Reactor Core 3.0
A lite Rx API for the JVM

Sébastien Deleuze - Stéphane Maldini

@sdeleuze - @smaldlini

Sébastien Deleuze

- Live in Lyon
- Remote worker @ Pivotal
- Spring Framework and Reactor committer
- Works on Spring Framework 5 upcoming Reactive support
- Co-worker @ La Cordée
- Mix-IT staff member
- @sdeleuze on Twitter

Stéphane Maldini

- Survive in London
- Social Engineering @
- Project Reactor lead
- Reactive Streams &
  Reactive Streams Commons contributor
- Works on Spring Framework 5
  upcoming Reactive support
- @smaldini on Twitter

+

Why going reactive?

- More for scalability and stability than for speed
- Use cases:
  - Webapp calling remote web services
  - Lot of slow clients
  - Big Data
  - Serve more clients on the same hardware
  - Event based development (web, mobile)

Reactive, what is it?

- Reactive is used to broadly define event-driven systems
- Reactive Manifesto defines qualities of reactive systems
- Reactive programming: moving imperative logic to async, non-blocking, functional-style code, in particular when interacting with external resources

More details on http://fr.slideshare.net/StphaneMaldini/intro-to-reactive-programming-52821416

For reactive programming, we need tools :

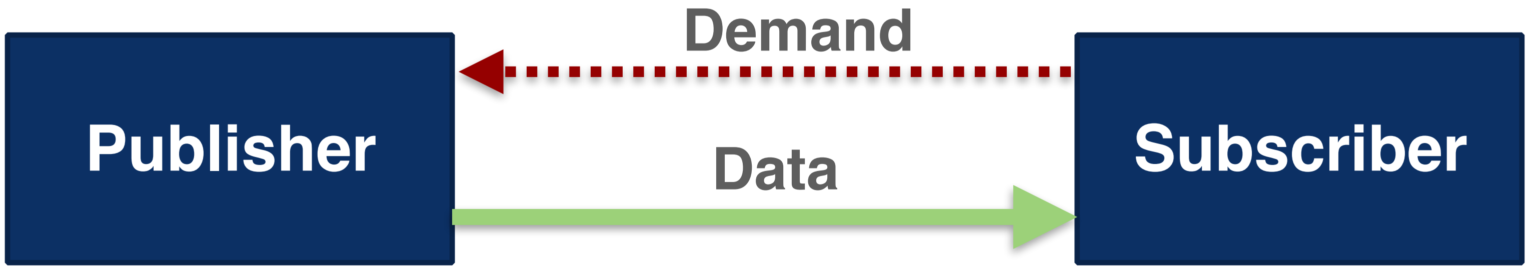
- ☐ [Reactive Streams]
- ☐ Reactive APIs

Reactive Streams

- Reactive Streams is a contract for asynchronous stream processing with non-blocking back pressure
- De facto standard for interop between reactive libraries
- To be included in Java 9 as java.util.concurrent.Flow

Reactive Streams principle

- **Max(InflightData) <= demand**
- No data sent without demand
- Demand can be unbounded
- The recipient controls how much data it will receive



Reactive Streams is 4 interfaces (+ a TCK)

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

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

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

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

For reactive programming, we need tools :

☑ Reactive Streams

☐ Reactive APIs

Reactive APIs

- Buffer, merge, concatenate, or apply a wide range of transformations to your data
- On the JVM:
    - Reactor 3.0 is 4th generation* and based on Reactive Streams
    - RxJava 1.x: 2nd generation* and most used implementation
    - Akka Stream 2.x: Lightbend 3rd generation* Reactive API
- Also for other languages, for example RxJS, MostJS

