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 sec06

## 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.

* We will create **multiple router functions**, each handling specific sets of endpoints.
* This helps in maintaining clean and organized code.

**Example Structure:**

@Bean  
public RouterFunction<ServerResponse> customerRoutes1() {  
 return RouterFunctions.route()  
 .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();  
}  
  
@Bean  
public RouterFunction<ServerResponse> customerRoutes2() {  
 return RouterFunctions.route()  
 .GET("/customers/paginated", this.customerRequestHandler::paginatedCustomers)  
 .GET("/customers/{id}", this.customerRequestHandler::getCustomer)  
 //Kept at end as it will satisfy other conditions  
 .GET("/customers", this.customerRequestHandler::allCustomers)  
 .build();  
}

* 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:**

* However, instead of exposing multiple beans, we want to create **one high-level router function** that delegate 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. If its simple logic we can handle like this

.filter((request,next) ->{  
 *return* ServerResponse.badRequest().build();  
 })

Or we can delegate

.filter(*this*::authorizationFilter)

@Component  
*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((request,next) ->{  
 *return* ServerResponse.badRequest().build();  
 })  
 .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> 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

The assignment is to create a simple calculator API using Spring WebFlux with functional routing. The application exposes a GET endpoint at /calculator/{a}/{b}. The operation to be performed is determined by an operation header, which can be one of +, -, \*, or /.

* **Endpoint Example:**
  + /calculator/5/4 with operation: + → Returns 9
  + /calculator/10/2 with operation: / → Returns 5
* **Validation Rules:**
  + If the second path variable b is 0, it returns **400 Bad Request** with a message: "b cannot be 0".
  + If the operation header is missing or invalid, it returns **400 Bad Request** with the message: "operation header should be + - \* /".
* **Implementation Details:**
  + Uses RequestPredicates to handle the operation header without if-else or switch statements.
  + Each route is defined using a combination of path variables and request predicates.
  + The response is a simple number in the response body without any JSON structure.

*@Configuration  
public class* CalculatorAssignment {  
  
 *@Bean  
 public* RouterFunction<ServerResponse> calculator() {  
 *return* RouterFunctions.route()  
 .path("calculator", *this*::calculatorRoutes)  
 .build();  
 }  
  
 *private* RouterFunction<ServerResponse> calculatorRoutes() {  
 *return* RouterFunctions.route()  
 .GET("/{a}/0", badRequest("b cannot be 0"))  
 .GET("/{a}/{b}", isOperation("+"), handle((a, b) -> a + b))  
 .GET("/{a}/{b}", isOperation("-"), handle((a, b) -> a - b))  
 .GET("/{a}/{b}", isOperation("\*"), handle((a, b) -> a \* b))  
 .GET("/{a}/{b}", isOperation("/"), handle((a, b) -> a / b))  
 .GET("/{a}/{b}", badRequest("operation header should be + - \* /"))  
 .build();  
 }  
  
 *private* RequestPredicate isOperation(String operation) { *// + -  
 return* RequestPredicates.headers(h -> operation.equals(h.firstHeader("operation")));  
 }  
  
 *private* HandlerFunction<ServerResponse> handle(BiFunction<Integer, Integer, Integer> function){  
 *return* req -> {  
 *var* a = Integer.parseInt(req.pathVariable("a"));  
 *var* b = Integer.parseInt(req.pathVariable("b"));  
 *var* result = function.apply(a, b);  
 *return* ServerResponse.ok().bodyValue(result);  
 };  
 }  
  
 *private* HandlerFunction<ServerResponse> badRequest(String message){  
 *return* req -> ServerResponse.badRequest().bodyValue(message);  
 }  
  
}

A RequestPredicate is a functional interface in Spring WebFlux that tests whether a given request meets certain criteria. It is essentially a condition that must evaluate to true for a request to be routed to a specific handler function.

private RequestPredicate isOperation(String operation) {

return RequestPredicates.headers(h -> operation.equals(h.firstHeader("operation")));

}

**Input Parameter:** operation — This is the expected value of the operation header (e.g., "+", "-", "\*", "/").

**RequestPredicate Creation:**

* RequestPredicates.headers(...) is a utility method provided by Spring to create a RequestPredicate based on request headers.
* The lambda (h -> operation.equals(h.firstHeader("operation"))) checks if the operation header matches the provided value.

# Section 9 Web Client sec07

## Introduction

 **What is WebClient?**

* A reactive alternative to RestTemplate, allowing non-blocking HTTP requests and responses.
* It is immutable and thread-safe once built.
* It can be configured with a base URL and then used for sending requests.

 **Setup and Configuration:**

* Create a WebClient bean for each external dependency (e.g., OrderService, PaymentService).
* Set base URL during initialization and expose the WebClient instance as a bean.
* Example:

java

CopyEdit

@Bean

public WebClient orderServiceClient() {

return WebClient.builder()

.baseUrl("http://orderservice.com")

.build();

}

* To modify a WebClient, use .mutate() to create a new instance with altered configurations.

 **Making Requests:**

* **GET Request:**
  + Use .get() to specify the path, .retrieve() to send the request, and .bodyToMono() or .bodyToFlux() to decode the response.
  + Example:

webClient.get()

.uri("/products/{id}", 1)

.retrieve()

.bodyToMono(Product.class);

* **POST Request:**
  + Use .post() and .bodyValue() to send data.
  + Example:

webClient.post()

.uri("/products")

.bodyValue(newProduct)

.retrieve()

.bodyToMono(Product.class);

 **Response Handling:**

* .bodyToMono() is used for single responses.
* .bodyToFlux() is used for streaming multiple responses.
* Both return reactive types (Mono or Flux), allowing further reactive processing.

## Abstract Class

abstract class AbstractWebClient {  
  
 private static final Logger log = LoggerFactory.getLogger(AbstractWebClient.class);  
  
 protected <T> Consumer<T> print(){  
 return item -> log.info("received: {}", item);  
 }  
  
 protected WebClient createWebClient() {  
 return createWebClient(b -> {});  
 }  
  
 protected WebClient createWebClient(Consumer<WebClient.Builder> consumer) {  
 var builder = WebClient.builder()  
 .baseUrl("http://localhost:7070/demo02");  
 consumer.accept(builder);  
 return builder.build();  
 }  
}

## Basic Get Request

public class Lec01MonoTest extends AbstractWebClient {  
  
 private final WebClient client = createWebClient();  
  
 @Test  
 public void simpleGet() throws InterruptedException {  
 this.client.get()  
 .uri("/lec01/product/1")  
 .retrieve()  
 .bodyToMono(Product.class)  
 .doOnNext(print())  
 .subscribe();  
  
 Thread.sleep(Duration.ofSeconds(2));  
 }

**Parameterizing variables:** Instead of contatanizing the variables we can also make it as parameters

@Test  
public void concurrentRequests() throws InterruptedException {  
 for (int i = 1; i <= 100; i++) {  
 this.client.get()  
 .uri("/lec01/product/{id}", i)  
 .retrieve()  
 .bodyToMono(Product.class)  
 .doOnNext(print())  
 .subscribe();  
 }  
  
 Thread.sleep(Duration.ofSeconds(2));  
}

We can also send map for multiple variables lec 100

Map map = Map.of("id", "1", "name", "Amit");  
  
@Test  
public void concurrentRequests() throws InterruptedException {  
 for (int i = 1; i <= 100; i++) {  
 this.client.get()  
 //.uri("/lec01/product/{id}", i)  
 .uri("/lec01/product/{id}", map)  
 .retrieve()  
 .bodyToMono(Product.class)  
 .doOnNext(print())  
 .subscribe();  
 }

## How event loop works

**Explanation of Non-Blocking I/O and Event Loop in WebClient (Reactive Programming)**

**Context and Objective:**

* The objective of this lecture is to explain how non-blocking I/O works internally with a reactive WebClient in Spring, similar to the concepts in Java Reactive Programming.
* The explanation focuses on the concept of the **event loop**, **inbound queue**, and **outbound queue** to handle multiple requests concurrently using minimal threads.

**Understanding the Event Loop and Queues:**

1. **Threads and CPU:**
   * Reactive programming leverages a small number of threads, typically one thread per CPU core.
   * Each thread continuously checks for tasks in a queue rather than waiting for a response from a network call.
   * This design helps in maximizing CPU utilization and minimizing thread overhead.
2. **Outbound Queue (Outgoing Requests):**
   * Whenever a request is sent via WebClient, it is added to the **outbound queue**.
   * For example, sending multiple requests for product IDs (1, 2, 3, 4, 5) will add them to this outbound queue.

rust

CopyEdit

Outbound Queue: [Request for ID 1, Request for ID 2, Request for ID 3, ...]

* + The event loop picks each request, sends it, and immediately moves on to the next request without waiting for a response.

1. **Non-Blocking Execution:**
   * When a request is sent, the thread **does not wait** for the response.
   * Instead, it continues processing the next request in the queue.
   * Example:
     + Send request for ID 1 – Takes 1 second to respond.
     + Send request for ID 2 – Takes 1 second to respond.
     + Send request for ID 3 – Takes 1 second to respond.
   * All these requests are sent **concurrently**, not sequentially.
2. **Inbound Queue (Incoming Responses):**
   * While the event loop is sending outgoing requests, responses start coming back asynchronously.
   * These responses are collected in an **inbound queue**.
   * The order of responses may not match the order of requests due to network latency or processing time.

rust

CopyEdit

Inbound Queue: [Response for ID 2, Response for ID 1, Response for ID 3, ...]

* + The thread processes responses as they arrive, based on the order they are received, not the order they were sent.

