Reactive Programming for Java Developers

Rossen Stoyanchev

About Me

- ❖ Spring Framework committer
- ❖ Spring MVC, WebSocket messaging
- ❖ Spring 5 Reactive

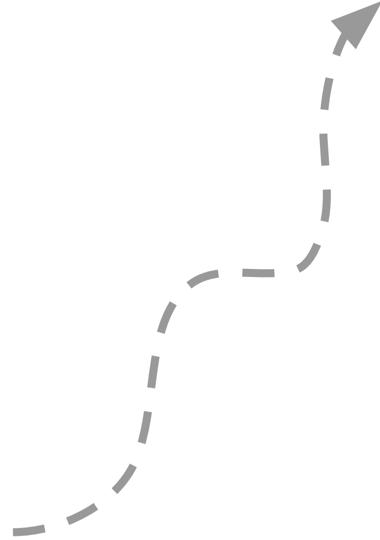
Long-Running Shift to Concurrency

Today

Independent services,

Cloud environment,

Distributed apps



10 years ago

Self-sufficient apps,

App server,

Keep it simple, don't distribute

Changing expectations

Internet scale & resilience,

Efficient use of resources,

Latency is common

Impact on programming model

Imperative logic not so simple when latency is the norm

Forced to deal with asynchronicity

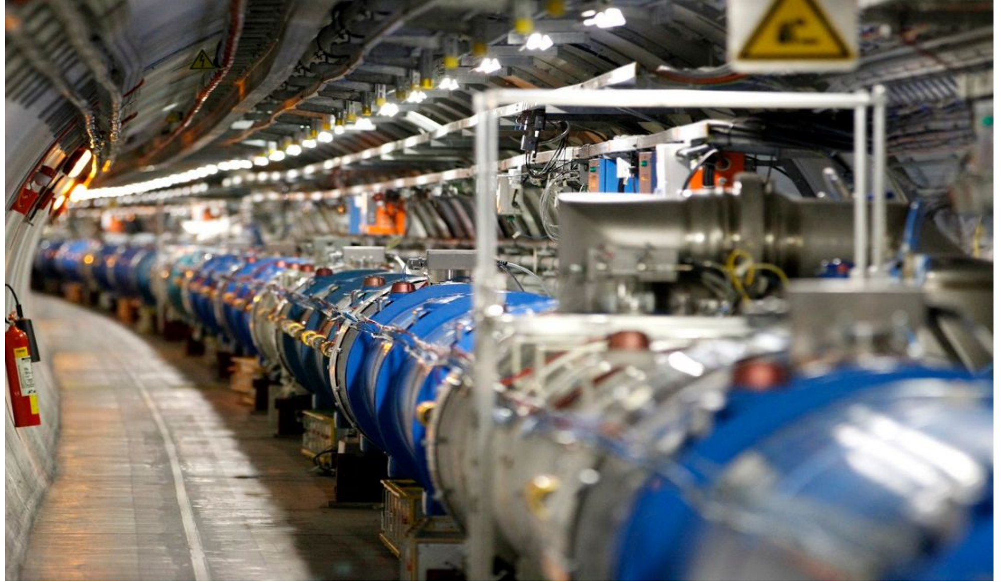
Limits of scale

There is another way

Fundamentally async & non-blocking

Using very few threads

Major shift but also major benefits



Reactive Programming?

In this talk

How would we design an async API in Java ?

Can we do better ?

Introducing reactive libraries

Spring reactive experience

Design async API in Java

Return one thing

```
public interface UserRepository {

    User findById(String id) throws IOException;

    ...

    ...

}
```

Usage

```
try {
    User user = userRepository.findById(id);
    // ...

}
catch (IOException e) {
    // ...
}
```

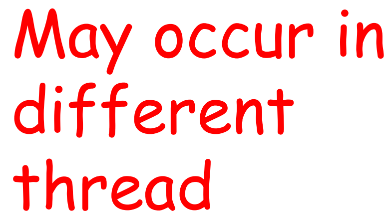
Return it async style

```
public interface UserRepository {

    Future<User> findById(String id) throws IOException;

    ...

}
```



Usage

```
try {
    Future<User> future = userRepository.findById(id);
    User user = future.get(); // block

}
catch (InterruptedException e) {
    // ...
}
catch (ExecutionException e) {
    // ...
}
```