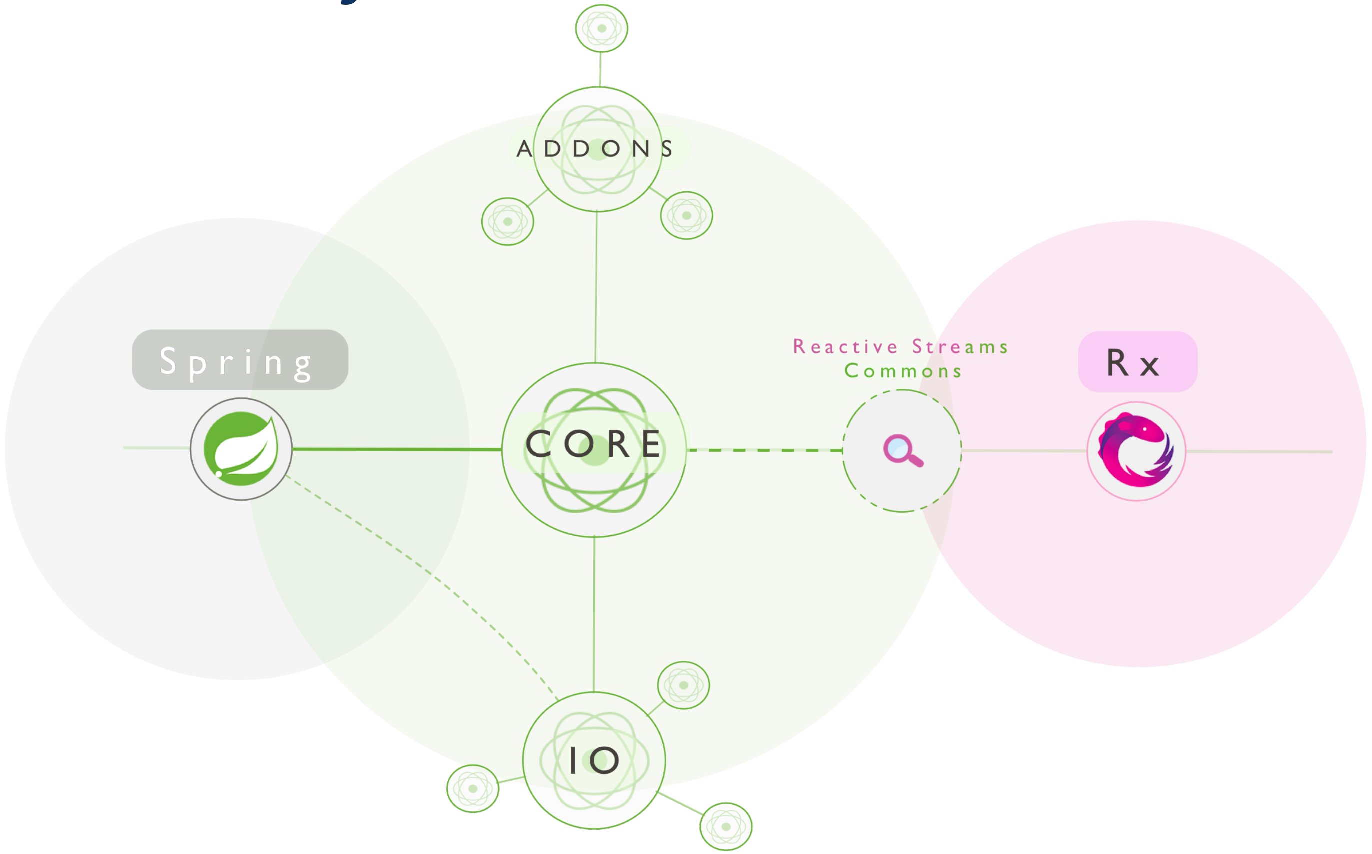
* Based on http://akarnokd.blogspot.fr/2016/03/operator-fusion-part-1.html