**Example of Execution Flow:**

* **Outbound Requests:**
  + Request for ID 1 → Sent (1 second to respond)
  + Request for ID 2 → Sent (1 second to respond)
  + Request for ID 3 → Sent (1 second to respond)
* **Inbound Responses:**
  + Response for ID 2 (arrives first, even though 1 was sent first)
  + Response for ID 1 (arrives second)
  + Response for ID 3 (arrives last)

**Key Takeaways:**

* Reactive programming in Spring leverages a small number of threads (one per CPU) to handle multiple requests concurrently.
* Tasks are managed through **outbound and inbound queues**, enabling non-blocking execution.
* Responses can arrive out of order, but they are processed as soon as they are received.
* This approach is highly efficient in handling high-concurrency scenarios without blocking threads.

## Streaming response Flux

public class Lec02FluxTest extends AbstractWebClient {  
  
 private final WebClient client = createWebClient();  
  
 @Test  
 public void streamingResponse() {  
 this.client.get()  
 .uri("/lec02/product/stream")  
 .retrieve()  
 .bodyToFlux(Product.class)  
 .take(Duration.ofSeconds(3))  
 .doOnNext(print())  
 .then()  
 //Step verifier is added as it will wait for the response to complete  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
  
}

## POST Body publisher vs Body value

**Two Methods for Sending Request Body:**

**1. Using bodyValue() Method:**

* This method is used when the request body is readily available in memory (e.g., a simple object or DTO).

public class Lec03PostTest extends AbstractWebClient {  
  
 private final WebClient client = createWebClient();  
  
 @Test  
 public void postBodyValue() {  
 var product = new Product(null, "iphone", 1000);  
 this.client.post()  
 .uri("/lec03/product")  
 .bodyValue(product)  
 .retrieve()  
 .bodyToMono(Product.class)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }

**Using body() Method:**

* This method is used when the request body is a **reactive publisher** (e.g., Mono, Flux).
* This is useful when the request body is not immediately available and will be emitted asynchronously.

@Test  
public void postBody() {  
 var mono = Mono.fromSupplier(() -> new Product(null, "iphone", 1000))  
 .delayElement(Duration.ofSeconds(1));  
 this.client.post()  
 .uri("/lec03/product")  
 .body(mono, Product.class)  
 .retrieve()  
 .bodyToMono(Product.class)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
}

**Key Takeaways:**

* bodyValue() is used for sending a simple, in-memory object.
* body() is used when the request body is a reactive publisher, enabling asynchronous request body emission.
* Both methods allow sending POST requests, but the body() method is more flexible and suitable for reactive programming scenarios.

## Setting Headers

public class Lec04HeaderTest extends AbstractWebClient {  
  
 private final WebClient client = createWebClient(b -> b.defaultHeader("caller-id", "order-service"));  
  
 @Test  
 public void defaultHeader() {  
 this.client.get()  
 .uri("/lec04/product/{id}", 1)  
 .retrieve()  
 .bodyToMono(Product.class)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
  
 @Test  
 public void overrideHeader() {  
 this.client.get()  
 .uri("/lec04/product/{id}", 1)  
 .header("caller-id", "new-value")  
 .retrieve()  
 .bodyToMono(Product.class)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
  
 @Test  
 public void headersWithMap() {  
 var map = Map.of(  
 "caller-id", "new-value",  
 "some-key", "some-value"  
 );  
 this.client.get()  
 .uri("/lec04/product/{id}", 1)  
 .headers(h -> h.setAll(map))  
 .retrieve()  
 .bodyToMono(Product.class)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
}

**1. Setting Default Header:**

* We can define a default header at the time of creating the WebClient instance.
* This is useful when the header is expected in every request.
* **Execution Flow:**
  + The Caller-ID header is automatically included in every request with the value **Order-Service**.
  + The request is sent and the server responds successfully since the required header is present.

*private final* WebClient client = createWebClient(b -> b.defaultHeader("caller-id", "order-service"));

**2. Overriding the Default Header:**

* There are cases where we need to override the default header for specific requests.
* This can be done by using the .header() method after specifying the URI.

**Implementation:**

* **Execution Flow:**
  + The default Caller-ID header value Order-Service is **overridden** with New-Value.
  + The server receives Caller-ID: New-Value and processes the request accordingly.

*@Test  
public void* overrideHeader() {  
 *this*.client.get()  
 .uri("/lec04/product/{id}", 1)  
 .header("caller-id", "new-value") *//override header* .retrieve()  
 .bodyToMono(Product.*class*)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
}

**3. Setting Multiple Headers Using a Map:**

* If there are multiple headers to be set or overridden, we can use a map to streamline the process.

*@Test  
public void* headersWithMap() {  
 *var* map = Map.of(  
 "caller-id", "new-value",  
 "some-key", "some-value"  
 );  
 *this*.client.get()  
 .uri("/lec04/product/{id}", 1)  
 .headers(h -> h.setAll(map)) *//setting header with Map* .retrieve()  
 .bodyToMono(Product.*class*)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
}

## Error Handling

public class Lec05ErrorResponseTest extends AbstractWebClient {  
  
 private static final Logger log = LoggerFactory.getLogger(Lec05ErrorResponseTest.class);  
  
 private final WebClient client = createWebClient();  
  
 @Test  
 public void handlingError() {  
 this.client.get()  
 .uri("/lec05/calculator/{a}/{b}", 10, 20)  
 .header("operation", "@")  
 .retrieve()  
 .bodyToMono(CalculatorResponse.class)  
 // .onErrorReturn(new CalculatorResponse(0, 0, null, 0.0))  
 .doOnError(WebClientResponseException.class, ex -> log.info("{}", ex.getResponseBodyAs(ProblemDetail.class)))  
 .onErrorReturn(WebClientResponseException.InternalServerError.class, new CalculatorResponse(0, 0, null, 0.0))  
 .onErrorReturn(WebClientResponseException.BadRequest.class, new CalculatorResponse(0, 0, null, -1.0))  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }

**Explanation:**

**✅ Error Handling Strategies:**

1. **onErrorReturn:**
   * Provides a default response in case of specific exceptions.
   * In the above example, we provide a default response when encountering a BadRequest (400) error.
2. **onErrorResume:**
   * Allows you to handle exceptions and return an alternative response or perform a specific action.
   * In the above example, we handle all WebClientResponseException errors and return a default response.
3. **doOnError:**
   * Executes a specific action upon encountering an error without altering the response.
   * Here, we log the error response and attempt to decode it into a ProblemDetail object.

**Summary and Key Points:**

* The onErrorReturn() operator provides a default response when specific exceptions occur.
* The onErrorResume() operator allows you to handle exceptions dynamically and respond accordingly.
* The doOnError() operator enables logging or additional processing when an error occurs.
* The getResponseBodyAs() method allows decoding error responses into custom DTOs (e.g., ProblemDetail).

## Retrieve Exchange

@Test  
public void exchange() {  
 this.client.get()  
 .uri("/lec05/calculator/{a}/{b}", 10, 20)  
 .header("operation", "+")  
 .exchangeToMono(this::decode)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
}  
  
private Mono<CalculatorResponse> decode(ClientResponse clientResponse) {  
 //clientResponse.cookies()  
 //clientResponse.headers()  
 log.info("status code: {}", clientResponse.statusCode());  
 if (clientResponse.statusCode().isError()) {  
 return clientResponse.bodyToMono(ProblemDetail.class)  
 .doOnNext(pd -> log.info("{}", pd))  
 .then(Mono.empty());  
 }  
 return clientResponse.bodyToMono(CalculatorResponse.class);  
}

**1. retrieve() Method**

* **Purpose:**
  + Simplifies the process of fetching a response body and mapping it to a desired type.
* **Usage:**
  + If the main focus is to extract the response body and handle basic error scenarios.
* **Example:**
* **Limitations of retrieve():**
  + It directly maps the response body to a specified type.
  + It does not provide access to the full ClientResponse object, so we **cannot access response headers, cookies, or status codes** easily.

**2. exchangeToMono() Method**

* **Purpose:**
  + Provides lower-level access to the entire ClientResponse object, allowing you to inspect headers, status codes, and more.
* **Usage:**
  + Use exchangeToMono() when you need more control over the response, such as accessing headers, cookies, and status codes.

**Why Use exchangeToMono()?**

* Provides access to the full ClientResponse object.
* Allows inspecting response headers, cookies, and status codes.
* Enables conditional handling based on response status codes.

**✅ Implementation Analysis:**

1. **Receiving ClientResponse Object:**
   * In exchangeToMono(), we receive the full ClientResponse object:

exchangeToMono(clientResponse -> decodeResponse(clientResponse));

1. **Logging the Status Code:**
   * Before handling the response body, we can log the status code:

System.out.println("Status Code: " + clientResponse.statusCode());

1. **Handling 400 Errors:**
   * If the response is a 400 Bad Request, we map the response to a ProblemDetail object:

if (clientResponse.statusCode().is4xxClientError()) {

return clientResponse.bodyToMono(ProblemDetail.class)

.doOnNext(problemDetail -> {

System.out.println("Problem Detail: " + problemDetail);

})

.then(Mono.empty()); // Emit empty signal

}

1. **Handling Successful Response:**
   * If the response is successful (e.g., 200 OK), we map it to a CalculatorResponse object:

return clientResponse.bodyToMono(CalculatorResponse.class)

.doOnNext(response -> {

System.out.println("Success Response: " + response);

});

## Query parameters

public class Lec06QueryParamsTest extends AbstractWebClient {  
  
 private final WebClient client = createWebClient();  
  
 @Test  
 public void uriBuilderVariables() {  
 var path = "/lec06/calculator";  
 var query = "first={first}&second={second}&operation={operation}";  
 this.client.get()  
 .uri(builder -> builder.path(path).query(query).build(10, 20, "+"))  
 .retrieve()  
 .bodyToMono(CalculatorResponse.class)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
  
 @Test  
 public void uriBuilderMap() {  
 var path = "/lec06/calculator";  
 var query = "first={first}&second={second}&operation={operation}";  
 var map = Map.of(  
 "first", 10,  
 "second", 20,  
 "operation", "\*"  
 );  
 this.client.get()  
 .uri(builder -> builder.path(path).query(query).build(map))  
 .retrieve()  
 .bodyToMono(CalculatorResponse.class)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
  
}

**Using UriBuilder with Direct Parameters:**

* We construct the URI using UriBuilder with direct query parameters:

*public void* uriBuilderVariables() {  
 *var* path = "/lec06/calculator";  
 *var* query = "first={first}&second={second}&operation={operation}";  
 *this*.client.get()  
 .uri(builder -> builder.path(path).query(query).build(10, 20, "+"))

**Using UriBuilder with a Map:**

* Alternatively, we can use a Map to pass query parameters:

*public void* uriBuilderMap() {  
 *var* path = "/lec06/calculator";  
 *var* query = "first={first}&second={second}&operation={operation}";  
 *var* map = Map.of(  
 "first", 10,  
 "second", 20,  
 "operation", "\*"  
 );  
 *this*.client.get()  
 .uri(builder -> builder.path(path).query(query).build(map))

**✅ Handling Missing Query Parameters:**

* If a required query parameter is missing or misspelled, the UriBuilder will not be able to build the URI, and an exception will be thrown.

