https://stackoverflow.blog/2020/03/16/how-event-driven-architecture-solves-modern-web-app-problems/

In [computer programming](https://en.wikipedia.org/wiki/Computer_programming), **event-driven programming** is a programming paradigm in which the [flow of the program](https://en.wikipedia.org/wiki/Control_flow) is determined by [events](https://en.wikipedia.org/wiki/Event_(computing)) such as user actions ([mouse](https://en.wikipedia.org/wiki/Computer_mouse) clicks, key presses), [sensor](https://en.wikipedia.org/wiki/Sensor) outputs, or [message passing](https://en.wikipedia.org/wiki/Message_passing) from other programs or [threads](https://en.wikipedia.org/wiki/Thread_(computer_science)). Event-driven programming is the dominant paradigm used in [graphical user interfaces](https://en.wikipedia.org/wiki/Graphical_user_interface) and other applications (e.g., JavaScript [web applications](https://en.wikipedia.org/wiki/Web_application)) that are centered on performing certain actions in response to [user input](https://en.wikipedia.org/wiki/Input/output).

**How event-driven architecture solves modern web app problems**

Web-based applications have come a long way since we used to serve static HTML content from servers. Nowadays, applications are much more complex and use multiple frameworks, data centers, and technologies. In the last couple of years, we’ve seen two  concepts dominate the IT market:

* moving our apps to the **cloud**;
* implementing a **microservices** architecture;

These ideas have shaped the way we design and build software. In a way, we are no longer building **applications**; instead, we are building **platforms**. Apps no longer share the same computational space. Instead, they have to communicate with each other through lightweight communication protocols, like REST APIs or RPC calls. This model made it possible to create some amazing software like Facebook, Netflix, Uber, and so many others.

In this article, we’ll discuss some of the problems driving innovation in modern web development. Then we’ll dive into the basics of event-driven architecture (EDA), which tries to address these problems by thinking about back-end architecture in a novel way.

## The problems facing the modern web

Each technology has to handle the challenges that always-on, multi-user, asynchronous applications face today:

### Availability

Instead of one application, we now have many, dozens maybe even hundreds of linked services, and each and one of them has to be ready to do its job 24/7. How can we do that? Most often the service is horizontally scaled to multiple instances, sometimes across multiple data centers, making it highly-available. All the requests coming into this specific service will be routed evenly across all instances. Some deployment tools offer self-healing capabilities, so if one instance goes down, it will create another instance to take its place.

### Scalability

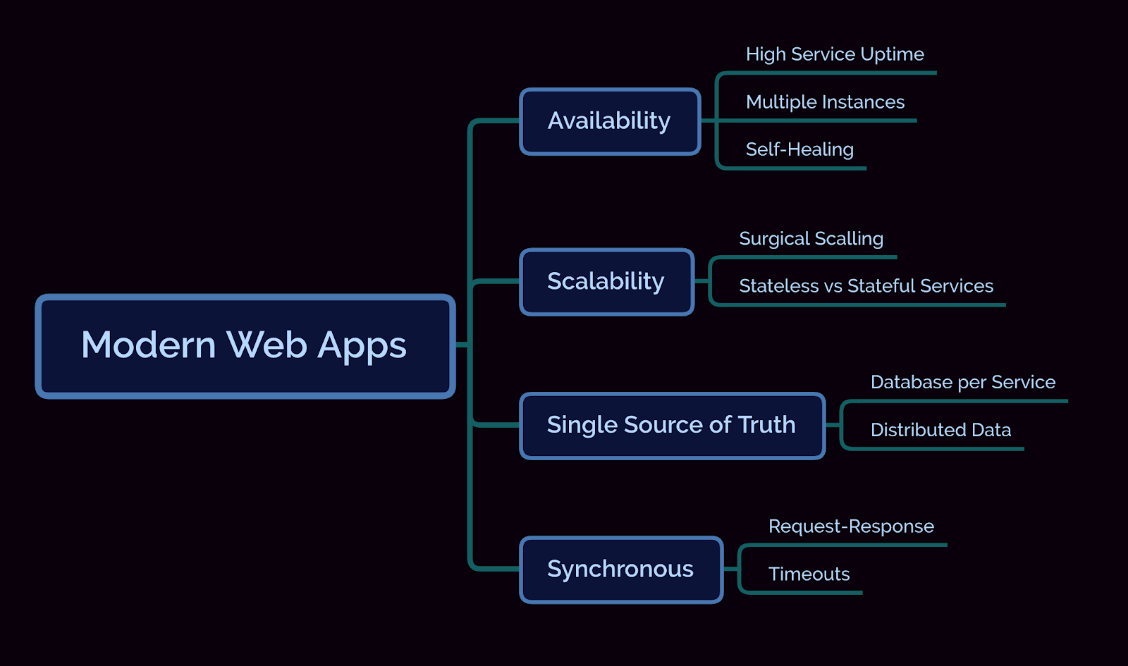
Scalability has much in common with **availability**. Availability is all about making sure that there is at least one instance of the service up and running, ready to serve incoming requests. Scalability, on the other hand, is focused on performance. If one application is overloaded, then we can create new instances of that application to accommodate the increased number of requests. But scaling up applications is not challenge-free, especially if we deal with [stateful applications](https://nordicapis.com/defining-stateful-vs-stateless-web-services/).

