStudentId(id); |

### Summary:

* **HttpClient**: Suitable for simple and advanced use cases but can be resource-intensive due to its blocking nature.
* **WebClient**: Ideal for modern, non-blocking, reactive applications that need to handle multiple requests efficiently.
* **FeignClient**: Best for simplifying REST client creation within a Spring Cloud environment, though it can be resource-intensive with many requests.