Imagina

Sesión 4

David Martínez Sepúlveda

Imagina

Gestión de Errores y fallas en la comunicación

David Martínez Sepúlveda

Gestión de Errores y Fallas en la comunicación

- La comunicación entre servicios es crítica y puede ser propensa a fallos.
- Problemas de red, sobrecarga de servicios, tiempos de espera largos o incluso errores internos en los servicios.

Patrones de resiliencia

Circuit Breaker

- Evita que un servicio continúe llamando a otro servicio que ya ha fallado repetidamente.
- Si un servicio detecta múltiples fallos en un determinado periodo de tiempo, "abre" el circuito y bloquea nuevas solicitudes hacia el servicio problemático durante un tiempo.

Patrones de resiliencia

Circuit Breaker

- Evita fallos en cascada y protege los servicios sanos del sistema.
- Mejora el tiempo de respuesta, ya que el sistema no intenta repetidamente acceder a un servicio fallido.

Patrones de resiliencia

Retry

- Reintenta una operación fallida varias veces antes de considerarla fallida permanentemente.
- Útil en escenarios donde los fallos pueden ser temporales, como una sobrecarga momentánea o un fallo de red transitorio.

Patrones de resiliencia

Retry

- Ayuda a mitigar fallos transitorios y mejora la robustez del sistema.
- Puede evitar que un fallo temporal se convierta en un fallo crítico.

Patrones de resiliencia

Timeout

- Asegura que un servicio no espere indefinidamente una respuesta de otro servicio.
- Si el servicio no responde dentro de un tiempo límite predefinido, la solicitud se interrumpe, evitando que el sistema se bloquee por solicitudes pendientes.

Patrones de resiliencia

Timeout

- Evita que las solicitudes queden colgadas, lo que mejora el tiempo de respuesta general del sistema.
- Protege al sistema de sobrecargas si un servicio está respondiendo demasiado lentamente.

Mecanismos de recuperación ante fallos

Cuando un error ocurre, es importante que el sistema pueda **recuperarse de manera eficiente** y evitar que el fallo se propague o cause más problemas.

Mecanismos de recuperación ante fallos

Fallbacks

- Alternativas que un microservicio puede ejecutar si no se puede completar la operación principal.
- En lugar de fallar completamente, el servicio ofrece una respuesta predeterminada o ejecuta una operación alternativa para mantener el sistema en funcionamiento.

Mecanismos de recuperación ante fallos

Fallbacks

- Mantiene la experiencia del usuario, incluso si el sistema está degradado.
- Proporciona una opción de respaldo para evitar la interrupción completa del servicio.

Mecanismos de recuperación ante fallos

Consistencia Eventual

- Permite que los sistemas distribuidos alcancen la consistencia con el tiempo, en lugar de requerir una consistencia inmediata.
- Útil en escenarios donde la sincronización perfecta de los datos no es crítica y se pueden tolerar pequeños retrasos en la actualización de los estados.

Mecanismos de recuperación ante fallos

Consistencia Eventual

- Mejora la escalabilidad y el rendimiento al evitar la necesidad de sincronización inmediata entre todos los microservicios.
- Adecuado para sistemas que no requieren datos completamente sincronizados en tiempo real.

Mecanismos de recuperación ante fallos

Consistencia Eventual

Ejemplo: Un sistema de inventario donde la cantidad de productos disponibles puede sincronizarse con un pequeño retraso entre el microservicio de inventario y los sistemas de ventas, permitiendo procesar más transacciones sin bloquearse.

Imagina

Monitoreo y Alertas de fallos

David Martínez Sepúlveda

Monitoreo y Alerta de Fallos

Monitoreo de Servicios (Prometheus y Grafana)

Herramientas como Prometheus y Grafana permiten monitorear métricas clave como **tiempos de respuesta, tasas de error y latencias de comunicación**. Estas métricas ayudan a detectar problemas antes de que se conviertan en fallos mayores.

Monitoreo y Alerta de Fallos

Monitoreo de Servicios (Prometheus y Grafana)

Alertmanager, integrado con Prometheus, permite configurar alertas proactivas para notificar a los equipos cuando los servicios están experimentando problemas. Las alertas pueden enviarse a plataformas como Slack, PagerDuty o correo electrónico.

