Reactive Programming

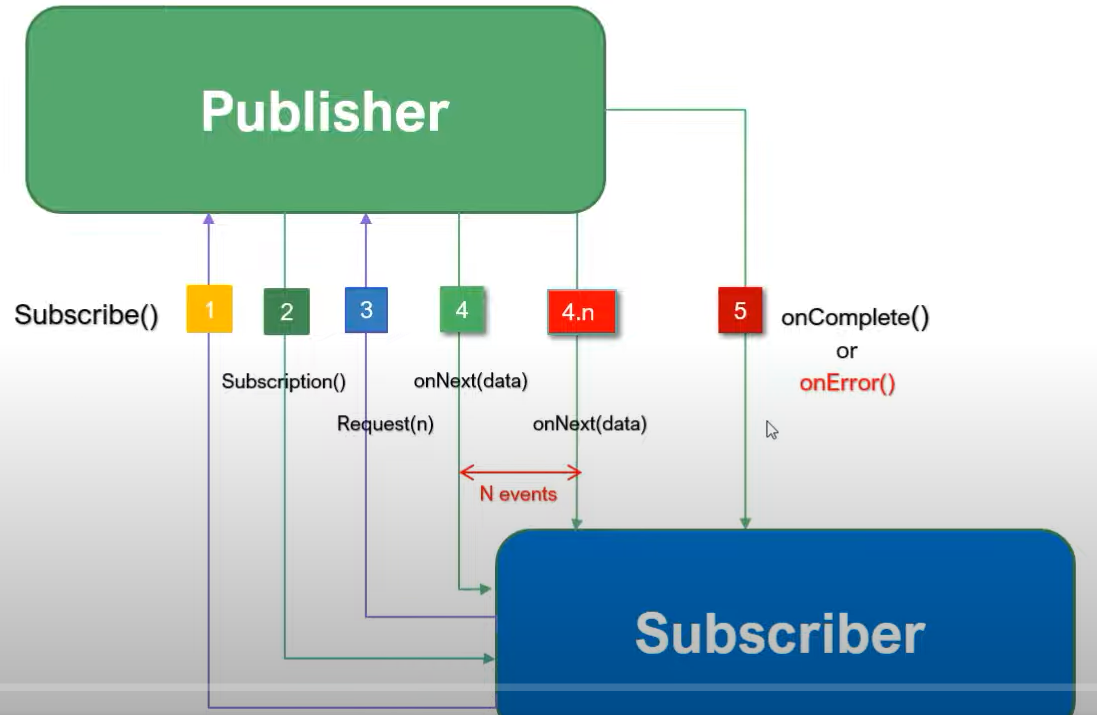
Core Features of Reactive Programming

* New Programming Paradigm
* Asynchronous and NonBlocking
* Functional Style code
* Data Flow as event driven stream
* Backpressure on data streams

Usually, the traditional Rest API will use one thread per call. But in Reactive programming the thread will call the event, the event will wait for DB to give the response and at last it will give the response for the client

A diagram of a process

Description automatically generated



A screenshot of a computer

Description automatically generated

In reactive programming, Mono and Flux are core types provided by Project Reactor, which is a reactive library for building non-blocking applications on the JVM. These types represent streams of data that can be manipulated and consumed in a reactive and declarative way. Understanding Mono and Flux is essential for working with reactive streams in Java applications.

Mono

Mono<T> is a reactive type that represents a single value or no value (0..1). It is used when you expect to work with a single item or a single event, such as retrieving a single record from a database or making an HTTP request that returns a single response.

Key Characteristics of Mono

Single Value: Emits zero or one item. If no item is emitted, it completes without a value.

Completion: Completes successfully with a value or completes empty (without a value), or with an error.

Common Use Cases: Fetching a single record, handling single-value computations, HTTP GET requests for single resources, etc.

Mono<String> mono = Mono.just("Hello, World!");

mono.subscribe(

value -> System.out.println("Received: " + value),

error -> System.err.println("Error: " + error),

() -> System.out.println("Completed")

);

---------------------------------------------------------------------------------------------------------------

Flux

Flux<T> is a reactive type that represents a stream of multiple values (0..N). It is used when you expect a sequence of items, such as retrieving a list of records from a database or handling a continuous stream of events like sensor data.

Key Characteristics of Flux

Multiple Values: Emits zero, one, or many items.

Completion: Completes successfully after emitting all items, or completes with an error.

Common Use Cases: Fetching lists or collections of data, streaming data, handling multiple events, etc.

Flux<String> flux = Flux.just("A", "B", "C");

flux.subscribe(

value -> System.out.println("Received: " + value),

error -> System.err.println("Error: " + error),

() -> System.out.println("Completed")

);

Mapping: Transform the items emitted by a Mono or Flux.

