Clientes HTTP de Spring

Para responder a la consulta “cómo usar FeignClient, o Spring RestClient, o Spring WebClient,”explicaré cómo cada una de estas herramientas de Spring puede usarse para realizar solicitudes HTTP, centrándome en su configuración y uso básico. Todas forman parte del ecosistema de Spring para interactuar con servicios RESTful, pero sirven a diferentes propósitos dependiendo de las necesidades de tu aplicación. A continuación, proporcionaré una guía clara y paso a paso para cada una, junto con ejemplos simples.

---

1. Usando FeignClient

FeignClient es un cliente REST declarativo proporcionado por Spring Cloud. Permite definir clientes HTTP como interfaces con anotaciones, lo que lo hace particularmente útil en arquitecturas de microservicios donde necesitas llamar a otros servicios.

Pasos para usar FeignClient

1. **Agregar Dependencias**: Incluye las dependencias de Spring Cloud en tu proyecto. Si usas Maven, agrega el starter de Spring Cloud para Feign a tu `pom.xml`:

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

   Asegúrate de tener también un bloque de gestión de dependencias para Spring Cloud, especificando una versión compatible.

2. **Habilitar Feign Clients**: Anota tu clase de aplicación principal o una clase de configuración con `@EnableFeignClients` para activar el soporte de Feign:

   ```
   import org.springframework.boot.SpringApplication;
   import org.springframework.boot.autoconfigure.SpringBootApplication;
   import org.springframework.cloud.openfeign.EnableFeignClients;

   @SpringBootApplication
   @EnableFeignClients
   public class MyApplication {
       public static void main(String[] args) {
           SpringApplication.run(MyApplication.class, args);
       }
   }
   ```

3. **Definir la Interfaz FeignClient**: Crea una interfaz anotada con `@FeignClient`, especificando el nombre del servicio o la URL, y define métodos correspondientes a los puntos finales REST:

```
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import java.util.List;

@FeignClient(name = "user-service", url = "http://localhost:8080")
public interface UserClient {
    @GetMapping("/users")
    List<User> getUsers();
}
```

   Aquí, `name` es un nombre lógico para el cliente, y `url` es la URL base del servicio objetivo. La anotación `@GetMapping` se mapea al punto final `/users`.

4. **Inyectar y Usar el Cliente**: Autoinyecta la interfaz en tu servicio o controlador y llama a sus métodos como si fueran locales:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class UserService {
    @Autowired
    private UserClient userClient;

    public List<User> fetchUsers() {
        return userClient.getUsers();
    }
}
```

Puntos Clave

- FeignClient es síncrono por defecto.
- Es ideal para microservicios con descubrimiento de servicios (por ejemplo, Eureka) cuando omites la `url` y dejas que Spring Cloud la resuelva.
- El manejo de errores se puede agregar con fallbacks o circuit breakers (por ejemplo, Hystrix o Resilience4j).

2. Usando Spring RestClient

Spring RestClient es un cliente HTTP síncrono introducido en Spring Framework 6.1 como una alternativa moderna al `RestTemplate` obsoleto. Proporciona una API fluida para construir y ejecutar solicitudes.

Pasos para usar RestClient

1. **Dependencias**: RestClient está incluido en `spring-web`, que es parte de `spring-boot-starter-web` de Spring Boot. No se necesitan dependencias adicionales:

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

2. **Crear una Instancia de RestClient**: Instancia `RestClient` usando su método estático `create()` o personalízalo con un constructor:

```
import org.springframework.web.client.RestClient;

RestClient restClient = RestClient.create();
```

   Para configuraciones personalizadas (por ejemplo, timeouts), usa `RestClient.builder()`.

3. **Construir y Ejecutar una Solicitud**: Usa la API fluida para especificar el método HTTP, URI, encabezados y cuerpo, luego recupera la respuesta:

```
import org.springframework.http.MediaType;
import org.springframework.web.client.RestClient;
import java.util.List;

public class UserService {
    private final RestClient restClient;

    public UserService() {
        this.restClient = RestClient.create();
    }

    public List<User> fetchUsers() {
        return restClient.get()
            .uri("http://localhost:8080/users")
            .accept(MediaType.APPLICATION_JSON)
            .retrieve()
```

```
            .body(new ParameterizedTypeReference<List<User>>() {});
    }
}
```

- `get()` especifica el método HTTP.
- `uri()` establece el punto final.
- `accept()` establece el tipo de contenido esperado.
- `retrieve()` ejecuta la solicitud, y `body()` extrae la respuesta, usando `ParameterizedTypeReference` para tipos genéricos como listas.

4. **Manejar la Respuesta**: La respuesta se devuelve directamente ya que RestClient es síncrono. Para más control (por ejemplo, códigos de estado), usa `toEntity()`:

```
import org.springframework.http.ResponseEntity;

