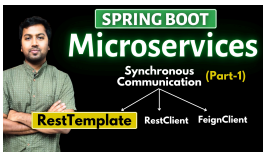
RestClient

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:

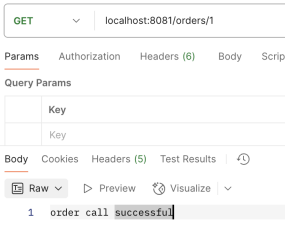
OrderService *(running on localhost:8081)* — *OrderService needs to invoke ProductService* → **ProductService** *(running on localhost:8082)*

```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");
    }
}
```

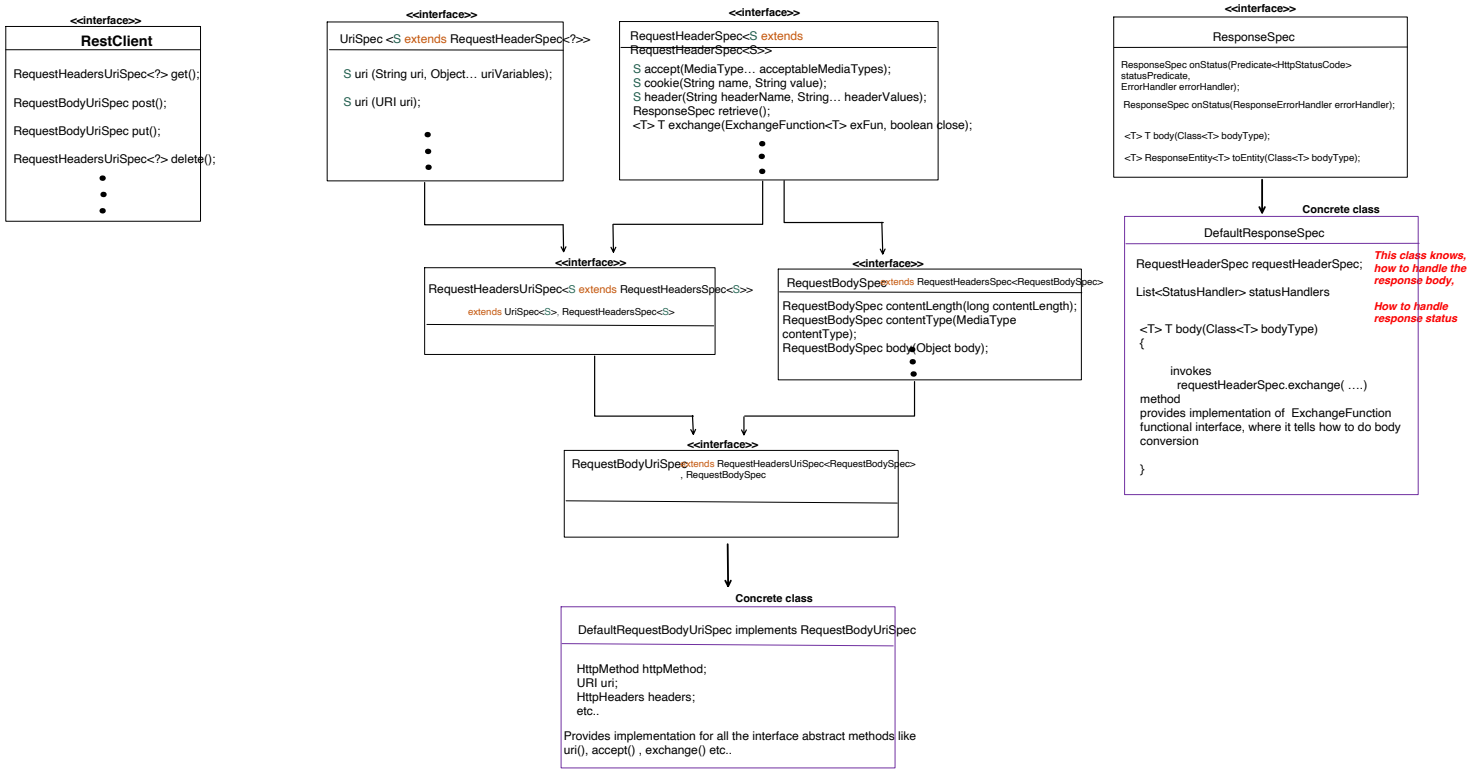
