ChatGPT

Spring WebFlux 핸즈온 교육 자료

대상: 이 자료는 Spring WebMVC에 익숙하고 JavaScript의 `async/await` 등 비동기 프로그래밍 경험이 있는 설계자 및 개발자를 위한 것입니다. 목표는 Spring WebFlux의 핵심 개념을 실용적으로 이해하여 실제 프로젝트에 적용할 수 있도록 돕는 것입니다.

1. WebFlux 개요

Spring WebMVC vs Spring WebFlux

Spring WebFlux는 Spring 5에서 도입된 **리액티브 웹 프레임워크**로, 전통적인 Spring WebMVC와는 처리 방식에서 큰 차이가 있습니다. WebMVC는 Servlet API 기반의 **동기적(블로킹)** 요청 처리 모델인 반면, WebFlux는 **비동기적(논블로킹) Reactive Streams** 기반으로 동작합니다. 다음은 두 웹 스택의 주요 차이점입니다:

- **동시성 모델:** Spring MVC에서는 들어온 각 HTTP 요청을 처리하기 위해 **별도의 스레드가 할당**됩니다. 한 요청을 처리하는 동안 해당 스레드는 응답이 완료될 때까지 대기 상태로 묶이게 됩니다 [1] [2]. 반면 Spring WebFlux에서는 **요청당 전용 스레드를 점유하지 않고**, **작은 개수의 이벤트 루프 스레드**가 여러 요청을 비동기로 처리합니다 [3] [4]. **즉, WebFlux는 CPU 코어 수 정도의 적은 스레드로도 매우 많은 동시 요청을 처리**할 수 있습니다 [4]. 이는 고부하 상황에서 스레드가 급증하고 컨텍스트 스위칭 비용이 커질 수 있는 MVC에 비해 **더 나은 확장성**을 제공합니다 [5] [6].

- **블로킹 vs 논블로킹 I/O:** WebMVC에서는 DB조회나 REST API 호출 같은 I/O 작업을 수행하면, 해당 **스레드가 응답을 받을 때까지 블로킹**됩니다 [2]. 따라서 많은 요청이 동시에 발생하면 대기 중인 스레드들로 인해 **스레드 풀 고갈** 및 성능 저하가 생길 수 있습니다 [7]. WebFlux에서는 **논블로킹 I/O**를 활용하여, I/O 요청을 보낸 후 **스레드를 블로킹하지 않고** 다른 작업에 활용합니다 [2]. **스레드는 어떤 I/O 응답을 기다리지 않으며**, 응답 도착 시 별도 이벤트로 처리합니다 [2]. 이러한 논블로킹 방식 덕분에 **높은 동시성 환경에서도 스레드 풀 고갈 없이** 효율적으로 처리할 수 있습니다 [7].

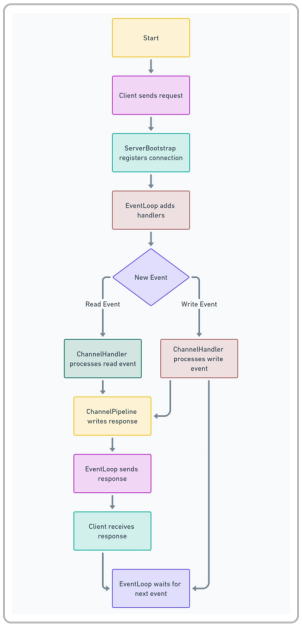
- **Reactive Streams와 백프레셔:** Spring WebFlux는 **Reactive Streams 표준** 위에 구축되어 있으며, Project Reactor를 기반으로 합니다. Reactive Streams는 **비동기 스트림 처리와 논블로킹 백프레셔(역압)**를 위한 표준으로, **퍼블리셔(Publisher)**와 **서브스크라이버(Subscriber)** 간에 **신뢰성 있는 데이터 흐름 제어**를 보장합니다 [8] [9]. **백프레셔**란 **데이터 소비자가 감당할 수 있는 만큼만 생산자가 제공하도록 신호를 보내는 메커니즘**으로, 빠른 생산자가 느린 소비자를 압도하지 않도록 조절해줍니다 [8]. WebFlux의 모든 내부 처리 흐름은 이 Reactive Streams 원칙에 따라 동작하며, **Mono**, **Flux** 등의 API를 통해 이러한 **비동기 스트림**을 다룹니다 [9]. 한편 WebMVC는 전통적인 `List`나 객체 반환을 통해 동기적으로 모든 데이터를 한꺼번에 처리하므로, 이런 스트림 형태의 백프레셔 개념이 적용되지 않습니다.

- **서블릿 컨테이너 vs 넷티(Netty):** Spring MVC는 서블릿 기술 위에서 동작하므로 주로 Tomcat, Jetty 같은 서블릿 컨테이너를 사용합니다. 반면 WebFlux는 **서블릿 없이 Netty같은 논블로킹 IO 서버**를 기본 사용합니다. 예를 들어 Spring Boot로 WebFlux 애플리케이션을 실행하면 기본적으로 **Reactor Netty** 서버가 내장되어 동작하며, Netty의 **이벤트 루프(event loop)** 모델을 활용합니다. 이벤트 루프란 **소수의 스레드가 I/O 이벤트를 감지하고 처리기(handler)에 전달**하는 방식으로 동작합니다 [10] [11]. 한 개의 이벤트 루프 스레드가 여러 소켓 연결의 읽기/쓰기 이벤트를 계속 감시하고 (non-blocking으로) 발생할 때마다 처리하기 때문에, **소수의 스레드로도 다수의 연결을 동시에 관리**할 수 있습니다 [10] [11].
  Spring WebFlux는 Netty 외에도 **서블릿 3.1+ 비동기 IO를 지원하는 Tomcat, Jetty, Undertow** 등 위에서

도 실행될 수 있지만, 동작 방식은 어디서든 **논블로킹 Reactive** 형태로 동일합니다 [12] [13]. (Spring Boot WebFlux 스타터는 기본 Netty를 사용하며, 종속성만 교체하면 Tomcat 등으로 바꿀 수 있습니다 [14].)

