Spring

Web

Flux

In normal programming when we do compute intensive tasks or I/O intensive tasks, the precious threads 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

# Section 2 Traditional API VS Web flux sec01

Traditional REST behaviour

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 Behaviour

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 behaviour 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 with .onErrorComplete()

[{"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).

**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).

## Reactive stack

**✅ Should the Entire Stack Be Reactive in a Spring WebFlux Application?**

**1. The Question:**

When using **Spring WebFlux**, should the entire stack be reactive, or can some parts remain synchronous?

**Short Answer:**

* Ideally, the **entire stack should be reactive** to fully leverage the benefits of reactive programming, such as streaming, backpressure handling, and non-blocking I/O.
* However, in a real-world scenario, it is common to have a **hybrid stack** where some parts are reactive, and others remain synchronous.

**2. Real-World Scenario:**

Imagine an architecture with **multiple applications**:

* **App1:** Migrated to Spring WebFlux and is fully reactive.
* **App2:** Still using traditional Spring MVC (synchronous).

If App1 is reactive and App2 is still synchronous, will it cause issues?

* ✅ **No, it won’t cause issues.**
* App1 can operate using a non-blocking, reactive model, making more efficient use of system resources (e.g., CPU and memory).
* App2 will continue to operate in a traditional blocking manner without affecting App1.

**3. Benefits of Partial Migration to Reactive:**

* If only one service is migrated to **Spring WebFlux**, that service can still benefit from:
  + **Non-blocking I/O:** Efficient resource usage during network calls, database operations, etc.
  + **Streaming and Backpressure:** Ability to handle large data streams with better resource management.
* The other synchronous services will continue operating as they did before, without any impact.

**4. Challenges with Partial Migration:**

* **Inter-Service Communication:**
  + If a reactive service (App1) calls a synchronous service (App2), it may need to **wrap synchronous calls** in a reactive wrapper (e.g., Mono.fromCallable() or Mono.fromFuture()).
* **Potential Thread Blocking:**
  + If the reactive service (App1) interacts with a blocking database or legacy synchronous APIs, the **reactive thread can still be blocked**, reducing the benefits of reactive programming.

**5. Gradual Migration Strategy:**

* Migrate individual services to **Spring WebFlux** one by one.
* Implement reactive patterns within each service while ensuring compatibility with synchronous services.
* As more services migrate to reactive, the overall architecture will gradually gain more of the benefits of a fully reactive stack.

**6. Key Takeaway:**

* Reactive programming is not an **all-or-nothing** approach.
* Start with one service, optimize it, and then gradually migrate other services.
* As more services become reactive, the system will progressively benefit from reduced resource usage, backpressure handling, and streaming capabilities.

# Section3- Spring Data R2DBC sec02

## What is R2DBC?

* **R2DBC (Reactive Relational Database Connectivity)** is a specification designed for **reactive programming** with relational databases.
* It is like JDBC but built specifically for asynchronous, non-blocking interactions with databases using Mono and Flux.

**2. How is R2DBC Different from JPA?**

| **Aspect** | **JPA (Java Persistence API)** | **R2DBC (Reactive Relational Database Connectivity)** |
| --- | --- | --- |
| Programming Model | Synchronous (Blocking I/O) | Asynchronous (non-blocking I/O) |
| Entity Mapping | Supports complex mappings like @OneToMany, @ManyToMany, @Cascade | Does **not** support complex mappings like @OneToMany, @ManyToMany |
| Backpressure | Not applicable | Supports backpressure to handle data flow control |
| Specification | JPA Specification | R2DBC Specification |
| Typical Libraries | Hibernate, EclipseLink, etc. | Postgres R2DBC Driver, MySQL R2DBC Driver, etc. |

**3. Why R2DBC Avoids Complex Mappings?**

* R2DBC prioritizes **performance, scalability, and streaming with backpressure**.
* Complex mappings like OneToMany and ManyToMany can **easily lead to the N+1 query problem**, causing severe performance issues.
* Instead of relying on these mappings, R2DBC encourages developers to use **direct queries and handle data composition manually**, thus providing more control and predictability.

**4. Supported Databases for R2DBC:**

As of now, R2DBC has drivers for several relational databases:

* **PostgreSQL**
* **MySQL**
* **MariaDB**
* **SQL Server**
* **H2 Database**
* **Oracle**
* **Cloud Spanner**

Initially, many of these drivers were not production-ready, but now they have matured and are **suitable for production use**.

**5. What is Spring Data R2DBC?**

* **Spring Data R2DBC** is a Spring module that provides a higher-level abstraction over the R2DBC specification.
* It follows the same pattern as **Spring Data JPA**, making it easier for developers familiar with Spring Data JPA to transition to reactive database access.

**6. How Spring Data R2DBC Simplifies Development:**

* In Spring Data JPA, we create repositories like this:

*@Repository  
public interface* CustomerRepository *extends* ReactiveCrudRepository<Customer, Integer> {  
 Flux<Customer> findByName(String name);  
 Flux<Customer> findByEmailEndingWith(String email);  
}

* In Spring Data R2DBC, the structure is almost identical but uses **Mono and Flux** instead of List:
* The key difference is the **return type**:
  + For a **single record**, use Mono<T>.
  + For **multiple records**, use Flux<T>.

**7. Example with Spring Data R2DBC:**

**Customer Entity:**

*@Table*("customer")  
*public class* Customer {  
  
 *@Id  
 private* Integer id;  
  
 *@Column*("name")  
 *private* String name;  
 *private* String email;

}

**Customer Repository:**

*@Repository  
public interface* CustomerRepository *extends* ReactiveCrudRepository<Customer, Integer> {  
 Flux<Customer> findByName(String name);  
 Flux<Customer> findByEmailEndingWith(String email);  
}

**8. R2DBC vs. JPA in Real-World Scenarios:**

* **If you need to handle streaming data or large data sets,** R2DBC is a better choice because it is non-blocking and supports backpressure.
* **If your application relies heavily on complex relationships and cascading operations,** JPA/Hibernate might be more convenient as it provides built-in support for those mappings.

**Summary:**

* R2DBC is designed for **reactive programming**, focusing on non-blocking, asynchronous database interactions.
* Unlike JPA, it **does not support complex relationships**, but this is intentional to avoid common pitfalls like the N+1 query problem.
* Spring Data R2DBC provides a familiar repository pattern, but with **Mono and Flux as return types**, enabling reactive data processing.

## Basic R2DBC query

Lec01CustomerRepositoryTest

*@Test  
public void* findAll() {  
 *this*.repository.findAll()  
 .doOnNext(c -> log.info("{}", c))  
 .as(StepVerifier::create)  
 .expectNextCount(10)  
 .expectComplete()  
 .verify();  
}

*@Test  
public void* findById() {  
 *this*.repository.findById(2)  
 .doOnNext(c -> log.info("{}", c))  
 .as(StepVerifier::create)  
 .assertNext(c -> Assertions.assertEquals("mike", c.getName()))  
 .expectComplete()  
 .verify();  
}

## Pagination

Lec02ProductRepositoryTest

**Objective:**

* Implement pagination in Spring Data **R2DBC**.
* Retrieve data in chunks (pages) using the Pageable interface, similar to how it is done in Spring Data JPA.

**✅ Why Pagination?**

* When dealing with large datasets (e.g., millions of records), fetching all records at once can consume significant memory and processing time.
* Pagination allows you to **retrieve subsets of data**, e.g., 10 records at a time, reducing memory overhead and enhancing performance.

**✅ Spring Data R2DBC Pagination Overview:**

* **R2DBC** does not natively support pagination the way JPA does, but Spring Data provides the Pageable interface and the PageRequest class.
* These are similar to the ones used in JPA, allowing developers to **specify the page size, page number, and sorting criteria**.

ProductRepository

Flux<Product> findBy(Pageable pageable); *// pagination*

Lec02ProductRepositoryTest

*@Test  
public void* pageable() {  
 *this*.repository.findBy(PageRequest.of(0, 3).withSort(Sort.by("price").ascending()))  
 .doOnNext(p -> log.info("{}", p))  
 .as(StepVerifier::create)  
 .assertNext(p -> Assertions.assertEquals(200, p.getPrice()))  
 .assertNext(p -> Assertions.assertEquals(250, p.getPrice()))  
 .assertNext(p -> Assertions.assertEquals(300, p.getPrice()))  
 .expectComplete()  
 .verify();  
}

**Explanation of the Test:**

* **First Page:**
  + PageRequest.of(0, 3) — First page (page index 0), 3 records per page.
  + Expected prices: 200.0, 250.0, 300.0.
* **Second Page:**
  + PageRequest.of(1, 3) — Second page (page index 1), 3 records per page.
  + Expected prices: 400.0, 450.0, 500.0.

