ENGINEERING

Notes on Reactive Programming Part I: The Reactive Landscape

ENGINEERING | DAVE SYER | JUNE 07, 2016 | 36 COMMENTS

ALL POSTS
ENGINEERING
RELEASES
NEWS AND EVENTS

Get updates from The Spring Team delivered to your inbox

Email Address

Yes, I would like to be contacted by The Spring Team and Pivotal for newsletters, promotions and events per the terms of Pivotal's Privacy Policy

Submit

Reactive Programming is interesting (again) and there is a lot of noise about it at the moment, not all of which is very easy to understand for an outsider and simple enterprise Java developer, such as the author. This article (the first in a series) might help to clarify your understanding of what the fuss is about. The approach is as concrete as possible, and there is no mention of "denotational semantics". If you are looking for a more academic approach and loads of code samples in Haskell, the internet is full of them, but you probably don't want to be here.

Reactive Programming is often conflated with concurrent programming and high performance to such an extent that it's hard to separate those concepts, when actually they are in principle completely different. This inevitably leads to confusion. Reactive Programming is also often referred to as or conflated with Functional Reactive Programming, or FRP (and we use the two interchangeably here). Some people think Reactive is nothing new, and it's what they do all day anyway (mostly they use JavaScript). Others seem to think that it's a gift from Microsoft (who made a big splash about it when they released some C# extensions a while ago). In the Enterprise Java space there has been something of a buzz recently (e.g. see the Reactive Streams initiative), and as with anything shiny and new, there are a lot of easy mistakes to make out there, about when and where it can and should be used.

What Is It?

Reactive Programming is a style of micro-architecture involving intelligent routing and consumption of events, all combining to change behaviour. That's a bit abstract, and so are many of the other definitions you will come across online. We attempt build up some more concrete notions of what it means to be reactive, or why it might be important in what follows.

The origins of Reactive Programming can probably be traced to the 1970s or even earlier, so there's nothing new about the idea, but they are really resonating with something in the modern enterprise. This resonance has arrived (not accidentally) at the same time as the rise of microservices, and the ubiquity of multi-core processors. Some of the reasons for that will hopefully become clear.

Here are some useful potted definitions from other sources:

```
The basic idea behind reactive programming is that there are certain
datatypes that represent a value "over time". Computations that
involve these changing-over-time values will themselves have values
that change over time.
```

and…

```
An easy way of reaching a first intuition about what it's like is to
imagine your program is a spreadsheet and all of your variables are
cells. If any of the cells in a spreadsheet change, any cells that
refer to that cell change as well. It's just the same with FRP. Now
imagine that some of the cells change on their own (or rather, are
taken from the outside world): in a GUI situation, the position of
the mouse would be a good example.
```

(from Terminology Question on Stackoverflow)

FRP has a strong affinity with high-performance, concurrency, asynchronous operations and non-blocking IO. However, it might be helpful to start with a suspicion that FRP has nothing to do with any of them. It is certainly the case that such concerns can be naturally handled, often transparently to the caller, when using a Reactive model. But the actual benefit, in terms of handling those concerns effectively or efficiently is entirely up to the implementation in question (and therefore should be subject to a high degree of scrutiny). It is also possible to implement a perfectly sane and useful FRP framework in a synchronous, single-threaded way, but that isn't really likely to be helpful in trying to use any of the new tools and libraries.

Reactive Use Cases

The hardest question to get an answer to as a newbie seems to be "what is it good for?" Here are some examples from an enterprise setting that illustrate general patterns of use:

External Service Calls Many backend services these days are REST-ful (i.e. they operate over HTTP) so the underlying protocol is fundamentally blocking and synchronous. Not obvious territory for FRP maybe, but actually it's quite fertile ground because the implementation of such services often involves calling other services, and then yet more services depending on the results from the first calls. With so much IO going on if you were to wait for one call to complete before sending the next request, your poor client would give up in frustration before you managed to assemble a reply. So external service calls, especially complex orchestrations of dependencies between calls, are a good thing to optimize. FRP offers the promise of "composability" of the logic driving those operations, so that it is easier to write for the developer of the calling service.

