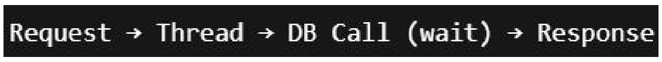
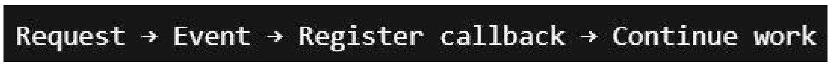
- What is the use of Reactive?
  - Traditional (Blocking) backend:

    ```
    Request → Thread → DB Call (wait) → Response
    ```

    - One request: one thread
    - Threads are expensive.
    - Waiting wastes CPU (waits till the process is completed).
    - Not scalable for I/O-heavy apps.
  - Reactive (Non-Blocking) backend:

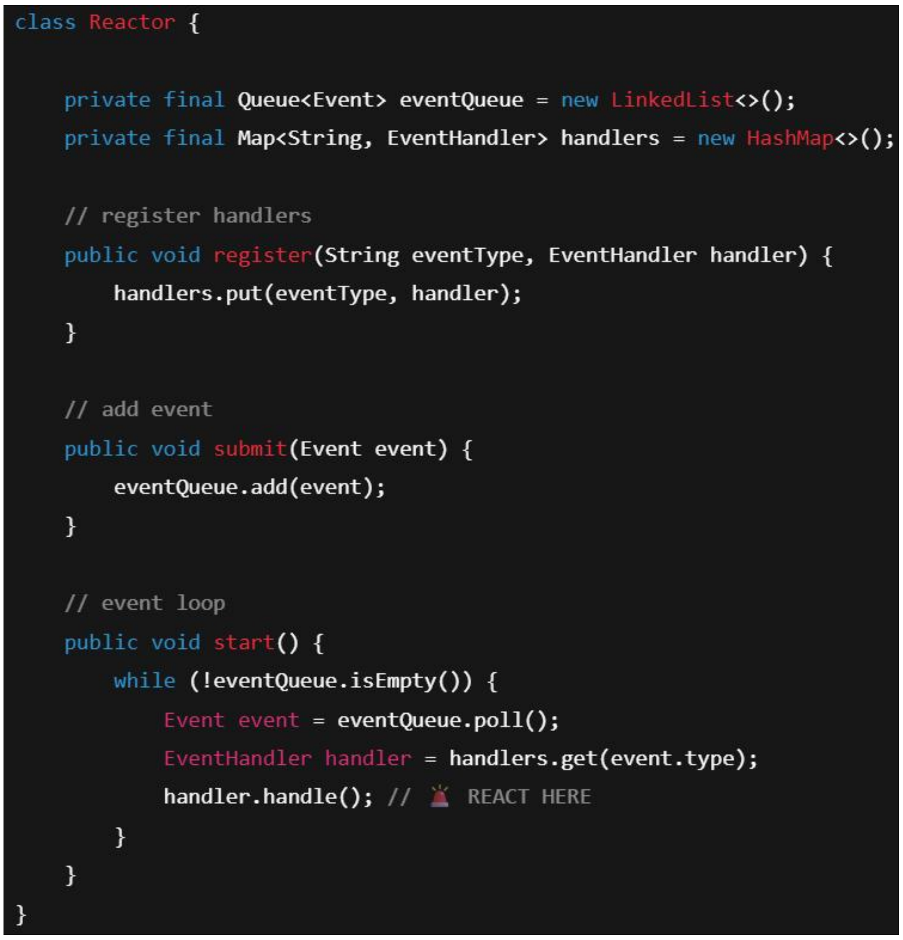
    ```
    Request → Event → Register callback → Continue work
    ```

    - Fewer threads.
    - Better CPU utilization.
    - Perfect for *APIs, Streaming, WebSockets.*
    - Highly scalable.
  - Reactive programming is basically:

    ”*Tell me when data arrives*” instead of “*Give me data now*”.

➢ What is Reactor?

- Reactor is just a design pattern; there is nothing called Reactor in Core Java.
- Reacter contains 2 things: **event queue**, **event handlers**.
- **event** **----** **event handler** === One-to-One mapping.
- For a specific event, its callback will be executed.
- For example 2 methods are there *login*, *signup*,
  - Instead of calling directly like **login()** and **signup()**, create 2 events of type 'LOGIN' & 'SIGNUP', and 2 handlers (*functional interface)* ***loginEventHandler*** and ***signupEventHandler*** .
  - Create a **reactor** object passing the events (loginEvent, signupEvent) & **callbacks** (handlers) (loginEventHandler, signupEventHandler).
  - Now when you start the reactor [**reactor.start()**] it'll *fetch the events from the event queue & run their respective handlers one-by-one*, this is reactive programming.
- Example



```java
class Reactor {

    private final Queue<Event> eventQueue = new LinkedList<>();
    private final Map<String, EventHandler> handlers = new HashMap<>();

    // register handlers
    public void register(String eventType, EventHandler handler) {
        handlers.put(eventType, handler);
    }

    // add event
    public void submit(Event event) {
        eventQueue.add(event);
    }

    // event loop
    public void start() {
        while (!eventQueue.isEmpty()) {
            Event event = eventQueue.poll();
            EventHandler handler = handlers.get(event.type);
            handler.handle(); // 🚨 REACT HERE
        }
    }
}
```

- This is just a normal implementation, you can make your own type of implementation.