**✅ Key Concepts in Pagination:**

* **Page Numbering:**
  + PageRequest.of(0, 3) refers to the **first page**.
  + PageRequest.of(1, 3) refers to the **second page**.
  + Page numbering starts at 0.
* **Sorting:**
  + Sorting is done using the .withSort() method in PageRequest.

PageRequest.of(0, 3).withSort(Sort.by("price").ascending()))

* **Pageable vs. Sort:**
  + Pageable can handle both **pagination and sorting**, whereas Sort handles only sorting.

**✅ Comparison with JPA Pagination:**

| **Feature** | **Spring Data JPA** | **Spring Data R2DBC** |
| --- | --- | --- |
| Annotation | @Query with Pageable | @Query with Pageable |
| Return Type | Page<T> | Flux<T> |
| Repository Method | findAll(Pageable pageable) | findAllBy(Pageable pageable) |
| Sorting Support | Yes | Yes |

**✅ Important Considerations:**

* **Memory Consumption:** Fetching large datasets without pagination can cause memory overload.
* **Performance:** Fetching smaller data chunks reduces database load and improves application responsiveness.
* **Scalability:** Implementing pagination is crucial for scalable, reactive applications that handle high data volumes.

## Join native Queries

Lec03CustomerOrderRepositoryTest

Repository

*@Query*("""  
 SELECT  
 p.\*  
 FROM  
 customer c  
 INNER JOIN customer\_order co ON c.id = co.customer\_id  
 INNER JOIN product p ON co.product\_id = p.id  
 WHERE  
 c.name = :name  
 """)  
Flux<Product> getProductsOrderedByCustomer(String name);

Lec03CustomerOrderRepositoryTest

*@Test  
public void* productsOrderedByCustomer() {  
 *this*.repository.getProductsOrderedByCustomer("mike")  
 .doOnNext(p -> log.info("{}", p))  
 .as(StepVerifier::create)  
 .expectNextCount(2)  
 .expectComplete()  
 .verify();  
}

**✅ Understanding the Execution Flow:**

1. **Query Execution:**
   * The query is executed when getProductsOrderedByCustomer() is invoked.
   * The parameter customerName is dynamically bound to the query using the :customerName placeholder.
2. **Query Mapping:**
   * Spring Data R2DBC automatically maps the query results to the Product entity based on field names.
3. **Reactive Stream Handling:**
   * The query returns a Flux<Product>, allowing reactive processing of multiple records.

**✅ Key Takeaways:**

* **Database Access without Repositories:**
  + By using R2DBC queries directly, we can execute custom SQL without relying on repository methods.
* **Named Parameter Binding:**
  + The parameter :customerName is bound using a colon (:) prefix, allowing for cleaner query structures.
* **Reactive Processing:**
  + Reactive types (Flux, Mono) enable non-blocking data access and stream processing.
* **Testing Strategy:**
  + StepVerifier is used to assert the results and verify the behavior of reactive streams.

## Projections

Lec03CustomerOrderRepositoryTest

**Projection:** Only the required fields are selected, reducing data transfer overhead.

**✅ Projection and DTO:**

* The output is a combination of fields from different tables, forming a single "view" or "projection."
* Instead of fetching entire entities, we fetch only the necessary fields and map them into a DTO class.
* With Java 17, we can use a record for this DTO.

*public record* OrderDetails(UUID orderId, String customerName, String productName, Integer amount,   
 Instant orderDate) {  
}

**Repository Method:**

* A custom query method is added to the repository to fetch these OrderDetails based on a product description.
* The method uses a JPQL/SQL query with a SELECT statement that joins multiple tables and retrieves specific fields.

*@Query*("""  
 SELECT  
 co.order\_id,  
 c.name AS customer\_name,  
 p.description AS product\_name,  
 co.amount,  
 co.order\_date  
 FROM  
 customer c  
 INNER JOIN customer\_order co ON c.id = co.customer\_id  
 INNER JOIN product p ON co.product\_id = p.id  
 WHERE  
 p.description = :description  
 ORDER BY co.amount DESC  
 """)  
Flux<OrderDetails> getOrderDetailsByProduct(String description);

**Test Method:**

* A test method is created to verify the query.
* The test sets up data using a product name like "iPhone 20".
* It then validates that the results are correctly ordered by amount in descending order.

Lec03CustomerOrderRepositoryTest

*@Test  
public void* orderDetailsByProduct() {  
 *this*.repository.getOrderDetailsByProduct("iphone 20")  
 .doOnNext(dto -> log.info("{}", dto))  
 .as(StepVerifier::create)  
 .assertNext(dto -> Assertions.assertEquals(975, dto.amount()))  
 .assertNext(dto -> Assertions.assertEquals(950, dto.amount()))  
 .expectComplete()  
 .verify();  
}

## Database client

Lec04DatabaseClientTest

**✅ Objective:**

* Execute a SQL query directly using DatabaseClient without a repository.
* Map the query result to a custom data transfer object (DTO).
* Demonstrate input binding using .bind() and output mapping using .map().

*@Autowired  
private* DatabaseClient client;  
  
*@Test  
public void* orderDetailsByProduct() {  
 *var* query = """  
 SELECT  
 co.order\_id,  
 c.name AS customer\_name,  
 p.description AS product\_name,  
 co.amount,  
 co.order\_date  
 FROM  
 customer c  
 INNER JOIN customer\_order co ON c.id = co.customer\_id  
 INNER JOIN product p ON co.product\_id = p.id  
 WHERE  
 p.description = :description  
 ORDER BY co.amount DESC  
 """;  
 *this*.client.sql(query)  
 .bind("description", "iphone 20")  
 .mapProperties(OrderDetails.*class*)  
 .all()  
 .doOnNext(dto -> log.info("{}", dto))  
 .as(StepVerifier::create)  
 .assertNext(dto -> Assertions.assertEquals(975, dto.amount()))  
 .assertNext(dto -> Assertions.assertEquals(950, dto.amount()))  
 .expectComplete()  
 .verify();  
}

**✅ Key Concepts and Benefits:**

1. **Direct SQL Execution:**
   * DatabaseClient allows for executing arbitrary SQL queries without the need for a repository or JPA entities.
2. **Input Binding:**
   * The .bind() method is used to set input parameters dynamically, allowing for parameterized queries.
3. **Mapping Results:**
   * The .map() method allows mapping each row to a specific DTO object, providing flexibility in handling the result set.
4. **Reactive Programming:**
   * The query result is returned as a Flux<OrderDetails>, allowing for reactive processing of data streams.

**✅ Advantages of Using DatabaseClient:**

* Flexibility in executing any SQL (SELECT, INSERT, UPDATE, DELETE).
* Direct access to raw SQL, useful for complex queries.
* Reactive support, allowing non-blocking data processing.
* No requirement for entity mapping, reducing overhead for simple projections.

# Section4 – R2DBC vs JPA/JDBC

## Set up

This explanation is about a comparative testing setup for **Spring Data JPA vs. Spring Data R2DBC (Reactive Database Client)** to measure **resource efficiency and throughput performance**. Let's break it down:

**✅ Context and Setup:**

* We have a different project set up to demonstrate a comparison between **Spring Data JPA and Spring Data R2DBC**.
* Section 04-r2dbc-vs-jdbc
* The goal is to evaluate two metrics:
  1. **Resource Efficiency:** How much system resources (like memory and CPU) are used.
  2. **Throughput:** How many tasks can be executed per unit of time.

**✅ Project Structure:**

* The project is structured with two separate Maven modules:
  + **Traditional Module:** Uses **Spring Data JPA**.
  + **Reactive Module:** Uses **Spring Data R2DBC**.
* **Database Setup:**
  + A single **Postgres database** is used.
  + Contains one table: customer with 10 million records.
  + Data insertion and table creation are handled via a Docker container.

