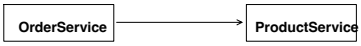
# RestTemplate

Tuesday, 13 May 2025 3:20PM

**In this topic today, will cover:**

**1st : Set up of 2 microservice i.e. "OrderService" and "ProductService" running on different port numbers.**

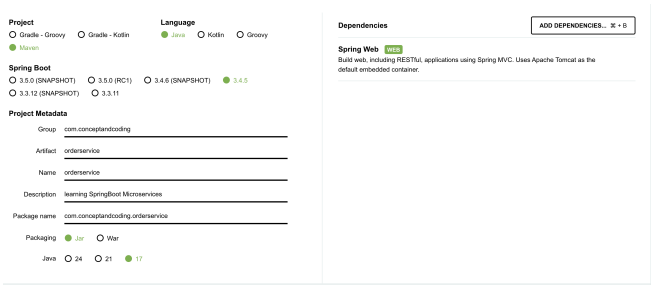

**2nd : How two microservices can communicate with each other**



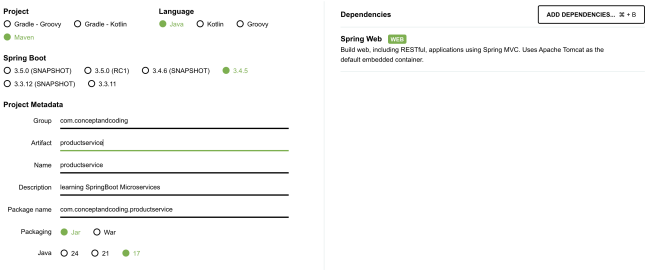

**Let's start:**

## 1st : Set up of 2 microservice

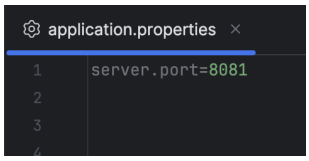
### OrderService Go to Spring Initializer (start.spring.io)




### Similarly, set up ProductService




### OrderService




```
:: Spring Boot ::                (v3.4.5)

2025-05-14T13:11:25.863+05:30  INFO 11440 --- [          main] c.c.o.OrderserviceApplication            : Starting OrderserviceApplication using Java 17.0.12 with PID 11440
2025-05-14T13:11:25.865+05:30  INFO 11440 --- [          main] c.c.o.OrderserviceApplication            : No active profile set, falling back to 1 default profile: "default"
2025-05-14T13:11:26.183+05:30  INFO 11440 --- [          main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8081 (http)
2025-05-14T13:11:26.188+05:30  INFO 11440 --- [          main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2025-05-14T13:11:26.188+05:30  INFO 11440 --- [          main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.40]
2025-05-14T13:11:26.203+05:30  INFO 11440 --- [          main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2025-05-14T13:11:26.204+05:30  INFO 11440 --- [          main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 320 ms
2025-05-14T13:11:26.331+05:30  INFO 11440 --- [          main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8081 (http) with context path '/'
2025-05-14T13:11:26.335+05:30  INFO 11440 --- [          main] c.c.o.OrderserviceApplication            : Started OrderserviceApplication in 0.618 seconds (process running
```

**ProductServic**

application.properties

```
server.port=8082
```

```
2025-05-14T13:11:28.522+05:30  INFO 11442 --- [           main] c.c.p.ProductserviceApplication          : Starting ProductserviceApplication using Java 17.0.12 with
2025-05-14T13:11:28.523+05:30  INFO 11442 --- [           main] c.c.p.ProductserviceApplication          : No active profile set, falling back to 1 default profile: "
2025-05-14T13:11:28.816+05:30  INFO 11442 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8082 (http)
2025-05-14T13:11:28.821+05:30  INFO 11442 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2025-05-14T13:11:28.821+05:30  INFO 11442 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.40]
2025-05-14T13:11:28.839+05:30  INFO 11442 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2025-05-14T13:11:28.840+05:30  INFO 11442 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 297
2025-05-14T13:11:28.964+05:30  INFO 11442 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8082 (http) with context path '/'
2025-05-14T13:11:28.967+05:30  INFO 11442 --- [           main] c.c.p.ProductserviceApplication          : Started ProductserviceApplication in 0.595 seconds (process
```

**2nd : Communication between 2 services**

- **Synchronous**
- **Asynchronous**

**In this part, will focus on Synchronous communication**

**Synchronous Communication :**

- **Client wait for the response from the Server before continuing.**
- **Blocking in nature, means thread waits till response is not received.**

**Synchronous Communication Types in SpringBoot**

- **RestTemplate**
- **RestClient**
- **FeignClient**

**Sample HTTP GET Request call:**

*HTTP method* — *Path(URI)* — *Protocol version*

```
GET /orders/1 HTTP/1.1
Host: localhost:8081
User-Agent: curl/8.7.1
Accept: application/json
```

*Target host*

*Which client/tool is used to make the request*

*Format in which client want response*

**Sample HTTP POST Request call:**

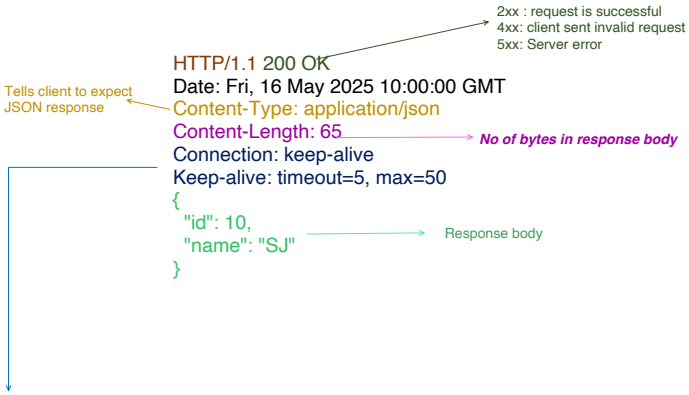
```
POST /products HTTP/1.1
Host: localhost:8081
User-Agent: curl/8.7.1
Accept: application/json
Content-Type: application/json
Content-Length: 65
{
  "name": "Ice-Cream",
  "price": 200
 }
```

*Tell server that, request body is JSON*

*Tells the number of bytes in the request body*

Actual data, a new product object

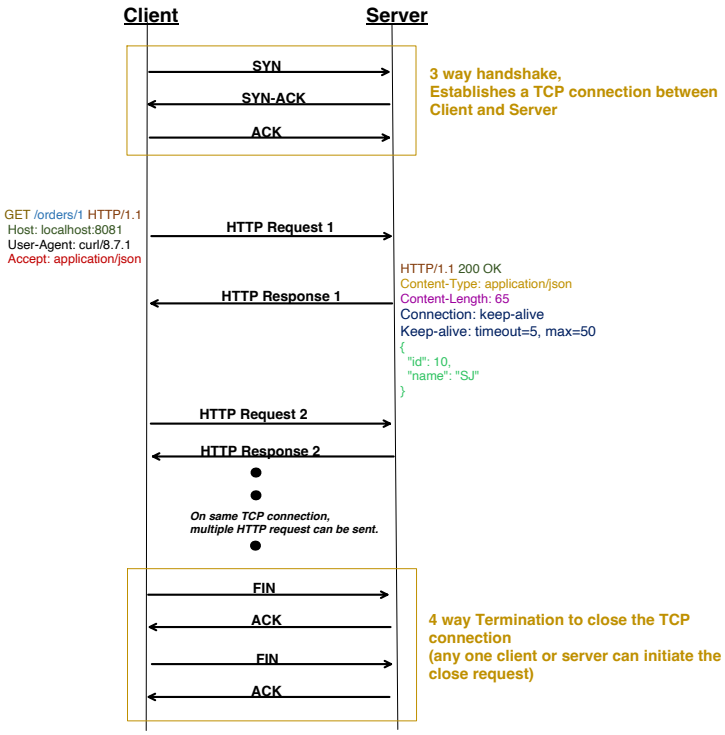
**Sample HTTP GET Response call, with keep alive:**



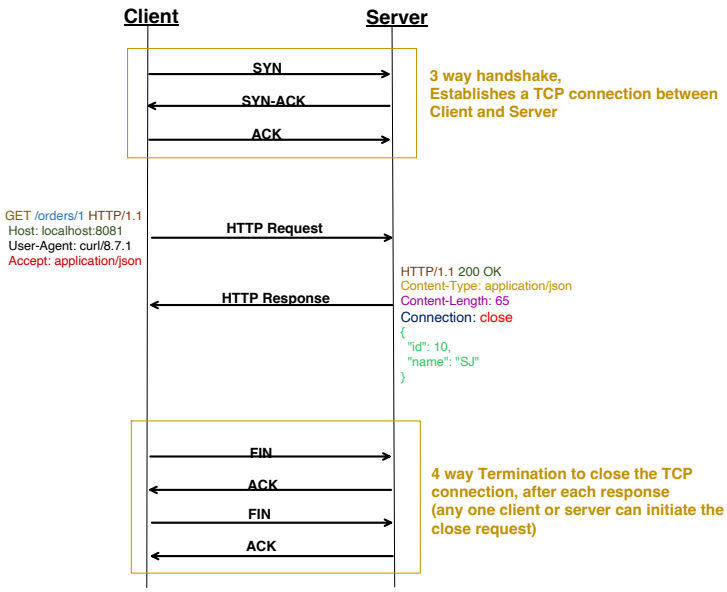

- By default, connection is set to keep-alive in HTTP/1.1
- In HTTP/1.0 by-default connection is set to close.
- Keep-alive:
  - timeout=5, tells close the TCP connection if its idle for 5 seconds
  - Max=50, tells the maximum number of requests can be send over same TCP connection.

- When Connection: close is set, it tells after every response from the server, TCP connection is closed, its not reused.

