Spring Cloud OpenFeign Features

Spring Cloud OpenFeign Features

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 de-

fault in Spring Web. Spring Cloud integrates Eureka, Spring Cloud CircuitBreaker, as well as Spring Cloud LoadBalancer to provide a load-balanced http client when using Feign.

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

```java
@SpringBootApplication
@EnableFeignClients
public class Application {

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

}
```

StoreClient.java

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

    @GetMapping("/stores")
    Page<Store> getStores(Pageable pageable);

    @PostMapping(value = "/stores/{storeId}", consumes = "application/json",
                params = "mode=upsert")
    Store update(@PathVariable("storeId") Long storeId, Store store);

    @DeleteMapping("/stores/{storeId:\\d+}")
    void delete(@PathVariable Long storeId);
}
```

In the `@FeignClient` annotation the String value ("stores" above) is an arbitrary client name, which is used to create a Spring Cloud LoadBalancer client. 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 `qualifiers` value of the `@FeignClient` annotation.

The load-balancer 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 configure a list of servers in your external configuration using `SimpleDiscoveryClient`.

Spring Cloud OpenFeign supports all the features available for the blocking mode of Spring Cloud LoadBalancer. You can read more about them in the project documentation.

TIP

To use `@EnableFeignClients` annotation on `@Configuration`-annotated-classes, make sure to specify where the clients are located, for example: `@EnableFeignClients(basePackages = "com.example.clients")` or list them explicitly: `@EnableFeignClients(clients = InventoryServiceFeignClient.class)`.

WARNING

Since `FactoryBean` objects may be instantiated before the initial context refresh should take place, and the instantiation of Spring Cloud OpenFeign Clients triggers a context refresh, they should not be declared within `FactoryBean` classes.

Attribute resolution mode

While creating `Feign` client beans, we resolve the values passed via the `@FeignClient` annotation. As of `4.x`, the values are being resolved eagerly. This is a good solution for most use-cases, and it also allows for AOT support.

If you need the attributes to be resolved lazily, set the `spring.cloud.openfeign.lazy-attributes-resolution` property value to `true`.

TIP

For Spring Cloud Contract test integration, lazy attribute resolution should be used.

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`. It is possible to override the name of that ensemble by using the `contextId` attribute of the `@FeignClient` annotation.

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

JAVA

```
@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).

NOTE

`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`.

NOTE

Using `contextId` attribute of the `@FeignClient` annotation in addition to changing the name of the `ApplicationContext` ensemble, it will override the alias of the client name and it will be used as part of the name of the configuration bean created for that client.

WARNING

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.

JAVA

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

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

- `Decoder` feignDecoder: `ResponseEntityDecoder` (which wraps a `SpringDecoder`)
- `Encoder` feignEncoder: `SpringEncoder`
- `Logger` feignLogger: `Slf4jLogger`
- `MicrometerObservationCapability` micrometerObservationCapability: If `feign-micrometer` is on the classpath and `ObservationRegistry` is available
- `MicrometerCapability` micrometerCapability: If `feign-micrometer` is on the classpath, `MeterRegistry` is available and `ObservationRegistry` is not available
- `CachingCapability` cachingCapability: If `@EnableCaching` annotation is used. Can be disabled via `spring.cloud.openfeign.cache.enabled`.
- `Contract` feignContract: `SpringMvcContract`

- `Feign.Builder` feignBuilder: `FeignCircuitBreaker.Builder`
- `Client` feignClient: If Spring Cloud LoadBalancer is on the classpath, `FeignBlockingLoadBalancerClient` is used. If none of them is on the classpath, the default feign client is used.

NOTE

`spring-cloud-starter-openfeign` supports `spring-cloud-starter-loadbalancer`. However, as is an optional dependency, you need to make sure it has been added to your project if you want to use it.

To use OkHttpClient-backed Feign clients and Http2Client Feign clients, make sure that the client you want to use is on the classpath and set `spring.cloud.openfeign.okhttp.enabled` or `spring.cloud.openfeign.http2client.enabled` to `true` respectively.