Mono<Integer> mono = Mono.just("Hello").map(String::length);

Flux<Integer> flux = Flux.just("Hello", "World").map(String::length);

Filtering: Emit only items that match a predicate.

Flux<Integer> numbers = Flux.range(1, 10).filter(n -> n % 2 == 0);

Combining: Combine multiple Monos or Fluxes.

Mono<String> mono1 = Mono.just("Hello");

Mono<String> mono2 = Mono.just("World");

Mono<String> combinedMono = mono1.zipWith(mono2, (a, b) -> a + " " + b);

Flux<String> flux1 = Flux.just("A", "B");

Flux<String> flux2 = Flux.just("1", "2");

Flux<String> combinedFlux = flux1.zipWith(flux2, (a, b) -> a + b);

Handling Errors: Provide fallback values or alternative streams.

Mono<String> monoWithError = Mono.error(new RuntimeException("Error"));

Mono<String> fallbackMono = monoWithError.onErrorReturn("Fallback");

Flux<String> fluxWithError = Flux.error(new RuntimeException("Error"));

Flux<String> fallbackFlux = fluxWithError.onErrorResume(e -> Flux.just("Fallback1", "Fallback2"));

ProductDto instead to Product

Using a ProductDto (Data Transfer Object) instead of directly using the Product class has several important benefits, particularly in a layered architecture. Here are the key reasons for using a ProductDto:

1. Separation of Concerns

Data Model vs. Business Model:

Product Entity: Typically represents the business model and is closely tied to the database schema. It may contain various annotations for persistence (e.g., JPA or Spring Data annotations).

ProductDto: Represents the data structure that is transferred between different layers of the application (e.g., between the controller and service layers) and is often designed to be decoupled from the underlying database schema.

2. Security

Data Exposure Control:

ProductDto: Allows you to control which fields are exposed to the client. Sensitive information (e.g., internal IDs, passwords, internal status flags) can be excluded from the DTO, ensuring that only necessary data is sent over the network.

3. API Stability

Independent API Layer:

ProductDto: Changes in the database schema or internal business logic won't directly affect the API exposed to clients. This decoupling allows for internal changes without breaking external contracts.

4. Data Transformation and Validation

Preprocessing and Validation:

ProductDto: Can be used to perform data transformation and validation before data is processed by the business logic or before sending data to clients. This can include formatting data, setting default values, or validating input data.

5. Performance

Optimized Data Transfer:

ProductDto: Can be designed to include only the necessary fields needed for specific operations, which can reduce the amount of data transferred over the network. This can lead to performance improvements, especially for large or complex objects.

6. Flexibility

Different Representations for Different Use Cases:

ProductDto: Allows you to create multiple DTOs for different use cases. For example, a ProductSummaryDto for listing products and a ProductDetailDto for detailed product information.

7. Mapping Between Layers

Layer Decoupling:

ProductDto: Facilitates the mapping between different layers of the application (e.g., mapping between the database layer and the presentation layer). This helps in maintaining clean code and adhering to the Single Responsibility Principle.

Usage of DTO instead of Entity

Using DTO (Data Transfer Object) classes instead of directly using Entity classes, even when they have the same fields, provides several significant benefits. Here are the key reasons for using DTOs in this context:

1. Decoupling Between Layers

Layered Architecture:

Entity: Typically represents the database schema and includes annotations and configurations related to persistence (e.g., JPA or Hibernate annotations).

DTO: Represents the data structure used for communication between different layers of the application, such as between the service layer and the presentation layer (e.g., controllers).

This decoupling allows each layer to evolve independently without affecting the other layers. Changes in the database schema or entity mappings do not directly impact the API or the data transfer logic.

2. Security

Data Exposure Control:

Entity: Might contain sensitive information or internal fields that should not be exposed to the client.

DTO: Allows you to expose only the necessary fields to the client, ensuring that sensitive information is kept internal.

3. API Stability

Independent API Layer:

Entity: Changes in the entity class, such as adding or removing fields, can break the API contracts with clients.

DTO: Ensures that the API contracts remain stable even if the underlying entity changes. The DTO structure can be designed to match the needs of the clients and remain consistent.

4. Data Transformation and Validation

Preprocessing and Validation:

Entity: Directly using entities for data transfer can make it difficult to enforce validation and transformation rules.

DTO: Provides a convenient place to apply validation and transformation rules before passing data to the business logic or before sending data to clients.

5. Performance

Optimized Data Transfer:

Entity: May contain more fields than necessary for a particular operation, leading to inefficient data transfer.

DTO: Can be designed to include only the fields needed for a specific operation, reducing the amount of data transferred and improving performance.

6. Flexibility

Different Representations for Different Use Cases:

Entity: Typically reflects the database structure and is not optimized for different use cases.

DTO: Allows creating multiple DTOs tailored for different use cases, such as a summary view, detailed view, or specific views for different client applications.