以上 요약하면, **Spring WebFlux는 대량의 동시 연결이나 스트리밍 데이터 처리 같은 고성능 요구사항에 적합한 반면**, **Spring MVC는 전통적인 동기 처리로 이해와 개발이 상대적으로 쉽고** JDBC 같은 **블로킹 API와의 통합이 용이**합니다. 이미 MVC로 충분한 성능을 내는 애플리케이션이라면 굳이 WebFlux로 전환할 필요는 없지만, **높은 동시성 환경이나 실시간 스트림 처리**가 필요한 경우 WebFlux 도입을 고려할 수 있습니다 [15]. 또한 WebFlux와 MVC를 **혼용**하는 것도 가능한데 (예: 일부 컨트롤러만 WebFlux Mono/Flux로 구현), 이러한 점은 프로젝트 요구사항과 팀의 Reactive 프로그래밍 숙련도에 따라 결정하면 됩니다 [16] [17].

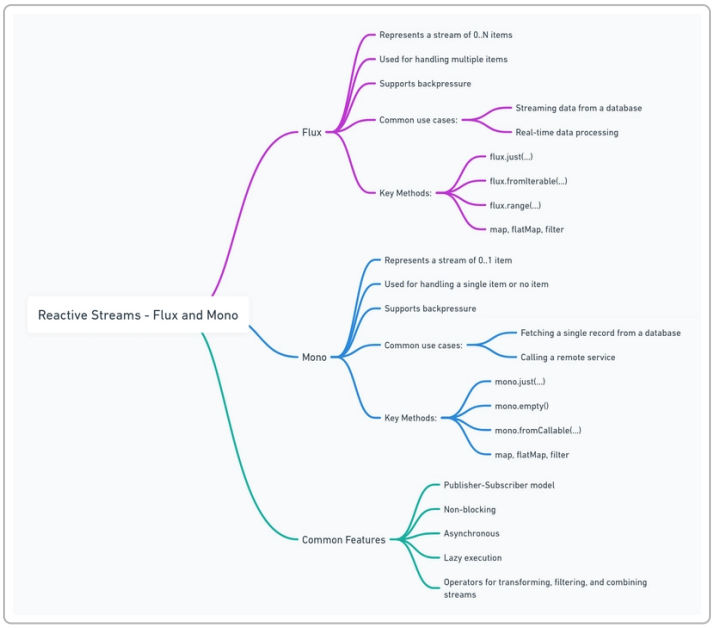
Servlet 스레드 모델 vs Netty 기반 아키텍처 (요약 도식)



Spring MVC의 전통적인 **"스레드-당-요청"** 처리와 Spring WebFlux (Netty)의 **"이벤트 루프"** 처리 흐름을 비교하면 위 그림과 같습니다. WebFlux/Netty에서는 클라이언트 요청이 들어오면 **이벤트 루프(EventLoop)**가 해당 연결을 등록하고, 읽기/쓰기 이벤트가 발생할 때마다 적절한 핸들러로 이벤트를 전달합니다. **Netty의 ChannelPipeline**을 통해 요청 데이터는 논블로킹으로 처리되며, 응답도 이벤트로 전송됩니다. 이 동안 **스레드는 어떤 작업도 블로킹하지 않고** 다음 이벤트를 기다리거나 다른 연결의 이벤트를 처리합니다. 반면 Servlet 기반 MVC에서는 요청별로 **전용 스레드**가 할당되어 Controller 로직을 수행하고, DB 질의 등의 I/O가 발생하면 그 스레드가 작업 완료까지 멈춰 있게 됩니다. 이러한 구조적 차이로 인해, **WebFlux는 적은 스레드로 높은 동시성을 달성**하며, **MVC는 직관적인 코드 작성이 가능하지만 많은 스레드 자원이 필요**합니다.

2. Mono와 Flux

WebFlux 프로그래밍의 기본 단위는 **Mono**와 **Flux**입니다. 이는 Project Reactor에서 제공하는 **Reactive Streams Publisher** 구현체로, **비동기 데이터 흐름을 표현**하는 데 사용됩니다 [9]. 각각의 정의와 활용 방법은 다음과 같습니다.



위 마인드맵은 Mono와 Flux의 개념을 비교한 것입니다. **Mono**는 0 또는 1개의 요소를 발행하는 스트림이며, **Flux**는 0개 이상N개의 요소를 발행하는 스트림을 나타냅니다. 둘 다 Publisher-Subscriber 모델을 따르는 비동기 논블로킹 스트림이며, 필요한 시점까지 실행을 지연하는 Lazy Execution 특성을 가집니다. 즉, Mono/Flux 자체만으로는 실행되지 않고, 최종적으로 구독(Subscribe)되거나 리턴되어 소비될 때 비로소 내부 로직이 수행*됩니다.