Reactor Core 3.0

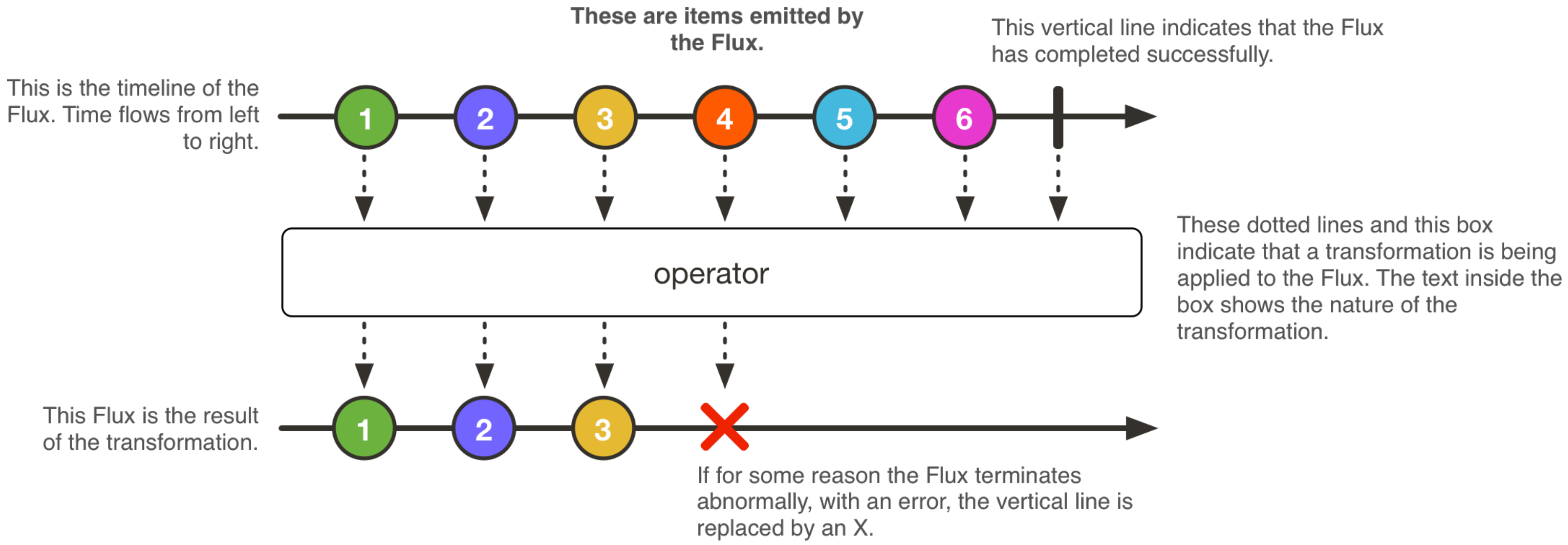
- Built with major contributions from Dávid Karnok (RxJava lead) and from some Spring Framework committers
- Natively based on Reactive Streams, RSC* and Java 8+
- Strong focus on efficiency
- Powerful Mono API
- Ever-improving debugging, logging, testing capabilities

* *ReactiveStreamsCommons is a research effort about reactive flows*

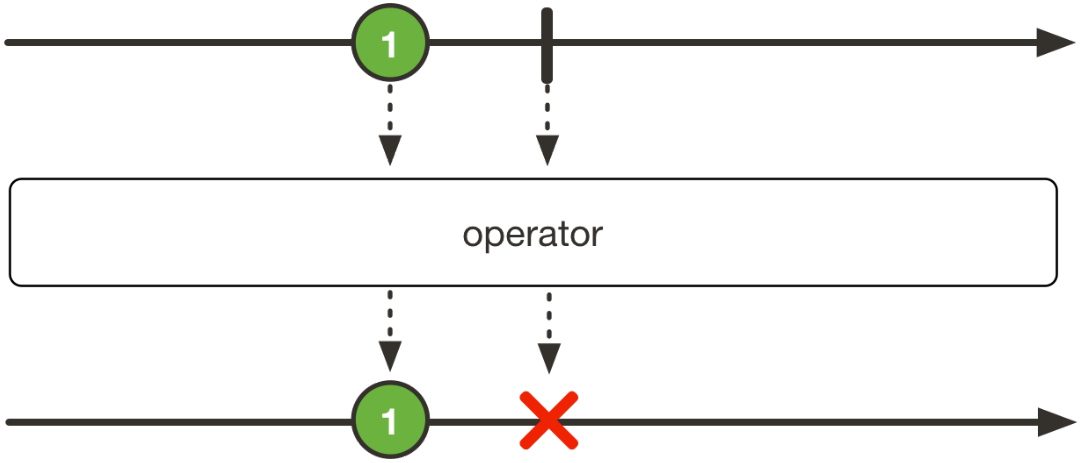
Reactor 3.0 ecosystem



Flux (0..N elements) with ReactiveX compliant API


These are items emitted by the Flux.
This vertical line indicates that the Flux has completed successfully.
This is the timeline of the Flux. Time flows from left to right.
1
2
3
4
5
6
operator
These dotted lines and this box indicate that a transformation is being applied to the Flux. The text inside the box shows the nature of the transformation.
This Flux is the result of the transformation.
1
2
3
If for some reason the Flux terminates abnormally, with an error, the vertical line is replaced by an X.

Mono (0..1 element)



This is the eventual item emitted by the Mono.

This vertical line indicates that the Mono has completed successfully.