Flow, when keep-alive is set:



Flow, when connection: close is set:




**Let's first see, without using above SpringBoot communication types, what it takes to invoke the REST endpoint just using plain JAVA.**

### OrderService

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @GetMapping("/{id}")
    public ResponseEntity<String> getOrder(@PathVariable String id) {

        HttpURLConnection httpURLConnection = null;
        try {
            String url = "http://localhost:8082/products/" + id;

            URL obj = new URL(url);
            httpURLConnection = (HttpURLConnection) obj.openConnection();

            // Setting http request method and header
            httpURLConnection.setRequestMethod("GET");
            httpURLConnection.setRequestProperty("Accept", "application/json");

            //max time to establish TCP connection, timeout in millisecond
            httpURLConnection.setConnectTimeout(100);
            //max time to wait for server response after connection is established, timeout in millisecond
            httpURLConnection.setReadTimeout(500);

            // Opens the TCP connection trigger the http request and Read response
            BufferedReader in = new BufferedReader(new InputStreamReader(httpURLConnection.getInputStream()));
            StringBuilder response = new StringBuilder();
            String responseLine;
            while ((responseLine = in.readLine()) != null) {
                response.append(responseLine);
            }
            in.close();
            System.out.println("Response: " + response.toString());

        } catch (Exception e) {
            //exception handling here
        } finally {
            if (httpURLConnection != null) {
                httpURLConnection.disconnect();
            }
        }
        return ResponseEntity.ok( body: "order call successful");
    }
}
```

*Creates an Object of HttpURLConnection, consider it like an envelop or request, in which we specify all the details like URL, Request Method, timeouts etc.*

*Here it opens up a TCP connection and send the HTTP request, also reads the response.*

*TCP connection is created, when we make the first Input/Output request on HttpURLConnection such as:*

- *getInputStream()*
- *getResponseCode()*
- *connect() etc....*

*HttpClient object is a wrapper around TCP connection, so before creating new HttpClient object, if first checks with "KeepAliveCache" class, if there is already an object present, if not it creates one object and also puts into the cache.*

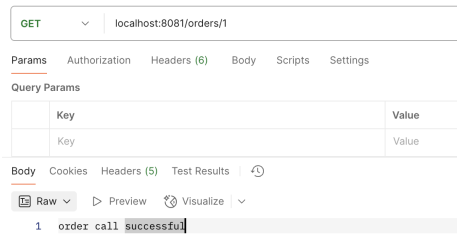
*key -> host:port*
*value -> httpClient object*

*If Response is fully read properly, the TCP Connection i.e. HttpClient is returned back to KeepAlive cache else TCP connection get closed.*

```
public void disconnect() {
    if (input Stream Not Fully Read) {
        httpClient.closeServer(); // Close TCP socket
    } else {
        httpClient.finished();    // Return to KeepAlive Cache
    }
}
```

### ProductService

```java
@RestController
@RequestMapping("/products")
public class ProductController {

    @GetMapping("/{id}")
    public String getProduct(@PathVariable String id) {
        return "Product fetched with id: " + id;
    }
}
```



```
2025-05-15T17:22:49.440+05:30  INFO 14790 --- [           main] c.c.o.OrderserviceApplication            : Starting OrderserviceApplication using
2025-05-15T17:22:49.441+05:30  INFO 14790 --- [           main] c.c.o.OrderserviceApplication            : No active profile set, falling back to
2025-05-15T17:22:49.760+05:30  INFO 14790 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8081 (http
2025-05-15T17:22:49.765+05:30  INFO 14790 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2025-05-15T17:22:49.765+05:30  INFO 14790 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat
2025-05-15T17:22:49.780+05:30  INFO 14790 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplica
2025-05-15T17:22:49.781+05:30  INFO 14790 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initializat
2025-05-15T17:22:49.902+05:30  INFO 14790 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8081 (http) with
2025-05-15T17:22:49.906+05:30  INFO 14790 --- [           main] c.c.o.OrderserviceApplication            : Started OrderserviceApplication in 0.61
2025-05-15T17:22:52.341+05:30  INFO 14790 --- [nio-8081-exec-3] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet '
2025-05-15T17:22:52.341+05:30  INFO 14790 --- [nio-8081-exec-3] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet
2025-05-15T17:22:52.341+05:30  INFO 14790 --- [nio-8081-exec-3] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
Response: Product fetched with id: 1
```

**Couple of Disadvantage of above approach is:**

- **Too much Boilerplate code:**
  - **Open connection**
  - **Setting headers**
  - **Reading response**
  - **Closing streams and connections.**
- **Response should be handled manually.**
  - **No automatic mapping to some Objects.**
- **Limited support for Advance features like**
  - **Connection pooling**
  - **Interceptors etc.**

## RestTemplate

- **Abstract low level code like creating HttpURLConnection object etc.**
- **Traditional/Legacy way to call REST APIs in Spring application.**

### OrderService

```java
@Configuration
public class AppConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

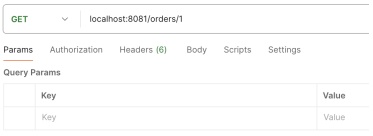
### ProductService

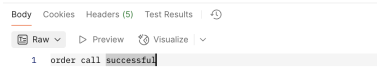
```java
@RestController
@RequestMapping("/products")
public class ProductController {

    @GetMapping("/{id}")
    public String getProduct(@PathVariable String id) {
        return "Product fetched with id: " + id;
    }
}
```

### Or, use below, if we want to set timeouts too

```java
@Configuration
public class AppConfig {

    @Bean
    public RestTemplate restTemplate() {

        SimpleClientHttpRequestFactory factory = new SimpleClientHttpRequestFactory();

        // Set the timeouts in milliseconds
        factory.setConnectTimeout(1000); // 1 sec for connection timeout
        factory.setReadTimeout(5000);    // 5 sec for response timeout

        return new RestTemplate();
    }
}
```

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    RestTemplate restTemplate;

    @GetMapping("/{id}")
    public ResponseEntity<String> getOrder(@PathVariable String id) {

        //invoke product API
        String response = restTemplate.getForObject( url: "http://localhost:8082/products/"+id, String.class);
        System.out.println("Response from Product APi called from order service: " + response);

        return ResponseEntity.ok( body: "order call successful");
    }
}
```





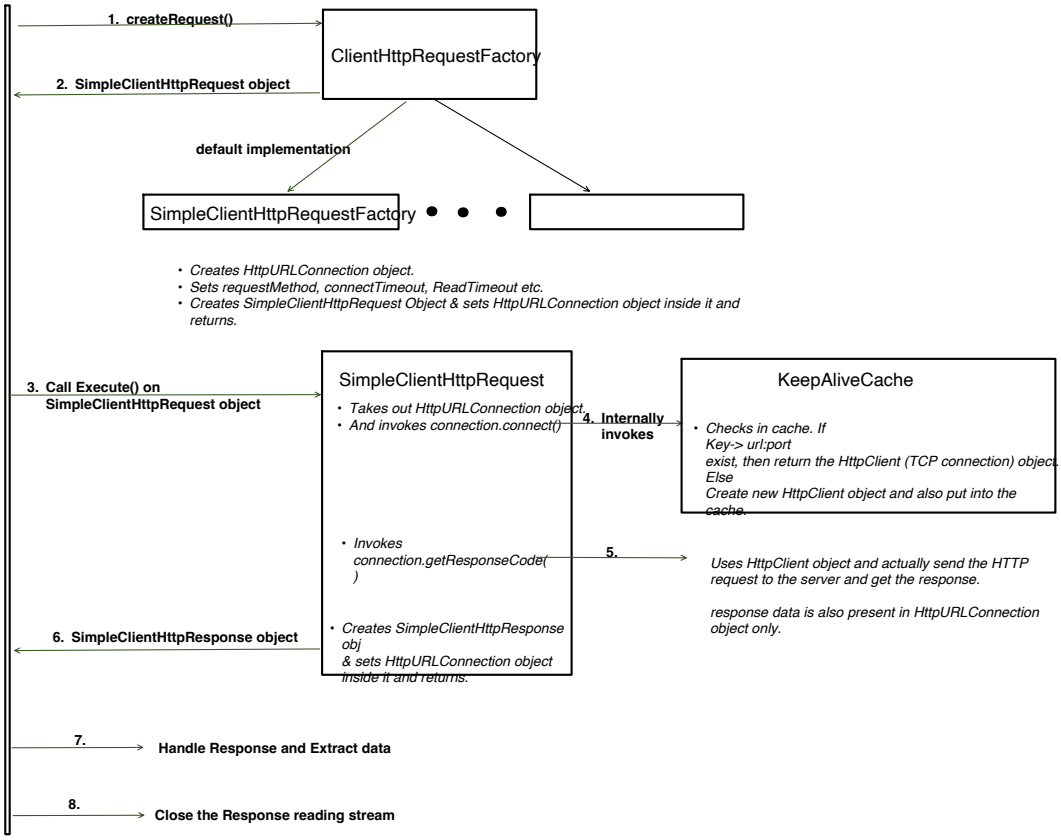

