



Microservicios (o cómo diseñar aplicaciones modernas)

Backend



Qualentum Lab

Microservicios (o cómo diseñar aplicaciones modernas)

En este fastbook vamos a ver la arquitectura de microservicios, sus ventajas y cómo nos puede ayudar a la hora de diseñar aplicaciones modernas con cierta escala. Usaremos una plataforma LMS (*Learn Machine System*) como referencia a lo largo de él, para que puedas ver la aplicabilidad de los distintos conceptos.

¡Vamos allá!

Autor: Antonio Blanco Oliva

-  **Introducción a los microservicios**
-  **Diseño práctico de aplicaciones en un modelo basado en los microservicios**
-  **API**
-  **Modelo de datos**
-  **Patrones y estrategias avanzadas en microservicios**
-  **Tendencias y futuro**
-  **Glosario**

Introducción a los microservicios



La arquitectura de microservicios es un enfoque para que una aplicación se componga de pequeños servicios independientes que se ejecuten en sus propios procesos y se comuniquen con mecanismos ligeros, habitualmente en una API HTTP.

Cada servicio se centra en una única función de negocio y se desarrolla, despliega y escala de manera independiente.

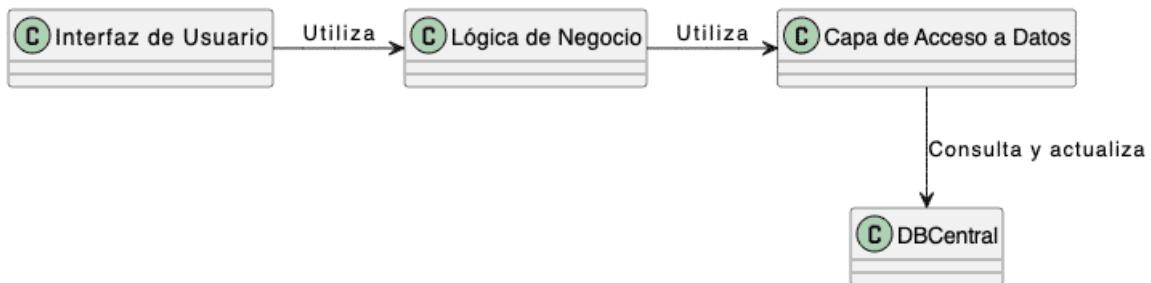
En contraste, en un sistema monolítico, la aplicación se construye como una sola unidad indivisible que engloba todas las funciones de negocio. Esta unidad única se desarrolla, despliega y escala como un solo bloque, lo que puede conllevar complejidades significativas a medida que la aplicación crece y evoluciona.

Por lo tanto, la dependencia entre los diferentes componentes de la aplicación es alta, y las actualizaciones o cambios requieren que el sistema completo sea desplegado nuevamente, en lugar de actualizar servicios individuales como en el caso de los microservicios.

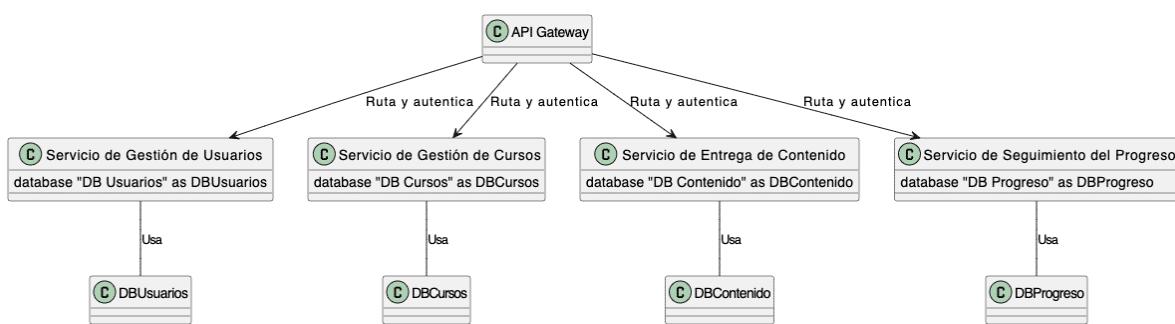
Nuestro LMS

En el contexto de una plataforma de enseñanza LMS, aplicar microservicios significa desarrollar determinadas funcionalidades: la gestión de usuarios, la administración de cursos y el seguimiento del progreso, como servicios separados. Por ejemplo, un microservicio puede manejar las inscripciones y los perfiles de usuario, mientras que otro podría gestionar la lógica de los cuestionarios y exámenes. **Esta separación permite que distintos equipos trabajen en diferentes componentes simultáneamente**, aumenta la agilidad y facilita la escalabilidad y el mantenimiento del sistema.

