**RestTemplate vs Webclient vs Completablefuture**

when to use resttemplate vs webclient in springboot

###### **ChatGPT said:**

In Spring Boot, you can use either RestTemplate or WebClient to make HTTP calls. However, the choice between them depends on your application's requirements and whether you are using a reactive or non-reactive (blocking) approach.

### **1. RestTemplate (Synchronous & Blocking)**

✅ **Use when:**

* You are working with a traditional Spring MVC application.
* You need a simple and straightforward way to make REST calls.
* You prefer blocking/synchronous calls.

⚠️ **Avoid if:**

* You need high concurrency and scalability.
* You want to leverage a reactive programming model.

📌 **Example Usage:**

java

CopyEdit

RestTemplate restTemplate = new RestTemplate();

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

System.out.println(response);

**🔴 Deprecation Note:** Spring recommends moving away from RestTemplate in favor of WebClient for new applications.

### **2. WebClient (Asynchronous & Non-blocking)**

✅ **Use when:**

* You are building a reactive application with Spring WebFlux.
* You need better scalability (especially in microservices).
* You want to perform concurrent, non-blocking API calls.
* You need advanced features like filters, retries, and timeouts.

⚠️ **Avoid if:**

* Your application is fully synchronous, and you don’t need the reactive approach.

📌 **Example Usage:**

java

CopyEdit

WebClient webClient = WebClient.create();

String response = webClient.get()

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

.retrieve()

.bodyToMono(String.class)

.block(); // Blocks for response, avoid in reactive flows

System.out.println(response);

**🔹 block() makes it synchronous. To stay fully reactive, use subscribe() or bodyToMono() without blocking.**

### **Which One to Use?**

| **Feature** | **RestTemplate** | **WebClient** |
| --- | --- | --- |
| Blocking/Synchronous | ✅ Yes | ❌ No |
| Non-blocking/Asynchronous | ❌ No | ✅ Yes |
| Spring MVC | ✅ Yes | ✅ Yes (but less ideal) |
| Spring WebFlux | ❌ No | ✅ Yes |
| Performance & Scalability | ❌ Less efficient | ✅ More efficient |
| Future Proof | ❌ Deprecated | ✅ Preferred |

### **Conclusion**

* If you're working with **Spring WebFlux** or need **high performance & scalability**, go for **WebClient**.
* If you're working in a **Spring MVC (blocking) environment**, you can still use RestTemplate, but consider migrating to WebClient for future-proofing.

Would you like an example of how to migrate from RestTemplate to WebClient? 🚀

##### **You said:**

WebClient webClient = WebClient.create(); String response = webClient.get() .uri("https://api.example.com/data") .retrieve() .bodyToMono(String.class) .block(); // Blocks for response, avoid in reactive flows System.out.println(response); provide complete asynchronous approach

###### **ChatGPT said:**

To make the WebClient call fully asynchronous and non-blocking, you should avoid using .block(). Instead, you can use **reactive programming** with **Mono** or **Flux**, and handle the response using **subscribe()** or chaining with .doOnSuccess().

### **Fully Asynchronous Approach with WebClient**

java

CopyEdit

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

import reactor.core.publisher.Mono;

public class WebClientExample {

public static void main(String[] args) {

WebClient webClient = WebClient.create();

webClient.get()

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

.retrieve()

.bodyToMono(String.class) // Returns a Mono<String> asynchronously

.doOnSuccess(response -> System.out.println("Response: " + response))

.doOnError(error -> System.err.println("Error: " + error.getMessage()))

.subscribe(); // Non-blocking call

System.out.println("Request sent asynchronously!");

// Sleep to prevent the main thread from exiting immediately (for demo purposes)

try {

Thread.sleep(5000); // Adjust as needed to let async execution complete

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

### **Explanation:**

1. .bodyToMono(String.class): Converts the response body into a Mono<String>, which is a reactive type representing a single value.
2. .doOnSuccess(response -> ...): Executes code when the response is received.
3. .doOnError(error -> ...): Handles errors without blocking the execution.
4. .subscribe(): Subscribes to the reactive stream, making the call non-blocking.
5. Thread.sleep(5000): Ensures the JVM doesn’t exit before the async response arrives (needed only in standalone Java apps).