```
2025-05-14T14:43:48.260+05:30  INFO 12346 --- [           main] c.c.o.OrderserviceApplication            : Starting OrderserviceApplication using Ja
2025-05-14T14:43:48.261+05:30  INFO 12346 --- [           main] c.c.o.OrderserviceApplication            : No active profile set, falling back to 1
2025-05-14T14:43:48.584+05:30  INFO 12346 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8081 (http)
2025-05-14T14:43:48.589+05:30  INFO 12346 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2025-05-14T14:43:48.589+05:30  INFO 12346 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/1
2025-05-14T14:43:48.604+05:30  INFO 12346 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicati
2025-05-14T14:43:48.605+05:30  INFO 12346 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initializatio
2025-05-14T14:43:48.729+05:30  INFO 12346 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8081 (http) with c
2025-05-14T14:43:48.734+05:30  INFO 12346 --- [           main] c.c.o.OrderserviceApplication            : Started OrderserviceApplication in 0.62 s
2025-05-14T14:43:51.223+05:30  INFO 12346 --- [nio-8081-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'di
2025-05-14T14:43:51.223+05:30  INFO 12346 --- [nio-8081-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2025-05-14T14:43:51.224+05:30  INFO 12346 --- [nio-8081-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
Response from Product APi called from order service: Product fetched with id: 1
```

**So, what exactly happened, RestTemplate works internally:**

**When below method *"getForObject"* invoked:**

String response = restTemplate.getForObject("http://localhost:8082/products/"+id, String.class);

**RestTemplate**




**Notice one thing that:**

- **RestTemplate do not close the TCP connection explicitly.**
- **Once Response stream is closed, TCP connection (i.e. HttpClient object, is ready to be re-used till is not expired based on idle Timeout or max Connection configuration)**

## Lets see, some of the other methods which are available in RestTemplate

| Method Name | Description |
|---|---|
| GET | |
| getForObject(String url, Class<T> responseType) | Returns the response body as an Object.<br><br>String url = "http://localhost:8080/api/products/1";<br>Product product = restTemplate.getForObject(url, Product.class); |
| getForEntity(String url, Class<T> responseType) | Returns full ResponseEntity with status and header<br><br>String url = "http://localhost:8080/api/products/1";<br>ResponseEntity<Product> response = restTemplate.getForEntity(url, Product.class);<br>HttpStatus status = response.getStatusCode();<br>Product product = response.getBody(); |
| POST | |
| postForObject(String url, Object request, Class<T> responseType) | Sends POST and get just the response body.<br><br>String url = "http://localhost:8080/api/products";<br>Product newProduct = new Product("Ice-cream", 100);<br>Product createdProduct = restTemplate.postForObject(url, newProduct, Product.class); |
| postForEntity(String url, Object request, Class<T> responseType) | Sends POST and get just the full ResponseEntity object.<br><br>String url = "http://localhost:8080/api/products";<br>Product newProduct = new Product("Ice-cream", 100);<br>ResponseEntity<Product> response = restTemplate.postForEntity(url, newProduct, Product.class);<br>Product createdProduct = response.getBody();<br>HttpStatus status = response.getStatusCode(); |
| PUT | |
| put(String url, Object request) | Sends PUT and no response body is expected.<br><br>String url = "http://localhost:8080/api/products/1";<br>Product updatedProduct = new Product("Ice-cream", 150);<br>restTemplate.put(url, updatedProduct); |
| DELETE | |
| delete(String url) | Sends DELETE request and no response body is expected.<br><br>String url = "http://localhost:8080/api/products/1";<br>restTemplate.delete(url); |
| GENERAL PURPOSE | |
| exchange(String url, HttpMethod method, HttpEntity<?> requestEntity, Class<T> responseType) | When we want to customize :<br>• HTTP method (GET, PUT, POST etc.)<br>• HTTP header and body (HttpEntity)<br>• But want Spring automatic Conversion<br><br>String url = "http://localhost:8080/api/products/";<br><br>//customizing header<br>HttpHeaders headers = new HttpHeaders();<br>headers.setContentType(MediaType.APPLICATION_JSON)headers.set("Authorization", "Bearer my-token");<br><br>//preparing http request body<br>Product product = new Product();<br>product.setName("Ice-cream");<br>product.setPrice(100);<br><br>//setting both header and body in the HttpEntity<br>HttpEntity<Product > requestEntity = new HttpEntity<>(product, headers);<br><br>ResponseEntity<Product> response = restTemplate.exchange(<br>url,<br>HttpMethod.POST,<br>requestEntity,<br>Product.class<br>);<br><br>Product product = response.getBody();<br>HttpStatus status = response.getStatusCode(); |
| execute(String url, HttpMethod method, RequestCallback requestCallback, ResponseExtractor<T> responseExtractor) | When we want full control like in plain java we use HttpURLConnection object.<br>Header, body, Request, Response, serialization etc. need to be handled manually.<br><br>RequestCallback interface, gives us full control over the request. We can set header, write body etc.<br><br>@FunctionalInterface<br>public interface RequestCallback {<br>void doWithRequest(ClientHttpRequest request) throws IOException;<br>}<br><br>Similarly, ResponseExtractor, gives us full control over, how the response is read and converted to desired object.<br><br>@FunctionalInterface<br>public interface ResponseExtractor<T> {<br>T extractData(ClientHttpResponse response) throws IOException;<br>}<br><br>RestTemplate restTemplate = new RestTemplate();<br><br>String url = "http://localhost:8080/api/products";<br><br>//setting both header and body<br>RequestCallback requestCallback = request -> {<br><br>request.getHeaders().setContentType(MediaType.APPLICATION_JSON);<br>Product product = new Product("Ice-cream", 100);<br>ObjectMapper mapper = new ObjectMapper();<br>byte[] body = mapper.writeValueAsBytes(product);<br>StreamUtils.copy(body, request.getBody());<br>};<br><br>//parsing the response<br>ResponseExtractor<String> responseExtractor = response -> {<br>return StreamUtils.copyToString(response.getBody(), StandardCharsets.UTF_8);<br>};<br><br>String response = restTemplate.execute(<br>url,<br>HttpMethod.POST,<br>requestCallback,<br>responseExtractor<br>);<br><br>System.out.println("response is: " + response); |

**Limitation of RestTemplate:**

- **In RestTemplate, there are already so many overloaded methods, so its hard to remember and maintain.(Above we have just covered few)**
- **RestTemplate was build before concepts like Retry, circuit breaker etc.. So adding support means more overloaded methods and not user friendly.**
- **RestTemplate is in Maintenance mode - means no new feature, only bug fixes.**

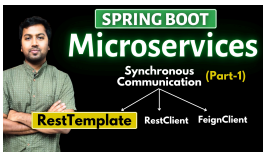
**That's where latest RestClient comes into the picture:**

- **Introduction of Fluent, builder-style API (more readable and user friendly way of configuring and invoking the endpoint)**
- **RestClient supports easy integration with interceptors, filters etc.**

# RestClient

Friday, 23 May 2025 12:54 PM

**Prerequisite to understand RestClient is, to first understand its predecessor RestTemplate**




**Kindly check the Part-1 (RestTemplate), if there is any doubt with that topic.**

**In previous video, we saw the Limitation of RestTemplate:**

- **In RestTemplate, there are already so many overloaded methods, so its hard to remember and maintain.**
- **RestTemplate was build before concepts like Retry, circuit breaker etc.. So adding support means more overloaded methods and not user friendly.**
- **RestTemplate is in Maintenance mode - means no new feature, only bug fixes.**

**Alternative of RestTemplate**

**WebClient**

- **Asynchronous / non blocking in nature.**
- **Client do not wait for the response from the Server before continuing.**
- **Introduced in Spring WebFlux for reactive programming.**

**RestClient**

- **Introduced in Spring Framework 6.0+ and SpringBoot 3.0+**
- **Synchronous/blocking in nature, means client wait for the response from the Server before continuing.**
- **Modern, fluent based API that is more readable and easy to maintain.**

**Let's start with RestClient:**

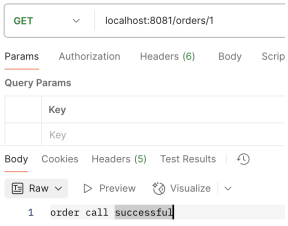
**OrderService** *(running on localhost:8081)* — *OrderService needs to invoke ProductService* → **ProductService** *(running on localhost:8082)*

```java
@Configuration
public class AppConfig {

    @Bean
    public RestClient restClientInstance() {
        return RestClient.create();
    }
}
```

```java
@RestController
@RequestMapping("/products")
public class ProductController {

    @GetMapping("/{id}")
    public String getProduct(@PathVariable String id) {
        return "Product fetched with id: " + id;
    }
}
```

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    RestClient restClient;

    @GetMapping("/{id}")
    public ResponseEntity<String> getOrder(@PathVariable String id) {

        String response = restClient
                .get()
                .uri( uri: "http://localhost:8082/products/" + id)
                .retrieve()
                .body(String.class);

        System.out.println("Response from Product APi called from order service: " + response);
        return ResponseEntity.ok( body: "order call successful");
    }
}
```

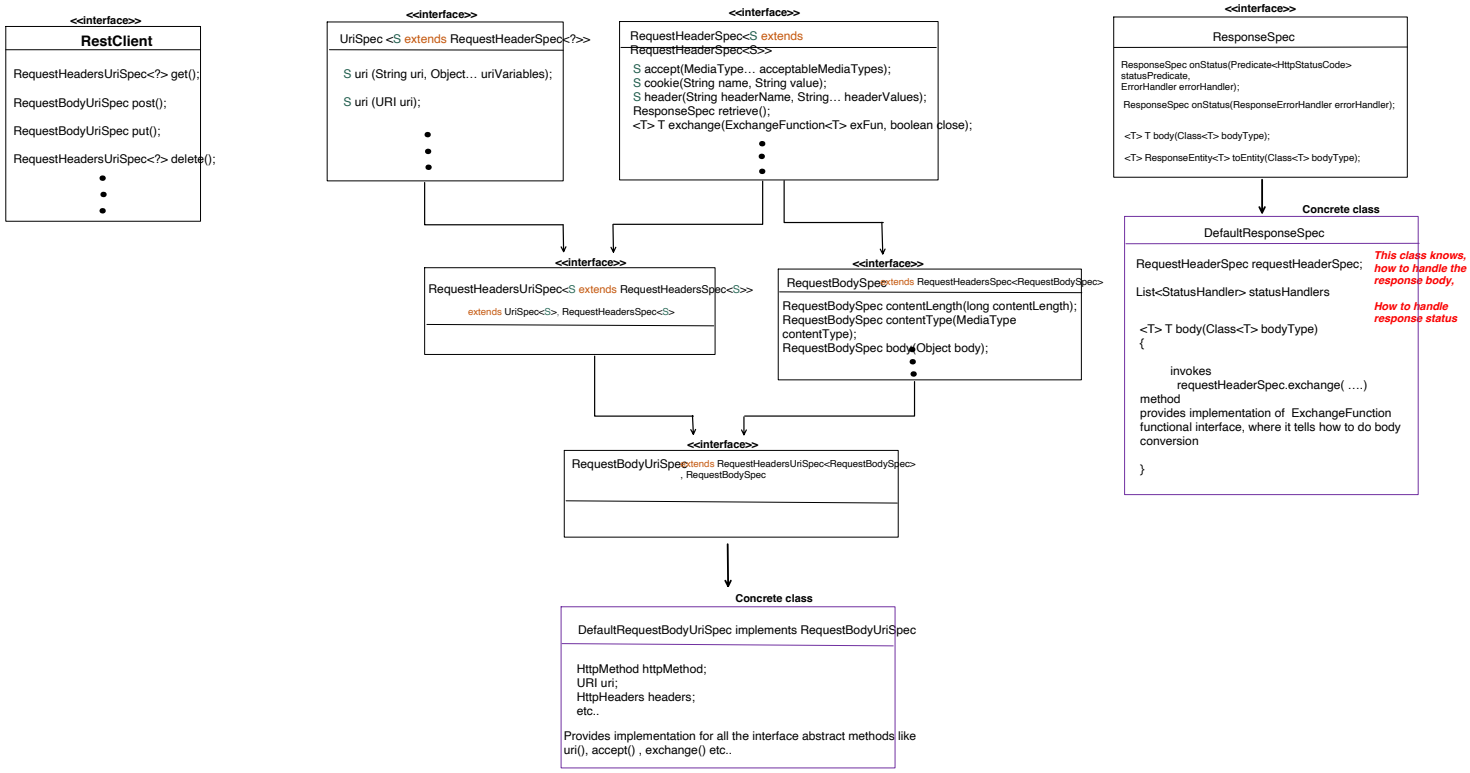
*Same template, we are going to use for all different operation like post, put, delete etc.*
*No different methods for different operations like RestTemplate*



```
2025-05-27T15:14:57.352+05:30  INFO 66070 --- [           main] c.c.o.OrderserviceApplication            : Starting OrderserviceApplication using J
2025-05-27T15:14:57.353+05:30  INFO 66070 --- [           main] c.c.o.OrderserviceApplication            : No active profile set, falling back to 1
2025-05-27T15:14:57.700+05:30  INFO 66070 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8081 (http)
2025-05-27T15:14:57.705+05:30  INFO 66070 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2025-05-27T15:14:57.705+05:30  INFO 66070 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/
2025-05-27T15:14:57.722+05:30  INFO 66070 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicat
2025-05-27T15:14:57.722+05:30  INFO 66070 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initializati
2025-05-27T15:14:57.986+05:30  INFO 66070 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8081 (http) with
2025-05-27T15:14:57.910+05:30  INFO 66070 --- [           main] c.c.o.OrderserviceApplication            : Started OrderserviceApplication in 0.713
2025-05-27T15:15:04.840+05:30  INFO 66070 --- [nio-8081-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'd
2025-05-27T15:15:04.840+05:30  INFO 66070 --- [nio-8081-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2025-05-27T15:15:04.841+05:30  INFO 66070 --- [nio-8081-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
Response from Product APi called from order service: Product fetched with id: 1
```

