What is serverless and when should you use it?

What is a microservice?

It’s impossible to talk about serverless without talking about its origins: the microservice. A microservice architecture is a type of organisation for software development. An application is composed of small independent services (called microservices) with well-defined APIs that communicate with other microservices. Naturally, each microservice has a single responsibility because it has its own code or implementation that is separated from other services making it only take on a single responsibility. The inputs and outputs of the microservice are unspecialized instead of only relating to the domain of the problem. Any application that is compatible with the microservice’s API (usually REST calls, but can be others) can interface with that service, instead of having to know an output’s data access model or the appropriate format of an input object. Another benefit of microservices is scalability. If a lot of traffic is reaching the microservice, you don’t need to scale up the entire application, you only need to scale up the service getting more traffic which reduces start up time and increases availability. For the same reason, if a microservice goes down unexpectedly you only need to bring up the service that went down, reducing downtime and improving the resilience of your application. Lastly, a microservice architecture is technologically free. You don’t need to write every microservice in the same language and framework as you would in a traditional application architecture giving you the freedom to do what you want technologically. Microservices do come with some disadvantages. What it comes down to mostly is the time it takes to manage a microservice architecture. It requires a lot of effort to design good APIs that communicate clearly with other microservices. You have to think of appropriate error codes to give your consumers the best information as to what’s going wrong with the application. It also requires a lot of time to build and maintain the servers. You may have to write custom networking on your for communicating between each microservice, as well as lots of custom configuration like auto scaling, readiness and health checks as well as availability during deployments.

What is serverless?

Serverless is an execution model that allows developers to run code without managing infrastructure. With a serverless computing platform, all developers need to do is write application code and send it to the platform. The platform takes care of provisioning infrastructure to run the code. This usually results in lower infrastructure costs. Because the infrastructure is provisioned for you, the cloud provider can make use of multiple isolated applications running on the same server. Behind the scenes, this costs the cloud provider less because they can maximise the use of their servers instead of provisioning an entire server for one application, and in turn costs you less. Scaling your application is also much simpler when it comes to serverless. If your serverless application is getting a lot of traffic, the cloud provider handles scaling up the application for you automatically. In many cases, serverless computing can simplify your backend code. It would be a no-brainer to move your stateless functions to a serverless computing platform so that you’re not paying to host servers when they are not running those functions. There’s no need to define an API layer, a business logic layer or a database layer in your application for stateless functions when you can just throw them up into a serverless computing platform where they will live for you. Lastly, and possibly the most important benefit of serverless computing is the quicker turnaround time for creating an application. WIthout the requirement to spin up instances to run your code, you can get your application to the market much quicker than with traditional application architectures. Things like security, capacity and observability are all (usually) built into the serverless computing platform giving you more time to write your application over building the infrastructure for it. Nevertheless, serverless is not invulnerable to disadvantages. For example if one of your functions is not getting very much usage, that function is prone to cold starting. Some cloud providers will shutdown your serverless application if it has not been used in a while to prevent over-provisioning. This means that the next time you want to run your application, it may take some time initially because the cloud platform has to start the function back up again. Additionally, it can take some time to get stateful applications up and running on some cloud computing platforms. By its design, serverless functions are meant to be turned off when they are not being used. But in a stateful application, where does the state go? Unfortunately, the state is gone from the server, just like everything else. Stateful serverless applications have seen some improvements recently, but it can take some time to set it up on some computing platforms.

When to use serverless

Serverless can be thought of as a managed microservice architecture. With recent improvements to stateful applications on serverless computing platforms, there are less and less reasons for not choosing to run your application on a serverless computing platform. To demonstrate when to use serverless, I'm going to use the names of some computing platforms to give each example some more practicality. Lambda is a serverless computing platform owned by Amazon Web Services (AWS). Later, I will show you how you can run any microservice application using Lambda (and also why you may not want to), but for now we are going to talk about what Lambda excels at. Possibly the most proficient use of Lambda is for event driven services. Let's say you create a new record in your AWS DynamoDB managed NoSQL database when a user places an order in your online shop. You can use Lambda to trigger a serverless function that sends an email to that user confirming their order when a database record is created. But event driven services are not only limited to web servers. For example, you can trigger a Lambda function that sends a push notification to your phone when your AWS Internet of Things home security camera catches movement at your front door. You might begin to see how the use of serverless computing platforms for event driven services are endless. But Lambda’s proficiency does not end there. Serverless functions can be used for complicated proxies that don’t always need to be running. For example, you can use a serverless function to manage a pool of database connections that perform queries for other functions. There is no need to exhaust database connections when you can have one serverless function that acts as a single open connection, and is only turned on when you need it.

