

FEIGN

FEIGN

Es una librería que forma parte del stack de spring-cloud, fue desarrollada por Netflix, con el objetivo de generar clientes de servicios REST de forma declarativa, esto abre paso a la comunicación entre microservicios de forma sencilla. Su forma de usarse es muy parecida a la capa repositorio que acostumbramos a usar en los proyectos spring boot, simplemente se escriben los métodos en interfaces, con anotaciones para identificar datos como url, verbo, parámetros de entrada, etc.

Para importar la dependencia a nuestro proyecto agregaremos el groupId

org. `springframework.cloud` con el artifactId `spring-cloud-starter-openfeign`.

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-openfeign</artifactId>
4 </dependency>
```

Adicional a esto es necesario definir la versión que manejarán las dependencias de spring-cloud, esto lo hacemos con el dependencyManagement dentro del pom.xml también.

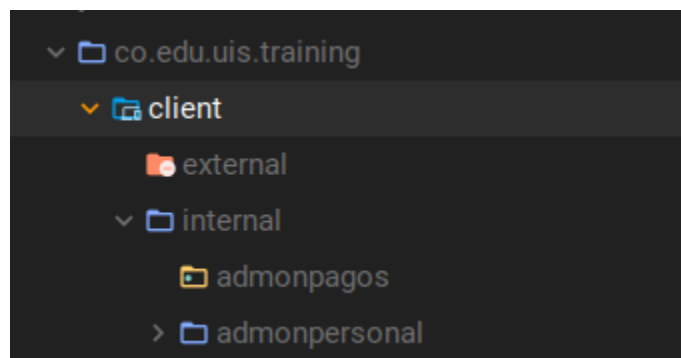
```
1 <dependencyManagement>
2   <dependencies>
3     <dependency>
4       <groupId>org.springframework.cloud</groupId>
5       <artifactId>spring-cloud-dependencies</artifactId>
6       <version>${spring-cloud.version}</version>
7       <type>pom</type>
8       <scope>import</scope>
9     </dependency>
10  </dependencies>
11 </dependencyManagement>
```

La versión de spring-cloud usada es **Hoxton.SR1**

Para habilitar Feign también es necesario poner una anotación en la clase principal de nuestro proyecto;

`@EnableFeignClients`, de esta manera se podrán empezar a crear interfaces y declararlas como clientes Feign que serán usadas en la lógica de nuestro proyecto para hacer peticiones a otros microservicios.

Ahora, con todo esto configurado, esta es la estructura que van a tener los paquetes donde se registrarán los clientes.



La estructura para el nombre de las interfaces debe ser: `INombreNombreProyectoClient`, por ejemplo: `IRolAdmonPersonalClient`.

La interface debe tener una anotación llamada `@FeignClient` que tiene unos parámetros llamados `name` y `url`, el `name` indicará el nombre del servicio, ejemplo: `rol-admonpersonal-service`, y la `url` es hacia donde irá dirigida la petición, para esto lo trabajaremos con variables que estarán puestas en el `application.properties`, estas variables contienen las `url`'s de los diferentes microservicios.

Los métodos se declararán del mismo modo que se declaran cuando se está trabajando con la capa controladora de spring-boot, usando las anotaciones ya conocidas, como: `@RequestMapping`, `@GetMapping`, `@PostMapping`, etc.

```
1 | @FeignClient(value = "rol-admonpersonal-service", url = "${admonpersonal.servicio}")
2 | @RequestMapping("/api/rol")
3 | public interface IRolAdmonPersonalClient {
4 |
5 |     @GetMapping("/all")
6 |     ResponseEntity<List<RolDTO>> findAll();
7 | }
```

Ahora bien, esto abre paso al hecho de poder comunicarse con otros microservicios, pero recordemos que los diferentes proyectos manejan seguridad; autorización y autenticación, por lo tanto es necesario enviar los headers con la información que solicita cualquier proyecto para autorizar una petición, para esto haremos uso de interceptores.

Implementación

Para usar los clientes de Feign se inyectan como cualquier otro componente de spring-boot, la recomendación es que los llamados a estos servicios o clientes se hagan desde la capa controller del proyecto a menos que

estos métodos involucren operaciones transaccionales asociadas al servicio que las llama, si esto es así, lo mejor es usar la anotación `@Transactional` dentro del método de la capa service que necesite llamar estos métodos. La inyección se hace como comunmente se han venido haciendo; declarando el componente y usando el `@Autowired` por medio de setters.

Dado que el uso de FEIGN implica comunicación entre microservicios, esto conlleva al transporte de información o de objetos, para esto es necesario utilizar DTO's, tanto para recibir información, como para enviarla.