### Single source of truth

Before microservices, this job was simple. All the data resided in a single place, typically some sort of relational database. But when multiple services share a database, you may create problems like dependencies between teams on schema changes or performance issues. A common pattern to solve this issue is to use a database per service. A distributed source of truth really helps maintain clean architecture, but we now have to deal with distributed transactions and the complexity of maintaining multiple databases.

### Synchronous

In a typical request-response scenario, the client waits for the server to respond; it blocks all its activities until it receives a response or the timeout expires. If we take this behaviour and put it in a microservices architecture using chained calls across the system, we can easily end up with what I call “Microservices Hell.” Everything starts with one service call, let’s call it service A. But then, service A needs to call service B, and the fun goes on. The problem with this behaviour is that if a service has resources blocked (e.g: a thread is hanging), timeouts are now exponential. If we allow a 500 ms timeout per service and there are five service calls in the chain, then the first service would need to have a 2500 ms (2.5 seconds) timeout, whereas the last service would need to have a 500 ms timeout.



## Introducing event-driven architecture

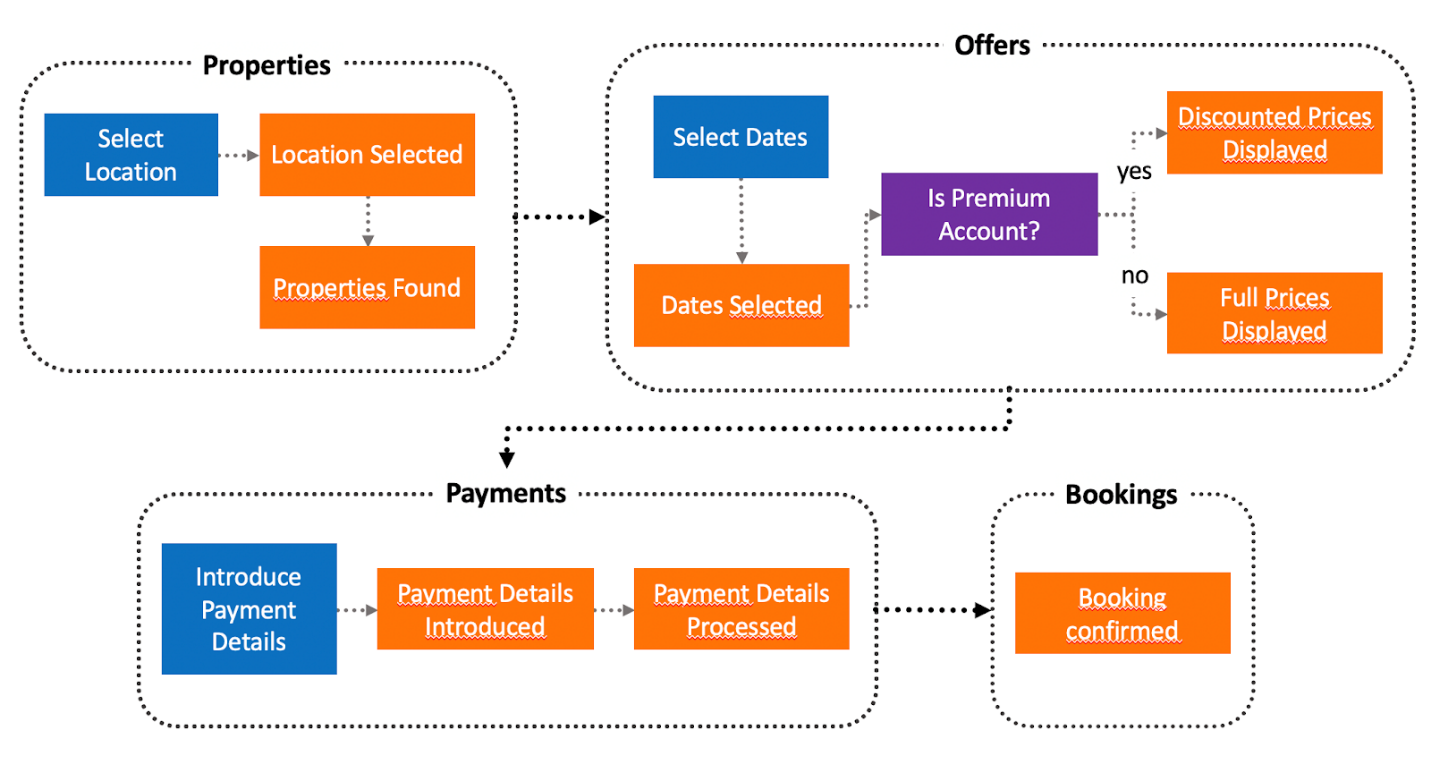
**Event-driven architecture** (**EDA**) is a [software architecture](https://en.wikipedia.org/wiki/Software_architecture) paradigm promoting the production, detection, consumption of, and reaction to [**events**](https://en.wikipedia.org/wiki/Event_(computing)).

