## **Understanding Reactive Programming**

### **What is Reactive Programming?**

**Declarative programming** paradigm focused on working with **data streams** and handling the **propagation of changes** efficiently.

1. **Declarative vs. Imperative:**
   * **Imperative:** You explicitly define each step to execute.
   * **Declarative:** You specify what you want to happen, and the system manages execution.
2. **Data Streams:**
   * A sequence of data elements over time.
   * Example: User events (clicks, inputs), API responses, database changes.
3. **Propagation of Change:**
   * Ensures that updates flow efficiently through the system.
   * Example: If a data source updates, all dependent components receive the update automatically.

### **Reactive vs. Asynchronous Programming**

* **Reactive Programming** does not always mean **asynchronous programming**.
* **Asynchronous:** Tasks are executed independently, not sequentially.
* **Reactive:** It can be synchronous or asynchronous but focuses on reacting to data changes.

### **Use Cases for Reactive Programming**

#### **1. User Events Handling (Frontend Development)**

* Example: Button clicks, form inputs, and event listeners in JavaScript.
* When a user clicks a button, an action is triggered immediately.

#### **2. I/O Operations (Backend Development)**

* Example: **Handling API requests** and database queries.
* When an API request is sent, once the response is ready, the **necessary action** is taken.

#### **3. Event-Driven Systems**

* Systems that react to events, such as **real-time data streaming** or **message queues**.

### **In Backend Development**

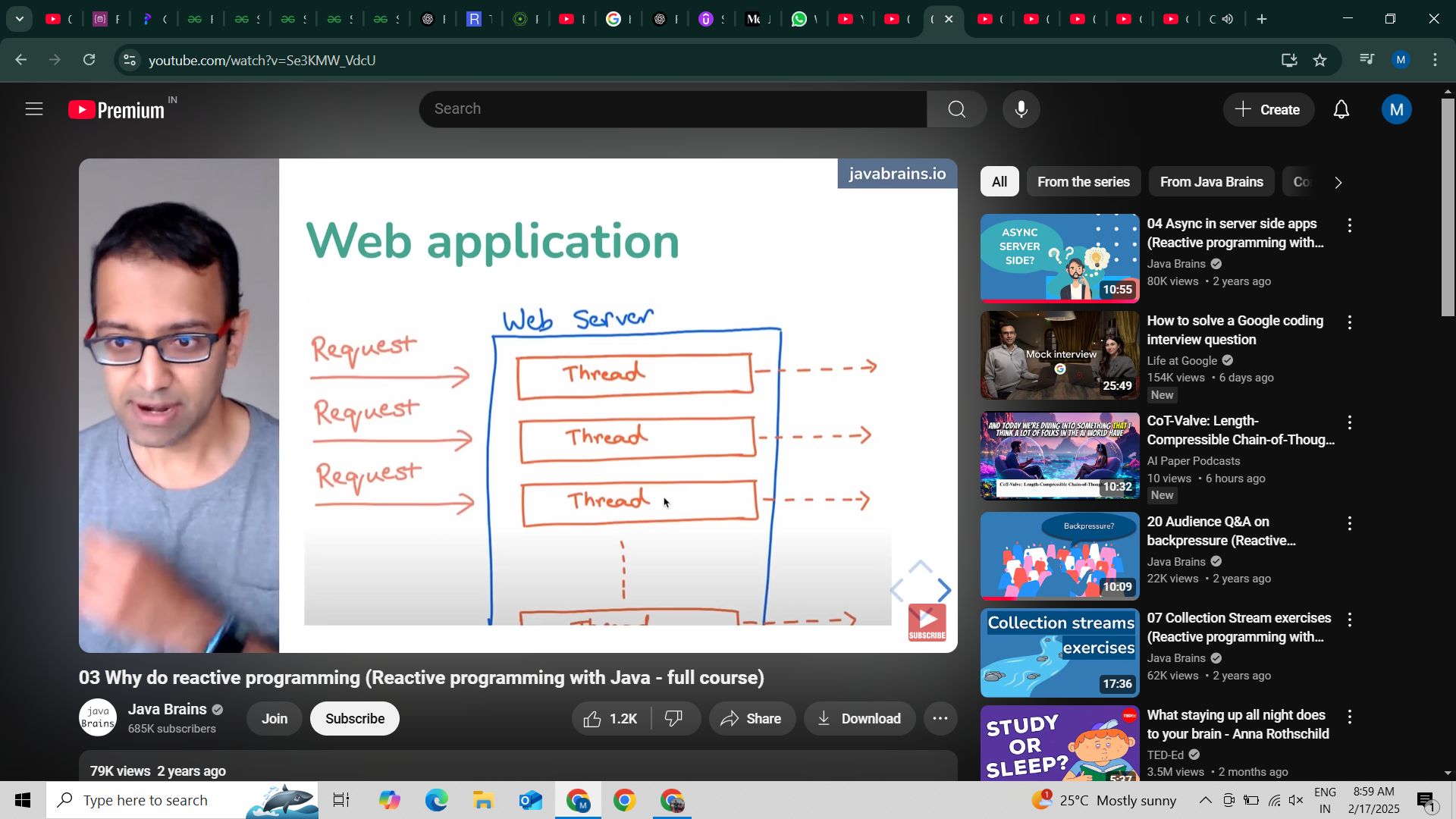
#### **Why Should Backend Developers Care?**