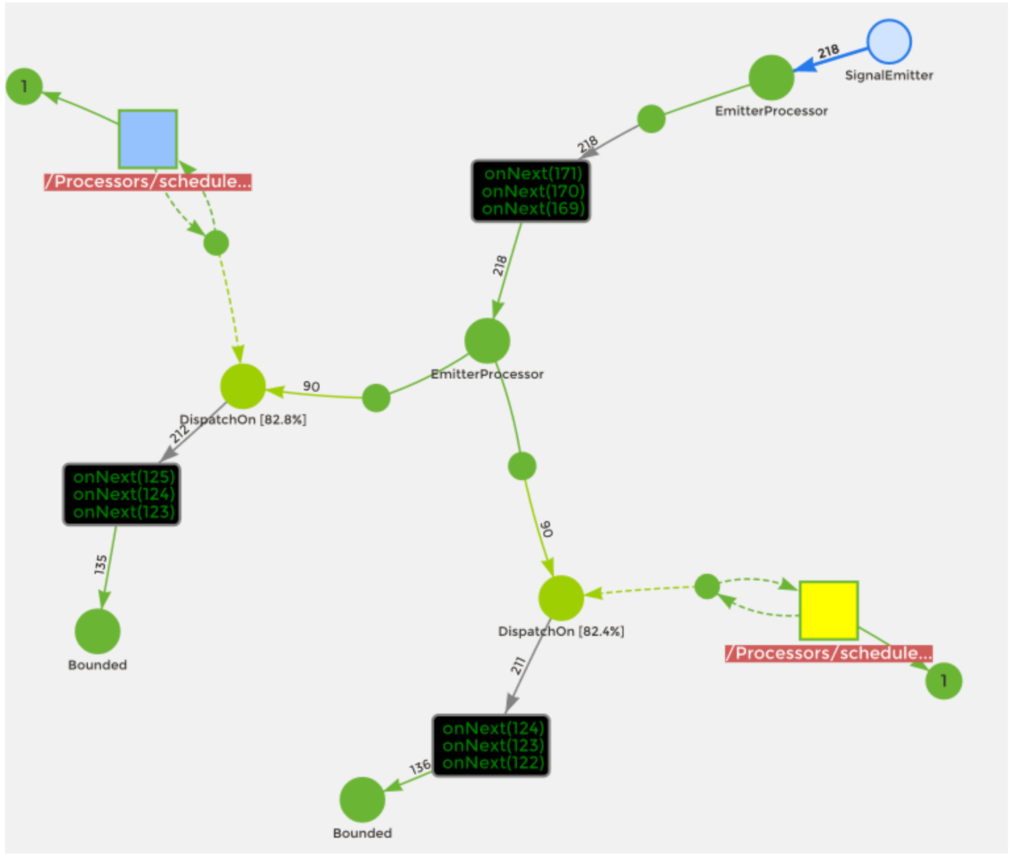
This is the timeline of the Mono. Time flows from left to right.

These dotted lines and this box indicate that a transformation is being applied to the Mono. The text inside the box shows the nature of the transformation.

This Mono is the result of the transformation.

If for some reason the Mono terminates abnormally, with an error, the vertical line is replaced by an X.

