Spring Start Here

Chapter-11
Consuming REST Endpoints

Upcode Software
Engineer Team

-2024-

CONTENT

1. Introduction
2. Calling REST endpoint using Spring Cloud OpenFeign
3. Calling REST endpoints using RestTemplate
4. Calling REST endpoints using WebClient
5. Conclusion
6. Resource
7. Reference

1. Introduction (1/10)

- “REST Endpoints” refers to the process of making HTTP requests to external RESTful web services or APIs from within a Spring application.
- This process involves sending HTTP requests to specific endpoints (URLs) and handling the responses returned by these endpoints.

As an example, in the provided URL "**restfulservices/v1/users/{id}**" represents a REST endpoint.

1. Introduction (2/10)

Restful Endpoint Path

restfulservices - is likely the RESTful API's context path or base URL.

v1 - denotes the version of the API.

users - represents the resource or collection of users.

{id} - is a path parameter placeholder indicating that the endpoint expects a dynamic value to be substituted at runtime (e.g., the unique identifier of a user).

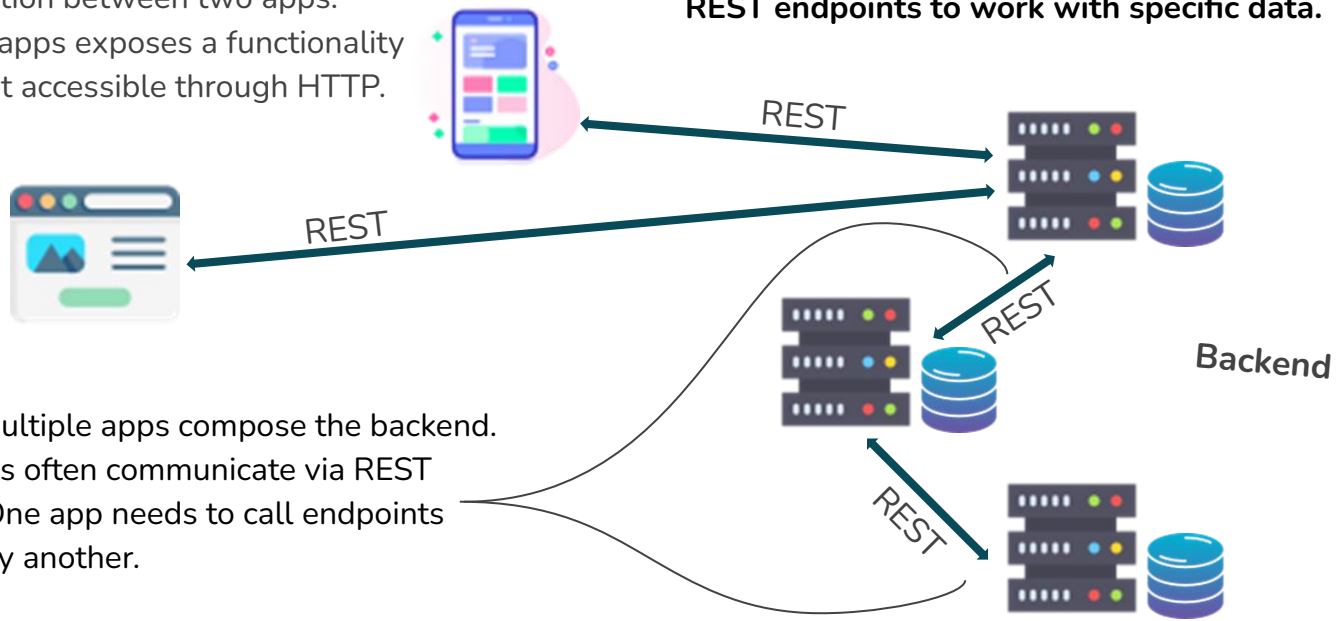
http://localhost:9999/restfulservices/v1/users/{id}

This endpoint is likely designed for retrieving, updating, or deleting user resources based on their unique identifiers. The specific actions supported by this endpoint (e.g., ***GET*** for retrieval, ***PUT/PATCH*** for update, ***DELETE*** for deletion) should be documented as part of the API specification or documentation.

1. Introduction (3/10)

● A REST endpoint is a way to implement communication between two apps.
One of the apps exposes a functionality by making it accessible through HTTP.

Often, a backend app needs to act as a client for another backend app, and calls exposed REST endpoints to work with specific data.



● Often, multiple apps compose the backend. These apps often communicate via REST services. One app needs to call endpoints exposed by another.

1. Introduction (4/10)

```java
@FeignClient(name = "book-service", url = "http://localhost:8080")
public interface BookClient {
    @GetMapping("/api/books")
    List<Book> getBooks();
}
```

1

	RestTemplate	WebClient	RestClient
From Spring Framework	3.0	5.0	6.1
Servlet Stack (synchronous)	✅	🔶	✅
Reactive Stack (asynchronous)	⛔	✅	⛔
Fluent & Functional API	⛔	✅	✅
Declarative HTTP Interface	✅	✅	✅
OTEL Support	✅	🔶	🔶

- **OpenFeign** - A tool provided by the Spring Cloud project. It provides a declarative way to interact with Restful web services using interfaces and annotations.