**✅ Testing Methodology:**

1. **Throughput Test:**
   * The goal is to perform 100,000 tasks by querying customers by ID (customer 1, customer 2, ... customer 100000).
   * The test is run **10 times** to account for JVM warm-up and to observe consistency.
   * Key points:
     + **Reactive Module:** Uses Flux.range() to simulate 100,000 tasks and flatMap() to handle concurrent processing.
     + **Traditional Module:** Since JPA is blocking, a thread pool (ExecutorService) is used to simulate concurrent requests.
     + Executor pool size is set to **256 threads**, aligning with Reactor's default behavior.
2. **Efficiency Test:**
   * The goal is to **fetch all 10 million records** using a single SELECT \* FROM customer query.
   * In the **reactive module**, data is streamed using Flux, allowing processing of records as they arrive without holding them all in memory.
   * In the **traditional module**, the findAll() method returns a List<Customer>, which requires holding all records in memory at once.

## The test

We conduct performance tests to compare the throughput and memory efficiency of two Spring Boot modules: a reactive module using **R2DBC** and a traditional module using **JDBC**.

**Setup:**

* The database is started using **Docker Compose**, ensuring that it is ready to accept connections.
* The project is built using mvn clean package, generating JAR files for both the reactive and traditional modules.
* The tests are run using a Makefile with specific targets for throughput and efficiency tests.

**1. Throughput Test:**

**Objective:**  
Measure the number of queries that can be executed per second when running 100,000 findById queries using both modules.

**✅ Reactive (R2DBC):**

* Command: Runs the reactive JAR with throughputTest=true.
* **RAM Allocation:** 1 GB.
* **Result:**
  + Executes **100,000 queries in ~2 seconds**.
  + Throughput: **50,000 queries per second**, consistently.
  + Warm-up run is ignored to eliminate initialization overhead.

**✅ Traditional (JDBC) Without Virtual Threads:**

* Command: Runs the traditional JAR with throughputTest=true.
* **RAM Allocation:** 1 GB.
* **Result:**
  + Executes **100,000 queries in ~3.9 seconds**.
  + Throughput: **25,000 queries per second**, consistently.

**✅ Traditional (JDBC) With Virtual Threads:**

* The same test is repeated with virtualThreadExecutor=true.
* **Observation:**
  + There is **no significant improvement** in throughput compared to the non-virtual threads run.

**Key Takeaway:**

* The reactive module using **R2DBC** achieves double the throughput (50k QPS) compared to the traditional module (25k QPS).
* R2DBC also uses **fewer database connections**, making it more efficient in terms of resource usage.

**2. Efficiency Test:**

**Objective:**  
Measure the memory usage when fetching **10 million records** in a single SELECT \* query.

**✅ Traditional (JDBC) - 4 GB RAM:**

* Command: Runs the traditional JAR with efficiencyTest=true.
* **Result:**
  + The application **fails with a Java heap space error**, indicating that **4 GB is insufficient** to handle the query.

**✅ Reactive (R2DBC) - 4 GB RAM:**

* Command: Runs the reactive JAR with efficiencyTest=true.
* **Result:**
  + The reactive module successfully retrieves **all 10 million records** with **4 GB RAM**.

**✅ Memory Reduction Tests for R2DBC:**

* The speaker reduces the RAM allocation to test the lower limit:
  + **1 GB RAM:** Successfully retrieves all 10 million records.
  + **500 MB RAM:** Still succeeds.
  + **200 MB RAM:** Still succeeds, impressing the speaker.

**Key Takeaway:**

* The reactive module can handle the full dataset with significantly less memory, down to **200 MB**, due to its streaming nature and non-blocking processing.
* The traditional JDBC module fails with a heap space error at **4 GB**, highlighting its higher memory consumption due to blocking I/O and memory-intensive data processing.

**Overall Analysis:**

* **Throughput Test:** R2DBC outperforms JDBC, achieving double the throughput with fewer connections.
* **Efficiency Test:** R2DBC demonstrates exceptional memory efficiency, functioning with as little as 200 MB of RAM while JDBC fails with 4 GB.
* **Virtual Threads Impact:** No notable improvement was observed in the JDBC module when using virtual threads, indicating that the bottleneck is likely in the blocking nature of JDBC itself rather than the threading model.

## How R2DBC works

In the previous demo, the memory usage of **Spring Data JPA (Traditional) vs. Spring Data R2DBC (Reactive)** while fetching **10 million records** from a database.

* **Traditional Approach:** Required more than **4 GB of memory**.
* **Reactive Approach:** Worked fine with just **200 MB of memory**, even as low as **100 MB**.

**Why the Huge Memory Difference?**

**1. Traditional Approach (Spring Data JPA):**

* **Process:**
  + The query repository.findAll() issues a SELECT \* FROM customer.
  + The database sends the result set (all 10 million records) as a stream of bytes.
  + The JPA driver collects all records, **decodes them into Customer entities**, and **stores them in a List**.
* **Memory Implications:**
  + The entire result set must be **held in memory** at once to populate the list.
  + If the JVM does not have enough heap space (e.g., 4 GB), it will result in an **OutOfMemoryError**.
  + Example:
    - If each record takes 100 bytes, 10 million records would need approximately **1 GB of memory**.
    - If the Customer entity is more complex or includes nested objects, the memory usage can easily increase to multiple gigabytes.
* **Key Takeaway:**
  + **Blocking nature:** The JPA approach waits to receive all records before proceeding, leading to higher memory usage.

**2. Reactive Approach (Spring Data R2DBC):**

* **Process:**
  + The query repository.findAll() issues a SELECT \* FROM customer.
  + The database starts sending records as a **stream of bytes**.
  + The R2DBC driver processes records **one at a time** or in small batches (e.g., 256 items at a time).
  + These records are not stored in a list but are passed through a **Flux pipeline** for processing.
* **Memory Implications:**
  + The reactive pipeline maintains a small, **fixed-size buffer** (e.g., 256 items).
  + Only a limited number of records (e.g., 256) are held in memory at any given time.
  + Once the consumer processes a record, the buffer space is **freed up** for the next batch of records.
* **Backpressure Mechanism:**
  + If the consumer is **too slow to process records**, the buffer fills up.
  + The reactive driver will signal the database to **pause sending more data** using **TCP backpressure**.
  + This prevents memory overflow, allowing the reactive approach to work efficiently with limited memory.

**3. Key Differences:**

| **Aspect** | **Traditional (JPA)** | **Reactive (R2DBC)** |
| --- | --- | --- |
| **Data Handling** | Collects all records into a List, consuming large memory. | Streams records through a Flux, consuming minimal memory. |
| **Memory Usage** | Proportional to the number of records. | Fixed size (e.g., 256 records at a time). |
| **Backpressure** | Not supported. Collects everything at once. | Built-in, using TCP backpressure. |
| **Concurrency** | Blocking. Waits for all records to load before processing. | Non-blocking. Processes records as they arrive. |
| **Scalability** | Memory-intensive. Can lead to OOM errors. | Memory-efficient. Can handle large datasets smoothly. |

**Real-world Analogy:**

* **Traditional JPA:** Think of it as downloading a huge ZIP file (10 million records), extracting everything to memory, and then processing it all at once.
* **Reactive R2DBC:** Think of it as a conveyor belt where data is processed one item at a time or in small batches. The conveyor pauses if the worker is too slow, preventing a pile-up.

**Conclusion:**

* The traditional approach tries to **load everything into memory**, causing memory spikes and potential crashes for large datasets.
* The reactive approach **processes data in small batches**, using backpressure to manage flow control, resulting in significantly lower memory usage.

## R2DBC usage

**1. R2DBC is New and Still Evolving:**

* **What is R2DBC?**
  + R2DBC (**Reactive Relational Database Connectivity**) is a specification designed for **reactive access to relational databases**. Unlike traditional JDBC, it is non-blocking and leverages reactive streams.
* **Current Limitations:**
  + R2DBC is relatively **new** and some features are **not yet implemented**, such as:
    - **Batch Insert:** The ability to insert multiple records in a single operation is not available at the time of recording.
    - It is on the roadmap, but not yet released.

