Microservices Explained

Problem with Monolithic architecture

At the beginning of application development, the standard way of doing it was via a monolithic architecture. Monolithic meaning that all components of an application for example an online shopping platform would be part of a single code-base.

Component such as user-authentication, the shopping-cart, product-catalog, notification system, and payment system code will all be in a single code-base.

Cons

Every component is developed, deployed and scaled as 1 unit. This also means that the application can only be in one tech stack.

Team that works on different component of this monolithic application must be careful to not affect each other's work. In addition, when you are deploying this monolithic application the entire application must be redeployed you cannot just redeploy just one single component of the application.

If the application is large and complex, then different components will be more tangled into each other. If you want to scale then you must scale the entire applications, not only parts of components that are in demand = higher infrastructure cost.

Release process takes longer, the CI/CD pipeline will need to rebuild the entire application from the ground, component to component in order to redeploy.

Microservices

Now instead of writing one giant monolithic application you break up the application into smaller, and independent services.

How to break down the application? How many microservices to break it into? How big/small it should be? How do they communicate with each other after you split them up? These are all questions when you are developing microservices.

Microservice architecture

Well you split up the application based on business functionalities. It is something that a business or a company does in order to generate value. Business features that can stand on it's own without other.

Each microservice should be responsible for one service, one specific job. They should be self-contained and independent of each other. Which means that they can be developed, deployed, and scaled separately on their own without relying on another.

Communication between microservices

1. API calls

How do they talk to each other if they are separate applications?

The common way of doing communication is via API calls. Each of the microservices has it's own API endpoint, then the microservices can then just communicate to each other by sending HTTP request to each fo the endpoint.

This is for synchronous communication, you sent the request and expect a reply.

2. Message broker

This is for asynchronous communication, messages will sent the request to a message broker and that will be stored in a queue, then the broker will forward that message to the actual microservices.

You sent the request but the actual message can be stored in queue and without knowing exactly when you will receive a reply.

Monorepo vs Polyrepo

Now when you are writing microservices how do you store them on cloud repository? Like Github or Gitlab?

With Monolithic repository you can just slap that big application into one repo and call it a day. But whatever Microservices? There is two ways of storing it into a Git repo.

Monorepo

Having one git repository that contains many projects (the microservices).

You would have a directory for each service/project/microservice.

This makes code management and development easier, just need to clone and work only with one repo. Changes can be tracked together, tested together.

But, having these project together is might result in tight coupling of projects. All these projects can result in large repository in the end.

Polyrepo

One microservice per one repository. Own CI/CD per repository.

You can group those microservice repository together under a group for easier management.

Switching between project is tedious. Sharing resources is more difficult.

Both have advantages and disadvantages.

Revision #1
Created 2023-02-04 19:58:29 UTC by Tamarine
Updated 2023-02-04 21:03:36 UTC by Tamarine