1. Introduction (5/10)

2

- **RestTemplate** - A widely used too since Spring 3 for invoking REST endpoints.
- **RestTemplate** is a synchronous HTTP client provided by the Spring Framework for making HTTP requests to external servers or RESTful services.
- It simplifies the process of consuming RESTful web services by providing a convenient API for performing various HTTP operations such as **GET, POST, PUT, DELETE**.

```
RestTemplate restTemplate = new RestTemplate();
List<Book> books =
  restTemplate.getForObject("http://localhost:8080/api/books",
                            List.class);
```

	RestTemplate	WebClient	RestClient
From Spring Framework	3.0	5.0	6.1
Servlet Stack (synchronous)	✅	🔶	✅
Reactive Stack (asynchronous)	⛔	✅	⛔
Fluent & Functional API	⛔	✅	✅
Declarative HTTP Interface	✅	✅	✅
OTEL Support	✅	🔶	🔶

1. Introduction (6/10)

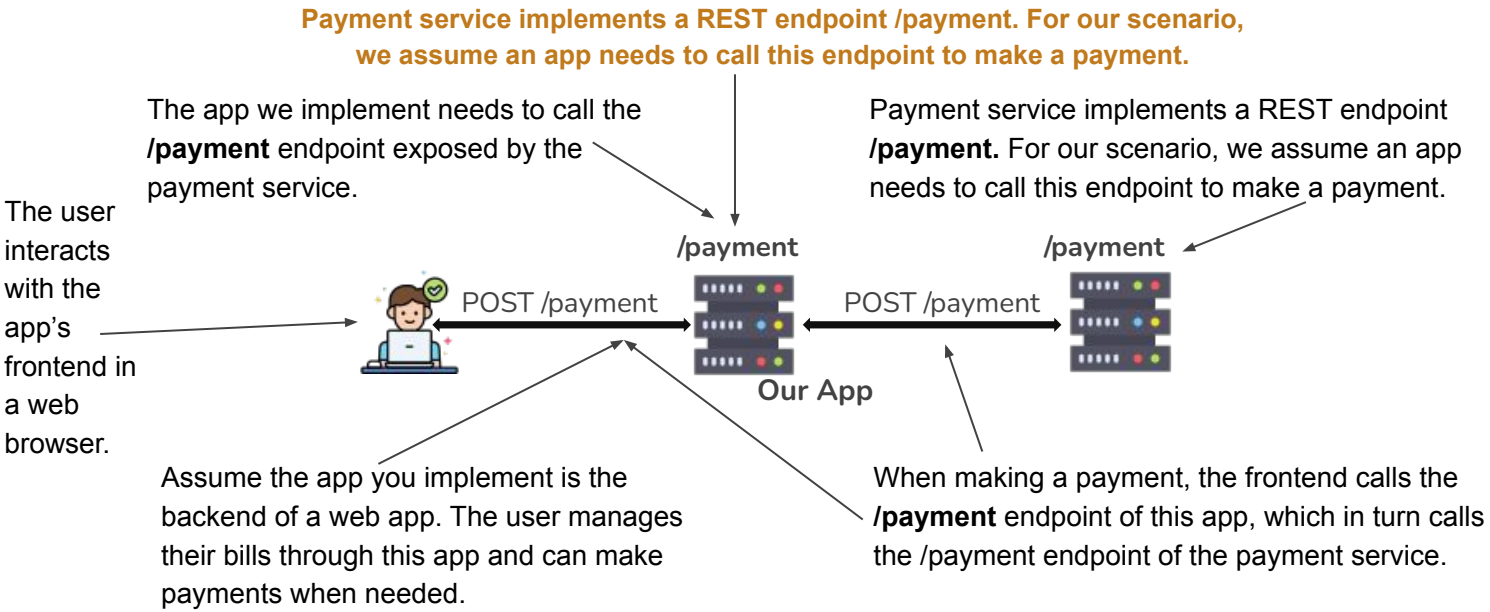
3

WebClient - A Spring feature introduced as an alternative to **RestTemplate**. Calling REST endpoints using **WebClient** involves utilizing the **WebClient** class, which is a part of the Spring Framework, to interact with RESTful services over HTTP.

- WebClient is a non-blocking, reactive HTTP client introduced in Spring 5. It allows you to asynchronously send HTTP requests to RESTful endpoints and process the responses in a reactive manner.

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

public class WebClientShortExample {

    public static void main(String[] args) {
        // Creating a WebClient instance
        WebClient webClient = WebClient.create("http://localhost:8080");

        // Performing a GET request and processing the response
        Mono<String> response = webClient.get()
            .uri("/api/books/1") // Specify the endpoint and path variable
            .retrieve() // Retrieve the response
            .bodyToMono(String.class); // Convert the response body to a
Mono<String>

        // Subscribe to the Mono to trigger the request and print the result
        response.subscribe(System.out::println);
    }
}
```

1. Introduction (7/10)

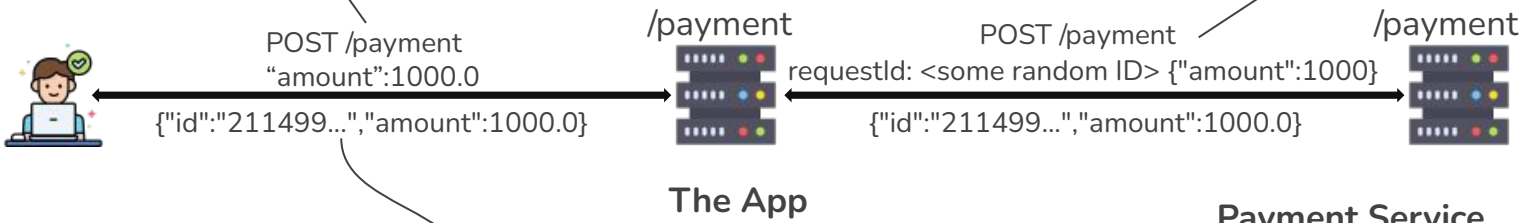


The payment service exposes an endpoint that requires an HTTP request body. The app uses **OpenFeign**, **RestTemplate**, or **WebClient** to send requests to the endpoint the payment service exposes.

1. Introduction (8/10)

The app we implement needs to call the **/payment** endpoint exposed by the payment service.

Payment service implements a REST endpoint **/payment**. For our scenario, we assume an app needs to call this endpoint to make a payment.



The payment service exposes an endpoint that requires an HTTP request body. The app uses **OpenFeign**, **RestTemplate**, or **WebClient** to send requests to the endpoint the payment service exposes.

This is the app we implement now and that we’ll use throughout the chapter.

1. Introduction (9/10)

Payment Service App

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

We add to the **pom.xml** file the web dependency

