# Spring Cloud Series – The Gateway Pattern

Last modified: July 13, 2022

Written by: Tim Schimandle

Architecture | Spring Cloud

Pattern | Spring Cloud Gateway



## 1. Overview

So far, in our cloud application, we've used the Gateway Pattern to support two main features.

First, we insulated our clients from each service, eliminating the need for cross-origin support. Next, we implemented locating inst

In this article, we are going to look at how to use the Gateway pattern to **retrieve data from multiple services with a single reque**

To read up on how to use the Feign client check out this article.

Spring Cloud now also provides the Spring Cloud Gateway project which implements this pattern.

## 2. Setup

Let's open up the *pom.xml* of our *gateway* server and add the dependency for Feign:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

For reference – we can find the latest versions on *Maven Central* (spring-cloud-starter-feign).

Now that we have the support for building a Feign client, let's enable it in the *GatewayApplication.java*:

```
@EnableFeignClients
public class GatewayApplication { ... }
```

Now let's set up Feign clients for the book and rating services.

## 3. Feign Clients

### 3.1. Book Client

Let's create a new interface called *BooksClient.java*:

```
@FeignClient("book-service")
public interface BooksClient {

    @RequestMapping(value = "/books/{bookId}", method = RequestMethod.GET)
    Book getBookById(@PathVariable("bookId") Long bookId);
}
```

With this interface, we're instructing Spring to create a Feign client that will access the "*/books/{bookId}*" endpoint. When called, t

To make this work we need to add a *Book.java* DTO:

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Book {

    private Long id;
    private String author;
    private String title;
    private List<Rating> ratings;

    // getters and setters
}
```

Let's move on to the *RatingsClient*.

### 3.2. Ratings Client

Let's create an interface called *RatingsClient*:

```
@FeignClient("rating-service")
public interface RatingsClient {

    @RequestMapping(value = "/ratings", method = RequestMethod.GET)
    List<Rating> getRatingsByBookId(
      @RequestParam("bookId") Long bookId,
      @RequestHeader("Cookie") String session);

}
```

Like with the *BookClient*, the method exposed here will make a rest call to our rating service and return the list of ratings for a boo

However, this endpoint is secured. **To be able to access this endpoint properly we need to pass the user's session to the reques**

We do this using the *@RequestHeader* annotation. This will instruct Feign to write the value of that variable to the request's heade

In our case, we are writing to the *Cookie* header because Spring Session will be looking for our session in a cookie.

Finally, let's add a *Rating.java* DTO:

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Rating {
    private Long id;
    private Long bookId;
    private int stars;
}
```

Now, both clients are complete. Let's put them to use!

## 4. Combined Request

One common use case for the Gateway pattern is to have endpoints that encapsulate commonly called services. This can increas

To do this let's create a controller and call it *CombinedController.java*:

```
@RestController
@RequestMapping("/combined")
public class CombinedController { ... }
```

Next, let's wire in our newly created feign clients:

```
private BooksClient booksClient;
private RatingsClient ratingsClient;

@Autowired
public CombinedController(
  BooksClient booksClient,
  RatingsClient ratingsClient) {

    this.booksClient = booksClient;
    this.ratingsClient = ratingsClient;
}
```

And finally let's create a GET request that combines these two endpoints and returns a single book with its ratings loaded:

```
@GetMapping
public Book getCombinedResponse(
  @RequestParam Long bookId,
  @CookieValue("SESSION") String session) {

    Book book = booksClient.getBookById(bookId);
    List<Rating> ratings = ratingsClient.getRatingsByBookId(bookId, "SESSION="+session);
    book.setRatings(ratings);
    return book;
}
```

Notice that we are setting the session value using the *@CookieValue* annotation that extracts it from the request.

There it is! We have a combined endpoint in our gateway that reduces network calls between the client and the system!

## 5. Testing

Let's make sure our new endpoint is working.

Navigate to *LiveTest.java* and let's add a test for our combined endpoint:

```
@Test
public void accessCombinedEndpoint() {
    Response response = RestAssured.given()
      .auth()
      .form("user", "password", formConfig)
      .get(ROOT_URI + "/combined?bookId=1");

    assertEquals(HttpStatus.OK.value(), response.getStatusCode());
    assertNotNull(response.getBody());

    Book result = response.as(Book.class);

    assertEquals(new Long(1), result.getId());
    assertNotNull(result.getRatings());
    assertTrue(result.getRatings().size() > 0);
}
```

Start up Redis, and then run each service in our application: *config, discovery, zipkin, gateway*, *book*, and the *rating* service.

Once everything is up, run the new test to confirm it is working.

## 6. Conclusion

We've seen how to integrate Feign into our gateway to build a specialized endpoint. We can leverage this information to build any
Using the Gateway pattern we can set up our gateway service to each client's needs uniquely. This creates decoupling giving our
As always, code snippets can be found over on GitHub.



Comments are closed on this article!