Por lo tanto, pasaríamos de este modelo que mostramos en la imagen:



A uno similar a este:



Diseño práctico de aplicaciones en un modelo basado en los microservicios



Qualentum Lab

El proceso de diseño de la aplicación, que recoge desde un problema o idea a una posible solución, es una de las fases más importantes en el desarrollo software. Si a eso le añadimos que la aplicación no la trataremos como una caja negra gigante, sino que vamos a ir un paso más allá, dividiéndola en pequeñas cajas negras con funcionalidades o alcances ya definidos, es primordial trabajar a fondo en esta primera fase del proceso.

¿Y qué tareas debemos contemplar en esta fase primaria? Vamos a explicarlas, a continuación, apoyándonos en el ejemplo del LMS.

Análisis de requerimientos

El análisis de requerimientos en la arquitectura de microservicios se enfoca en entender las necesidades del negocio y **cómo pueden ser satisfechas a través de servicios independientes y autónomos**. Este proceso incluye la identificación de funcionalidades clave, la definición de interfaces de servicio, y la planificación de la interacción entre servicios. Es crucial para definir los límites de cada microservicio y para asegurar que el diseño global del sistema sea coherente y escalable.

Nuestro LMS

En nuestro LMS, el análisis de requerimientos implica mapear las funcionalidades educativas y administrativas en servicios separados, por ejemplo:

- **Gestión de usuarios:** definir cómo el servicio manejará la autenticación, registro y perfiles de usuarios.
- **Administración de cursos:** establecer los requerimientos para la creación, actualización y eliminación de cursos, así como la asignación de instructores y estudiantes.
- **Gestión de contenidos:** determinar el manejo de material educativo, incluyendo el almacenamiento, la recuperación y la presentación de diferentes tipos de medios.
- **Evaluación y seguimiento:** identificar cómo se rastreará el progreso del alumno, cómo se administrarán los exámenes y cómo se reportarán los resultados.

Cada uno de estos aspectos debe ser analizado para crear un conjunto de microservicios que trabajen juntos sin problemas, garantizando que las necesidades de todos los usuarios del LMS se satisfagan eficientemente.

Diseño de servicios

El diseño de servicios en la arquitectura de microservicios se enfoca en la creación de servicios autónomos con interfaces claramente definidas. **Cada microservicio encapsula una funcionalidad de negocio específica y expone una API para interactuar con otros servicios.** Se deben considerar varios principios fundamentales como la separación de responsabilidades, la granularidad adecuada de servicios y la gestión de datos descentralizada.

Nuestro LMS

El diseño de servicios podría incluir los siguientes microservicios autónomos:

- **Servicio de autenticación:** un servicio dedicado a la autenticación y autorización de usuarios.
- **Servicio de gestión de cursos:** el que permite a los profesores o instructores configurar y gestionar sus cursos, incluyendo la estructura del curso, los materiales y las asignaciones.
- **Servicio de contenido multimedia:** un servicio diseñado para almacenar, catalogar y servir contenido educativo como vídeos, documentos y presentaciones.
- **Servicio de evaluación y calificaciones:** gestiona la lógica para los exámenes, el seguimiento del progreso del alumno y el otorgamiento de certificados.

Cada servicio, por tanto, debe ser diseñado para operar de manera independiente, permitiendo que los cambios y mejoras se realicen con la mínima interrupción del resto del sistema LMS.

Comunicación entre servicios

La comunicación entre servicios es un aspecto crucial en la implementación de microservicios.

-  Los métodos síncronos, como las solicitudes HTTP/REST o [gRPC](#), son bloqueantes y esperan una respuesta inmediata.

Los métodos asíncronos, como las colas de mensajes y los eventos, **permiten una comunicación no bloqueante y mejoran la resiliencia y escalabilidad**. La consistencia de los datos se mantiene a través de patrones como bases de datos distribuidas, sagas o eventos de dominio (dispones de la definición de estos conceptos en el glosario, al final del fastbook).

Nuestro LMS

Entre las distintas conexiones de los microservicios de nuestra aplicación de enseñanza, la comunicación entre el servicio de gestión de cursos y el de seguimiento del progreso son vitales. Por ejemplo: cuando un curso se completa, el servicio de gestión de cursos envía un evento de dominio que el servicio de seguimiento del progreso escucha y actúa en consecuencia, actualizando el estado de progreso del estudiante.

¿Y sabías que un API Gateway puede servir como intermediario?

Encamina las solicitudes y respuestas entre estos servicios, proporcionando un punto de acceso unificado para los clientes frontend. Dispones de más detalles del API Gateway en el glosario de términos, al final del fastbook.

