Microservicios

Prólogo

Hace tiempo, las arquitecturas monolíticas eran el estándar a la hora de desarrollar software. En pocas palabras, todo el código que necesitaba un servicio web estaba en la misma unidad y todos tenían que trabajar sobre esta estructura en el caso de haber un equipo de desarrollo, por lo cual una comunicación podría significar la caída total de todo el proyecto.

¿Qué son los microservicios?

Los microservicios surgen como una respuesta al problema de trabajar con una misma unidad; en vez de eso, la arquitectura de microservicios separa todas las funcionalidades que no tienen una relación muy estrecha en servicios más pequeños e independientes.

¿Qué distingue a cada microservicio?

Es natural preguntarse cuáles son los límites de cada microservicio cuando en realidad todo es parte de una misma aplicación. Para contestar esta pregunta, cada microservicio debe:

- estar separado acorde a las funcionalidades del negocio. Por ejemplo, algunas funcionalidades de una tienda en línea pueden ser los productos, el carrito de compras, todo lo relacionado al usuario, etcétera.
- estar pensado para cumplir un trabajo específico. No vas a meter una función para agregar al carrito de compras directamente en el mismo lugar donde el usuario puede cambiar sus datos personales.
- ser autocontenido e independiente. Cada microservicio puede estar implementado en distintos lenguajes de programación, por lo cual pueden haber equipos de desarrollo que sólo se enfoquen al desarrollo y mantenimiento de este microservicio.

¿Cómo se comunican entre sí los microservicios?

Ya que tenemos la idea de que cada microservicio es visto como una entidad independiente, ahora la pregunta que se nos viene a la mente es "*¿cómo se comunican entre sí?"*, lo cual puede lograrse de diferentes formas:

1. Por llamadas a la API: cada microservicio cuenta con su propia Interfaz de Programación de Aplicaciones (API), por lo cual se comunican usando llamadas a la API, es decir, se mandan *HTTP Request* para obtener y enviar información al endpoint de la API con la que se quieren comunicar. Esta forma de comunicación es síncrona.
2. *Message Broker*: cada microservicio mandan mensajes a un intermediario (*broker*), el cuál reenvía al servicio correspondiente. Esta manera de comunicación es asíncrona.
3. *Service Mesh*: imaginemos que ahora existe una especie de servicio auxiliar que se hace cargo de toda la lógica de comunicación sin que no nos tengamos que preocupar por la implementación de la comunicación. Es muy popular en el campo de los Kubernetes.

Desventajas de los microservicios

Dado a que se trata de un sistema distribuido, se tienen ciertas dificultades a la hora de configurar la comunicación entre servicios; es decir, si se cae un microservicio y otro se está comunicando con él, no sabrá que está caído. Además, puede complicarse monitorear cada instancia de los servicios entre servidores, lo cual hace más difícil darnos cuenta cuándo un servicio está fallando.

Kubernetes es una plataforma con la cual podemos ejecutar aplicaciones grandes con muchos microservicios y nos ayudará para monitorearlos y nos sea más fácil manejar cualquier error que surja.

Pipeline de Integración y distribución continua para microservicios (CI/CD Pipeline)

Muchas aplicaciones populares, como Google o Amazon, utilizan cientos de microservicios en cada momento. El cómo se manejan estas grandes aplicaciones, considerando que existen diversos microservicios, es una pregunta que se podemos responder con los repositorios git.

De manera general se consideran dos categorías, *monorepo* (un solo repositorio) y *polyrepo* (múltiples repositorios).

1. *Monorepo*: se crea un repositorio que contiene varios proyectos. Los distintos proyectos se alojan en distintas carpetas. El proyecto completo comparte el mismo *pipeline*.

   Ventajas:

   - Hace más fácil el desarrollo y mantenimiento del código.
   - Se puede clonar y trabajar sobre este repositorio sin utilizar otro más.
   - Los cambios pueden ser rastreados, probados y lanzados juntos.

   Desventajas:

   - Los diferentes proyectos pueden llegar a ser dependientes entre sí.
   - Es más fácil que un error cometido en un proyecto diferente a otro rompa la aplicación y que ésto afecte a otros equipos.
   - Cuando el proyecto crece, se vuelve más lenta la interacción con el repositorio al querer clonarlo, hacerle `push` o `pull.`

2. *Polyrepo*: se crea un repositorio por cada servicio. Esto implica que el código de cada servicio está aislado y cada uno se trabaja de forma separada. La manera con la cuál se conecta cada repositorio puede ser con herramientas que ofrecen las mimas plataformas de desarrollo colaborativo (como GitHub o GitLab). Cada repositorio tiene su propio *pipeline*.

   Como desventajas, los cambios transversales son más difíciles de hacer. Cada cambio que se esparza a través de los servicios deben ser subidos como *merge requests* separados en vez de sólo tener que hacer uno, además, buscar, probar y hacer *debugging* es más complicado, como lo es también compartir recursos.