**Asynchronous vs Non-blocking**

<http://www.programmr.com/blogs/difference-between-asynchronous-and-non-blocking>

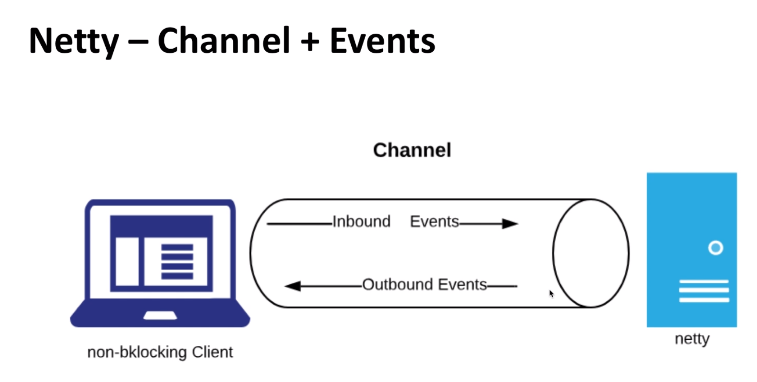
So, Non-blocking does not mean multi-threaded or concurrent by default.

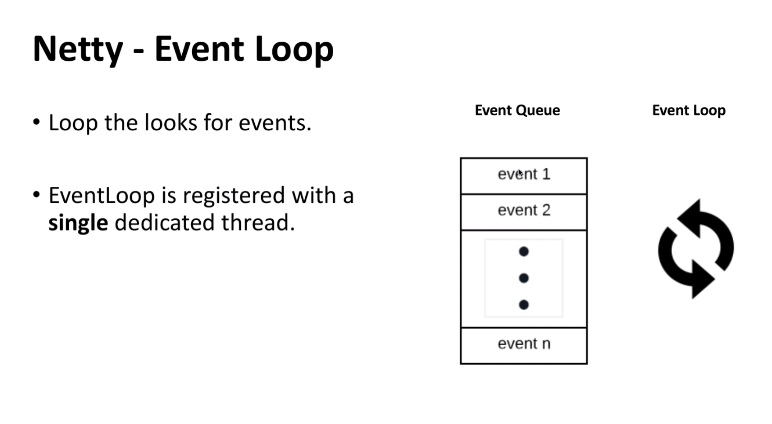
**Flux and Mono are single threaded**

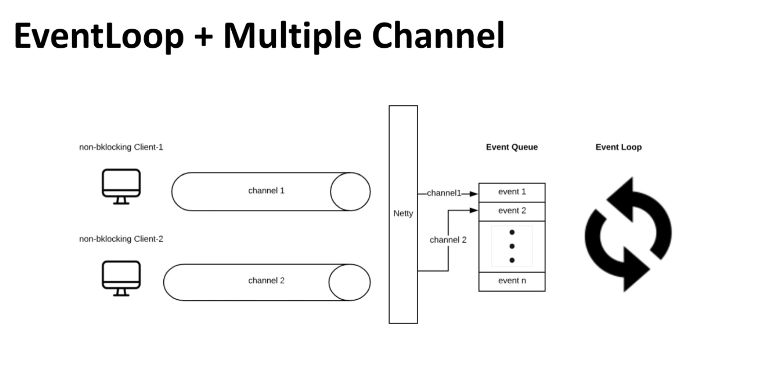
1. Flux and Mono are not concurrent by default, they are **concurrency-agnostic.** That is, they does not enforce a concurrency model. Rather, it leaves you, the developer, in command.
2. They run on single thread by default
3. We have to use schedulers to make them parallel execution

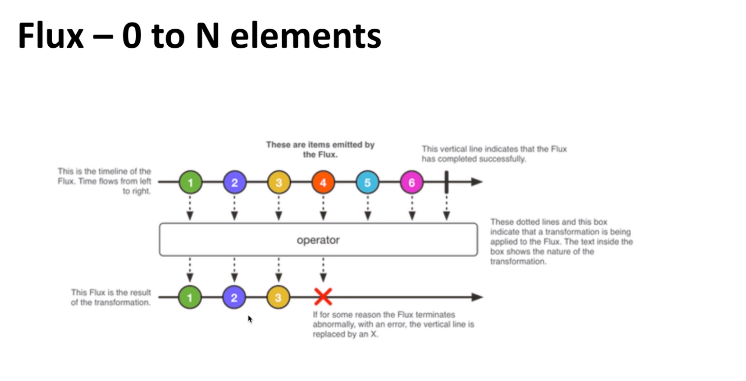
<https://projectreactor.io/docs/core/release/reference/#schedulers>