1. **Efficient Resource Utilization:**
   * Traditional synchronous programming **blocks execution until a task is completed**.
   * Reactive programming allows **non-blocking execution**, improving system efficiency.
2. **Scalability:**
   * Supports handling a large number of **concurrent requests** efficiently.
   * Example: **Spring WebFlux** in Java enables reactive, non-blocking web services.
3. **Handling High-Load Applications:**
   * Useful in applications requiring real-time updates, like stock market applications or chat systems.

### **How Reactive Programming Works in Web Development**

1. **Request Comes In**: A user makes an HTTP request.
2. **Processing Starts**: The request triggers various tasks, such as fetching data from the database or calling an API.
3. **Non-Blocking Execution**: Instead of waiting, other tasks continue executing.
4. **Response is Sent**: Once all tasks are completed, the response is sent back to the user.

**Scaling Approaches**:

* **Vertical Scaling**: Adding more power to a single machine (limited scalability).
* **Horizontal Scaling**: Adding more servers (preferred in modern systems).
* **Optimization First**: Before scaling, optimize code and resource utilization.

**Code Efficiency Issue**:

* **Blocking Calls**: Sequential API calls (getUser(), getPreferences()) wait for completion.
* **Unnecessary Waiting**: No dependency between calls, but still executed sequentially.
* **Performance Cost**: Slower response time due to inefficient execution.

**Solution**:

* **Parallel Execution**: Execute independent service calls concurrently.
* **Async Programming**: Use non-blocking approaches for better performance.

# **Java Concurrency in Spring Boot with CompletableFuture**

## **1. The Problem: Blocking Calls in Spring MVC**

Spring MVC handles each request using a **separate thread** from the thread pool. However, many backend operations involve **blocking calls**, such as:

* Database queries
* External API calls
* File I/O operations

When a request involves these operations, the thread remains **idle** while waiting for a response. This results in **wasted resources** and limits the application’s **scalability**.

## **2. Solution: Using CompletableFuture for Asynchronous Execution**

**CompletableFuture** is part of Java's java.util.concurrent package and allows executing tasks asynchronously, improving performance by **utilizing more CPU cores** efficiently.

## **3. Combining Multiple CompletableFutures**

When multiple tasks need to run in parallel and combine results, thenCombine() or allOf() can be used.

### **Example: Running Multiple Tasks in Parallel**

public CompletableFuture<String> fetchDataFromAPI() {

return CompletableFuture.supplyAsync(() -> {

return "API Data";

});

}

public CompletableFuture<String> fetchDataFromDatabase() {

return CompletableFuture.supplyAsync(() -> {

return "Database Data";

});

}

public CompletableFuture<String> processData() {

return fetchDataFromAPI()