Highly Concurrent Message Consumers Message processing, in particular when it is highly concurrent, is a common enterprise use case. Reactive frameworks like to measure micro benchmarks, and brag about how many messages per second you can process in the JVM. The results are truly staggering (tens of millions of messages per second are easy to achieve), but possibly somewhat artificial - you wouldn't be so impressed if they said they were benchmarking a simple "for" loop. However, we should not be too quick to write off such work, and it's easy to see that when performance matters, all contributions should be gratefully accepted. Reactive patterns fit naturally with message processing (since an event translates nicely into a message), so if there is a way to process more messages faster we should pay attention.

Spreadsheets Perhaps not really an enterprise use case, but one that everyone in the enterprise can easily relate to, and it nicely captures the philosophy of, and difficulty of implementing FRP. If cell B depends on cell A, and cell C depends on both cells A and B, then how do you propagate changes in A, ensuring that C is updated before any change events are sent to B? If you have a truly reactive framework to build on, then the answer is "you don't care, you just declare the dependencies," and that is really the power of a spreadsheet in a nutshell. It also highlights the difference between FRP and simple event-driven programming — it puts the "intelligent" in "intelligent routing".

Abstraction Over (A)synchronous Processing This is more of an abstract use case, so straying into the territory we should perhaps be avoiding. There is also some (a lot) of overlap between this and the more concrete use cases already mentioned, but hopefully it is still worth some discussion. The basic claim is a familiar (and justifiable) one, that as long as developers are willing to accept an extra layer of abstraction, they can forget about whether the code they are calling is synchronous or asynchronous. Since it costs precious brain cells to deal with asynchronous programming, there could be some useful ideas there. Reactive Programming is not the only approach to this issue, but some of the implementaters of FRP have thought hard enough about this problem that their tools are useful.

This Netflix blog has some really useful concrete examples of real-life use cases: Netflix Tech Blog: Functional Reactive in the Netflix API with RxJava

Comparisons

If you haven't been living in a cave since 1970 you will have come across some other concepts that are relevant to Reactive Programming and the kinds of problems people try and solve with it. Here are a few of them with my personal take on their relevance:

Ruby Event-Machine The Event Machine is an abstraction over concurrent programming (usually involving non-blocking IO). Rubyists struggled for a long time to turn a language that was designed for single-threaded scripting into something that you could use to write a server application that a) worked, b) performed well, and c) stayed alive under load. Ruby has had threads for quite some time, but they aren't used much and have a bad reputation because they don't always perform very well. The alternative, which is ubiquitous now that it has been promoted (in Ruby 1.9) to the core of the language, is Fibers(sic). The Fiber programming model is sort of a flavour of coroutines (see below), where a single native thread is used to process large numbers of concurrent requests (usually involving IO). The programming model itself is a bit abstract and hard to reason about, so most people use a wrapper, and the Event Machine is the most common. Event Machine doesn't necessarily use Fibers (it abstracts those concerns), but it is easy to find examples of code using Event Machine with Fibers in Ruby web apps (e.g. see this article by Ilya Grigorik, or the fibered example from em-http-request). People do this a lot to get the benefit of scalability that comes from using Event Machine in an I/O intensive application, without the ugly programming model that you get with lots of nested callbacks.

Actor Model Similar to Object Oriented Programming, the Actor Model is a deep thread of Computer Science going back to the 1970s. Actors provide an abstraction over computation (as opposed to data and behaviour) that allows for concurrency as a natural consequence, so in practical terms they can form the basis of a concurrent system. Actors send each other messages, so they are reactive in some sense, and there is a lot of overlap between systems that style themselves as Actors or Reactive. Often the distinction is at the level of their implementation (e.g. `Actors` in Akka can be distributed across processes, and that is a distinguishing feature of that framework).

Deferred results (Futures) Java 1.5 introduced a rich new set of libraries including Doug Lea's "java.util.concurrent", and part of that is the concept of a deferred result, encapsulated in a `Future`. It's a good example of a simple abstraction over an asynchronous pattern, without forcing the implementation to be asynchronous, or use any particular model of asynchronous processing. As the Netflix Tech Blog: Functional Reactive in the Netflix API with RxJava shows nicely, `Futures` are great when all you need is concurrent processing of a set of similar tasks, but as soon as any of them want to depend on each other or execute conditionally you get into a form of "nested callback hell". Reactive Programming provides an antidote to that.