Same template, we are going to use for all different operation like post, put, delete etc.
No different methods for different operations like RestTemplate



```
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).**



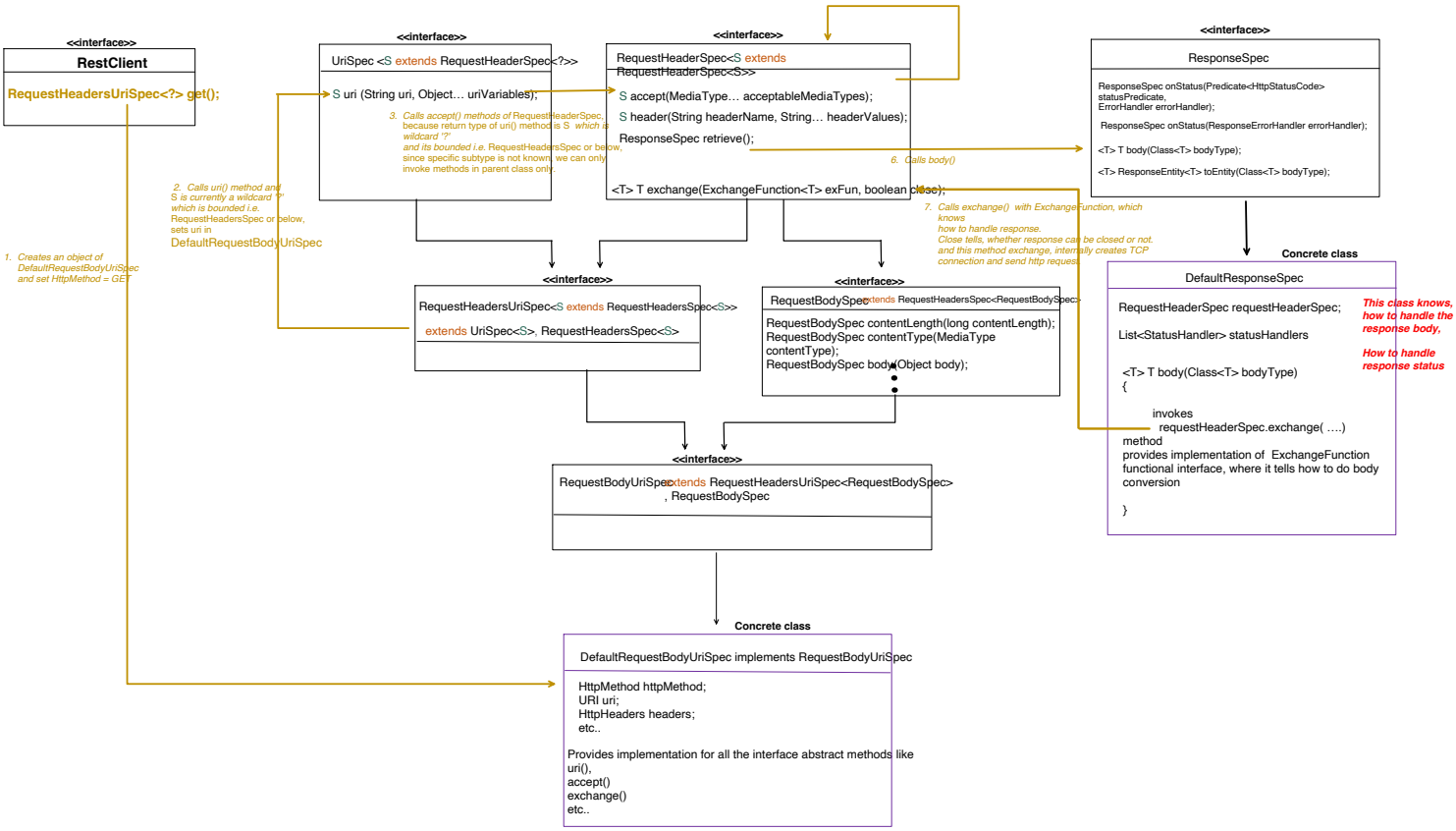
I am assuming that, we all know about Generics Class and Wildcards in Java

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);
```

