Spring Cloud OpenFeign

Table of Contents

2.0.3.BUILD-SNAPSHOT

This project provides OpenFeign integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.

1. Declarative REST Client: Feign

Feign is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud adds support for Spring MVC annotations and for using the same `HttpMessageConverters` used by default in Spring Web. Spring Cloud integrates Ribbon and Eureka to provide a load balanced http client when using Feign.

1.1 How to Include Feign

To include Feign in your project use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-openfeign`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

Example spring boot app

```
@SpringBootApplication
@EnableFeignClients
public class Application {

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

}
```

StoreClient.java.

```
@FeignClient("stores")
public interface StoreClient {
    @RequestMapping(method = RequestMethod.GET, value = "/stores")
    List<Store> getStores();

    @RequestMapping(method = RequestMethod.POST, value = "/stores/{storeId}", consumes = "application/json")
    Store update(@PathVariable("storeId") Long storeId, Store store);
}
```

In the `@FeignClient` annotation the String value ("stores" above) is an arbitrary client name, which is used to create a Ribbon load balancer (see below for details of Ribbon support). You can also specify a URL using the `url` attribute (absolute value or just a hostname). The name of the bean in the application context is the fully qualified name of the interface. To specify your own alias value you can use the `qualifier` value of the `@FeignClient` annotation.

The Ribbon client above will want to discover the physical addresses for the "stores" service. If your application is a Eureka client then it will resolve the service in the Eureka service registry. If you don’t want to use Eureka, you can simply configure a list of servers in your external configuration (see above for example).

1.2 Overriding Feign Defaults

A central concept in Spring Cloud’s Feign support is that of the named client. Each feign client is part of an ensemble of components that work together to contact a remote server on demand, and the ensemble has a name that you give it as an application developer using the `@FeignClient` annotation. Spring Cloud creates a new ensemble as an `ApplicationContext` on demand for each named client using `FeignClientsConfiguration`. This contains (amongst other things) an `feign.Decoder`, a `feign.Encoder`, and a `feign.Contract`.

Spring Cloud lets you take full control of the feign client by declaring additional configuration (on top of the `FeignClientsConfiguration`) using `@FeignClient`. Example:

```
@FeignClient(name = "stores", configuration = FooConfiguration.class)
public interface StoreClient {
    //..
}
```

In this case the client is composed from the components already in `FeignClientsConfiguration` together with any in `FooConfiguration` (where the latter will override the former).

`FooConfiguration` does not need to be annotated with `@Configuration`. However, if it is, then take care to exclude it from any `@ComponentScan` that would otherwise include this configuration as it will become the default source for `feign.Decoder`, `feign.Encoder`, `feign.Contract`, etc., when specified. This can be avoided by putting it in a separate, non-overlapping package from any `@ComponentScan` or `@SpringBootApplication`, or it can be explicitly excluded in `@ComponentScan`.

The `serviceId` attribute is now deprecated in favor of the `name` attribute.

Previously, using the `url` attribute, did not require the `name` attribute. Using `name` is now required.

Placeholders are supported in the `name` and `url` attributes.

```
@FeignClient(name = "${feign.name}", url = "${feign.url}")
public interface StoreClient {
    //..
}
```

Spring Cloud Netflix provides the following beans by default for feign (`BeanType` beanName: `ClassName`):

- `Decoder` feignDecoder: `ResponseEntityDecoder` (which wraps a `SpringDecoder`)
- `Encoder` feignEncoder: `SpringEncoder`
- `Logger` feignLogger: `Slf4jLogger`
- `Contract` feignContract: `SpringMvcContract`
- `Feign.Builder` feignBuilder: `HystrixFeign.Builder`
- `Client` feignClient: if Ribbon is enabled it is a `LoadBalancerFeignClient`, otherwise the default feign client is used.

The OkHttpClient and ApacheHttpClient feign clients can be used by setting `feign.okhttp.enabled` or `feign.httpclient.enabled` to `true`, respectively, and having them on the classpath. You can customize the HTTP client used by providing a bean of either `ClosableHttpClient` when using Apache or `OkHttpClient` when using OK HTTP.