## Basic Auth

public class Lec07BasicAuthTest extends AbstractWebClient {  
  
 private final WebClient client = createWebClient(b -> b.defaultHeaders(h -> h.setBasicAuth("java", "secret")));  
  
 @Test  
 public void basicAuth() {  
 this.client.get()  
 .uri("/lec07/product/{id}", 1)  
 .retrieve()  
 .bodyToMono(Product.class)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
  
}

## Bearer Auth

public class Lec08BearerAuthTest extends AbstractWebClient {  
  
 private final WebClient client = createWebClient(b -> b.defaultHeaders(h -> h.setBearerAuth("eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9")));  
  
 @Test  
 public void bearerAuth() {  
 this.client.get()  
 .uri("/lec08/product/{id}", 1)  
 .retrieve()  
 .bodyToMono(Product.class)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
  
}

## Exchange Filter Function

**✅ Overview:**

* In a **WebFlux** application, we use a WebFilter to handle **incoming requests**. This is analogous to the Servlet Filter in traditional Spring MVC applications.
* However, when making **outgoing requests** using WebClient, we need a similar mechanism to handle cross-cutting concerns (e.g., authentication, logging, monitoring).
* This mechanism is the **Exchange Filter Function**.

**✅ What is an Exchange Filter Function?**

* An ExchangeFilterFunction is a functional interface in Spring WebClient that allows us to intercept and modify the outgoing request and/or the response.
* It is similar to the WebFilter but operates on **outgoing requests** rather than incoming requests.

**Definition:**

@FunctionalInterface

public interface ExchangeFilterFunction {

Mono<ClientResponse> filter(ClientRequest request, ExchangeFunction next);

}

* **ClientRequest**: Represents the outgoing request that can be modified.
* **ExchangeFunction**: Allows us to pass the modified request to the next filter or to execute the request.

public class Lec09ExchangeFilterTest extends AbstractWebClient {  
  
 private static final Logger log = LoggerFactory.getLogger(Lec09ExchangeFilterTest.class);  
 private final WebClient client = createWebClient(b -> b.filter(tokenGenerator())  
 .filter(requestLogger()));  
  
 @Test  
 public void exchangeFilter() {  
 for (int i = 1; i <= 5; i++) {  
 this.client.get()  
 .uri("/lec09/product/{id}", i)  
 .attribute("enable-logging", i % 2 == 0)  
 .retrieve()  
 .bodyToMono(Product.class)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
 }  
  
 private ExchangeFilterFunction tokenGenerator() {  
 return (request, next) -> {  
 var token = UUID.randomUUID().toString().replace("-", "");  
 log.info("generated token: {}", token);  
 var modifiedRequest = ClientRequest.from(request).headers(h -> h.setBearerAuth(token)).build();  
 return next.exchange(modifiedRequest);  
 };  
 }  
  
 private ExchangeFilterFunction requestLogger() {  
 return (request, next) -> {  
 var isEnabled = (Boolean) request.attributes().getOrDefault("enable-logging", false);  
 if(isEnabled){  
 log.info("request url - {}: {}", request.method(), request.url());  
 }  
 return next.exchange(request);  
 };  
 }  
  
  
}

**✅ Key Takeaways:**

* ExchangeFilterFunction is a powerful mechanism to handle cross-cutting concerns for outgoing requests using WebClient.
* It allows us to modify the request (e.g., setting headers, logging) or the response (e.g., error handling).
* It helps in maintaining the **Single Responsibility Principle (SRP)** by keeping the token generation logic outside the service classes.

## Web client attributes

**✅ What Are WebClient Attributes?**

* **Attributes in WebClient** are essentially key-value pairs that can be attached to a ClientRequest.
* These attributes can be accessed and modified by multiple filters in the WebClient filter chain.
* This allows for **reusable, modular filter logic**, where specific filters can behave differently based on attribute values.

**✅ Use Case in the Lecture:**

* We have a single WebClient instance that is injected into multiple service classes.
* We want to control the behavior of a logging filter based on a flag (enableLogging).
* Some service classes may want logging enabled, while others may not.

*@Test  
public void* exchangeFilter() {  
 *for* (*int* i = 1; i <= 5; i++) {  
 *this*.client.get()  
 .uri("/lec09/product/{id}", i)  
 *//setting attribute* .attribute("enable-logging", i % 2 == 0)  
 .retrieve()  
 .bodyToMono(Product.*class*)  
 .doOnNext(print())  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
}

*private* ExchangeFilterFunction requestLogger() {  
 *return* (request, next) -> {  
 *//Getting attribute  
 var* isEnabled = (Boolean) request.attributes().getOrDefault("enable-logging", *false*);  
 *if* (isEnabled) {  
 log.info("request url - {}: {}", request.method(), request.url());  
 }  
 *return* next.exchange(request);  
 };  
}

# Section 10 Streaming sec08

Streaming is like when we send or receive continuous data and not request response like REST. For e.g.

1. File upload
2. File Download
3. Both upload and Download
4. Location tracking
5. IOT devices like apple watch etc.

## JSON Lines

**1. What is JSON Line Format?**

* **JSON Line (JSONL)** is a data format where each line is a valid JSON object.
* It is also known as **Newline Delimited JSON (NDJSON)** or **Line Delimited JSON (LDJSON)**.
* Each line is independent and can be parsed individually, making it ideal for **streaming large datasets**.

**2. JSON Array Format vs. JSON Line Format**

**JSON Array Format:**

* Typically used to send a collection of objects as a single array.
* Example:

[

{"id": 1, "name": "Product A"},

{"id": 2, "name": "Product B"},

{"id": 3, "name": "Product C"}

]

* ✅ **Problem with JSON Array Format:**
  + If the server crashes before sending the closing ] bracket, the entire response becomes **invalid**.
  + The client cannot parse the data until the entire array is received and closed.
  + The entire array must be loaded into memory before parsing, making it **inefficient for large datasets**.

**JSON Line Format:**

* Each line is a self-contained JSON object.
* Example:

{"id": 1, "name": "Product A"}

{"id": 2, "name": "Product B"}

{"id": 3, "name": "Product C"}

* ✅ **Advantages of JSON Line Format:**
  + Each line can be processed independently, so even if the server crashes, the client can still parse the received lines.
  + Memory usage is significantly lower because each line can be processed and discarded without holding the entire dataset in memory.
  + Ideal for **streaming data**, such as logs or large datasets.

**3. Real-World Use Cases of JSONL:**

* **Big Data Processing:** Used in tools like Apache Spark, Hadoop, and Google BigQuery for processing large datasets efficiently.
* **Streaming APIs:** Allows processing of data in chunks rather than waiting for the entire dataset.
* **Log Files:** Each log entry can be a separate JSON object, allowing real-time processing.

**4. Example Scenario:**

Imagine a server sending data for **1 million products**:

* **Using JSON Array:**
  + The server must create a single array with 1 million products.
  + The client must wait for the closing ] to parse the data.
  + Memory usage spikes due to holding the entire array in memory.
* **Using JSON Line Format:**
  + The server sends each product as a separate line.
  + The client can start processing data immediately, without waiting for the entire dataset.
  + Memory usage is minimized as each line is processed independently.

**5. Parsing JSONL in Java:**

Example of reading a JSONL file in Java:

import java.io.\*;

import com.fasterxml.jackson.databind.ObjectMapper;

import com.fasterxml.jackson.databind.JsonNode;

public class JsonlReader {

public static void main(String[] args) {

ObjectMapper objectMapper = new ObjectMapper();

String filePath = "products.jsonl";

try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {

String line;

while ((line = reader.readLine()) != null) {\

JsonNode jsonNode = objectMapper.readTree(line);

System.out.println(jsonNode);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

* This code reads a JSONL file line by line and processes each JSON object independently.

**✅ Summary:**

* **JSON Array:** Suitable for small datasets; requires complete data to be valid.
* **JSONL:** Ideal for large datasets or streaming data; each line is a separate JSON object, allowing for incremental processing.