*-Wikipedia*

In the classic three-tier applications, the core of our system was the data(base). In EDA, the focus is shifted towards the events and how they are flowing through the system. This shift allowed us to completely change the way we design applications tackling the problems mentioned above.

Before actually seeing how EDA does that, let’s see what exactly an event is. An event is an action that triggers either a notification or some kind of change in the state of the application. A light has been switched on (*notification*), the thermostat turned off the heating system (*notification*), a user changed his address (*state change*), or one of your friends changed his phone number (*state change*). All of these are events, but that doesn’t mean we should add them to an event-driven solution. In order for an event to be added, it must be relevant to the business. A user placing a new order is a relevant event for that specific business, but him/her eating pizza for lunch is not.

Which events are relevant to a business might be obvious when you think about them, but some of them might not. Especially those events that occur as a reaction to other events. To discover events that are flowing through the system, use a technique called [**Event Storming**](https://www.eventstorming.com/). Bring together the stakeholders on an application (from software engineers to business people and domain experts) and map out all the business processes as specific events. After all the business processes are mapped, the result can be used by engineering teams as requirements to build their applications.



aving figured out what events are and how they can be identified, let’s have a look over how they can solve the common problems mentioned earlier.

Events flow in a single direction, from a producer to a consumer. Compare this with a REST call. The event producer never expects a response from the consumer, while in a REST call there will always be a response. No response, no need to block the code execution until something else happens. This makes events **asynchronous** by nature, completely eliminating the risk of running through timeouts.

Events happen as the result of an action, so there is no target system; we can’t really say service A triggers events to service B; what we can say is service B is interested in the events produced by service A. But there may be some other parties interested as well, like service C or D.

So how can we make sure that an event triggered by one system, reaches all the interested services? Most of the time this problem is solved by message brokers. A broker is nothing more than an application that acts as a middle-man between the event generator (the application that created the event) and the event consumer. This way, the applications are nicely decoupled taking care of the **Availability** issue I talked about earlier in the post. If an application is not available momentarily, when it comes back online, it will start consuming events and processing them, catching up with all the events triggered when the application was down.

What about storage? Can events be stored in a database or will there be something else in place? Events can definitely be stored in databases, but by doing so, they lose their “event” aspect. After an event happens, we cannot change it, so events are immutable. Databases on the other hand… they are mutable, we can actually change data after it has been inserted.

A better approach to store events is by using ***Event Logs***. Event logs are nothing more than a **centralized** data store where each event is stored as a sequence of immutable records, also called a log. Imagine the log as a journal, where each new event is appended to the end of the list. We can always recreate the latest state by replaying all the events from log from the beginning until present.

The only bit that I haven’t covered yet is scalability. Services built using the event-driven mindset are designed to be deployed in a multi-instance scenario. Since the state itself is stored in the event log, the service itself will be stateless, which allows surgical scaling of any particular service that we want.

The only exception to this pattern are services that have to create **materialized views**. In essence, a materialized view represents the state in a point in time of an event log. This approach is used to query the data more easily. Coming back to our scalability issue, materialized view is nothing more than events aggregated in a table like format, but where do we store these tables? Most often, we see these aggregations performed in memory, which automatically transforms our service into a stateful one. A quick and easy solution is to add a local database to each service that creates materialized views. This way, the state is stored in the database and the service is once again stateless.