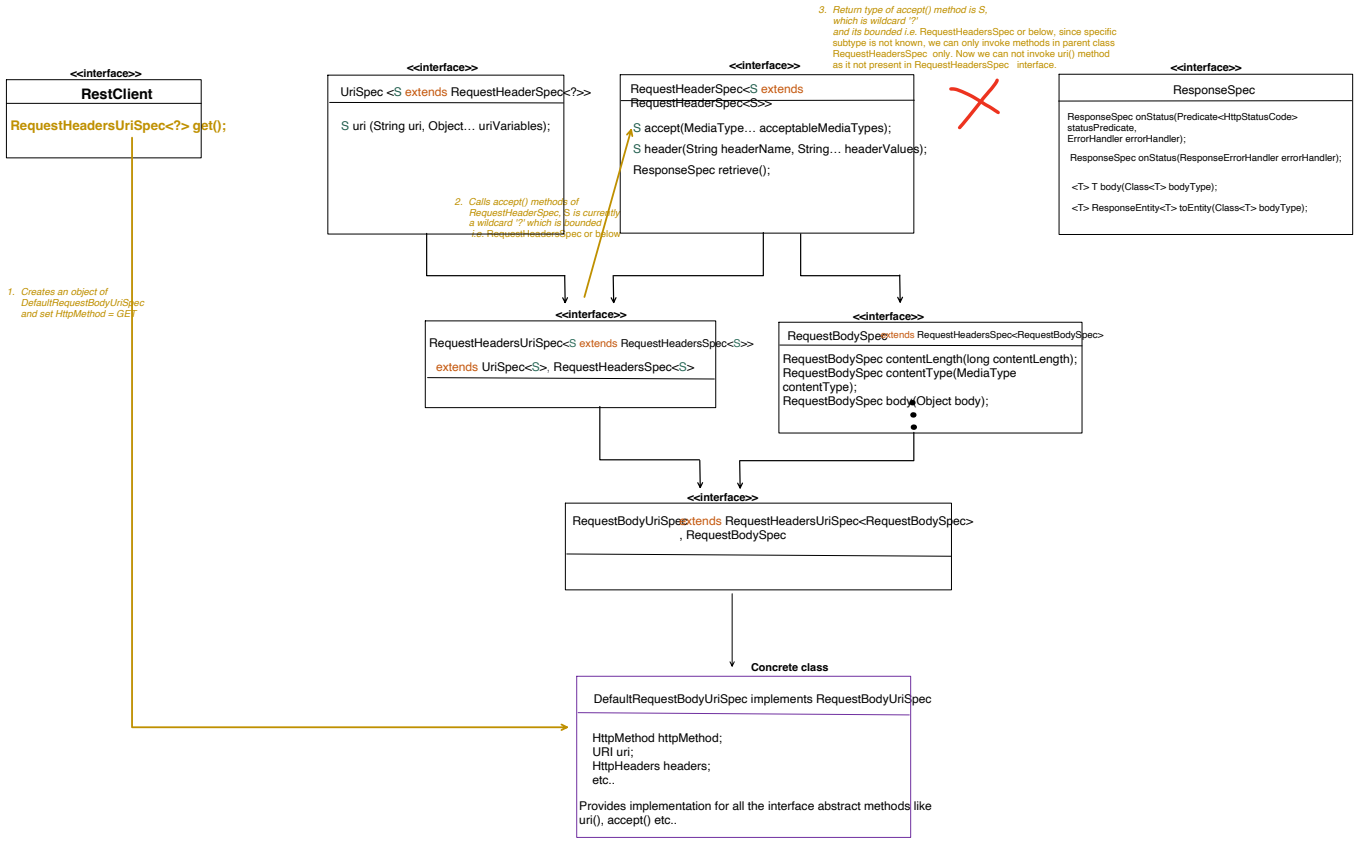


Let see, how flow can be broken

GET

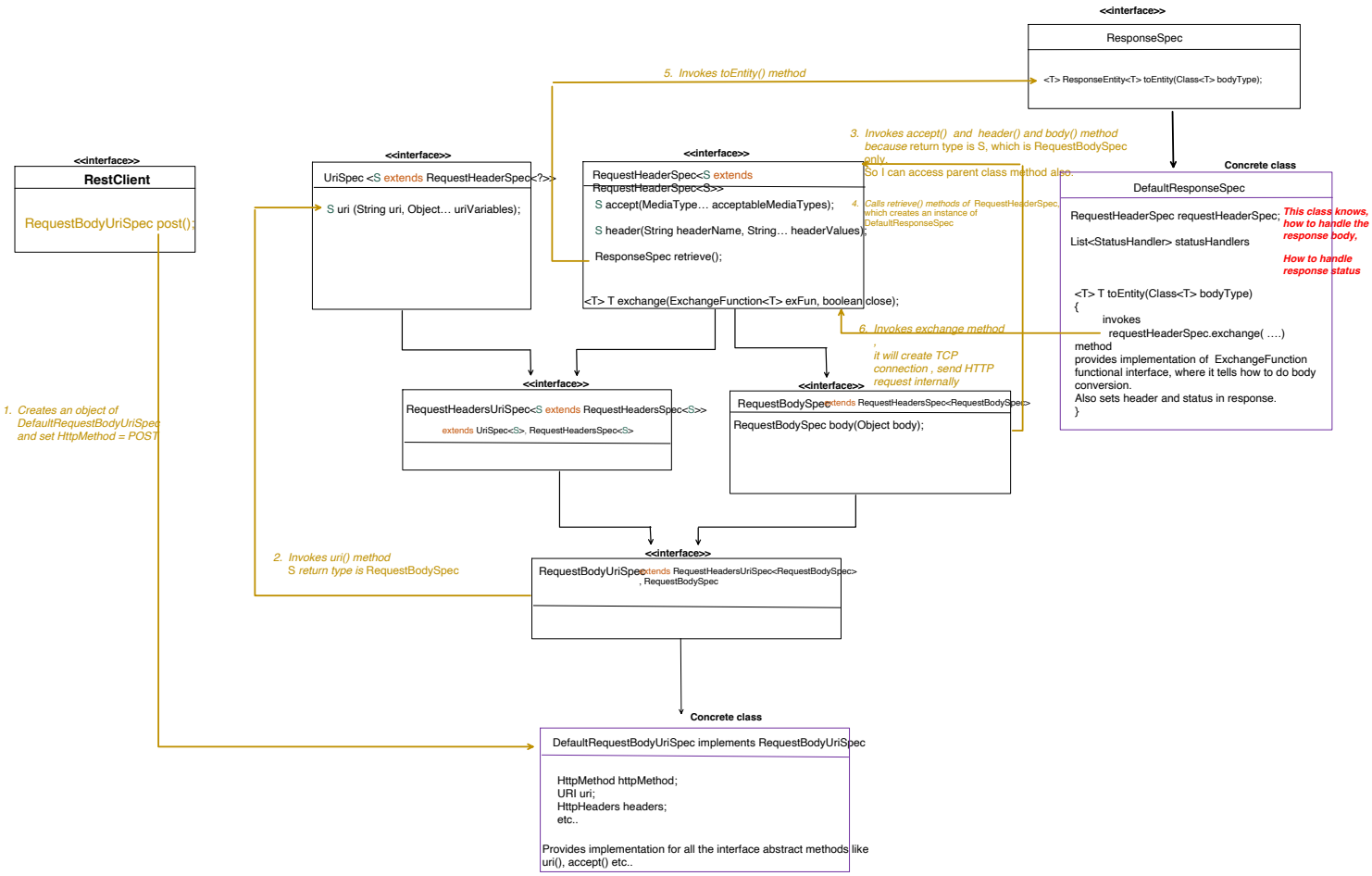
```