Map-reduce and fork-join Abstractions over parallel processing are useful and there are many examples to choose from. Map-reduce and fork-join that have evolved recently in the Java world, driven by massively parallel distributed processing (MapReduce and Hadoop) and by the JDK itself in version 1.7 (Fork-Join). These are useful abstractions but (like deferred results) they are shallow compared to FRP, which can be used as an abstraction over simple parallel processing, but which reaches beyond that into composability and declarative communication.

Coroutines A "coroutine" is a generalization of a "subroutine" — it has an entry point, and exit point(s) like a subroutine, but when it exits it passes control to another coroutine (not necessarily to its caller), and whatever state it accumulated is kept and remembered for the next time it is called. Coroutines can be used as a building block for higher level features like Actors and Streams. One of the goals of Reactive Programming is to provide the same kind of abstraction over communicating parallel processing agents, so coroutines (if they are available) are a useful building block. There are various flavours of coroutines, some of which are more restrictive than the general case, but more flexible than vanilla subroutines. Fibers (see the discussion on Event Machine) are one flavour, and Generators (familiar in Scala and Python) are another.

Reactive Programming in Java

Java is not a "reactive language" in the sense that it doesn't support coroutines natively. There are other languages on the JVM (Scala and Clojure) that support reactive models more natively, but Java itself does not until version 9. Java, however, is a powerhouse of enterprise development, and there has been a lot of activity recently in providing Reactive layers on top of the JDK. We only take a very brief look at a few of them here.

Reactive Streams is a very low level contract, expressed as a handful of Java interfaces (plus a TCK), but also applicable to other languages. The interfaces express the basic building blocks of `Publisher` and `Subscriber` with explicit back pressure, forming a common language for interoperable libraries. Reactive Streams have been incorporated into the JDK as `java.util.concurrent.Flow` in version 9. The project is a collaboration between engineers from Kaazing, Netflix, Pivotal, Red Hat, Twitter, Typesafe and many others.

RxJava: Netflix were using reactive patterns internally for some time and then they released the tools they were using under an open source license as Netflix/RxJava (subsequently re-branded as "ReactiveX/RxJava"). Netflix does a lot of programming in Groovy on top of RxJava, but it is open to Java usage and quite well suited to Java 8 through the use of Lambdas. There is a bridge to Reactive Streams. RxJava is a "2nd Generation" library according to David Karnok's Generations of Reactive classification.

Reactor is a Java framework from the Pivotal open source team (the one that created Spring). It builds directly on Reactive Streams, so there is no need for a bridge. The Reactor IO project provides wrappers around low-level network runtimes like Netty and Aeron. Reactor is a "4th Generation" library according to David Karnok's Generations of Reactive classification.

Spring Framework 5.0 (first milestone June 2016) has reactive features built into it, including tools for building HTTP servers and clients. Existing users of Spring in the web tier will find a very familiar programming model using annotations to decorate controller methods to handle HTTP requests, for the most part handing off the dispatching of reactive requests and back pressure concerns to the framework. Spring builds on Reactor, but also exposes APIs that allow its features to be expressed using a choice of libraries (e.g. Reactor or RxJava). Users can choose from Tomcat, Jetty, Netty (via Reactor IO) and Undertow for the server side network stack.

Ratpack is a set of libraries for building high performance services over HTTP. It builds on Netty and implements Reactive Streams for interoperability (so you can use other Reactive Streams implementations higher up the stack, for instance). Spring is supported as a native component, and can be used to provide dependency injection using some simple utility classes. There is also some autoconfiguration so that Spring Boot users can embed Ratpack inside a Spring application, bringing up an HTTP endpoint and listening there instead of using one of the embedded servers supplied directly by Spring Boot.

Akka is a toolkit for building applications using the Actor pattern in Scala or Java, with interprocess communication using Akka Streams, and Reactive Streams contracts are built in. Akka is a "3rd Generation" library according to David Karnok's Generations of Reactive classification.

Why Now?