**So, first important thing to understand is, how this Fluent API works?**

- **Fluent API means chaining of method calls.**
- **Each method call returns an object (of next step) that exposes next set of operations (methods).**



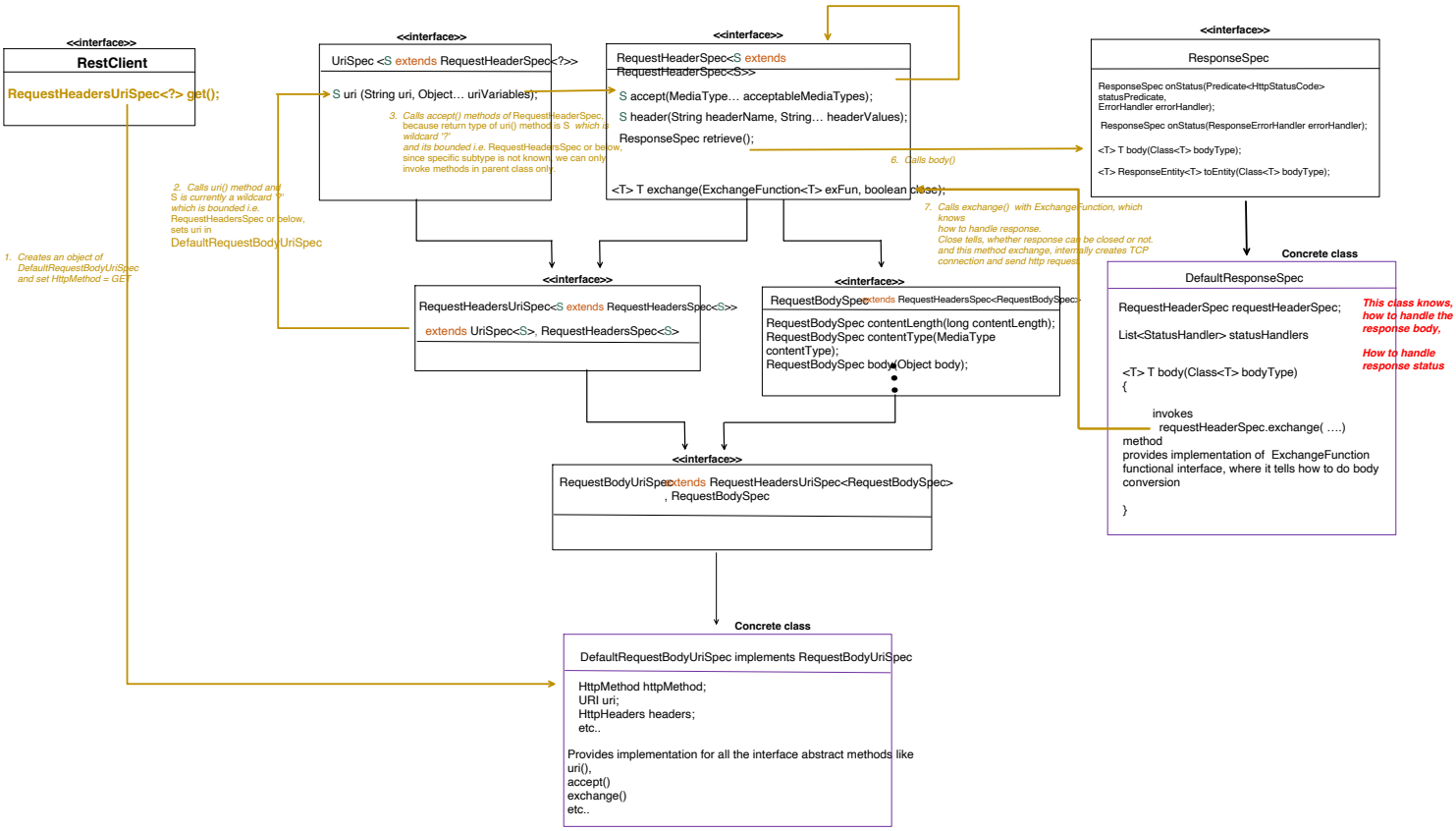

**I am assuming that, we all know about Generics Class and Wildcards in Java**


**Kindly check this video in JAVA playlist, if there is any doubt with this topic**

**Now, lets see how Fluent API works for GET, POST, DELETE operation.**

## GET

```
RestClient restClient = RestClient.create();

String response = restClient
        .get()
        .uri( uri: "http://localhost:8082/products/" + id)
        .accept(MediaType.APPLICATION_JSON)
        .header( headerName: "X-Custom-Header",  ...headerValues: "xyz")
        .retrieve()
        .body(String.class);
```