When it comes to the Apache HttpClient 5-backed Feign clients, it's enough to ensure HttpClient 5 is on the classpath, but you can still disable its use for Feign Clients by setting `spring.cloud.openfeign.httpclient.hc5.enabled` to `false`. You can customize the HTTP client used by providing a bean of either `org.apache.hc.client5.http.impl.classic.CloseableHttpClient` when using Apache HC5.

You can further customise http clients by setting values in the `spring.cloud.openfeign.httpclient.xxx` properties. The ones prefixed just with `httpclient` will work for all the clients, the ones prefixed with `httpclient.hc5` to Apache HttpClient 5, the ones prefixed with `httpclient.okhttp` to OkHttpClient and the ones prefixed with `httpclient.http2` to Http2Client. You can find a full list of properties you can customise in the appendix. If you can not configure Apache HttpClient 5 by using properties, there is an `HttpClientBuilderCustomizer` interface for programmatic configuration.

TIP

Starting with Spring Cloud OpenFeign 4, the Feign Apache HttpClient 4 is no longer supported. We suggest using Apache HttpClient 5 instead.

Spring Cloud OpenFeign *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`
- `QueryMapEncoder`

- `Capability` (`MicrometerObservationCapability` and `CachingCapability` are provided by default)

A bean of `Retryer.NEVER_RETRY` with the type `Retryer` is created by default, which will disable retrying. Notice this retrying behavior is different from the Feign default one, where it will automatically retry IOExceptions, treating them as transient network related exceptions, and any RetryableException thrown from an ErrorDecoder.

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:

JAVA

```
@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

YAML

```
spring:
    cloud:
        openfeign:
            client:
                config:
                    feignName:
                        url: http://remote-service.com
                        connectTimeout: 5000
                        readTimeout: 5000
                        loggerLevel: full
                        errorDecoder: com.example.SimpleErrorDecoder
                        retryer: com.example.SimpleRetryer
                        defaultQueryParameters:
                            query: queryValue
                        defaultRequestHeaders:
                            header: headerValue
                        requestInterceptors:
```

```
                - com.example.FooRequestInterceptor
                - com.example.BarRequestInterceptor
            responseInterceptor: com.example.BazResponseInterceptor
            dismiss404: false
            encoder: com.example.SimpleEncoder
            decoder: com.example.SimpleDecoder
            contract: com.example.SimpleContract
            capabilities:
                - com.example.FooCapability
                - com.example.BarCapability
            queryMapEncoder: com.example.SimpleQueryMapEncoder
            micrometer.enabled: false
```

`feignName` in this example refers to `@FeignClient value`, that is also aliased with `@FeignClient name` and `@FeignClient contextId`. In a load-balanced scenario, it also corresponds to the `serviceId` of the server app that will be used to retrieve the instances. The specified classes for decoders, retryer and other ones must have a bean in the Spring context or have a default constructor.

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 configure all `@FeignClient`, you can create configuration properties with `default` feign name.

You can use `spring.cloud.openfeign.client.config.feignName.defaultQueryParameters` and `spring.cloud.openfeign.client.config.feignName.defaultRequestHeaders` to specify query parameters and headers that will be sent with every request of the client named `feignName`.

application.yml

YAML

```
spring:
    cloud:
        openfeign:
            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 `spring.cloud.openfeign.client.default-to-properties` to `false`.

If we want to create multiple feign clients with the same name or url so that they would point to the same server but each with a different custom configuration then we have to use `contextId` attribute of the `@FeignClient` in order to avoid name collision of these configuration beans.

```java
@FeignClient(contextId = "fooClient", name = "stores", configuration =
FooConfiguration.class)
public interface FooClient {
    //..
}
```

```java
@FeignClient(contextId = "barClient", name = "stores", configuration =
BarConfiguration.class)
public interface BarClient {
    //..
}
```

It is also possible to configure FeignClient not to inherit beans from the parent context. You can do this by overriding the `inheritParentConfiguration()` in a `FeignClientConfigurer` bean to return `false`:

```java
@Configuration
public class CustomConfiguration {
    @Bean
    public FeignClientConfigurer feignClientConfigurer() {
        return new FeignClientConfigurer() {
            @Override
            public boolean inheritParentConfiguration() {
                return false;
            }
        };
    }
}
```

TIP

By default, Feign clients do not encode slash `/` characters. You can change this behaviour, by setting the value of `spring.cloud.openfeign.client.decodeSlash` to `false`.