```
public class Payment {
private String id;
private double amount;
// Omitted getters and setters
}
```

We'll model the payment with the Payment class

We'll use this app in all our next examples. Let's create the project "sq-ch11-payments," which represents the payments service.

1. Introduction (10/10)

The app exposes the endpoint with HTTP POST at the path /payment.

```
@RestController
public class PaymentsController {
private static Logger logger =
Logger.getLogger(PaymentsController.class.getName());

@PostMapping("/payment")
public ResponseEntity<Payment> createPayment(

@RequestHeader String requestId,
@RequestBody Payment payment) {
logger.info("Received request with ID " + requestId +
" ;Payment Amount: " + payment.getAmount());
payment.setId(UUID.randomUUID().toString());

 return ResponseEntity
.status(HttpStatus.OK)
.header("requestId", requestId)
.body(payment);
  }
}
```

We use a logger to prove the right controller's method gets the correct data when the endpoint is called.

The endpoint needs to get a request header and the request body from the caller. The controller method gets these two details as parameters.

The method sets a random value for the payment's ID.

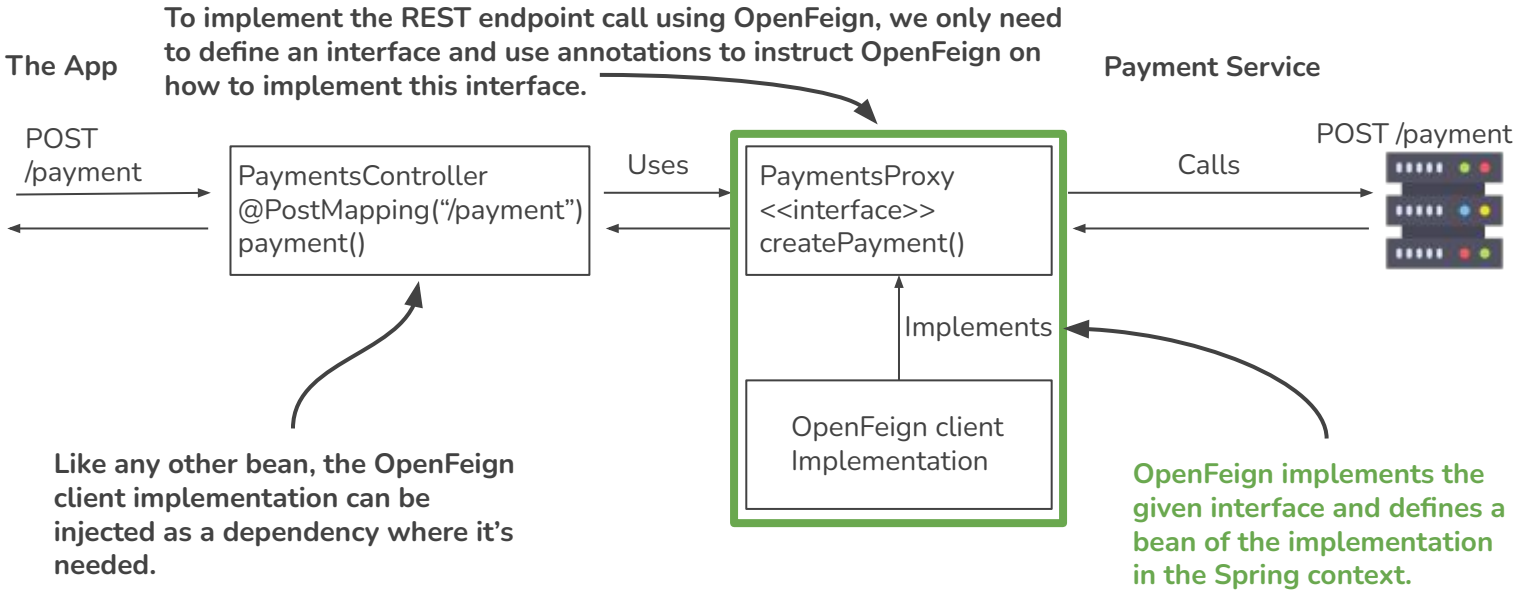
The controller action returns the HTTP response. The response body that contains the payments with the random ID value set.

The code shows above, the endpoint's implementation in the controller class.

1. Introduction (10/10)

Let’s we discuss deeper about each methods of calling REST Endpoints on the next pages...

2. REST Endpoints Using Spring Cloud OpenFeign (1/5)



With OpenFeign, you only need to define an interface (a contract) and tell OpenFeign where to find this contract to implement it.

2. REST Endpoints Using Spring Cloud OpenFeign (2/5)

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

Our pom.xml file needs to define the dependency, then we create the proxy interface.

Declaring an OpenFeign client interface

```
@FeignClient(name = "payments",
             url = "${name.service.url}")
public interface PaymentsProxy {