ResponseEntity<List<User>> response = restClient.get()
    .uri("http://localhost:8080/users")
    .accept(MediaType.APPLICATION_JSON)
    .retrieve()
    .toEntity(new ParameterizedTypeReference<List<User>>() {});
List<User> users = response.getBody();
```

Puntos Clave

- RestClient es síncrono, lo que lo hace adecuado para aplicaciones tradicionales y bloqueantes.
- Lanza excepciones (por ejemplo, `RestClientException`) en errores HTTP, que puedes capturar y manejar.
- Es un reemplazo para `RestTemplate` con una API más intuitiva.

---

3. Usando Spring WebClient

Spring WebClient es un cliente HTTP reactivo y no bloqueante introducido en Spring WebFlux. Está diseñado para operaciones asíncronas e integra con flujos reactivos (Mono y Flux).

Pasos para usar WebClient

1. **Agregar Dependencias**: Incluye la dependencia de WebFlux en tu proyecto:

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

2. **Crear una Instancia de WebClient**: Instancia `WebClient` con una URL base o configuraciones predeterminadas:

```
import org.springframework.web.reactive.function.client.WebClient;

WebClient webClient = WebClient.create("http://localhost:8080");
```

   Usa `WebClient.builder()` para configuraciones personalizadas (por ejemplo, codecs, filtros).

3. **Construir y Ejecutar una Solicitud**: Usa la API fluida para construir la solicitud y recuperar una respuesta reactiva:

```
import org.springframework.http.MediaType;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;
import java.util.List;

public class UserService {
    private final WebClient webClient;

    public UserService(WebClient webClient) {
        this.webClient = webClient;
    }

    public Mono<List<User>> fetchUsers() {
        return webClient.get()
            .uri("/users")
            .accept(MediaType.APPLICATION_JSON)
            .retrieve()
            .bodyToFlux(User.class)
            .collectList();
    }
}
```

   - `bodyToFlux(User.class)` maneja un flujo de objetos `User`.
   - `collectList()` convierte el `Flux<User>` a un `Mono<List<User>>`.

4. **Suscribirse a la Respuesta**: Dado que WebClient es reactivo, debes suscribirte al `Mono` o `Flux` para desencadenar la solicitud:

```
Mono<List<User>> usersMono = fetchUsers();
usersMono.subscribe(users -> System.out.println(users));
```

   Alternativamente, encadénalo en una tubería reactiva o bloquea (no recomendado en contextos reactivos):

```
List<User> users = fetchUsers().block();
```

Puntos Clave

- WebClient es no bloqueante e ideal para aplicaciones reactivas construidas con Spring WebFlux.
- Usa `Mono` para respuestas de un solo valor y `Flux` para respuestas de múltiples valores.
- El manejo de errores se puede hacer con operadores como `onErrorResume()` o `retry()`.

Comparación y Cuándo Usar Cada Uno

- **FeignClient**: Usa para clientes declarativos basados en interfaces en configuraciones de microservicios. Es síncrono y simplifica las llamadas de servicio a servicio.
- **RestClient**: Usa para solicitudes HTTP síncronas en aplicaciones tradicionales y bloqueantes de Spring. Es una alternativa moderna a `RestTemplate`.
- **WebClient**: Usa para solicitudes HTTP reactivas y no bloqueantes en aplicaciones basadas en WebFlux o cuando manejas alta concurrencia.

Ejemplo: Llamar a un Punto Final POST

Supongamos que necesitas crear un usuario llamando a `POST /users` con un objeto `User`.

FeignClient

```
@FeignClient(name = "user-service", url = "http://localhost:8080")
public interface UserClient {
    @PostMapping("/users")
    User createUser(@RequestBody User user);
}

@Service
public class UserService {
    @Autowired
    private UserClient userClient;

    public User createUser(User user) {
        return userClient.createUser(user);
    }
}
```

RestClient

```
@Service
public class UserService {
    private final RestClient restClient = RestClient.create();

    public User createUser(User user) {
        return restClient.post()
            .uri("http://localhost:8080/users")
            .contentType(MediaType.APPLICATION_JSON)
            .body(user)
            .retrieve()
            .body(User.class);
    }
}
```

WebClient

```
@Service
public class UserService {
    private final WebClient webClient;

    public UserService(WebClient webClient) {
        this.webClient = WebClient.create("http://localhost:8080");
    }

    public Mono<User> createUser(User user) {
        return webClient.post()
            .uri("/users")
            .contentType(MediaType.APPLICATION_JSON)
            .bodyValue(user)
            .retrieve()
            .bodyToMono(User.class);
    }
}
```

Conclusión

- **FeignClient**: Define una interfaz, habilítala y úsala como un bean local.

- **RestClient**: Construye solicitudes de manera fluida y obtén respuestas de manera síncrona.
- **WebClient**: Construye solicitudes de manera fluida, recupera flujos reactivos y procésalos de manera asíncrona.

Elige según la arquitectura de tu aplicación: FeignClient para microservicios, RestClient para aplicaciones bloqueantes y WebClient para aplicaciones reactivas. Cada uno sigue un patrón de configuración y uso sencillo como se muestra arriba.