Ugh

CompletableFuture (JDK 1.8)

- Future with actions
- Actions trigger when Future completes
- Callback mechanism

Return it async style with Java 1.8

```
public interface UserRepository {
    CompletableFuture<User> findById(String id);
    ...
    ...
}
```

Usage

```
CompletableFuture<User> future = repository.findById(id);

future.whenComplete((user, throwable) -> {
    // ...
});
```

Usage

```
CompletableFuture<User> future = repository.findById(id);

future.whenComplete((user, throwable) -> {
    // ...
});
```

Requires null check

Return many

```java
public interface UserRepository {

    ...

    CompletableFuture<List<User>> findAll();

    ...
}
```

Return many

```
public interface UserRepository {
    ...
    CompletableFuture<List<User>> findAll();
    ...
}
```

No callback till all users collected

Return many

```
public interface UserRepository {

    ...

    CompletableFuture<List<User>> findAll();

    ...
}
```

It may be too many

Return nothing

```
public interface UserRepository {

    ...

    ...

    CompletableFuture<Void> save(User user);

}
```

Return nothing

```
public interface UserRepository {

    ...

    ...

    CompletableFuture<Void> save(User user);

}
```

Async notification: success or failure?

Can we do better?

Async results as a stream

- One notification per data item
- One notification for either completion or error

Return Type	Description	Notifications
void	Success	onComplete()
void	Failure	onError(Throwable)
User	Match	onNext(User), onComplete()
User	No match	onComplete()
User	Failure	onError(Throwable)
List<User>	Two matches	onNext(User), onNext(User), onComplete()
List<User>	No match	onComplete()
List<User>	Failure	onError(Throwable)

Stream abstraction

- ➢ Functional, declarative programming model
- ➢ Combine, transform, reduce sequences
- ➢ Focus on what, not how

Java 8 Stream

- Great example of the benefits of a stream API
- However built for collections mainly
- Pull-based, usable once

Beyond collections

- ➢ Latency-sensitive data streams
- ➢ Infinite sequences
- ➢ Push-based notifications

Reactive Libraries

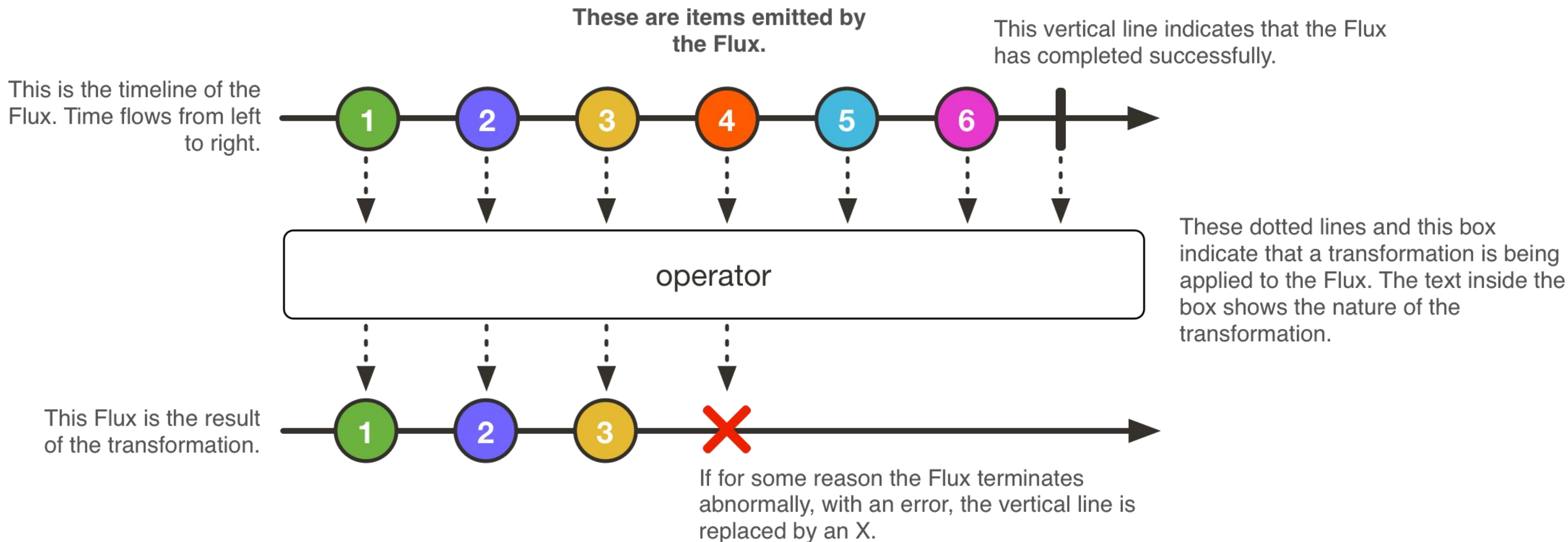
Reactive library?