   @PostMapping("/payment")
   Payment createPayment(
       @RequestHeader String requestId,
       @RequestBody Payment payment);
}
```

We use the @FeignClient annotation to configure the REST client. A minimal configuration defines a name and the endpoint base URI.

We specify the endpoint's path and HTTP method.

We define the request headers and body.

Here, we implement an app that allows users to make payments. To make a payment, we need to call an endpoint of another system.

2. REST Endpoints Using Spring Cloud OpenFeign (3/5)

OpenFeign needs to know where to find the interfaces defining the client contracts. We use the **@EnableFeignClients** annotation on a configuration class to enable the OpenFeign functionality and tell OpenFeign where to search for the client contracts.

Enabling the OpenFeign clients in the configuration class

```
@Configuration
@EnableFeignClients(
   basePackages = "com.example.proxy")
public class ProjectConfig {
}
```

We enable the OpenFeign clients and tell the OpenFeign dependency where to search for the proxy contracts.

2. REST Endpoints Using Spring Cloud OpenFeign (4/5)

Injecting and using the OpenFeign client

```
@RestController
public class PaymentsController {

  private final PaymentsProxy paymentsProxy;

  public PaymentsController(PaymentsProxy paymentsProxy) {
    this.paymentsProxy = paymentsProxy;
  }

  @PostMapping("/payment")
  public Payment createPayment(
      @RequestBody Payment payment) {
    String requestId = UUID.randomUUID().toString();
    return paymentsProxy.createPayment(requestId, payment);
  }
}
```

Following snippet code shows us the controller class that injects the **FeignClient**.

2. REST Endpoints Using Spring Cloud OpenFeign (5/5)

```
curl -X POST -H 'content-type:application/json' -d
'{"amount":1000}'
➥ http://localhost:9090/payment

/* In the console where you executed the cURL command,
you'll find a response, as presented in the next snippet:
*/

{"id":"1c518ead-2477-410f-82f3-
54533b4058ff","amount":1000.0}

/* In the payment service's console, you find the log
proving that the app correctly sent
the request to the payment service:
*/

Received request with ID 1c518ead-2477-410f-82f3-
54533b4058ff ;Payment
➥ Amount: 1000.0
```

2. Calling REST Endpoints Using RestTemplate (1/6)

In this section, we again implement the app that calls the **/payment** endpoint of the payment service using RestTemplate.

STEP 1

Define the HTTP headers by creating and configuring an HttpHeaders object instances.

```
HttpHeaders headers = new
HttpHeaders();
headers.add("requested", "ID Value");
```

STEP 2

Define the HTTP request data by defining an HttpEntity object instances.

```
HttpEntity<Payment> httpEntity =
      new HttpEntity<>(
                  payment, headers);
```

STEP 3

Use the exchange() method to send the HTTP request.

```
ResponseEntity<Payment>response=
      rest exchange(url,
            HttpMethod.POST,
            httpEntity,
            Payment.class);
```

To define a more complex HTTP request, we have to use the HttpHeaders class to define the headers, then the HttpEntity class to represent the full request data. Once we defined the data on the request, we call the **exchange()** method to send it.

2. Calling REST Endpoints Using RestTemplate (2/6)

We take the URL to the payment service from the properties file.

```
@Component
public class PaymentsProxy {
  private final RestTemplate rest;

  @Value("${name.service.url}")
  private String paymentsServiceUrl;

  public PaymentsProxy(RestTemplate rest) {
    this.rest = rest;
  }

  public Payment createPayment(Payment payment) {
```

We take the URL to the payment service from the properties file.

We inject the RestTemplate from the Spring context using constructor DI.

2.Calling REST Endpoints Using RestTemplate(3/6)

```
      String uri = paymentsServiceUrl + "/payment";
      HttpHeaders headers = new HttpHeaders();
      headers.add("requestId",
                  UUID.randomUUID().toString());

      HttpEntity<Payment> httpEntity =
        new HttpEntity<>(payment, headers);

      ResponseEntity<Payment> response =
          rest.exchange(uri,
              HttpMethod.POST,
              httpEntity,
              Payment.class);

      return response.getBody();
  }
}
```

We build the HttpHeaders object to define the HTTP request headers.

We build the HttpEntity object to define the request data.

We send the HTTP request and retrieve the data on the HTTP response.

We return the HTTP response body.

2.Calling REST Endpoints Using RestTemplate(4/6)

Here, in this example of our second project, we find the definition of the proxy class.
Observe how the **createPayment()** method defines the header by creating an **HttpHeaders** instance and adding the needed header "*requestId*" to this instance using the **add()** method.
It then creates a HttpEntity instance based on the headers and the body (received by the method as a parameter).
The method then sends the HTTP request using **RestTemplate**'s **exchange()** method.
The **exchange()** method's parameters are the URI and the HTTP method, followed by the HttpEntity instance (that holds the request data) and the type expected for the response body.

2. Calling REST Endpoints Using RestTemplate (5/6)

Defining a controller class to test the implementation

```
@RestController
public class PaymentsController {

  private final PaymentsProxy paymentsProxy;

  public PaymentsController(PaymentsProxy paymentsProxy) {
    this.paymentsProxy = paymentsProxy;
  }

