Spring HTTP-Clients

Um die Anfrage “wie man FeignClient, Spring RestClient oder Spring WebClient verwendet”zu beantworten, werde ich erklären, wie jedes dieser Spring-Tools verwendet werden kann, um HTTP-Anfragen zu senden, wobei der Fokus auf der Einrichtung und der grundlegenden Verwendung liegt. Diese sind alle Teil des Spring-Ökosystems zur Interaktion mit RESTful-Diensten, aber sie erfüllen unterschiedliche Zwecke je nach den Anforderungen Ihrer Anwendung. Im Folgenden werde ich eine klare, schrittweise Anleitung für jedes Tool mit einfachen Beispielen bereitstellen.

---

1. Verwendung von FeignClient

FeignClient ist ein deklarativer REST-Client, der von Spring Cloud bereitgestellt wird. Er ermöglicht es Ihnen, HTTP-Clients als Schnittstellen mit Annotationen zu definieren, was ihn besonders nützlich in Mikrodienstarchitekturen macht, in denen Sie andere Dienste aufrufen müssen.

Schritte zur Verwendung von FeignClient

1. **Abhängigkeiten hinzufügen**: Fügen Sie die Spring Cloud-Abhängigkeiten zu Ihrem Projekt hinzu. Wenn Sie Maven verwenden, fügen Sie den Spring Cloud Starter für Feign zu Ihrer `pom.xml` hinzu:

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

   Stellen Sie sicher, dass Sie auch einen Abhängigkeitsverwaltungsblock für Spring Cloud haben, der eine kompatible Version angibt.

2. **Feign Clients aktivieren**: Kennzeichnen Sie Ihre Hauptanwendungsklasse oder eine Konfigurationsklasse mit `@EnableFeignClients`, um die Feign-Unterstützung zu aktivieren:

   ```
   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. **Die FeignClient-Schnittstelle definieren**: Erstellen Sie eine Schnittstelle, die mit `@FeignClient` gekennzeichnet ist, und geben Sie den Dienstnamen oder die URL an, und definieren Sie Methoden, die den REST-Endpunkten entsprechen:

```
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();
}
```

   Hier ist `name` ein logischer Name für den Client, und `url` ist die Basis-URL des Zieldienstes. Die `@GetMapping`-Annotation wird auf den `/users`-Endpunkt abgebildet.

4. **Client injizieren und verwenden**: Autowire die Schnittstelle in Ihrem Service oder Controller und rufen Sie deren Methoden auf, als wären sie lokal:

```
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();
    }
}
```

Wichtige Punkte

- FeignClient ist standardmäßig synchron.
- Es ist ideal für Mikrodienste mit Service-Discovery (z.B. Eureka), wenn Sie die `url` weglassen und Spring Cloud die Auflösung übernimmt.

- Fehlerbehandlung kann mit Fallbacks oder Schaltkreisen (z.B. Hystrix oder Resilience4j) hinzugefügt werden.

---

2. Verwendung von Spring RestClient

Spring RestClient ist ein synchroner HTTP-Client, der in Spring Framework 6.1 als modernes Pendant zum veralteten `RestTemplate` eingeführt wurde. Er bietet eine flüssige API zum Erstellen und Ausführen von Anfragen.

Schritte zur Verwendung von RestClient

1. **Abhängigkeiten**: RestClient ist in `spring-web` enthalten, das Teil von Spring Boots `spring-boot-starter-web` ist. Normalerweise sind keine zusätzlichen Abhängigkeiten erforderlich:

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

2. **Erstellen Sie eine RestClient-Instanz**: Instanziieren Sie `RestClient` mit seiner statischen `create()`-Methode oder passen Sie sie mit einem Builder an:

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

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

   Für benutzerdefinierte Konfigurationen (z.B. Timeouts) verwenden Sie `RestClient.builder()`.

3. **Erstellen und Ausführen einer Anfrage**: Verwenden Sie die flüssige API, um die HTTP-Methode, URI, Header und den Body festzulegen, und holen Sie die Antwort ab:

```
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()` gibt die HTTP-Methode an.
- `uri()` setzt den Endpunkt.
- `accept()` setzt den erwarteten Inhalts-Typ.
- `retrieve()` führt die Anfrage aus, und `body()` extrahiert die Antwort, wobei `ParameterizedTypeReference` für generische Typen wie Listen verwendet wird.