```
1 private IRolAdmonPersonalClient admonPersonalRol;  
2  
3 @Autowired  
4 public void setAdmonPersonalRol(IRolAdmonPersonalClient admonPersonalRol) {  
5     this.admonPersonalRol = admonPersonalRol;  
6 }
```

Una vez inyectado el componente del cliente se pueden usar sus métodos, para este ejemplo a continuación estamos haciendolo para un servicio de tipo GET que trae todos los roles, este servicio está en el back de admonPersonal.

```
1 @GetMapping("/feign/test")  
2 public ResponseEntity<List<RolDTO>> getRolesByFeign(){  
3     return this.admonPersonalRol.findAll();  
4 }
```

También se podrían implementar los métodos de los clientes sin que devuelvan el `ResponseEntity`, esto depende de la lógica que se quiera aplicar con el uso del método que va ser invocado, el código es muy similar lo única diferencia es que la respuesta ya no irá encapsulada por un objeto de tipo `ResponseEntity`.

```
1 /**  
2  * Mismo método anterior pero respuesta sin el ResponseEntity  
3  */  
4 @GetMapping("/all")  
5 List<RolDTO> getAllWithoutResponseEntity();
```

Llamado desde el controller.

```
1 @GetMapping("/feign/noRE")
```

```

2   public ResponseEntity<List<RolDTO>> getRolesNoRE(){
3       return new ResponseEntity<>(this.admonPersonalRol.getAllWithoutResponseEni
4   }

```

Como se puede evidenciar anteriormente, esto lo que hace es que FEIGN toma el cuerpo de la petición y eso es lo que retornaría una vez se invoca el método.

Ahora haremos un ejemplo para una petición de tipo POST que está ubicada en el proyecto de admonPeronal, tiene un `@RequestBody` y lo que hace es guardar una clase situación administrativa en base de datos y posterior a eso eliminarla.

```

1   /**
2    * Método para guardar una clase situación administrativa
3    * @param claseSituacionAdministrativa -
4    */
5   @PostMapping("/")
6   ResponseEntity<Boolean> testingInterceptorPost(@RequestBody ClaseSituacionAdmini

```

Como es un `@RequestBody` es un parámetro obligatorio que se debe pasar al método cuando va ser invocado.

```

1   @PostMapping("/feign/test/post")
2   public ResponseEntity<Boolean> testPostRequest(@RequestBody ClaseSituacionAdmini
3       return this.auditoriaAdmonPersonalClient.testingInterceptorPost(claseSituac
4   }

```

Para el verbo PUT se comporta de la misma manera.

```

1   /**
2    * Método para testear petición PUT con @RequestBody
3    * @param claseSituacionAdministrativa -
4    */
5   @PutMapping("/update")
6   ResponseEntity<Boolean> testingInterceptorput(@RequestBody ClaseSituacionAdmini

```

Controller:

```

1   @PutMapping("/feign/test/put")
2   public ResponseEntity<Boolean> testPutRequest(@RequestBody ClaseSituacionAdmini

```

```

3      return this.auditoriaAdmonPersonalClient.testingInterceptorPut(claseSituac:
4  }

```

Ahora, un método que usa `@RequestParam` con el verbo DELETE, cuando lo invocamos este parámetro, irá como un parámetro de entrada propio del método, de esta manera redirigirá el parámetro a la petición.

```

1  /**
2   * Elimina una clase situación administrativa por id
3   * @param id -
4   */
5  @DeleteMapping("/")
6  ResponseEntity<Boolean> testingInterceptorDelete(@RequestParam(name = "id") Long id) {

```

Controller: Cuando el método está declarado con `ResponseEntity` podremos tomar también el cuerpo de la petición con el método `getBody()`, lo mismo si necesitáramos un header o similar, método `getHeader()`.