- ➢ Stream-like API similar to Java 8
- ➢ Suited for **any** data sequence
- ➢ Latency-sensitive, infinite, collections

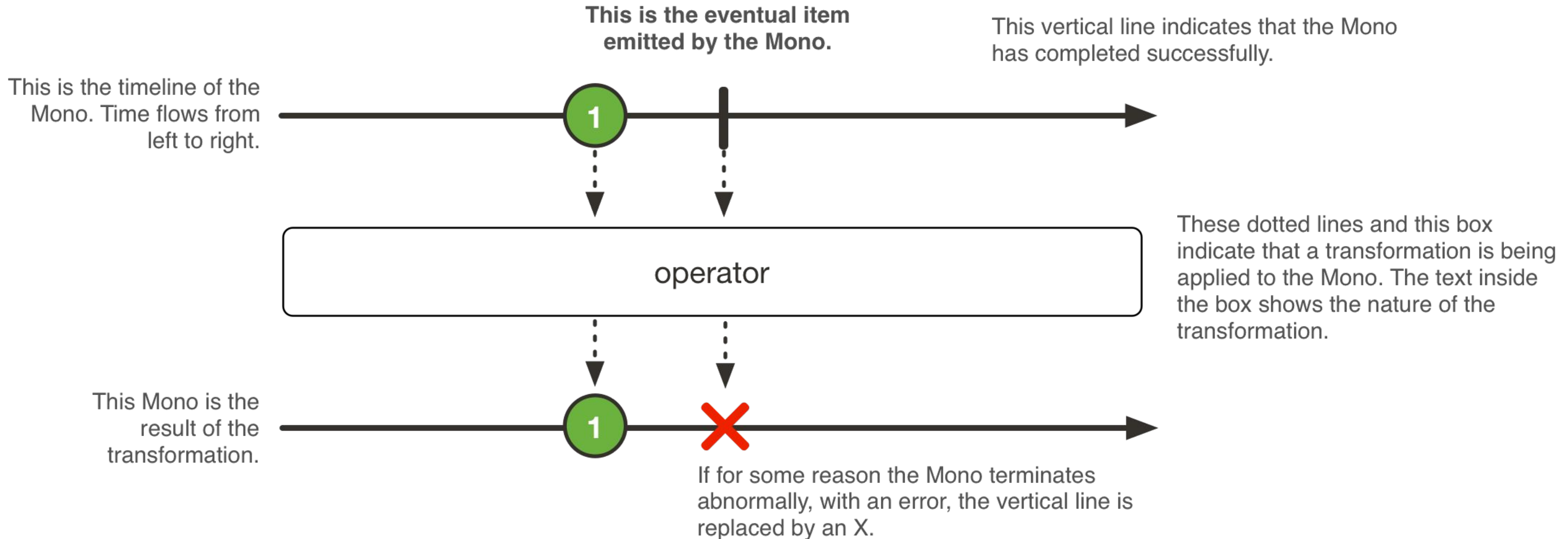
Project Reactor

- Reactive Streams foundation for the JVM
- API similar to ReactiveX
- Easy to bridge to Java 8 Stream

Flux – sequence of 0..N



Mono – sequence of 0..1



Flux to Java Stream

```
Stream<?> stream = Flux.fromStream(anotherStream)
        .timeout(Duration.ofSeconds(30))
        .log("hello")
        .stream();
```

