SERVERLESS COMPUTING

According to the definition provided by Red Hat, "Serverless is a cloud-native development model that allows developers to build and run applications without having to manage servers." It doesn't mean that there are no more servers behind our services, but their management is completely demanded to cloud providers. So, end users have only to concentrate on the business logic of their applications.

The main difference between serverless and traditional cloud computing is that the cloud provider is responsible for managing the scaling of the deployed apps, as well as maintaining the cloud infrastructure. Serverless apps are deployed in containers that automatically launch on demand when called. In this way, apps are launched only when needed, with the possibility of the so called "zero scaling": no computational resources are allocated if there are no requests.

Serverless encompasses two main models:

1- BaaS: initially, serverless referred to the usage of plug-and-play cloud services such as databases, authentication services and so on. These types of services are called Backend as a Service.
2- FaaS: serverless can also mean applications where server-side logic is still written by developers, but it's run in stateless compute containers that are event-triggered, ephemeral and fully managed by the providers. This model is called Function as a Service.

Function as a Service:

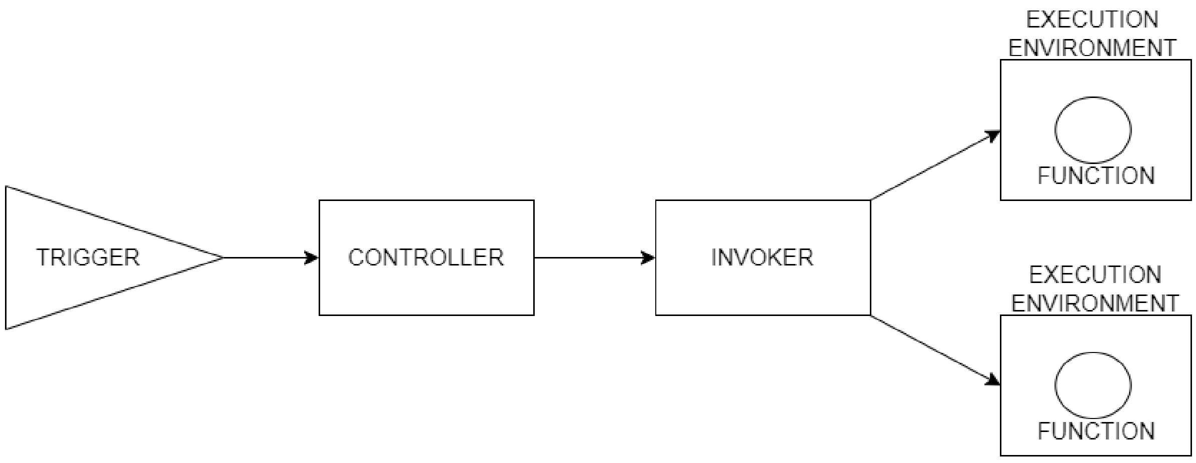
Fundamentally, FaaS is computational model based on events. "Functions" are triggered upon the arrival of predefined events, and they are run on ephemeral containers that are terminated at the end of the execution. For this reason, FaaS is designed mainly for stateless computation. The management of the horizontal scaling, containers' lifecycle and external resources is automatically provided by the infrastructure.

The general architecture of FaaS platforms includes at least the following components:

- Trigger: it creates a bridge between external events and those manageable by the FaaS infrastructure. Events can be generated even by sources defined by the provider and HTTP requests via API gateway.
- Controller: it associates functions with events received from the trigger. It manages the lifecycle of functions and other components of the FaaS infrastructure. Users provide

configurations (called workflows) specifying which functions to activate in response to certain events. The controller is responsible for triggering the workflows.

- Invoker: it receives the events from the controller and instantiates the functions with their execution environments to produce the correspondent output.



Benefits and use cases:

Given the characteristics previously described, it is clear that FaaS is mostly suitable for highly asynchronous and stateless scenarios, due to its event driven architecture and the short lifetime of functions. Many use cases include also situations in which there is an unbalanced and infrequent workload, since the scalability of the functions is completely managed by the provider and it's also possible to have no instances running. In fact, the greatest benefit of using FaaS platforms is that we only have to pay for the real time of execution of our functions, instead of paying for the hardware infrastructure even when our application is not running.

Drawbacks:

Despite the clear advantages brought by the adoption of FaaS model, there are some issues that are intrinsically related to its architecture and its paradigm, such as:

- Vendor lock-in: all serverless vendors implement their features differently from others, so, if you want to switch vendors, you will probably need to modify functions' code.
- Security: since serverless function consume input data from a variety of event sources, the attack surface considerably increases. Furthermore, our application can be composed by many functions, each one accompanied by its own configuration file. So, functions' configuration gains importance in order to limit the access points to the system.
- No in-server state: we can't assume that the local state from one invocation of a function will be available to another invocation of the same function. In theory, it is possible only if the same instance of the function is kept alive by subsequent requests

or events for as long as we need its state. This can be a problem if we need a reliable and fast way to access state. We are forced to use third party services with higher-latency access with respect to local caches and on-machine persistence.

Another class of drawbacks concerns the current state of the art of serverless computing. The previous ones are likely always going to exist, and they can only be mitigated. The following issues will probably be resolved in the near future:

- Execution duration: current FaaS platforms allow functions to run only for a certain predetermined period of time. Afterward, these functions are terminated even if they are still running. So, we have to predict their execution time and ensure that it is under the threshold.
- Cold starts: given that we don't have a server constantly running our code, the platform will need to execute the following steps when a function must be instantiated:
    1) Allocate an underlying VM resource to host the function
    2) Instantiate a Linux container that will run the code on the VM
    3) Copy the code to the container
    4) Start the language Runtime we specified
    5) Load the code
    6) Instantiate the function

  These steps reduce the throughput of our functions.
- Testing: it is quite simple to test a single function because it is in essence a piece of code that doesn't need to use specific libraries or to implement particular interfaces to work. Problems come when we want to perform integration tests on serverless applications. In fact, this kind of application usually leverages externally provided systems to achieve certain features such as persistence and access control. So, it isn't probably enough to perform tests in a local environment to ensure that apps work properly.

Best practices:

To overcome the drawbacks of adopting FaaS architecture, users' communities and the principal platform vendors have compiled a list of best practices to follow when developing functions. For example, AWS Lambda suggests to:

- Separate handler from core logic to make more unit-testable functions
- Use environment variables to pass operational parameters to the functions
- Write idempotent code to ensure that duplicate events are handled the same way
- Minimize the complexity of the dependencies needed by the functions

In particular, to achieve the last point of the previous list and to simplify the testing phase of our applications' lifecycle it can be useful to resort to the concept of "dependency injection".

We will discuss about it and how to implement it in a serverless scenario in the rest of this work, focusing on already existent solutions and open source FaaS platforms. First of all, we will discuss about what dependency injection is, how it works, and which benefits it can bring.

References:

https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless

https://martinfowler.com/articles/serverless.html

https://blog.symphonia.io/posts/2017-11-14_learning-lambda-part-8

https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html