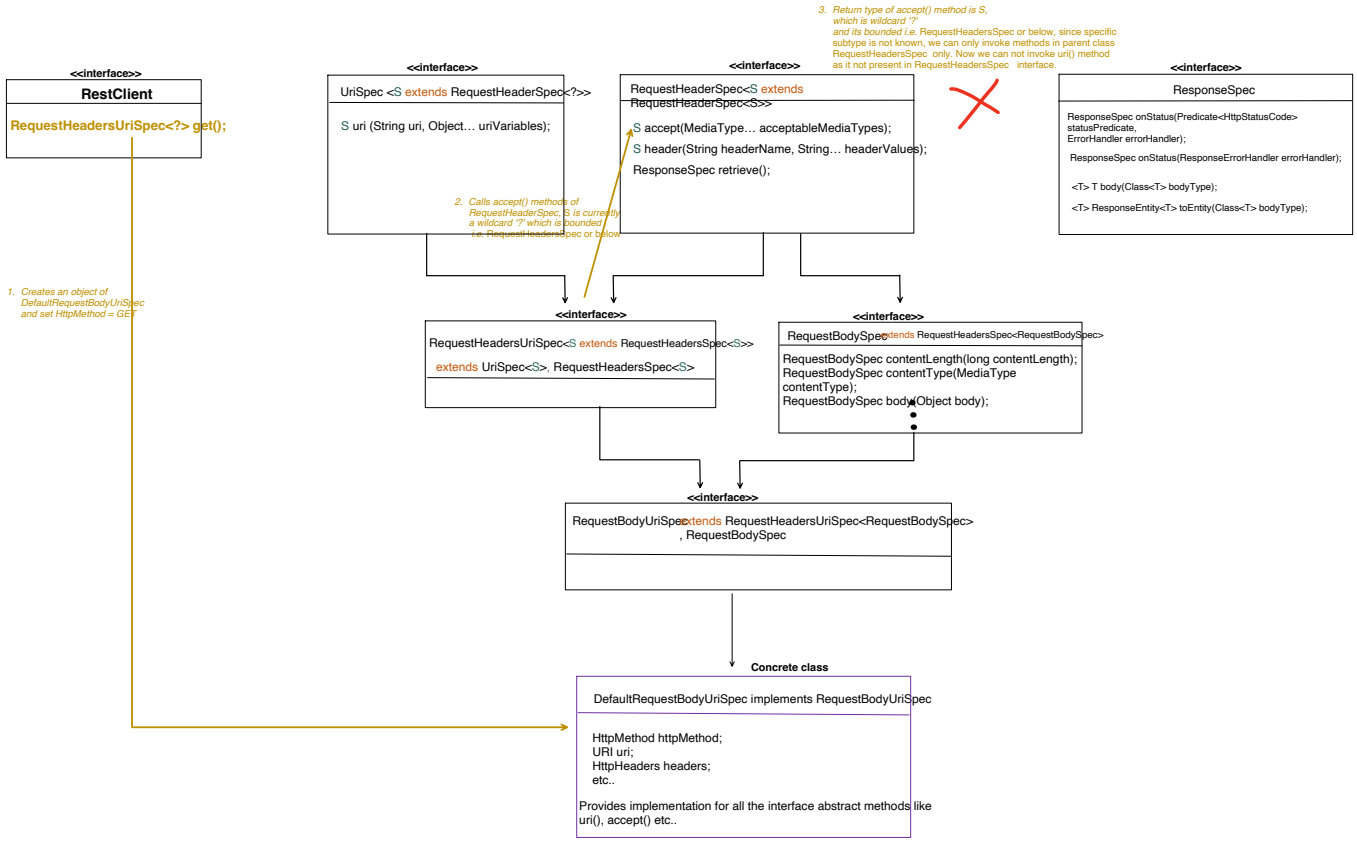



**Let see, how flow can be broken**

## GET

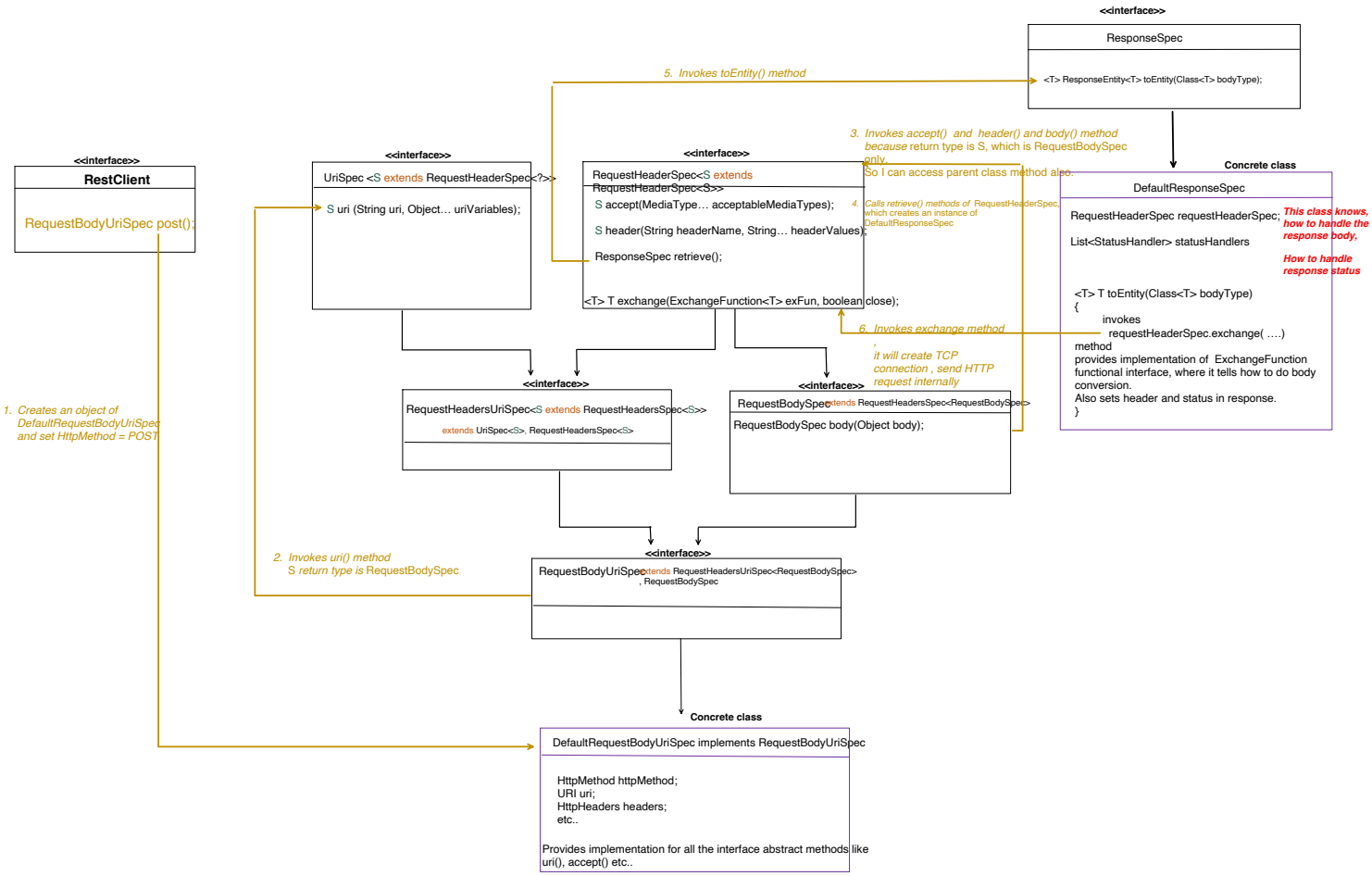
```
RestClient restClient = RestClient.create();

String response = restClient
        .get()
        .accept(MediaType.APPLICATION_JSON)
                .uri
```

***Instead of uri method first, I invoked the accept method first and now I can not set the URI itself. Why, does sequence of call matters?***



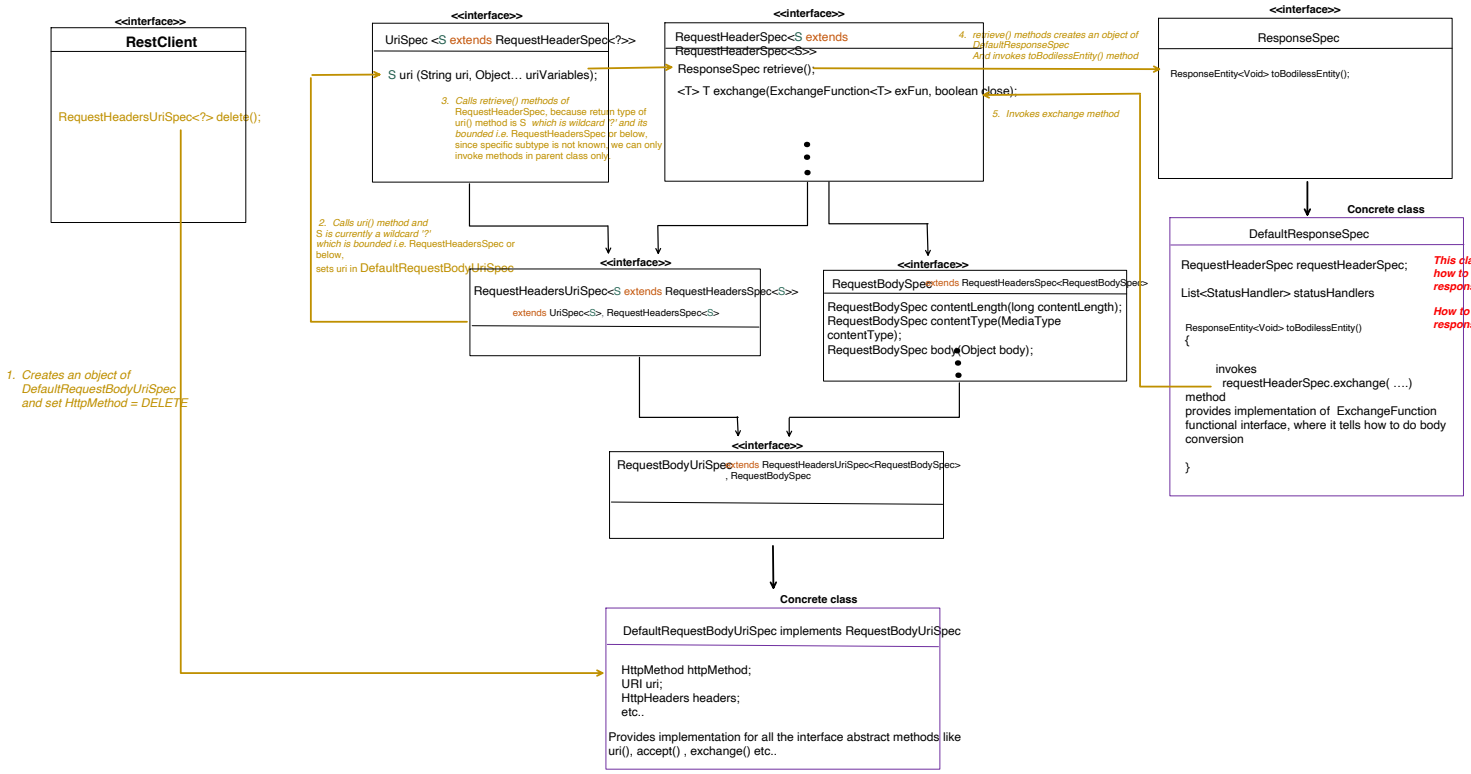

**POST**

```
RestClient restClient = RestClient.create();

ResponseEntity<ProductEntity> response = restClient
        .post()
        .uri( uri: "http://localhost:8082/products/create")
        .accept(MediaType.APPLICATION_JSON)
        .header( headerName: "Content-Type",  ...headerValues: "application/json")
        .body(new ProductEntity( name: "Ice-cream")) //some new object which need to be created
        .retrieve()
        .toEntity(ProductEntity.class);

ProductEntity responseBody = response.getBody();
```



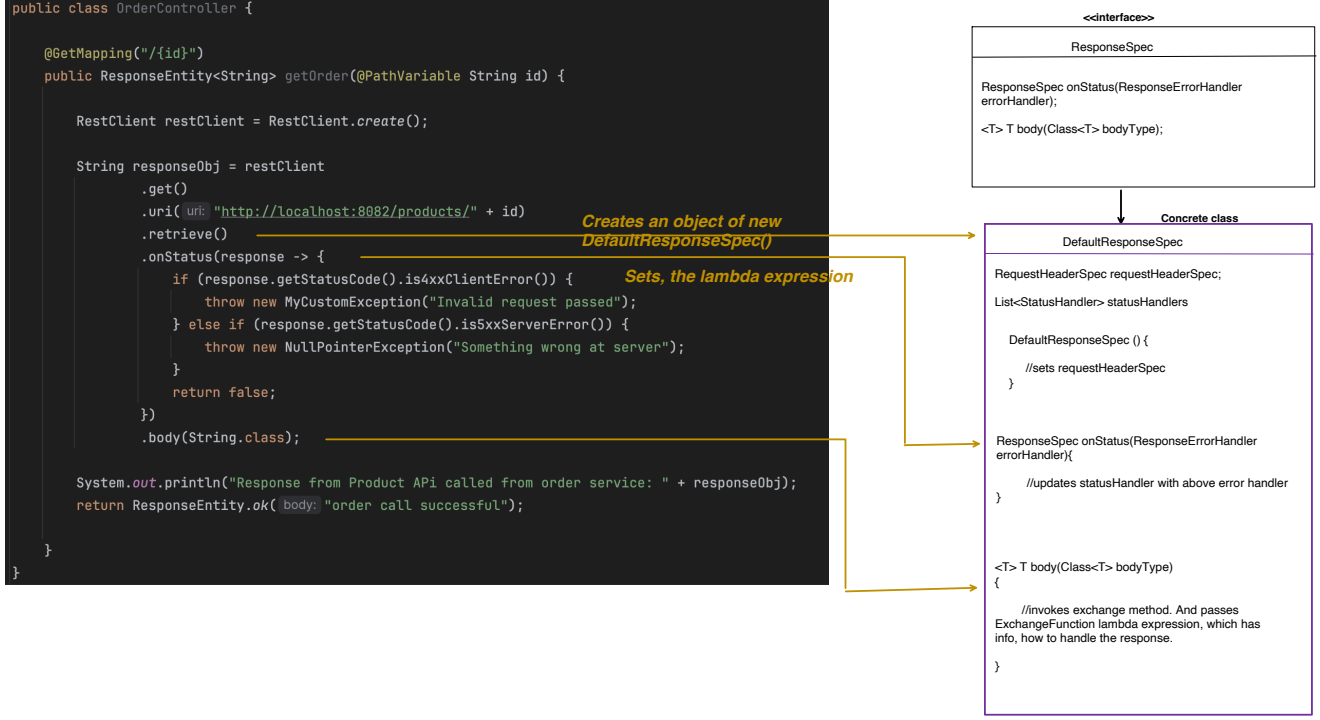

## DELETE

```
RestClient restClient = RestClient.create();

ResponseEntity<Void> response = restClient
        .delete()
        .uri( uri: "http://localhost:8082/products/"+id)
        .retrieve()
        .toBodilessEntity();

HttpStatusCode deletionStatus = response.getStatusCode();
```