## Implementation

We have below end points for upload and download

@PostMapping(value = "upload", consumes = MediaType.APPLICATION\_NDJSON\_VALUE)  
public Mono<UploadResponse> uploadProducts(@RequestBody Flux<ProductDto> flux) {  
 log.info("invoked");  
 return this.service.saveProducts(flux)  
 .then(this.service.getProductsCount())  
 .map(count -> new UploadResponse(UUID.randomUUID(), count));  
}  
  
@GetMapping(value = "download", produces = MediaType.APPLICATION\_NDJSON\_VALUE)  
public Flux<ProductDto> downloadProducts(){  
 return this.service.allProducts();  
}

We test by uploading and downloading 1 million records

public class ProductsUploadDownloadTest {  
  
 private static final Logger log = LoggerFactory.getLogger(ProductsUploadDownloadTest.class);  
 private final ProductClient productClient = new ProductClient();  
  
 @Test  
 public void upload() {  
 var flux = Flux.range(1, 1\_000\_000)  
 .map(i -> new ProductDto(null, "product-" + i, i));  
  
 this.productClient.uploadProducts(flux)  
 .doOnNext(r -> log.info("received: {}", r))  
 .then()  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
  
 @Test  
 public void download() {  
 this.productClient.downloadProducts()  
 .map(ProductDto::toString)  
 .as(flux -> FileWriter.create(flux, Path.of("products.txt")))  
 .as(StepVerifier::create)  
 .expectComplete()  
 .verify();  
 }  
}

# Section 11 Server Sent Events sec09

## Introduction

**✅ Problem Explanation:**

* **Scenario:** In a trading platform, news site, or live game scoreboard, the frontend frequently requests updates from the backend to display the latest data (e.g., stock prices, election results, game scores).
* **Traditional Approach:** The frontend keeps sending periodic requests (polling) to check for updates.
  + This results in unnecessary requests and increased server load, especially when there are no new updates.

**✅ What is Server-Sent Events (SSE)?**

* **Definition:** SSE is a mechanism that allows the backend server to push updates to the frontend automatically whenever new data is available.
* **One-Way Communication:** The server can send events to the client, but the client cannot send messages back through the same connection.
* **Implementation in WebFlux:**
  + WebFlux, being a reactive framework in Spring, provides a way to handle streams of data using Flux.
  + The response content type is set to text/event-stream, which is the format the browser expects for SSE.

**✅ Use Case Example:**

* **Business Scenario:** Suppose we have a product service, and users want to be notified whenever a new product is added.
  + The backend will send notifications to the frontend whenever a new product is added to the database.

**✅ Implementation Approach:**

1. **How to Detect New Product Additions:**
   * The main challenge is to detect when a new product is added.
   * This is where **Sinks** come into play in reactive programming.
2. **What is a Sink?**
   * A Sink can act as both a publisher and a subscriber.
   * Multiple threads can emit data to the Sink (input end), and subscribers can consume the data as a Flux (output end).
3. **Implementation Steps:**
   * Create a Sink that will emit events whenever a new product is added.
   * The frontend subscribes to this Sink and listens for new product notifications in real-time.

## Set up

We create a sink. replay().limit(1) put to show last event if the consumer missed it

@Configuration  
public class ApplicationConfig {  
  
 @Bean  
 public Sinks.Many<ProductDto> sink(){  
 return Sinks.many().replay().limit(1);  
 }  
  
}

AN endpoint which will take a maxPrice as path variable. In the service layer it fetches data from the above sink and if the value is greater than the maxPrice then the filter in controller will emit it

Controller

@GetMapping(value = "/stream/{maxPrice}", produces = MediaType.TEXT\_EVENT\_STREAM\_VALUE)  
public Flux<ProductDto> productStream(@PathVariable Integer maxPrice) {  
 return this.service.productStream()  
 .filter(dto -> dto.price() <= maxPrice);  
}

Service

public Flux<ProductDto> productStream(){  
 return this.sink.asFlux();  
}

There is another end point to save product

@PostMapping  
public Mono<ProductDto> saveProduct(@RequestBody Mono<ProductDto> mono) {  
 return this.service.saveProduct(mono);  
}

In the service layer if the save is successful it is emitted to sink

public Mono<ProductDto> saveProduct(Mono<ProductDto> mono) {  
 return mono.map(EntityDtoMapper::toEntity)  
 .flatMap(this.repository::save)  
 .map(EntityDtoMapper::toDto)  
 //send to sink  
 .doOnNext(this.sink::tryEmitNext);  
}

There is also a service class set as command line runner which will insert data to sink periodically so we don’t have to hit save API.

@Service  
public class DataSetupService implements CommandLineRunner {  
  
 @Autowired  
 private ProductService productService;  
  
 @Override  
 public void run(String... args) throws Exception {  
 Flux.range(1, 1000)  
 .delayElements(Duration.ofSeconds(1))  
 .map(i -> new ProductDto(null, "product-" + i, ThreadLocalRandom.current().nextInt(1, 100)))  
 .flatMap(dto -> this.productService.saveProduct(Mono.just(dto)))  
 .subscribe();  
 }  
  
}

HTML to test

In resources/static/index.html

# Section 12 Performance Optimization sec10

**✅ WebFlux is Powerful, but Not a Magic Wand:**

* Using WebFlux does not automatically solve all performance and scalability issues.
* Simply adopting a reactive framework like WebFlux does not guarantee that your application will scale well.
* Reactive programming requires careful planning and understanding to be effective.

**✅ Scaling Challenges and Realistic Expectations:**

* **Misconception:** Many developers assume that just using WebFlux will allow them to handle a large number of concurrent users without performance degradation.
* **Reality:** Scaling still depends on several factors, such as proper configuration, non-blocking I/O, and optimal resource usage.

## GZIP

**✅ Problem Context in Microservices Architecture:**

* In a microservices architecture, multiple services communicate over a network.
* The network could be congested, and responses may need to travel across different data centers or to third-party services (AWS, GCP, etc.).
* If the **response size is large**, it takes longer for the client to receive it, leading to:
  + Increased response time
  + Perception that the service is slow or underperforming
  + Potentially unnecessary scaling efforts (e.g., adding more servers to handle perceived load)

**✅ Gzip Compression – The Solution:**

* **What is Gzip?**
  + Gzip is a technique that compresses the response before sending it over the network.
  + It reduces the response size, thereby decreasing the time taken to transfer data.
  + The browser/client decompresses the response upon receipt.

**✅ When to Use Gzip:**

* Gzip is **beneficial in specific scenarios**:
  + When the response size is **large** (e.g., JSON data, large HTML pages, etc.).
  + When the network is **congested**, as it reduces the overall data transfer size.
* **When Not to Use Gzip:**
  + If the response size is **very small**, Gzip may increasethe payload size due to compression overhead.
  + Example: A **5-byte file** becomes **792 bytes** after Gzip due to the added headers and compression overhead.

**✅ Testing Gzip – Common Pitfall:**

* Testing Gzip **on a local machine** may not show noticeable improvements because:
  + There is **no network latency** on localhost.
  + Gzip primarily addresses **network latency**, not processing time.
  + The benefit is more pronounced in real-world networks, especially in congested or high-latency scenarios.

**✅ Important Considerations:**

* **CPU Overhead:** Compressing responses requires additional CPU processing.
* **Thresholds:** Configure Gzip to **only compress responses above a certain size** to avoid unnecessary CPU usage for small responses.
* **Monitoring:** After enabling Gzip, monitor the system for **CPU usage, memory usage, and response time** to ensure that compression is not introducing new bottlenecks.

**✅ Summary:**

* Gzip can significantly **improve throughput and response time** in network-intensive applications by reducing payload size.
* It is most effective for **large responses in congested networks** but may have a negative impact for small responses due to compression overhead.
* Testing Gzip locally may not reveal its full benefits, as network latency is not a factor on a local machine

**Server Side:**

**✅ Why Gzip Compression?**

* Gzip reduces the size of HTTP responses, thereby **reducing network latency** and **improving response times** for clients.
* However, enabling Gzip does **not automatically improve performance**. It requires proper configuration and testing to verify its effectiveness.

**✅ Important Considerations Before Enabling Gzip:**

* **Testing on a Local Machine:**
  + Testing Gzip locally may **not show noticeable improvements** because there is minimal network latency.
  + Gzip's benefits are more evident in real-world network conditions, especially over congested or high-latency networks.
* **Performance Testing:**
  + Do **not assume that Gzip will improve performance** without testing.
  + The compression process requires CPU resources, so the overall impact must be measured to avoid negative effects on server performance.

**✅ Enabling Gzip in Spring Boot:**

To enable Gzip in a Spring Boot application, you need to configure the following properties in the application.properties or application.yml file.

**1. Basic Gzip Configuration:**

**In application.properties:**

server.compression.enabled=true

server.compression.min-response-size=2048

server.compression.mime-types=application/json,application/xml

**Explanation:**

* server.compression.enabled=true: Enables Gzip compression.
* server.compression.min-response-size=2048: The **minimum response size** (in bytes) that should be compressed. In this case, 2 KB.
* server.compression.mime-types: Specifies the **MIME types** to compress. Only the specified MIME types will be compressed.

**2. Why the Minimum Response Size?**

* Gzip is most effective for larger responses (e.g., large JSON responses, HTML content).
* For small responses (e.g., a few bytes of text), the **compression overhead** can actually increase the response size due to the Gzip headers.
* Hence, setting a minimum response size avoids unnecessary CPU overhead.

**3. Client-Side Configuration – Accept-Encoding Header:**

* Even if Gzip is enabled on the server, the client must **explicitly indicate that it can handle compressed responses**.
* The client does this by sending the Accept-Encoding header in the HTTP request:

Client Side Request Header:

Accept-Encoding: gzip

* If the client **does not send this header**, the server **will not compress the response**, even if Gzip is enabled.

**4. Conditions for Gzip Compression to Work:**

Gzip will only be applied if **all the following conditions are met**:

1. server.compression.enabled=true is set.
2. The **response size is greater than or equal to the configured minimum size** (min-response-size).
3. The **response MIME type** is included in the list of compressible MIME types (mime-types).
4. The client sends the **Accept-Encoding: gzip** header.

