**Spring Boot 3.2: Replace Your RestTemplate With RestClient**

This Spring Boot 3.2 tutorial explores an addition built upon WebClient called RestClient, a more intuitive and modern approach to consuming RESTful services.

In the world of [Spring Boot](https://dzone.com/articles/spring-boot-nice-amp-easy-video-8), making HTTP requests to external services is a common task. Traditionally, developers have relied on RestTemplate for this purpose. However, with the evolution of the Spring Framework, a new and more powerful way to handle HTTP requests has emerged: the [WebClient](https://dzone.com/articles/resttemplate-vs-webclient). In Spring Boot 3.2, a new addition called RestClient builds upon WebClient, providing a more intuitive and modern approach to consuming [RESTful](https://dzone.com/refcardz/rest-foundations-restful) services.

**Origins of RestTemplate**

RestTemplate has been a staple in the Spring ecosystem for years. It's a synchronous client for making HTTP requests and processing responses. With RestTemplate, developers could easily interact with RESTful APIs using familiar [Java](https://dzone.com/refcardz/core-java) syntax. However, as applications became more asynchronous and non-blocking, the limitations of RestTemplate started to become apparent.

Here's a basic example of using RestTemplate to fetch data from an external API:

var restTemplate = new RestTemplate();

var response = restTemplate.getForObject("https://api.example.com/data", String.class);

System.out.println(response);

**Introduction of WebClient**

With the advent of [Spring WebFlux](https://dzone.com/articles/embracing-reactive-programming-with-spring-webflux), an asynchronous, non-blocking web framework, WebClient was introduced as a modern alternative to RestTemplate. WebClient embraces reactive principles, making it well-suited for building reactive applications. It offers support for both synchronous and asynchronous communication, along with a fluent API for composing requests.

Here's how you would use WebClient to achieve the same HTTP request:

Java

var webClient = WebClient.create();

var response = webClient.get()

.uri("https://api.example.com/data")

.retrieve()

.bodyToMono(String.class);

response.subscribe(System.out::println);

**Enter RestClient in Spring Boot 3.2**

[Spring Boot 3.2](https://dzone.com/articles/getting-started-with-boot-spring-320-building-a-he) brings RestClient, a higher-level abstraction built on top of WebClient. RestClient simplifies the process of making HTTP requests even further by providing a more intuitive fluent API and reducing boilerplate code. It retains all the capabilities of WebClient while offering a more developer-friendly interface.

Let's take a look at how RestClient can be used:

var response = restClient

.get()

.uri(cepURL)

.retrieve()

.toEntity(String.class);

System.out.println(response.getBody());

With RestClient, the code becomes more concise and readable. The RestClient handles the creation of WebClient instances internally, abstracting away the complexities of setting up and managing HTTP clients.

**Comparing RestClient With RestTemplate**

Let's compare RestClient with RestTemplate by looking at some common scenarios:

**Create**

RestTemplate:

var response = new RestTemplate();

RestClient:

var response = RestClient.create();

Or we can use our old RestTemplate as well:

var myOldRestTemplate = new RestTemplate();

var response = RestClient.builder(myOldRestTemplate);

**GET Request**

RestTemplate:

var response = restTemplate.getForObject("https://api.example.com/data", String.class);

​

RestClient:

var response = restClient

.get()

.uri(cepURL)

.retrieve()

.toEntity(String.class);

**POST Request**

RestTemplate:

ResponseEntity<String> response = restTemplate.postForEntity("https://api.example.com/data", request, String.class);

RestClient:

var response = restClient

.post()

.uri("https://api.example.com/data")

.body(request)

.retrieve()

.toEntity(String.class);

**Error Handling**

RestTemplate:

try {

String response = restTemplate.getForObject("https://api.example.com/data", String.class);

} catch (RestClientException ex) {

// Handle exception

}

RestClient:

String request = restClient.get()

.uri("https://api.example.com/this-url-does-not-exist")

.retrieve()

.onStatus(HttpStatusCode::is4xxClientError, (request, response) -> {

throw new MyCustomRuntimeException(response.getStatusCode(), response.getHeaders())

})

  .body(String.class);

As seen in these examples, RestClient offers a more streamlined approach to making HTTP requests compared to RestTemplate.

Spring Documentation gives us many other examples.

https://docs.spring.io/spring-framework/reference/integration/rest-clients.html#\_migrating\_from\_resttemplate\_to\_restclient