- **Mono\<T>:** 최대 **하나의 결과**(혹은 결과 없음)를 내는 비동기 시퀀스입니다. 예를 들어 **HTTP 요청 하나당 객체 한 개**를 반환하거나, **데이터베이스에서 특정 ID로 조회**하는 경우 Mono를 사용합니다. **값이 없을 수도 있는 상황**(예: 조회 결과 없음)도 Mono로 표현할 수 있습니다. WebFlux의 컨트롤러에서는 `Mono<ResponseType>` 리턴을 통해 **단일 객체를 비동기로 응답**할 수 있습니다.

- **Flux\<T>: 여러 개의 결과**를 차례로 내는 비동기 시퀀스입니다. 예를 들어 **리스트 조회**(다수의 요소)나, **스트리밍 API** (서버가 지속적으로 이벤트를 흘려보내는 경우)에서는 Flux를 사용합니다. WebFlux 컨트롤러에서 `Flux<StreamElement>`를 리턴하면 **HTTP 응답 바디를 스트리밍** 형태로 보낼 수 있습니다 (특히 `text/event-stream` SSE나 `application/stream+json` 같은 콘텐츠 타입으로 응답 시).

- **Reactive 연산자:** Mono와 Flux는 자바 스트림(`Stream API`)과 유사한 다채로운 **연산자(operator)**들을 제공합니다 9 . 이를 사용해 **비동기 데이터 흐름을 처리, 변환, 조합**할 수 있습니다. 주요 연산자는 다음과 같습니다:

- `map` – 스트림의 각 요소를 다른 형태로 **동기적으로 변환**합니다. (`Flux<String> -> Flux<Integer>` 등) 예를 들어 문자열을 해싱하여 숫자로 변환하거나, DTO를 응답용 VO로 변환할 때 사용합니다.
- `flatMap` – 각 요소를 처리하여 **또 다른 Mono/Flux를 반환하고, 그 내부 발행 요소들을 평탄화**하여 하나의 스트림으로 합칩니다. 주로 비동기 작업 (예: 다른 서비스 호출, DB쿼리 등)을 요소마다 수행해야 할 때 사용됩니다. (`Flux<A>` 에서 각 A를 처리해 `Mono<B>` 를 얻고, 결과를 모아 최종적으로 `Flux<B>` 로 받는 등). flatMap은 내부 비동기 작업들이 **병렬**로도 처리될 수 있기 때문에 (동시에 여러 요소 처리), 순서를 유지해야 하면 `concatMap` 을 사용할 수도 있습니다.
- `filter` – 조건에 맞지 않는 요소를 **걸러내는** 연산자입니다. 예를 들어 `flux.filter(x -> x.isValid())` 라고 하면 `isValid()` 가 `true` 인 요소만 내려보냅니다.

- **그 밖에** `reduce`, `concat`, `zip`, `take`, `delayElements`, `onErrorResume` 등 수십 가지 연산자가 존재합니다. 예를 들어 둘 이상의 Publisher 결과를 합치는 `zip`이나, 에러 발생 시 대체 스트림을 이어 붙이는 `onErrorResume` 등이 흔히 사용됩니다.

- **예제:** 다음 예시는 간단한 Flux 파이프라인을 보여줍니다. 문자열 목록에서 길이가 3 초과인 것만 남기고, 대문자로 바꾼 뒤, 비동기 처리로 변환하며, 에러가 발생하면 기본값으로 대체합니다. 코드에 연산자별 주석을 달았습니다:

```
Flux<String> flux = Flux.just("one", "two", "three", "four")
  .filter(s -> s.length() > 3)        // 1) 문자열 길이가 3 초과인 것만 통과 ("three", "four")
  .map(String::toUpperCase)           // 2) 각 문자열을 대문자로 변환 ("THREE", "FOUR")
  .flatMap(s -> callExternalService(s)) // 3) 각 요소를 비동기 외부 서비스 호출 결과 Flux로 변환하여 평탄화
  .onErrorResume(e -> Flux.just("DEFAULT")); // 4) 에러 발생 시 대체 값 스트림으로 복구
flux.subscribe(System.out::println);      // 5) 구독하여 스트림 실행 시작
```

위 코드에서 `filter → map → flatMap` 순으로 **데이터가 흐르며 변환**되고, 최종적으로 `subscribe`가 호출되어야 실제 실행이 이루어집니다. `onErrorResume`은 처리 과정 중 예외가 발생했을 때 `"DEFAULT"` 값을 대신 발행하도록 하는 **에러 처리** 예시입니다.

- **에러 처리:** Reactive 스트림에서는 에러도 **데이터 신호의 일종**으로 취급합니다. 연산 중 예외가 발생하면 일반적인 throw로 제어 흐름을 중단하는 대신, 스트림이 **onError 신호**를 방출하며 종료됩니다. 따라서 try-catch로 잡는 대신, **전용 에러 처리 연산자**를 통해 다룹니다. 대표적으로:
- `onErrorReturn(Fallback)` – 에러 발생 시 지정한 Fallback 값으로 **대체**하여 **정상 완료**시킵니다 (단일 값).
- `onErrorResume(error -> 다른 Flux/Mono)` – 에러 발생 시 **대체 스트림**을 생성하여 이어 붙입니다. 예를 들어 외부 서비스 호출 실패 시 캐시된 데이터를 Mono로 리턴하는 식으로 활용 가능합니다.
- `doOnError(error -> ...)` – 에러 발생 시 별도의 부가 작업(로깅 등)을 수행하고, 최종 처리는 그대로 에러로 흘려보냅니다.