If any of these conditions are not met, Gzip **will not be applied**.

**✅ Example: Testing Gzip with cURL:**

You can test Gzip compression using the curl command:

**Without Accept-Encoding Header:**

curl -I http://localhost:8080/api/products

**With Accept-Encoding Header:**

curl -I -H "Accept-Encoding: gzip" http://localhost:8080/api/products

* If Gzip is enabled and the response meets the criteria, the response headers will include:

Content-Encoding: gzip

**✅ Summary:**

* Gzip compression is **not automatically applied** just by setting server.compression.enabled=true.
* It requires configuration of the **minimum response size**, **MIME types**, and the client must explicitly indicate support for Gzip using the Accept-Encoding header.
* Testing should be done in **real-world network conditions**, not on localhost, to observe actual performance benefits.

## Keep alive/Connection pooling

**✅ Context and Problem Definition:**

* **Scenario:**
  + A client application needs to send multiple concurrent requests to a remote server application running on port **8080**.
* **Problem:**
  + Every time the client sends a request, it needs to establish a new connection.
  + Establishing a connection involves a **three-way TCP handshake**, which is a time-consuming process in computing terms.
  + When multiple requests are made concurrently, each request requires a new connection, leading to unnecessary overhead and resource consumption.

**✅ Understanding TCP Handshake and Port Allocation:**

1. **Connection Setup:**
   * When the client sends a request to the server, a TCP connection is established.
   * The OS allocates a **random outbound port** (e.g., 53123) for the connection.
   * The connection is defined as:

Client (localhost:53123) → Server (localhost:8080)

1. **Data Transmission:**
   * Once the connection is established, data is **written to the connection and flushed**, allowing it to reach the server.
   * The server processes the request, writes the response, and **flushes it back** to the client.

**✅ The Limitation of HTTP/1.1:**

* In **HTTP/1.1**, each connection is **dedicated to a single request-response cycle**.
* This means:
  + If a request is sent via a connection, that connection **cannot be reused** for another request until the response for the first request is received.
  + If the server is slow to respond, the connection remains occupied and cannot be reused.

**✅ Example Scenario:**

* Suppose the client has to send **three requests** to the server in quick succession.
* If the server is slow, the first request is still being processed, and the response has not yet been received.
* In this case:
  1. **First Request:**
     + Connection is established using **random outbound port 53123**.
     + Request is sent, and the client waits for the response.
  2. **Second Request:**
     + Since the first connection is still awaiting a response, a **new connection is created** using a new random outbound port (e.g., 53124).
     + This connection is also occupied until a response is received.
  3. **Third Request:**
     + The second connection is still waiting for a response, so a **third connection is established** with yet another random port (e.g., 53125).
* This process continues for every new request.
* As more requests are made, more connections are opened, consuming system resources unnecessarily.

**✅ Impact of Not Reusing Connections:**

* **Increased Resource Usage:**
  + Each connection consumes system resources (memory, CPU, ports).
* **Performance Degradation:**
  + The time taken to set up each connection adds up, leading to increased latency.
* **Network Congestion:**
  + Unused connections can remain open for some time, consuming available ports and causing congestion.

**✅ Solution: Connection Pooling and Keep-Alive:**

* **Keep-Alive (HTTP/1.1):**
  + Allows a single TCP connection to be **reused for multiple requests/responses**.
  + This reduces the overhead of establishing new connections for each request.
  + Example Header:

Connection: keep-alive

* **Connection Pooling:**
  + A pool of pre-established connections is maintained and **reused** as needed.
  + If a request is made and a connection is available in the pool, it is reused instead of establishing a new connection.
  + This minimizes the time spent in the handshake process and reduces resource consumption.

**✅ Example with WebClient (Spring WebFlux):**

* In **Spring WebFlux**, the WebClient can be configured to use a connection pool with a library like **Reactor Netty**.
* This setup creates a pool of 10 connections that can be **reused across multiple requests**, reducing the overhead of establishing new connections.

**✅ Summary:**

* **HTTP/1.1** by default uses a single connection per request-response cycle, leading to resource overhead and increased latency.
* **Keep-Alive** and **Connection Pooling** allow connections to be reused, minimizing the number of connections and reducing the time spent on handshakes.
* Proper configuration of connection pools is crucial in reactive systems like **Spring WebFlux** to maximize throughput and reduce latency.

**Recreating the scenario**

*@Test  
public void* concurrentRequests() {  
 *var* max = 10000;  
 Flux.range(1, max)  
 .flatMap(*this*::getProduct, max)  
 .collectList()  
 .as(StepVerifier::create)  
 .assertNext(l -> Assertions.assertEquals(max, l.size()))  
 .expectComplete()  
 .verify();  
}  
  
*private* Mono<Product> getProduct(*int* id) {  
 *return this*.client.get()  
 .uri("/product/{id}", id)  
 .retrieve()  
 .bodyToMono(Product.*class*);  
}

**Explanation:**

* **Flux.range(1, max)**: Generates a sequence of integers from **1 to max**.
* **flatMap()**: Sends concurrent requests using the getProduct() method for each product ID.
  + Since flatMap() is **non-blocking**, all requests are sent concurrently.
  + If the max value is 10, it sends **10 requests concurrently**.
* **collectList()**: Gathers all responses into a single List<Product>.
* **StepVerifier**: Verifies the reactive stream:
  + assertNext() checks if the size of the list matches the number of requests.
  + expectComplete() ensures that all requests have completed successfully.

**✅ Running the Test:**

* Before running the test, ensure that the **external service** (product service) is up and running on **port 7070**.
* Run the test class.

**✅ Expected Output and Analysis:**

* Since the external service is **slow (5 seconds per request)**:
  + If connection pooling is not effectively utilized, each request would sequentially take **5 seconds**, resulting in a total of **50 seconds** for 10 requests.
  + With proper connection pooling and concurrent requests using flatMap(), the requests will be sent concurrently, reducing the overall time significantly.

**✅ Summary:**

* The test demonstrates the behavior of **connection pooling** by sending multiple concurrent requests to a slow service.
* The use of **flatMap()** simulates concurrency, allowing the test to observe how the connection pool manages multiple outgoing requests.
* By validating the response size, we can confirm whether all requests were processed and received successfully.

Netstat command to monitor the network connections  
netstat -an| grep -w 127.0.0.1.7070

To watch  
watch 'netstat -an| grep -w 127.0.0.1.7070'

**✅ Understanding TCP Connections Monitoring Using netstat and watch**

This passage explains how to **monitor TCP connections** established during a connection pooling test using the netstat command. The goal is to observe how many connections are created when multiple concurrent requests are sent to a service running on port **7070**.

**✅ Commands and Tools:**

1. **netstat**:
   * A command-line tool used to display network connections, routing tables, interface statistics, and more.
   * In this context, it is used to **monitor TCP connections** to the remote service.
2. **watch**:
   * A utility that executes a command periodically, refreshing the output every **two seconds** by default.

**✅ Command Structure:**

The command used is:

watch -n 2 netstat -ant | grep 7070

* watch -n 2: Runs the command every **2 seconds**, refreshing the output.
* netstat -ant:
  + -a: Shows all connections.
  + -n: Displays addresses and port numbers in numerical format (instead of resolving names).
  + -t: Displays **TCP connections only**.
* grep 7070: Filters the output to display only connections related to port **7070**.

**✅ Initial State Check:**

* Before running the test, the command is executed:
  + No connections are established yet, so **no entries** are displayed.

**✅ Test Execution with Sleep:**

* The test is modified to **intentionally sleep for 1 minute** using:

Thread.sleep(Duration.ofMinutes(1).toMillis());

* Why this is done:
  + Normally, the test would complete in **5 seconds**, closing the connections immediately.
  + By introducing the sleep, the connections remain **active and visible** for a longer duration, allowing for better observation in netstat.

**✅ Running the Test with Single Request (max = 1):**

* The test is run with **one request**.

**Expected Output in netstat:**

* Since one request is made, **one connection** will be established.
* The output will show **two entries**:

tcp 0 0 192.168.x.x:50515 192.168.x.x:7070 ESTABLISHED

tcp 0 0 192.168.x.x:7070 192.168.x.x:50515 ESTABLISHED

* These two entries represent **one logical connection**, but they are displayed as two separate lines:
  + **Outbound Connection:** Client (local port 50515) to Server (7070).
  + **Inbound Connection:** Server (7070) to Client (50515).

**✅ Connection Termination:**

* When the test completes or is interrupted:
  + The connection state transitions to **TIME\_WAIT**, indicating that the connection is being gracefully closed.

tcp 0 0 192.168.x.x:50515 192.168.x.x:7070 TIME\_WAIT

* After some time (e.g., 1 minute), the entry will **disappear**, indicating that the connection is fully closed.

**✅ Testing with Multiple Requests (max = 3):**

* The test is modified to send **3 concurrent requests**:

int max = 3;

* Expected Output:
  + Since there are 3 concurrent requests, **3 connections** will be established.
  + The netstat output will show **6 entries** (3 outbound, 3 inbound).

tcp 0 0 192.168.x.x:50516 192.168.x.x:7070 ESTABLISHED

tcp 0 0 192.168.x.x:50517 192.168.x.x:7070 ESTABLISHED

tcp 0 0 192.168.x.x:50518 192.168.x.x:7070 ESTABLISHED

tcp 0 0 192.168.x.x:7070 192.168.x.x:50516 ESTABLISHED

tcp 0 0 192.168.x.x:7070 192.168.x.x:50517 ESTABLISHED

tcp 0 0 192.168.x.x:7070 192.168.x.x:50518 ESTABLISHED

**✅ Testing with 10 Concurrent Requests (max = 10):**

* The test is run again with **10 concurrent requests**:

int max = 10;

* Expected Output:
  + **10 connections** will be established, resulting in **20 entries** (10 outbound, 10 inbound).