Spring Cloud Netflix *does not* provide the following beans by default for feign, but still looks up beans of these types from the application context to create the feign client:

- `Logger.Level`
- `Retryer`
- `ErrorDecoder`
- `Request.Options`
- `Collection<RequestInterceptor>`
- `SetterFactory`

Creating a bean of one of those type and placing it in a `@FeignClient` configuration (such as `FooConfiguration` above) allows you to override each one of the beans described. Example:

```
@Configuration
public class FooConfiguration {
    @Bean
    public Contract feignContract() {
        return new feign.Contract.Default();
    }

    @Bean
    public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
        return new BasicAuthRequestInterceptor("user", "password");
    }
}
```

This replaces the `SpringMvcContract` with `feign.Contract.Default` and adds a `RequestInterceptor` to the collection of `RequestInterceptor`.

`@FeignClient` also can be configured using configuration properties.

application.yml

```
feign:
  client:
    config:
      feignName:
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: full
        errorDecoder: com.example.SimpleErrorDecoder
        retryer: com.example.SimpleRetryer
        requestInterceptors:
          - com.example.FooRequestInterceptor
          - com.example.BarRequestInterceptor
        decode404: false
        encoder: com.example.SimpleEncoder
        decoder: com.example.SimpleDecoder
        contract: com.example.SimpleContract
```

Default configurations can be specified in the `@EnableFeignClients` attribute `defaultConfiguration` in a similar manner as described above. The difference is that this configuration will apply to *all* feign clients.

If you prefer using configuration properties to configured all `@FeignClient`, you can create configuration properties with `default` feign name.

application.yml

```
feign:
  client:
    config:
      default:
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: basic
```

If we create both `@Configuration` bean and configuration properties, configuration properties will win. It will override `@Configuration` values. But if you want to change the priority to `@Configuration`, you can change `feign.client.default-to-properties` to `false`.

If you need to use `ThreadLocal` bound variables in your `RequestInterceptor`s you will need to either set the thread isolation strategy for Hystrix to `SEMAPHORE` or disable Hystrix in Feign.

application.yml

```
# To disable Hystrix in Feign
feign:
  hystrix:
    enabled: false

# To set thread isolation to SEMAPHORE
hystrix:
  command:
    default:
      execution:
        isolation:
          strategy: SEMAPHORE
```

1.3 Creating Feign Clients Manually

In some cases it might be necessary to customize your Feign Clients in a way that is not possible using the methods above. In this case you can create Clients using the Feign Builder API. Below is an example which creates two Feign Clients with the same interface but configures each one with a separate request interceptor.

```
@Import(FeignClientsConfiguration.class)
class FooController {

        private FooClient fooClient;

        private FooClient adminClient;

        @Autowired
        public FooController(Decoder decoder, Encoder encoder, Client client, Contract contract) {
                this.fooClient = Feign.builder().client(client)
                                .encoder(encoder)
                                .decoder(decoder)
                                .contract(contract)
                                .requestInterceptor(new BasicAuthRequestInterceptor("user", "user"))
                                .target(FooClient.class, "http://PROD-SVC");

                this.adminClient = Feign.builder().client(client)

                                .encoder(encoder)
                                .decoder(decoder)
                                .contract(contract)
                                .requestInterceptor(new BasicAuthRequestInterceptor("admin", "admin"))
                                .target(FooClient.class, "http://PROD-SVC");
    }
}
```

In the above example `FeignClientsConfiguration.class` is the default configuration provided by Spring Cloud Netflix.

`PROD-SVC` is the name of the service the Clients will be making requests to.

The Feign `Contract` object defines what annotations and values are valid on interfaces. The autowired `Contract` bean provides supports for SpringMVC annotations, instead of the default Feign native annotations.

1.4 Feign Hystrix Support

If Hystrix is on the classpath and `feign.hystrix.enabled=true`, Feign will wrap all methods with a circuit breaker. Returning a `com.netflix.hystrix.HystrixCommand` is also available. This lets you use reactive patterns (with a call to `.toObservable()` or `.observe()` or asynchronous use (with a call to `.queue()`).