What is driving the rise of Reactive in Enterprise Java? Well, it's not (all) just a technology fad — people jumping on the bandwagon with the shiny new toys. The driver is efficient resource utilization, or in other words, spending less money on servers and data centres. The promise of Reactive is that you can do more with less, specifically you can process higher loads with fewer threads. This is where the intersection of Reactive and non-blocking, asynchronous I/O comes to the foreground. For the right problem, the effects are dramatic. For the wrong problem, the effects might go into reverse (you actually make things worse). Also remember, even if you pick the right problem, there is no such thing as a free lunch, and Reactive doesn't solve the problems for you, it just gives you a toolbox that you can use to implement solutions.

Conclusion

In this article we have taken a very broad and high level look at the Reactive movement, setting it in context in the modern enterprise. There are a number of Reactive libraries or frameworks for the JVM, all under active development. To a large extent they provide similar features, but increasingly, thanks to Reactive Streams, they are interoperable. In the next article in the series we will get down to brass tacks and have a look at some actual code samples, to get a better picture of the specifics of what it means to be Reactive and why it matters. We will also devote some time to understanding why the "F" in FRP is important, and how the concepts of back pressure and non-blocking code have a profound impact on programming style. And most importantly, we will help you to make the important decision about when and how to go Reactive, and when to stay put on the older styles and stacks.

36 Comments spring.io Login
Recommend 42 Tweet Share Sort by Best
Join the discussion…
LOG IN WITH OR SIGN UP WITH DISQUS
Name

Sanjay Yadav • a year ago
great article!!
39 • Reply • Share ›

zeitgeb3r • 3 years ago
No mention of Vert.x? Vert.x originally came from Pivotal, until the major committer moved to Redhat.
7 • Reply • Share ›

Julien Viet → zeitgeb3r • 3 years ago
yes I'm a bit disappointed that this article does not mention it.

In the spirit of fairness, it should definitely have been there.
5 • Reply • Share ›

Pieter_H → zeitgeb3r • 3 years ago
..and then it went to eclipse, and the major committer left the project altogether
3 • Reply • Share ›

Julien Viet → Pieter_H • 3 years ago
Vert.x is alive and kicking.

One tenet of great open source project, is that the project continues even if the creator of the project has left.

Vert.x is an open source project is certainly a project of that rank!
5 • Reply • Share ›

Pieter_H → Julien Viet • 3 years ago
what is the main github repo? I'd like to track the activity but not sure which one to look at.
Reply • Share ›

Julien Viet → Pieter_H • 3 years ago
I would say this repo is the one you are looking for
https://github.com/eclipse/...
1 • Reply • Share ›

Pieter_H → Julien Viet • 3 years ago
thank you!
1 • Reply • Share ›

Adam Smith → zeitgeb3r • a year ago
If someone already knows the subject, he shouldn't be reading the article. It's too long already, and adding more isn't going to help.

It should be trimmed to half the size. Answer the question "what is it" in the first couple sentences, and assume the reader doesn't know much. Immediately explain some
problems it solves. Show a couple examples. Done. Once I know that, I can go find more specialized articles.
2 • Reply • Share ›

Bruno Santos • 2 years ago
Here's a great article about this subject:
https://www.oreilly.com/ide...
2 • Reply • Share ›

Juanjo Martín • 3 years ago
IMHO, Reactive Programming is the programming model that comes out in response to the Reactive Manifesto (http://www.reactivemanifest....
It has got quite adoption and now, several frameworks, libraries, tools and other stuff are emerging. I do not know which one will win the battle in the end. Nowadays, maybe the Scala+Play+Akka can be the strongest bet because it was born "Reactive" (though hard to adapt if you come from traditional Java OO).
However, the Spring 5 intention to offer a parallel non-blocking MVC framework (with the same programming model) and the standarization of Reactive Streams can change this landscape in the future. Who knows...
3 • Reply • Share ›

Rahul Gulabani → Juanjo Martín • 2 years ago
Currently with Akka -Http is out, the stack for reactive application is Akka -http(REST API) + Akka + Scala. Spray was also there sometime ago now they merged with Akka http.
Regarding winning the race again I say both languages has It's own advantages
Java is highly used in professional development but still need to catch up with Scala in race of reactive support .
Scala is born with reactive support but still many programmers don't use it still.
Reply • Share ›

Anbu Sampath • 3 years ago
Good Article on reactive.. looking forward for rest of the series.
1 • Reply • Share ›