**✅ Summary:**

* The purpose of the command is to observe how many TCP connections are established for a given number of concurrent requests.
* By monitoring the connections, we can **visualize the effect of connection pooling** and understand how the operating system manages outbound ports for each request.
* The usage of Thread.sleep() is a key trick to **keep the connections open** long enough to observe them in netstat

**✅ Understanding WebClient Connection Pooling and Configuration**

This explanation provides a deep dive into how **WebClient in Spring WebFlux** handles connection pooling and how we can adjust its behavior to handle concurrent requests effectively.

**✅ Background and Initial Setup:**

1. **Previous Setup:**
   * In the previous lecture, a test was run with concurrent requests using flatMap.
   * The default behavior of flatMap allows a maximum of **256 concurrent requests**.
   * Beyond this limit, requests are **queued** internally, causing delays.
2. **Current Objective:**
   * The goal is to understand how connection pooling works in WebClient and how to **adjust the connection pool size** and **pending queue size** to handle more concurrent requests efficiently.

**✅ Removing Sleep and Initial Testing:**

* The sleep is removed from the test method.
* The test is now configured to send **250 concurrent requests**.

java

CopyEdit

int max = 250;

* When the test is run, it completes in **5 seconds**, as expected.

**✅ Increasing Concurrent Requests to 260:**

* The max value is increased to **260**.

java

CopyEdit

int max = 260;

* When the test is run:
  + The first **256 requests** are sent concurrently.
  + The remaining **4 requests** are queued and only processed after the first set completes.
  + This results in the test taking **10 seconds**:
    - First 256 requests take **5 seconds**.
    - Remaining 4 requests take an **additional 5 seconds**.

**✅ Customizing FlatMap Concurrency:**

* We can adjust the concurrency level of flatMap by specifying a concurrency parameter:

java

CopyEdit

.flatMap(this::getProduct, 260)

* Setting it to **260** allows **all 260 requests** to be processed concurrently.
* The test now completes in **5 seconds**.

**✅ Increasing Concurrent Requests to 500:**

* The max value is increased to **500**.

java

CopyEdit

int max = 500;

.flatMap(this::getProduct, 500)

* The test completes in **5 seconds**, as the WebClient is capable of handling 500 concurrent connections.

**✅ Testing with 501 Requests:**

* The max value is set to **501**.
* When the test is run:
  + The first **500 requests** are sent concurrently.
  + The **501st request** is queued until one of the existing connections is released.
  + Thus, the test takes **10 seconds**:
    - First 500 requests take **5 seconds**.
    - The 501st request takes an **additional 5 seconds**.

**✅ Understanding WebClient Connection Pooling Limits:**

* The behavior observed above is due to WebClient's default **connection pool limit** of **500 connections**.
* This limit can be adjusted by configuring the connection provider.

**✅ Customizing Connection Pool Settings:**

*private final* WebClient client = createWebClient(b -> {  
 *var* poolSize = 10000;  
 *var* provider = ConnectionProvider.builder("vins")  
 .lifo()  
 .maxConnections(poolSize)  
 .pendingAcquireMaxCount(poolSize \* 5)  
 .build();  
 *var* httpClient = HttpClient.create(provider)  
 .compress(*true*)  
 .keepAlive(*true*);  
 b.clientConnector(*new* ReactorClientHttpConnector(httpClient));  
});

1. **Define Pool Size:**

*var* poolSize = 10000;

1. **Create Connection Provider:**

*var* provider = ConnectionProvider.builder("vins")  
 .lifo()  
 .maxConnections(poolSize)  
 .pendingAcquireMaxCount(poolSize \* 5)  
 .build();

* fifo() / lifo():
  + **FIFO (First-In-First-Out)**: Processes requests in the order they arrive.
  + **LIFO (Last-In-First-Out)**: Processes the most recently arrived requests first.
* **Why LIFO?**
  + It is recommended because **unused connections can be terminated quickly**, allowing the system to release resources effectively.

1. **Create HTTP Client with Connection Provider:**

*var* httpClient = HttpClient.create(provider)  
 .compress(*true*)  
 .keepAlive(*true*);

* compress(true): Adds the **Accept-Encoding** header with gzip to the request.
* keepAlive(true): Keeps the connections **open for reuse**, reducing connection setup overhead.

1. **Integrating HTTP Client with WebClient:**

b.clientConnector(*new* ReactorClientHttpConnector(httpClient));

* The **custom HTTP client** is now passed to the WebClient builder.

**✅ Testing the Configuration:**

* With the new configuration, we can now handle **600 concurrent requests** effectively.
* Example Test Run:
  + Set max to **600**.
  + All requests complete in **5 seconds**.
* If max is set to **601**, the extra request will be queued, causing the test to take **10 seconds**.

**✅ Why This Configuration?**

* By **customizing the connection provider**, we can:
  + Increase the number of concurrent connections.
  + Adjust the size of the pending request queue.
  + Control the connection release strategy (LIFO vs. FIFO).
  + Enable additional configurations like keepAlive, compress, etc.

**✅ Important Note on Connection Pooling Configuration:**

**Understanding the Context:**

* The previous lectures demonstrated how to adjust the connection pool size and pending queue size to handle a large number of concurrent requests using WebClient in Spring WebFlux.
* The purpose was to **simulate a high-latency service** that takes **5 seconds** to respond, allowing us to observe how the connection pool size and request queueing affect response times.

**✅ Key Takeaway:**

* The instructor emphasizes that **just because we can adjust the pool size and concurrency settings, it does not mean we should do it by default**.

**✅ Why Not Adjust Pool Size by Default?**

1. **Default Settings Are Optimized:**
   * Spring WebFlux is configured with sensible defaults based on common use cases.
   * By default, the connection pool size is **500 connections**, which is sufficient for most scenarios.
2. **High-Latency Scenario is Simulated:**
   * In the demo, the external service is intentionally configured to respond in **5 seconds**.
   * This is a simulated scenario to demonstrate how connection pooling works under heavy load.
3. **In Real Applications:**
   * Actual services typically respond much faster — e.g., **100 milliseconds** per request.
   * With a **100ms response time**, a single connection can handle **10 requests per second** (1000ms / 100ms = 10 requests).
   * With **500 connections**, the application can handle 500 x 10 = 5000 requests per second.

**✅ Why Adjust the Pool Size?**

* Adjusting the pool size should only be done if:
  + The remote service is **consistently slow** and cannot be optimized.
  + The application needs to handle a **higher number of concurrent requests** than the default limit.
  + There is a need to **increase throughput** in high-traffic scenarios.

**✅ Best Practices for Adjusting Pool Size:**

1. **Measure First:**
   * Before adjusting, **monitor the actual throughput and response times** to identify bottlenecks.
2. **Avoid Overprovisioning:**
   * Increasing the pool size unnecessarily can lead to **resource exhaustion**, such as:
     + Excessive memory usage.
     + Too many open connections.
     + Network congestion.
3. **Consider Service Response Time:**
   * If the service responds in **100ms**, then even with a default pool size of 500 connections, the system can handle **5000 requests per second**, which is substantial.

**✅ Summary:**

* The demo configuration was intentionally adjusted to simulate high latency and high concurrency scenarios.
* In real-world applications, adjusting the pool size may **not be necessary** unless there is a clear performance requirement.
* Always **measure actual throughput and response times** before adjusting avoid unnecessary resource consumption.

**✅ Bottleneck Analysis: Too Many Open Connections**

**Context:**

* In the previous lecture, the instructor demonstrated how to adjust the connection pool size to handle a higher number of concurrent requests using WebClient in Spring WebFlux.
* The objective was to **increase the number of concurrent requests** gradually, starting from 2000, then 3000, 5000, and finally 10,000 concurrent requests.

**✅ What Happened?**

1. **2000 Concurrent Requests:**
   * The pool size was set to **2000 connections**.
   * All requests were processed concurrently without issues.
2. **3000 Concurrent Requests:**
   * The pool size was increased to **3000 connections**.
   * Still no issues; the application handled the load.
3. **5000 Concurrent Requests:**
   * Pool size was increased to **5000 connections**.
   * The application managed to handle the load successfully.
4. **10,000 Concurrent Requests:**
   * Pool size was increased to **10,000 connections**.
   * **Problem occurred:**
     + **Socket Exception: Connection Reset**
     + **Too Many Open Files Error**
     + The operating system could not handle such a high number of open connections.

**✅ Why Did This Happen?**

* The underlying issue is that the **operating system has a limit on the number of open file descriptors**, which includes open sockets, files, and other resources.
* When the number of concurrent connections exceeds this limit, the OS starts to **reject new connections** and throws exceptions like:
  + java.net.SocketException: Too many open files
  + java.net.SocketException: Connection reset

**✅ Understanding File Descriptors:**

* A file descriptor is a **unique identifier for open files, sockets, and other resources**.
* Operating systems have a limit on the number of file descriptors that can be opened by a single process or user.
* On Linux systems, you can check this limit using:

ulimit -n

* To increase the limit, you can use:

ulimit -n 20000

Or permanently set it in /etc/security/limits.conf:

\* soft nofile 20000

\* hard nofile 20000

**✅ Key Takeaways:**

1. **Increasing the Pool Size is Not a Solution:**
   * Simply increasing the connection pool size is not a sustainable approach, as the OS has finite resources.
2. **Optimize Connection Reuse:**
   * Instead of opening 10,000 connections, **reuse existing connections**.
   * Implement connection pooling and reuse strategies effectively.
3. **Enable GZIP Compression:**
   * Enabling **GZIP compression** can **reduce response size**, leading to faster response times and quicker connection releases.

