Spring Declarative HTTP Client using @HttpExchange

Description

Introduction

In this blog post we will look at how to implement Declarative HTTP Client using @HttpExchange. This feature was introduced in Spring Framework release 6 and Spring Boot version 3. If anyone of you have come across or implemented **FeignClient**, then as you go along the post, you will notice how similar it is to define declarative HTTP services using Java interfaces.

In the first step we will see how to define HTTP interface and then we will check the exchange method annotations. After that we will see how to define the method parameters and their return values. I'll try to cover as much as possible so that you are comfortable using this feature.

Maven Dependencies required

Spring declarative HTTP interface functionality is part of the **spring-boot-starter-web** dependency. In order to add the reactive support we need to include the second dependency i.e. **spring-boot-starter-webflux** dependency.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Creating an Http Service Interface

The main purpose of Declarative HTTP interface is to reduce the boilerplate code. It is basically a Java interface that generates a proxy implementing this interface and does the exchanges at the framework level.

In Spring, Http service interface is a Java interface with @HttpExchange methods. These methods are annotated and they are treated as HTTP endpoint. Every Http method has an equivalent exchange annotation. Let's see the working and usage of each annotation present in the below example code.

```
import com.springmicroservices.model.Student;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.service.annotation.DeleteExchange;
import org.springframework.web.service.annotation.GetExchange;
import org.springframework.web.service.annotation.HttpExchange;
import org.springframework.web.service.annotation.PostExchange;
import org.springframework.web.service.annotation.PutExchange;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@HttpExchange(url = "/students", accept = "application/json", contentType = "a
public interface StudentClient {

  @GetExchange("/")
  Flux<Student> getAll();

  @GetExchange("/{id}")
  Mono<Student> getById(@PathVariable("id") Long id);

  @PostExchange("/")
  Mono<ResponseEntity<Void>> save(@RequestBody Student student);

  @PutExchange("/{id}")
  Mono<ResponseEntity<Void>> update(@PathVariable Long id, @RequestBody Studen

  @DeleteExchange("/{id}")
  Mono<ResponseEntity<Void>> delete(@PathVariable Long id);
}
```


Http Exchange annotations

- **@HttpExchange** annotation is given at the interface level and it applies to all the methods. We can also attach an endpoint to this annotation as seen in the url property.
- **@GetExchange** annotation is used for HTTP GET requests.
- **@PostExchange** annotation is used for HTTP POST requests.
- **@PutExchange** annotation is used for HTTP PUT requests.
- **@DeleteExchange** annotation is used for HTTP DELETE requests.
- **@PatchExchange** annotation is used for HTTP PATCH requests.

Method Arguments

- **@RequestBody** provides the body of the request.
- **@PathVariable** is used to replace the placeholder with a value in the uri.
- **@RequestParam** is used to add the request parameters which are added as URL query parameters.
- **@RequestHeader** is used to add request header names and values.
- **@CookieValue** is used to add the cookies to the request.

- **@RequestPart** is used to add a request part (form field, resource orÂ HttpEntityÂ etc).

Return Types

The HTTP exchange method return types can be of below types

- blocking
- reactive i.e. Mono/Flux
- It does not return anything i.e. void.
- HTTP status code and/or response headers

```
//Reactive way
@GetExchange("/{id}")
Mono<Student> getById(@PathVariable("id") Long id);

//Blocking way
@GetExchange("/{id}")
User getById(@PathVariable("id") Long id);
```


Creating HttpServiceProxyFactory

HttpServiceProxyFactory is a factory that is used to create client proxy from the HTTP service interface. In this class we set the remote API's base URL in the WebClient bean. The relative paths will be present in the exchange methods.

```
import com.fasterxml.jackson.databind.ObjectMapper;
import com.springmicroservices.model.client.StudentClient;
import lombok.SneakyThrows;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.client.WebClient;
import org.springframework.web.reactive.function.client.support.WebClientAdapt
import org.springframework.web.service.invoker.HttpServiceProxyFactory;

@Configuration
public class WebConfig {

  @Bean
  WebClient webClient(ObjectMapper objectMapper) {
    return WebClient.builder()
        .baseUrl("http://localhost:8901/")
        .build();
  }

  @SneakyThrows
  @Bean
  StudentClient postClient(WebClient webClient) {
```

```
    HttpServiceProxyFactory httpServiceProxyFactory =
        HttpServiceProxyFactory.builder(WebClientAdapter.forClient(webClient))
            .build();
    return httpServiceProxyFactory.createClient(StudentClient.class);
  }
}
```

Invoking the remote service

We are now ready to invoke our remote service which we were waiting for. Final step remaining is to inject the StudentClient bean into the Service layer or Integrator layer from wherever we want to invoke.

```
@Autowired
StudentClient studentClient;

//Get All Students
studentClient.getAll().subscribe(
    data -> log.info("All Students: {}", data)
);

//Get Student By Id
studentClient.getById(1L).subscribe(
    data -> log.info("Student by studentId: {}", data)
);

//Create Student
studentClient.save(new Student(1, "Ravi", "Kada", "ravi.kada@xyz.com"))
    .subscribe(
        data -> log.info("Student created: {}", data)
    );


//Delete By Student Id
studentClient.delete(1L).subscribe(
    data -> log.info("Student deleted : {}", data)
);
```


Conclusion

In this tutorial we learnt how to create and use declarative HTTP client interface with exchange methods in the interface. We also learnt how to invoke them using proxy implementation created by Spring framework. Please feel free to comment down below if you have any suggestions or concerns.

Category

1. Spring Boot

Date Created
July 6, 2023
Author

kk-ravi144gmail-com