स्प्रिंग एचटीटीपी क्लाइंट्स

“प्रश्न ‘□□□□□□□□□□□□, या □□□□□□□ □□□□□□□□□□□□, या □□□□□□□ □□□□□□□□□□ कैसे उपयोग करें?’का जवाब देने के लिए, मैं प्रत्येक □□□□□□□ उपकरण को □□□□□ अनुरोध बनाने के लिए कैसे उपयोग किया जा सकता है, उनके सेटअप और मूल उपयोग पर ध्यान केंद्रित करूँगा। ये सभी □□□□□□□ इकोसिस्टम के हिस्से हैं जो □□□□□□□□ सेवाओं के साथ बातचीत करने के लिए हैं, लेकिन वे आपके एप्लिकेशन की जरूरतों के अनुसार अलग-अलग उद्देश्यों को पूरा करते हैं। नीचे, मैं प्रत्येक के लिए एक स्पष्ट, कदम-दर-कदम मार्गदर्शन प्रदान करूँगा, साथ ही सरल उदाहरण भी दूँगा।

---

1. □□□□□□□□□□□□ का उपयोग

□□□□□□□□□□□□ एक घोषणात्मक □□□□□ क्लाइंट है जो □□□□□□□ □□□□□□ द्वारा प्रदान किया गया है। यह आपको अननोटेशन के साथ इंटरफेस के रूप में □□□□□ क्लाइंट परिभाषित करने की अनुमति देता है, जिससे यह विशेष रूप से माइक्रोसर्विस आर्किटेक्चर में उपयोगी होता है जहां आपको अन्य सेवाओं को कॉल करना होता है।

□□□□□□□□□□□□ का उपयोग करने के कदम

1. □□□□□□□□□□□□□ **जोड़ें**: अपने प्रोजेक्ट में □□□□□□□ □□□□□□ □□□□□□□□□□□□□ जोड़ें। अगर आप □□□□□□ का उपयोग कर रहे हैं, तो अपने `pom.xml` में □□□□□□□ □□□□□□ □□□□□□□□ □□□□ □□□□□□ जोड़ें:

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

   सुनिश्चित करें कि आपके पास □□□□□□□ □□□□□□ के लिए एक संगत संस्करण को स्पष्ट करने वाला एक □□□□□□□□□□□□ □□□□□□□□□□□□ ब्लॉक भी है।

2. □□□□□□ **क्लाइंट्स को सक्रिय करें**: अपने मुख्य एप्लिकेशन क्लास या एक कॉन्फ़िगरेशन क्लास को `@EnableFeignClients` के साथ अननोटेट करें ताकि □□□□□□ समर्थन सक्रिय हो जाए:

   ```
   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. □□□□□□□□□□□□ **इंटरफेस परिभाषित करें**: एक इंटरफेस बनाएं जो `@FeignClient` के साथ अननोटेट किया गया है, सेवा नाम या □□□ को स्पष्ट करें, और □□□□ एंडपॉइंट्स के साथ मेल खाते हुए विधियों को परिभाषित करें:

```
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();
}
```

यहाँ, `name` क्लाइंट के लिए एक तार्किक नाम है, और `url` लक्ष्य सेवा का आधार □□□ है। `@GetMapping` अननोटेशन `/users` एंडपॉइंट के साथ मेल खाता है।

4. **क्लाइंट को इंजेक्ट और उपयोग करें**: इंटरफेस को अपने सेवा या कंट्रोलर में ऑटोवायर करें और उसके विधियों को जैसे ही वे स्थानीय हों, कॉल करें:

```
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();
    }
}
```

मुख्य बिंदु

- □ □□□□□□□□□□□□ डिफ़ॉल्ट में सिंक्रोनस होता है।
- □ यह माइक्रोसर्विसों के साथ सेवा खोज (जैसे □□□□□□□) के लिए आदर्श है जब आप `url` को छोड़ देते हैं और □□□□□□□ □□□□□□ को इसे हल करने देते हैं।
- □ त्रुटि हैंडलिंग को फॉलबैक या सर्किट ब्रेकर (जैसे □□□□□□□□ या □□□□□□□□□□□4□) के साथ जोड़ा जा सकता है।

---

2. [illegible] **का उपयोग**

[illegible] एक सिंक्रोनस [illegible] क्लाइंट है जो [illegible] 6.1 में `RestTemplate` के अप्रचलित विकल्प के रूप में पेश किया गया है। यह अनुरोध बनाना और उन्हें कार्यान्वित करने के लिए एक फ्लूएंट [illegible] प्रदान करता है।

का उपयोग करने के कदम

1. [illegible]: [illegible] `spring-web` में शामिल है, जो [illegible] के `spring-boot-starter-web` का हिस्सा है। आम तौर पर कोई अतिरिक्त [illegible] की आवश्यकता नहीं होती है:

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

2. [illegible] **इंस्टेंस बनाएं**: `RestClient` का इंस्टेंस `create()` विधि के साथ बनाएं या इसे एक बिल्डर के साथ कस्टमाइज़ करें:

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

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

कस्टम कॉन्फ़िगरेशन (जैसे टाइमआउट) के लिए `RestClient.builder()` का उपयोग करें।

3. **एक अनुरोध बनाएं और कार्यान्वित करें**: फ्लूएंट [illegible] का उपयोग करें [illegible] विधि, [illegible], हेडर और बॉडी को स्पष्ट करने के लिए, फिर जवाब प्राप्त करें:

```
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()` □□□□ विधि को स्पष्ट करता है।
- `uri()` एंडपॉइंट को सेट करता है।
- `accept()` अपेक्षित कंटेंट टाइप को सेट करता है।
- `retrieve()` अनुरोध को कार्यान्वित करता है, और `body()` जवाब को निकालता है, `ParameterizedTypeReference` के लिए जनरिक टाइप जैसे कि सूची का उपयोग करते हुए।

4. **उत्तर को हैंडल करें**: जवाब सीधे लौटाया जाता है क्योंकि □□□□□□□□□□ सिंक्रोनस होता है। अधिक नियंत्रण (जैसे स्टेटस कोड) के लिए `toEntity()` का उपयोग करें:

```
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();
```

मुख्य बिंदु

- □□□□□□□□□□ सिंक्रोनस होता है, जिससे यह पारंपरिक, ब्लॉकिंग एप्लिकेशन के लिए उपयुक्त होता है।
- यह □□□□ त्रुटियों पर अपवाद (जैसे `RestClientException`) फेंकता है, जिन्हें आप पकड़ सकते हैं और हैंडल कर सकते हैं।
- यह `RestTemplate` के साथ एक अधिक समझदारीपूर्ण □□□ के साथ एक विकल्प है।

---

3. □□□□□□ □□□□□□□□□ का उपयोग

□□□□□□ □□□□□□□□□ एक प्रतिक्रियात्मक, नॉन-ब्लॉकिंग □□□□ क्लाइंट है जो □□□□□□ □□□□□□□ में पेश किया गया है। यह असिंक्रोनस ऑपरेशंस के लिए डिज़ाइन किया गया है और प्रतिक्रियात्मक स्ट्रीम (□□□□ और □□□□) के साथ एकीकृत होता है।

□□□□□□□□□ **का उपयोग करने के कदम**

1. □□□□□□□□□□□□ **जोड़ें**: अपने प्रोजेक्ट में □□□□□□□ □□□□□□□□□ जोड़ें:

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

2. **□□□□□□□□□□ इंस्टेंस बनाएं**: एक आधार □□□ या डिफ़ॉल्ट सेटिंग्स के साथ `WebClient` का इंस्टेंस बनाएं:

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

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

कस्टम कॉन्फ़िगरेशन (जैसे कोडेक्स, फिल्टर) के लिए `WebClient.builder()` का उपयोग करें।