**2. Performance Testing Recommendations:**

* **Performance claims** should not be blindly trusted. Instead, recommend a **data-driven approach**:
  + **Identify Key Scenarios:** Focus on frequently used queries or operations in the application.
  + **Monitor Production Load:** Gather real-world performance data for these scenarios.
  + **Compare R2DBC and JPA:** Run the identified scenarios using both R2DBC and traditional JPA.
  + **Tools for Monitoring:**
    - **New Relic:** Advanced monitoring and analytics.
    - **JConsole:** JVM monitoring and profiling.
    - **PgAdmin:** Monitoring for PostgreSQL.
    - **Netstat:** Network connection monitoring to observe data transfer rates and TCP connections.

**3. What if R2DBC Fails to Perform Well?**

If R2DBC **does not meet performance expectations**, the speaker suggests an alternative:

* **Option: Use JPA with WebFlux:**
  + You can **still use traditional JPA** in a reactive WebFlux application.
  + However, JPA is **blocking**, so it must be run in a **separate thread pool** to avoid blocking the main event loop.
* **Implementation Approach:**
  + Use Mono.fromSupplier() to execute JPA operations asynchronously.
  + Apply the subscribeOn() operator with the boundedElastic() scheduler:
    - boundedElastic() is a dedicated scheduler optimized for blocking I/O tasks. It maintains a pool of worker threads that can handle blocking calls without impacting the main event loop.

**Example:**

Mono.fromSupplier(() -> customerRepository.findAll())

.subscribeOn(Schedulers.boundedElastic())

.subscribe(customers -> {

// Process the data here

});

* + This way, the JPA query is executed on a **separate thread**, allowing the reactive WebFlux application to remain non-blocking.

**Why Use boundedElastic()?**

* **Event Loop Threads:** WebFlux uses a small, fixed number of event loop threads for handling requests. Blocking operations on these threads can cause severe performance issues.
* **Elastic Scheduler:** boundedElastic() is a **bounded thread pool** designed to handle potentially long-running blocking operations. It dynamically adjusts the pool size based on demand.

**Summary:**

* R2DBC is promising but still **maturing**; some features like batch inserts are **not yet available**.
* Before adopting R2DBC, **conduct targeted performance tests** to validate its suitability.
* If R2DBC does not meet performance requirements, you can still use **JPA with WebFlux** by executing blocking operations on a **separate elastic scheduler** to prevent blocking the event loop.

## Reactive Manifesto

The **Reactive Manifesto** outlines a set of principles for designing highly responsive, resilient, and scalable systems, often referred to as **reactive systems or reactive microservices**. These principles are:

1. **Responsive:**
   * **Definition:** The system responds to user input quickly and consistently, providing immediate feedback.
   * **Example:**
     + In the example given, the traditional blocking approach took **10-15 seconds** to fetch all 10 million records before any processing could begin.
     + The **reactive approach**, on the other hand, started processing immediately by streaming the records as they became available, making the system **responsive** right from the start.
   * **Real-World Analogy:**
     + ChatGPT streams its responses gradually rather than waiting to process the entire request before displaying anything.
2. **Resilient:**
   * **Definition:** The system remains responsive even in the face of **failures**, handling them gracefully without crashing.
   * **Example:**
     + In the demo, the product service crashed, but the error was **handled as a signal** in the reactive pipeline. Instead of sending a 500 error to the client, the system gracefully handled the failure without cascading the error.
   * **Real-World Analogy:**
     + If a payment service in an e-commerce system fails, a resilient system might return a fallback response ("Service temporarily unavailable") rather than crashing the entire checkout process.
3. **Elastic:**
   * **Definition:** The system **adapts to varying workloads**, scaling up or down as needed to maintain responsiveness.
   * **Example:**
     + In the example, the reactive system managed to process **10 million records with just 200 MB of memory**, whereas the traditional system needed over **4 GB**.
     + This efficiency allows the reactive system to handle large loads without consuming excessive resources, making it highly **elastic**.
   * **Real-World Analogy:**
     + A streaming service like Netflix can handle thousands of concurrent streams during peak hours without degradation in quality, dynamically adjusting resources.
4. **Message-Driven:**
   * **Definition:** The system communicates internally via **asynchronous messages**, allowing components to remain decoupled and non-blocking.
   * **Example:**
     + The reactive system uses a **streaming mechanism** to process data as it arrives, instead of collecting it all at once. This approach uses **backpressure** to control the data flow based on processing speed.
   * **Real-World Analogy:**
     + A messaging system like Kafka sends data as events, allowing consumers to process messages at their own pace without blocking the producer.

**Interrelationship Between Principles:**

* The principles are **interconnected**:
  + If the system is not **resilient**, it cannot remain **responsive** during failures.
  + If it is not **elastic**, it may crash or slow down under heavy load, impacting **responsiveness**.
  + If it is not **message-driven**, it cannot effectively **scale** or **handle failures** without blocking components.

**Overall Takeaway:**

The **Reactive Manifesto** is a design philosophy that promotes systems capable of **scaling efficiently, remaining responsive during failures, and handling dynamic workloads** using asynchronous messaging and streaming. This approach is particularly beneficial for **distributed architectures, cloud computing, and real-time data processing** scenarios.

# Section5 Reactive Crud API sec03

**✅ Summary of Key Concepts in Spring WebFlux:**

**Objective:**

* Recap CRUD API development using **Spring WebFlux**, focusing on handling response statuses, streaming data with Flux, and testing with WebTestClient.

**✅ 1. CRUD APIs in Spring WebFlux:**

* In **Spring WebFlux**, we handle asynchronous, non-blocking requests using Mono and Flux.
* **Mono:** Represents a single asynchronous value or empty result.
* **Flux:** Represents a stream of 0 to N values, potentially infinite.

**Example: Basic CRUD API using WebFlux:**

java

CopyEdit

import org.springframework.web.bind.annotation.\*;

import reactor.core.publisher.Mono;

import reactor.core.publisher.Flux;

@RestController

@RequestMapping("/api/products")

public class ProductController {

private final ProductService productService;

public ProductController(ProductService productService) {

this.productService = productService;

}

@GetMapping("/{id}")

public Mono<Product> getProductById(@PathVariable String id) {

return productService.findById(id);

}

@GetMapping

public Flux<Product> getAllProducts() {

return productService.findAll();

}

@PostMapping

public Mono<Product> createProduct(@RequestBody Product product) {

return productService.save(product);

}

}

**✅ 2. Setting HTTP Status Codes with ResponseEntity:**

* In Spring WebFlux, we use ResponseEntity to **set custom HTTP status codes**.
* ResponseEntity can be used with Mono to set specific status codes, headers, etc.

**Example: Using ResponseEntity with Mono:**

java

CopyEdit

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import reactor.core.publisher.Mono;

@RestController

@RequestMapping("/api/orders")

public class OrderController {

private final OrderService orderService;

public OrderController(OrderService orderService) {

this.orderService = orderService;

}

@GetMapping("/{id}")

public Mono<ResponseEntity<Order>> getOrderById(@PathVariable String id) {

return orderService.findById(id)

.map(order -> ResponseEntity.ok(order))

.defaultIfEmpty(ResponseEntity.notFound().build());

}

@PostMapping

public Mono<ResponseEntity<Order>> createOrder(@RequestBody Order order) {

return orderService.save(order)

.map(savedOrder -> ResponseEntity.status(201).body(savedOrder));

}

}

**✅ 3. Limitations of ResponseEntity with Flux:**

* ResponseEntity is suitable for single responses (Mono), but **not appropriate for streaming data (Flux)**.
* Why?
  + With a Flux, we are dealing with **multiple responses**, but an HTTP response can only have **one status code**.
  + If we attempt to use Flux<ResponseEntity<T>>, it **does not make sense**, as the status is set only once when the response starts.

**Example - Incorrect Usage:**

java

CopyEdit

@GetMapping("/stream")

public Flux<ResponseEntity<Product>> streamProducts() {

return productService.findAll()

.map(product -> ResponseEntity.ok(product)); // ❌ Avoid this approach

}

* Instead, for streaming data, we **only return the data stream** (Flux<T>), not ResponseEntity.

**Correct Approach:**

java

CopyEdit

@GetMapping("/stream")

public Flux<Product> streamProducts() {

return productService.findAll();

}

* The HTTP response will be 200 OK for the entire stream.

**✅ 4. Handling Errors with WebFlux:**

* Since we cannot directly return different status codes for individual elements in a Flux, we handle errors using **Controller Advice or Exception Handlers**.

