

[LIVE WEBINAR] Wednesday, June 10th: Getting Started with DevOps on Snowflake

[Sign Up Now](#)

DZone > Java Zone > An Introduction to Reactive Programming With Spring

An Introduction to Reactive Programming With Spring

by Anna Eriksson · Jun. 08, 20 · Java Zone · Presentation

Introduction

This is part one of a series of articles that will serve as an introduction on how to build reactive web applications using Spring Boot, Project Reactor and WebFlux. It will be a beginners guide to the reactive world, but the reader is assumed to have previous knowledge on Java and Spring Boot.

Why Reactive Programming?

Reactive programming has been around for some time but gained much higher interest during the last couple of years. The reason for this relates to the fact that traditional imperative programming has some limitations when it comes to coping with the demands of today, where applications need to have high availability and provide low response times also during high load.

Thread per Request Model

To understand what reactive programming is and what benefits it brings, let's first consider the traditional way of developing a web application with Spring — using Spring MVC and deploying it on a servlet container, such as Tomcat.

The servlet container has a dedicated thread pool to handle the HTTP requests, where each incoming request will have a thread assigned, and that thread will handle the entire lifecycle of the request ("thread per request model"). This means that the application will only be able to handle a number of concurrent requests that equal the size of the thread pool. It is possible to configure the size of the thread pool, but since each thread reserves some memory (typically 1MB), the higher thread pool size we configure, the higher the memory consumption.

If the application is designed according to a microservice-based architecture, we have better possibilities to scale based on load, but a high memory utilization still comes with a cost. So, the thread per request model could become quite costly for applications with a high number of concurrent requests.

An important characteristic of microservice-based architectures is that the application is distributed, running as a high number of separate processes, usually across multiple servers. Using traditional imperative programming with synchronous request/response calls for inter-service communication means that threads frequently get blocked waiting for a response from another service. This results in a huge waste of resources.

Waiting for I/O Operations

Same type of waste also occurs while waiting for other types of I/O operations to complete such as a database call or reading from a file. In all these situations the thread making the I/O request will be blocked and waiting idle until the I/O operation has completed, this is called blocking I/O. Such situations where the executing thread gets blocked, just waiting for a response, means a waste of threads and therefore a waste of memory.





Figure 1 - Thread blocked waiting for response

Response Times

Another issue with traditional imperative programming is the resulting response times when a service needs to do more than one I/O request. For example, service A might need to call service B and C, as well as do a database lookup and then return some aggregated data as a result. This would mean that service A's response time would, besides its own processing time, be a sum of:

- Response time of service B (network latency + processing).
- Response time of service C (network latency + processing).
- Response time of database request (network latency + processing).

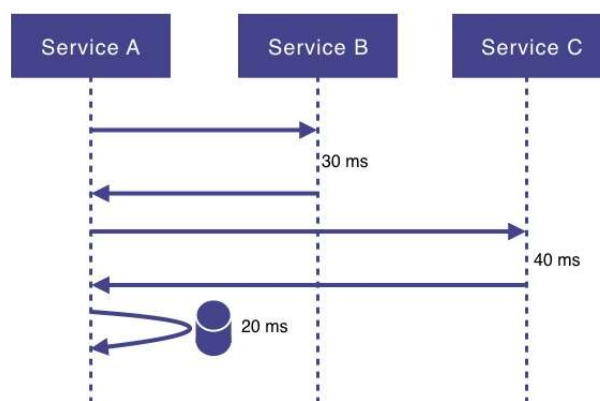


Figure 2 - Calls executed in sequence

If there is no actual logical reason to do these calls in sequence, it would certainly have a very positive effect on service A's response time if these calls would be executed in parallel. Even though there is support for doing asynchronous calls in Java using `CompletableFuture` and registering callbacks, using such an approach extensively in an application would make the code more complex and harder to read and maintain.