  @PostMapping("/payment")
  public Payment createPayment(
      @RequestBody Payment payment
      ) {
    return paymentsProxy.createPayment(payment);
  }
}
```

We define a controller action and map it to the /payment path.

We get the payment data as a request body.

We call the proxy method, which in turn calls the endpoint of the payments service. We get the response body and return the body to the client.

2. Calling REST Endpoints Using RestTemplate (6/6)

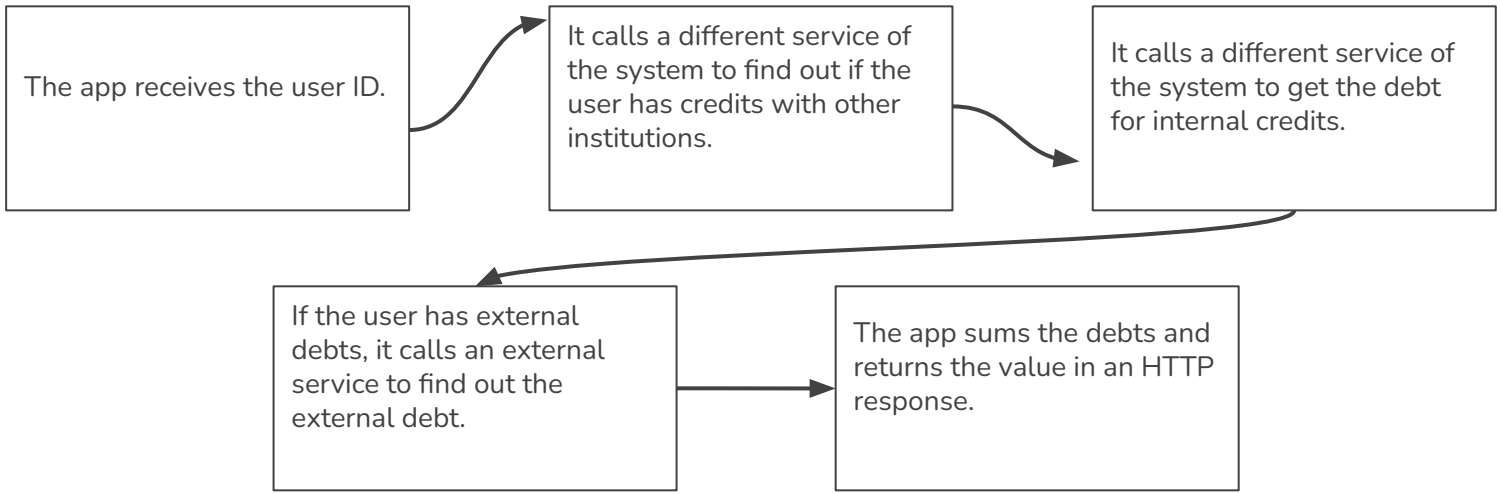
```
curl -X POST -H 'content-type:application/json' -d
'{"amount":1000}'
➥ http://localhost:9090/payment
{
"id":"21149959-d93d-41a4-a0a3-426c6fd8f9e9",
"amount":1000.0
}
/* In the payment service's console, you find the log
proving that the app correctly sent
the payment service request:
*/
Received request with ID e02b5c7a-c683-4a77-bd0e-
38fe76c145cf ;Payment
➥ Amount: 1000.0
```

3. Calling REST Endpoints Using WebClient (1/15)

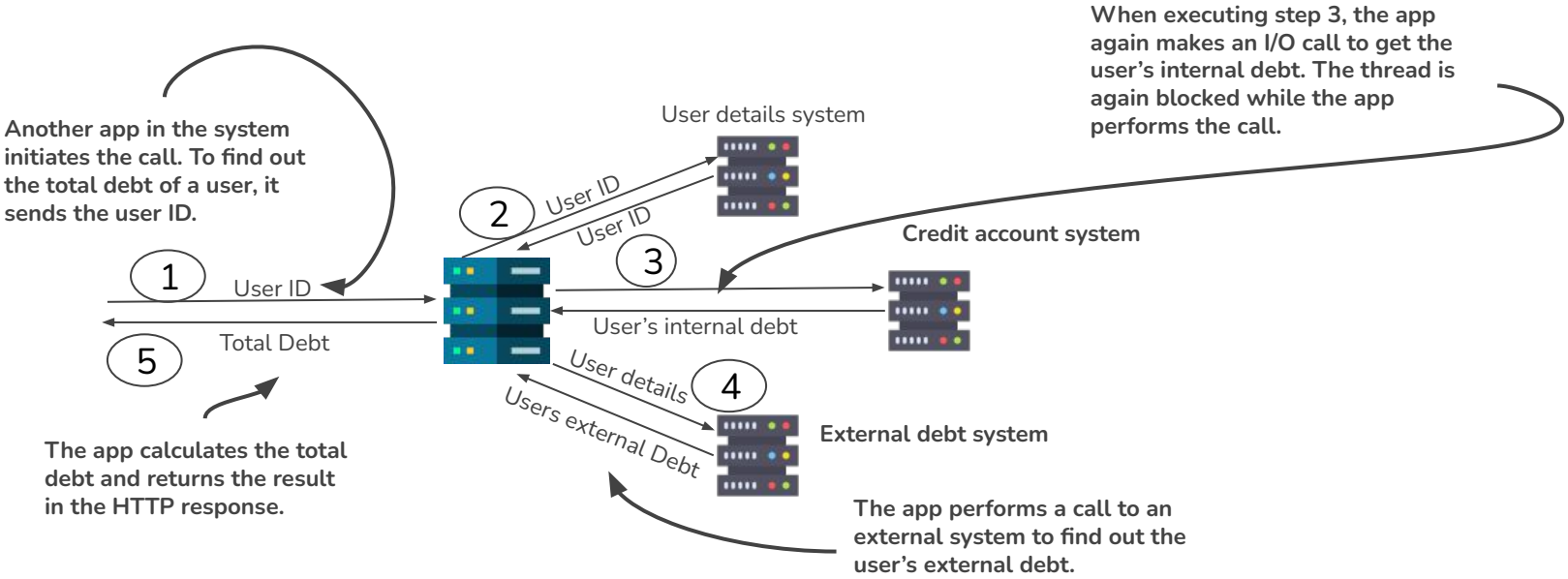
In this section, we discuss using **WebClient** to call *REST endpoints*. WebClient is a tool used in different apps and is built on a methodology we call a reactive approach.

3. Calling REST Endpoints Using WebClient (2/15)

Suppose we implement a banking application where a bank's client has one or more credit accounts. The system component we implement calculates the total debt of a bank's client. To use this functionality, other system components make a REST call to send a unique ID to the user. To calculate this value, the flow we implement includes the following steps...

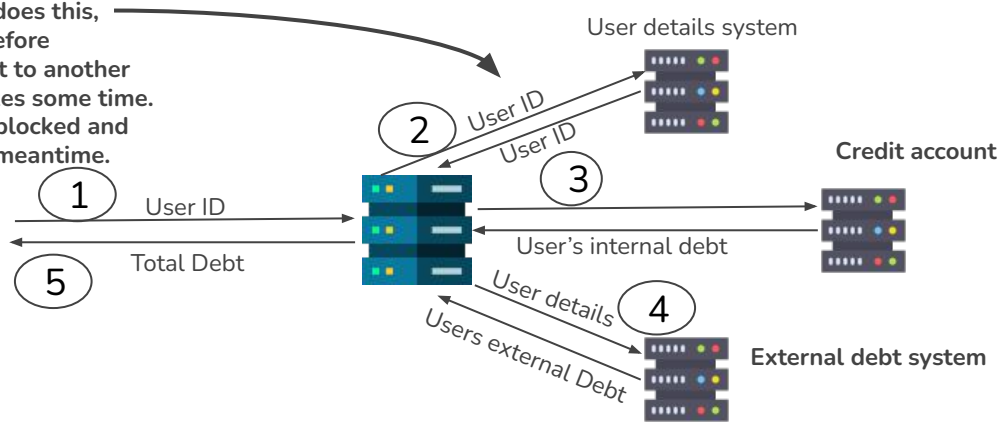


3. Calling REST Endpoints Using WebClient (3/15)



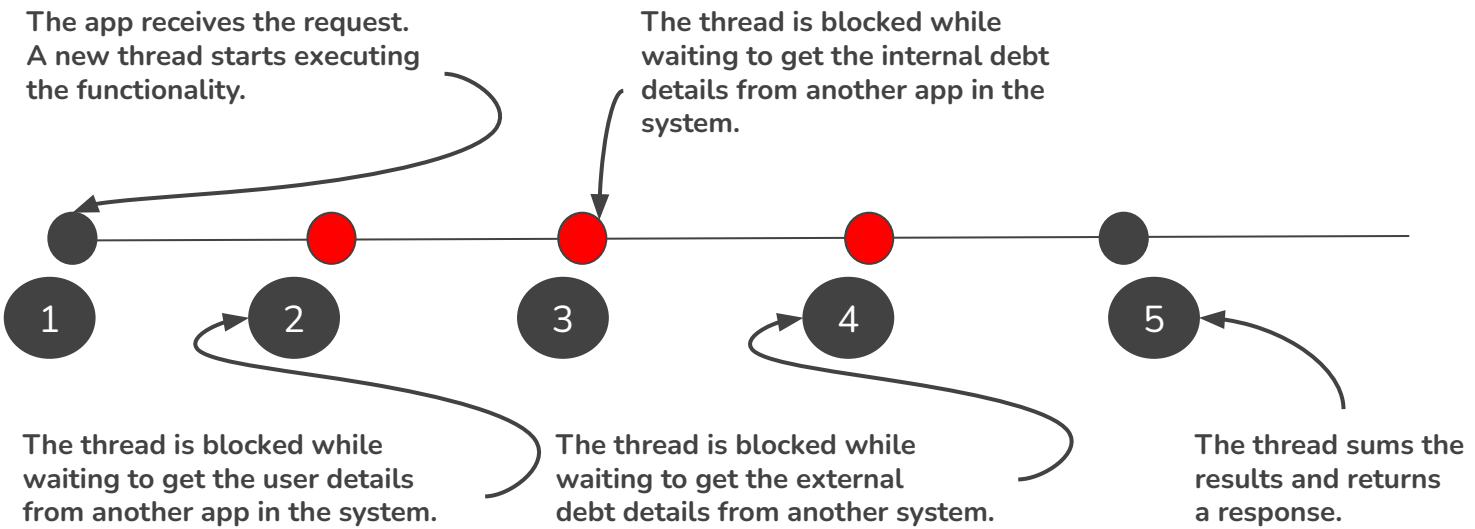
3. Calling REST Endpoints Using WebClient (4/15)

The functionality you implement needs to first call another service in the system to find the user details. When it does this, the app waits for the response before proceeding to step 3. Any request to another component is an I/O call, so it takes some time. The thread executing this call is blocked and cannot do something else in the meantime.