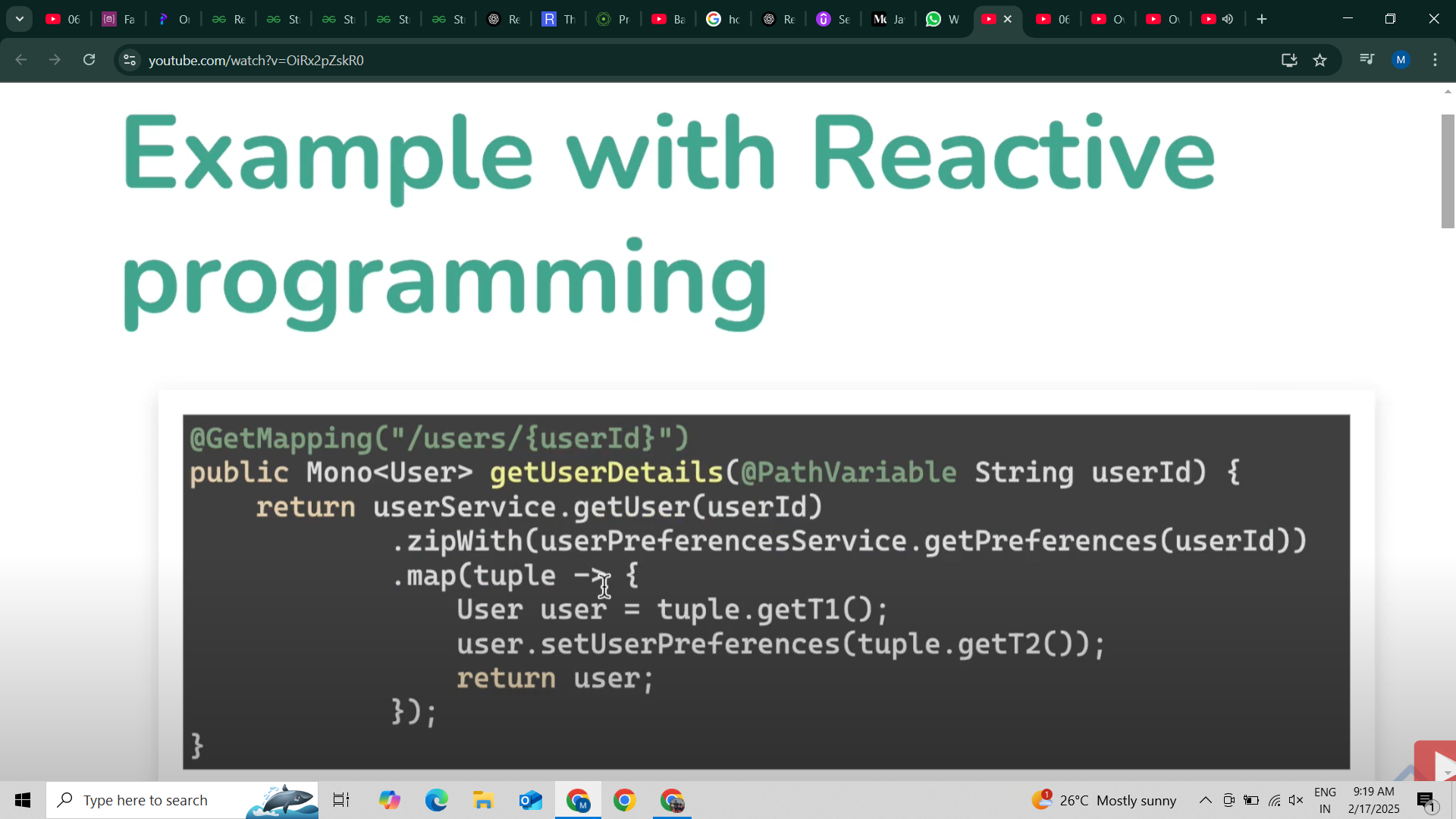
.thenCombine(fetchDataFromDatabase(), (apiData, dbData) -> apiData + " + " + dbData);

}

## **4. When to Use Reactive Programming (Project Reactor)**

While CompletableFuture is great for simple async tasks, **Project Reactor (Spring WebFlux)** provides a **more powerful alternative** using Mono and Flux for reactive programming.