RestClient restClient = RestClient.create();

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?



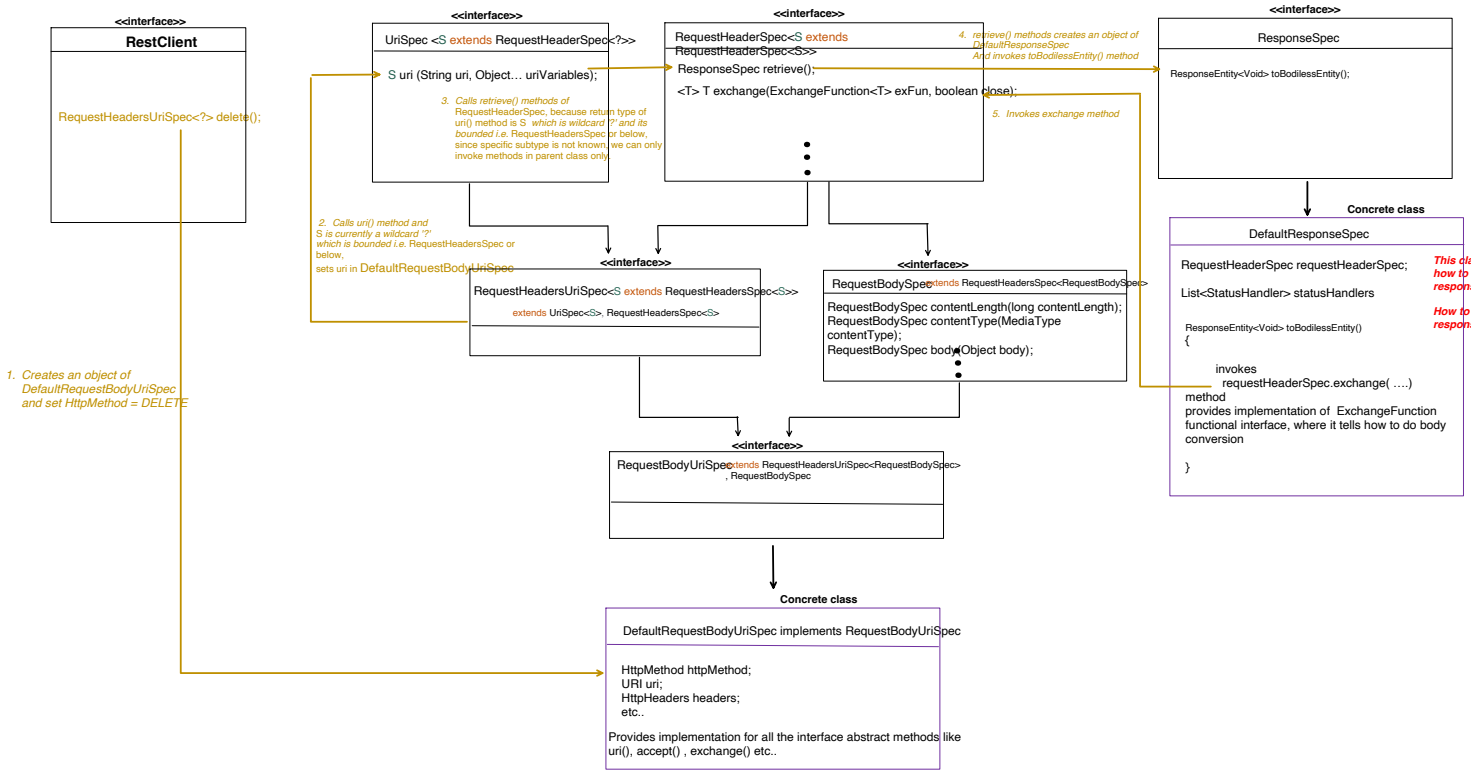
POST