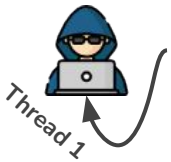
A functionality scenario for demonstrating the usefulness of a reactive approach. A banking app needs to call several other apps to calculate the total debt of a user. Due to these calls, the thread executing the request is blocked several times while waiting for I/O operations to finish.

3. Calling REST Endpoints Using WebClient (5/15)



The execution of the scenario functionality from the thread point of view. The arrow represents the timeline of the thread. Some of the steps cause details to block the thread, which needs to wait for the task to finish before proceeding.

3. Calling REST Endpoints Using WebClient (6/15)



Come on folks! We have three simultaneous requests!
Hey Ginny, I'll take **step 2 of the first request.**

The tasks that need to be performed

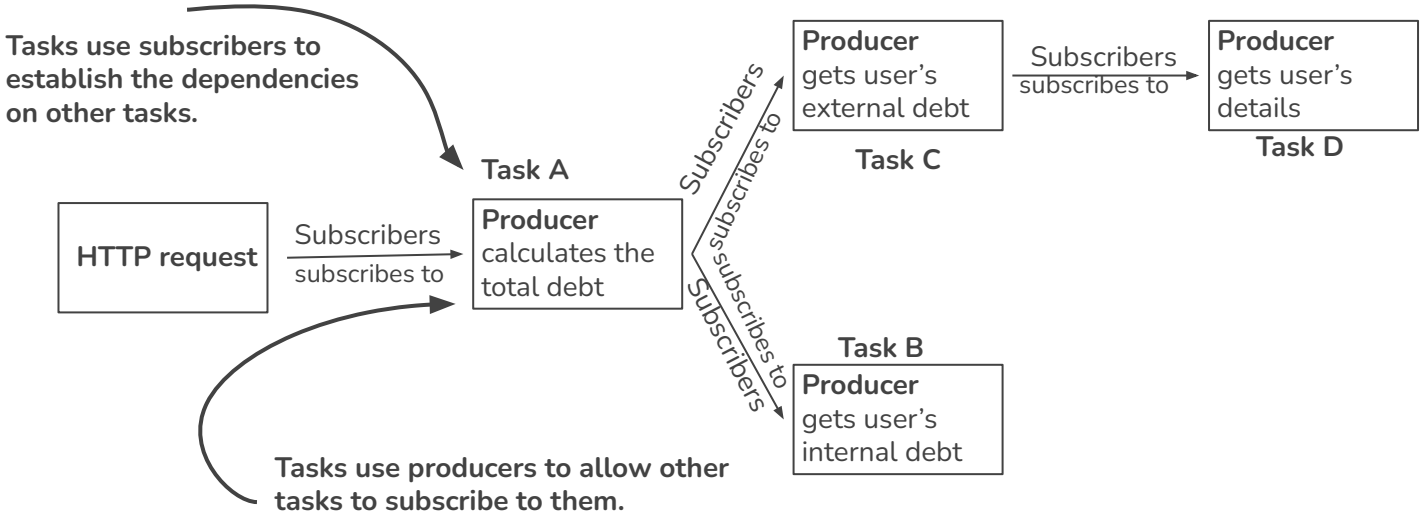
Okay George! I'll take **step 3 of the first request.** I see it doesn't depend on you!

Guys, I'm blocked waiting for the external credit list on **step 4 of the second request.** I'll leave it for later and start on step 1 of the third request.

Ginny, the task you left earlier because it was blocked can be solved now. I'll continue it!

An analogy of the way a reactive app works. A thread doesn't take a request's tasks in order and wait when it's blocked. Instead, all tasks from all requests are on a backlog. Any available thread can work on tasks from any request. This way, independent tasks can be solved in parallel, and the threads don't stay idle.

3. Calling REST Endpoints Using WebClient (7/15)



In a reactive app, the steps become tasks. Each task marks its dependencies on other tasks and allows other tasks to depend on them. Threads are free to execute any task.

3. Calling REST Endpoints Using WebClient (8/15)

Adding a WebClient bean to the Spring context in the configuration class

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

Adding dependency named **WebFlux**