- Reactor itself is ***not async or non-blocking***.

  Async and non-blocking behavior comes from the event source, like Java NIO Selector.

  Reactor only dispatches events when they are ready.

Reactor in Spring

Libraries & Usage

- An open-source library **Project Reactor** built for JVM is used by Spring.
- There are 4 interfaces being used in Spring Reactive Programming:
  - **Producer** : creates data
  - **Publisher** : exposes data stream
  - **Subscriber** : consumes data
  - **Subscription** : Connection + Control
  - **Subscriber never pulls data directly; Publisher sends data only when *requested*.**
  - Blocking:
    ```
    List<User> users = userRepository.findAll();
    ```
  - Reactive:
    ```
    Flux<User> users = userRepository.findAll();
    ```
- It provides 2 core types: **Mono**<T>, **Flux**<T> (these implements **Publisher**)
- **Mono** : **1** or None result.

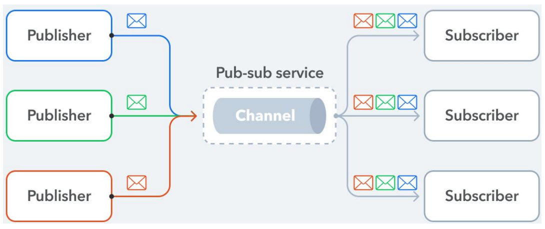
```
Mono<String> mono = Mono.just("Hello");
```

```
mono.subscribe(value -> System.out.println(value));
```

- **Flux** : **1** or More result.

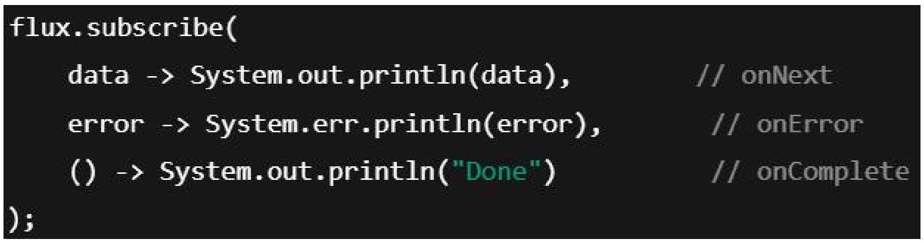
```
Flux<Integer> flux = Flux.just(1, 2, 3, 4);
```

```
flux.subscribe(System.out::println);
```

- 

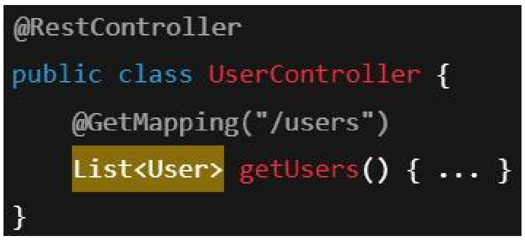
-
```
subscribe()
 ↓
onSubscribe(subscription)
 ↓
request(n)
 ↓
onNext(data)
 ↓
onNext(data)
 ↓
onComplete()
```

onNext() will be called **n** times if the flux contains **n** items.

- 

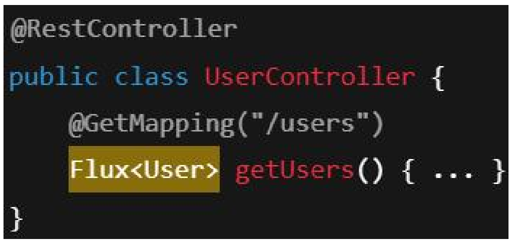
```
flux.subscribe(
    data -> System.out.println(data),         // onNext
    error -> System.err.println(error),       // onError
    () -> System.out.println("Done")          // onComplete
);
```

- **Normal usecase**



```
@RestController
public class UserController {
    @GetMapping("/users")
    List<User> getUsers() { ... }
}
```

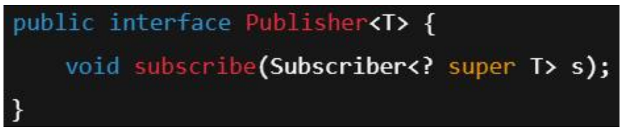
- **Reactive usecase**



```
@RestController
public class UserController {
    @GetMapping("/users")
    Flux<User> getUsers() { ... }
}
```

➢ The interfaces (Publisher, Subscriber, Subscription, Processor)

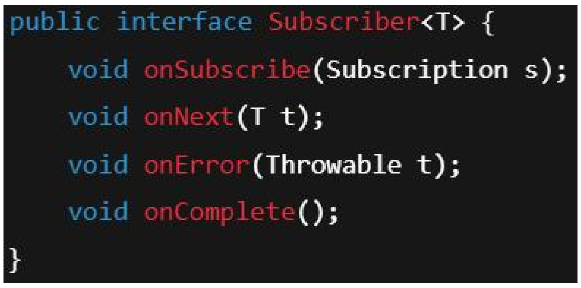
- Publisher



```
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> s);
}
```