Let’s take a look at an example application using the microservice architecture, then I will show you how you can run the same application in AWS Lambda. For this example, we will use AWS’ Fargate with Elastic Container Service (ECS). Each container in ECS will represent one microservice, and each microservice can be scaled up infinitely based on the amount of traffic it’s getting. We will use the example of a ride sharing application. In this application, there are four microservices: the customer user interface, the customer service, the driver user interface and the driver service. Each user interface hosts a collection of Hyper Text Markup Language (HTML) pages that link to each other and make calls to the two service containers. Each service makes API calls related to their service’s single responsibility. For example, the customer user interface may make a call to the customer service that makes a call to the driver service which makes a notification appear on the driver user interface letting the driver know that the customer is requesting a ride. The driver then presses an accept button on the driver user interface, which makes a call to the driver service, which makes a call to the customer service and in turn displays the driver’s name, car and how far they are on the customer user interface. You would be required to write the code in each container that accepts these requests and the code that returns the required information based on the request. For example, the user interface's would return the appropriate HTML page and the service requests would return the appropriate HyperText Transfer Protocol (HTTP) status codes based on their results. Additionally, you would be required to configure how quick you want your containers to scale up (how many concurrent requests to a container are needed before you spin up another container and distribute the load for quicker response times) as well as your availability when deploying new containers with new features (do you need 100% of your required containers up all the time or can you risk having 25% of your containers up at the benefit of a quicker deployment?).

To implement the same application using AWS Lambda, we would need to create seven different functions. There are three customer interface functions, one for the customer user interface to request a driver, one for the driver user interface as they get a request for a ride, and another for the customer interface showing that a rider has accepted an order. Then we would need four functions for the service implementation, one for the customer interface to call when the customer has requested a driver, one for notifying our application that the customer is looking for a driver, one for notifying the driver that a customer is requesting a ride, and one for notifying the customer that the driver has accepted their ride request (and then calls the customer interface that shows the driver has accepted the ride. All you would need to do is write the code that calls these functions. You can see that there is no need to provision any infrastructure or configure routing, scaling or availability. You could imagine, though, that the more complex your application gets, the less that you would want to create hundreds of different functions, and instead group them into a microservice.

Overall, you should only choose Lambda if you're unconcerned about managing the infrastructure of your application. Lambda can be thought of as a managed microservice computing platform. If your application requires some sort of infrastructure configuration, Fargate with ECS or EKS offers these services. How do you know if your application requires some sort of infrastructure configuration? Generally, the more complex your application gets, the more that you’re going to want to control its infrastructure. It would be annoying if your application has various downstream API calls and you’re trying to figure out how to mitigate the speed of these calls without being able to manage cold starts for your services. It would also be difficult managing an application with hundreds of different Lambda functions that would normally be grouped together into one service under a microservice architecture.

Sources

Indirect

Amazon ECS on AWS Fargate - Amazon Elastic Container Service. (n.d.). Retrieved March 26, 2023, from https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html

Larsson, M. (2014). *Microservices*. Amazon. Retrieved March 26, 2023, from https://aws.amazon.com/microservices/

What is serverless computing? IBM. (n.d.). Retrieved March 14, 2023, from https://www.ibm.com/topics/serverless

What is serverless computing? | *Serverless definition* | *cloudflare*. (n.d.). Retrieved March 14, 2023, from https://www.cloudflare.com/learning/serverless/what-is-serverless/

YouTube. (2021, March 17). *Serverless Computing in 100 seconds*. YouTube. Retrieved March 26, 2023, from https://www.youtube.com/watch?v=W_VV2Fx32_Y