## Exception Handling

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @GetMapping("/{id}")
    public ResponseEntity<String> getOrder(@PathVariable String id) {

        RestClient restClient = RestClient.create();

        String responseObj = restClient
                .get()
                .uri( uri: "http://localhost:8082/products/" + id)
                .retrieve()
                .onStatus(response -> {
                    if (response.getStatusCode().is4xxClientError()) {
                        throw new MyCustomException("Invalid request passed");
                    } else if (response.getStatusCode().is5xxServerError()) {
                        throw new NullPointerException("Something wrong at server");
                    }
                    return false;
                })
                .body(String.class);

        System.out.println("Response from Product APi called from order service: " + responseObj);
        return ResponseEntity.ok( body: "order call successful");

    }
}
```



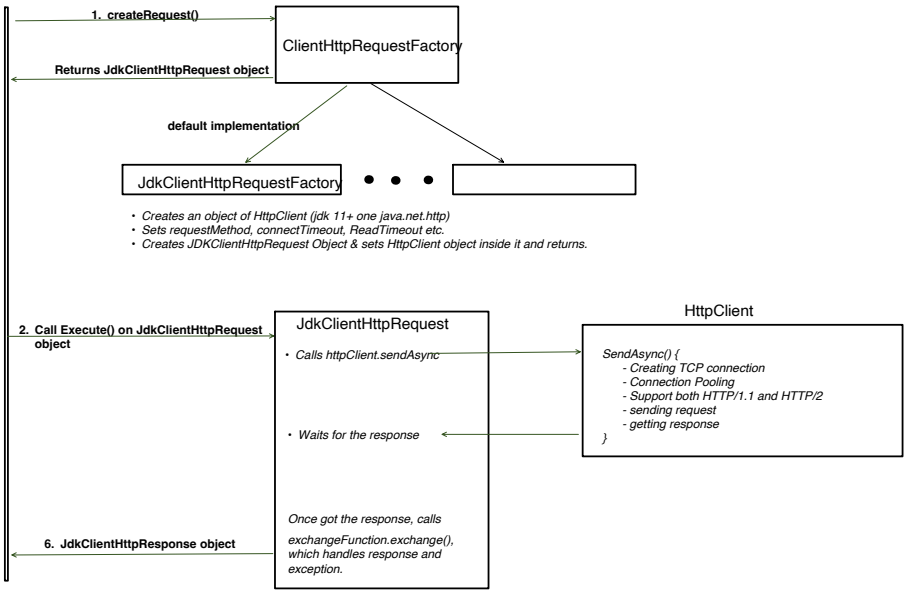

## If we want full control over Response building and Exception handling

**Then we can directly call exchange method, instead of relying of ResponseSpec to do that.**

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @GetMapping("/{id}")
    public ResponseEntity<String> getOrder(@PathVariable String id) {

        RestClient restClient = RestClient.create();

        String responseObj = restClient
                .get()
                .uri( uri: "http://localhost:8082/products/" + id)
                .exchange((request, response) -> {
                    if (response.getStatusCode().is4xxClientError()) {
                        throw new MyCustomException("Invalid request passed");
                    } else if (response.getStatusCode().is5xxServerError()) {
                        throw new InternalError("Something wrong at server");
                    } else {

                        //response mapping logic, if there is no error
                        return StreamUtils.copyToString(response.getBody(), StandardCharsets.UTF_8);
                    }
                });

        System.out.println("Response from Product APi called from order service: " + responseObj);
        return ResponseEntity.ok( body: "order call successful");

    }
}
```

**exchange() method:**

```java
private <T> T exchangeInternal(ExchangeFunction<T> exchangeFunction, boolean close) {
    Assert.notNull(exchangeFunction,  message: "ExchangeFunction must not be null");

    ClientHttpResponse clientResponse = null;
    Observation observation = null;
    Observation.Scope observationScope = null;
    URI uri = null;
    try {
        uri = initUri();
        String serializedCookies = serializeCookies();
        if (serializedCookies != null) {
            getHeaders().set(HttpHeaders.COOKIE, serializedCookies);
        }
        HttpHeaders headers = initHeaders();

        ClientHttpRequest clientRequest = createRequest(uri);  ——— (1)
        if (headers != null) {
            clientRequest.getHeaders().addAll(headers);
        }
        Map<String, Object> attributes = getAttributes();
        clientRequest.getAttributes().putAll(attributes);
        ClientRequestObservationContext observationContext = new ClientRequestObservationContext(clientRequest);
        observationContext.setUriTemplate((String) attributes.get(URI_TEMPLATE_ATTRIBUTE));
        observation = ClientHttpObservationDocumentation.HTTP_CLIENT_EXCHANGES.observation(observationConvention,
                DEFAULT_OBSERVATION_CONVENTION, () -> observationContext, observationRegistry).start();
        observationScope = observation.openScope();
        if (this.body != null) {
            this.body.writeTo(clientRequest);
        }
        if (this.httpRequestConsumer != null) {
            this.httpRequestConsumer.accept(clientRequest);
        }
        clientResponse = clientRequest.execute();  ——— (2)
        observationContext.setResponse(clientResponse);
        ConvertibleClientHttpResponse convertibleWrapper = new DefaultConvertibleClientHttpResponse(clientResponse);
        return exchangeFunction.exchange(clientRequest, convertibleWrapper);  ——— (3)
    }
```




## Adding Interceptors

### OrderService

```
public class MyCustomRequestInterceptor implements ClientHttpRequestInterceptor {

    @Override
    public ClientHttpResponse intercept(HttpRequest request, byte[] body,
                                        ClientHttpRequestExecution execution) throws IOException {

        request.getHeaders().add( headerName: "X-Custom-Header",  headerValue: "myvalue");
        return execution.execute(request, body);
    }
}
```

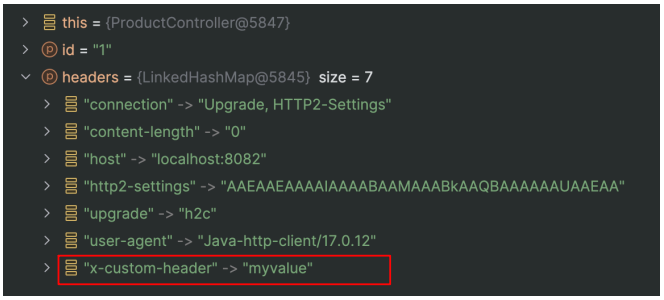
```
@Configuration
public class AppConfig {

    @Bean
    public RestClient restClientInstance(ClientHttpRequestInterceptor myCustomInterceptor) {
        return RestClient.builder()
                .requestInterceptor(myCustomInterceptor)
                .build();
    }

    @Bean
    public ClientHttpRequestInterceptor customRequestInterceptor() {
        return new MyCustomRequestInterceptor();
    }
}
```

```
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    RestClient restClient;

    @GetMapping("/{id}")
    public ResponseEntity<String> getOrder(@PathVariable String id) {

        ResponseEntity<String> responseEntityObj = restClient
                .get()
                .uri( uri: "http://localhost:8082/products/" + id)
                .retrieve()
                .toEntity(String.class);

        System.out.println("Response from Product APi called from order service: " + responseEntityObj.getBody());
        return ResponseEntity.ok( body: "order call successful");
    }
}
```

### ProductService

```
@RestController
@RequestMapping("/products")
public class ProductController {

    @GetMapping("/{id}")
    public String getProduct(@PathVariable String id) {
        return "Product fetched with id: " + id;
    }
}
```

**When product service got invoked, able to see the header which we added using interceptor**




**Interceptor invokes at step2**

```java
private <T> T exchangeInternal(ExchangeFunction<T> exchangeFunction, boolean close) {
    Assert.notNull(exchangeFunction, message: "ExchangeFunction must not be null");

    ClientHttpResponse clientResponse = null;
    Observation observation = null;
    Observation.Scope observationScope = null;
    URI uri = null;
    try {
        uri = initUri();
        String serializedCookies = serializeCookies();
        if (serializedCookies != null) {
            getHeaders().set(HttpHeaders.COOKIE, serializedCookies);
        }
        HttpHeaders headers = initHeaders();

        ClientHttpRequest clientRequest = createRequest(uri);
        if (headers != null) {
            clientRequest.getHeaders().addAll(headers);
        }
        Map<String, Object> attributes = getAttributes();
        clientRequest.getAttributes().putAll(attributes);
        ClientRequestObservationContext observationContext = new ClientRequestObservationContext(clientRequest);
        observationContext.setUriTemplate((String) attributes.get(URI_TEMPLATE_ATTRIBUTE));
        observation = ClientHttpObservationDocumentation.HTTP_CLIENT_EXCHANGES.observation(observationConvention,
                DEFAULT_OBSERVATION_CONVENTION, () -> observationContext, observationRegistry).start();
        observationScope = observation.openScope();
        if (this.body != null) {
            this.body.writeTo(clientRequest);
        }
        if (this.httpRequestConsumer != null) {
            this.httpRequestConsumer.accept(clientRequest);
        }
        clientResponse = clientRequest.execute();
        observationContext.setResponse(clientResponse);
        ConvertibleClientHttpResponse convertibleWrapper = new DefaultConvertibleClientHttpResponse(clientResponse);
        return exchangeFunction.exchange(clientRequest, convertibleWrapper);
    }
```



Interceptor, will get invoked when this method is trying to execute, our execute call is wrapped in InterceptingClientHttpRequest

# FeignClient

- Feign is a **Declarative HTTP client** developed by Netflix.
- **Declarative** means "we tell What to do, not How to do".

**In SpringBoot , Feign capability is available via Spring Cloud OpenFeign library.**

- **Spring Cloud** provides a set of tool and libraries, which helps to build distributed microservices.
- As it provides seamless integration with :
  - Service Discovery
  - Client side Load Balancing
  - Circuit Breaker and Resilience
  - Api Gateway
  - Distributed Tracing
  - Centralized Configuration etc....