```

1  @DeleteMapping("/feign/test/delete")
2  public ResponseEntity<Boolean> testDeleteRequest(@RequestParam Long id){
3      var deleted = this.auditoriaAdmonPersonalClient.testingInterceptorDelete(id);
4      if (Boolean.TRUE.equals(deleted)){
5          return new ResponseEntity<>(true, HttpStatus.OK);
6      }
7      return new ResponseEntity<>(false, HttpStatus.CONFLICT);
8  }

```

Transaccionalidad

Cuando se vayan a hacer peticiones por medio de FEIGN que involucren operaciones transaccionales lo mejor es que esto se haga en la capa service con métodos de tipo `@Transactional`, esto con el fin de que si algo falla u ocurre una excepcion se haga un "rollback" del proceso.

Esto involucra que dentro de la capa service dadas estas ocasiones se manejen DTO's, esto quedaría contemplado para los casos donde se tenga que trabajar lógica de negocio de operaciones transaccionales con FEIGN.

```

1  @Override
2  @Transactional
3  public Boolean testFeignTransactional(ClasaSituacionAdministrativaDTOA claseSi
4      var saved = this.saveClaseSA(ClasaSituacionAdministrativaMapper
5          .INSTANCE.toClaseSituacionAdministrativa(claseSituacionAdministrat:

```

```

6         this.auditoriaAdmonPersonalClient.testingInterceptorPost(claseSituacionAdm:
7         this.deleteClaseSA(saved.getId());
8         return true;
9     }

```

Tratamiento de excepciones

Se pueden capturar las excepciones de FEIGN y redirigirlas como lo trabaja actualmente el poryecto RSI, para esto definimos un `ExceptionHandler` en el `RSIControllerException`.

```

1  @ExceptionHandler(FeignException.class)
2      public ResponseEntity<ErrorDTO> handleFeignStatusException(FeignException e) {
3          response.setStatus(e.status());
4          final HttpStatus codigoHttp = HttpStatus.BAD_REQUEST;
5
6          return new ResponseEntity<>(
7              new ErrorDTO(e.status(), e.getMessage(), TIPO_WARNING, request.getRequestURI(),
8              codigoHttp);
9      }

```

RequestInterceptor de FEIGN

FEIGN nos permite configurar un Bean con un `RequestInterceptor` del cuál podremos configurar los headers que enviará en cada petición, en este caso los que nos interesan son el token que va en `Authorization`, el `idRol` y `idUsuario`, estos los podemos tomar de la petición actual y luego envíoslos a todas las peticiones que se hagan por medio de FEIGN.

```

1  @Bean
2  public feign.RequestInterceptor requestInterceptor() {
3      return new feign.RequestInterceptor() {
4          @Override
5          public void apply(RequestTemplate requestTemplate) {
6              if (RequestContextHolder.getRequestAttributes() != null && requestTemplate != null) {
7                  requestTemplate.header(IConstantes.ID_USUARIO_HEADER, httpServletRequest.getHeader("id_usuario"));
8                  requestTemplate.header(IConstantes.ID_ROL_USUARIO_HEADER, httpServletRequest.getHeader("id_rol_usuario"));
9                  requestTemplate.header(IConstantes.ID_AUTHORIZATION, httpServletRequest.getHeader("Authorization"));
10                 requestTemplate.header(IConstantes.ACCEPT_LANGUAGE, httpServletRequest.getHeader("Accept-Language"));
11             }
12         }
13     };
14 }

```

El interceptor de FEIGN está adaptado también para cuando se hacen peticiones que no provienen de un request específico, es decir; que no son hechas desde un cliente sino algunos mecanismos automáticos disparan estas peticiones, haciendo que el interceptor detecte que el request es nulo y no añade headers innecesarios.

Ahora en los casos donde el cliente de FEIGN sea dirigido a servicios externos, debemos poner una clase específica llamada `PublicInterceptorFeign` en el atributo `configuration` de la anotación `@FeignClient`, esto para que no añada headers innecesarios a la petición, tal como lo muestra el siguiente ejemplo de código.

```
1  @FeignClient(value = "public-configuration-example", url = "${admonpersonal.sei
2      configuration = PublicInterceptorFeign.class)
3  @RequestMapping("/api-publica/ejemplo")
4  public interface PublicConfigurationInterface {
5
6      /**
7       * Método de ejemplo
8       */
9      @GetMapping("/")
10     ResponseEntity<HashMap<String,String>> exampleMethod();
11
12 }
```

Código

Los ejemplos mostrados anteriormente también se pueden visualizar en el proyecto de training.

Video: Charla sobre feign se puede encontrar en el link:

<https://drive.google.com/drive/u/4/folders/1fyd88F9mijdzjsRi54JbzsRuj2cPyuv> 