Por último, recuerda que **cada interacción debe ser diseñada para asegurar la integridad de los datos y proporcionar** una experiencia de usuario fluida y consistente en el LMS.

API



Qualentum Lab

Las APIs (interfaces de programación de aplicaciones) son fundamentales en la arquitectura de microservicios, ya que actúan como el principal medio de comunicación e interacción entre los servicios independientes.

En un entorno de microservicios, cada servicio **expone su funcionalidad a través de una API**, lo que permite a otros servicios y aplicaciones interactuar con él sin necesidad de conocer los detalles internos de su implementación.

Definir los distintos puntos de conexión de las APIs, los endpoints, es algo que no se debe hacer jamás deprisa y corriendo, ya que definirá el idioma en el que los microservicios hablarán entre ellos.

En la implementación de APIs en una arquitectura de microservicios, existen varias estrategias de implementación, ¡vamos a verlas!

API Gateway

Habrá un único punto de entrada para todas las llamadas de clientes externos. El API Gateway puede manejar tareas como el enruteamiento, la autenticación y la agregación de respuestas de varios microservicios.

APIs basadas en contratos

En otras palabras: establecer contratos claros para las APIs usando especificaciones como [OpenAPI](#). Esto asegura una integración consistente y predecible entre servicios, sin llevar a ambigüedades y que todos los equipos sepan exactamente qué se espera en cada comunicación.

A continuación, también te comarto algunas prácticas que debemos seguir:

APIs versionadas

Debemos mantener versiones de las APIs para evitar interrupciones en los servicios consumidores cuando se realizan cambios en los microservicios. Normalmente nos encontramos con URLs de tipo: /v1/... o /v2/... Esto permitirá tener varias versiones activas a la vez y que los servicios que usen la versión anterior puedan ir adaptándose al cambio sin perder la comunicación.

Patrones de diseño RESTful o GraphQL

Hay que elegir entre REST, que utiliza HTTP para realizar operaciones CRUD, y GraphQL para consultas más eficientes y personalizadas (dispones de más información en el glosario de términos).

Seguridad de las APIs

Es necesario implementar medidas como la autenticación, la autorización y los límites de tasa para proteger las APIs y los datos que manejan. La limitación de la tasa consiste en establecer qué número de veces puede acceder un usuario a una API.

Monitoreo y logging

También es importante llevar un control y supervisión del uso de las APIs y mantener registros detallados para la depuración y la optimización del rendimiento.



Los servicios como [Swagger](#) nos facilitan la definición y compartición de APIs entre equipos.

Nuestro LMS

Ahora vamos a ver para nuestra aplicación LMS con microservicios, algunos ejemplos de posibles *endpoints* y sus contratos.

Servicio de gestión de usuarios

- Endpoint: /usuarios/registro
 - **Método:** POST.
 - **Contrato:** recibe datos de registro de usuario (nombre, correo electrónico y contraseña) y crea una nueva cuenta de usuario.
- Endpoint: /usuarios/login
 - **Método:** POST.
 - **Contrato:** recibe credenciales de usuario y retorna un token de autenticación si las credenciales son válidas.

Servicio de gestión de cursos

- Endpoint: /cursos
 - **Método:** GET.
 - **Contrato:** retorna una lista de todos los cursos disponibles.
- Endpoint: /cursos/{idCurso}
 - **Método:** GET.
 - **Contrato:** retorna detalles de un curso específico basado en el ID proporcionado.

Servicio de entrega de contenido

- Endpoint: /contenido/{idCurso}
 - **Método:** GET.
 - **Contrato:** retorna el contenido del curso, como videos y documentos, para el ID de curso proporcionado.

Servicio de seguimiento del progreso

- Endpoint: /progreso/{idUsuario}
 - **Método:** GET.
 - **Contrato:** retorna el progreso de aprendizaje del usuario basado en su ID.

Modelo de datos



En una arquitectura de microservicios, el manejo del modelo de datos puede presentar **desafíos**, especialmente en términos de consistencia y sincronización entre servicios. Aquí te comarto algunas soluciones habituales:

1

Bases de datos por servicio: cada microservicio tiene su propia base de datos o esquema, lo que asegura la independencia y la descentralización de los datos.

2

Patrón saga: para operaciones que abarcan múltiples servicios, se utiliza el patrón saga, una secuencia de transacciones locales en cada servicio con mecanismos para manejar fallas y garantizar la consistencia eventual.