Mono to CompletableFuture

```java
CompletableFuture<String> future =
        Mono.fromCompletableFuture(someCompletableFuture)
                .timeout(Duration.ofSeconds(30))
                .log("hello")
                .toCompletableFuture();
```

More than a stream API

- Reactor is back-pressure ready
- Reactive Streams spec
- Producers must not overwhelm consumers

Reactive Streams Spec

- Industry collaboration
- Small API, rules, TCK
- Reactive interoperability across libraries

Reactive Streams included in Java 9

“No single best fluent async/parallel API. CompletionStage best supports continuation-style programming on futures, and java.util.stream best supports (multi-stage, possibly-parallel) "pull" style operations on the elements of collections. Until now, one missing category was "push" style operations on items as they become available from an active source.“

Doug Lea, from [initial announcement]

Reactive Streams in Java 9

- Interfaces in `java.util.concurrent.Flow`
- `SubmissionPublisher`
  standalone bridge to Reactive Streams
- Tie-ins to `CompletableFuture` and `Stream`

Reactive Streams API

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

Reactive Streams API

```
public interface Subscriber<T> {
  void onSubscribe(Subscription sub);
  void onNext(T item);
  void onError(Throwable ex);
  void onComplete();
}
```

Reactive Streams API

```
public interface Subscriber<T> {
  void onSubscribe(Subscription sub);
  void onNext(T item);
  void onError(Throwable ex);
  void onComplete();
}
```

Reactive repository

```
public interface UserRepository {

    Mono<User> findById(Long id);

    Flux<User> findAll();

    Mono<Void> save(User user);

}
```

Using the reactive repository

```
repository.findAll()
        .filter(user -> user.getName().matches("J.*"))
        .map(user -> "User: " + user.getName())
        .log()
```

Using the reactive repository

```
repository.findAll()
        .filter(user -> user.getName().matches("J.*"))
        .map(user -> "User: " + user.getName())
        .log()
        .subscribe(user -> {});
```

Subscriber triggers flow of data

Using the reactive repository

```
repository.findAll()
        .filter(user -> user.getName().matches("J.*"))
        .map(user -> "User: " + user.getName())
        .log()
        .subscribe(user -> {});
```

Consume all data by default

Output

```
onSubscribe

request(unbounded)

onNext(User: Jason)

onNext(User: Jay)

...

onComplete()
```

Usage

```
repository.findAll()
        .filter(user -> user.getName().matches("J.*"))
        .map(user -> "User: " + user.getName())
        .useCapacity(2)
        .log()
        .subscribe(user -> {});
```

Output

```
onSubscribe
request(2)
onNext(User: Jason)
onNext(User: Jay)
request(2)
onNext(User: Joe)
onNext(User: John)
...
```

More on Reactor

- Currently 2.5 M4 (might change to 3.0 label)
- GA release scheduled for July
- Hands-on exercise, blog post series

Reactive Spring

Reactive
Spring MVC ?

Annotated controllers

Controller Methods

```java
@RequestMapping("/users")
public Flux<User> getUsers() {
    return this.userRepository.findAll();
}
```

```java
@RequestMapping("/users")
public Observable<User> getUsers() {
    return this.userRepository.findAll();
}
```

Annotated controllers

Spring MVC

Spring Web Reactive

```java
public interface HandlerMapping {

    Object getHandler( … );

}
```

```
public interface HandlerMapping {

  Mono<Object>
    Object getHandler( ... );

}
```

@MVC

Spring MVC

Spring Web Reactive

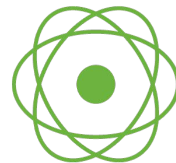
Servlet API

???

@MVC

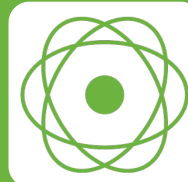
Spring MVC	Spring Web Reactive
Servlet API	???
Servlet Container	???

@MVC



RxJava

Spring Web Reactive



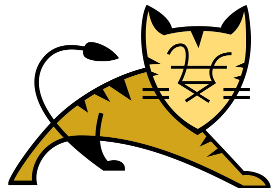
HTTP

Reactive Streams

Servlet 3.1

Reactor I/O

RxNetty



jetty://

Netty

Spring Framework 5.0 M1

spring-reactive

More Reactive Efforts

Reactive Journey

@rstoya05