Aula 03: Design de micro serviços

Introdução ao Design de Micro Serviços

O design de micro serviços envolve o planejamento e a organização de sistemas que são divididos em serviços menores e independentes, cada um executando uma função específica. Ao contrário da arquitetura monolítica, onde toda a lógica de negócios e funcionalidades estão centralizadas em uma única aplicação, a arquitetura de micro serviços adota uma abordagem modular.

Ao projetar micro serviços, é essencial pensar em como dividir a aplicação em unidades coesas que sejam suficientemente independentes para serem desenvolvidas, implantadas e escaladas de forma autônoma. No entanto, o design de micro serviços também precisa considerar a comunicação entre os serviços, o manuseio de dados distribuídos e como garantir a resiliência do sistema como um todo.

Abordagens de Design de Micro Serviços

1. Decomposição Baseada em Domínio (Domain-Driven Design)

O **Domain-Driven Design (DDD)** é uma abordagem amplamente utilizada no design de micro serviços. Ele foca em dividir a aplicação com base no domínio de negócios, onde cada serviço é modelado em torno de uma unidade de negócio, como "usuários", "produtos", "pagamentos", etc.

Componentes do DDD:

- **Entidade**: Um objeto de domínio com identidade e comportamento, por exemplo, "Usuário".
- **Agregado**: Um grupo de entidades relacionadas que representam uma unidade coesa, como "Pedido" e "Itens de Pedido".
- **Contexto Delimitado (Bounded Context)**: Define os limites do serviço em torno de um domínio de negócio específico.

Exemplo de Decomposição de Serviços Usando DDD:

Imagine um sistema de e-commerce. Podemos decompor esse sistema em:

- **Serviço de Usuários**: Gerencia o cadastro, login e dados do cliente.
- **Serviço de Produtos**: Gerencia o catálogo de produtos.
- **Serviço de Pedidos**: Gerencia o carrinho de compras e o processamento de pedidos.

Cada um desses serviços corresponde a um **contexto delimitado**.

```
// Serviço de Usuários em Node.js
const express = require('express');
const app = express();
let users = [{ id: 1, name: 'Alice' }, { id: 2, name: 'Bob'
}];

app.get('/users', (req, res) => {
    res.json(users);
});

app.listen(3001, () => {
    console.log('Micro serviço de Usuários rodando na porta
3001');
});

// Serviço de Produtos em Node.js
const express = require('express');
const app = express();
let products = [{ id: 1, name: 'Laptop' }, { id: 2, name:
'Phone' }];

app.get('/products', (req, res) => {
    res.json(products);
});

app.listen(3002, () => {
    console.log('Micro serviço de Produtos rodando na porta
```

```
3002');
});
```

Quando Usar DDD:

- Sistemas complexos com vários domínios de negócios.
- Quando há uma equipe dedicada a diferentes áreas do negócio (times de produtos, times de pagamento, etc.).

---

2. Decomposição Baseada em Casos de Uso

Nesta abordagem, os serviços são desenhados em torno de funcionalidades ou casos de uso específicos, em vez de domínios de negócios. Isso é útil em sistemas que exigem funcionalidades específicas que não estão necessariamente vinculadas a um único domínio.

Por exemplo, em um aplicativo de entrega de alimentos, podemos ter os seguintes serviços baseados em casos de uso:

- **Serviço de Rastreamento**: Fornece informações de localização em tempo real.
- **Serviço de Pagamentos**: Gerencia pagamentos e faturas.
- **Serviço de Recomendação**: Sugere restaurantes ou itens com base em preferências do usuário.

```
// Serviço de Rastreamento em Node.js
const express = require('express');
const app = express();

app.get('/tracking/:orderId', (req, res) => {
    const { orderId } = req.params;
    // Lógica de rastreamento fictícia
    res.json({ orderId, status: 'Entregador a caminho' });
});

app.listen(3003, () => {
    console.log('Micro serviço de Rastreamento rodando na p
```

```
orta 3003');
});
```

Vantagens da Decomposição Baseada em Casos de Uso:

- Foco em resolver problemas específicos.
- Fácil implementação para funcionalidades isoladas que não interagem muito com outros serviços.

Quando Usar:

- Para serviços que não se encaixam claramente em domínios de negócios.
- Funcionalidades altamente especializadas, como relatórios ou notificações.

---

3. Banco de Dados por Serviço

Um dos princípios fundamentais do design de micro serviços é que cada serviço deve ser o dono de seu próprio banco de dados. Isso significa que cada micro serviço mantém o controle sobre os dados que ele gerencia, e não há compartilhamento direto de um banco de dados centralizado entre serviços.