```
@Configuration
public class ProjectConfig {

  @Bean
  public WebClient webClient() {
    return WebClient
            .builder()
            .build();
  }
}
```

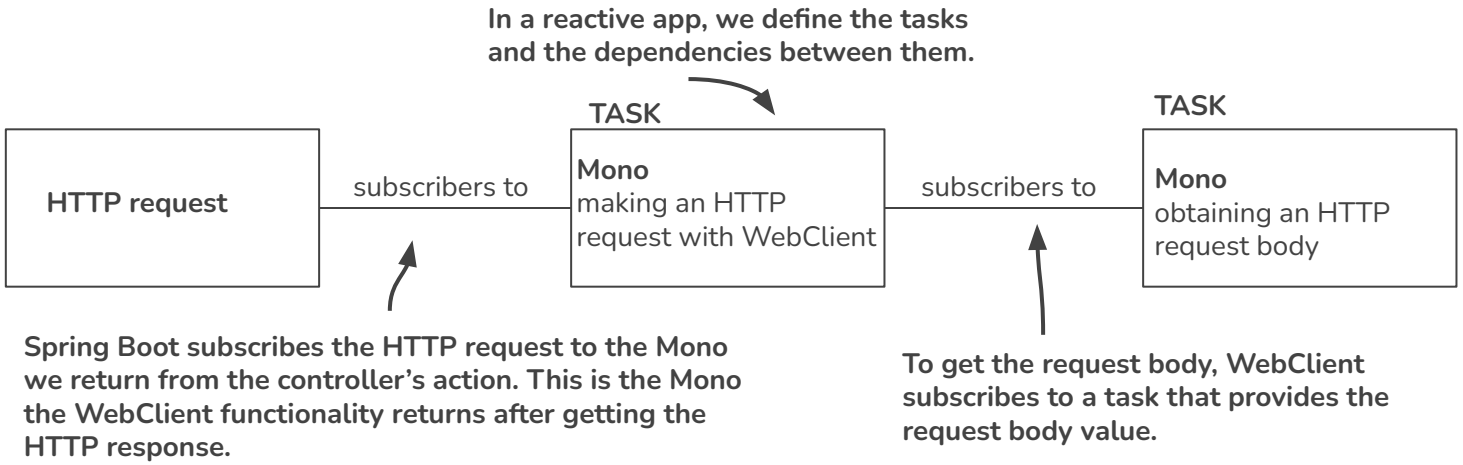
Creates a WebClient bean and adds it in the Spring context

To call the REST endpoint, you need to use a **WebClient** instance. The best way to create easy access is to put it in the Spring context using the **@Bean** annotation with a configuration class method

3. Calling REST Endpoints Using WebClient (9/15)

Implementing a proxy class with WebClient

3. Calling REST Endpoints Using WebClient (10/15)



*The tasks chain in a reactive app. When building a reactive web app, we define the tasks and the dependencies between them. The **WebFlux** functionality initiating the **HTTP** request subscribes to the task we create through the producer the controller's action returns. In our case, this producer is the one we get by sending the **HTTP** request with **WebClient**. For **WebClient** to make the request, it subscribes to another task that provides the request body.*

3. Calling REST Endpoints Using WebClient (10/15)

The tasks chain in a reactive app.

When building a reactive web app, we define the tasks and
the dependencies between them.
The **WebFlux** functionality initiating the **HTTP** request subscribes to the task we create through the producer the controller's action returns.
In our case, this producer is the one we get by sending the **HTTP** request with **WebClient**.
For **WebClient** to make the request, it subscribes to another task that provides the request body.

3. Calling REST Endpoints Using WebClient (11/15)

A controller class exposing an endpoint and calling the proxy