Example configuration in WebClient:

HttpClient httpClient = HttpClient.create()

.compress(true); // Enable GZIP compression

1. **Reduce Response Time:**
   * If the remote service is slow (e.g., 5 seconds), consider optimizing it or using **fallback mechanisms**.

**✅ Summary:**

* The OS has a limit on the number of open file descriptors.
* Setting a high pool size (e.g., 10,000) can easily exhaust these resources.
* The solution is to **reuse connections**, **compress responses**, and **optimize response times** to prevent resource exhaustion.

## HTTP2

**✅ Understanding HTTP/1.1 vs HTTP/2**

**✅ HTTP/1.1: The Limitations**

* **Standardized in the late 1990s.**
* **Request-Response Model:**
  + Each request requires a separate TCP connection.
  + To handle multiple concurrent requests, multiple connections must be established.
  + This leads to high resource usage, especially in microservices architecture where numerous requests are common.
* **Head-of-Line Blocking:**
  + If a request is being processed on a connection, subsequent requests must **wait until the current request is completed**, causing delays.
* **Text-Based Protocol:**
  + HTTP/1.1 uses a **text-based format**, making it less efficient in terms of parsing and data transmission.

**✅ Why the Need for HTTP/2?**

* **Google’s Challenge:**
  + With massive traffic, Google encountered the limitations of HTTP/1.1.
  + They developed an internal protocol called **SPDY (Speedy)** to address these issues.
* **SPDY to HTTP/2:**
  + The SPDY protocol laid the groundwork for HTTP/2, which was standardized in **2015**.
  + HTTP/2 is now widely adopted in modern web servers and clients.

**✅ Key Features and Benefits of HTTP/2:**

1. **Multiplexing:**
   * **Single Connection, Multiple Requests:**
     + Instead of opening multiple connections for multiple requests, HTTP/2 uses a **single TCP connection** to send and receive multiple requests concurrently.
   * **How?**
     + Each request is sent as a **stream**, and multiple streams can be processed simultaneously over the same connection.

Example:

* + In HTTP/1.1, 10 requests would require **10 separate connections**.
  + In HTTP/2, 10 requests can be handled over **1 connection** using multiplexed streams.

1. **Binary Protocol:**
   * HTTP/2 is a **binary protocol**, unlike HTTP/1.1, which is text-based.
   * **Why Binary?**
     + Binary framing is more compact, easier to parse, and less error-prone compared to text-based protocols.
2. **Header Compression:**
   * HTTP/2 uses the **HPACK compression algorithm** to reduce the size of HTTP headers.
   * This is particularly beneficial when the same headers are sent repeatedly in a session (e.g., cookies, authentication tokens).
3. **Server Push:**
   * The server can **push resources** (like CSS, JavaScript, or images) to the client **before the client even requests them**.
   * This reduces the number of round trips and speeds up page loading.
4. **Flow Control and Prioritization:**
   * HTTP/2 allows **prioritization of streams**, ensuring that critical requests are handled first.
   * It also has built-in **flow control mechanisms** to avoid congestion.

**✅ Impact on Microservices Architecture:**

* In microservices, multiple services often need to communicate concurrently.
* HTTP/1.1 would require **multiple connections**, leading to high resource usage and increased connection setup/teardown overhead.
* HTTP/2 can handle the same number of requests using **fewer connections** and **less system resources**, leading to:
  + Reduced CPU and memory usage.
  + Lower latency due to fewer connection setups.
  + Enhanced throughput due to multiplexing.

**✅ Summary:**

* HTTP/1.1 requires **one connection per request**, resulting in high resource usage.
* HTTP/2 introduces **multiplexing**, allowing multiple requests over a **single connection**, reducing the resource footprint.
* HTTP/2 also uses a **binary format**, **header compression**, and **server push**, making it **faster and more efficient** than HTTP/1.1.

**✅ Enabling HTTP/2 in Spring Boot and Testing with WebClient**

1. **Configuration:**

To enable HTTP/2, add the following property to application.properties:

server.http2.enabled=true

* This property **enables HTTP/2 support** in Spring Boot.
* When enabled, the server will support **both HTTP/1.1 and HTTP/2**, allowing the client to choose the protocol.
* This is necessary because not all clients may support HTTP/2. Thus, the server can **negotiate the protocol** based on the client capabilities.

1. **Why Both Protocols?**

* When HTTP/2 is enabled, the server will **still support HTTP/1.1** as a fallback.
* This is crucial for clients that **do not support HTTP/2**, ensuring backward compatibility.

1. **Forcing HTTP/2 Only:**

If we want to **enforce HTTP/2 only** and prevent HTTP/1.1 fallback, we can add:

server.http.enabled=false

However, in most cases, it is better to **support both protocols** to maintain compatibility with older clients.

**✅ Configuring WebClient for HTTP/2:**

* By default, WebClient uses **HTTP/1.1**.
* To enable HTTP/2, we need to explicitly configure the WebClient to use **H2C (HTTP/2 Cleartext)** or **H2 (HTTP/2 Secure)**.

**Why Two Options?**

* h2c: HTTP/2 without SSL/TLS (for development/testing).
* h2: HTTP/2 with SSL/TLS (for production).

**✅ Example Configuration in the Code:**

1. **Setup a WebClient using H2C (Cleartext):**

*public class* Lec02Http2Test *extends* AbstractWebClient {  
  
 *private final* WebClient client = createWebClient(b -> {  
 *var* poolSize = 1;  
 *var* provider = ConnectionProvider.builder("vins")  
 .lifo()  
 .maxConnections(poolSize)  
 .build();  
 *var* httpClient = HttpClient.create(provider)  
 .protocol(HttpProtocol.H2C)  
 .compress(*true*)  
 .keepAlive(*true*);  
 b.clientConnector(*new* ReactorClientHttpConnector(httpClient));  
 });  
  
 *@Test  
 public void* concurrentRequests() {  
 *var* max = 20000;  
 Flux.range(1, max)  
 .flatMap(*this*::getProduct, max)  
 .collectList()  
 .as(StepVerifier::create)  
 .assertNext(l -> Assertions.assertEquals(max, l.size()))  
 .expectComplete()  
 .verify();  
 }  
  
 *private* Mono<Product> getProduct(*int* id) {  
 *return this*.client.get()  
 .uri("/product/{id}", id)  
 .retrieve()  
 .bodyToMono(Product.*class*);  
 }  
}

* We set the pool size to **1 connection**, demonstrating HTTP/2’s multiplexing capability.
* HTTP/2 allows sending **multiple concurrent requests over a single connection**, unlike HTTP/1.1.

**✅ Testing with Concurrent Requests:**

* **Testing with 3 Requests:**
  + The instructor sends **3 requests concurrently**.
  + In HTTP/1.1, this would require **3 separate connections**.
  + With HTTP/2, only **1 connection** is used, handling all 3 requests concurrently.
* **Testing with 100 Requests:**
  + Again, with HTTP/1.1, we would need **100 connections**.
  + With HTTP/2, we still use **1 connection** to handle all 100 requests.
* **Testing with 10,000 Requests:**
  + In HTTP/1.1, attempting 10,000 connections would likely lead to resource exhaustion and socket exceptions.
  + With HTTP/2, we can handle **10,000 concurrent requests over a single connection**, thanks to multiplexing.

**✅ Performance Analysis and CPU Usage:**

* When we send **20,000 requests**, it takes **7.5 seconds** to complete.
* Why the delay?
  + Even though HTTP/2 allows all requests to be processed over one connection, the **client still has to decode 20,000 responses**, consuming CPU cycles.
  + Thus, while the network bottleneck is mitigated, the client-side processing becomes the limiting factor.

**✅ Summary and Key Takeaways:**

* HTTP/2 can handle **multiple concurrent requests over a single connection**, unlike HTTP/1.1 which requires separate connections.
* HTTP/2 uses **binary framing and multiplexing**, making it more efficient and less resource-intensive.
* WebClient in Spring Boot **uses HTTP/1.1 by default**, but can be configured for HTTP/2 using protocol() method in the HTTP client configuration.
* For development (without SSL/TLS), use **H2C**.
* For production (with SSL/TLS), use **H2**.
* Large batches of requests (e.g., 10,000 or 20,000) demonstrate the true power of HTTP/2, but client-side processing can still become a bottleneck.

## IMP Handling non-reactive item

**✅ Handling Non-Reactive Libraries in a Reactive Spring WebFlux Application**

**Context:**

* In a reactive system built using **Spring WebFlux**, we aim to use **reactive libraries** throughout the application stack (e.g., reactive drivers for databases, messaging systems, etc.).
* However, there are cases where a third-party library or legacy code may be **blocking and non-reactive**, such as JDBC, legacy HTTP clients, or other blocking APIs.

**✅ Problem:**

* If we invoke a **blocking call directly within a reactive pipeline**, it will **block the event loop thread**, defeating the purpose of reactive programming.
* The event loop thread is designed to be **non-blocking and handle multiple requests concurrently**. If it gets blocked, other requests will be stalled.

**✅ Solution: Using Mono.fromSupplier() with Schedulers.boundedElastic()**

1. **Mono.fromSupplier()**

* Converts a **synchronous blocking call** into a reactive Mono.
* The method provided to fromSupplier() is executed **asynchronously**, allowing the reactive pipeline to remain non-blocking.

**Example:**

Mono<String> result = Mono.fromSupplier(() -> {

// Simulate a blocking call (e.g., legacy HTTP call, database query)

return performBlockingOperation();

});

1. **subscribeOn(Schedulers.boundedElastic())**

* By default, the reactive pipeline operates on the **event loop thread**.
* To offload the blocking work to a separate thread, we use the subscribeOn() operator with a dedicated scheduler.

