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

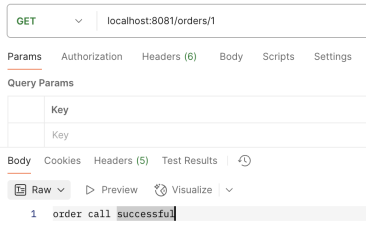
```
server.port=8081

#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
```