**Event driven Architecture (EDA)**

Традиционная модель Клиент-Сервер говорит о том, что сервер ждет запроса от клиента, а только потом отвечает ему.

Например если нужно узнать приехал ли автобус, то мы посылаем запросы на сервер “Приехал ли автобус” и делаем это каждый раз, пока ответ нас не устроит (автобус приехал)



В Event driven Architecture клиент и сервер подписаны друг на друга и отправляют друг другу сообщения, когда условие будет выполнено (приедет автобус) - **Webhooks**

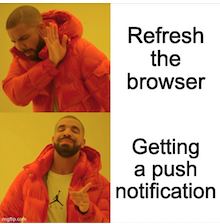


В такой архитектуре Event Producers и Consumers не зависят друг от друга. И могут отправлять асинхронные запросы

**Polling is dead. We must move on**

We, as information consumers, have a craving desire to know things as they happen.

Where is my package right now? What’s the score of the game? How is Dogecoin performing today? Likewise, the list goes on. A majority of Internet users today would like information to be pushed towards them rather than pulling.



As an application builder, how do you push information to users as they happen?

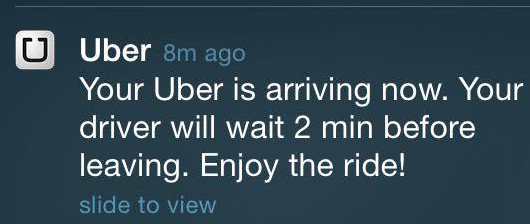
Today, a good amount of Internet applications are powered by HTTP APIs. Their interaction model is request-response driven and synchronous. The only way for a consumer to know what’s going on the server is to keep polling the server. Polling is dreadful, wastes valuable CPU cycles on both ends.

So, there must be some way to improve this interaction model.

**The modern user experience**

But, some exciting developments are happening in this space lately.

If you happen to use applications produced by major league Internet companies such as FaceBook, Instagram, or Uber, you must’ve experienced the seductive real-time interactions offered by them. For example, Facebook notifies you instantly when someone liked your content. Uber tells you where your ride is at the moment and how long it’ll take to get there.



These interactions glue users to their platform, improving the engagement never before.

Now, our question is, how do they do that? If they can, why can’t you? Let’s find out.

**The foundations of event-driven APIs**

The REST-style HTTP APIs often have a unidirectional and synchronous interaction model with their consumers. To get the latest information, the consumers always have to poll the API backend.

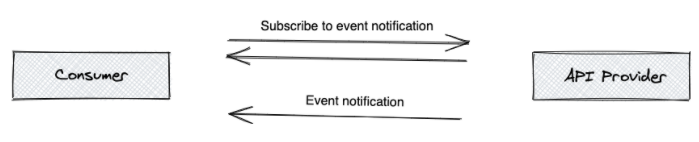
What if we turn this model upside down and allow the backend to notify the client when something interesting happens?

Well, that is the basis of “**Event-enabled**” APIs. They are often called “**asynchronous**,” “**push**,” or “**streaming**” APIs as they keep pushing information to the client, in contrast to the polling.

An event-driven API must offer two capabilities to its consumers.

1. A mechanism to allow consumers to subscribe to events of their interest.
2. Deliver events to subscribed consumers in an asynchronous manner.

Based on these, we can define an interaction model like the following for event-driven APIs.



**1. The client subscribes with the API**

At this point, the client of the event-driven API registers its intent to receive asynchronous updates with the API. That is called the subscription. Upon subscription, the client often specifies an endpoint to which the API should post update events.

**2. Asynchronous event delivery**

When something interesting happens at the backend, API delivers that to all subscribed clients in the form of an event, asynchronously.

**Usa cases of all technologies**



**Documenting event-driven APIs with AsyncAPI**

A good API definition is complemented by comprehensive documentation and a set of language-specific code generators. REST APIs are meeting that need with the OpenAPI specification. Fortunately, for event-driven APIs, we have the AsyncAPI specification.

The AsyncAPI specification is a machine-readable document that documents and describes your event-driven APIs. It is not only just a specification but a rich ecosystem full of code generators, validators, and test generators.

AsyncAPI is designed along with the same elements of OpenAPI and shares many common constructs to simplify the adoption, but it also comes with additional features to accommodate eventing. It supports a wide variety of messaging protocols and transports (such as AMQP, MQTT, WebSockets, Kafka, JMS, STOMP, HTTP, etc.) and event schema formats. Therefore, the API definition will contain the event payload definition, channel name, application/transport headers, protocols, and other eventing semantics to connect, publish, and subscribe to the API. — Dakshitha Ratnayake

If you are starting your event-driven API journey today, I strongly suggest you follow the AsyncAPI initiative.