Reactive Programming for Java Developers

Rossen Stoyanchev
About Me

- Spring Framework committer
- Spring MVC, WebSocket messaging
- Spring 5 Reactive
Long-Running Shift to Concurrency
10 years ago

Self-sufficient apps,

App server,

Keep it simple, don’t distribute

Today

Independent services,

Cloud environment,

Distributed apps
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
public interface UserRepository {

    User findById(String id) throws IOException;

    ...

    ...

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

catch (IOException e) {
    // ...
}
public interface UserRepository {

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

  ...

}
try {
    Future<User> future = userRepository.findById(id);
    User user = future.get(); // block
}
catch (InterruptedException e) {
    // ...
}
catch (ExecutionException e) {
    // ...
}
CompletableFuture (JDK 1.8)

- Future with actions
- Actions trigger when Future completes
- Callback mechanism
public interface UserRepository {
    CompletableFuture<User> findById(String id);
    ...
    ...
}
CompletableFuture<User> future = repository.findById(id);
future.whenComplete((user, throwable) -> {
    // ...
});

Async callback!
CompletableFuture<User> future = repository.findById(id);
future.whenComplete((user, throwable) -> {
    // ...
});

Requires null check
public interface UserRepository {

    ...  

    CompletableFuture<List<User>> findAll();

    ...  

}
public interface UserRepository {

    ... 

    CompletableFuture<List<User>> findAll();

    ... 

}
Return many

```java
public interface UserRepository {

    ...

    CompletableFuture<List<User>> findAll();

    ...

}
```

It may be too many
Return nothing

```java
public interface UserRepository {

    ...

    ...

    CompletableFuture<Void> save(User user);

}
```
Return nothing

```java
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
<table>
<thead>
<tr>
<th>Return Type</th>
<th>Description</th>
<th>Notifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>Success</td>
<td>onComplete()</td>
</tr>
<tr>
<td>void</td>
<td>Failure</td>
<td>onError(Throwable)</td>
</tr>
<tr>
<td>User</td>
<td>Match</td>
<td>onNext(User), onComplete()</td>
</tr>
<tr>
<td>User</td>
<td>No match</td>
<td>onComplete()</td>
</tr>
<tr>
<td>User</td>
<td>Failure</td>
<td>onError(Throwable)</td>
</tr>
<tr>
<td>List&lt;User&gt;</td>
<td>Two matches</td>
<td>onNext(User), onNext(User), onComplete()</td>
</tr>
<tr>
<td>List&lt;User&gt;</td>
<td>No match</td>
<td>onComplete()</td>
</tr>
<tr>
<td>List&lt;User&gt;</td>
<td>Failure</td>
<td>onError(Throwable)</td>
</tr>
</tbody>
</table>
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

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

These are items emitted by the Flux.

This vertical line indicates that the Flux has completed successfully.

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.

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

This Flux is the result of the transformation.
Mono – sequence of 0..1

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

This Mono is the result of the transformation.

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

This vertical line indicates that the Mono has completed successfully.

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.
Flux to Java Stream

```java
Stream<? extends T> 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
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> subscriber);
}
```java
public interface Subscriber<T> {
    void onSubscribe(Subscription sub);
    void onNext(T item);
    void onError(Throwable ex);
    void onComplete();
}
```
public interface Subscriber<T> {
    void onSubscribe(Subscription sub);
    void onNext(T item);
    void onError(Throwable ex);
    void onComplete();
}
Reactive repository

```java
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

```java
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

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

Consume all data by default
onSubscribe

request (unbounded)

onNext (User: Jason)

onNext (User: Jay)

...

onComplete()
Usage

Consume two at a time

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

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

Spring MVC  Spring Web Reactive
public interface HandlerMapping {

    Object getHandler(...);

}
public interface HandlerMapping {

    Mono<Object> getHandler ...);

}
@MVC

Spring MVC

Servlet API

Spring Web Reactive

???
Spring Framework 5.0 M1

spring-reactive
More Reactive Efforts

mongoDB

Thymeleaf

Couchbase
Reactive Journey
Questions?

@rstoya05