## Pom.xml

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

**In future, we might use more Spring Cloud libraries (for Load balancer, for Service Discovery etc.) and all those Spring Cloud libraries should have compatible version, therefore we use below dependency management, so that we don't have to mange it manually.**

**That's why we are not specifying the version with above "**spring-cloud-starter-openfeign"
dependency, it will be taken care by our dependency management.

```
<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-dependencies</artifactId>
         <version>2023.0.1</version> <!-- Use latest compatible version-->
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>
```

**OrderService** *(running on localhost:8081)* → *OrderService needs to invoke ProductService* → **ProductServic** *(running on localhost:8082)*

```java
@FeignClient(name = "product-service",
        url = "${feign.client.product-service.url}")
public interface ProductClient {

    @GetMapping("/products/{id}")
    String getProductById(@PathVariable("id") String id);

}
```

*We are not writing any logic, just told what to call.*

```java
@RestController
@RequestMapping("/products")
public class ProductController {

    @GetMapping("/{id}")
    public String getProduct(@PathVariable String id) {
        return "Product fetched with id: " + id;
    }
}
```

## application.properties

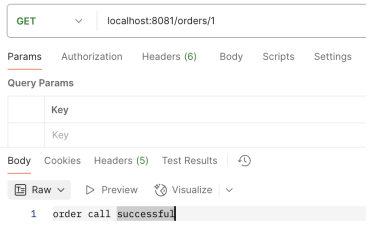
```
server.port=8081

#Base URL for Product Service
feign.client.product-service.url=http://localhost:8082
```

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    ProductClient productClient;

    @GetMapping("/{id}")
    public ResponseEntity<String> getOrder(@PathVariable String id) {

        String responseFromProductAPI = productClient.getProductById(id);
        System.out.println("Response from Product api call is: " + responseFromProductAPI);

        return ResponseEntity.ok( body: "order call successful");
    }
}
```

```java
@SpringBootApplication
@EnableFeignClients
public class OrderserviceApplication {

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

*It enable Feign support and tells SpringBoot to scan for interfaces annotated with @FeignClient*

**Start the application and invoke the Order Endpoint**




```
2025-06-06T21:11:17.438+05:30  INFO 18774 --- [           main] c.c.o.OrderserviceApplication            : Starting OrderserviceApplication using
2025-06-06T21:11:17.439+05:30  INFO 18774 --- [           main] c.c.o.OrderserviceApplication            : No active profile set, falling back to
2025-06-06T21:11:17.705+05:30  INFO 18774 --- [           main] o.s.cloud.context.scope.GenericScope     : BeanFactory id=f55ac312-243e-3904-baf4-
2025-06-06T21:11:17.810+05:30  INFO 18774 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8081 (http
2025-06-06T21:11:17.814+05:30  INFO 18774 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2025-06-06T21:11:17.814+05:30  INFO 18774 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat
2025-06-06T21:11:17.836+05:30  INFO 18774 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplica
2025-06-06T21:11:17.837+05:30  INFO 18774 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initializat
2025-06-06T21:11:17.999+05:30  INFO 18774 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8081 (http) with
2025-06-06T21:11:18.005+05:30  INFO 18774 --- [           main] c.c.o.OrderserviceApplication            : Started OrderserviceApplication in 0.71
2025-06-06T21:11:28.409+05:30  INFO 18774 --- [nio-8081-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet '
2025-06-06T21:11:28.409+05:30  INFO 18774 --- [nio-8081-exec-2] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet
2025-06-06T21:11:28.410+05:30  INFO 18774 --- [nio-8081-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
Response from Product api call is: fetch the product details with id:1
```

**So, first important thing to understand is, how this Declarative HTTP Calls works?**

```java
@FeignClient(name = "product-service",
        url = "${feign.client.product-service.url}")
public interface ProductClient {

    @GetMapping("/products/{id}")
    String getProductById(@PathVariable("id") String id);

}
```

*We have not provide any implementation, but how come we are able to Autowired it, without any exception?*

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    ProductClient productClient;

    @GetMapping("/{id}")
    public ResponseEntity<String> getOrder(@PathVariable String id) {

        String responseFromProductAPI = productClient.getProductById(id);
        System.out.println("Response from Product api call is: " + responseFromProductAPI);

        return ResponseEntity.ok( body: "order call successful");
    }
}
```

Application Startup → *Scan for @FeignClient interface* → *For each interface, invokes*

ReflectiveFeign.newInstance()

1. For each method in an interface → *Create an Object of MethodHandler and holds it in Map<Method, MethodHandler>*
2. *Creates an Object of InvocationHandler and populate the variable: Map<Method, MethodHandler> with info which we get in #1.*
3. *Uses **Dynamic Proxy** to create runtime implementation of our interface.*

***There is no such class, its just my understanding and demo of how Dynamic proxy would have implemented our interface ProductClient.***

```java
public class FeignProductClientProxy implements ProductClient {
    private final InvocationHandler handler;

    public FeignProductClientProxy(InvocationHandler handler) {
        this.handler = handler;
    }

    @Override
    public String getProductById(String id) throws Throwable {
        Method method = ProductClient.class.getMethod( name: "getProductById", String.class);
        return (String) handler.invoke( proxy: this, method, new Object[]{id});
    }
}
```

***MethodHandler** object holds complete info that is parsed from a method and its annotations, which is needed to execute the HTTP request.*

*Below are some of the fields which **MethodHandler** object holds:*

- **Target URL**: info regarding Base URL + relative path
  - @FeignClient(name="product-service", *url="http://localhost:8082"*)
  - @GetMapping(*"/products/{id}"*)
- **HTTP Method**: GET, PUT, POST etc.
  - Derived from *@GetMapping*, *@PostMapping* annotations etc.
- **Header Info**
  - Derived from *@RequestHeader* or *interceptors* etc.
- **HTTP Client**: actual client to make a HTTP call.
  - We can configure to either choose:
    - *HttpURLConnection* (default)
    - *OkHttp*
    - *ApacheHttpClient*
- **Encoder**: Converts request body from Java object to HTTP format like JSON
  - Configurable, we can provide our own implementation
- **Decoder**: Converts response to Java object
  - Configurable, we can provide our own implementation
- **ErrorDecoder**: Tells what to do, when exception is thrown
  - Configurable, we can provide our own implementation
- **Logger**: logs the request and response info.
  - Configurable, we can provide our own implementation
- **Retryer**: handle retries when failure happens.
  - Configurable, we can provide our own implementation

*Invocation Handler, act as a bridge between :*

- *Proxy Implementation of the interface and*
- *The MethodHandler, which has logic to build HTTP request and invoke the HTTP call.*

```java
@Override
public Object invoke(Object proxy, Method method, Object[] args)  {
    // Step 1: Look up the MethodHandler for this method
    MethodHandler methodHandler = map.get(method);

    // Step 2: Use MethodHandler to perform HTTP logic
    return methodHandler.invoke(args);
}
```

***MethodHandler** also has method invoke(), which knows how to create HTTP request object from above info and make a HTTP call.*

```java
@Override
public Object invoke(Object[] argv) throws Throwable {
  RequestTemplate template = buildTemplateFromArgs.create(argv);
  Options options = findOptions(argv);
  Retryer retryer = this.retryer.clone();
  while (true) {
    try {
      return executeAndDecode(template, options);
    } catch (RetryableException e) {
      try {
        retryer.continueOrPropagate(e);
      } catch (RetryableException th) {
        Throwable cause = th.getCause();
        if (propagationPolicy == UNWRAP && cause != null) {
          throw cause;
        } else {
          throw th;
        }
      }
      if (logLevel != Logger.Level.NONE) {
        logger.logRetry(metadata.configKey(), logLevel);
      }
      continue;
    }
  }
}
```

**An example below, with various annotation usage demo like @GetMapping, @PutMapping, @PathVariable, @RequetParam, @RequestHeader, @RequestBody etc..**

## Order Application

```java
@FeignClient(name = "product-service", url = "${feign.client.product-service.url}")
public interface ProductClient {

    @GetMapping("/products/{id}")
    String getProductById(@PathVariable("id") String id);

    @PutMapping(value = "/products/update/{id}", consumes = "application/json")
    Product updateProduct(
            @PathVariable("id") String id,
            @RequestParam("sendMail") boolean sendMail,
            @RequestHeader("X-ConceptCoding-ID") String uniqueID,
            @RequestBody Product updatedProductDetails
    );
}
```

*Ordering is not mandatory, Springboot will take care to pass values according to annotations.*

## Product Application

```java
@RestController
@RequestMapping("/products")
public class ProductController {

    @GetMapping("/{id}")
    public ResponseEntity<String> getProduct(@PathVariable String id) {
        return ResponseEntity.ok().body("fetch the product details with id:" + id);
    }

    @PutMapping("/update/{id}")
    public ResponseEntity<Product> createProduct(@PathVariable String id,
                                                 @RequestBody Product product,
                                                 @RequestParam("sendMail") boolean sendMail,
                                                 @RequestHeader("X-ConceptCoding-ID") String uniqueID) {

        //Product with id is fetched from DB & it is updated with incoming product values.
        Product dbProductObject = getProductFromDB(id);
        dbProductObject.setName(product.getName());

        //save the updated object back to db and return
        return ResponseEntity.ok().body(dbProductObject);
    }
}
```

## Encoder and Decoder in FeignClient

- **Encoder**: Converts a Java object into a request body (say JSON).
- **Decoder**: Converts the HTTP response body (say JSON) into a Java object.