`SpringEncoder` configuration

In the `SpringEncoder` that we provide, we set `null` charset for binary content types and `UTF-8` for all the other ones.

You can modify this behaviour to derive the charset from the `Content-Type` header charset instead by setting the value of `spring.cloud.openfeign.encoder.charset-from-content-type` to `true`.

Timeout Handling

We can configure timeouts on both the default and the named client. OpenFeign works with two timeout parameters:

- `connectTimeout` prevents blocking the caller due to the long server processing time.
- `readTimeout` is applied from the time of connection establishment and is triggered when returning the response takes too long.

NOTE

In case the server is not running or available a packet results in *connection refused*. The communication ends either with an error message or in a fallback. This can happen *before* the `connectTimeout` if it is set very low. The time taken to perform a lookup and to receive such a packet causes a significant part of this delay. It is subject to change based on the remote host that involves a DNS lookup.

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.

JAVA

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

    private FooClient fooClient;

    private FooClient adminClient;

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

        this.adminClient = Feign.builder().client(client)
                .encoder(encoder)
                .decoder(decoder)
                .contract(contract)
                .addCapability(micrometerObservationCapability)
                .requestInterceptor(new BasicAuthRequestInterceptor("admin", "admin"))
                .target(FooClient.class, "https://PROD-SVC");
```

```
    }
}
```

NOTE

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

NOTE

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

NOTE

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.

You can also use the `Builder`to configure FeignClient not to inherit beans from the parent context. You can do this by overriding calling `inheritParentContext(false)` on the `Builder` .

Feign Spring Cloud CircuitBreaker Support

If Spring Cloud CircuitBreaker is on the classpath and `spring.cloud.openfeign.circuitbreaker.enabled=true` , Feign will wrap all methods with a circuit breaker.

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

JAVA

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

The circuit breaker name follows this pattern `<feignClientClassName>#<calledMethod>(<parameterTypes>)` . When calling a `@FeignClient` with `FooClient` interface and the called interface method that has no parameters is `bar` then the circuit breaker name will be `FooClient#bar()` .

NOTE

NOTE

As of 2020.0.2, the circuit breaker name pattern has changed from `<feignClientName>_<calledMethod>`. Using `CircuitBreakerNameResolver` introduced in 2020.0.4, circuit breaker names can retain the old pattern.

Providing a bean of `CircuitBreakerNameResolver`, you can change the circuit breaker name pattern.

JAVA

```
@Configuration
public class FooConfiguration {
    @Bean
    public CircuitBreakerNameResolver circuitBreakerNameResolver() {
        return (String feignClientName, Target<?> target, Method method) ->
feignClientName + "_" + method.getName();
    }
}
```

To enable Spring Cloud CircuitBreaker group set the `spring.cloud.openfeign.circuitbreaker.group.enabled` property to `true` (by default `false`).

Configuring CircuitBreakers With Configuration Properties

You can configure CircuitBreakers via configuration properties.

For example, if you had this Feign client

JAVA

```
@FeignClient(url = "http://localhost:8080")
public interface DemoClient {

    @GetMapping("demo")
    String getDemo();
}
```

You could configure it using configuration properties by doing the following

YAML

```
spring:
  cloud:
    openfeign:
      circuitbreaker:
        enabled: true
        alphanumeric-ids:
          enabled: true
resilience4j:
  circuitbreaker:
    instances:
      DemoClientgetDemo:
        minimumNumberOfCalls: 69
```

```
  timelimiter:
    instances:
      DemoClientgetDemo:
        timeoutDuration: 10s
```

NOTE

If you want to switch back to the circuit breaker names used prior to Spring Cloud 2022.0.0 you can set `spring.cloud.openfeign.circuitbreaker.alphanumeric-ids.enabled` to `false`.

Feign Spring Cloud CircuitBreaker Fallbacks

Spring Cloud CircuitBreaker supports the notion of a fallback: a default code path that is executed when the 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.

JAVA