**Example - Centralized Error Handling with @ControllerAdvice:**

java

CopyEdit

import org.springframework.web.bind.annotation.\*;

import org.springframework.http.ResponseEntity;

import reactor.core.publisher.Mono;

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ProductNotFoundException.class)

public Mono<ResponseEntity<String>> handleNotFound(ProductNotFoundException ex) {

return Mono.just(ResponseEntity.status(404).body(ex.getMessage()));

}

}

**✅ 5. Testing with WebTestClient:**

* WebTestClient is a **reactive testing tool** for testing WebFlux endpoints.

**Steps:**

1. **Setup the client.**
2. **Invoke the API endpoint.**
3. **Assert the response status and body.**

**Example - Testing with WebTestClient:**

java

CopyEdit

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;

import org.springframework.test.web.reactive.server.WebTestClient;

@WebFluxTest(ProductController.class)

public class ProductControllerTest {

@Autowired

private WebTestClient webTestClient;

@Test

void testGetProductById() {

webTestClient.get()

.uri("/api/products/1")

.exchange()

.expectStatus().isOk()

.expectBody()

.jsonPath("$.name").isEqualTo("iPhone 20");

}

}

* exchange() sends the request and triggers the response.
* expectStatus() verifies the HTTP status.
* expectBody() verifies the response content.

**✅ 6. Testing Streaming Responses:**

* Streaming responses (using Flux) can also be tested using WebTestClient.

**Example:**

java

CopyEdit

@Test

void testStreamProducts() {

webTestClient.get()

.uri("/api/products/stream")

.exchange()

.expectStatus().isOk()

.returnResult(Product.class)

.getResponseBody()

.as(StepVerifier::create)

.expectNextMatches(product -> product.getName().equals("iPhone 20"))

.expectNextMatches(product -> product.getName().equals("MacBook Pro"))

.verifyComplete();

}

* Here, we use StepVerifier to **assert the streaming data**.

**✅ Summary:**

* ResponseEntity is useful for single responses (Mono), allowing us to set status codes and headers.
* For streaming data (Flux), we do **not use ResponseEntity** since we cannot set multiple status codes.
* For error handling in streaming endpoints, use **Controller Advice**.
* WebTestClient provides a comprehensive testing mechanism for both **single and streaming responses**

## @RequestBody Mono<T> vs T

In **Spring WebFlux**, when defining the request body in a controller method, you can

1. choose to use either a regular object (CustomerDto)
2. or a reactive type (Mono<CustomerDto>).

Both approaches work, but they behave differently under the hood. Let's break down the differences.

**1. Using CustomerDto as Request Body:**

@PostMapping("/customers")

public Mono<ResponseEntity<CustomerDto>> createCustomer(@RequestBody CustomerDto customerDto) {

// Business logic here

}

* **What Happens Under the Hood?**
  + When the request is received, Spring **waits for the entire request body** to be received before invoking the method.
  + It **collects all the bytes**, deserializes them into a CustomerDto object, and then passes that object to the method.
  + This process is **blocking** in the sense that the method cannot proceed until the entire body has been received and deserialized.
* **Implications:**
  + Suitable for **small payloads** where waiting for the entire request body is not an issue.
  + Simpler to work with, as you receive the fully constructed object.

**2. Using Mono<CustomerDto> as Request Body:**

@PostMapping("/customers")

public Mono<ResponseEntity<CustomerDto>> createCustomer(@RequestBody Mono<CustomerDto> customerDtoMono) {

return customerDtoMono.flatMap(customerDto -> {

// Business logic here

});

}

* **What Happens Under the Hood?**
  + The method can be invoked **immediately**, even before the complete request body has been received.
  + The request body is treated as a **stream of bytes**, and the deserialization process happens **as data is received**.
  + The method itself **does not block**, and the actual processing is deferred until the data is fully received and the Mono emits the object.
* **Implications:**
  + This approach is more **memory efficient** because it does not need to hold the entire request body in memory before processing.
  + It is more suitable for **streaming scenarios** or when dealing with **large payloads**.
  + The business logic is **triggered only when the Mono emits a value**, ensuring that the processing is non-blocking.

**Comparison and Example:**

* **Scenario:** Imagine a microservice that receives a large JSON payload representing customer data.
  + With CustomerDto, the entire payload must be received and deserialized before the method is invoked.
  + With Mono<CustomerDto>, the method can be invoked immediately, and the processing logic will **wait for the payload to be received**, allowing other work to proceed concurrently.

**Why is This Important?**

* In reactive programming, the goal is to **maximize system throughput** by handling requests asynchronously and in a non-blocking manner.
* By using Mono<CustomerDto>, the method is invoked earlier, and the processing pipeline is constructed without waiting for the entire payload, allowing the system to **optimize resource usage** and handle more requests concurrently.

**Future Implications:**

* The benefit of using the Mono type may not be obvious in simple use cases.
* However, when dealing with **streaming data**, the advantage becomes significant. You can start processing the data as it arrives without waiting for the entire payload, enabling **real-time processing and backpressure handling**.

## Mono Flux Response Entity

In Spring WebFlux, we typically return Mono or Flux from controller methods to indicate that the response will be handled asynchronously. However, understanding when and how to use Mono<ResponseEntity>, Flux<ResponseEntity>, or ResponseEntity<Mono> can be confusing. Let's break it down.

**1. How Spring WebFlux Interprets Mono and Flux:**

* When a controller returns a Mono or Flux, Spring WebFlux **subscribes to the publisher** and waits for the emitted signals:
  + If it receives **data or an empty signal**, it will respond with a **200 OK** status.
  + If it receives an **error signal**, it will respond with a **500 Internal Server Error**.

**2. Why Use Mono<ResponseEntity>?**

If you want to **control the HTTP status codes**, you need to wrap the response in a ResponseEntity.

**✅ Example 1: Mono of ResponseEntity**

@GetMapping("/customers/{id}")

public Mono<ResponseEntity<CustomerDto>> getCustomer(@PathVariable String id) {

return customerService.findById(id)

.map(customer -> ResponseEntity.ok(customer))

.defaultIfEmpty(ResponseEntity.notFound().build());

}

* If the Mono emits a value, the response will be 200 OK with the customer data.
* If the Mono completes empty, the response will be 404 Not Found.
* If an error occurs during processing, the response will be 500 Internal Server Error.

**3. Why Not Flux<ResponseEntity>?**

* Flux is intended for **streaming data**, i.e., a series of data items over time.
* HTTP status and headers are sent **only once**, at the start of the response.
* Thus, if you return Flux<ResponseEntity>, you are effectively trying to **send multiple HTTP responses**, which is not possible.

**Correct Approach:**

* Instead of Flux<ResponseEntity>, you can use **ResponseEntity<Flux>**, where:
  + The ResponseEntity sets the status and headers once.
  + The Flux streams the response body as a series of items.

**✅ Example 2: ResponseEntity of Flux**

@GetMapping("/stream/customers")

public ResponseEntity<Flux<CustomerDto>> streamCustomers() {

Flux<CustomerDto> customerStream = customerService.getAllCustomers();

return ResponseEntity.ok().body(customerStream);

}

* The status (200 OK) and headers are sent once.
* The body (a stream of CustomerDto objects) is sent over time.

**4. ResponseEntity of Mono vs. Mono of ResponseEntity**

| **Scenario** | **Return Type** | **Behavior** |
| --- | --- | --- |
| Single request, single response with status and headers | Mono<ResponseEntity<CustomerDto>> | Status and body are both set asynchronously. |
| Single request, streaming response | ResponseEntity<Flux<CustomerDto>> | Status is set synchronously; body is streamed asynchronously. |
| Single request, single response (synchronous status and headers, async body) | ResponseEntity<Mono<CustomerDto>> | Status and headers are set synchronously; body is resolved asynchronously. |

**5. Why Avoid Complex Types?**

* Types like Mono<ResponseEntity<Mono<CustomerDto>>> or ResponseEntity<Mono<Flux<CustomerDto>>> can be technically correct but are highly **confusing and unnecessary**.
* Instead, keep it simple:
  + For a typical request-response interaction, use Mono<ResponseEntity>.
  + For streaming responses, use ResponseEntity<Flux>.

**Summary:**

* Use Mono<ResponseEntity> to control the HTTP status code asynchronously.
* Use Flux for streaming data.
* Avoid Flux<ResponseEntity> as it is conceptually incorrect.
* For streaming scenarios, use ResponseEntity<Flux> to set the status and headers once and stream the body.

