**Serverless architectures** are application designs that incorporate third-party “Backend as a Service” (BaaS) services, and/or that include custom code run in managed, ephemeral containers on a “Functions as a Service” (FaaS) platform. By using these ideas, and related ones like single-page applications, such architectures remove much of the need for a traditional always-on server component. Serverless architectures may benefit from significantly reduced operational cost, complexity, and engineering lead time, at a cost of increased reliance on vendor dependencies and comparatively immature supporting services.

**What is Serverless?**

1. Serverless was first used to describe applications that significantly or fully incorporate third-party, cloud-hosted applications and services, to manage server-side logic and state. These are typically “rich client” applications—think single-page web apps, or mobile apps—that use the vast ecosystem of cloud-accessible databases (e.g., Parse, Firebase), authentication services (e.g., Auth0, AWS Cognito), and so on. These types of services have been previously described as “(Mobile) Backend as a Service", and I use "BaaS" as shorthand in the rest of this article.
2. Serverless can also mean applications where server-side logic is still written by the application developer, but, unlike traditional architectures, it’s run in stateless compute containers that are event-triggered, ephemeral (may only last for one invocation), and fully managed by a third party. One way to think of this is “Functions as a Service” or "FaaS". (Note: The original source for this name—a tweet by @marak—is no longer publicly available.) AWS Lambda is one of the most popular implementations of a Functions-as-a-Service platform at present, but there are many others, too.

We’ll primarily focus on FaaS. Not only is it the area of Serverless that’s newer and driving a lot of the hype, but it has significant differences to how we typically think about technical architecture.

BaaS and FaaS are related in their operational attributes (e.g., no resource management) and are frequently used together. The large cloud vendors all have “Serverless portfolios” that include both BaaS and FaaS products—for example, here’s Amazon’s Serverless product page. Google’s Firebase BaaS database has explicit FaaS support through Google Cloud Functions for Firebase.

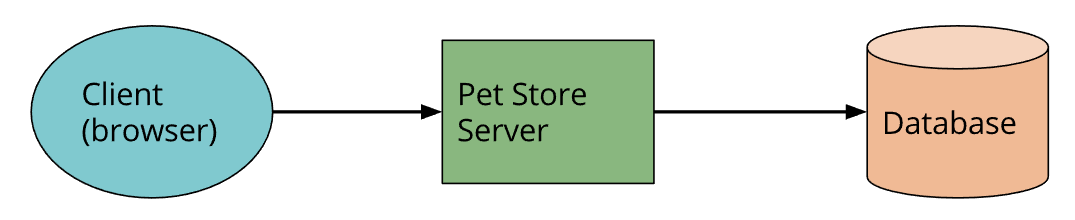
There is similar linking of the two areas from smaller companies too. Auth0 started with a BaaS product that implemented many facets of user management, and subsequently created the companion FaaS service Webtask. The company have taken this idea even further with Extend, which enables other SaaS and BaaS companies to easily add a FaaS capability to existing products so they can create a unified Serverless product.

**Examples**

**UI-driven applications**

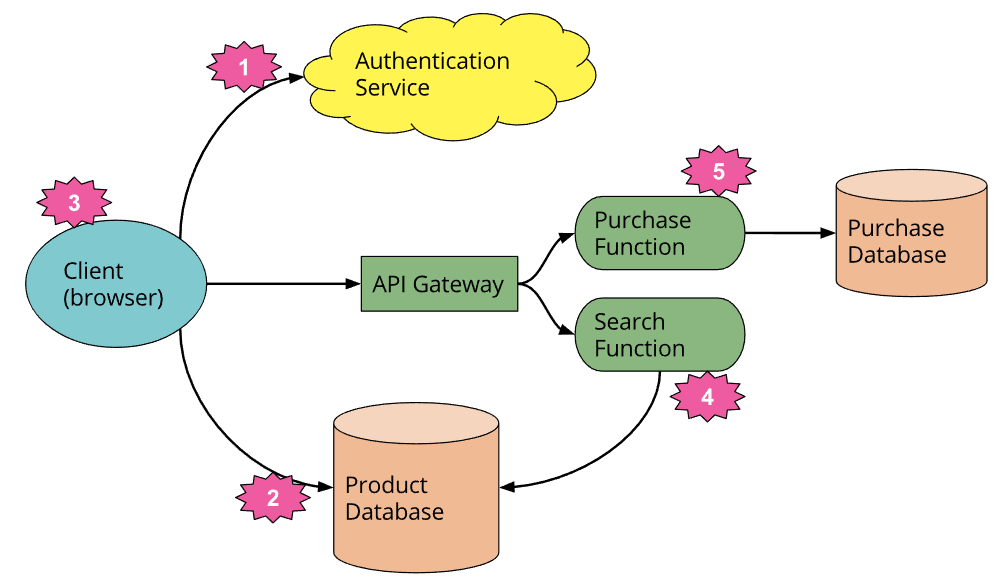
Let’s think about a traditional three-tier client-oriented system with server-side logic. A good example is a typical ecommerce app—dare I say an online pet store?

Traditionally, the architecture will look something like the diagram below. Let’s say it’s implemented in Java or Javascript on the server side, with an HTML + Javascript component as the client



With this architecture the client can be relatively unintelligent, with much of the logic in the system—authentication, page navigation, searching, transactions—implemented by the server application.