Reactor Web Console



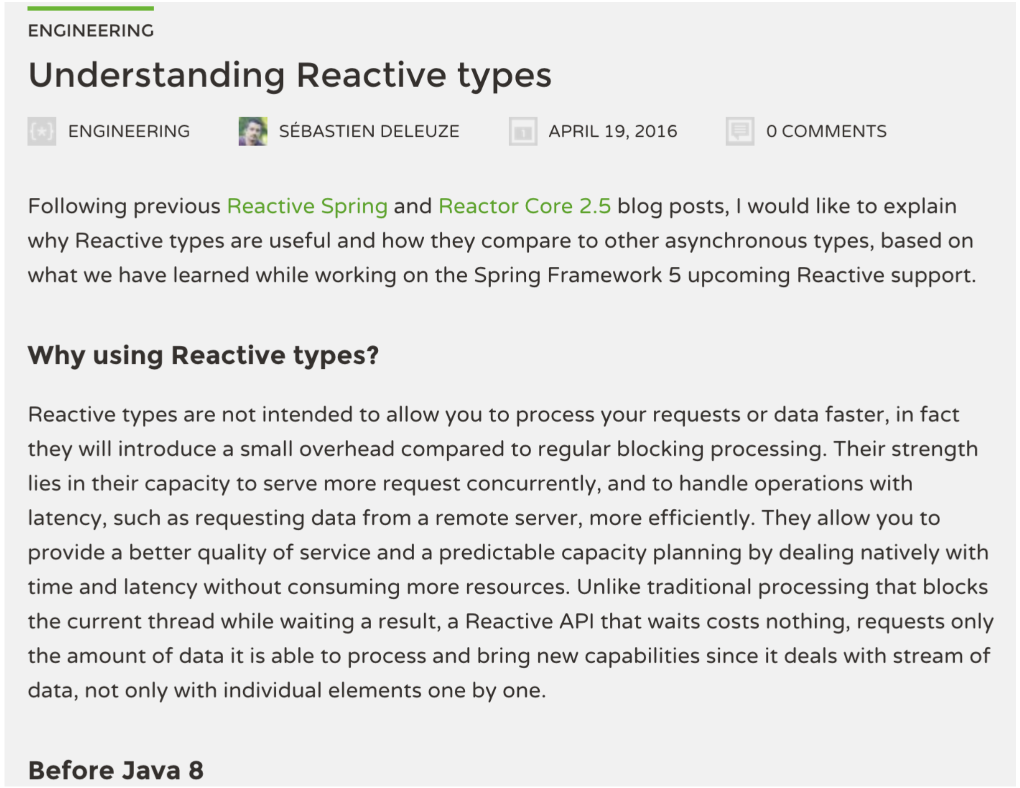
Type comparaison

	No value	Single value	Multiple values

Blocking	`void`	`T`
		`Future<T>`
			`Iterable<T>`
			`Collection<T>`
			`Stream<T>`

Non-blocking	`CompletableFuture<Void>`	`CompletableFuture<T>`	`CompletableFuture<List<T>>`
Reactive Streams	`Publisher<Void>`	`Publisher<T>`	`Publisher<T>`
RxJava	`Completable`	`Single<T>`	`Observable<T>`
Reactor	`Mono<Void>`	`Mono<T>`	`Flux<T>`

https://spring.io/blog/2016/04/19/understanding-reactive-types



ENGINEERING

Understanding Reactive types

ENGINEERING SÉBASTIEN DELEUZE APRIL 19, 2016 0 COMMENTS

Following previous Reactive Spring and Reactor Core 2.5 blog posts, I would like to explain why Reactive types are useful and how they compare to other asynchronous types, based on what we have learned while working on the Spring Framework 5 upcoming Reactive support.

Why using Reactive types?

Reactive types are not intended to allow you to process your requests or data faster, in fact they will introduce a small overhead compared to regular blocking processing. Their strength lies in their capacity to serve more request concurrently, and to handle operations with latency, such as requesting data from a remote server, more efficiently. They allow you to provide a better quality of service and a predictable capacity planning by dealing natively with time and latency without consuming more resources. Unlike traditional processing that blocks the current thread while waiting a result, a Reactive API that waits costs nothing, requests only the amount of data it is able to process and bring new capabilities since it deals with stream of data, not only with individual elements one by one.

Before Java 8

Reactive Spring

- Spring projects are going reactive
- Reactor Core is the reactive foundation
- RxJava adapters provided
- You will be able to choose your web engine: Tomcat, Jetty, Undertow or Netty
- Most impact on Web and Data support (IO intensive)
    - Spring Reactive experiment
    - Spring Reactive Playground sample application

Well known Controller example

```
@RestController
public class UserController {
    private BlockingRepository<User> repository;

    @RequestMapping(path = "/save-capitalized", method = RequestMethod.POST)
    public void saveCapitalized(@RequestBody List<User> users) {
        users.forEach(u -> u.setName(u.getName().toUpperCase()));
        repository.save(users);
    }
}
```