Imagina

Implementación de Caché en Microservicios

David Martínez Sepúlveda

Implementación de Caché en Microservicios

La caché es un almacenamiento temporal de datos que permite acceder rápidamente a información que ya ha sido calculada o recuperada previamente.

Implementación de Caché en Microservicios

Tipos de caché

- **Caché local.** Se almacena directamente en la memoria de la instancia del microservicio.
- **Caché distribuida.** Se utiliza un sistema de caché externo y compartido, como **Redis** o **Memcached**, que permite que múltiples instancias de un microservicio accedan a los mismos datos en caché.

Implementación de Caché en Microservicios

Beneficios

- Reducción de latencia.
- Aliviar la carga en la base de datos.
- Mejor rendimiento y escalabilidad.
- Optimización de recursos.

Implementación de Caché en Microservicios

Estrategias

Caché-aside (Lazy Loading)

El microservicio verifica primero si los datos están en la caché. Si no están, los recupera de la fuente original (por ejemplo, una base de datos), los almacena en la caché, y luego los devuelve.

Implementación de Caché en Microservicios

Estrategias

Write-through cache

Cuando los datos son actualizados o insertados en la base de datos, también se actualizan inmediatamente en la caché.

Implementación de Caché en Microservicios

Estrategias

Write-behind cache

Las actualizaciones se almacenan primero en la caché y luego se escriben en la base de datos de forma asíncrona. Esto mejora el rendimiento de escritura, pero puede resultar en inconsistencias si la caché no se sincroniza correctamente con la base de datos.

Implementación de Caché en Microservicios

Estrategias

Time-to-Live (TTL) o Expiración de caché

Configura un TTL para los datos en caché, lo que asegura que después de un cierto periodo de tiempo, los datos sean eliminados o actualizados automáticamente.

Imagina

Patrones de diseño de Microservicios

David Martínez Sepúlveda

Patrón de diseño de agregados

El patrón de agregados agrupa entidades relacionadas y garantiza que las transacciones y modificaciones solo ocurran dentro de los límites del agregado.

Patrón de diseño de agregados

Un **agregado** es un *grupo de objetos relacionados que se tratan como una única unidad de datos*. Dentro de este conjunto de objetos, existe una entidad principal llamada **Entidad raíz** del agregado que gestiona el acceso a las demás entidades.

Patrón de diseño de agregados

Transacciones dentro de un agregado

Un aspecto clave del patrón de agregados es garantizar que todas las operaciones que ocurren dentro del agregado sean atómicas y consistentes.

Patrón de diseño de agregados

Transacciones dentro de un agregado

Atomicidad en el agregado. Todas las operaciones que modifiquen entidades dentro del agregado deben ser tratadas como una transacción única. Esto significa que o todas las modificaciones se aplican, o ninguna lo hace.

Patrón de diseño de agregados

Transacciones dentro de un agregado

Reglas de negocio. Las reglas de negocio complejas, como la validación de un estado del pedido o la verificación de inventario, deben manejarse dentro del agregado y no deben depender de agregados externos.

Patrón de diseño de agregados

Transacciones dentro de un agregado

Consistencia eventual. Los cambios en un agregado pueden ser aplicados de forma inmediata en su contexto local, y otros agregados pueden ser informados de esos cambios de manera asíncrona, permitiendo que el sistema llegue a un estado consistente después de cierto tiempo.

Patrón de diseño de agregados

```
public class Pedido {
    private String id;
    private List<LineaPedido> lineasPedido;
    private DireccionEnvio direccionEnvio;
    private MetodoPago metodoPago;

    // Constructor
    public Pedido(List<LineaPedido> lineasPedido, DireccionEnvio direccionEnvio, MetodoPago
metodoPago) {
        this.lineasPedido = lineasPedido;
        this.direccionEnvio = direccionEnvio;
        this.metodoPago = metodoPago;
    }
```

Patrón de diseño de agregados

```
    // Métodos para modificar el pedido solo a través de la entidad raíz
    public void agregarLineaPedido(Producto producto, int cantidad) {
        lineasPedido.add(new LineaPedido(producto, cantidad));
    }

    public void actualizarDireccionEnvio(DireccionEnvio nuevaDireccion) {
        this.direccionEnvio = nuevaDireccion;
    }

    // Otros métodos de negocio
    public double calcularTotal() {
        return lineasPedido.stream()
            .mapToDouble(LineaPedido::calcularSubtotal)
            .sum();
    }
}
```

Patrón de diseño de agregados

Beneficios

- Control de la consistencia
- Manejo de complejidad
- Desacoplamiento entre microservicios

Patrón de diseño de agregados

Desafíos

- Tamaño de los agregados
- Transacciones distribuidas.
- Diseño correcto de límites.