4. **Antwort verarbeiten**: Die Antwort wird direkt zurückgegeben, da RestClient synchron ist. Für mehr Kontrolle (z.B. Statuscodes) verwenden Sie `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();
```

Wichtige Punkte

- RestClient ist synchron, was es für traditionelle, blockierende Anwendungen geeignet macht.
- Es wirft Ausnahmen (z.B. `RestClientException`) bei HTTP-Fehlern, die Sie fangen und verarbeiten können.
- Es ist ein Ersatz für `RestTemplate` mit einer intuitiveren API.

---

3. Verwendung von Spring WebClient

Spring WebClient ist ein reaktiver, nicht blockierender HTTP-Client, der in Spring WebFlux eingeführt wurde. Er ist für asynchrone Operationen entwickelt und integriert sich in reaktive Streams (Mono und Flux).

Schritte zur Verwendung von WebClient

1. **Abhängigkeiten hinzufügen**: Fügen Sie die WebFlux-Abhängigkeit zu Ihrem Projekt hinzu:

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

2. **Erstellen Sie eine WebClient-Instanz**: Instanziieren Sie `WebClient` mit einer Basis-URL oder Standard-Einstellungen:

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

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

   Verwenden Sie `WebClient.builder()` für benutzerdefinierte Konfigurationen (z.B. Codecs, Filter).

3. **Erstellen und Ausführen einer Anfrage**: Verwenden Sie die flüssige API, um die Anfrage zu konstruieren und eine reaktive Antwort abzurufen:

```
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)` verarbeitet einen Stream von `User`-Objekten.

- `collectList()` konvertiert den `Flux<User>` in ein `Mono<List<User>>`.

4. **Antwort abonnieren**: Da WebClient reaktiv ist, müssen Sie das `Mono` oder `Flux` abonnieren, um die Anfrage auszulösen:

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

Alternativ können Sie es in einer reaktiven Pipeline verketten oder blockieren (in reaktiven Kontexten nicht empfohlen):

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

Wichtige Punkte

- WebClient ist nicht blockierend und ideal für reaktive Anwendungen, die mit Spring WebFlux erstellt wurden.
- Verwenden Sie `Mono` für einwertige Antworten und `Flux` für mehrwertige Antworten.
- Fehlerbehandlung kann mit Operatoren wie `onErrorResume()` oder `retry()` erfolgen.

Vergleich und wann man jeden verwenden sollte

- **FeignClient**: Verwenden Sie für deklarative, schnittstellenbasierte Clients in Mikrodienst-Setups. Es ist synchron und vereinfacht Dienst-zu-Dienst-Aufrufe.
- **RestClient**: Verwenden Sie für synchrone HTTP-Anfragen in traditionellen, blockierenden Spring-Anwendungen. Es ist ein modernes Pendant zu `RestTemplate`.
- **WebClient**: Verwenden Sie für reaktive, nicht blockierende HTTP-Anfragen in WebFlux-basierten Anwendungen oder bei der Handhabung hoher Konkurrenz.

Beispiel: Aufruf eines POST-Endpunkts

Angenommen, Sie müssen einen Benutzer erstellen, indem Sie `POST /users` mit einem `User`-Objekt aufrufen.

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);
    }
}
```

Schlussfolgerung

- **FeignClient**: Definieren Sie eine Schnittstelle, aktivieren Sie sie und verwenden Sie sie wie ein lokales Bean.
- **RestClient**: Bauen Sie Anfragen flüssig und erhalten Sie Antworten synchron.
- **WebClient**: Bauen Sie Anfragen flüssig, holen Sie sich reaktive Streams und verarbeiten Sie sie asynchron.

Wählen Sie basierend auf der Architektur Ihrer Anwendung: FeignClient für Mikrodienste, RestClient für blockierende Anwendungen und WebClient für reaktive Anwendungen. Jeder folgt einem einfachen Einrichtungs- und Verwendungsmuster, wie oben gezeigt.