```java
@Override
public Object invoke(Object[] argv) throws Throwable {
    RequestTemplate template = buildTemplateFromArgs.create(argv);
    Options options = findOptions(argv);
    Retryer retryer = this.retryer.clone();
    while (true) {
        try {
            return executeAndDecode(template, options);
        } catch (RetryableException e) {
            try {
                retryer.continueOrPropagate(e);
            } catch (RetryableException th) {
                Throwable cause = th.getCause();
                if (propagationPolicy == UNWRAP && cause != null) {
                    throw cause;
                } else {
                    throw th;
                }
            }
            if (logLevel != Logger.Level.NONE) {
                logger.logRetry(metadata.configKey(), logLevel);
            }
            continue;
        }
    }
}
```

*Internally this method, it make use of encoder, by-default it uses Jackson, and in RestTemplate body will be put as Raw JSON*

```
encoder.encode(body, metadata.bodyType(), mutable);
```

*Internally this method, it make use of decoder and by-default it uses Jackson. Reads the Response body (byte Stream) and covert it into Java object (return type of the FeignClient method)*

```
decoder.decode(response, returnType);
```

If we want our custom Encoder and Decoder implementation:

*All custom configuration defined in ProductClientConfig, is applicable for this ProductClient only.*

*We can have many ClientConfig like SalesClientConfig, CustomerClientConfig etc. with their own custom configuration & implementation. Not impacting each other.*

```java
@FeignClient(name = "product-service",
        url = "${feign.client.product-service.url}",
        configuration = ProductClientConfig.class)
public interface ProductClient {

    @PutMapping(value = "/products/update/{id}", consumes = "application/json")
    Product updateProduct(
            @PathVariable("id") String id,
            @RequestParam("sendMail") boolean sendMail,
            @RequestHeader("X-ConceptCoding-ID") String uniqueID,
            @RequestBody Product updatedProductDetails
    );
}
```

My custom Encoder Class:

```java
public class MyCustomProductClientEncoder implements Encoder {


    @Override
    public void encode(Object object, Type bodyType, RequestTemplate template) throws EncodeException {

        // manually converting object to JSON
        try {
            String jsonString = new ObjectMapper().writeValueAsString(object);
            template.body(jsonString);
        } catch (Exception e) {
            throw new EncodeException("Unable to encode object");
        }
    }
}
```

```java
@Configuration
public class ProductClientConfig {

    @Bean
    public Encoder myCustomEncoder() {
        return new MyCustomProductClientEncoder();
    }

    @Bean
    public Decoder myCustomDecoder() {
        return new MyCustomProductClientDecoder();
    }
}
```

My custom Decoder Class:

```java
public class MyCustomProductClientDecoder implements Decoder {

    @Override
    public Object decode(Response response, Type type) throws IOException, DecodeException, FeignException {

        // reading raw response body
        InputStream responseBody = response.body().asInputStream();

        //parsing JSON and coverts to Java object type
        return new ObjectMapper().readValue(responseBody, new TypeReference<Object>() {
            @Override public Type getType() { return type; }
        });
    }
}
```

## ErrorDecoder in FeignClient

- It is used to handle non 2xx status codes like 4xx and 5xx.

```java
@Override
public Object invoke(Object[] argv) throws Throwable {
    RequestTemplate template = buildTemplateFromArgs.create(argv);
    Options options = findOptions(argv);
    Retryer retryer = this.retryer.clone();
    while (true) {
        try {
            return executeAndDecode(template, options);
        } catch (RetryableException e) {
            try {
                retryer.continueOrPropagate(e);
            } catch (RetryableException th) {
                Throwable cause = th.getCause();
                if (propagationPolicy == UNWRAP && cause != null) {
                    throw cause;
                } else {
                    throw th;
                }
            }
            if (logLevel != Logger.Level.NONE) {
                logger.logRetry(metadata.configKey(), logLevel);
            }
            continue;
        }
    }
}
```

*Internally while handling the response, if status is not 2xx, it invokes ErrorDecoder decode() method.*

```java
errorDecoder.decode(methodKey, response);
```

*ErrorDecoder.java*

```java
@Override
public Exception decode(String methodKey, Response response) {
    FeignException exception = errorStatus(methodKey, response, maxBodyBytesLength,
            maxBodyCharsLength);
    Long retryAfter = retryAfterDecoder.apply(firstOrNull(response.headers(), RETRY_AFTER));
    if (retryAfter != null) {
        return new RetryableException(
                response.status(),
                exception.getMessage(),
                response.request().httpMethod(),
                exception,
                retryAfter,
                response.request());
    }
    return exception;
}
```

*Return FeignException, which includes: HttpStatus, Response Body and header. sample error example:*

*feign.FeignException$BadRequest: [400] during [PUT] to [http://localhost:8082/products/update/123?sendMail=false] [ProductClient#updateProduct(String,boolean,String,Product)]: [{"name":"newProduct","id":null}]*

If we want our custom ErrorDecoder implementation:

```java
@FeignClient(name = "product-service",
        url = "${feign.client.product-service.url}",
        configuration = ProductClientConfig.class)
public interface ProductClient {

    @PutMapping(value = "/products/update/{id}", consumes = "application/json")
    Product updateProduct(
            @PathVariable("id") String id,
            @RequestParam("sendMail") boolean sendMail,
            @RequestHeader("X-ConceptCoding-ID") String uniqueID,
            @RequestBody Product updatedProductDetails
    );
}
```

```java
public class MyCustomProductClientErrorDecoder implements ErrorDecoder {

    private final ErrorDecoder defaultErrorDecoder = new Default();

    @Override
    public Exception decode(String methodKey, Response response) {

        HttpStatus statusCode = HttpStatus.valueOf(response.status());

            if (statusCode.is4xxClientError()) {
                return new MyCustomBadRequestException("Client Error");
            }
            else if (statusCode.is5xxServerError()) {
                return new MyCustomServerException("Server Error");
            }
            else {
                return defaultErrorDecoder.decode(methodKey, response);
            }
    }
}
```

```java
@Configuration
public class ProductClientConfig {

    @Bean
    public ErrorDecoder myCustomErrorDecoder() {
        return new MyCustomProductClientErrorDecoder();
    }
}
```

```
2025-06-07T20:26:24.770+05:30  INFO 24295 --- [nio-8081-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
2025-06-07T20:26:24.812+05:30 ERROR 24295 --- [nio-8081-exec-1] o.a.c.c.C.[.[.[/].[dispatcherServlet]    : Servlet.service() for servlet [dispatcherServlet]

com.conceptandcoding.orderservice.BadRequestException Create breakpoint : Client Error
    at com.conceptandcoding.orderservice.MyCustomProductClientErrorDecoder.decode(MyCustomProductClientErrorDecoder.java:17) ~[classes/:na]
    at feign.InvocationContext.decodeError(InvocationContext.java:126) ~[feign-core-13.2.1.jar:na]
    at feign.InvocationContext.proceed(InvocationContext.java:72) ~[feign-core-13.2.1.jar:na]
    at feign.ResponseHandler.handleResponse(ResponseHandler.java:63) ~[feign-core-13.2.1.jar:na]
```

## Retryer in FeignClient

```
@Override
public Object invoke(Object[] argv) throws Throwable {
    RequestTemplate template = buildTemplateFromArgs.create(argv);
    Options options = findOptions(argv);
    Retryer retryer = this.retryer.clone();
    while (true) {
        try {
            return executeAndDecode(template, options);
        } catch (RetryableException e) {
            try {
                retryer.continueOrPropagate(e);
            } catch (RetryableException th) {
                Throwable cause = th.getCause();
                if (propagationPolicy == UNWRAP && cause != null) {
                    throw cause;
                } else {
                    throw th;
                }
            }
            if (logLevel != Logger.Level.NONE) {
                logger.logRetry(metadata.configKey(), logLevel);
            }
            continue;
        }
    }
}
```

*During execute, if any exception happens, first its checked, if it can be retried or not.*

- Retry only happens when there is either:
  - Connection time out
  - Network related exception like (IOException)
- After all retry finished, then ErrorDecoder is invoked.
- For 4xx and 5xx, retry do not happens, its handled by ErrorDecoder directly.

```
public interface Retryer extends Cloneable {

    if retry is permitted, return (possibly after sleeping). Otherwise, propagate the exception.
    void continueOrPropagate(RetryableException e);

    Retryer clone();

    class Default implements Retryer {

        private final int maxAttempts;
        private final long period;
        private final long maxPeriod;
        int attempt;
        long sleptForMillis;

        public Default() {      this( period: 100, SECONDS.toMillis( duration: 1),  maxAttempts: 5); }
```

- Default it uses Retryer.Default :
  - 5 times call attempt (includes the 1st call too)
  - Initial wait time between retries is 100ms
  - Wait time double with each retry
  - But max wait time can be 1second

- Try 1 : (immediate attempt)
- Try 2 : wait 100ms
- Try 3 : wait 200ms
- Try 4 : wait 400ms
- Try 5 : wait 800ms (but max capped at 1 second)

After all retry attempt finished, ErrorDecoder is invoked.

If, we don't want to retry at all, we can use Retryer.NEVER_RETRY (this already present in Retryer class)

```
@Configuration
public class ProductClientConfig {

    @Bean
    public Retryer myCustomRetryer() {
        return Retryer.NEVER_RETRY;
    }
}
```

If, we want custom implementation

```
@FeignClient(name = "product-service",
        url = "${feign.client.product-service.url}",
        configuration = ProductClientConfig.class)
public interface ProductClient {

    @PutMapping(value = "/products/update/{id}", consumes = "application/json")
    Product updateProduct(
            @PathVariable("id") String id,
            @RequestParam("sendMail") boolean sendMail,
            @RequestHeader("X-ConceptCoding-ID") String uniqueID,
            @RequestBody Product updatedProductDetails
    );
}
```

UserCase-1 : I only want to control Attempt, wait time and max period, rest I want to reuse the "Retryer.Default" logic.

```
public class MyCustomRetryer extends Retryer.Default {

    //i just need to control the attempts, wait time only, rest using Default implementation
    public MyCustomRetryer() {
        super( period: 200,  maxPeriod: 1000,  maxAttempts: 4);

    }
}
```

```
@Configuration
public class ProductClientConfig {

    @Bean
    public Retryer myCustomRetryer() {
        return new MyCustomRetryer();
    }
}
```

UserCase-2 : I want full control, then I have to implement Retryer itself and provide the custom implementation for the "continueOrPropogate()" method.

```
public class MyCustomRetryer implements Retryer {

    private int attempt = 1;
    private final int maxAttempts = 5;

    @Override
```

```
        return new MyCustomRetryer();
    }
}
```

```
    public void continueOrPropagate(RetryableException e) {

        //your custom logic, to check if attempt increases the max attempt
        //then throw exception

        if(attempt >= maxAttempts) {
            throw e;
        }
        attempt++;
        try {
            Thread.sleep( millis: 100);
        }
        catch (InterruptedException ie) {
                //do something
        }
    }

    @Override
    public Retryer clone() {
        return new MyCustomRetryer();
    }
}
```

Last but not the least:

During start, we discussed that, this name is just a arbitrary value. And its just we are giving the name to our FeignClient.

```
@FeignClient(name = "product-service",
        url = "${feign.client.product-service.url}")
public interface ProductClient {

    @GetMapping("/products/{id}")
    String getProductById(@PathVariable("id") String id);

}
```

But where its exactly used?

Yes, this name comes handy, when we have to provide any configuration in application.properties

If we want to set request and connection timeout only for product-service FeignClient

application.properties

```
#request and connection timeout applicable to only Product-service FeignClient
feign.client.config.product-service.connectTimeout=3000
feign.client.config.product-service.readTimeout=5000
```

If we want to set request and connection timeout for all FeignClient

application.properties

```
#request and connection timeout applicable for all FeignClient
feign.client.config.default.connectTimeout=3000
feign.client.config.default.readTimeout=5000
```