To disable Hystrix support on a per-client basis create a vanilla `Feign.Builder` with the "prototype" scope, e.g.:

```
@Configuration
public class FooConfiguration {
        @Bean
        @Scope("prototype")
        public Feign.Builder feignBuilder() {
                return Feign.builder();
        }
}
```

Prior to the Spring Cloud Dalston release, if Hystrix was on the classpath Feign would have wrapped all methods in a circuit breaker by default. This default behavior was changed in Spring Cloud Dalston in favor for an opt-in approach.

1.5 Feign Hystrix Fallbacks

Hystrix supports the notion of a fallback: a default code path that is executed when they circuit is open or there is an error. To enable fallbacks for a given `@FeignClient` set the `fallback` attribute to the class name that implements the fallback. You also need to declare your implementation as a Spring bean.

```
@FeignClient(name = "hello", fallback = HystrixClientFallback.class)
protected interface HystrixClient {
    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    Hello iFailSometimes();
}

static class HystrixClientFallback implements HystrixClient {
    @Override
    public Hello iFailSometimes() {
        return new Hello("fallback");
    }
}
```

If one needs access to the cause that made the fallback trigger, one can use the `fallbackFactory` attribute inside `@FeignClient`.

```
@FeignClient(name = "hello", fallbackFactory = HystrixClientFallbackFactory.class)
protected interface HystrixClient {
        @RequestMapping(method = RequestMethod.GET, value = "/hello")
        Hello iFailSometimes();
}

@Component
static class HystrixClientFallbackFactory implements FallbackFactory<HystrixClient> {
        @Override
        public HystrixClient create(Throwable cause) {
                return new HystrixClient() {
                        @Override
                        public Hello iFailSometimes() {
                                return new Hello("fallback; reason was: " + cause.getMessage());
                        }
                };
        }
}
```

There is a limitation with the implementation of fallbacks in Feign and how Hystrix fallbacks work. Fallbacks are currently not supported for methods that return `com.netflix.hystrix.HystrixCommand` and `rx.Observable`.

1.6 Feign and `@Primary`

When using Feign with Hystrix fallbacks, there are multiple beans in the `ApplicationContext` of the same type. This will cause `@Autowired` to not work because there isn't exactly one bean, or one marked as primary. To work around this, Spring Cloud Netflix marks all Feign instances as `@Primary`, so Spring Framework will know which bean to inject. In some cases, this may not be desirable. To turn off this behavior set the `primary` attribute of `@FeignClient` to false.

```
@FeignClient(name = "hello", primary = false)
public interface HelloClient {
        // methods here
}
```

1.7 Feign Inheritance Support

Feign supports boilerplate apis via single-inheritance interfaces. This allows grouping common operations into convenient base interfaces.

UserService.java.

```
public interface UserService {

    @RequestMapping(method = RequestMethod.GET, value ="/users/{id}")
    User getUser(@PathVariable("id") long id);
}
```

UserResource.java.

```
@RestController
public class UserResource implements UserService {

}
```

UserClient.java.

```
package project.user;

@FeignClient("users")
public interface UserClient extends UserService {

}
```

It is generally not advisable to share an interface between a server and a client. It introduces tight coupling, and also actually doesn't work with Spring MVC in its current form (method parameter mapping is not inherited).

1.8 Feign request/response compression

You may consider enabling the request or response GZIP compression for your Feign requests. You can do this by enabling one of the properties:

```
feign.compression.request.enabled=true
feign.compression.response.enabled=true
```

Feign request compression gives you settings similar to what you may set for your web server:

```
feign.compression.request.enabled=true
feign.compression.request.mime-types=text/xml,application/xml,application/json
feign.compression.request.min-request-size=2048
```

These properties allow you to be selective about the compressed media types and minimum request threshold length.

1.9 Feign logging

A logger is created for each Feign client created. By default the name of the logger is the full class name of the interface used to create the Feign client. Feign logging only responds to the `DEBUG` level.

application.yml.

```
logging.level.project.user.UserClient: DEBUG
```

The `Logger.Level` object that you may configure per client, tells Feign how much to log. Choices are:

- `NONE`, No logging (**DEFAULT**).
- `BASIC`, Log only the request method and URL and the response status code and execution time.
- `HEADERS`, Log the basic information along with request and response headers.
- `FULL`, Log the headers, body, and metadata for both requests and responses.

For example, the following would set the `Logger.Level` to `FULL`:

```
@Configuration
public class FooConfiguration {
    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}
```