- 이 외에도 `retry(n)` (에러 시 재시도), `onErrorMap` (에러를 다른 예외로 변환) 등이 있습니다. **중요한 점은, 한 번 에러 신호가 발생하면 기본적으로 그 스트림은 더 이상 진행되지 않고 종료**되므로, 필요한 경우 위와 같은 연산자로 **에러를 잡아주고 스트림을 계속** 이어가거나 대체해야 합니다.

- **백프레셔(Backpressure):** 앞서 언급했듯 Reactive Streams의 중요한 요소는 백프레셔입니다. Mono와 Flux를 구독하면 Subscriber는 **처음에 요청 가능한 요소 개수 (예:** `request(unbounded)` **또는** `request(1)` **등)**를 결정할 수 있습니다. Publisher는 Subscriber가 **요청한 수만큼의 onNext를 보내고** 더 보낼 수 없을 때까지 대기합니다 [8]. Subscriber가 처리를 마치고 추가 요소를 처리할 준비가 되면 **다시 request 신호**를 보내 더 많은 데이터를 흘려받습니다. 이 **수요-공급 조절 메커니즘** 덕분에, 예를 들어 **매우 빠른 생산자(Publisher)**가 **느린 소비자(Subscriber)**를 압도하지 않도록 제어할 수 있습니다 [8]. WebFlux 개발 시 일반적으로 백프레셔를 직접 제어할 일은 많지 않지만, 내부적으로는 이러한 흐름 제어로 **리소스 고갈을 방지**합니다. 스트림을 처리하다보면 가끔 `onBackpressureBuffer`, `onBackpressureDrop` 같은 연산자를 볼 수 있는데, 이는 **백프레셔 신호를 대응하지 못하는 생산자**의 경우 **임시로 버퍼링하거나 데이터 드롭** 등을 통해 조절하는 방법입니다 [18]. 대체로 Reactor에서 제공하는 Flux 연산자들은 기본적으로 Reactive Streams 사양을 준수하므로, 일반적인 use-case에서는 크게 신경 쓰지 않아도 됩니다. 다만 **사용자가 임의로 생성한 Flux**가 너무 빠르게 데이터를 발생시키는 경우 등에 이러한 운영자들을 사용할 수 있습니다.

요약하면, **Mono는 단일값 비동기 처리**, **Flux는 다중값 스트림 처리**에 쓰이며, 이들을 조합하는 다양한 연산자를 통해 복잡한 비동기 로직도 함수형 스타일로 구성할 수 있습니다. 처음에는 `map`, `flatMap`, `filter`부터 시작해 점차

다른 연산자를 익히는 것을 권장합니다. Reactive 연산은 **데이터 흐름 중심의 선언적 프로그래밍**이므로, 익숙해지면 병렬 호출, 순차 처리, 에러 복구 등을 비교적 적은 코드로 구현할 수 있습니다.

3. WebClient 활용

Spring WebFlux에서는 HTTP 클라이언트로 **WebClient**를 사용합니다. WebClient는 RestTemplate을 대체하는 **리액티브 HTTP 클라이언트**로, **완전한 논블로킹** 방식으로 동작합니다 [19] [20]. **외부 API 호출**이나 **마이크로서비스 간 통신** 시 WebClient를 사용하면 요청/응답을 **논블로킹 스트림으로 처리**할 수 있어, 쓰레드를 효율적으로 사용하면서도 높은 성능을 얻을 수 있습니다 [20]. WebClient의 주요 특징과 사용법은 다음과 같습니다:

- **논블로킹 & 이벤트 루프 기반:** WebClient는 내부적으로 Reactor Netty의 HTTP 클라이언트를 사용하며, **이벤트 기반 논블로킹 I/O**로 구현되어 있습니다 [20] [21]. 요청을 보내고 응답을 기다리는 동안 스레드가 블로킹되지 않고, **응답 도착 시 이벤트 콜백으로 처리**됩니다. 개발자는 `Mono`나 `Flux`로 응답을 받게 되므로, 마치 **데이터 스트림을 다루듯** 후속 처리를 정의할 수 있고, **스레드 관리나 동기화에 신경 쓸 필요가 없습니다** [20].

- **사용법 개요:** WebClient 인스턴스는 보통 애플리케이션 전역으로 하나만 만들어 재사용합니다. Spring Boot에서는 `WebClient.builder()` 빈을 주입받아 설정을 구성하거나, `WebClient.create("baseUrl")`로 간단히 생성할 수 있습니다 [22] [23].

```
@Bean
public WebClient webClient(WebClient.Builder builder) {
  return builder.baseUrl("https://api.example.com")      // 기본 베이스 URL 설정
          .defaultHeader(HttpHeaders.CONTENT_TYPE,
MediaType.APPLICATION_JSON_VALUE)  // 기본 헤더 설정 예시
          .build();
}
```

위와 같이 빈으로 등록해두고 `@Autowired`로 주입받으면, 필요 시마다 `webClient`를 이용해 요청을 보낼 수 있습니다. (`WebClient.create()`로 즉석에서 만들 수도 있지만, 커넥션 풀 등의 내부자원을 공유하기 위해 어플리케이션 범위 빈으로 두는 것이 좋습니다.)