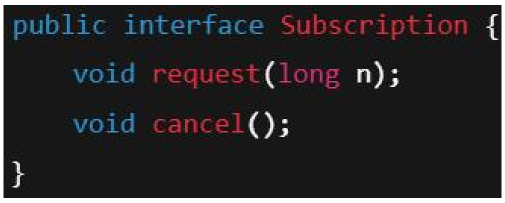
(Flux & Mono)

- Subscriber



```
public interface Subscriber<T> {
    void onSubscribe(Subscription s);
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
}
```

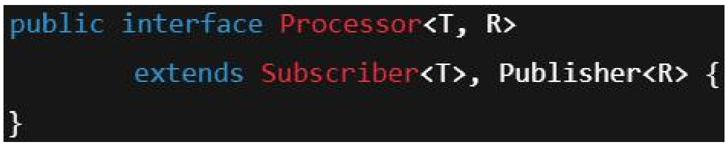
- Subscription



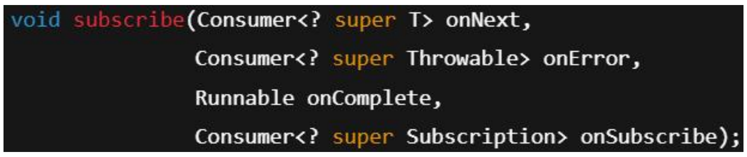
```
public interface Subscription {
    void request(long n);
    void cancel();
}
```

  - Here, **request** method is used for Backpressure handling, Reactor does it by default.

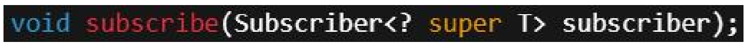
- Processor (Publisher + Subscriber)

  ```java
  public interface Processor<T, R>
          extends Subscriber<T>, Publisher<R> {
  }
  ```

- **subscribe()** method
  - In the **Publisher** interface, only one subscribe method exists which is: **subscribe(Subscriber sub)**, but in **Flux** class, it has different overloaded methods which call that main subscribe(subscriberObj) method at the end building one *Subscriber* type of object out of those *Consumer* and *Runnable* interface objects.
  - There are multiple overloaded methods of this.
  - 

    ```java
    void subscribe(Consumer<? super T> onNext,
                   Consumer<? super Throwable> onError,
                   Runnable onComplete,
                   Consumer<? super Subscription> onSubscribe);
    ```

    (0 to 4: all overloading methods are there)
    - It is used mostly.
    - Lambda functions
    - onSubscribe is not needed to pass,
  - 

    ```java
    void subscribe(Subscriber<? super T> subscriber);
    ```

    - You can directly pass object of Subscriber directly.
- NOTE: If you are passing **Subscriber** type of object inside the **subscribe()** method, then you need to call the **subscription.request()** method inside *onSubscribe* method otherwise it'll not call the ***onNext*** method and wait till infinity.

Key points

- You just need to create a **Publisher** object (Flux, Mono or something else)
- While calling the **subscribe()** method from this object (it takes **Subscriber** as an argument; or builds one **Subscriber** object with those lambda method being passed), you need to pass 4 methods which are *onNext*, *onError*, *onComplete*, *onSubscribe*;

  here, ***onSubscribe*** method provides you one **Subscription** type of object with which you can do something; No need to create *subscription* object by yourself.
- F

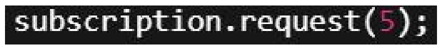
Event Flow

- Subscriber ----- subscribe(Subscriber r) ----------- Publisher
- Publisher ----- onSubscribe(Subscription s)----- Subscriber
- Subscriber ----- request(n) -------------------------- Pubilsher
- Publisher ----- onNext(data) ---------------------- Subscriber
- .
- .
- Publisher ----- onNext(data) ---------------------- Subscriber (***n*** times call)
- Publisher ----- onComplete() ---------------------- Subscriber
-

Operators

- **map**, **flatMap**, **filter**
- map is synchrnous, one-to-one; **flatMap** is asynchronous, one-to-many.
-

Backpressure

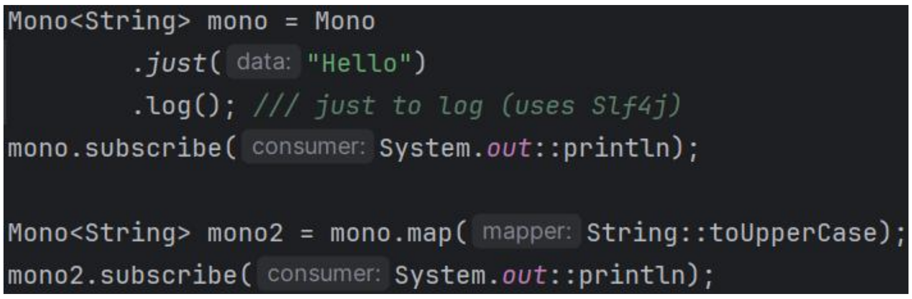
- *Producer is **fast**, Consumer is **slow*** -→ Memory Crash
- 

  ```
  subscription.request(5);
  ```

