A Arquitetura de Micro Serviços e a Evolução do Backend

PROF. MARCOS ANTÔNIO DOS SANTOS

Objetivo da Aula:

- Apresentar a evolução do desenvolvimento backend, desde os sistemas monolíticos até a arquitetura de micro serviços.
- Explicar os componentes e as características da arquitetura de micro serviços.
- Mostrar exemplos de como essa arquitetura é organizada e implementada.

Sistemas monolíticos

- No começo da era da computação, as aplicações eram monolíticas. Isso significa que todo o código-fonte e as funcionalidades estavam centralizadas em um único projeto. Embora simples de desenvolver e implantar, os sistemas monolíticos apresentavam desafios à medida que os projetos cresciam.

Características dos Monolitos:

- **Código Centralizado**: Todo o código da aplicação (autenticação, banco de dados, lógica de negócios) está em um único pacote.
- **Escalabilidade Limitada**: A aplicação inteira precisa ser escalada mesmo que apenas uma funcionalidade necessite de mais recursos.
- **Manutenção Complexa**: Alterações ou atualizações em uma pequena parte da aplicação exigem recompilar e reinstalar o sistema inteiro.

Exemplo de um sistema monolítico (Node.js):

```
const express = require('express');
const app = express();

// Funcionalidade de autenticação
app.get('/login', (req, res) => {
    res.send('Login Page');
});

// Funcionalidade de listagem de produtos
app.get('/products', (req, res) => {
    res.send('Product List');
});

app.listen(3000, () => {
    console.log('Aplicação monolítica rodando na porta 3000');
});
```

Arquitetura em Camadas (MVC)

Arquitetura em Camadas (MVC)

- Para lidar com a complexidade crescente, as arquiteturas baseadas em camadas, como o **Model-View-Controller (MVC)**, começaram a surgir.
- Elas separavam a lógica da interface do usuário (view), a lógica de negócios (model) e o controle de fluxo (controller), melhorando a organização do código.

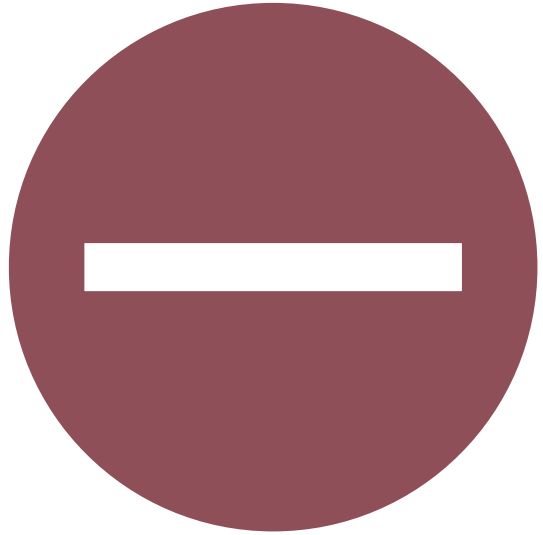
Características da Arquitetura MVC

- Separação de responsabilidades (UI, lógica de negócios e controle).
- Mais organizada, mas ainda sofre com escalabilidade limitada em sistemas grandes.

Arquitetura em Camadas (MVC)

```
/app
    /src
        /controllers
        /models
        /routes
        /services
        /views
    app.js
```

No entanto, conforme os sistemas começaram a crescer ainda mais, o MVC mostrou limitações em termos de escalabilidade e complexidade de manutenção.



Arquitetura Orientada a Serviços (SOA)

Arquitetura Orientada a Serviços (SOA)

- A **Arquitetura Orientada a Serviços (SOA)** foi uma tentativa de modularizar os sistemas de software. Em vez de um grande monolito, a SOA dividia a aplicação em serviços reutilizáveis que se comunicavam entre si. Embora tenha sido um avanço, a SOA apresentava problemas de sobrecarga de comunicação e complexidade na integração de sistemas.