- **요청 보내기:** WebClient는 **비동기 Fluent API**를 제공하여 요청을 구성합니다. 예를 들어 GET 요청을 보내고 JSON을 받아오는 기본 코드는 다음과 같습니다:

```
webClient.get()
      .uri("/path/{id}", idVar)     // HTTP GET, URL 및 경로변수 설정
      .accept(MediaType.APPLICATION_JSON)
      .retrieve()                   // 응답 받아오는 정책: 상태코드 에러 처리 등 기본
      .bodyToMono(MyResponse.class);  // 응답 본문을 MyResponse 타입의 Mono로 변환
```

- `method().uri()` 부분에서 HTTP 메서드와 URI를 설정합니다. `uri` 메서드는 문자열에 `{}` 플레이스홀더를 써서 경로변수 바인딩 또는 `uri(builder -> ...)` 형태로 쿼리파람 구성도 가능합니다 [24] [25].
- `accept(...)`는 서버에 보낼 `Accept` 요청 헤더를 지정합니다 (생략 시 기본 값 사용).
- `retrieve()`는 요청을 실행하여 응답을 받는 **단축 메서드**입니다 [26]. HTTP 4xx/5xx 응답은 자동으로 `WebClientResponseException`으로 onError 신호가 발생하며, 정상 2xx 응답은 body extraction 단계로 넘어갑니다 [27]. (`exchangeToMono/Flux` 등의 메서드는 lower-level 제어를 위해 사용되지만, 간단한 경우 retrieve로 충분합니다. 향후 필요 시 응답 상태별 처리로 `onStatus(...)` 등을 사용 가능 [27].)

- `bodyToMono(...)` 또는 `bodyToFlux(...)`를 통해 응답 본문을 원하는 타입으로 **직렬화 및 Publisher로 변환**합니다. Jackson 등 HttpMessageConverter가 등록되어 있으면 JSON을 객체로 변환해 줍니다. Flux로 받으면 다중 요소 스트림(예: JSON 배열 스트리밍)으로 처리하고, Mono로 받으면 단일 값으로 처리합니다.
- 이렇게 얻은 Mono/Flux는 **비동기 결과**이므로, 보통 WebFlux 컨트롤러에서는 그대로 리턴하거나, 필요하면 다른 Flux/Mono와 조합한 뒤 리턴합니다. 만약 즉시 값을 얻고자 하면 `.block()`으로 동기 처리할 수도 있지만, **WebFlux 환경에서는 block() 사용을 지양**해야 합니다 (블로킹은 스레드 낭비를 초래하여 WebFlux의 이점을 없앨 수 있습니다).

다음으로, WebClient 활용의 대표적인 예제를 살펴보겠습니다. 하나는 **외부 API의 스트리밍 응답을 받아 SSE(Server-Sent Events)로 프론트엔드에 전달**하는 경우, 또 하나는 **내부 마이크로서비스 API를 호출하여 데이터를 가져오는 경우**입니다. 코드 예시는 Spring Boot 3+ (Java 17 이상) 환경을 가정하며, **모두 논블로킹으로 동작**합니다.

3.1 외부 API 스트리밍 SSE 전달 예제

가정: 우리 애플리케이션이 **외부의 SSE(Stream) API**를 소비하여, 그 데이터를 **실시간으로 클라이언트 브라우저에 전달**해야 한다고 합시다. (예: 실시간 주식 시세, 센서 데이터 스트림 등.) WebFlux에서는 WebClient를 사용해 외부 스트림을 `Flux`로 받고, 이를 **서버 측 이벤트(Server-Sent Events)** 형태로 브라우저에 push 할 수 있습니다. 클라이언트는 JS의 `EventSource` API를 통해 SSE 스트림을 받을 수 있습니다.

구현 방식: Spring WebFlux의 컨트롤러 메서드에서 `Flux`를 반환하고 응답 Content-Type을 `text/event-stream`으로 설정하면, 해당 Flux의 각 요소가 SSE 이벤트로 클라이언트에 전송됩니다. 우리가 할 일은 WebClient로 외부 SSE를 구독하여 Flux로 얻은 뒤, 그대로 리턴하는 것입니다 [28].

```
@RestController
@RequestMapping("/stream")
public class StreamProxyController {

  private final WebClient webClient = WebClient.create();
  // 주: 실제론 WebClient를 빈 주입받고 사용. baseUrl 등 설정 가능.

  // 외부 스트리밍 API -> SSE로 브라우저에 중계
  @GetMapping(value="/external-events", produces=MediaType.TEXT_EVENT_STREAM_VALUE)
  public Flux<String> streamExternalEvents() {
    return webClient.get()
      .uri("http://external-service.example/api/events")    // 외부 SSE 제공 API URL
      .accept(MediaType.TEXT_EVENT_STREAM)                  // SSE 스트림 수신 요청
      .retrieve()
      .bodyToFlux(String.class)  // 문자열 스트림으로 수신 (필요시 도메인 타입으로 변환 가능)
      .doOnError(e -> {
        // 에러 발생 시 로그 등 처리 (onErrorResume으로 대체 스트림 연결도 가능)
        System.err.println("외부 스트림 오류: " + e.getMessage());
      });
  }
}
```