Vantagens:

- Autonomia total: Cada serviço pode escolher o tipo de banco de dados que melhor atende às suas necessidades (SQL, NoSQL, etc.).
- Independência entre serviços: Modificações nos esquemas de banco de dados não afetam outros serviços.

Exemplo:

No sistema de e-commerce, o serviço de "Usuários" pode usar um banco de dados relacional, enquanto o serviço de "Produtos" pode usar um banco de dados NoSQL.

```
// Exemplo de serviço com MongoDB (Produtos)
const express = require('express');
const mongoose = require('mongoose');
```

```
mongoose.connect('mongodb://localhost/products', { useNewUr
lParser: true, useUnifiedTopology: true });

const productSchema = new mongoose.Schema({
    name: String,
    price: Number,
});

const Product = mongoose.model('Product', productSchema);

const app = express();

app.get('/products', async (req, res) => {
    const products = await Product.find();
    res.json(products);
});

app.listen(3004, () => {
    console.log('Serviço de Produtos rodando com MongoDB');
});
```

Desafios:

- Garantir consistência entre dados distribuídos.
- Transações distribuídas podem ser mais complicadas de implementar (sistemas de mensageria como Kafka podem ser úteis).

---

4. Comunicação entre Micro Serviços

Uma parte crítica do design de micro serviços é a comunicação entre os serviços. Existem duas abordagens principais para isso:

- **Comunicação síncrona**: Via APIs REST ou gRPC.
- **Comunicação assíncrona**: Via filas de mensagens, como RabbitMQ ou Kafka.

Exemplo de Comunicação Síncrona (API REST):

Um serviço de "Pedidos" pode fazer uma chamada ao serviço de "Usuários" para obter informações sobre um cliente.

```
const axios = require('axios');

app.get('/order/:orderId', async (req, res) => {
    const userId = req.query.userId;

    try {
        const userResponse = await axios.get(`http://localh
ost:3001/users/${userId}`);
        res.json({ orderId: req.params.orderId, user: userR
esponse.data });
    } catch (error) {
        res.status(500).send('Erro ao buscar usuário');
    }
});
```

Exemplo de Comunicação Assíncrona (RabbitMQ):

A comunicação assíncrona é útil quando a resposta imediata não é necessária, como em sistemas de notificações.

```
// Enviando mensagem com RabbitMQ
const amqp = require('amqplib/callback_api');

amqp.connect('amqp://localhost', (err, conn) => {
    conn.createChannel((err, ch) => {
        const queue = 'notifications';
        const message = 'Pedido confirmado!';

        ch.assertQueue(queue, { durable: false });
        ch.sendToQueue(queue, Buffer.from(message));

        console.log('Mensagem enviada: %s', message);
    });
```

```
});
```

Quando Usar Comunicação Síncrona:

- Quando é necessário obter uma resposta imediata, como na validação de um pagamento.

Quando Usar Comunicação Assíncrona:

- Para eventos que podem ser processados posteriormente, como envio de e-mails ou notificações.

Metodologias de Design de Micro Serviços

1. Event-Driven Architecture (Arquitetura Orientada a Eventos)

Nessa metodologia, os serviços reagem a eventos gerados por outros serviços. Cada evento representa uma mudança de estado no sistema, e os serviços interessados no evento executam suas ações com base nisso.

Exemplo:

Um serviço de "Pedidos" pode gerar um evento "Pedido Confirmado", e o serviço de "Envios" pode agir sobre esse evento para iniciar o processo de entrega.

```
// Publicação de evento (Pedido Confirmado)
const amqp = require('amqplib/callback_api');
amqp.connect('amqp://localhost', (err, conn) => {
    conn.createChannel((err, ch) => {
        const queue = 'orderConfirmed';
        const message = 'Pedido 1234 confirmado';

        ch.assertQueue(queue, { durable: false });
        ch.sendToQueue(queue, Buffer.from(message));

        console.log('Evento enviado: %s', message);
    });
```

```
});
```

2. Circuit Breaker Pattern

Para garantir a resiliência do sistema, o **Circuit Breaker** é um padrão que interrompe chamadas para um serviço falho temporariamente. Isso evita sobrecarregar serviços que estão em falha.

Conclusão

O design de micro serviços envolve diversas abordagens e decisões arquiteturais que impactam diretamente a escalabilidade, resiliência e manutenção de sistemas distribuídos. As metodologias como **Domain-Driven Design**, **Arquitetura Orientada a Eventos** e **Circuit Breaker Pattern** oferecem maneiras de construir sistemas robustos e independentes, e as ferramentas como APIs REST, RabbitMQ e bancos de dados distribuídos tornam possível sua implementação na prática.