Web client is for sending non bocking request

Web test client is to write unit integration test

It supports below things to validate jsonPath is to validate the JSON response directly

@Test  
public void updateCustomer() {  
 var dto = new CustomerDto(null, "noel", "noel@gmail.com");  
 this.client.put()  
 .uri("/customers/10")  
 .bodyValue(dto)  
 .exchange()  
 .expectStatus().is2xxSuccessful()  
 .expectBody()  
 .consumeWith(r -> log.info("{}", new String(Objects.requireNonNull(r.getResponseBody()))))  
 .jsonPath("$.id").isEqualTo(10)  
 .jsonPath("$.name").isEqualTo("noel")  
 .jsonPath("$.email").isEqualTo("noel@gmail.com");  
}

# Section6 Input Validation Error Handling sec 04

Define custom exceptions

*public class* CustomerNotFoundException *extends* RuntimeException {  
  
 *private static final* String MESSAGE = "Customer [id=%d] is not found";  
  
 *public* CustomerNotFoundException(Integer id) {  
 *super*(MESSAGE.formatted(id));  
 }  
  
}

*public class* InvalidInputException *extends* RuntimeException {  
  
 *public* InvalidInputException(String message) {  
 *super*(message);  
 }  
  
}

Create a method which will create this exception

*public class* ApplicationExceptions {  
  
 *public static* <T> Mono<T> customerNotFound(Integer id){  
 *return* Mono.error(*new* CustomerNotFoundException(id));  
 }  
  
 *public static* <T> Mono<T> missingName(){  
 *return* Mono.error(*new* InvalidInputException("Name is required"));  
 }  
  
 *public static* <T> Mono<T> missingValidEmail(){  
 *return* Mono.error(*new* InvalidInputException("Valid email is required"));  
 }  
  
}

Create a class which will do validation

*public class* RequestValidator {  
  
 *public static* UnaryOperator<Mono<CustomerDto>> validate() {  
 *return* mono -> mono.filter(hasName())  
 .switchIfEmpty(ApplicationExceptions.missingName())  
 .filter(hasValidEmail())  
 .switchIfEmpty(ApplicationExceptions.missingValidEmail());  
 }  
  
 *private static* Predicate<CustomerDto> hasName() {  
 *return* dto -> Objects.nonNull(dto.name());  
 }  
  
 *private static* Predicate<CustomerDto> hasValidEmail() {  
 *return* dto -> Objects.nonNull(dto.email()) && dto.email().contains("@");  
 }  
  
}

Do the validation in controller and throw exception

*@PutMapping*("{id}")  
*public* Mono<CustomerDto> updateCustomer(*@PathVariable* Integer id, *@RequestBody* Mono<CustomerDto> mono) {  
 *return* mono.transform(RequestValidator.validate()) *//Validate customer* .as(validReq -> *this*.customerService.updateCustomer(id, validReq))  
 .switchIfEmpty(ApplicationExceptions.customerNotFound(id)); *// Throw error if validation fails*}

*@DeleteMapping*("{id}")  
*public* Mono<Void> deleteCustomer(*@PathVariable* Integer id) {  
 *return this*.customerService.deleteCustomerById(id)  
 .filter(b -> b)  
 *//Throw error if customer not found* .switchIfEmpty(ApplicationExceptions.customerNotFound(id))  
 .then();  
}

Global handler for error handling

*@ControllerAdvice  
public class* ApplicationExceptionHandler {  
  
 *@ExceptionHandler*(CustomerNotFoundException.*class*)  
 *public* ProblemDetail handleException(CustomerNotFoundException ex){  
 *var* problem = ProblemDetail.forStatusAndDetail(HttpStatus.NOT\_FOUND, ex.getMessage());  
 problem.setType(URI.create("http://example.com/problems/customer-not-found"));  
 problem.setTitle("Customer Not Found");  
 *return* problem;  
 }  
  
 *@ExceptionHandler*(InvalidInputException.*class*)  
 *public* ProblemDetail handleException(InvalidInputException ex){  
 *var* problem = ProblemDetail.forStatusAndDetail(HttpStatus.BAD\_REQUEST, ex.getMessage());  
 problem.setType(URI.create("http://example.com/problems/invalid-input"));  
 problem.setTitle("Invalid Input");  
 *return* problem;  
 }  
  
}

# Sec 7 Web filter sec05

## Introduction

**✅ Understanding Web Filters in Spring WebFlux**

**What is a Web Filter?**

* A **Web Filter** is an intermediary component that **intercepts requests and responses** in a Spring WebFlux application.
* It **executes before the request reaches the controller**, allowing us to implement cross-cutting concerns such as:
  + **Authentication and Authorization**
  + **Logging and Monitoring**
  + **Rate Limiting**
  + **Custom Header Validation**

**✅ How Does a Web Filter Work?**

* When a request is made to the application, it follows this flow:

Client → Web Filter → Controller → Service → Response

* If the filter **rejects the request**, it will **not proceed to the controller** and will immediately return a response.

**✅ Why Use Web Filters?**

* Imagine a use case where every request to the application **must include a specific header** (X-Custom-Header).
* If the header is missing or has an invalid value, the request should be **immediately rejected** with a 400 Bad Request response.
* Without a filter, we would need to **check the header in every controller method**, leading to repetitive and error-prone code.
* A **Web Filter** provides a centralized place to handle such common concerns.

**✅ 4. Advanced Use Cases for Web Filters:**

* **Logging and Monitoring:** Capture request and response data for monitoring.
* **Authorization:** Verify user roles or tokens.
* **Rate Limiting:** Implement request throttling.
* **CORS Handling:** Customize CORS headers.

**✅ Important Considerations:**

* **Request Body Access:**
  + The request body is **not deserialized** in the filter. It will only be available in the controller.
  + Filters are suitable for checking headers, query parameters, and path variables.
* **Avoid Business Logic in Filters:**
  + Keep filters focused on **cross-cutting concerns**, not business logic.
* **Order of Filters:**
  + Filters can be ordered using @Order or Ordered interface.

**✅ Summary:**

* Web Filters act as middleware, intercepting requests **before they reach controllers**.
* They are ideal for handling cross-cutting concerns like **authentication, validation, and logging**.
* We implemented a simple filter to **validate a custom header** and reject requests if the header is missing or invalid.
* Filters are not suitable for **request body validation** or business logic processing.

## Multiple web filters

**✅** Understanding Multiple Web Filters and Their Order in Spring WebFlux

Scenario Demonstrated:

* If we create two web filters, WebFilterDemoOne and WebFilterDemoTwo.
* The objective was to demonstrate:
  + How to create multiple web filters.
  + How to control the order of execution of these filters.
  + How to propagate the request to the next filter or controller in the chain.

**✅ Understanding the Filter Execution Flow:**

* When a request is sent to the application, the flow is as follows:

Client → WebFilterDemoOne → WebFilterDemoTwo → Controller → Response

* By default, spring processes the filters in the order they are declared or scanned.
* If a filter **does not call chain.filter(exchange)**, the request will **not proceed to the next filter or controller**, and a 200 OK response with an empty body is returned.

**✅ 4. Propagating the Request to the Next Filter or Controller**

* To allow the request to proceed to the next filter or controller, we **must call**:

return chain.filter(exchange);

* If we do this the request proceeds to WebFilterDemoOne and then to the controller.

**✅ 5. Controlling the Order of Filter Execution**

* By default, the filter execution order is **undefined**, but we can specify the order using the @Order annotation.
* Suppose we want WebFilterDemoTwo to run **before** WebFilterDemoOne.

**WebFilterDemoOne.java:**

@Component

@Order(2)

public class WebFilterDemoOne implements WebFilter {

**WebFilterDemoTwo.java:**

@Component

@Order(1)

public class WebFilterDemoTwo implements WebFilter {

* Now, the request will be handled by WebFilterDemoTwo **first**, followed by WebFilterDemoOne.

**✅ 7. Key Takeaways:**

* Multiple web filters can be defined, and they will be **chained in the specified order**.
* If a filter **does not call chain.filter(exchange)**, the request processing will **terminate at that filter**.
* The @Order annotation controls the execution order of filters:
  + Lower values have **higher precedence**, meaning they run **first**.
* Web filters are best used for **cross-cutting concerns**, such as logging, authentication, and request validation.