위 코드에서는 `/stream/external-events` 엔드포인트를 통해 외부의 `http://external-service.example/api/events` SSE 스트림을 받아 그 내용을 **그대로 클라이언트에게 흘려보냅니다**. 설명을 덧붙이면:

- `@GetMapping(value="/external-events", produces=MediaType.TEXT_EVENT_STREAM_VALUE)`: 응답을 **SSE(MediaType.TEXT_EVENT_STREAM)**로 지정합니다. 이 설정이 매우 중요합니다. 이 경우 Spring WebFlux는 반환하는 Flux의 데이터를 **플러시 가능한 서버 이벤트 스트림**으로 처리하여, HTTP 응답을 끊지 않고 데이터를 **스트리밍 전송**합니다 [28]. 브라우저에서는 `EventSource`를 통해 지속적인 이벤트 스트림으로 받게 됩니다.

- `webClient.get().uri(...).accept(TEXT_EVENT_STREAM)`: WebClient를 사용해 **외부 API 호출**을 구성합니다. 외부 API가 SSE를 제공한다고 가정하므로 `Accept: text/event-stream` 헤더를 보내고 있습니다. (`accept`를 명시하지 않아도 외부가 SSE일 경우 응답 헤더로 `text/event-stream`을 줄 것이지만, 명시하면 가독성이 높아집니다.) WebClient는 이 요청을 비동기로 보내며, 아직 `.subscribe()`하지 않았으므로 실제 호출은 이 Flux가 구독될 때 일어납니다.

- `.retrieve().bodyToFlux(String.class)`: 응답 스트림을 String으로 변환했습니다. 외부 이벤트가 만약 JSON이나 특정 객체라면 `bodyToFlux(MyEvent.class)`처럼 도메인 클래스 타입으로 binding할 수 있습니다. 외부 API가 `event:` / `data:` 형태로 SSE 이벤트를 보내면 Spring이 적절히 잘라서 문자열 Flux로 제공하며, JSON 문자열일 경우 Jackson으로 클래스 객체로 변환도 가능합니다.

- `return Flux`: 이렇게 얻은 Flux를 컨트롤러에서 반환하면 WebFlux는 자동으로 **Flux가 완료될 때까지** (또는 무한 스트림이면 클라이언트가 연결을 끊을 때까지) **데이터를 지속적으로 전송**합니다. 한 요소가 onNext로 올 때마다 그것을 SSE 프레임으로 직렬화해 응답 스트림에 쓰고 플러시합니다. 클라이언트는 이벤트를 실시간으로 받게 됩니다.

- 에러 처리: 위 예제에서는 `doOnError`로 에러를 로그만 하고 있습니다. 외부 스트림 연결에 오류가 생기면 SSE 스트림은 onError와 함께 **종료**되고 클라이언트 연결도 닫힙니다. 필요하다면 `.onErrorResume`을 사용해 에러 발생 시 대체 Flux (예: 재접속 시도나, 완료되지 않은 스트림에 대한 종료 이벤트 전송 등)를 반환할 수도 있습니다.

SSE 서버 & 클라이언트 측 참고: SSE는 **단방향 서버->클라이언트 스트림**입니다. 클라이언트(브라우저)는 JavaScript의 `new EventSource("/stream/external-events")`로 연결을 열고, 메시지를 수신합니다. 한편 서버(WebFlux)는 위와 같이 `Flux`를 SSE로 반환하기만 하면 구현이 완료됩니다. Spring WebFlux는 내부적으로 **응답을 플러시모드로 전환**하고, `\n\n` 등 SSE 프로토콜에 맞게 데이터를 구분지어 전송해줍니다 [29]. 별도의 SSE 관리 객체(`SseEmitter`) 등을 수동으로 쓸 필요가 없다는 점이 WebFlux의 편리한 기능입니다.

3.2 내부 서비스 API 호출 예제

이번에는 **마이크로서비스 간 통신** 시 WebClient 사용 예를 살펴보겠습니다. 예를 들어 사용자의 상세 정보를 구성하기 위해 **User 서비스**와 **Order 서비스**에서 데이터를 동시에 가져와 결합하여 응답하는 시나리오를 생각해보겠습니다. Spring MVC에서는 RestTemplate으로 순차 호출하거나, 컴바이너 쓰레드풀로 병렬 실행을 해야 했겠지만, WebFlux+WebClient 조합에서는 **논블로킹으로 병렬 호출 결과를 조합(zip)**할 수 있습니다. 아래 코드는 `user-service`에서 사용자 정보를 받고 `order-service`에서 주문 목록을 받아 합쳐 응답하는 컨트롤러 예시입니다:

```
@RestController
@RequestMapping("/compose")
```

```
public class ComposeController {

  private final WebClient webClient = WebClient.create();
  // 실제 구현에서는 각 서비스별로 WebClient를 분리하거나, Builder로 주입 받아 사용

  // 내부 User 서비스와 Order 서비스의 응답을 병렬 호출하여 조합
  @GetMapping("/user/{userId}/details")
  public Mono<UserDetails> getUserDetails(@PathVariable Long userId) {
    Mono<User> userMono = webClient.get()
      .uri("http://user-service/api/users/{id}", userId)  // User 서비스 호출
      .retrieve()
      .bodyToMono(User.class);                    // 결과를 User 객체로 받음 (Mono)

    Mono<List<Order>> ordersMono = webClient.get()
      .uri("http://order-service/api/users/{id}/orders", userId)  // Order 서비스 호출
      .retrieve()
      .bodyToFlux(Order.class)                    // Order 여러건 Flux로 수신
      .collectList();                         // Flux<Order> -> Mono<List<Order>>로 변환

    // 두 Mono 병합하여 UserDetails 객체 생성
    return Mono.zip(userMono, ordersMono,
            (user, orders) -> new UserDetails(user, orders));
  }
}
```