3. **एक अनुरोध बनाएं और कार्यान्वित करें**: फ्लूएंट □□□ का उपयोग करें अनुरोध को बनाना और एक प्रतिक्रियात्मक जवाब प्राप्त करना:

```
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)` एक `User` वस्तुओं की स्ट्रीम को हैंडल करता है।
   - `collectList()` `Flux<User>` को `Mono<List<User>>` में परिवर्तित करता है।

4. **उत्तर पर सब्सक्राइब करें**: क्योंकि □□□□□□□□□□ प्रतिक्रियात्मक है, आपको `Mono` या `Flux` पर सब्सक्राइब करना होगा ताकि अनुरोध को ट्रिगर किया जा सके:

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

या इसे एक प्रतिक्रियात्मक पाइपलाइन में चेन कर सकते हैं या ब्लॉक करें (प्रतिक्रियात्मक संदर्भों में अनुशंसित नहीं है):

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

मुख्य बिंदु

- □□□□□□□□□□ नॉन-ब्लॉकिंग और प्रतिक्रियात्मक एप्लिकेशन के लिए आदर्श है जो □□□□□□ □□□□□□□□ के साथ बनाए गए हैं या उच्च सम्पर्कता को हैंडल करने के लिए।
- एकमात्र मूल्य वाले जवाब के लिए `Mono` का उपयोग करें और बहु-मूल्य वाले जवाब के लिए `Flux` का उपयोग करें।
- त्रुटि हैंडलिंग को ऑपरेटर्स जैसे `onErrorResume()` या `retry()` के साथ किया जा सकता है।

तुलना और प्रत्येक का उपयोग

- □□□□□□□□□□□□: घोषणात्मक, इंटरफेस-आधारित क्लाइंटों के लिए माइक्रोसर्विस सेटअप में उपयोग करें। यह सिंक्रोनस है और सेवा-से-सेवा कॉल को सरल बनाता है।
- □□□□□□□□□□□: सिंक्रोनस □□□□ अनुरोधों के लिए पारंपरिक, ब्लॉकिंग □□□□□□ एप्लिकेशन में उपयोग करें। यह `RestTemplate` के साथ एक आधुनिक विकल्प है।
- □□□□□□□□□□: प्रतिक्रियात्मक, नॉन-ब्लॉकिंग □□□□ अनुरोधों के लिए □□□□□□□□-आधारित एप्लिकेशन या उच्च सम्पर्कता को हैंडल करने के लिए उपयोग करें।

उदाहरण: एक □□□□ एंडपॉइंट को कॉल करना

अगर आपको एक `User` वस्तु के साथ `POST /users` को कॉल करके एक उपयोगकर्ता बनाना है।

□□□□□□□□□□□□

```
@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);
    }
}
```

□□□□□□□□□□□□

```java
@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);
    }
}
```

□□□□□□□□□□□

```java
@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);
    }
}
```

निष्कर्ष

- □ □□□□□□□□□□□□□: एक इंटरफेस परिभाषित करें, इसे सक्रिय करें, और इसे एक स्थानीय बीन के रूप में उपयोग करें।

- □ □□□□□□□□□□□: अनुरोधों को फ्लूएंट रूप से बनाएं और जवाबों को सिंक्रोनस रूप से प्राप्त करें।
- □ □□□□□□□□□□: अनुरोधों को फ्लूएंट रूप से बनाएं, प्रतिक्रियात्मक स्ट्रीम प्राप्त करें, और उन्हें असिंक्रोनस रूप से प्रोसेस करें।

अपने एप्लिकेशन के आर्किटेक्चर के आधार पर चुनें: □□□□□□□□□□□□ के लिए माइक्रोसर्विस, □□□□□□□□□□□ के लिए ब्लॉकिंग एप्लिकेशन, और □□□-□□□□□□ के लिए प्रतिक्रियात्मक एप्लिकेशन। प्रत्येक एक सरल सेटअप और उपयोग पैटर्न का पालन करता है जैसा ऊपर दिखाया गया है।