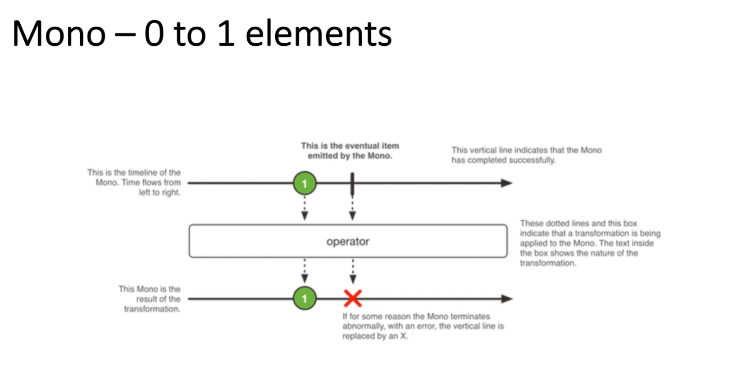
**Spring WebFlux execution model**

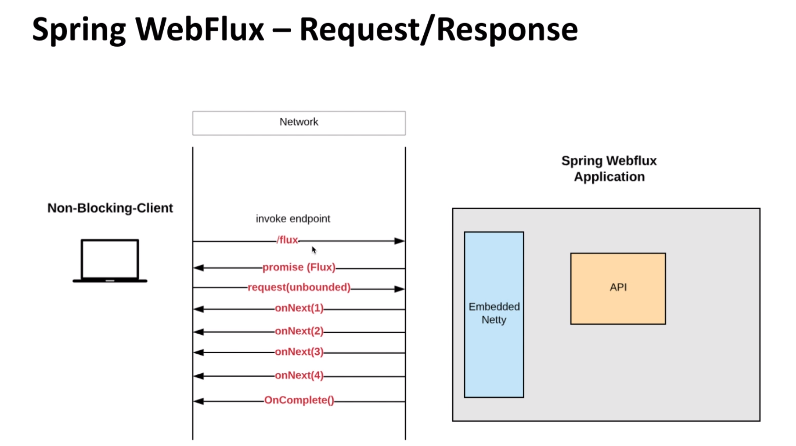






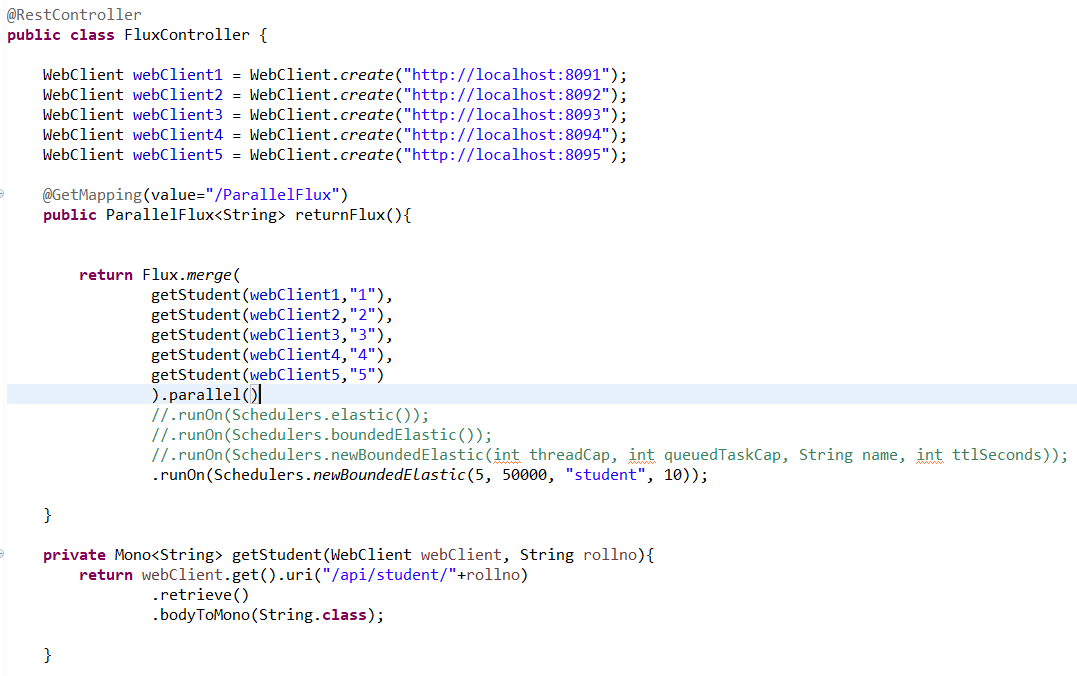






**Parallel Flux Example**

Refer: <https://projectreactor.io/docs/core/release/reference/#advanced-parallelizing-parralelflux>