```
@RestController
public class PaymentsController {

  private final PaymentsProxy paymentsProxy;

  public PaymentsController(PaymentsProxy paymentsProxy) {
    this.paymentsProxy = paymentsProxy;
  }

  @PostMapping("/payment")
  public Mono<Payment> createPayment(
    @RequestBody Payment payment
    ) {

    String requestId = UUID.randomUUID().toString();
    return paymentsProxy.createPayment(requestId,
                                       payment);
  }
}
```

3. Calling REST Endpoints Using WebClient (12/15)

```
curl -X POST -H 'content-type:application/json' -d
'{"amount":1000}'
➥ http://localhost:9090/payment
/* In the console where you executed the cURL command,
you'll find a response like
the next snippet:
*/
{
"id":"e1e63bc1-ce9c-448e-b7b6-268940ea0fcc",
"amount":1000.0
}
/* In the payment service console, you find the log proving
that this section's app correctly sends the request to the
payment service:

Received request with ID e1e63bc1-ce9c-448e-b7b6-
268940ea0fcc ;Payment
➥ Amount: 1000.0
*/
```

3. Calling REST Endpoints Using WebClient (13/15)

Explanations of using Mono and Flux in WebClient

- **Mono** and **Flux** are **reactive types** provided by **Project Reactor**, which is a *library for building reactive applications*.
- They are used in Spring WebClient to handle asynchronous and non-blocking operations.

3. Calling REST ndpoints Using WebClient (14/15)

***Example**: Fetching a Single Book by ID*

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

// Use Mono to fetch a single book
Mono<Book> bookMono = webClient.get()
    .uri("/api/books/{id}", 1)  // The endpoint URI
    .retrieve()                 // Send the request and retrieve the response
    .bodyToMono(Book.class);    // Convert the response body to Mono<Book>

// Process the result asynchronously
bookMono.subscribe(book -> System.out.println("Book: " + book));
```

Mono - represents a single value or an empty value. We use **Mono** when the result of our HTTP request is a single resource.

3. Calling REST Endpoints Using WebClient (15/15)

***Example**: Fetching a Single Book by ID*

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

// Use Flux to fetch a list of books
Flux<Book> booksFlux = webClient.get()
    .uri("/api/books")          // The endpoint URI
    .retrieve()                 // Send the request and retrieve the response
    .bodyToFlux(Book.class);    // Convert the response body to Flux<Book>

// Process each book asynchronously
booksFlux.subscribe(book -> System.out.println("Book: " + book));
```

Flux represents a stream of multiple values. We use Flux when the result of our HTTP request is multiple resources.

Conclusion

In a real-world backend solution, we often find cases when a backend app needs to call endpoints exposed by another backend app.

Spring offers multiple solutions for implementing the client side of a REST service. Three of the most relevant solutions are as follows:

- **OpenFeign** - A solution offered by the **Spring Cloud** project that successfully simplifies the code we need to write to call a REST endpoint and adds several features relevant to how we implement services today.

Conclusion

- **RestTemplate** - A simple tool used to call REST endpoints in Spring apps.
- **WebClient** - A reactive solution for calling REST endpoints in a Spring app.

  We shouldn't use **RestTemplate** in new implementations. We can choose between **OpenFeign** and **WebClient** to call REST endpoints.
  For an app following a standard (nonreactive) approach, the best choice is using **OpenFeign**.
- **WebClient** is an excellent tool for an app designed on a reactive approach. But before using it, you should deeply understand the reactive approach and how to implement a reactive app with Spring.

Resources


Spring
Start Here
LEARN WHAT YOU NEED
AND LEARN IT WELL
LAURENȚIU SPILCĂ
FOREWORD BY VICTOR RENTEA
MANNING

Reference

1. Spring Start Here
2. Baeldung.com
3. w3schools.com

Thank you!

Presented by

Nizomiddinov Shakhobiddin

(shohobiddindeveloper@gmail.com)