**Reactive Programming - Example**



## **Java Streams**

* Provides a modern and efficient way to work with collections of data in a functional programming style.
* Focuses on the *computation* of data rather than its *storage*.
* Easy to handle operations like filtering, transforming, & processing data in a cleaner & more readable way.

### **Key Concepts of Streams:**

* **Stream:** Represents a sequence of elements supporting sequential and parallel aggregate operations.
* **Functional Programming:** Streams enable the use of functional programming techniques like map, filter, and flatMap.
* **Internal Iteration:** Unlike collections where you manually control iteration (using for loops), streams handle iteration internally, allowing you to focus on processing the elements.

## **Common Stream Operations**

1. **map:** Transforms each element in the stream.
   * Example: stream.map(element -> element \* 2); (Doubles each element in the stream)
2. **filter:** Filters out elements that don't meet a given condition.
   * Example: stream.filter(element -> element > 5); (Keeps only elements greater than 5)
3. **flatMap:** Flattens nested streams into a single stream.
   * Example: stream.flatMap(list -> list.stream()); (Flattens a list of lists into a single stream)
4. **forEach:** Performs an action on each element in the stream.
   * Example: stream.forEach(element -> System.out.println(element));
5. **findFirst:** Returns the first element that matches a condition.
   * Example: stream.filter(element -> element > 10).findFirst();
6. **collect:** Collects the results of a stream into a collection (e.g., List, Set).
   * Example: stream.collect(Collectors.toList());

**Reactive Streams API in Java**

**Key Abstractions in Reactive Streams:**

* **Publisher:** **Source** of events
* **Subscriber:** A **consumer** of events
* **Subscription:** **Controls** the flow of data between publishers & subscribers.

**Flow API:**

* Java 9 introduced the Flow API, which includes the Publisher, Subscriber, Subscription, and Processor interfaces.
* This API supports stream-oriented publish-subscribe patterns with two design patterns:
  + **Iterator Pattern (Pull):** Subscribers request a certain number of items at a time from the publisher.
  + **Observer Pattern (Push):** Publishers push events to subscribers as they occur.

**Key Methods and Flow Control:**

* The Publisher calls the subscribe() method to send data to the Subscriber.
* Subscriber methods like onNext(), onError(), and onComplete() handle incoming data, errors, and completion signals.
* Flow control is implemented by Subscription, which allows the subscriber to request a certain number of items from the publisher.

**Event Flow:**

* The subscriber subscribes to a publisher via subscribe(), triggering data flow.
* Data flow only happens after the subscription is made, following the **lazy** nature of reactive streams.
* Once the subscriber requests data, the publisher sends data using the onNext() method until either all data is sent (onComplete()) or an error occurs (onError()).

### **Backpressure in Systems**

#### **Introduction**

Backpressure is a resistance in the flow of data within a system. It occurs when data is produced faster than it can be consumed, leading to system inefficiencies, memory overload, or failures.

#### **Backpressure in File Processing**

* **Scenario:** Reading a file is often faster than writing it.
* **Problem:** If reading is too fast, the system accumulates data that it cannot write quickly.
* **Solution:** Implement controlled reading, ensuring the system reads only as much as it can write without causing memory overflow.

#### **Backpressure in Microservices**

* **Scenario:** A microservice (App A) sends requests to another (App B).
* **Problem:** If App B can process 75 requests per second but App A sends 100, the excess 25 requests cause backpressure.
* **Solution:** Implement strategies to balance the request rate.

##### **Backpressure Strategies**

1. **Buffering:** Store unprocessed requests temporarily.
   * ⚠ **Risk:** Uncontrolled buffering can lead to memory leaks.
2. **Source Control:** Limit requests at the source to prevent overload.
   * ✅ **Best Practice:** Reduces unnecessary system stress.
3. **Ignoring Requests:** Skip redundant or low-priority requests.
   * 🔥 **Use Case:** Useful when real-time data is unnecessary.

#### **Streams and Backpressure**

* **Pull Streams:** The consumer dictates the rate of data intake.
* **Push Streams:** The producer sends data but checks if the consumer is ready.

Backpressure management is crucial for **system stability**. Implementing proper strategies like buffering, source control, and **reactive programming** helps prevent performance degradation and ensures smooth data processing.