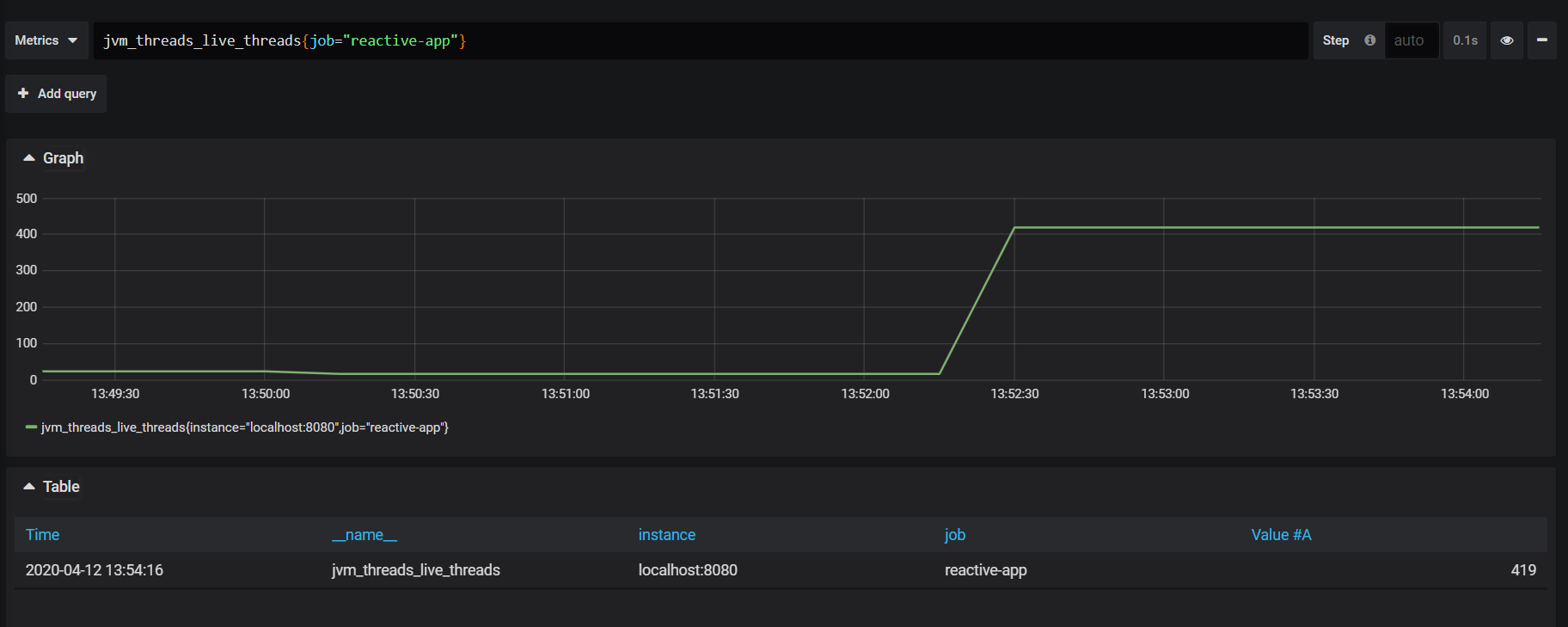


**Schedulers**

Refer: <https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html>

* [parallel()](https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#parallel--): Optimized for fast [Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html?is-external=true) non-blocking executions
* [single()](https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#single--): Optimized for low-latency [Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html?is-external=true) one-off executions
* [elastic()](https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#elastic--): Optimized for longer executions, an alternative for blocking tasks where the number of active tasks (and threads) can grow indefinitely
* [boundedElastic()](https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#boundedElastic--): Optimized for longer executions, an alternative for blocking tasks where the number of active tasks (and threads) is capped
* [immediate()](https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#immediate--): to immediately run submitted [Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html?is-external=true) instead of scheduling them (somewhat of a no-op or "null object" [Scheduler](https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Scheduler.html))
* [fromExecutorService(ExecutorService)](https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#fromExecutorService-java.util.concurrent.ExecutorService-) to create new instances around [Executors](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html?is-external=true)

**Elastic:** [scheduler](https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Scheduler.html) that dynamically creates ExecutorService-based Workers and caches the thread pools, reusing them once the Workers have been shut down.



**boundedElastic:** Default maximum size for the global [boundedElastic()](https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#boundedElastic--) [Scheduler](https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Scheduler.html), initialized by system property reactor.schedulers.defaultBoundedElasticSize and falls back to 10 x number of processors available to the runtime on init.



**newBoundedElastic(int threadCap, int queuedTaskCap, String name, int ttlSeconds):**

newBoundedElastic(5, 50000, "student", 10)

****