  (5 items at a time)
-

- F
- F
- F
- F
- F

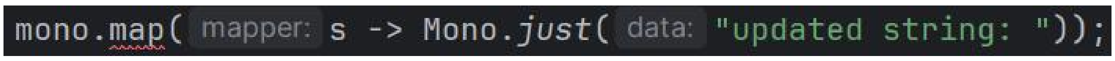
- Dependencies
  - **spring-boot-starter-web**
    - For blocking/servler based.
    - Default embedded server: **Tomcat**
    - Comes with Jackson, Spring MVC.
    - Uses threads for request.
  - **spring-boot-starter-webflux**
    - Spring WebFlux   (Project Reactor: Flux, Mono)
    - Default embedded server: **Netty** (Non-Blocking)
    - It also contains **Jackson**,
    - Event-Loop based; non blocking I/O.
    - Uses less threads.
    - Spring WebFlux can run blocking code as long as it is offloaded from Netty event-loop threads using boundedElastic schedulers. WebFlux does not prohibit blocking APIs; it enforces non-blocking request handling.
- **Mono**
  - There are some methods
    - **just** (static method) : to create a Mono object
    - **subscribe(…..).then(*monoObj*)** : execute another mono after the current one.
    - **zip** (static method) : to combine one or more mono objects.
  - **map** & **flatMap** (*instance methods*, not static)
    - **map** provides the data ***synchronously***.



```
Mono<String> mono = Mono
        .just( data: "Hello")
        .log(); /// just to log (uses Slf4j)
mono.subscribe( consumer: System.out::println);

Mono<String> mono2 = mono.map( mapper: String::toUpperCase);
mono2.subscribe( consumer: System.out::println);
```

      - Convert **Mono<T>** with value **v1** to **Mono<R>** with value **v2**

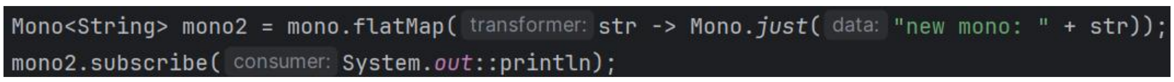
        (its upto you to change the type or value or both)

      - 

```
mono.map( mapper: s -> Mono.just( data: "updated string: "));
```

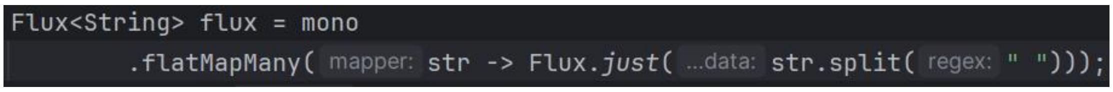
        it'll return **Mono<Mono<String>>**

    - **flatMap** provides the data ***asynchronously***.

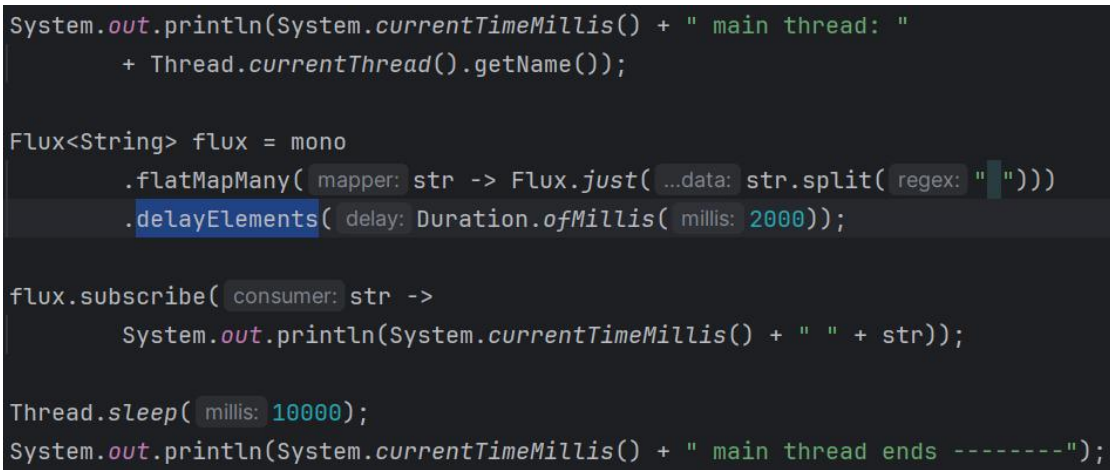


```
Mono<String> mono2 = mono.flatMap( transformer: str -> Mono.just( data: "new mono: " + str));
mono2.subscribe( consumer: System.out::println);
```

    - In this case, we return the **Mono** type of object inside *flatMap* method. (in case of **map**, we had to just return the value that has to be present inside mono object).
- You can use **flatMapMany** to create a **Flux** object from **Mono** object.