### **Alternative Approach Using CompletableFuture**

If you want to use Java's CompletableFuture to handle the response:

java

CopyEdit

import java.util.concurrent.CompletableFuture;

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

public class WebClientAsyncExample {

public static void main(String[] args) {

WebClient webClient = WebClient.create();

CompletableFuture<String> futureResponse = webClient.get()

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

.retrieve()

.bodyToMono(String.class)

.toFuture(); // Converts Mono to CompletableFuture

futureResponse.thenAccept(response -> System.out.println("Response: " + response))

.exceptionally(error -> {

System.err.println("Error: " + error.getMessage());

return null;

});

System.out.println("Request sent asynchronously!");

}

}

### **Key Takeaways:**

* ✅ **Use .subscribe()** for fire-and-forget behavior.
* ✅ **Use .doOnSuccess()** for additional handling.
* ✅ **Use .toFuture()** if integrating with Java’s CompletableFuture.
* ❌ **Avoid .block()** unless explicitly needed.

Would you like to integrate this into a Spring Boot application with WebFlux? 🚀

##### **You said:**

why to use completablefuture if webclient provides doonsuccess

###### **ChatGPT said:**

Great question! Both CompletableFuture and doOnSuccess() provide ways to handle asynchronous responses from WebClient, but they serve different use cases. Let's break down when and why you might use one over the other.

## **Why Use CompletableFuture If WebClient Provides doOnSuccess()?**

### **1️⃣ doOnSuccess() - Purely Reactive Approach (Preferred in WebFlux)**

✅ **Use When:**

* You are working in a **fully reactive** Spring WebFlux application.
* You want to chain multiple operations without converting to blocking types.
* You are handling responses **within the reactive pipeline**.

🔹 **Example Using doOnSuccess() (Preferred for WebFlux)**

java

CopyEdit

WebClient webClient = WebClient.create();

webClient.get()

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

.retrieve()

.bodyToMono(String.class)

.doOnSuccess(response -> System.out.println("Response: " + response)) // Reactive handling

.doOnError(error -> System.err.println("Error: " + error.getMessage()))

.subscribe(); // Fire-and-forget, non-blocking

📌 **Key Takeaways:**

* **Completely non-blocking** and reactive.
* **Cannot return values** outside the reactive chain.
* **Best for Spring WebFlux applications**.

### **2️⃣ CompletableFuture - Useful for Java Interoperability**

✅ **Use When:**

* You need to integrate **WebClient with non-reactive parts of your code**.
* You are working in a **Spring MVC (non-reactive) application**.
* You need **explicit control over async execution**, such as using thenApply(), thenCombine(), or join().

🔹 **Example Using CompletableFuture**

java

CopyEdit

import java.util.concurrent.CompletableFuture;

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

public class WebClientAsyncExample {

public static void main(String[] args) {

WebClient webClient = WebClient.create();

CompletableFuture<String> futureResponse = webClient.get()

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

.retrieve()

.bodyToMono(String.class)

.toFuture(); // Converts to CompletableFuture

futureResponse.thenAccept(response -> System.out.println("Response: " + response))

.exceptionally(error -> {

System.err.println("Error: " + error.getMessage());

return null;

});

System.out.println("Request sent asynchronously!");

}

}

📌 **Key Takeaways:**

* **Bridges reactive WebClient with imperative Java code**.
* **Easier to work with legacy code that expects Future-like objects**.
* **Works well when using Java’s CompletableFuture APIs for composition**.

## **When to Use Which?**

| **Feature** | **doOnSuccess() (Reactive)** | **CompletableFuture (Imperative)** |
| --- | --- | --- |
| Works with Spring WebFlux? | ✅ Yes (Preferred) | ❌ No (Better for Spring MVC) |
| Fully non-blocking? | ✅ Yes | ✅ Yes |
| Return values outside of WebFlux? | ❌ No | ✅ Yes |
| Interoperability with non-reactive Java? | ❌ No | ✅ Yes |
| Composable in reactive chains? | ✅ Yes | ❌ No |

### **Conclusion**

* If you're building a **Spring WebFlux** application, prefer **doOnSuccess() and .subscribe()** for full reactivity.
* If you need to **bridge WebClient with traditional Java code**, **use CompletableFuture**.
* If you're in a **Spring MVC application** and want asynchronous execution, **CompletableFuture is a better fit**.

Would you like an example of integrating WebClient with Spring Boot services?