FeignClient

Friday, 6 June 2025 5:29 PM

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 →

ProductService

(running on localhost:8082)

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

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

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

    @GetMapping("/{id}")
    public String getProduct(@P
        return "Product fetched
    }
}
```

```
}
```

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

```
}
```

application.properties

```
server.port=8081

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

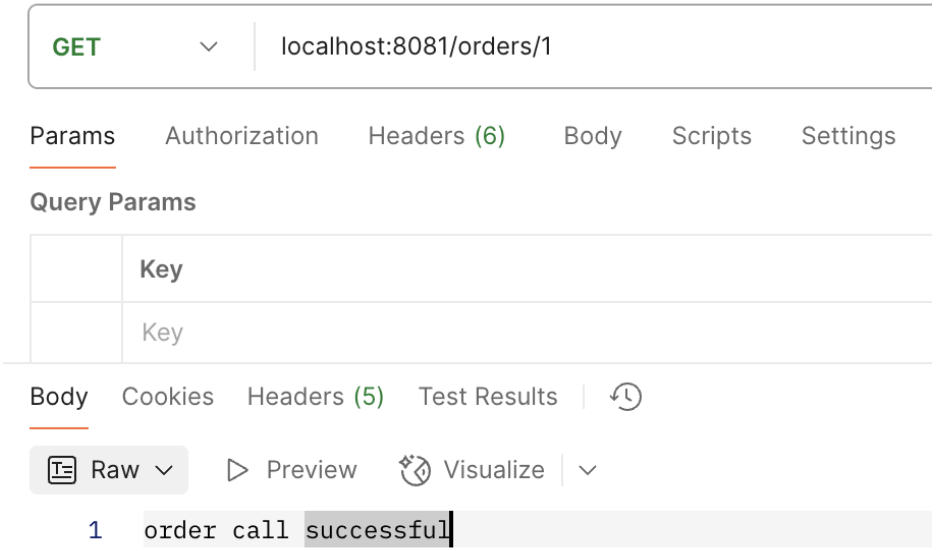
```
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
```

```java
    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 usin
2025-06-06T21:11:17.439+05:30  INFO 18774 --- [           main] c.c.o.OrderserviceApplication            : No active profile set, falling back t
2025-06-06T21:11:17.705+05:30  INFO 18774 --- [           main] o.s.cloud.context.scope.GenericScope     : BeanFactory id=f55ac312-243e-3904-baf
2025-06-06T21:11:17.810+05:30  INFO 18774 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8081 (ht
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 Tomc
2025-06-06T21:11:17.836+05:30  INFO 18774 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebAppli
2025-06-06T21:11:17.837+05:30  INFO 18774 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initializ
2025-06-06T21:11:17.999+05:30  INFO 18774 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8081 (http) wi
2025-06-06T21:11:18.005+05:30  INFO 18774 --- [           main] c.c.o.OrderserviceApplication            : Started OrderserviceApplication in 0.
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 'dispatcherServl
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
```

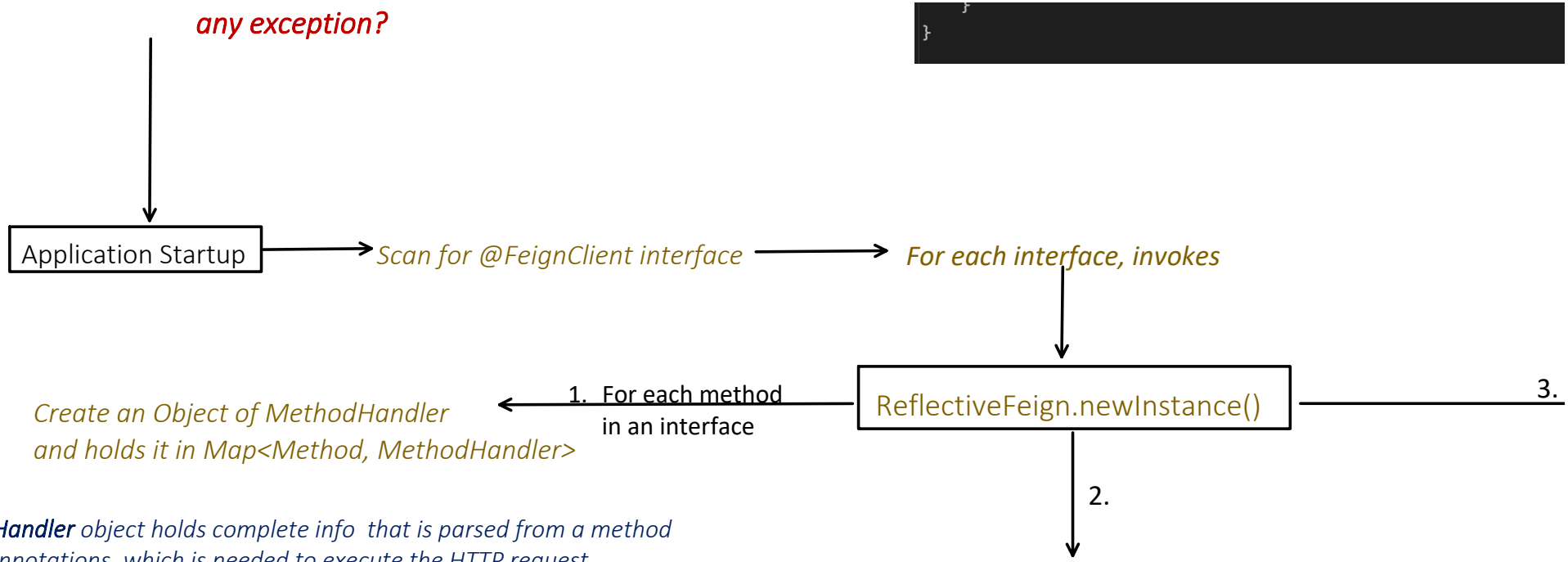
So, first important thing to understand is, how this Declarative HTTP Calls works?

```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

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

    @Autowired
    ProductClient productClient;

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

        String responseFromProductAPI = productClient.getProductBy
        System.out.println("Response from Product api call is: " +

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