```
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();
```



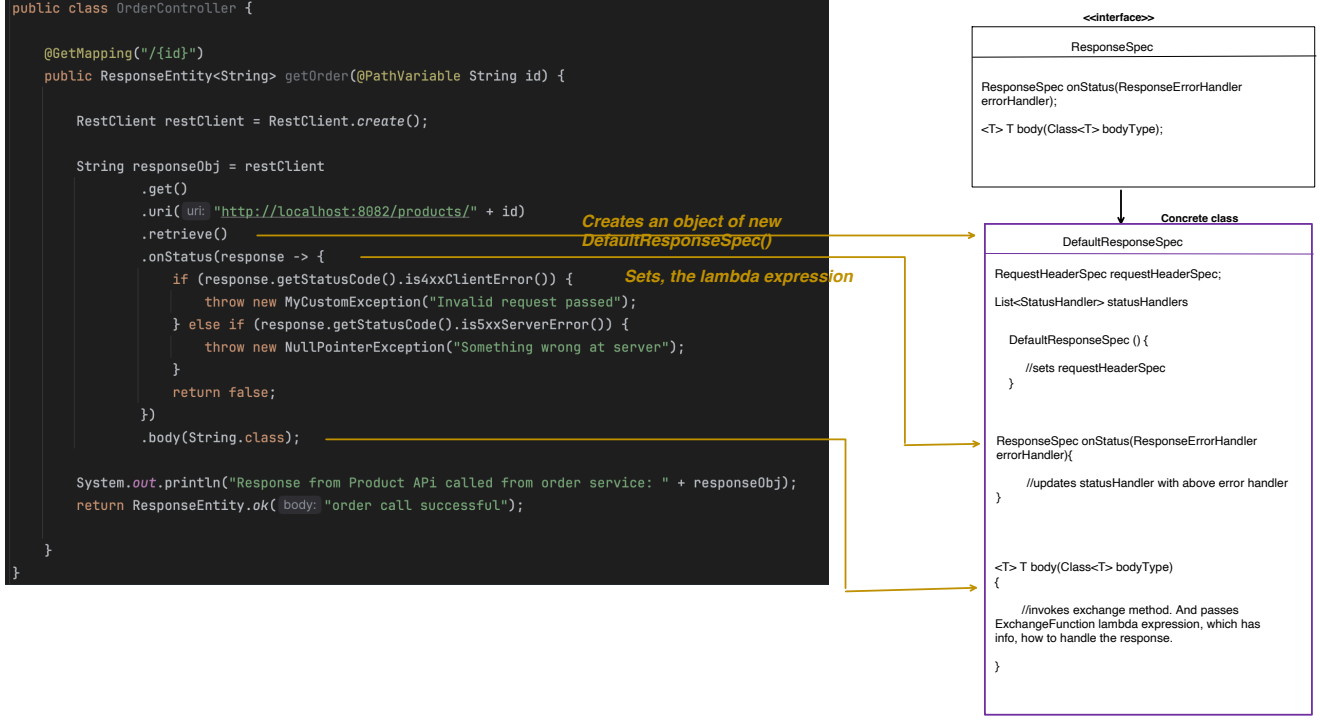
DELETE