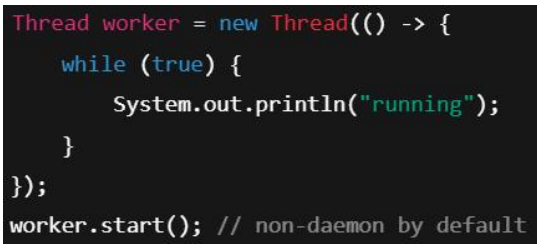
```
Flux<String> flux = mono
        .flatMapMany( mapper: str -> Flux.just( ...data: str.split( regex: " ")));
```



```
System.out.println(System.currentTimeMillis() + " main thread: "
        + Thread.currentThread().getName());

Flux<String> flux = mono
        .flatMapMany( mapper: str -> Flux.just( ...data: str.split( regex: " ")))
        .delayElements( delay: Duration.ofMillis( millis: 2000));

flux.subscribe( consumer: str ->
        System.out.println(System.currentTimeMillis() + " " + str));

Thread.sleep( millis: 10000);
System.out.println(System.currentTimeMillis() + " main thread ends --------");
```

- Here, **delayElements** give some delay between consequitive elements of **Flux** object.
- But, at the bottom, if you don't give **Thread.sleep(**some_random_sleep**)** then the flux will stop executing.
- If the main thread is the only **non-daemon** thread and it ends, then the JVM shuts down and daemon threads will not run.

  JVM will run if at least one **non-daemon** thread is running (*main* thread is a *non-daemon* thread, that's it; there is no speciality of main thread otherwise);



```
Thread worker = new Thread(() -> {
    while (true) {
        System.out.println("running");
    }
});
worker.start(); // non-daemon by default
```

(its also a **non-daemon** thread)

➢ F

➢ F

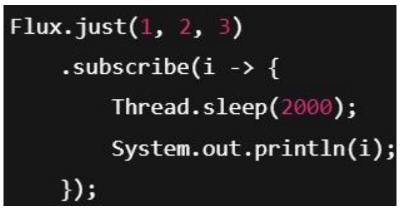
➢ F

➢ F

➢ F

➢ NOTE

- The Flux or Mono are non-blocking in nature, but that doesn’t mean they are ***asynchronous***.
- Consider the below example:



```
Flux.just(1, 2, 3)
    .subscribe(i -> {
        Thread.sleep(2000);
        System.out.println(i);
    });
```

  - Here, it’ll stream the values 1, 2, 3, but here in the ***onNext*** method we have given a sleep of 2 seconds.
  - So, the client will get the response in the interval of 2 seconds.
  - If we replace this sleep with some heavy **IO** bound task, then it’ll execute ***onNext*** method, perform the IO task, then again execute *onNext* method for next value.
  - So, ***Flux* does streaming without making the client wait to get complete output, but whatever it does, it does *synchronously***.

➢ How JVM, Tomcat, Java achives multi-threading and concurrency?

➢ First, OS level:

- Lets say in our OS, only 1 core is there, and we have 2 threads that have to execute.
- *T1* contains some heavy IO bound task, and *T2* is a normal task.
- So, lets say CPU execute *T1* [as we have only 1 core, so only 1 thread can run at once]
- Now, *T1* will execute its task that do heavy IO operation. [Core executes Thread, Thread executes Task]
- So, *T1* is blocked now till the IO operation completes, but the core that was running *T1* has no work to do now.
- Now, Core will be assigned to execute *T2*.
- When the IO is completed by *T1*, it becomes ***runnable***.
- Now, depending upon the algorithm OS is chosing to allocate thread to core (priority based, round-robin …etc) *T1* will be executed.
- So, Cores are **limited**, **hardware** level. Threads are **unlimited**, **software** level.
- Now you might have question, what is the use of *Threads* where the parallelism is defined by *Core*?
    - There are 2 things: **Concurrency**, **Parallelism**.
    - Concurrency: many task **in progress**
    - Parallelism : many tasks ***running*** **at same time**.
- In the above case, if you had only one thread that performs both heavy IO task and normal task, then CPU will allocate that thread to core, then it'll go to perform IO bound task. And now the Core is free, idle, have nothing to execute.
    - That normal task will be executed after that heavy IO task is completed.
    - So, **to make sure CPU doesn't sit *idle* during the heavy IO bound task we need to create threads**.

➢ Now, JVM level

- JVM is nothing but a **virtual machine that runs Java byte-code on top of OS**.
- At the JVM / OS level, **all threads are the same**.

  There is no special difference between "Tomcat threads" and "Spring worker threads" at this level; they are all **JVM (OS) threads**.

  The difference exists only at the *application/framework level* (who created them and for what purpose).

- **Tomcat is also just Java code running inside the JVM.**

  It creates and manages its own request thread pool (default maxThreads ≈ 200, configurable).
- These Tomcat threads are used **only to serve HTTP requests.**

  Tomcat assigns **one thread per HTTP request.**
- Once a request thread is assigned, Tomcat *invokes Spring's DispatcherServlet.*

  The Spring application runs on the **same thread that Tomcat allocated for the request.**

  Let's call this thread *T1.*
- Note: At the JVM / OS level, this thread **T1 is just a normal thread.**

  It is simultaneously:
  - a Tomcat request-handling thread, and
  - the thread executing Spring application code.
- Now, suppose this request needs to perform a heavy or blocking task.

  If this heavy task runs on the same thread T1, **it will block the Tomcat request thread until the operation completes.**
- To avoid blocking the request thread, the Spring application may offload the heavy task **to a separate worker thread** (this happens *only if the application is explicitly designed* to do so).
- The worker thread is created or obtained by the application (via JVM + OS).