3

Event Sourcing y CQRS: estos patrones permiten que los servicios manejen y almacenen datos de manera más eficiente, facilitando la separación entre las operaciones de lectura y escritura y manteniendo un registro completo de cambios en el estado de los datos.

4

APIs para la integración de datos: es recomendable utilizar APIs para compartir datos entre servicios de manera controlada y consistente.

Patrones y estrategias avanzadas en microservicios



Qualentum Lab

Circuit Breaker

El patrón de Circuit Breaker es una estrategia de manejo de fallos que previene que un alto número de llamadas fallidas a un servicio pueda degradar la red completa.

Cuando se detecta un umbral de fallas, el circuito se abre y corta las llamadas al servicio problemático, permitiendo que este se recupere y evitando que el problema se propague.

Nuestro LMS

En un LMS, el servicio de entrega de contenido puede ser **particularmente vulnerable durante períodos de alta demanda**, como durante los exámenes. Implementar un Circuit Breaker aquí puede detectar si el servicio está casi saturado o ya saturado, y temporalmente desviar o rechazar tráfico, protegiendo así el sistema de una interrupción mayor y manteniendo una experiencia de usuario estable para otras funciones del LMS.

Un pseudocódigo podría ser este:

```
class CircuitBreaker:
    estado = CERRADO
    umbral_fallos = 5
    tiempo_recuperacion = 60000 // tiempo en milisegundos
    contador_fallos = 0
    ultimo_fallo = null

    // Función para intentar llamar al servicio
    function llamar_servicio(solicitud):
        si estado == ABIERTO:
            si tiempo_actual - ultimo_fallo >
            tiempo_recuperacion:
                estado = MEDIO_ABIERTO
                // Llama para comprobar si el servicio se ha
                recuperado
            sino:
                lanzar ExcepcionServicioNoDisponible

        intentar:
            respuesta = servicio_llamada(solicitud)
            reiniciar() // Reinicia el contador de fallos si la
            llamada fue exitosa
            devolver respuesta
        excepto ExcepcionLlamadaServicio:
            registrar_fallo() // Registra el fallo e incrementa
            el contador
            lanzar

    // Función para registrar un fallo y actualizar el estado
    del circuit breaker
    function registrar_fallo():
        contador_fallos += 1
        ultimo_fallo = tiempo_actual
        si contador_fallos >= umbral_fallos:
            estado = ABIERTO

    // Función para reiniciar el circuit breaker
    function reiniciar():
        contador_fallos = 0
        estado = CERRADO

// Usar el RompeCircuito para una solicitud
circuitBreaker = CircuitBreaker()
circuitBreaker.llamar_servicio(solicitud)
```

Service Discovery

El Service Discovery es el proceso automático mediante el cual los servicios, en una arquitectura de microservicios, se encuentran y se comunican entre sí.

Es fundamental para el funcionamiento en entornos dinámicos, especialmente en la nube, donde las instancias de servicios pueden cambiar frecuentemente de dirección IP o puerto. El Service Discovery puede ser implementado por un servicio dedicado dentro de la arquitectura de microservicios, a menudo conocido como **Service Registry**.

Los clientes u otros microservicios consultan el Service Registry para descubrir las direcciones de los servicios que necesitan utilizar.

En algunos casos, como en Kubernetes, el propio sistema de orquestación de contenedores puede proporcionar capacidades de Service Discovery sin la necesidad de un nodo adicional específico.

Nuestro LMS

En un LMS, el Service Discovery permite a los servicios de reportes y análisis **registrarse en el sistema para que otros microservicios**, como el de gestión de cursos o el de seguimiento del progreso, puedan localizarlos y solicitar los datos necesarios para generar informes detallados del aprendizaje y la participación de los usuarios.

Balanceadores de carga

Los balanceadores de carga distribuyen el tráfico entrante entre múltiples instancias de un servicio para optimizar el uso de recursos, maximizar el rendimiento, reducir la latencia y garantizar la alta disponibilidad.

En entornos de microservicios, son esenciales para manejar eficientemente las solicitudes a servicios potencialmente desplegados en múltiples nodos o contenedores.

Nuestro LMS

En un LMS como el nuestro, los balanceadores de carga podrían utilizarse para distribuir las solicitudes entre **diferentes instancias del servicio de gestión** de cursos durante períodos de alta demanda, como el inicio de un nuevo semestre o durante los exámenes finales. Esto garantiza que todos los usuarios tengan acceso consistente y eficiente al sistema, incluso bajo una carga pesada.

Tendencias y futuro



Qualentum Lab

El futuro de los microservicios podría incluir una mayor adopción de tecnologías *serverless* (dispones de más información en el glosario), donde los servicios son aún más granulares y administrados por proveedores de la nube.