위 구현을 설명하면 다음과 같습니다:

- `webClient.get().uri("http://user-service/api/users/{id}", userId).retrieve().bodyToMono(User.class)`: **내부 User 서비스**의 REST API를 호출하여 `User` 정보를 Mono로 받아옵니다. (예시 URI는 `user-service`라는 서비스명 DNS를 사용하고, 경로는 가정입니다.)

- `webClient.get().uri("http://order-service/api/users/{id}/orders", userId).retrieve().bodyToFlux(Order.class).collectList()`: **내부 Order 서비스**의 API를 호출하여 해당 사용자의 주문 목록을 Flux로 받고, `collectList()`를 통해 Mono<List<Order>>로 변환합니다. 즉 **여러 개의 Order**를 한 번에 리스트로 모읍니다. (만약 Order 수가 매우 많아 스트림으로 처리해야 한다면 collectList를 쓰지 않고 Flux 자체를 사용하여 SSE로 내려보내는 것도 고려할 수 있습니다. 여기서는 UserDetails 객체에 주문 리스트를 포함시키기 위해 리스트로 모았습니다.)

- `Mono.zip(userMono, ordersMono, (user, orders) -> new UserDetails(user, orders))`: 두 개의 Mono를 **zip** 연산으로 합칩니다. 이는 각 Mono에서 값이 나오면 (`user`, `orders`) 두 값을 합쳐 새로운 `UserDetails` 객체를 생성하는 람다를 적용합니다. **중요한 점은, userMono와 ordersMono가 모두 완료될 때까지 대기했다가 둘 다 값이 준비되면 UserDetails를 만든다는 것**입니다. `zip`은 내부적으로 **두 호출을 병렬로 실행**합니다. `userMono`와 `ordersMono`는 각자 WebClient를 통해 비동기 요청을 날린 상태이고, 둘 다 응답이 오면 zip의 조합 함수가 호출되어 결과 Mono가 완료됩니다. 따라서 **두 내부 API 호출이 병렬 수행되어 전체 응답 시간이 단축**됩니다.

- 컨트롤러 메서드는 최종적으로 `Mono<UserDetails>`를 리턴하므로, Spring WebFlux는 이 Mono를 subscribe하여 결과가 나오면 JSON으로 직렬화해 HTTP 응답으로 반환합니다. 마치 동기적으로

UserDetails를 반환하는 것처럼 보이지만, 실제로는 **비동기로 두 서비스를 호출하여 데이터를 모아온 후 응답**하는 것입니다.

- 에러 처리: 위 예제에서는 따로 예외 처리를 하지 않았습니다. `retrieve()` 사용 시 HTTP 4xx/5xx 에러는 `userMono`나 `ordersMono`에서 onError 신호로 처리됩니다[27]. 둘 중 하나라도 에러가 나면 zip 전체가 에러로 종료되어 요청은 오류 응답으로 반환될 것입니다. 필요한 경우 `.onErrorResume` 등을 각 Mono에 체인해서 일부 서비스 오류 시 fallback 데이터를 쓰거나, zip 바깥에서 처리할 수 있습니다. 예를 들어 user 서비스 실패시 orders만으로 응답한다든지 하는 로직도 가능하지만, 여기서는 기본 동작에 맡깁니다.

  참고: WebClient를 여러 서비스 호출에 활용할 때 공통 설정(예: 공통 baseUrl, 헤더 등)을 서비스별로 분리하면 편리합니다. 예를 들어 User 서비스 전용 WebClient 빈(`WebClient.builder().baseUrl("http://user-service").build()`)과 Order 서비스 전용 WebClient를 각각 만들어서 사용하면 `uri` 작성이 단순해지고 코드의 **의존 서비스가 명확해집니다**. 또한 마이크로서비스 환경에서 **타임아웃**이나 **재시도** 같은 설정도 WebClient `builder` 단계나 `exchange().timeout(...)` 등으로 적용할 수 있습니다. WebClient는 비동기 논블로킹이므로, **동기 RestTemplate처럼 timeouts를 꼭 걸지 않아도 스레드 붕괴는 없지만**, 응답 지연에 대한 별도 대응이 필요하면 Reactor 연산자 (`timeout`, `retryWhen` 등)를 적용하면 됩니다.