  This **worker thread is independent of Tomcat's request thread pool.**
- **Case 1: Waiting for the worker thread**
  - If the Spring application waits for the worker thread to finish (e.g., to include its result in the response), then:
    - the request thread T1 remains blocked anyway,
    - and the response is sent only after the heavy task completes.
- **Case 2: Fire-and-forget**
  - If the Spring application does not wait for the heavy task to finish:
    - the task runs in the worker thread,
    - the response is sent immediately,
    - and the request thread T1 is not blocked by the heavy operation.
- F
-
- F
- F

ـم F

ـم F

ـم F

ـم F

ـم F

- Http connection doesn't depends on the thread (in Tomcat) that is handling the request.
- <u>**Normal request & response (synchronous)**</u>
  - **Client → Socket opened → Thread-1 assigned → Filters → DispatcherServlet → Controller → Response created → Response written → Connection closed → Thread-1 released**
  - The **Socket** is the ***TCP*** connection that carries the HTTP request and response.
  - So, when client send a request, this Socket is in opened.
  - Tomcat assigns one thread to handle that request.
  - In this model, this thread will handle everything like
    - **Getting request from client.**
    - **Delegating that to dispatcher servlet.**
    - **Getting response from dispatcher servlet.**
    - **Sending response back to the client.**
  - After this,
    - HTTP response is completed.
    - **Socket** is closed (or returned to keep-alive, depending upon the HTTP version).
  - In this model
    - The **thread lifetime** and **HTTP connection lifetime** are ***almost* the same**.
  - The HTTP connection remains open as long as the Socket is open. It **doesn't depend upon the Thread's lifetime**.
- **Streaming request & response (asynchronous)**
  - **Client → Socket opened → Thread-1 assigned → Filters → DispatcherServlet → Controller returns Flux → request.startAsync() → Thread-1 released → HTTP connection kept open → (<u>Flux emits data</u> → Container borrows a thread → <u>Data written to socket</u> → Thread released) → (repeats) → Flux completes / client disconnects → Connection closed**
  - In this case, *up to dispatching the request to DispatcherServlet*, the flow is the same as the synchronous request–response model.
  - If the response is of **streaming** type, then ***request.startAsync()*** is called.

```
@GetMapping(value = "test", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
```

  - The **Servlet container (Tomcat)** creates an ***AsyncContext***, keeps the **Socket** inside it, and releases the request thread.

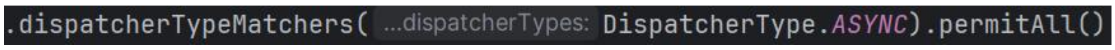
    - When the servlet produces streaming data:
        - The Servlet container (Tomcat) temporarily borrows an available thread
        - Uses that thread to write the data to the Socket
        - Then releases the thread back to the pool
    - This process repeats for each streamed chunk until:
        - The stream completes, or
        - The client disconnects
    - After the streaming is done,
        - the **Socket** is closed and the AsyncContext is removed/discarded.
        - All references to request, response, socket are released.
    - Tomcat does not keep a fixed thread for the connection — it only borrows threads when there is data to write.
- So, in simple terms:
    - **Flux** or **Mono** are not asynchronous by default. They are just *reactive* data types.
    - When Flux is used as a ***streaming HTTP response*** (e.g. **text/event-stream**), Spring switches to **Servlet async mode**, which allows Tomcat to *release the initial request thread* and *borrow threads later* when it needs to write streaming data to the response.
    - This is NOT multithreading in the concurrent sense, because:
        - Multithreading usually means multiple threads executing at the same time.
        - Here, threads are released and reused over time, not running concurrently for the same request.
    - The execution is sequential, even though:
        - Multiple threads may be involved at different times
        - For a given HTTP stream, only one thread writes to the connection at a time.
- My blog link: Medium Blog Link
- When **DispatcherType.ASYNC** is set?
    - When request has been put to **Servlet Async Mode**.

      ```
      request.startAsync();
      ```
    - The original thread has returned (after sending first chunk).
    - The container later re-dispatch the request to continue processing.
    -

- Re-Dispatching of request
  - First request comes from the client; Tomcat receives it and delegates it to Spring.
  - Spring checks whether the response is a streaming type (for example *text/event-stream*).

    If yes, Spring calls *request.startAsync( )*
  - The original Tomcat request thread is released back to the thread pool.
  - Spring subscribes to the **Flux**
  - Whenever a new element (chunk) is emitted by the Flux
    - Spring writes the chunk to the **ServletOutputStream**
    - Tomcat picks any free *http-nio-** thread
    - The chunk is written to the client
    - The thread is released again
  - This process continues for every emitted chunk.
  - After the last chunk:
    - If the Flux *completes* OR
    - An *error* occurs OR
    - Async *timeout* happens OR
    - Client *closes* the connection
  - The async request must now be formally completed, because it was previously put into async mode using *request.startAsync( )*
  - Tomcat performs an ASYNC **re-dispatch**
    - **DispatcherType** is set to **ASYNC**
    - Filters configured for ASYNC are invoked
    - DispatcherServlet resumes async processing
    - Controller method is NOT called again
  - Spring calls **AsyncContext.complete()**
  - Tomcat:
    - Finalizes the response
    - Triggers async listeners (onComplete, onError, onTimeout)
    - Cleans up request/response resources
    - Ends the request lifecycle

