Mono and Flux - Full Tutorial with Operators

---

1. What is Mono?

`Mono<T>` is a reactive type that represents a stream of 0 or 1 item. It's useful for asynchronous operations that return a **single value**.

Creating a Mono:

```
Mono<String> mono = Mono.just("Hello");
```

Empty or Error Mono:

```
Mono.empty();

Mono.error(new RuntimeException("Something went wrong"));
```

Common Mono Operators:

- **map()** – Transforms the item:

```
Mono.just("hello").map(String::toUpperCase); // "HELLO"
```

- **flatMap()** – Flattens nested Monos:

```
Mono.just("hello").flatMap(this::asyncMethod);
```

- **filter()** – Filters the item:

```
Mono.just("hello").filter(s -> s.length() > 3);
```

- **then()** – Ignores the result, returns `Mono<Void>`:

```
Mono.just("value").then();
```

- **zipWith()** – Combines results from two Monos:

```
Mono<String> mono1 = Mono.just("A");
Mono<String> mono2 = Mono.just("B");
mono1.zipWith(mono2, (a, b) -> a + b); // "AB"
```

---

2. What is Flux?

`Flux<T>` represents a stream of 0 to N items. It’s useful for working with **sequences** of data.

Creating a Flux:

```
Flux<String> flux = Flux.just("One", "Two", "Three");
```

Common Flux Operators:

- **map()**:

```
Flux.range(1, 5).map(i -> i * 2); // 2, 4, 6, 8, 10
```

- **filter()**:

```
Flux.range(1, 10).filter(i -> i % 2 == 0); // 2, 4, 6, 8, 10
```

- **flatMap()**:

```
Flux.just("a", "b").flatMap(this::asyncFetch);
```

- **zip()**:

```
Flux<String> f1 = Flux.just("A", "B");
Flux<String> f2 = Flux.just("1", "2");
Flux.zip(f1, f2, (a, b) -> a + b); // "A1", "B2"
```

- **concat()**:

```
Flux.concat(Flux.just("A"), Flux.just("B")); // A, B
```

- **merge()**:

```
Flux.merge(flux1, flux2); // Interleaves values
```

- **delayElements()** – adds delay per element.
- **take(n)**:

```
Flux.range(1, 10).take(3); // 1, 2, 3
```

---

3. Error Handling Operators

- **onErrorReturn()**:

```
Mono.error(new Exception()).onErrorReturn("fallback");
```

- **onErrorResume()**:

```
Mono.error(new Exception()).onErrorResume(e -> Mono.just("fallback"));
```

- **retry()**:

```
Mono.just("data").retry(3);
```

---

4. Combination Operators

- **zip()** – Combine multiple publishers.
- **combineLatest()** – Emit when any source emits.
- **concat()** – Wait for first to complete before next.
- **merge()** – Emits as data comes in.
- **then()** – Ignore previous result and proceed.

---

5. Blocking (Use Sparingly)

Blocking should be avoided in reactive streams but is sometimes used in testing:

```
String value = Mono.just("Hi").block();
List<String> list = Flux.just("A", "B").collectList().block();
```

---

6. Common Use Cases

- **Mono**:
  - Fetch user by ID
  - Login request
  - Save data
- **Flux**:
  - Load all users
  - Stream logs
  - Live data feeds (WebSocket)