```
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");

    }
}
```



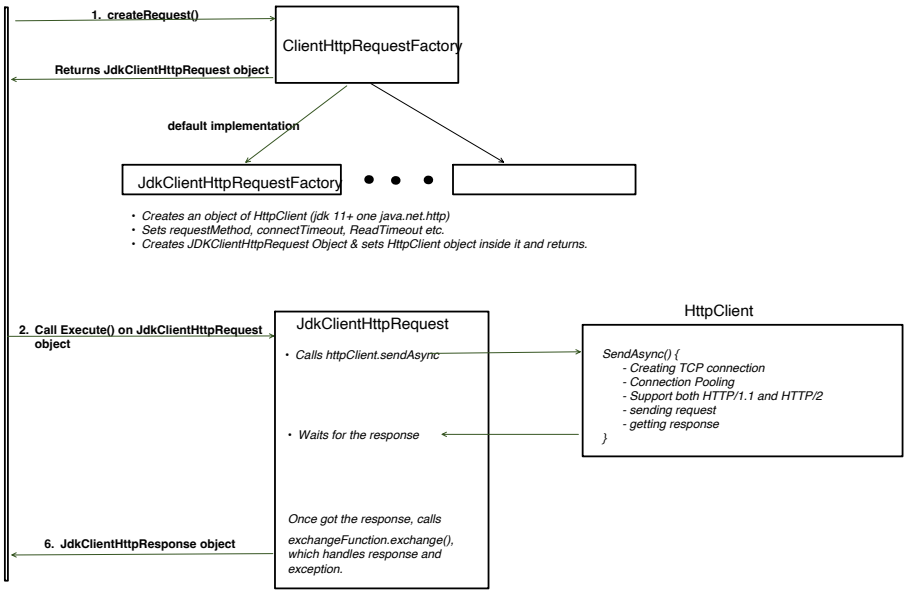
If we want full control over Response building and Exception handling

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);
    }
}
```

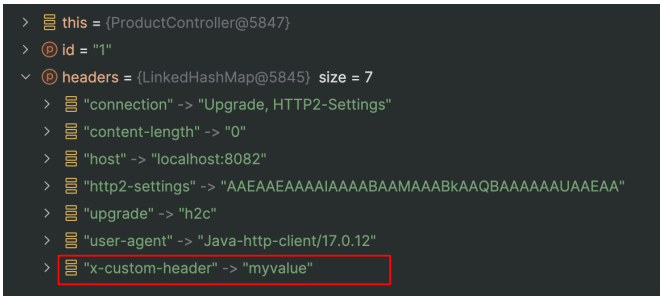
```
@Configuration
public class AppConfig {

    @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