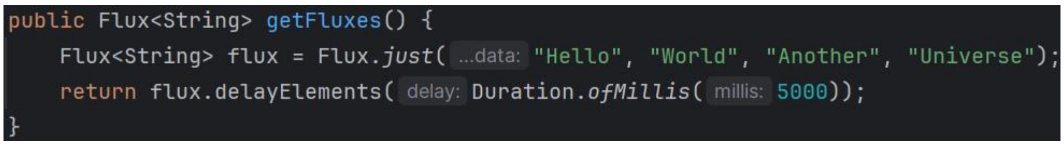
- Dispatcher.ASYNC allowance in security filter chain.
  - By default, Spring Security authorization rules are applied mainly to REQUEST dispatcher type (here the context is the dispatcher, not the user).
  - If permitAll() is configured for a route, that route is allowed for all dispatcher types, including ASYNC.
  - If a route is protected:
    - The user is authenticated during the initial REQUEST dispatch
    - Authentication is not performed again during async processing because the SecurityContext already contains the Authentication object for the duration of the request
  - In case of streaming responses, Spring enters async mode and at the end performs an ASYNC re-dispatch
  - This ASYNC re-dispatch targets the same protected route, but by default only REQUEST dispatcher type is permitted, so the ASYNC dispatch may be blocked by authorization rules.
  - Therefore, DispatcherType.ASYNC must be explicitly permitted to allow the async lifecycle to complete successfully



```
.dispatcherTypeMatchers( ...dispatcherTypes: DispatcherType.ASYNC).permitAll()
```

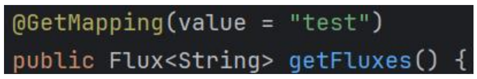
  - In simple term: **DispatcherType.ASYNC** must be explicitly allowed *only for protected routes* using async/streaming, because async re-dispatch is a lifecycle continuation that is otherwise *blocked by default request-only authorization rules.*
  - F

- I am just returning an flux object with some delay



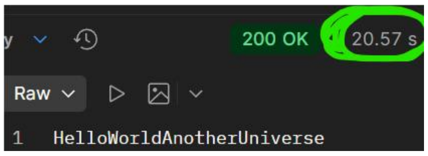
```
public Flux<String> getFluxes() {
    Flux<String> flux = Flux.just( ...data: "Hello", "World", "Another", "Universe");
    return flux.delayElements( delay: Duration.ofMillis( millis: 5000));
}
```

  - Without ***delay***, these will come immediately.
  - Now, without writing ***produces*** as ***stream*** inside the mapping.



```
@GetMapping(value = "test")
public Flux<String> getFluxes() {
```

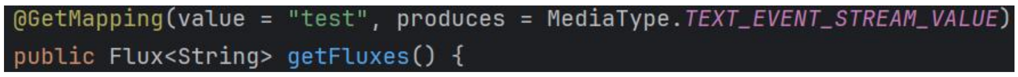
    - It'll wait for **20 seconds** (delay: 5 seconds, elements: 4; so 5 * 4 = 20 seconds delay).
    - In this case, spring will subscribe this response (**flux**) and after getting all the elements, it'll send the response in ***application/json*** format (which is default).



```
y   200 OK   20.57 s
Raw
1   HelloWorldAnotherUniverse
```

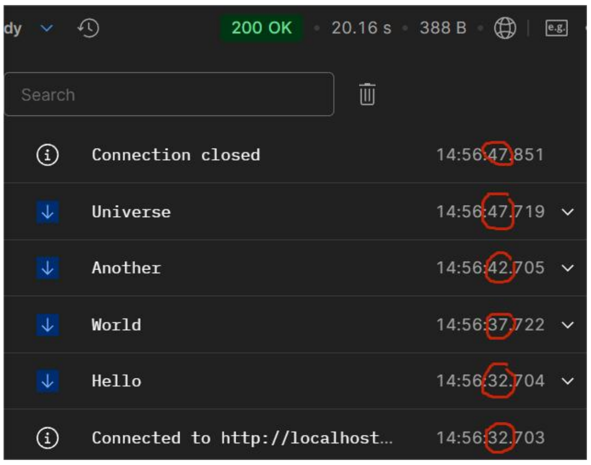
    20 seconds, and all outputs came at once.

    - in this case, **Socket** and **Thread** lifetimes are almost same.
  - Now, when we give ***produces*** as ***stream*** inside the mapping.