Overwhelming the Client

Another type of problem that might occur in a microservice landscape is when service A is requesting some information from service B, let's say for example all the orders placed during last month. If the amount of orders turns out to be huge, it might become a problem for service A to retrieve all this information at once. Service A might be overwhelmed with the high amount of data and it might result in for example an out of memory-error.

To Summarize

The different issues described above are the issues that reactive programming is intended to solve.

In short, the advantages that comes with reactive programming is that we:

- Move away from the thread per request model and can handle more requests with a low number of threads
- Prevent threads from blocking while waiting for I/O operations to complete
- Make it easy to do parallel calls
- Support "back pressure", giving the client a possibility to inform the server on how much load it can handle

What Is Reactive Programming?

Definition

A short definition of reactive programming used in the Spring documentation is the following:

"In plain terms reactive programming is about non-blocking applications that are asynchronous and event-driven and require a small number of threads to scale. A key aspect of that definition is the concept of backpressure which is a mechanism to ensure producers don't overwhelm consumers."

Explanation

So how is all of this achieved?

In short: by programming with asynchronous data streams. Let's say service A wants to retrieve some data from service B. With the reactive programming style approach, service A will make a request to service B, which returns immediately (being non-blocking and asynchronous).

Then, the data requested will be made available to service A as a data stream, where service B will publish an `onNext`-event for each data item one by one. When all the data has been published, this is signaled with an `onComplete` event. In case of an error, an `onError` event would be published and no more items would be emitted.

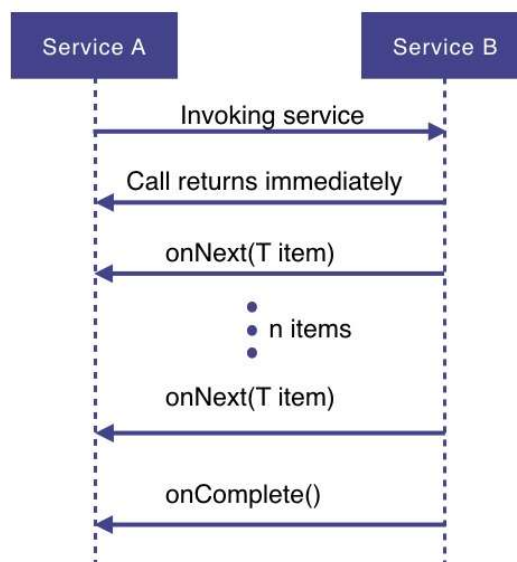


Figure 3 - Reactive event stream

Reactive programming uses a functional style approach (similar to the Streams API), which gives the possibility to perform different kinds of transformations on the streams. A stream can be used as an input to another one. Streams can be merged, mapped and filtered.

Reactive Systems

Reactive programming is an important implementation technique when developing "reactive systems", which is a concept described in the "Reactive Manifesto", highlighting the need for modern applications to be designed to be:

- Responsive (responding in a timely manner)
- Resilient (staying responsive also in failure situations)
- Elastic (staying responsive under varying workload)
- Message Driven (relying on asynchronous message passing)

Building a reactive system means to deal with questions such as separation of concerns, data consistency, failure management, choice of messaging implementation etc.

Reactive programming can be used as an implementation technique to ensure that the individual services use an asynchronous, non-blocking model, but to design the system as a whole to be a reactive system requires a design that takes

care or all these other aspects as well.

Background

ReactiveX

In 2011, Microsoft released the Reactive Extensions (ReactiveX or Rx) library for .NET, to provide an easy way to create asynchronous, event-driven programs. In a few years time, Reactive Extensions was ported to several languages and platforms including Java, JavaScript, C++, Python and Swift. ReactiveX quickly became a cross-language standard. The development of the Java implementation - RxJava - was driven by Netflix and version 1.0 was released in 2014.

ReactiveX uses a mix of the Iterator pattern and the Observer pattern from the Gang of Four. The difference is that a push model is used compared to Iterators normal pull-based behavior. Along with observing changes, also completion and errors are signaled to the subscriber.