```java
@FeignClient(name = "test", url = "http://localhost:${server.port}/", fallback =
Fallback.class)
protected interface TestClient {

    @GetMapping("/hello")
    Hello getHello();

    @GetMapping("/hellonotfound")
    String getException();

}

@Component
static class Fallback implements TestClient {

    @Override
    public Hello getHello() {
        throw new NoFallbackAvailableException("Boom!", new RuntimeException());
    }

    @Override
    public String getException() {
        return "Fixed response";
    }

}
```

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

```java
@FeignClient(name = "testClientWithFactory", url = "http://localhost:${server.port}/",
            fallbackFactory = TestFallbackFactory.class)
protected interface TestClientWithFactory {

    @GetMapping("/hello")
    Hello getHello();

    @GetMapping("/hellonotfound")
    String getException();

}

@Component
static class TestFallbackFactory implements FallbackFactory<FallbackWithFactory> {

    @Override
    public FallbackWithFactory create(Throwable cause) {
        return new FallbackWithFactory();
    }

}

static class FallbackWithFactory implements TestClientWithFactory {

    @Override
    public Hello getHello() {
        throw new NoFallbackAvailableException("Boom!", new RuntimeException());
    }

    @Override
    public String getException() {
        return "Fixed response";
    }

}
```

Feign and `@Primary`

When using Feign with Spring Cloud CircuitBreaker 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 OpenFeign 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.

```java
@FeignClient(name = "hello", primary = false)
public interface HelloClient {
```

```
    // methods here
}
```

Feign Inheritance Support

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

UserService.java

```java
public interface UserService {

    @GetMapping("/users/{id}")
    User getUser(@PathVariable("id") long id);
}
```

UserResource.java

```java
@RestController
public class UserResource implements UserService {

}
```

UserClient.java

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

}
```

WARNING

`@FeignClient` interfaces should not be shared between server and client and annotating `@FeignClient` interfaces with `@RequestMapping` on class level is no longer supported.

[[feign-request/response-compression]] === 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:

```java
spring.cloud.openfeign.compression.request.enabled=true
spring.cloud.openfeign.compression.response.enabled=true
```

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

```java
spring.cloud.openfeign.compression.request.enabled=true
spring.cloud.openfeign.compression.request.mime-
```

```
types=text/xml,application/xml,application/json
spring.cloud.openfeign.compression.request.min-request-size=2048
```

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

TIP

Since the OkHttpClient uses "transparent" compression, that is disabled if the `content-encoding` or `accept-encoding` header is present, we do not enable compression when `feign.okhttp.OkHttpClient` is present on the classpath and `spring.cloud.openfeign.okhttp.enabled` is set to `true`.

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

YAML

```
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`:

JAVA

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

Feign Capability support

The Feign capabilities expose core Feign components so that these components can be modified. For example, the capabilities can take the `Client`, *decorate* it, and give the decorated instance back to Feign. The support for Micrometer is a good real-life example for this. See Micrometer Support.

Creating one or more `Capability` beans and placing them in a `@FeignClient` configuration lets you register them and modify the behavior of the involved client.

```java
@Configuration
public class FooConfiguration {
    @Bean
    Capability customCapability() {
        return new CustomCapability();
    }
}
```

Micrometer Support

If all of the following conditions are true, a `MicrometerObservationCapability` bean is created and registered so that your Feign client is observable by Micrometer:

- `feign-micrometer` is on the classpath
- A `ObservationRegistry` bean is available
- feign micrometer properties are set to `true` (by default)
  - `spring.cloud.openfeign.micrometer.enabled=true` (for all clients)
  - `spring.cloud.openfeign.client.config.feignName.micrometer.enabled=true` (for a single client)

NOTE

If your application already uses Micrometer, enabling this feature is as simple as putting `feign-micrometer` onto your classpath.

You can also disable the feature by either:

- excluding `feign-micrometer` from your classpath
- setting one of the feign micrometer properties to `false`
  - `spring.cloud.openfeign.micrometer.enabled=false`
  - `spring.cloud.openfeign.client.config.feignName.micrometer.enabled=false`

NOTE

`spring.cloud.openfeign.micrometer.enabled=false` disables Micrometer support for **all** Feign clients regardless of the value of the client-level flags: `spring.cloud.openfeign.client.config.feignName.micrometer.enabled`. If you want to enable or disable Micrometer support per client, don't set `spring.cloud.openfeign.micrometer.enabled` and use `spring.cloud.openfeign.client.config.feignName.micrometer.enabled`.

You can also customize the `MicrometerObservationCapability` by registering your own bean:

```java
@Configuration
public class FooConfiguration {
    @Bean
    public MicrometerObservationCapability
micrometerObservationCapability(ObservationRegistry registry) {
        return new MicrometerObservationCapability(registry);
    }
}
```

It is still possible to use `MicrometerCapability` with Feign (metrics-only support), you need to disable Micrometer support (`spring.cloud.openfeign.micrometer.enabled=false`) and create a `MicrometerCapability` bean:

```java
@Configuration
public class FooConfiguration {
    @Bean
    public MicrometerCapability micrometerCapability(MeterRegistry meterRegistry) {
        return new MicrometerCapability(meterRegistry);
    }
}
```

Feign Caching

If `@EnableCaching` annotation is used, a `CachingCapability` bean is created and registered so that your Feign client recognizes `@Cache*` annotations on its interface:

```java
public interface DemoClient {

    @GetMapping("/demo/{filterParam}")
    @Cacheable(cacheNames = "demo-cache", key = "#keyParam")
    String demoEndpoint(String keyParam, @PathVariable String filterParam);
}
```

You can also disable the feature via property `spring.cloud.openfeign.cache.enabled=false` .

Spring @RequestMapping Support

Spring Cloud OpenFeign provides support for the Spring `@RequestMapping` annotation and its derived annotations (such as `@GetMapping` , `@PostMapping` , and others) support. The attributes on the `@RequestMapping` annotation (including `value` , `method` , `params` , `headers` , `consumes` , and `produces`) are parsed by `SpringMvcContract` as the content of the request.

Consider the following example:

Define an interface using the `params` attribute.

```java
@FeignClient("demo")
public interface DemoTemplate {

        @PostMapping(value = "/stores/{storeId}", params = "mode=upsert")
        Store update(@PathVariable("storeId") Long storeId, Store store);
}
```

In the above example, the request url is resolved to `/stores/storeId?mode=upsert`.
The params attribute also supports the use of multiple `key=value` or only one `key`:

- When `params = { "key1=v1", "key2=v2" }`, the request url is parsed as `/stores/storeId?key1=v1&key2=v2`.
- When `params = "key"`, the request url is parsed as `/stores/storeId?key`.

Feign @QueryMap support

Spring Cloud OpenFeign provides an equivalent `@SpringQueryMap` annotation, which is used to annotate a POJO or Map parameter as a query parameter map.

For example, the `Params` class defines parameters `param1` and `param2`:

```java
// Params.java
public class Params {
    private String param1;
    private String param2;

    // [Getters and setters omitted for brevity]
}
```

The following feign client uses the `Params` class by using the `@SpringQueryMap` annotation:

```java
@FeignClient("demo")
public interface DemoTemplate {

    @GetMapping(path = "/demo")
    String demoEndpoint(@SpringQueryMap Params params);
}
```

If you need more control over the generated query parameter map, you can implement a custom `QueryMapEncoder` bean.

HATEOAS support

Spring provides some APIs to create REST representations that follow the HATEOAS principle, Spring Hateoas and Spring Data REST.

If your project use the `org.springframework.boot:spring-boot-starter-hateoas` starter or the `org.springframework.boot:spring-boot-starter-data-rest` starter, Feign HATEOAS support is enabled by default.

When HATEOAS support is enabled, Feign clients are allowed to serialize and deserialize HATEOAS representation models: EntityModel, CollectionModel and PagedModel.