With a Serverless architecture this may end up looking more like this:



This is a massively simplified view, but even here we see a number of significant changes:

1. We’ve deleted the authentication logic in the original application and have replaced it with a third-party BaaS service (e.g., Auth0.)
2. Using another example of BaaS, we’ve allowed the client direct access to a subset of our database (for product listings), which itself is fully hosted by a third party (e.g., Google Firebase.) We likely have a different security profile for the client accessing the database in this way than for server resources that access the database.
3. These previous two points imply a very important third: some logic that was in the Pet Store server is now within the client—e.g., keeping track of a user session, understanding the UX structure of the application, reading from a database and translating that into a usable view, etc. The client is well on its way to becoming a [Single Page Application](https://en.wikipedia.org/wiki/Single-page_application).
4. We may want to keep some UX-related functionality in the server, if, for example, it’s compute intensive or requires access to significant amounts of data. In our pet store, an example is “search.” Instead of having an always-running server, as existed in the original architecture, we can instead implement a FaaS function that responds to HTTP requests via an API gateway (described later). Both the client and the server “search” function read from the same database for product data.

If we choose to use AWS Lambda as our FaaS platform we can port the search code from the original Pet Store server to the new Pet Store Search function without a complete rewrite, since Lambda supports Java and Javascript—our original implementation languages.

1. Finally, we may replace our “purchase” functionality with another separate FaaS function, choosing to keep it on the server side for security reasons, rather than reimplement it in the client. It too is fronted by an API gateway. Breaking up different logical requirements into separately deployed components is a very common approach when using FaaS.

Stepping back a little, this example demonstrates another very important point about Serverless architectures. In the original version, all flow, control, and security was managed by the central server application. In the Serverless version there is no central arbiter of these concerns. Instead we see a preference for **choreography over orchestration**, with each component playing a more architecturally aware role—an idea also common in a microservices approach.

There are many benefits to such an approach. As Sam Newman notes in his [*Building Microservices*](https://samnewman.io/books/building_microservices/) book, systems built this way are often “more flexible and amenable to change,” both as a whole and through independent updates to components; there is better division of concerns; and there are also some fascinating cost benefits, a point that Gojko Adzic discusses in [this excellent talk](https://gojko.net/2017/10/05/serverless-design-gotocph.html).

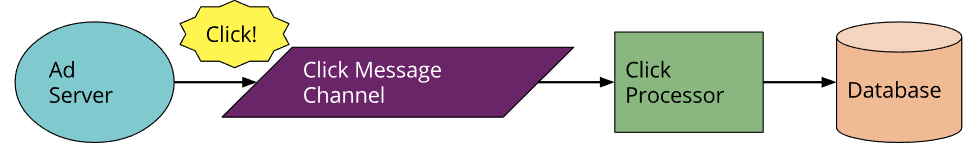
Of course, such a design is a trade-off: it requires better distributed monitoring (more on this later), and we rely more significantly on the security capabilities of the underlying platform. More fundamentally, there are a greater number of moving pieces to get our heads around than there are with the monolithic application we had originally. Whether the benefits of flexibility and cost are worth the added complexity of multiple backend components is very context dependent.

**Message-driven applications**

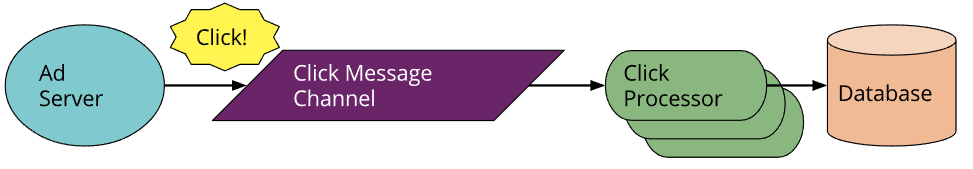
A different example is a backend data-processing service.

Say you’re writing a user-centric application that needs to quickly respond to UI requests, and, secondarily, it needs to capture all the different types of user activity that are occurring, for subsequent processing. Think about an online advertisement system: when a user clicks on an ad you want to very quickly redirect them to the target of that ad. At the same time, you need to collect the fact that the click has happened so that you can charge the advertiser.

Traditionally, the architecture may look as below. The “Ad Server” synchronously responds to the user (not shown) and also posts a “click message” to a channel. This message is then asynchronously processed by a “click processor” application that updates a database, e.g., to decrement the advertiser’s budget.



In the Serverless world this looks as follows:



Can you see the difference? The change in architecture is much smaller here compared to our first example—this is why asynchronous message processing is a very popular use case for Serverless technologies. We’ve replaced a long-lived message-consumer *application* with a FaaS *function*. This function runs within the event-driven context the vendor provides. Note that the cloud platform vendor supplies both the message broker *and* the FaaS environment—the two systems are closely tied to each other.

The FaaS environment may also process several messages in parallel by instantiating multiple copies of the function code. Depending on how we wrote the original process this may be a new concept we need to consider.

**Advantages**

* Pay per usage time. Мы по сути арендуем серверное время в облаке и не поддерживаем сам сервер (за нас это делают провайдеры облачного сервера). И в отличие от простой аренды сервера в облаке (где нам бы просто выделили HardWare ресурсы и создали виртуалку на ESXi), здесь при каждом обращении к серверу создается контейнер, который отрабатывает и удаляется. Из за этого мы не платим за время простоя, а только за работ сервера.
* Decrease the load on DevOps. Девопсам не нужно поддерживать сервера, так как они поддерживаются провайдерами
* Just write your business logic.

**Disadvantages**

* Tied to a particular vendor (AWS Lambda, …)
* May become expensive at some point
* Limited resources