**What is Serverless?**

Serverless Computing execution model in which the cloud partner runs the server, and dynamically manages the allocation of machine resources.

**Difference Between Cloud Services and Serverless:**

The key differentiator between Cloud services and Serverless is the **pricing model**. With Serverless, you pay exactly for the computing power and resources you use. When nothing's running on your EC2 instance, you still have to pay for it - your bill is based on how long the instance was up, even though nothing was running on it. On the other hand, when using the Serverless equivalent, such as AWS Lambda, the bill would be based on the duration of your actual code invocation. There is no infrastructure **maintenance** as such.

**Disadvantages of Serverless:**

Every technology has some drawbacks which are given below:

**Local Development**: As the technology is new, the ecosystem of tools that allows for the local development of serverless-based services is still quite small. To overcome that, developers often have to stub Cloud services such as databases. It's also possible to run a cloud service locally using tools like [Local Stack](https://github.com/localstack/localstack) that try to emulate the most popular AWS services. Serverless and event-based architectures require us a high coverage of unit tests as that is often the only way to test the application locally.

**Performance**: A new term that comes with this technology is "a cold start". That is the time needed to spin up a Lambda function or AWS Aurora database for the first time - after the first invocation or first query to the database. For Lambda, in the worst-case scenario, it can be up to 10-20 seconds and AWS Aurora Serverless may need about 30 seconds to respond. However, many techniques could be used to reduce that time - starting from warming up the functions and ending with choosing a programming language with less overhead.

**Serverless-Powered Services:**

When looking at the product portfolio of major Cloud providers, we can see that most of them are already available in the Serverless model, starting from the most well-known computing platforms, such as AWS Lambda or Google Cloud Functions, and ending with more exotic application services such as AWS Cognito, used for authentication.

|  |  |
| --- | --- |
| **Service Type** | **Examples** |
| Computing | * AWS Lambda * Azure Functions * Google Cloud Functions |
| Databases | * AWS Aurora Serverless * AWS DynamoDB * Google Cloud Datastore |
| Files Storage | * AWS S3 * Azure Blob Storage * Google Cloud Storage |
| Data Warehouse | * Google Cloud Big Query |
| Queuing, messaging | * AWS SQS, AWS SNS * Google Cloud Pub/Sub |
| Content Delivery | * AWS CloudFront * Azure CDN * Google Cloud CDN |
| Authentication | * AWS Cognito * Google Firebase Authentication |

**Serverless Architectures:**

In the given below, we talk about that how we can compose our architecture to fit into the serverless model - which services we can choose and what characteristics are. Moreover, you will also learn of the key pricing factors of each architecture.

**Static marketing page:**

We start with a simple architecture that allows us to host a static HTML site with all its assets for that, we will use two well-known AWS services - **Amazon CloudFront** and **Amazon S3**.

**Architecture:**



**Key components**

**Amazon CloudFront** is a CDN service that allows us to deploy our website under a custom domain with a free, auto-renewed SSL certificate. With enabled caching on the CloudFront edges, we can expect a response time of about 40ms across the whole globe.

**Amazon S3** is an object storage service that allows for the storage of our website's static files. It can't be used for dynamic pages; however, it suits perfectly as a place to keep files generated with static site generators

**Single page application:**

The SPA architecture consists of two layers:

* **Frontend** - a static HTML site, commonly using modern JavaScript frameworks like Vue, React, or Angular.
* **Backend –** a dynamic code that handles business logic and stores data in a database.

For the frontend part, the choice is easy as we can just use the previous architecture. However, as we have here a dynamic part that moreover has to store data somewhere, we can't end up with Amazon S3. We have to use cloud services that allow for serverless **computing** and a **database**. That's why we will use **Amazon API Gateway** for handling AJAX requests proxied to **Amazon Lambda** which invokes our Node.js code. For our database, we will go with **Amazon DynamoDB**.

**Architecture:**

**Key components:**

**Amazon Lambda** is a service on which we can run our custom, dynamic code. It supports a variety of programming languages such as Node.js, Python, Ruby, Java, Go, C#, and even PowerShell. For vertical scaling, we choose a memory that receives our function which corresponds linearly in proportion to CPU power. We do the required business logic in Lambda and store the results in Amazon DynamoDB.

**Amazon DynamoDB** is the last brick of the architecture. It's a key-value and document database. It provides consistent, single-digit millisecond response times. As it's a fully-managed service, we get high availability and auto-scaling for both computing and storage out of the box. However, as with every NoSQL, we have to think about our data design quite differently than in a traditional relational database - with DynamoDB we focus on how we will query data. Forget about the third normal form and denormalize your data to be easily retrievable.

**Event-driven architecture:**

The previous SPA architecture is extended with the ability to asynchronously communicate with external and internal services using Event-Driven Architecture. **Amazon SNS** and **Amazon SQS** are used to enable communication between systems in a highly scalable and reliable way. During the execution of our Lambda function, we send notifications to other systems that a new event happened in the system.