## Authentication filter implementation

**Scenario Overview:**

* We need to implement **authentication and authorization logic** using Web Filters.
* The application will have **two user categories** based on a custom header:
  + **Standard Users**: Only allowed to perform GET requests.
  + **Prime Users**: Allowed to perform all request types (GET, POST, PUT, DELETE).

**✅ Requirements:**

1. **Authentication:**
   * Incoming requests must include a header named auth-token.
   * Allowed values for auth-token:
     + "secret123" - Standard User.
     + "secret456" - Prime User.
   * If the header is **missing or has an invalid value**, respond with **401 Unauthorized**.
2. **Authorization:**
   * **Standard Users** can only perform GET requests. All other requests should be **rejected with 403 Forbidden**.
   * **Prime Users** can perform **any request type**.

**✅ Implementing Authentication Using WebFilter in Spring WebFlux**

**Scenario Overview:**

* We are implementing a **simple authentication filter** using WebFilter in Spring WebFlux.
* The objective is to:
  + Accept a custom header named auth-token.
  + Determine the user category (STANDARD or PRIME) based on the token value.
  + If the token is missing or invalid, respond with **401 Unauthorized**.
  + If the token is valid, allow the request to proceed to the next filter or the controller.

**✅ Implementation Steps:**

**1. Create the Category Enum**

* This enum represents the user categories: STANDARD and PRIME.

**Category.java**

public enum Category {  
 STANDARD,  
 PRIME;  
}

**2. Create the Authentication Filter**

* This filter will handle the authentication logic and determine the user category based on the token.

@Order(1)  
@Service  
public class AuthenticationWebFilter implements WebFilter {  
  
 private static final Map<String, Category> TOKEN\_CATEGORY\_MAP = Map.of(  
 "secret123", Category.STANDARD,  
 "secret456", Category.PRIME  
 );  
  
 @Override  
 public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {  
 var token = exchange.getRequest().getHeaders().getFirst("auth-token");  
 if(Objects.nonNull(token) && TOKEN\_CATEGORY\_MAP.containsKey(token)){  
 exchange.getAttributes().put("category", TOKEN\_CATEGORY\_MAP.get(token));  
 return chain.filter(exchange);  
 }  
 return Mono.fromRunnable(() -> exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED));  
 }  
  
}

**✅ Explanation of the Implementation:**

1. **Enum Category:**
   * Defines the user categories as STANDARD and PRIME.
2. **Token-Category Mapping:**
   * A static map TOKEN\_CATEGORY\_MAP is used to store valid tokens and their respective categories.
3. **Header Extraction:**
   * The header value is accessed using:

var token = exchange.getRequest().getHeaders().getFirst("auth-token");

* + The getFirst() method returns the **first value** of the specified header. If the header is absent, it returns null.

1. **Token Validation Logic:**
   * The isValidToken boolean is set based on whether the token is present and found in the TOKEN\_CATEGORY\_MAP.
2. **Response Handling:**
   * If the token is invalid or missing:
     + The response status is set to 401 Unauthorized.
     + An empty Mono is returned using Mono.fromRunnable().
3. **Proceeding to the Next Filter:**
   * If the token is valid, the request proceeds to the next filter using:

return chain.filter(exchange);

**✅ Testing the Implementation:**

**Testing the Filter using cURL or Postman:**

1. **Without auth-token Header:**

curl -X GET http://localhost:8080/api/customers/all

* **Response:** 401 Unauthorized

1. **With Invalid auth-token:**

curl -X GET http://localhost:8080/api/customers/all -H "auth-token: invalidToken"

* **Response:** 401 Unauthorized

1. **With Valid auth-token for Standard User:**

curl -X GET http://localhost:8080/api/customers/all -H "auth-token: secret123"

* **Response:** 200 OK

1. **With Valid auth-token for Prime User:**

curl -X GET http://localhost:8080/api/customers/all -H "auth-token: secret456"

* **Response:** 200 OK

## Authorization Filter attributes

In authorization filter we again have to do the same header check so instead of duplicating the code we can set the token attribute which we read from header in AuthenticationWebFilter and in AuthorizationWebFilter we can use the same values

AuthenticationWebFilter

exchange.getAttributes().put("category", TOKEN\_CATEGORY\_MAP.get(token));

AuthorizationWebFilter we can get the token value from headers using attributes. Thus we don’t have to duplicate the code

@Order(2)  
@Service  
public class AuthorizationWebFilter implements WebFilter {  
  
 @Override  
 public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {  
 //reading from attributes  
 var category = exchange.getAttributeOrDefault("category", Category.STANDARD);  
 return switch (category){  
 case STANDARD -> standard(exchange, chain);  
 case PRIME -> prime(exchange, chain);  
 };  
 }  
  
 private Mono<Void> prime(ServerWebExchange exchange, WebFilterChain chain) {  
 return chain.filter(exchange);  
 }  
  
 private Mono<Void> standard(ServerWebExchange exchange, WebFilterChain chain) {  
 var isGet = HttpMethod.GET.equals(exchange.getRequest().getMethod());  
 if(isGet){  
 return chain.filter(exchange);  
 }  
 return Mono.fromRunnable(() -> exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN));  
 }  
  
}

We can also get the attributes in controller also

@GetMapping  
public Flux<CustomerDto> allCustomers(@RequestAttribute("category") Category category) {  
 return this.customerService.getAllCustomers();  
}

# Section 8 Functional endpoints

## Introduction

*@Configuration  
public class* RouterConfiguration {  
  
 *@Autowired  
 private* CustomerRequestHandler customerRequestHandler;  
  
 *@Autowired  
 private* ApplicationExceptionHandler exceptionHandler;  
  
 *@Bean  
 public* RouterFunction<ServerResponse> customerRoutes() {  
 *return* RouterFunctions.route()  
 .GET("/customers", *this*.customerRequestHandler::allCustomers)  
 .GET("/customers/paginated", *this*.customerRequestHandler::paginatedCustomers)  
 .GET("/customers/{id}", *this*.customerRequestHandler::getCustomer)  
 .POST("/customers", *this*.customerRequestHandler::saveCustomer)  
 .PUT("/customers/{id}", *this*.customerRequestHandler::updateCustomer)  
 .DELETE("/customers/{id}", *this*.customerRequestHandler::deleteCustomer)  
 .onError(CustomerNotFoundException.*class*, *this*.exceptionHandler::handleException)  
 .onError(InvalidInputException.*class*, *this*.exceptionHandler::handleException)  
 .build();  
 }  
  
}

**✅ Router Function:**

* The RouterFunction is a function that takes a request and returns a response.
* The configuration method returns a RouterFunction<ServerResponse> object.
* This object defines multiple routes using methods like .route(), .andRoute(), etc.

*@Service  
public class* CustomerRequestHandler {  
  
 *@Autowired  
 private* CustomerService customerService;  
  
 *public* Mono<ServerResponse> allCustomers(ServerRequest request) {  
 *//request.pathVariable()  
 //request.headers()  
 //request.queryParam()  
 return this*.customerService.getAllCustomers()  
 .as(flux -> ServerResponse.ok().body(flux, CustomerDto.*class*));  
 }  
  
 *public* Mono<ServerResponse> paginatedCustomers(ServerRequest request) {  
 *var* page = request.queryParam("page").map(Integer::parseInt).orElse(1);  
 *var* size = request.queryParam("size").map(Integer::parseInt).orElse(3);  
 *return this*.customerService.getAllCustomers(page, size)  
 .collectList()  
 .flatMap(ServerResponse.ok()::bodyValue);  
 }

**✅ Understanding Publisher Types:**

* Mono<T>: Emits **zero or one** item.
* Flux<T>: Emits **zero or more** items.

For example:

* bodyValue() is used when we expect a **single object** (e.g., Mono<CustomerDto>).
* body() is used when we expect a **stream of objects** (e.g., Flux<CustomerDto>).

## Order of routes

**✅ Problem Context:**

* The explanation discusses how Spring WebFlux's RouterFunction resolves routes based on **HTTP method** and **path pattern**.
* The issue arises when similar paths are defined in the router configuration, leading to unexpected route resolution.

