Clients HTTP de Printemps

Pour répondre à la question “comment utiliser FeignClient, ou Spring RestClient, ou Spring WebClient,”j’ expliquerai comment utiliser chacun de ces outils Spring pour effectuer des requêtes HTTP, en me concentrant sur leur configuration et leur utilisation de base. Tous font partie de l’écosystème Spring pour interagir avec des services REST, mais ils servent des objectifs différents en fonction des besoins de votre application. Ci-dessous, je fournirai un guide étape par étape clair pour chacun, accompagné d’exemples simples.

---

1. Utilisation de FeignClient

FeignClient est un client REST déclaratif fourni par Spring Cloud. Il permet de définir des clients HTTP sous forme d’interfaces avec des annotations, ce qui le rend particulièrement utile dans les architectures de microservices où vous devez appeler d’autres services.

Étapes pour utiliser FeignClient

1. **Ajouter les dépendances** : Incluez les dépendances Spring Cloud dans votre projet. Si vous utilisez Maven, ajoutez le démarreur Spring Cloud pour Feign à votre `pom.xml` :

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

   Assurez-vous également d’avoir un bloc de gestion des dépendances pour Spring Cloud, en spécifiant une version compatible.

2. **Activer les clients Feign** : Annotez votre classe principale de l’application ou une classe de configuration avec `@EnableFeignClients` pour activer le support 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. **Définir l'interface FeignClient** : Créez une interface annotée avec `@FeignClient`, en spécifiant le nom du service ou l'URL, et définissez les méthodes correspondant aux points de terminaison REST :

```java
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();
}
```

   Ici, `name` est un nom logique pour le client, et `url` est l'URL de base du service cible. L'annotation `@GetMapping` est mappée au point de terminaison `/users`.

4. **Injecter et utiliser le client** : Autowirez l'interface dans votre service ou contrôleur et appelez ses méthodes comme si elles étaient locales :

```java
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();
    }
}
```

Points clés

- FeignClient est synchrone par défaut.
- Il est idéal pour les microservices avec découverte de services (par exemple, Eureka) lorsque vous omettez l'URL et laissez Spring Cloud la résoudre.
- La gestion des erreurs peut être ajoutée avec des retours en arrière ou des disjoncteurs (par exemple, Hystrix ou Resilience4j).

2. Utilisation de Spring RestClient

Spring RestClient est un client HTTP synchrone introduit dans Spring Framework 6.1 comme une alternative moderne à l'obsolète `RestTemplate`. Il fournit une API fluide pour construire et exécuter des requêtes.

Étapes pour utiliser RestClient

1. **Dépendances** : RestClient est inclus dans `spring-web`, qui fait partie de Spring Boot's `spring-boot-starter-web`. Aucune dépendance supplémentaire n'est généralement nécessaire :

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

2. **Créer une instance de RestClient** : Instanciez `RestClient` en utilisant sa méthode statique `create()` ou personnalisez-la avec un constructeur :

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

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

   Pour des configurations personnalisées (par exemple, des délais d'attente), utilisez `RestClient.builder()`.

3. **Construire et exécuter une requête** : Utilisez l'API fluide pour spécifier la méthode HTTP, l'URI, les en-têtes et le corps, puis récupérez la réponse :

```
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()` spécifie la méthode HTTP.
- `uri()` définit le point de terminaison.
- `accept()` définit le type de contenu attendu.
- `retrieve()` exécute la requête, et `body()` extrait la réponse, en utilisant `ParameterizedTypeReference` pour les types génériques comme les listes.

4. **Gérer la réponse** : La réponse est retournée directement puisque RestClient est synchrone. Pour plus de contrôle (par exemple, les codes de statut), utilisez `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();
```

Points clés

- RestClient est synchrone, ce qui le rend adapté aux applications traditionnelles, bloquantes.
- Il lance des exceptions (par exemple, `RestClientException`) en cas d'erreurs HTTP, que vous pouvez attraper et gérer.
- C'est un remplacement pour `RestTemplate` avec une API plus intuitive.

---

3. Utilisation de Spring WebClient

Spring WebClient est un client HTTP réactif, non bloquant, introduit dans Spring WebFlux. Il est conçu pour des opérations asynchrones et s'intègre avec les flux réactifs (Mono et Flux).

Étapes pour utiliser WebClient

1. **Ajouter les dépendances** : Incluez la dépendance WebFlux dans votre projet :

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

2. **Créer une instance de WebClient** : Instanciez `WebClient` avec une URL de base ou des paramètres par défaut :

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

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

   Utilisez `WebClient.builder()` pour des configurations personnalisées (par exemple, codecs, filtres).

3. **Construire et exécuter une requête** : Utilisez l'API fluide pour construire la requête et récupérer une réponse réactive :

```
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)` gère un flux d'objets `User`.
   - `collectList()` convertit le `Flux<User>` en un `Mono<List<User>>`.

4. **S'abonner à la réponse** : Puisque WebClient est réactif, vous devez vous abonner au `Mono` ou `Flux` pour déclencher la requête :

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

Alternativement, vous pouvez le chaîner dans un pipeline réactif ou le bloquer (non recommandé dans les contextes réactifs) :

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

Points clés

- WebClient est non bloquant et idéal pour les applications réactives construites avec Spring WebFlux.
- Utilisez `Mono` pour les réponses à valeur unique et `Flux` pour les réponses à valeurs multiples.
- La gestion des erreurs peut être effectuée avec des opérateurs comme `onErrorResume()` ou `retry()`.

Comparaison et quand utiliser chacun

- **FeignClient** : Utilisez pour les clients déclaratifs basés sur des interfaces dans les configurations de microservices. Il est synchrone et simplifie les appels de service à service.
- **RestClient** : Utilisez pour les requêtes HTTP synchrones dans les applications Spring traditionnelles, bloquantes. C'est une alternative moderne à `RestTemplate`.
- **WebClient** : Utilisez pour les requêtes HTTP réactives, non bloquantes dans les applications WebFlux ou lorsque vous gérez une haute concurrence.

Exemple : Appel à un point de terminaison POST

Supposons que vous devez créer un utilisateur en appelant `POST /users` avec un objet `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);
    }
}
```

Conclusion

- **FeignClient** : Définissez une interface, activez-la et utilisez-la comme un bean local.
- **RestClient** : Construisez des requêtes de manière fluide et obtenez des réponses de manière synchrone.
- **WebClient** : Construisez des requêtes de manière fluide, récupérez des flux réactifs et traitez-les de manière asynchrone.

Choisissez en fonction de l'architecture de votre application : FeignClient pour les microservices, RestClient pour les applications bloquantes et WebClient pour les applications réactives. Chacun suit un modèle de configuration et d'utilisation simple comme montré ci-dessus.