Patrón de diseño de eventos y mensajes

Este patrón permite que los microservicios estén más desacoplados, ya que no dependen de respuestas inmediatas entre sí, mejorando la escalabilidad y la resiliencia.

Patrón de diseño de eventos y mensajes

- **Emisor y receptor desacoplados.** El servicio que emite un evento no necesita conocer qué servicios lo consumen.
- **Flexibilidad en la integración.** Al emitir eventos en lugar de hacer llamadas directas, se pueden añadir nuevos microservicios que reaccionen a esos eventos sin necesidad de modificar el servicio que los genera.

Patrón de diseño de eventos y mensajes

El patrón de eventos se integra bien con otros patrones arquitectónicos como Event Sourcing y CQRS.

Patrón de diseño de eventos y mensajes

Event Sourcing

En lugar de almacenar el estado actual de los datos, Event Sourcing almacena una secuencia de eventos que representan todos los cambios de estado que han ocurrido en el sistema. Esto permite reconstruir el estado actual del sistema a partir de los eventos pasados.

Patrón de diseño de eventos y mensajes

CQRS (Command Query Responsibility Segregation)

El patrón CQRS separa las operaciones de lectura y escritura del sistema. Los eventos juegan un papel clave aquí, ya que los comandos pueden desencadenar eventos que actualizan el estado de lectura y escritura de manera separada.

Patrón de diseño de eventos y mensajes

CQRS (Command Query Responsibility Segregation)

El patrón CQRS separa las operaciones de lectura y escritura del sistema. Los eventos juegan un papel clave aquí, ya que los comandos pueden desencadenar eventos que actualizan el estado de lectura y escritura de manera separada.

Patrón de diseño de eventos y mensajes

Beneficios

- Desacoplamiento fuerte.
- Escalabilidad.
- Tolerancia a fallos.

Patrón de diseño de eventos y mensajes

Desafíos

- Consistencia eventual.
- Orden de los eventos.
- Duplicación de eventos.

Gateway y API Composition

En una arquitectura de microservicios, los servicios están diseñados para ser independientes y autónomos. Sin embargo, los clientes o consumidores del sistema a menudo necesitan interactuar con múltiples microservicios para obtener una respuesta completa.

Gateway y API Composition

Un API Gateway actúa como un punto de entrada único para todas las solicitudes de los clientes a un sistema de microservicios. Este patrón centraliza la lógica de manejo de solicitudes, de modo que los clientes no interactúan directamente con cada microservicio individual.

Gateway y API Composition

Características

- Autenticación y autorización.
- Enrutamiento inteligente.
- Transformación de solicitudes/respuestas.
- Caché.
- Monitoreo y logging.

Gateway y API Composition

API Composition

Cuando un cliente necesita datos de múltiples microservicios para completar una solicitud, el patrón de API Composition entra en juego. Este patrón se utiliza para combinar las respuestas de varios microservicios en una única respuesta que se envía de vuelta al cliente.

Gateway y API Composition

API Composition

Cuando un cliente necesita datos de múltiples microservicios para completar una solicitud, el patrón de API Composition entra en juego. Este patrón se utiliza para combinar las respuestas de varios microservicios en una única respuesta que se envía de vuelta al cliente.

Gateway y API Composition

Manejo de lógica compleja en el gateway

- Transformación de datos.
- Manejo de fallos.

Gateway y API Composition

Patrones avanzados en API Gateway

- Circuit Breaker en el Gateway.
- Rate Limiting.

Gateway y API Composition

Beneficios del patrón de gateway y API composition

- Simplificación para el cliente.
- Optimización de la red.
- Manejo centralizado de la seguridad y la autenticación.

Gateway y API Composition

Desafíos del patrón de gateway y API composition

- Punto único de fallo.
- Latencia adicional.


Imagina