```java
@FeignClient("demo")
public interface DemoTemplate {

    @GetMapping(path = "/stores")
    CollectionModel<Store> getStores();
}
```

Spring @MatrixVariable Support

Spring Cloud OpenFeign provides support for the Spring `@MatrixVariable` annotation.

If a map is passed as the method argument, the `@MatrixVariable` path segment is created by joining key-value pairs from the map with a `=`.

If a different object is passed, either the `name` provided in the `@MatrixVariable` annotation (if defined) or the annotated variable name is joined with the provided method argument using `=`.

IMPORTANT

> Even though, on the server side, Spring does not require the users to name the path segment placeholder same as the matrix variable name, since it would be too ambiguous on the client side, Spring Cloud OpenFeign requires that you add a path segment placeholder with a name matching either the `name` provided in the `@MatrixVariable` annotation (if defined) or the annotated variable name.

For example:

```java
@GetMapping("/objects/links/{matrixVars}")
Map<String, List<String>> getObjects(@MatrixVariable Map<String, List<String>>
matrixVars);
```

Note that both variable name and the path segment placeholder are called `matrixVars`.

```java
@FeignClient("demo")
public interface DemoTemplate {

    @GetMapping(path = "/stores")
```

```
    CollectionModel<Store> getStores();
}
```

Feign `CollectionFormat` support

We support `feign.CollectionFormat` by providing the `@CollectionFormat` annotation. You can annotate a Feign client method (or the whole class to affect all methods) with it by passing the desired `feign.CollectionFormat` as annotation value.

In the following example, the `CSV` format is used instead of the default `EXPLODED` to process the method.

JAVA

```
@FeignClient(name = "demo")
protected interface DemoFeignClient {

    @CollectionFormat(feign.CollectionFormat.CSV)
    @GetMapping(path = "/test")
    ResponseEntity performRequest(String test);

}
```

Reactive Support

As the OpenFeign project does not currently support reactive clients, such as Spring WebClient, neither does Spring Cloud OpenFeign.

Since Spring Cloud OpenFeign project is now considered feature-complete, we're not planning on adding support even if it becomes available in the upstream project. We suggest migrating over to Spring Interface Clients instead. Both blocking and reactive stacks are supported there.

Until that is done, we recommend using feign-reactive for Spring WebClient support.

Early Initialization Errors

We discourage using Feign clients in the early stages of application lifecycle, while processing configurations and initialising beans. Using the clients during bean initialisation is not supported.

Similarly, depending on how you are using your Feign clients, you may see initialization errors when starting your application. To work around this problem you can use an `ObjectProvider` when autowiring your client.

JAVA

```
@Autowired
ObjectProvider<TestFeignClient> testFeignClient;
```

Spring Data Support

If Jackson Databind and Spring Data Commons are on the classpath, converters for `org.springframework.data.domain.Page` and `org.springframework.data.domain.Sort` will be added automatically.

To disable this behaviour set

```
spring.cloud.openfeign.autoconfiguration.jackson.enabled=false
```
JAVA

See `org.springframework.cloud.openfeign.FeignAutoConfiguration.FeignJacksonConfiguration` for details.

Spring `@RefreshScope` Support

If Feign client refresh is enabled, each Feign client is created with:

- `feign.Request.Options` as a refresh-scoped bean. This means properties such as `connectTimeout` and `readTimeout` can be refreshed against any Feign client instance.
- A url wrapped under `org.springframework.cloud.openfeign.RefreshableUrl`. This means the URL of Feign client, if defined with `spring.cloud.openfeign.client.config.{feignName}.url` property, can be refreshed against any Feign client instance.

You can refresh these properties through `POST /actuator/refresh`.

By default, refresh behavior in Feign clients is disabled. Use the following property to enable refresh behavior:

```
spring.cloud.openfeign.client.refresh-enabled=true
```
JAVA

TIP

DO NOT annotate the `@FeignClient` interface with the `@RefreshScope` annotation.

OAuth2 Support

OAuth2 support can be enabled by adding the `spring-boot-starter-oauth2-client` dependency to your project and setting following flag:

```
spring.cloud.openfeign.oauth2.enabled=true
```

When the flag is set to true, and the oauth2 client context resource details are present, a bean of class `OAuth2AccessTokenInterceptor` is created. Before each request, the interceptor resolves the

required access token and includes it as a header. `OAuth2AccessTokenInterceptor` uses the `OAuth2AuthorizedClientManager` to get `OAuth2AuthorizedClient` that holds an `OAuth2AccessToken` . If the user has specified an OAuth2 `clientRegistrationId` using the `spring.cloud.openfeign.oauth2.clientRegistrationId` property, it will be used to retrieve the token. If the token is not retrieved or the `clientRegistrationId` has not been specified, the `serviceId` retrieved from the `url` host segment will be used.

TIP

Using the `serviceId` as OAuth2 client registrationId is convenient for load-balanced Feign clients. For non-load-balanced ones, the property-based `clientRegistrationId` is a suitable approach.

TIP

If you do not want to use the default setup for the `OAuth2AuthorizedClientManager` , you can just instantiate a bean of this type in your configuration.

Transform the load-balanced HTTP request

You can use the selected `ServiceInstance` to transform the load-balanced HTTP Request.

For `Request` , you need to implement and define `LoadBalancerFeignRequestTransformer` , as follows:

JAVA

```
@Bean
public LoadBalancerFeignRequestTransformer transformer() {
    return new LoadBalancerFeignRequestTransformer() {

        @Override
        public Request transformRequest(Request request, ServiceInstance instance) {
            Map<String, Collection<String>> headers = new HashMap<>(request.headers());
            headers.put("X-ServiceId",
Collections.singletonList(instance.getServiceId()));
            headers.put("X-InstanceId",
Collections.singletonList(instance.getInstanceId()));
            return Request.create(request.httpMethod(), request.url(), headers,
request.body(), request.charset(),
                    request.requestTemplate());
        }
    };
}
```

If multiple transformers are defined, they are applied in the order in which beans are defined. Alternatively, you can use `LoadBalancerFeignRequestTransformer.DEFAULT_ORDER` to specify the order.

X-Forwarded Headers Support

`X-Forwarded-Host` and `X-Forwarded-Proto` support can be enabled by setting following flag:

```
spring.cloud.loadbalancer.x-forwarded.enabled=true
```
PROPERTIES

Supported Ways To Provide URL To A Feign Client

You can provide a URL to a Feign client in any of the following ways:

Case	Example	De
The URL is provided in the `@FeignClient` annotation.	`@FeignClient(name="testClient", url="http://localhost:8081")`	Th of
The URL is provided in the `@FeignClient` annotation and in the configuration properties.	`@FeignClient(name="testClient", url="http://localhost:8081")` and the property defined in `application.yml` as `spring.cloud.openfeign.client.config.testClient.url=http://localhost:8081`	Th of Th ert
The URL is not provided in the `@FeignClient` annotation but is provided in configuration properties.	`@FeignClient(name="testClient")` and the property defined in `application.yml` as `spring.cloud.openfeign.client.config.testClient.url=http://localhost:8081`	Th ert sp en fig scr
The URL is neither provided in the `@FeignClient` annotation nor in configuration properties.	`@FeignClient(name="testClient")`	Th an

AOT and Native Image Support

Spring Cloud OpenFeign supports Spring AOT transformations and native images, however, only with refresh mode disabled, Feign clients refresh disabled (default setting) and lazy `@FeignClient` attribute resolution disabled (default setting).

WARNING

If you want to run Spring Cloud OpenFeign clients in AOT or native image modes, make sure to set `spring.cloud.refresh.enabled` to `false`.

TIP

TIP

If you want to run Spring Cloud OpenFeign clients in AOT or native image modes, ensure `spring.cloud.openfeign.client.refresh-enabled` has not been set to `true`.

TIP

If you want to run Spring Cloud OpenFeign clients in AOT or native image modes, ensure `spring.cloud.openfeign.lazy-attributes-resolution` has not been set to `true`.

TIP

However, if you set the `url` value via properties, it is possible to override the `@FeignClient url` value by running the image with `-Dspring.cloud.openfeign.client.config.[clientId].url=[url]` flag. In order to enable overriding, a `url` value also has to be set via properties and not `@FeignClient` attribute during buildtime.

Configuration properties

To see the list of all Spring Cloud OpenFeign related configuration properties please check the Appendix page.