Event-Driven Architecture - Part 2

This lesson contains the second part of the discussion on event-driven architecture. We will continue where we left off in the previous lesson.

We'll cover the following

What are events?#

There are generally two kinds of processes in applications: *CPU intensive* and *IO intensive*. In the context of web applications, IO means *events*. A large number of *IO* operations mean a lot of events occurring over a period of time, and an event can be anything from a tweet to a click of a button, an HTTP request, an ingested message, a change in the value of a variable, etc.

We know that Web 2.0 real-time applications have a lot of events. For instance, there is a lot of request-response between the client and the server, typically in an online game, messaging app etc. *Events* happening too often is called a *stream of events*. In the previous chapter, we discussed how stream processing works.

Event-driven architecture#

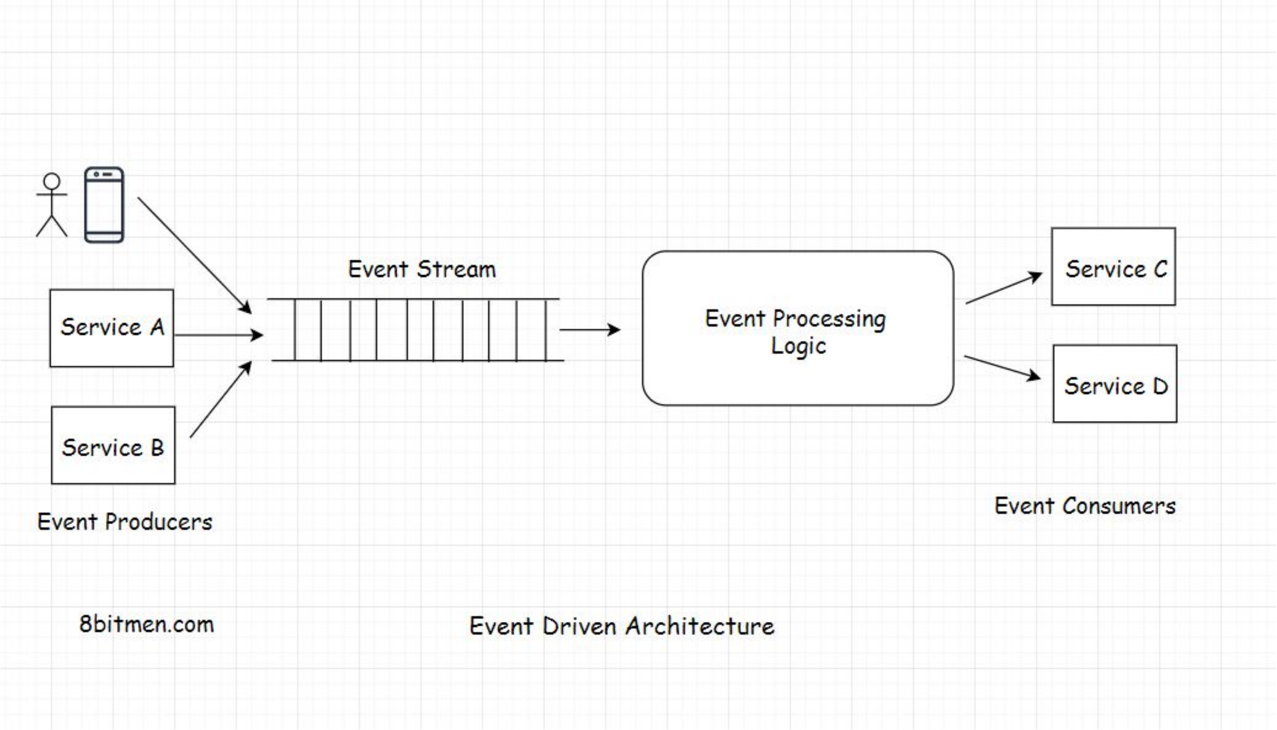
Non-blocking architecture is also known as *reactive* or *event-driven* architecture. *Event-driven* architectures are pretty popular in modern web application development.

Technologies like *NodeJS*, frameworks in the *Java* ecosystem like *Play*, and *Akka.io (http://Akka.io)* are *non-blocking* in nature and are built for modern high *IO* scalable applications.

They are capable of handling a big number of concurrent connections with minimal resource consumption. Modern applications need a fully asynchronous model to scale. These modern web frameworks provide more reliable behavior in a distributed environment. They are built to run on a cluster, handle large scale concurrent scenarios, and tackle problems that generally occur in a clustered environment. They enable us to write code without worrying about handling *multi-threads, thread lock, out of memory issues* due to high *IO* etc.

Returning to *Event-driven reactive* architecture. It simply means reacting to the events occurring regularly. The code is written to react to the events as opposed to sequentially moving through the lines of codes.

I've already brought this up, but the sequence of events occurring over a period of time is called a *stream of events*. In order to react to the events, the system has to continually monitor the stream. *Event-driven* architecture is all about processing *asynchronous data streams*. The application becomes inherently asynchronous.



Technologies for implementing the event driven architecture#

With the advent of Web 2.0, people in the tech industry felt the need to evolve the technologies to be powerful enough to implement modern web application use cases. *Spring Framework* added *Spring Reactor* module to the core *Spring repo*. Developers wrote *NodeJS, Akka.io (http://Akka.io), Play* etc.

So, you would have already figured that *reactive, event-driven* applications are difficult to implement with thread-based frameworks. As dealing with *threads*, shared *mutable state*, and *locks* make things a lot more complex, in an *event-driven* system everything is treated as a *stream*. The level of abstraction is good, developers don't have to worry about managing the low-level memory stuff.

low level memory stuff.

I am sure you are well aware of the data streaming use cases that apply here, like handling a large number of transaction events, handling changing stock prices, user events on an online shopping application, etc.

NodeJS is a single-threaded *non-blocking* framework written to handle more *IO* intensive tasks. It has an *event loop* architecture. This is a good read on it. (https://nodejs.org/fa/docs/guides/event-loop-timers-and-nexttick/)

LinkedIn uses the *Play* framework for identifying the online status of its users. (https://www.8bitmen.com/linkedin-real-time-architecture-how-does-linkedin-identify-its-users-online/)

At the same time, I want to assert the fact that the emergence of *non-blocking* tech does not mean that traditional tech is obsolete. Every tech has its use cases.

NodeJS is not fit for *CPU intensive* tasks. *CPU intensive* operations are operations that require a good amount of computational power like for graphics rendering, running ML algorithms, handling data in enterprise systems etc. It would be a mistake to pick *NodeJS* for these purposes.

In the upcoming lessons, I will discuss the general guidelines to keep in mind when picking the server-side technology. This will give a better insight into how to pick the right backend technology.

← Back

Event-Driven Architecture - Part 1

Next →

Webhooks

☑ Mark as Completed

Report an Issue

Report an issue