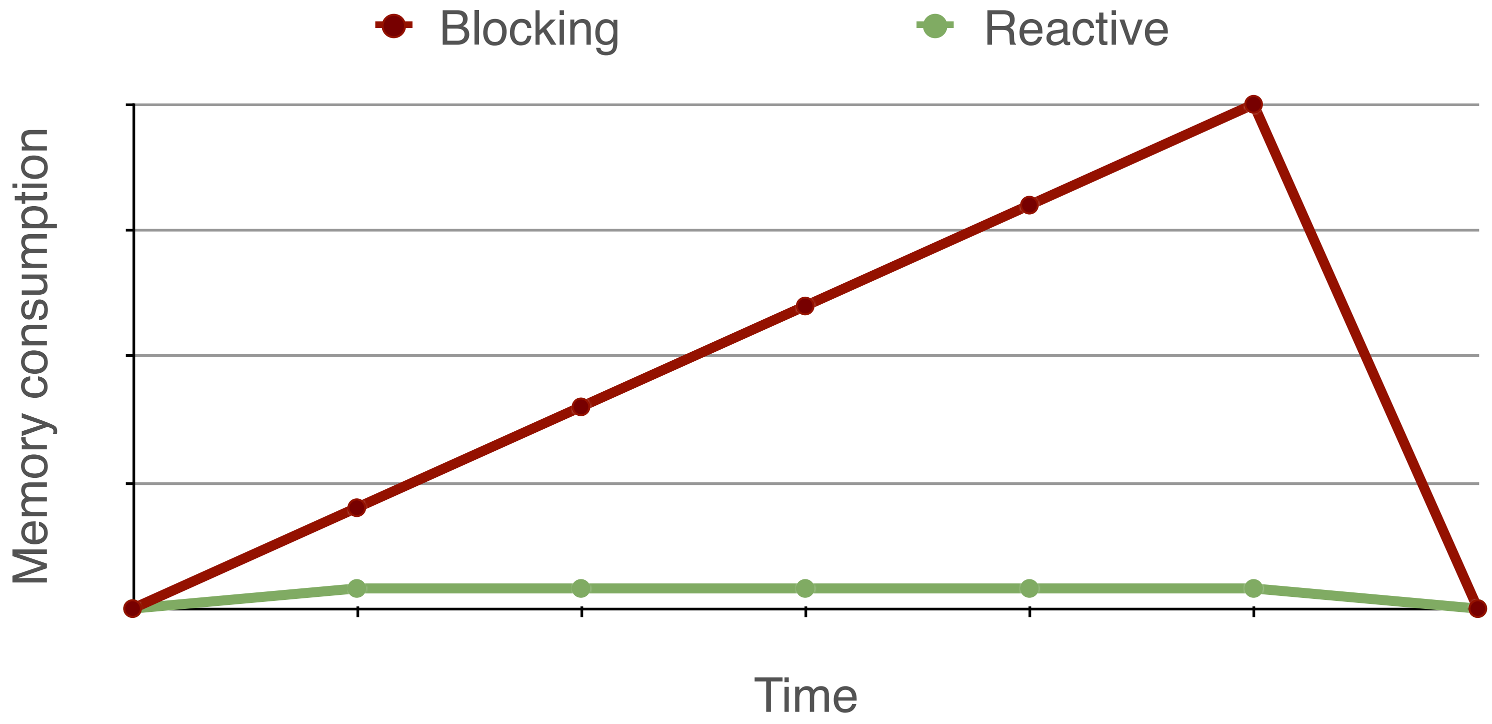
```
public interface BlockingRepository<T> {
    void save(List<T> elements);
    Iterable<T> findAll();
}
```