**✅ Example Scenario:**

Suppose we have two routes defined like this:

.route(GET("/customers/{id}"), handler::getCustomerById)

.route(GET("/customers/paginated"), handler::getCustomersPaginated)

Now, when a request comes to GET /customers/paginated, the router function processes the request as follows:

1. **Path Matching Logic:**
   * The router function checks routes in the order they are defined.
   * It first checks GET("/customers/{id}").
   * Since paginated is a valid path segment, it will treat paginated as the {id} variable and route it to the getCustomerById handler.
2. **Unexpected Behavior:**
   * The route /customers/paginated gets incorrectly matched to the {id} path variable route.
   * The request handler getCustomerById() will receive paginated as the id, leading to unexpected behavior (e.g., a NumberFormatException when trying to parse paginated as an Integer).

**✅ Solution:**

To resolve this issue, **route ordering** is crucial. Specific routes should be defined before more generic routes.

**Revised Router Configuration:**

.route(GET("/customers/paginated"), handler::getCustomersPaginated) // Specific route first

.route(GET("/customers/{id}"), handler::getCustomerById) // Generic route second

* By placing the /customers/paginated route **before** the /customers/{id} route, the router will correctly resolve requests to /customers/paginated first.
* The router function evaluates routes in the order they are defined, so more specific routes should always come before more generic ones.

**✅ Why Does This Happen?**

* Spring WebFlux's router function uses the **first-match wins** strategy.
* The path pattern /customers/{id} is more generic and can potentially match any path segment (paginated, 123, etc.).
* Hence, if it is defined before the /customers/paginated route, it will capture /customers/paginated as a dynamic path variable.

**✅ Best Practices for Defining Routes:**

1. **Specific Routes First:**
   * Define more specific routes (/customers/paginated) before generic routes (/customers/{id}).
2. **Use Explicit Path Variables:**
   * If /customers/paginated is a fixed path, avoid using {id} in a way that can mistakenly capture it.
3. **Add Type Checks or Regex Constraints:**
   * For numeric IDs, use patterns like /customers/{id:[0-9]+} to prevent non-numeric paths from being captured as IDs.

## Multiple router functions

**✅ 1. Organizing Router Functions:**

In a real-world application, there may be **many endpoints** (e.g., 50 or 100). If all routes are managed in a single router function, it can become unmanageable.

**What was demonstrated?**

* The instructor created **multiple router functions**, each handling specific sets of endpoints.
* This helps in maintaining clean and organized code.

**Example Structure:**

@Bean

public RouterFunction<ServerResponse> customerRoutes(CustomerRequestHandler handler) {

return RouterFunctions

.route(GET("/customers"), handler::getAllCustomers)

.andRoute(POST("/customers"), handler::saveCustomer)

.andRoute(PUT("/customers/{id}"), handler::updateCustomer);

}

@Bean

public RouterFunction<ServerResponse> customerGetRoutes(CustomerRequestHandler handler) {

return RouterFunctions

.route(GET("/customers/{id}"), handler::getCustomerById)

.andRoute(GET("/customers/paginated"), handler::getPaginatedCustomers);

}

@Bean

public RouterFunction<ServerResponse> customerDeleteRoutes(CustomerRequestHandler handler) {

return RouterFunctions

.route(DELETE("/customers/{id}"), handler::deleteCustomer);

}

* Each router function bean is now responsible for a specific set of routes (GET, POST, DELETE).
* This modular approach enhances maintainability and readability.

## Nested Router Function

**✅ 1. Problem Overview:**

* Initially, the instructor had multiple router functions, each handling specific routes (e.g., /customers, /customers/{id}, /customers/paginated).
* However, instead of exposing multiple beans, we want to create **one high-level router function** that delegates to child router functions based on path patterns.
* This approach helps keep routes organized while avoiding multiple beans for each route group.

**✅ 1. Solution:**

The code is split into two main router functions:

* **customerRoutes1()** - This is a high-level router function.
* **customerRoutes2()** - This is a child router function, intended to handle specific sub-paths under /customers.

*@Bean  
public* RouterFunction<ServerResponse> customerRoutes1() {  
 *return* RouterFunctions.route()  
 .path("customers", *this*::customerRoutes2) *// Delegates to customerRoutes2* .POST("/customers", *this*.customerRequestHandler::saveCustomer)  
 .PUT("/customers/{id}", *this*.customerRequestHandler::updateCustomer)  
 .DELETE("/customers/{id}", *this*.customerRequestHandler::deleteCustomer)  
 .onError(CustomerNotFoundException.*class*, *this*.exceptionHandler::handleException)  
 .onError(InvalidInputException.*class*, *this*.exceptionHandler::handleException)  
 .build();  
}  
  
*private* RouterFunction<ServerResponse> customerRoutes2() {  
 *return* RouterFunctions.route()  
 .GET("/paginated", *this*.customerRequestHandler::paginatedCustomers)  
 .GET("/{id}", *this*.customerRequestHandler::getCustomer)  
 .GET("", *this*.customerRequestHandler::allCustomers)  
 .build();  
}

**Explanation:**

* **Path Delegation:**
  + The path("customers", this::customerRoutes2) line delegates the routing for the /customers path to the customerRoutes2() function.
  + This allows for a more modular structure, separating specific GET requests into a separate method.
* **HTTP Methods:**
  + **POST /customers** - Calls saveCustomer() to create a new customer.
  + **PUT /customers/{id}** - Calls updateCustomer() to update an existing customer.
  + **DELETE /customers/{id}** - Calls deleteCustomer() to delete a customer.
* **Error Handling:**
  + Exceptions CustomerNotFoundException and InvalidInputException are globally handled by exceptionHandler.

**✅ Summary:**

* The code demonstrates a modular approach to routing using nested router functions in Spring WebFlux.
* The parent router function handles general paths (POST, PUT, DELETE), while the child router function is dedicated to GET requests.
* Path delegation and route order are critical to ensure proper routing behavior.
* Error handling is centralized in the parent router function, allowing for consistent error handling across all nested routes.

## Filters

**✅ 3. Applying Filters to Router Functions:**

In the router function approach, filters can be applied using the .filter() method.

@Component  
*public class* CustomerRouter {  
  
 *private static final* Map<String, String> MOCK\_USERS = Map.of(  
 "user1", "USER",  
 "admin1", "ADMIN"  
 );  
  
 *@Bean  
 public* RouterFunction<ServerResponse> routes() {  
 *return* RouterFunctions.route()  
 .GET("/customers", *this*::getAllCustomers)  
 .GET("/customers/{id}", *this*::getCustomerById)  
 .POST("/customers", *this*::createCustomer)  
 .filter(*this*::authenticationFilter)  
 .filter(*this*::authorizationFilter)  
 .build();  
 }  
  
 *private* Mono<ServerResponse> getAllCustomers(ServerRequest request) {  
 *return* ServerResponse.ok().bodyValue("All Customers");  
 }  
  
 *private* Mono<ServerResponse> getCustomerById(ServerRequest request) {  
 String id = request.pathVariable("id");  
 *return* ServerResponse.ok().bodyValue("Customer ID: " + id);  
 }  
  
 *private* Mono<ServerResponse> createCustomer(ServerRequest request) {  
 *return* ServerResponse.status(HttpStatus.CREATED).bodyValue("Customer Created");  
 }  
  
 *private* Mono<ServerResponse> authenticationFilter(ServerRequest request, RouterFunction<ServerResponse> next) {  
 List<String> authHeaders = request.headers().header("Authorization");  
  
 *if* (authHeaders.isEmpty()) {  
 *return* ServerResponse.status(HttpStatus.UNAUTHORIZED).bodyValue("Missing Authorization Header");  
 }  
  
 String username = authHeaders.get(0);  
  
 *if* (!MOCK\_USERS.containsKey(username)) {  
 *return* ServerResponse.status(HttpStatus.UNAUTHORIZED).bodyValue("User not found");  
 }  
  
 request.attributes().put("user", username);  
 request.attributes().put("role", MOCK\_USERS.get(username));  
  
 *return* next.route(request);  
 }  
  
 *private* Mono<ServerResponse> authorizationFilter(ServerRequest request, RouterFunction<ServerResponse> next) {  
 String path = request.path();  
 String role = (String) request.attribute("role").orElse("NONE");  
  
 *if* (path.startsWith("/customers/") && path.split("/").length == 3) {  
 *if* (!"ADMIN".equals(role)) {  
 *return* ServerResponse.status(HttpStatus.FORBIDDEN).bodyValue("Access Denied");  
 }  
 }  
  
 *return* next.route(request);  
 }  
}

## Request Predicates