Spring

Web

Flux

In normal programming when we do compute intensive tasks or I/O intensive tasks, the precious treads are in waiting

Reactive manifesto outlines 4 important principals site

1. Responsive
2. Resilient
3. Message Driven
4. Elastic

Spring web flux communication patterns

1. Request, response
2. Request, Streaming response
3. Streaming request, response
4. Bidirectional streaming

# Basic

## Traditional API VS Web flux

Traditional REST behavior

curl <http://localhost:8080/traditional/products>

* It takes 10 seconds.
* Response comes when all 10 records are generated
* Even if we cancel the requests the backend traditional service is still processing the request
* On cancel since the traditional service is not stopping the external service running on 7070 is still processing. So its cascading wastage of resource

Web Flux Behavior

curl <http://localhost:8080/reactive/products>

* It takes 10 seconds.
* Response comes when all 10 records are generated. We can get streaming response on console by disabling buffer with curl -N <http://localhost:8080/reactive/products>
* When we cancel the requests the backend traditional service stops
* Backend service on 7070 also stops

We can see streaming behavior in Browser with streaming endpoint <http://localhost:8080/reactive/products/stream> which has below end MediaType defined. By default it uses Application/Json hence it waits for response

*@GetMapping*(value = "products/stream", produces = MediaType.TEXT\_EVENT\_STREAM\_VALUE)  
*public* Flux<Product> getProductsStream() {

Error handling

On our backend service running on 7070 we have an endpoint /demo01/products/notorious which will crash. We will configure that endpoint with <http://localhost:8080/reactive/products> and <http://localhost:8080/traditional/products> like below

ReactiveWebController.java

@GetMapping("products")  
public Flux<Product> getProducts() {  
 return this.webClient.get()  
 //.uri("/demo01/products")  
 .uri("/demo01/products/notorious")  
 .retrieve()  
 .bodyToFlux(Product.class)  
 .onErrorComplete()  
 .doOnNext(p -> log.info("received: {}", p));  
}

TraditionalWebController.java

@GetMapping("products")  
public List<Product> getProducts() {  
 var list = this.restClient.get()  
 //.uri("/demo01/products")  
 .uri("/demo01/products/notorious")  
 .retrieve()  
 .body(new ParameterizedTypeReference<List<Product>>() {  
 });  
 log.info("received response: {}", list);  
 return list;  
}

After configuring when we hit these endpoints, we see that:

1. In traditional API it will fail with 500 with no response
2. In Web flux it will give partial response.

[{"id":1,"description":"product-1","price":1},{"id":2,"description":"product-2","price":2},{"id":3,"description":"product-3","price":3},{"id":4,"description":"product-4","price":4}curl: (18) transfer closed with outstanding read data remaining

This is not proper JSON. We can handle this easily

[{"id":1,"description":"product-1","price":1},{"id":2,"description":"product-2","price":2},{"id":3,"description":"product-3","price":3},{"id":4,"description":"product-4","price":4}]

## How web flux works

**1. Reactive Programming & Publisher-Subscriber Pattern**

* **Reactive programming** is based on the **Publisher-Subscriber model**.
  + **Publisher**: Emits data (e.g., Flux, Mono in Reactor).
  + **Subscriber**: Consumes data reactively (e.g., browser, Postman, another service).
* In reactive microservices, everything is visualized as either a **publisher** or a **subscriber**.

**2. Example: Browser (Subscriber) → Backend (Publisher)**

* **Scenario**: A browser requests data from a Spring WebFlux backend.
  + **Backend (Publisher)**: Exposes an API (returns Flux<Product>).
  + **Browser (Subscriber)**: Sends an HTTP request (implicitly subscribes).
* **Key Point**: The browser doesn’t explicitly "subscribe." Instead, the **Spring framework subscribes on its behalf** when the request arrives.

**3. How Reactive Execution Works**

**Step-by-Step Flow:**

1. **Browser sends a request** (e.g., GET /products).
2. **Spring WebFlux**:
   * Accepts the TCP connection.
   * Routes the request to the controller.
   * Sees the controller returns a Flux<Product> (a Publisher).
   * **Subscribes to the Flux** (triggering execution).
3. **Reactive Pipeline**:
   * The controller’s Flux is lazy—no data is fetched until subscription.
   * On subscription, WebClient (non-blocking HTTP client) sends a request to a remote service.
   * Responses are streamed back incrementally via Flux.
4. **Data Streaming**:
   * As each Product arrives, Spring writes it to the HTTP response (chunked transfer encoding).
   * The browser receives data **incrementally** (no waiting for all 10 items).
5. **Cancellation Handling**:
   * If the browser closes the connection, Spring detects it and **cancels the subscription**.
   * This propagates to WebClient, stopping further requests (efficient resource cleanup).

**4. Traditional vs. Reactive Comparison**

| **Aspect** | **Traditional (Blocking)** | **Reactive (Non-Blocking)** |
| --- | --- | --- |
| **Return Type** | List<Product> (sync) | Flux<Product> (async stream) |
| **Execution** | Blocks until all data is fetched | Streams data incrementally |
| **Client Cancellation** | Wastes resources (no early exit) | Immediate cancellation possible |
| **Responsiveness** | Slow (waits for full response) | Fast (streams partial responses) |

**5. Key Tools in Reactive Spring**

* **WebClient**: Non-blocking HTTP client (wrapper around Reactor Netty).
* **R2DBC**: Reactive database driver (alternative to blocking JDBC).
* **Flux/Mono**: Publishers representing async streams (0..N or 0..1 items).

**6. Real-World Analogy: ChatGPT Streaming**

* ChatGPT uses **text/event-stream** to stream responses incrementally.
  + Users see partial answers immediately.
  + They can **cancel early** if the response isn’t useful.
* Reactive systems work similarly: **responsiveness** and **efficiency** are prioritized.

**Why This Matters**

* **Efficiency**: No threads blocked waiting for I/O (scales better).
* **Responsiveness**: Clients get data as soon as it’s available.
* **Resource Optimization**: Cancellation stops work immediately (no wasted effort).

Sec 2-11

# Spring Data R2DBC

Sec 3-14

Sec 3-17