Controller with Reactive types

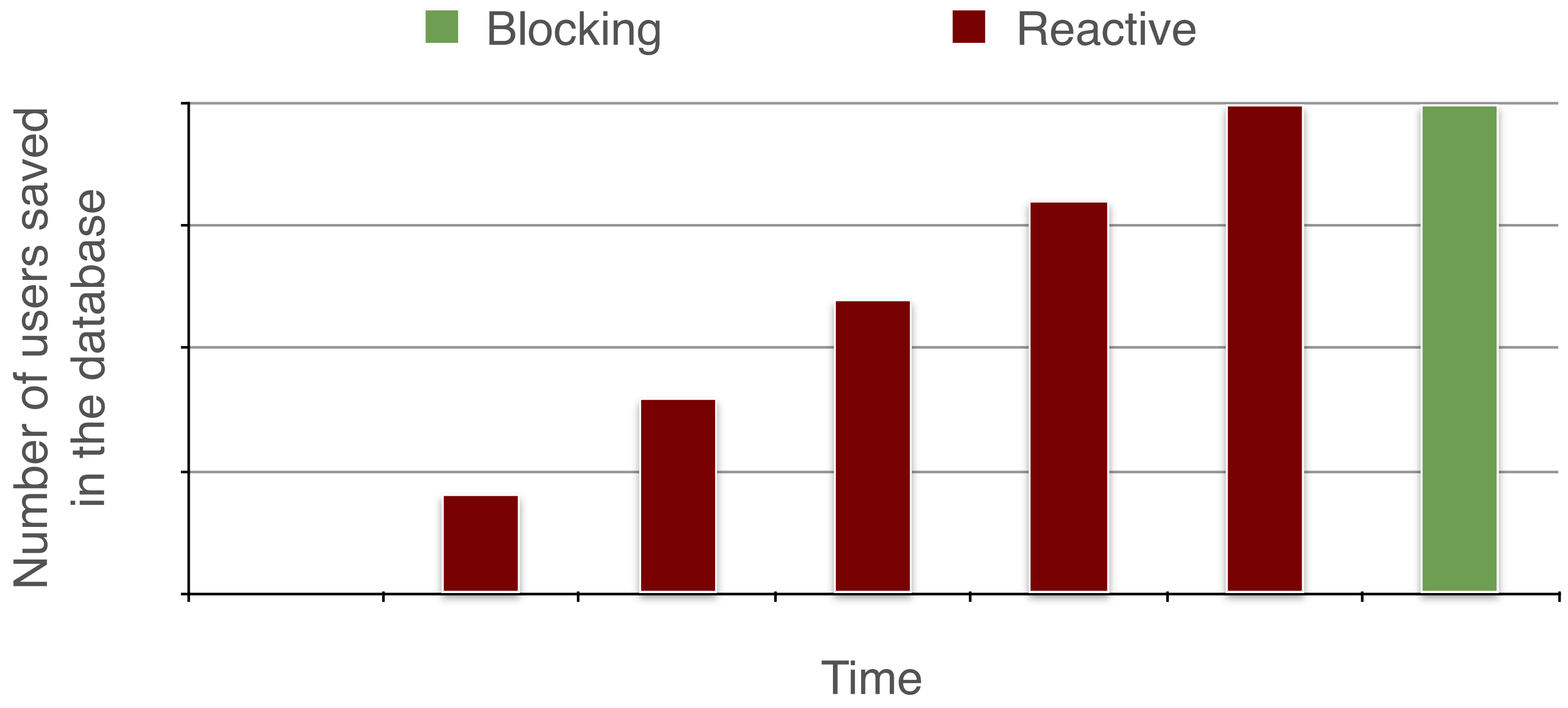
```
@RestController
public class UserController {

    private ReactiveRepository<User> repository;

    @RequestMapping(path = "/save-capitalized", method = RequestMethod.POST)
    public Mono<Void> saveCapitalized(@RequestBody Flux<User> users) {
        return repository.save(users.map(u -> new User(u.getName().toUpperCase()));
    }
}
```

```
public interface ReactiveRepository<T> {
    Mono<Void> save(Publisher<T> elements);
    Flux<T> findAll();
}
```

Blocking vs Reactive: memory consumption



Blocking vs Reactive: streaming updates



Controller with Reactive return values

```
@RestController
public class UserController {

    private ReactiveRepository<User> repository;

    @RequestMapping(path = "/", method = RequestMethod.GET)
    public Flux<User> findAll() {
        return repository.findAll();
    }
}
```

- Optimized serialization when using Flux instead of List
- Also perfectly suitable for Server-Sent Events

Reactive HTTP client with Mono

```
import static org.springframework.web.client.reactive.HttpRequestBuilders.*;
import static org.springframework.web.client.reactive.WebResponseExtractors.*;

Mono<Person> result = webClient
   .perform(get("http://localhost:8080/person")
   .header("X-Test-Header", "testvalue")
   .accept(MediaType.APPLICATION_JSON))
   .extract(body(Person.class));
```

Reactive HTTP client with Flux

```
import static org.springframework.web.client.reactive.HttpRequestBuilders.*;
import static org.springframework.web.client.reactive.WebResponseExtractors.*;

Flux<Person> response = webClient
  .perform(get("http://localhost:8080/persons")
  .accept(MediaType.APPLICATION_JSON))
  .extract(bodyStream(Person.class));
```

Works for:

- JSON array `[{"foo":"bar"},{"foo":"baz"}]`
- JSON Streaming `{"foo":"bar"}{"foo":"baz"}`
- SSE with something like `.extract(sseStream(Person.class))`

Perfect fit for Microservices

Can handle bidirectional stream processing

Now let’s go to the code!

https://github.com/reactor/lite-rx-api-hands-on/