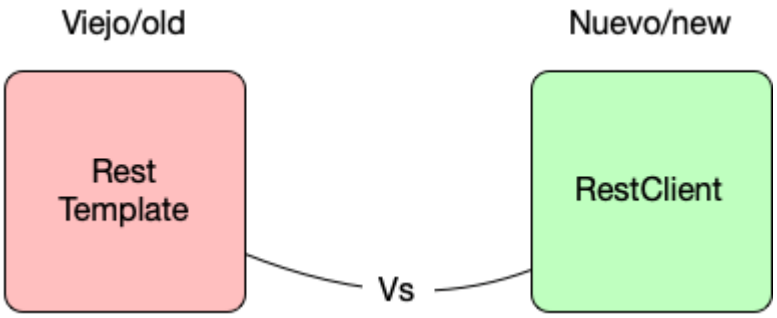
Tabla de Contenidos

En el ecosistema Spring siempre ha existido la necesidad de comunicarnos con otros servicios HTTP. Durante años hemos utilizado RestTemplate y más recientemente WebClient, pero con la llegada de Spring Framework 6.1 / Spring Boot 3.2 aparece un nuevo actor: RestClient.



En este artículo veremos qué es RestClient, por qué surge, en qué casos usarlo y cómo utilizarlo con ejemplos claros, siguiendo un enfoque práctico y orientado a arquitectura, como solemos hacer cuando hablamos de Spring Boot.

¿Por qué un nuevo cliente HTTP?

Spring ha tenido históricamente dos opciones principales:

- RestTemplate: sencillo y muy usado, pero obsoleto (deprecated).
- WebClient: potente, reactivo y no bloqueante, pero excesivo para muchos casos donde solo queremos llamadas HTTP simples y bloqueantes.
  Aquí es donde entra RestClient:

> Un cliente HTTP sincrónico, moderno, con una API fluida y alineada con WebClient, pero pensado para aplicaciones imperativas.

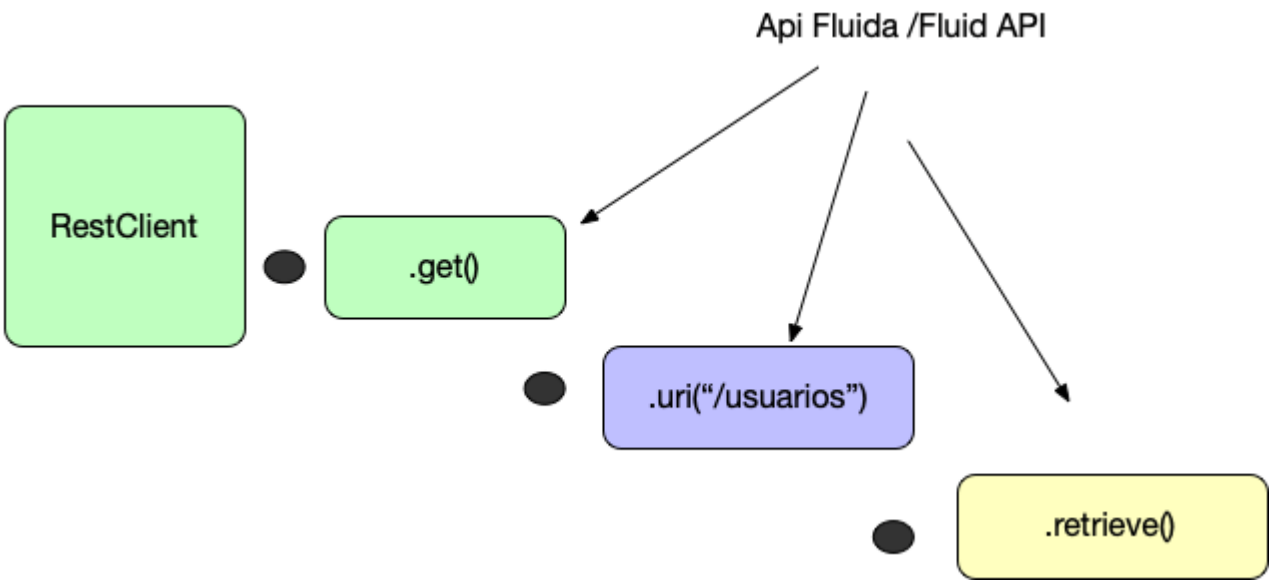
En otras palabras:

- No es reactivo
- No sustituye a WebClient
- Sustituye de forma natural a RestTemplate

¿Qué es RestClient?

