Consuming REST Services

WebClient & WebFlux

What is WebClient?

- WebClient is a non-blocking, reactive HTTP client introduced in Spring 5.
- It is part of the Spring WebFlux module.
- Replaces the RestTemplate for asynchronous, reactive web requests.
- Can be created like this:

```
WebClient webClient = WebClient.builder()
        .baseUrl( baseUrl: "http://localhost:8080")
            .build();
```

What is Spring WebFlux?

- WebFlux is the reactive web framework in Spring 5+
- Built on Project Reactor, supporting Reactive Streams
- Enables asynchronous and non-blocking request handling

Spring WebFlux Key Concepts

- Mono<T>: 0 or 1 element
- Flux<T>: 0 to many elements
- Built to scale for I/O-intensive applications

WebClient vs RestTemplate

Feature	RestTemplate	WebClient
Blocking	Yes	No
Asynchronous	No	Yes
Reactive support	No	Yes (WebFlux)

Spring WebFlux vs. Spring MVC

- Spring MVC:
  - Imperative programming model
  - Based on the Servlet API and Servlet containers (e.g., Tomcat)
  - Handles one request per thread – thread is blocked while waiting
  - Good for traditional, blocking use cases
- Spring WebFlux:
  - Reactive programming model
  - Built on Project Reactor (Mono, Flux)
  - Uses non-blocking I/O (Netty, Servlet 3.1+ async)
  - Designed for high concurrency, scalable systems

When to Use WebClient & WebFlux

- Use WebClient/WebFlux when:
  - Building non-blocking REST APIs
  - Integrating with external APIs asynchronously
  - Needing high concurrency with low resources
- Not always needed for simple, blocking apps

WebClient vs RestClient vs RestTemplate

```
Customer customer = webClient.get()
        .uri( uri: "/customer/{id}",  ...uriVariables: id)
        .retrieve()
        .bodyToMono( elementClass: Customer.class).block();
```

Method

URI

Blocking only for demo purpose

Type of response

```
Customer customer = restClient.get()
        .uri( uri: "/customer/{id}",  ...uriVariables: id)
        .retrieve()
        .body( bodyType: Customer.class);
```

Method

URI

Type of response

```
Customer customer = restTemplate.getForObject( url: "http://localhost:8080/customer/" + id,  responseType: Customer.class);
```

Method

URI

Type of response

WebClient vs RestClient vs RestTemplate

```
webClient.post()                                    Method
        .uri( uri: "/customer")                     URI
        .bodyValue( body: customer)                 Object
        .retrieve()
        .bodyToMono( elementClass: Void.class)
        .block();                                   Blocking only for demo purpose
```

```
restClient.post()                                   Method
        .uri( uri: "/customer")                     URI
        .body( body: customer)                      Object
        .retrieve();
```

```
restTemplate.postForObject( url: "http://localhost:8080/customer", request: customer, responseType: Customer.class);
```

Method URI Object Type of response

Using .block() in WebClient - Demo Purposes Only

- Context:
  - We're currently building a Spring MVC application.
  - Calling an external REST API using WebClient.
  - This is a demo scenario to illustrate API communication.
- Why .block()?
  - .block() waits synchronously for the response.
  - Makes the reactive call behave in a blocking (imperative) way.
  - Suitable only in non-reactive (Spring MVC) contexts for simplicity.

HTTP methods

- WebClient methods for sending different HTTP methods:
- POST - to Create - webClient.post()
- GET - to Read - webClient.get()
- PUT - to Update - webClient.put()
- DELETE - to Delete - webClient.delete()

Important Reminder about blocking

- Don’t use .block() in production with Spring WebFlux.
- It defeats the purpose of non-blocking I/O.
- Here in this demo, it's acceptable only because Spring MVC is already blocking.

Demo/Exercise - WebClient

- Download and open the ConsumingRESTWebClientStarter project
- It's a working application. Try it out and look at the code and comments
- All calls to the DogRepository should be replaced by calls made by a WebClient to the REST Service created in the exercise for Creating REST Services
- A WebClient is already declared in the Controller class. Run the REST Service in another IntelliJ window on port 8080, this application will run on port 8081. Try it out when you have switched from repository to REST with the WebClient!