以上의 WebClient 예제들은 **Spring WebFlux 애플리케이션에서 외부와 통신하는 방식**을 잘 보여줍니다. **정리하면:**

- WebFlux에서는 **외부 API 호출조차도 논블로킹 비동기로 처리**하므로, 다중 서비스 호출을 합치거나 스트림을 중계하는 등의 작업을 해도 **주요 스레드가 블로킹되지 않으며**, 높은 성능과 응답성을 유지할 수 있습니다[19].
- **코드 면에서도** Reactive 연산자 (`zip`, `flatMap` 등)을 활용하면 복잡한 비동기 로직을 비교적 간결하게 구현할 수 있습니다. 다만 **가독성**을 위해서는 적절한 메서드 분리와 주석이 필요하며, 팀원들이 Reactive 스타일에 익숙해지는 시간이 필요할 수 있습니다.
- **에러 전파 및 처리**가 기존 MVC와 다르므로, WebClient 호출에서 발생한 에러가 최종 사용자에게 어떻게 전달될지 (예: 500 오류) 이해하고, 필요시 onStatus, onErrorResume 등으로 별도 처리해야 합니다. 또한 `subscribe()`를 직접 호출하면 Spring MVC에서 비동기 결과를 처리하듯 콜백을 작성해야 하는데, **일반적으로는 컨트롤러에서는 subscribe하지 않고 Mono/Flux를 리턴하는 것으로 충분**합니다. subscribe는 최종적으로 WebFlux 프레임워크가 해주므로, 개발자가 임의로 호출하지 않도록 주의합니다.

4. 결론 및 정리

이 핸즈온 자료를 통해 **Spring WebFlux의 비동기 논블로킹 원리와 주요 구성요소(Mono, Flux, WebClient)**를 살펴보았습니다. **정리하면 다음과 같습니다:**

- Spring WebFlux는 **Reactive Streams** 기반으로 동작하여 **스레드 효율성**과 **고성능 I/O 처리**를 가능케 합니다. 반면 코드는 선언적/비동기 스타일이라 **학습 곡선**이 존재하므로, 기존 MVC 대비 충분한 연습과 팀 합의가 필요합니다.
- **Mono와 Flux**는 WebFlux의 반환 타입으로 쓰이며, 각각 **단일값**과 **다중값 스트림**을 표현합니다. `map`, `flatMap`, `filter` 등의 **연산자**를 사용해 데이터 흐름을 처리하며, **에러와 백프레셔** 개념을 내재하고 있습니다. 이를 잘 활용하면 복잡한 비동기 로직도 비교적 쉽게 구현할 수 있습니다.
- **WebClient**는 **외부 서비스 연동**에 필수적인 논블로킹 HTTP 클라이언트입니다. SSE 같은 스트리밍부터 다중 서비스 호출 병렬화까지 유용하게 활용할 수 있으며, WebFlux 환경에서는 RestTemplate 대신 WebClient를 사용해야 진정한 **end-to-end 리액티브**가 완성됩니다.
- 본문의 코드 예제들은 **Spring Boot 3** 기준 Java 코드로 작성되었으며, 실제 프로젝트에서 바로 사용하거나 응용할 수 있습니다. 코드를 보며 직접 따라해보는 **핸즈온**을 통해, WebFlux에 대한 이해를 높이고 프로젝트에 적용해 보시기 바랍니다.

마지막으로, WebFlux 도입 시 **현재 시스템의 특성**을 고려해야 합니다. **CPU 바운드 작업이 대부분**인 경우 WebFlux로 얻는 이점은 크지 않을 수 있고, **블로킹 API(JDBC 등)**를 함께 사용하면 별도 스케줄러로 전환하는 부가 작업이 필요합니다. 하지만 **대규모 동시성, 스트리밍 데이터, 마이크로서비스 간 비동기 통신**이 중요한 시스템이라면 WebFlux는 Spring 생태계 안에서 이를 구현하기에 매우 강력한 도구가 될 것입니다. **기존 MVC 지식에 Reactive 패러다임을 더해**, 필요한 곳에 WebFlux를 활용해보세요. 📱

참고 자료: Spring 공식 문서, Baeldung 튜토리얼, Dev.to 블로그 등 [5] [4] [9] [28] [19] (각주 번호는 본문 해당 내용에 연결된 자료 출처를 나타냅니다). 성공적인 WebFlux 활용을 기원합니다!

---

[1] [3] [5] [6] [15] Spring MVC vs. Spring WebFlux: Choosing the Right Framework for Your Project - DEV Community
https://dev.to/jottyjohn/spring-mvc-vs-spring-webflux-choosing-the-right-framework-for-your-project-4cd2

[2] [7] java - Spring Web MVC vs Spring WebFlux. Blocking and Non-blocking - Stack Overflow
https://stackoverflow.com/questions/70997077/spring-web-mvc-vs-spring-webflux-blocking-and-non-blocking

[4] Threading model of Spring WebFlux and Reactor - GeeksforGeeks
https://www.geeksforgeeks.org/advance-java/threading-model-of-spring-webflux-and-reactor/

[8] [10] [11] [18] Reactive Streams in Java: Using Project Reactor - DEV Community
https://dev.to/tutorialq/reactive-streams-in-java-using-project-reactor-d0h

[9] [12] [13] [14] [16] [17] Overview :: Spring Framework
https://docs.spring.io/spring-framework/reference/web/webflux/new-framework.html

[19] [22] [23] [26] [27] [28] Get Your Microservices Reactive. In the last blog post I described how… | by Rareş Popa | Medium
https://medium.com/@rarepopa_68087/get-your-microservices-reactive-3c355aae4095

[20] [21] [24] [25] Efficient API Communication with Spring WebClient | by Taras Ivashchuk | Medium
https://medium.com/@ia_taras/efficient-api-communication-with-spring-webclient-5c5dea18a6ba

[29] Server-Sent Events (SSE) in Spring WebFlux - GeeksforGeeks
https://www.geeksforgeeks.org/advance-java/server-sent-events-sse-in-spring-webflux/