**Why Schedulers.boundedElastic()?**

* It is designed for **blocking I/O tasks**.
* It manages a pool of threads that **expands and shrinks dynamically**.
* Suitable for tasks like network calls, file I/O, and database queries that can be slow and potentially blocking.

**Updated Example:**

Mono<String> result = Mono.fromSupplier(() -> {

// Simulate a blocking call

return performBlockingOperation();

})

.subscribeOn(Schedulers.boundedElastic()); // Offload to a separate thread pool

**✅ Why Not Use the Event Loop Thread?**

* If we **do not use a separate scheduler**, the blocking call will run on the event loop thread.
* This will **block other reactive tasks**, leading to poor throughput and latency issues.

**✅ Example with Blocking I/O:**

import reactor.core.publisher.Mono;

import reactor.core.scheduler.Schedulers;

public class ReactiveExample {

public static void main(String[] args) {

Mono<String> response = Mono.fromSupplier(() -> {

// Simulating a blocking call (e.g., legacy HTTP call)

return performBlockingCall();

})

.subscribeOn(Schedulers.boundedElastic())

.doOnNext(result -> System.out.println("Result: " + result));

response.subscribe();

}

private static String performBlockingCall() {

try {

Thread.sleep(3000); // Simulate blocking I/O

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

return "Blocking call result";

}

}

* The above code will **execute the blocking call on a separate thread pool**, preventing the event loop thread from being blocked.

**✅ Java 21 Virtual Threads:**

* Java 21 introduces **virtual threads**, which provide lightweight concurrency without the overhead of traditional threads.
* If running on Java 21 or higher, we can consider using virtual threads instead of bounded elastic.

**Example with Virtual Threads:**

import reactor.core.publisher.Mono;

import reactor.core.scheduler.Schedulers;

public class VirtualThreadExample {

public static void main(String[] args) {

Mono<String> response = Mono.fromSupplier(() -> {

return performBlockingCall();

})

.subscribeOn(Schedulers.newVirtualBoundedElastic(100, 1000)); // Using virtual threads

response.subscribe(result -> System.out.println("Result: " + result));

}

private static String performBlockingCall() {

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

return "Blocking call completed using virtual threads";

}

}

* This uses a **custom scheduler** based on virtual threads, allowing for a large number of concurrent tasks without consuming a large number of OS threads.

**✅ Summary and Key Takeaways:**

* In a **reactive Spring WebFlux application**, it is essential to **avoid blocking calls on the event loop thread**.
* If a third-party library or legacy code is blocking, we can use **Mono.fromSupplier() and Schedulers.boundedElastic()** to offload the blocking work.
* **Virtual threads** in Java 21 provide a more efficient and lightweight alternative for managing blocking operations without consuming excessive OS threads.

## Summary

**✅ Summary of Techniques to Improve Application Scalability:**

**1. GZIP Compression:**

* **Problem:**
  + When the response size is large and the network is congested, the client may take longer to receive the response, leading to blocked connections. This affects the overall throughput.
* **Solution:**
  + Enable **GZIP compression** to reduce the response size and improve transmission speed.
* **Recommendation:**
  + Conduct **production-grade performance tests** to assess the actual impact of enabling GZIP. Testing on a local machine may not provide accurate results.

**2. Connection Pooling and Keep-Alive:**

* **Problem:**
  + Setting up a new connection for each request incurs significant overhead, reducing application throughput.
* **Solution:**
  + Enable **connection pooling** to reuse existing connections instead of creating new ones.
  + Enable **Keep-Alive** to maintain connections across multiple requests.
* **WebClient Configuration:**
  + Spring WebClient automatically handles connection pooling, but it can be **tuned** for specific use cases.
* **Formula for Calculating Pool Size:**
  + Number of Connections / Average Response Time
  + Example:
    - Pool Size: 500 connections
    - Average Response Time: 100ms
    - Throughput: (500 / 0.1) = 5000 requests per second
  + **Considerations:**
    - Before adjusting pool size, ask yourself: **Do you actually need to handle more than 5000 requests per second?**
* **Caveat:**
  + The **operating system has finite resources** and cannot handle an arbitrary number of connections. Requesting 10,000 or 20,000 connections may lead to issues like SocketException or Too Many Open Files.

**3. HTTP/2 and Multiplexing:**

* **Problem with HTTP/1.1:**
  + Requires **one connection per request**, consuming a large number of connections for concurrent requests.
* **Solution:**
  + Use **HTTP/2**, which supports **multiplexing**, allowing multiple requests and responses to be sent concurrently over a **single connection**.
* **Advantages of HTTP/2:**
  + Reduced number of connections.
  + Improved network utilization.
  + Header compression for smaller payload sizes.

**4. Avoid Blocking the Event Loop Thread in WebFlux:**

* **Problem:**
  + If a blocking call is executed on the event loop thread, it will block other reactive tasks, reducing concurrency.
* **Solution:**
  + Use **reactive drivers** wherever possible (e.g., Reactive MongoDB, Redis, etc.).
  + For blocking libraries, use:
    - Mono.fromSupplier() to wrap the blocking call.
    - subscribeOn(Schedulers.boundedElastic()) to offload blocking work to a separate thread pool.

**Example:**

java

CopyEdit

Mono<String> result = Mono.fromSupplier(() -> {

return performBlockingOperation();

}).subscribeOn(Schedulers.boundedElastic());

**5. Additional Recommendations and Tools:**

* **Redis:**
  + Fast, in-memory data store for **caching and data management**.
  + Reduces response times for frequently accessed data.
* **GraphQL:**
  + Efficient data querying and transfer mechanism.
  + Reduces the number of API calls by allowing the client to specify the exact data structure required.
* **Design Patterns for Microservices:**
  + Helps handle **distributed transactions, resiliency, and scalability**.
  + Examples include **Circuit Breaker, Bulkhead, Saga, and Retry** patterns.

**✅ Summary:**

* Use **GZIP** to reduce payload size in congested networks.
* Use **connection pooling** to minimize connection overhead.
* Consider **HTTP/2** for multiplexing multiple requests over a single connection.
* Use **bounded elastic scheduler** to handle blocking operations in a reactive pipeline.
* Evaluate **Redis, GraphQL, and design patterns** for improving overall application architecture and scalability.

# Section 13 reactive microservice

**✅ Assignment Overview: Developing Reactive Microservices for a Trading Platform**

**High-Level Overview:**

We are developing a reactive trading platform with the following services:

* **Stock Service (Port: 7070)**
* **Customer Service (Port: 6060)**
* **Aggregator Service (Port: 8080)**

The objective is to build a reactive system using Spring WebFlux, focusing on streaming data, event-driven architecture, and non-blocking communication.

**✅ Service Details and Workflow:**

**1. Stock Service (Port 7070) – External Service:**

* **Role:** This service emits **stock price changes periodically**, simulating market data.
* **Example Data Emitted:**
  + Google: $10
  + Amazon: $8
  + This data is continuously streamed using **Server-Sent Events (SSE)** or **WebFlux Flux**.
* **Implementation Note:**
  + You **do not implement this service.**
  + Assume it is a third-party service providing real-time stock price updates.

**2. Aggregator Service (Port 8080):**

* **Role:** Acts as a **backend-for-frontend (BFF)** and the main orchestrator of the trading system.
* **Responsibilities:**
  + **Consume Stock Prices:** Subscribes to the stock service SSE endpoint to get real-time stock price changes.
  + **Expose SSE Endpoint:** Emits stock price changes to clients as SSE (or WebFlux Flux).
  + **Trade Processing:**
    - Receives trade requests from customers (e.g., buying or selling stocks).
    - Calls the stock service to **fetch the current price** of the requested stock.
    - Sends the trade request along with stock data to the **Customer Service** for processing.

**3. Customer Service (Port 6060):**

* **Role:** Manages customer accounts, balances, and portfolios.
* **Responsibilities:**
  + **Validate Trade Requests:**
    - Check customer balance for buy requests.
    - Check stock availability for sell requests.
  + **Manage Customer Portfolio:**
    - Keep track of owned stocks and their quantities.
  + **Profile API:**
    - Provide customer profile data, including current stock holdings and account balance.

**✅ Workflow Example:**

1. **Stock Price Streaming:**
   * The **stock service** emits real-time price updates as SSE (e.g., Google: $10, Amazon: $8).
   * The **aggregator service** consumes these updates and forwards them to clients.
2. **Customer Initiates Trade:**
   * Customer sends a **buy request** for 10 Google stocks to the aggregator service.
   * Aggregator service:
     + Calls the stock service to get the **current price of Google** stock.
     + Sends the trade request and stock price to the **customer service**.
3. **Customer Service Processes Trade:**
   * Checks if the customer has sufficient balance to buy 10 Google stocks.
   * If yes, the trade is fulfilled, and the customer's portfolio is updated.
   * If no, the trade is rejected with an appropriate error response.
4. **Portfolio Management:**
   * Customers can **view their profile** to see the current stock holdings and balance.

**✅ Port Assignment:**

* **7070:** Stock Service (External, already running)
* **6060:** Customer Service (Reactive, to be implemented)
* **8080:** Aggregator Service (Reactive, to be implemented)

**✅ Next Steps:**

* Next, we will define the **data model and API endpoints** for the **Customer Service**.
* This includes database structure, request/response payloads, and service interfaces.

# Important

**R2DBC - Entity Callback** <https://www.vinsguru.com/r2dbc-entity-callback/>

**Spring WebFlux - File Upload Example** [**https://www.vinsguru.com/spring-webflux-file-upload/\**](https://www.vinsguru.com/spring-webflux-file-upload/\)

**Spring WebFlux Security** [**https://www.vinsguru.com/spring-webflux-security/**](https://www.vinsguru.com/spring-webflux-security/)