Mohammad Nawazish Khan • 9 months ago
What exactly is the protocol behind a reactive framework like Webflux? How to make an http endpoint reactive? Do they employ websockets underneath the hood? Also, what is the mechanism of streaming Http?
Reply • Share ›

Jason.Chen • 10 months ago
Really great article introducing reactive programing. Thanks a lot for sharing! ~ Glad I found it.
Reply • Share ›

Uttar • 10 months ago
Good Introduction to reactive programming....
Reply • Share ›

Sanjay Yadav • a year ago
Wriiten a basic rest rest api , using reactive spring , its fun .
http://frugalisminds.com/sp...
Reply • Share ›

li yang • a year ago
in Reactive Use Cases:External Service Calls
assume we have 3 external calls, A,B,C external services. If they do not coordinate with each other, we can use java Future to call in parallel without reactive; if they depend on each other, more specifically, we need A's result when i call B, and we need B's result when i call C, so they must execute one by one , and reactive can not help it.
So how can Reactive fit for External Service Calls?
Reply • Share ›

kuldeep • a year ago
dont think reactive has any direct connection with microservices....having multi core cpus after 2004 is the right motivation.
Reply • Share ›

Aaron Zhang • a year ago
Great article, thank you!
Reply • Share ›

BecarioEstrella • a year ago
we is the next article of the serie and why not link it to the previous?
Reply • Share ›

minseok kim • a year ago
it is too late to comment but ,thank you for nice blog
Reply • Share ›

Gui • a year ago
Nice post on Reactive. Thank you !
Reply • Share ›

Binh Thanh Nguyen • 2 years ago
Thanks, nice post
Reply • Share ›

Clemente Biondo • 2 years ago
What a well written article. Thank you.
Reply • Share ›

Weiping Guo • 2 years ago
I was looking for some Reactive Programming intro for Java developer. Glad I found it! Excellent article, thank you!
Reply • Share ›

Miloš Milivojević • 3 years ago
Excellent article, thank you!
Reply • Share ›

Mark • 3 years ago
It looks like the part III article somehow replaced the part I article that should be here?
Reply • Share ›

Dave Syer → Mark • 3 years ago
Oops, thanks (copy-paste error I think). Should be fixed now.
1 • Reply • Share ›

Abhijit Sarkar → Dave Syer • 3 years ago
Looks like part I and II are exactly the same. Another copy-paste error?
The URLs below:
https://spring.io/blog/2016...
https://spring.io/blog/2016...
Reply • Share ›

Dave Syer → Abhijit Sarkar • 3 years ago
Indeed, yes. Fixed it now. Late night copy-paste.
Reply • Share ›

Jose Miguel Salcido Aguilar • 3 years ago
NOICE!
Reply • Share ›

yuraminsk • 3 years ago
good intro!
Reply • Share ›

Subrahmanyam Devarakonda • 3 years ago
Thanks a lot. That clears a lot of ground on Reactive programming paradigms.
Reply • Share ›

Diego Lovison • 3 years ago
Excelent article.. Congrats
Reply • Share ›

Jianhui Zhou • 3 years ago
Good jon, thanks
Reply • Share ›

ALSO ON SPRING.IO

Spring Cloud Finchley.SR3 Now Available
2 comments • 15 days ago
Hronom — When looking on Spring Cloud updates always curious about Spring Boot version compatibility, where I can check …

Spring Tips: JavaFX
2 comments • 2 months ago
Julian Jupiter — Thank you for this simple yet powerful tutorial. I've been looking for tutorial for Spring Boot and JavaFX. I can't seem …

Flight of the Flux 1 - Assembly vs Subscription
3 comments • 7 days ago
Ông Ngoại Của Bạn — hello,thanks for your interesting posti'm wondering if there is a solution whre subcriber retreives data data …

Spring Cloud Data Flow and Skipper 2.0 RC1 Released
1 comment • 19 days ago
Kim Nielsen — It's stated that the GA is planned for end of February, and we're now in March?!

Subscribe | Add Disqus to your site | Disqus' Privacy Policy | DISQUS

© 2019 Pivotal Software, Inc. All Rights Reserved. Terms of Use • Privacy • Trademark Guidelines