```
    }
}
```



Creates an Object of InvocationHandler and populate the variable: Map<Method, MethodHandler> with info which we get in #1.

InvocationHandler, 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.*

```
@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
```

***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

```
    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.*

```
@Override
public Object invoke(Object[] argv) throws Throwable {
```

```java
    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, @PathVariab

@RequetParam, @RequestHeader, @RequestBody etc..

Order Application

```
@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.

```
@RestController
@RequestMapping("/product
public class ProductContr

    @GetMapping("/{id}")
    public ResponseEntity
        return ResponseEn
    }

    @PutMapping("/update/
    public ResponseEntity


        //Product with id
        Product dbProduct
        dbProductObject.s

        //save the update
        return ResponseEn
    }
}
```

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) {
```

Internally this method, it make use
body will be put as Raw JSON

```java
encoder.encode(body, metada
```

Internally this method, it make u
Reads the Response body (byte S
FeignClient method)

```
                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;
        }
    }
}
```

```
decoder.decode(respor
```

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.

```
@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(
```

```
public class MyCustomProduc

    @Override
    public void encode(Obje

        // manually convert
        try {
```

```
            @PathVariable("id") String id,
            @RequestParam("sendMail") boolean sendMail,
            @RequestHeader("X-ConceptCoding-ID") String uniqueID,
            @RequestBody Product updatedProductDetails
    );
}
```

```
            String jsonStri
            template.body(j
        } catch (Exception
            throw new Encod
        }
    }
}
```

```
@Configuration
public class ProductClientConfig {

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

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

```
public class MyCustomProductCli

    @Override
    public Object decode(Respor

        // reading raw response
        InputStream responseBod

        //parsing JSON and cove
        return new ObjectMapper
            @Override public Ty
        });
    }
}
```

ErrorDecoder in FeignClient

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

```
@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();
```

Internally while handling the respc
ErrorDecoder decode() method.

```
errorDecoder.decode(
```

```
                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;
        }
    }
}
```

```
@Override
public Exception de
    FeignException
            maxBody
    Long retryAfter
    if (retryAfter
        return new
                res
                exc
                res
                exc
                ret
                res
    }
    return exceptio
}
```

If we want our custom ErrorDecoder 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,
```

```
public class MyCustomProductClientErro

    private final ErrorDecoder default

    @Override
    public Exception decode(String met
```

```
        @RequestParam("sendMail") boolean sendMail,
        @RequestHeader("X-ConceptCoding-ID") String uniqueID,
        @RequestBody Product updatedProductDetails
    );
}
```

```
        HttpStatus statusCode = HttpSt

            if (statusCode.is4xxClient
                return new MyCustomBad
            }
            else if (statusCode.is5xxS
                return new MyCustomSer
            }
            else {
                return defaultErrorDec
            }
    }
}
```

```
@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 initializati
2025-06-07T20:26:24.812+05:30 ERROR 24295 --- [nio-8081-exec-1] o.a.c.c.C.[.[.[/].[dispatcherServlet]    : Servlet.service() for

com.conceptandcoding.orderservice.BadRequestException Create breakpoint : Client Error
    at com.conceptandcoding.orderservice.MyCustomProductClientErrorDecoder.decode(MyCustomProductClientErrorDecoder.java:17) ~[c
    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

```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) {
```

During execute, if any exception happe
checked, if it can be retried or not.

```
                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;
        }
    }
}
```

- 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);
```

```
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 invok

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 {
```

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

If, we want custom 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
    );
}
```

UserCase-1 : I only want to contrc
reuse the "Retryer.Default" logic.

```java
public class MyCustomRetryer exte

    //i just need to control the
    public MyCustomRetryer() {
        super( period: 200,  maxPeri

    }
}
```

UserCase-2 : I want full control, th

```java
@Configuration
```

```java
@Configuration
public class ProductClientConfig {

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

custom implementation for the "c

```java
public class MyCustomRetryer implem

    private int attempt = 1;
    private final int maxAttempts =

    @Override
    public void continueOrPropagate

        //your custom logic, to che
        //then throw exception

        if(attempt >= maxAttempts)
            throw e;
        }
        attempt++;
        try {
            Thread.sleep( millis: 100)
        }
        catch (InterruptedExceptior
                //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
```