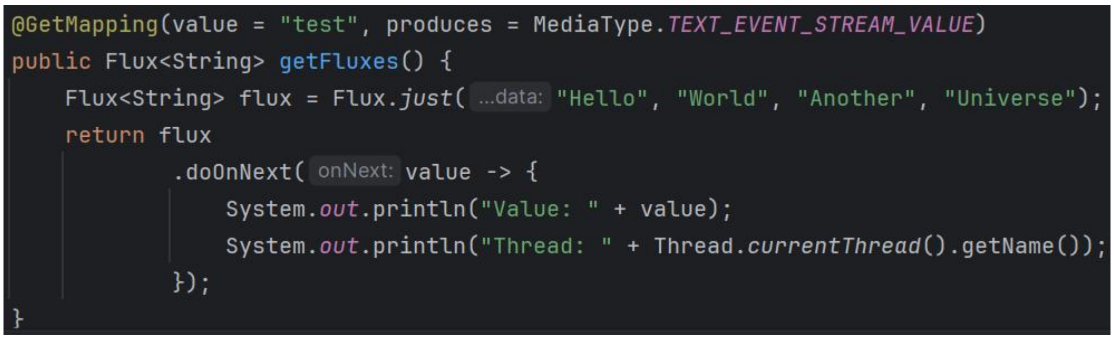
```
@GetMapping(value = "test", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<String> getFluxes() {
```

    - Now, it'll stream.
    - After each **5 seconds**, one output will come.
    - Now the output will be sent in ***text/event-stream*** format.
    - Spring will subscribe this, and whenever it gets the content, it'll immediately send to the client without waiting for all the elements to be received.

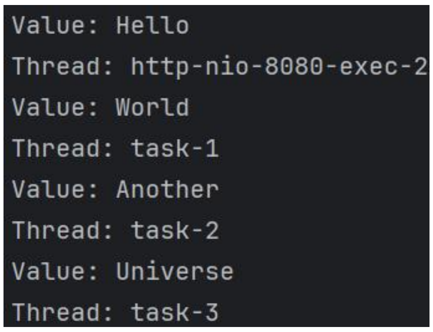


```
dy   200 OK • 20.16 s • 388 B
Search
Connection closed                  14:56:47.851
Universe                           14:56:47.719
Another                            14:56:42.705
World                              14:56:37.722
Hello                              14:56:32.704
Connected to http://localhost...   14:56:32.703
```

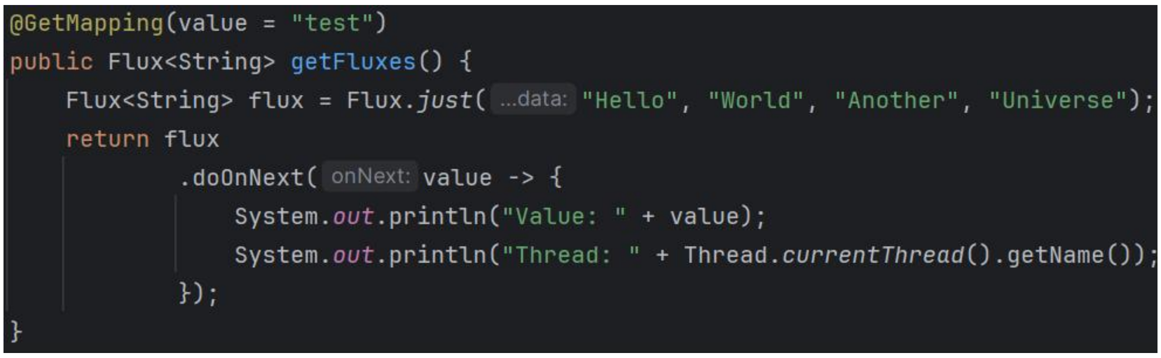
- Here also the time between starting and ending of http connection took **20** seconds but the responses were being sent in each ***5 seconds inverval***.
- In this case, the main thread was released, and different borrwed threads sent the responses, so, **Socket** and **Thread** lifetime are not same here.

- I wrote the below code to print the thread's name for each values.
  - **JSON type request/response model** (***Non-Streaming***)
    - 

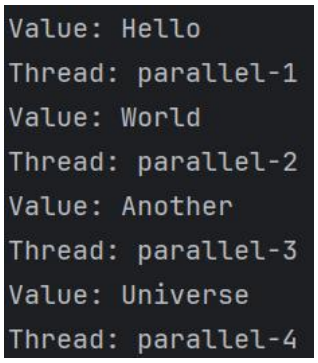
```
@GetMapping(value = "test", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<String> getFluxes() {
    Flux<String> flux = Flux.just( ...data: "Hello", "World", "Another", "Universe");
    return flux
            .doOnNext( onNext: value -> {
                System.out.println("Value: " + value);
                System.out.println("Thread: " + Thread.currentThread().getName());
            });
}
```

    - 

```
Value: Hello
Thread: http-nio-8080-exec-2
Value: World
Thread: task-1
Value: Another
Thread: task-2
Value: Universe
Thread: task-3
```

  - **Streaming type request/response model**
    - 

```
@GetMapping(value = "test")
public Flux<String> getFluxes() {
    Flux<String> flux = Flux.just( ...data: "Hello", "World", "Another", "Universe");
    return flux
            .doOnNext( onNext: value -> {
                System.out.println("Value: " + value);
                System.out.println("Thread: " + Thread.currentThread().getName());
            });
}
```

    - 

```
Value: Hello
Thread: parallel-1
Value: World
Thread: parallel-2
Value: Another
Thread: parallel-3
Value: Universe
Thread: parallel-4
```

- F

- Without produce:
  - delay: all parallel threads,
  - Non delay: http-nio-8080-exec-2 and tasks
- With produce:
  - Delay: all parallel threads
  - Non delay: http-nio-8080-exec-2 and tasks
- F
-
  -