RestClient es un cliente HTTP introducido en Spring Framework 6.1 que:

- Usa una API fluida (builder)
- Está basado internamente en WebClient
- Es bloqueante (imperativo)
- Tiene soporte nativo en Spring Boot 3.2+
  Su objetivo es claro: simplificar las llamadas HTTP en aplicaciones Spring Boot tradicionales.



Dependencias necesarias

Si estás usando Spring Boot 3.2 o superior, no necesitas añadir nada especial. Basta con tener:

- spring-boot-starter-web
  Spring Boot ya te proporciona la infraestructura necesaria para usar RestClient.

Creando un RestClient

La forma recomendada es definirlo como bean, lo que encaja perfectamente con una arquitectura limpia y desacoplada.

Ejemplo de configuración:

```
@Configuration
public class RestClientConfig {

    @Bean
```

```
    RestClient restClient(RestClient.Builder builder) {
        return builder
                .baseUrl("https://api.ejemplo.com")
                .defaultHeader(HttpHeaders.CONTENT_TYPE,
MediaType.APPLICATION_JSON_VALUE)
                .build();
    }
}
```

Puntos clave:

- Usamos RestClient.Builder (inyectado por Spring)
- Centralizamos configuración
- Facilitamos testing y mantenimiento

Realizando una petición GET

Supongamos un servicio externo que devuelve un `Usuario`.

```
public record Usuario(Long id, String nombre, String email) {}
```

Uso del RestClient:

```
@Service
public class UsuarioClient {

    private final RestClient restClient;

    public UsuarioClient(RestClient restClient) {
        this.restClient = restClient;
    }

    public Usuario obtenerUsuario(Long id) {
```

```
        return restClient
                .get()
                .uri("/usuarios/{id}", id)
                .retrieve()
                .body(Usuario.class);
    }
}
```

Observa la claridad del código:

- API fluida
- Sin plantillas verbosas
- Tipado fuerte

Peticiones POST

Ejemplo enviando un objeto al servicio remoto:

```
public Usuario crearUsuario(Usuario usuario) {
    return restClient
            .post()
            .uri("/usuarios")
            .body(usuario)
            .retrieve()
            .body(Usuario.class);
}
```

La lectura es casi natural: post → uri → body → retrieve → body.

Manejo de errores

Uno de los puntos importantes en cualquier integración HTTP es el control de errores.

Con RestClient podemos hacerlo de forma explícita:

```
return restClient
        .get()
        .uri("/usuarios/{id}", id)
        .retrieve(response -> {
            if (response.getStatusCode().is4xxClientError()) {
                throw new RuntimeException("Usuario no encontrado");
            }
            if (response.getStatusCode().is5xxServerError()) {
                throw new RuntimeException("Error en el servicio
remoto");
            }
            return response.body(Usuario.class);
        });
```

Esto encaja muy bien con una estrategia de excepciones de dominio.

RestClient y arquitectura

Desde un punto de vista arquitectónico:

- RestClient no debe usarse directamente en controladores
- Es ideal ubicarlo en una capa de infraestructura
- Encaja perfectamente en arquitecturas Hexagonales o Clean Architecture
  Ejemplo:

- application
- domain
- infrastructure/client/UsuarioClient
  Así mantenemos:

- Bajo acoplamiento
- Alta cohesión
- Código testeable

¿RestClient o WebClient?

La decisión es sencilla:

ESCENARIO	CLIENTE RECOMENDADO
App imperativa	RestClient
App reactiva	WebClient
Migración desde RestTemplate	RestClient

No se trata de elegir uno u otro, sino de usar el adecuado según el contexto.

Conclusiones

RestClient llega para ocupar un hueco muy claro en Spring Boot:

- Sustituye a RestTemplate
- Simplifica integraciones HTTP
- Mantiene una API moderna y coherente
- Se adapta perfectamente a arquitecturas bien diseñadas
  Si trabajas con Spring Boot 3, microservicios y arquitectura limpia, RestClient es una herramienta que deberías empezar a usar desde ya.

Como siempre, en arquitectura no se trata solo de que el código funcione, sino de que sea mantenible, claro y evolutivo.

- ¿Que es un CharSequence?
- Spring REST Client con RestTemplates
- REST HTTP return codes y sus curiosidades
- JAX-RS Servicios RESTFull en Java
- Seguridad y JAAS (II)