Se espera que la inteligencia artificial (IA) y el machine learning (ML) desempeñen un papel crucial en la optimización de la operación y el mantenimiento de los microservicios de las siguientes maneras:

En la automatización de operaciones

Utilizando IA y ML para automatizar tareas rutinarias de operaciones, como el escalado automático de servicios basado en el tráfico en tiempo real.

En la detección de anomalías y prevención de fallos

Implementando sistemas de IA para monitorizar constantemente la salud de los microservicios, detectar comportamientos anómalos y prevenir fallos potenciales antes de que afecten a los usuarios.

De cara a la optimización del rendimiento

Aplicando técnicas de ML para analizar patrones de uso y optimizar la distribución de recursos y el balanceo de carga de manera más eficiente.

Y en cuanto a mejoras en la seguridad

Utilizando IA para mejorar la seguridad de los microservicios mediante la detección proactiva de amenazas y vulnerabilidades.

Además, la adopción de estándares de comunicación como gRPC y GraphQL probablemente continuará creciendo y ofreciendo alternativas más eficientes al tradicional REST.

Glosario



- **Bases de datos distribuidas:** sistemas de bases de datos en los que los datos se almacenan en múltiples ubicaciones físicas. Proporcionan redundancia y pueden mejorar la disponibilidad y la escalabilidad, así como dividir los datos según dominios de negocio. Permitirán pasar de un modelo con una base de datos central, con toda la información, a un modelo con múltiples bases de datos cada una con una porción de la información.
- **Sagas:** secuencia de transacciones interrelacionadas que se ejecutan de forma distribuida entre diferentes microservicios y que juntas logran un objetivo de negocio. Si alguna de las transacciones falla, se ejecutan operaciones compensatorias para revertir el impacto. Todas actúan como una unidad indivisible.
- **Eventos de dominio:** mensajes o notificaciones que indican que algo importante ha sucedido en el dominio de la aplicación. Son utilizados para desencadenar procesos o para la comunicación entre microservicios.
- **API Gateway:** servidor que actúa como puerta de entrada única para las solicitudes externas a la arquitectura de microservicios, encaminándolas al servicio interno correspondiente. Actúa como un proxy, recibiendo peticiones y repartiéndolas al microservicio correspondiente. Tener a este elemento intermedio, nos permitirá, por ejemplo: añadir una capa de seguridad, chequeando si es una petición segura, o tener control sobre el número de llamadas, evitando sobrecargas o ataques a la red.
- **Endpoint:** punto final de una ruta de comunicación en una red donde se envían y reciben los mensajes. En el contexto de una API, se refiere a una URL específica donde se pueden realizar operaciones de API.

- **REST (Representational State Transfer)**: conjunto de principios de diseño de arquitectura para sistemas distribuidos, en particular para APIs web, que utilizan métodos HTTP estandarizados. Las operaciones en una API REST se basan en las acciones definidas por los métodos HTTP, como *GET* para recuperar recursos, *POST* para crear nuevos recursos, *PUT* o *PATCH* para actualizar recursos existentes y *DELETE* para eliminarlos.
- **Operaciones CRUD**: acrónimo de crear, leer, actualizar y borrar (*Create, Read, Update, Delete*), que son las cuatro operaciones básicas de almacenamiento persistente.
- **GraphQL**: lenguaje de consulta y un entorno de ejecución para APIs que ofrece una alternativa más eficiente y poderosa a las API REST. Fue desarrollado por Facebook y liberado en 2015. Permite a los clientes definir exactamente qué datos necesitan, lo que puede reducir el número de solicitudes y la cantidad de datos transmitidos entre cliente y servidor. A diferencia de las API REST, que tienen múltiples endpoints, GraphQL dispone de un único endpoint que maneja todas las solicitudes, interpretando las consultas y retornando la información en la forma especificada por el cliente.
- **gRPC**: framework de llamada a procedimiento remoto (RPC) desarrollado por Google. Utiliza HTTP/2 para la comunicación, ofrece ventajas como múltiples llamadas simultáneas sobre un solo TCP connection. Se le conoce por su eficiencia y rendimiento, y es ampliamente utilizado para conectar microservicios, sistemas distribuidos y aplicaciones móviles con servicios backend.
- **Serverless**: modelo de computación en la nube que permite a los desarrolladores construir y ejecutar aplicaciones y servicios sin tener que gestionar la infraestructura subyacente. Los recursos y servidores no necesitan estar corriendo constantemente, sino que se activan dinámicamente según sea necesario, optimizando el uso de recursos y reduciendo costos.

¡Enhorabuena! Fastbook superado



Qualentum.com