Reactive Streams Specification

As time went on, a standardization for Java was developed through the Reactive Streams effort. Reactive Streams is a small specification intended to be implemented by the reactive libraries built for the JVM. It specifies the types to implement to achieve interoperability between different implementations. The specification defines the interaction between asynchronous components with back pressure. Reactive Streams was adopted in Java 9, by the Flow API. The purpose of the Flow API is to act as an interoperation specification and not an end-user API like RxJava.

The specification covers the following interfaces:

Publisher

This represents the data producer/data source and has one method which lets the subscriber register to the publisher.

Java

```
1 public interface Publisher<T> {
2
3     public void subscribe(Subscriber<? super T> s);
4
5 }
```

Subscriber:

This represents the consumer and has the following methods:

Java

```
1 public interface Subscriber<T> {
2
3     public void onSubscribe(Subscription s);
4
5     public void onNext(T t);
6
7     public void onError(Throwable t);
8
9     public void onComplete();
10
11 }
```

- `onSubscribe` is to be called by the `Publisher` before the processing starts and is used to pass a `Subscription` object from the `Publisher` to the `Subscriber`
- `onNext` is used to signal that a new item has been emitted
- `onError` is used to signal that the `Publisher` has encountered a failure and no more items will be emitted
- `onComplete` is used to signal that all items were emitted successfully

Subscription:

The subscriptions holds methods that enables the client to control the Publisher's emission of items (i.e. providing backpressure support).

Java

```
1 public interface Subscription {
2
3     public void request(long n);
4
5     public void cancel();
6
7 }
```

- `request` allows the `Subscriber` to inform the `Publisher` on how many additional elements to be published
- `cancel` allows a subscriber to cancel further emission of items by the `Publisher`

Processor:

If an entity shall transform incoming items and then pass it further to another Subscriber, an implementation of the Processor interface is needed. This acts both as a Subscriber and as a Publisher.

Java

```
1 public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
2
3 }
```

Project Reactor

Spring Framework supports reactive programming since version 5. That support is build on top of Project Reactor.

Project Reactor (or just Reactor) is a Reactive library for building non-blocking applications on the JVM and is based on the Reactive Streams Specification.

It's the foundation of the reactive stack in the Spring ecosystem. This will be the topic for the second blog post in this series!

References

[A brief history of ReactiveX and RxJava](#)

[Build Reactive RESTFUL APIs using Spring Boot/WebFlux](#)

[Java SE Flow API](#)

[Project Reactor](#)

[Reactive Manifesto](#)

[Reactive Streams Specification](#)

[ReactiveX](#)

[Spring Web Reactive Framework,](#)

Like This Article? Read More From DZone



DZone Article
A Reactive Emoji Tracker With WebClient and Reactor: Consuming SEE



DZone Article
Spring WebFlux: Writing Filters



DZone Article
Spring 5 WebFlux and JDBC: To Block or Not to Block



Free DZone Refcard
Java 14

Topics: [JAVA](#) , [PROJECT REACTOR](#) , [REACTIVE PROGRAMMING](#) , [SPRING](#) , [TUTORIAL](#) , [WEBFLUX](#)

Published at DZone with permission of Anna Eriksson. See the original article [here](#). 

Published at DZone with permission of Anna Eriksson . [See the original article here.](#) 

Opinions expressed by DZone contributors are their own.

ABOUT US

[About DZone](#)
[Send feedback](#)
[Careers](#)

ADVERTISE

[Developer Marketing Blog](#)
[Advertise with DZone](#)
[+1 \(919\) 238-7100](#)

CONTRIBUTE ON DZONE




[MVB Program](#)
[Zone Leader Program](#)
[Become a Contributor](#)
[Visit the Writers' Zone](#)

LEGAL

[Terms of Service](#)
[Privacy Policy](#)

CONTACT US

600 Park Offices Drive
Suite 150
Research Triangle Park, NC 27709
support@dzone.com
[+1 \(919\) 678-0300](#)

Let's be friends:    [in](#)

DZone.com is powered by

AnswerHub
A DEVADA SOFTWARE PRODUCT