Características da SOA:

- Reutilização de serviços em diferentes partes da aplicação.
- Pesada, com integração complexa entre serviços.

Arquitetura Orientada a Serviços (SOA)

Este exemplo mostra uma arquitetura orientada a serviços com **Node.js**, onde cada serviço é autônomo, tem sua própria lógica de negócios e banco de dados, e se comunica com outros serviços via **HTTP REST**. A abordagem modular torna o sistema mais escalável e fácil de manter, sendo um excelente ponto de partida para um projeto de micro serviços.

```
/user-service
    /src
        /controllers
        /models
        /routes
        /services
    app.js

/product-service
    /src
        /controllers
        /models
        /routes
        /services
    app.js
```

Arquitetura de Micro Serviços

Arquitetura de Micro Serviços

- A **arquitetura de micro serviços** é um estilo arquitetural que divide uma aplicação em um conjunto de serviços menores, independentes e orientados a tarefas específicas. Esses serviços podem ser desenvolvidos, implantados e escalados de forma autônoma.

Características da Arquitetura de Micro Serviços:

- **Descentralização**: Diferentes serviços têm suas próprias responsabilidades e podem ser mantidos e desenvolvidos independentemente.
- **Escalabilidade Independente**: Cada serviço pode ser escalado de acordo com suas necessidades, sem impactar os outros.
- **Desenvolvimento Autônomo**: Times diferentes podem trabalhar em diferentes serviços sem interferência, usando diferentes linguagens e tecnologias, se necessário.
- **Resiliência**: A falha de um serviço não impacta o sistema inteiro. Outros serviços continuam funcionando.

Componentes da Arquitetura de Micro Serviços

- **Serviço Individual:** Cada serviço na arquitetura de micro serviços representa uma pequena unidade de funcionalidade. Um serviço pode ser responsável pelo gerenciamento de usuários, outro por processamento de pagamentos, e assim por diante.
- **API Gateway:** é uma camada que atua como uma fachada para o conjunto de micro serviços. Ele fornece uma única entrada para os clientes e distribui as solicitações para os micro serviços apropriados. O API Gateway também pode implementar autenticação, roteamento, monitoramento e balanceamento de carga.

Componentes da Arquitetura de Micro Serviços

Componentes da Arquitetura de Micro Serviços

- **Monitoramento e Logging:** Com múltiplos serviços rodando em paralelo, é fundamental implementar um sistema robusto de monitoramento e logs para detectar falhas e gargalos. Ferramentas como **Prometheus** para monitoramento e **ELK (Elasticsearch, Logstash, Kibana)** para centralização de logs são comuns em arquiteturas de micro serviços.

Monolítico vs Micro Serviços

Monolítico	Micro Serviços
Um único código que contém todas as funcionalidades	Dividido em pequenos serviços independentes
Atualizações e deploys lentos	Ciclos de desenvolvimento e deploys rápidos
Escalabilidade limitada	Escalabilidade independente de cada serviço
Falha de um componente afeta o sistema todo	Falhas isoladas a serviços individuais

Arquiteturas de aplicação: Exercício

- Pesquise e descreva em um documento PDF os seguintes tópicos sobre as três arquiteturas de software apresentadas:

1. **Um cenário ideal de uso;**
2. **Vantagens de cada arquitetura;**
3. **Desvantagens de cada arquitetura;**
4. **Desafios para implementação e sustentação;**

API Gateway

- Centralização de recursos como log, autenticação, restrição
- Retentativas em caso de falha
- Unidade nas requisições dos clientes
- Segurança
- Versionamento
- Encapsulamento
- Independência de tecnologias

Design de micro serviços

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

- **Decomposição Baseada em Domínio (Domain-Driven Design)**
- **Decomposição Baseada em Casos de Uso**
- **Banco de Dados por Serviço**
- **Comunicação entre Micro Serviços**

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:

- **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.

Exemplo DDD

```
// 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.).

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.

Exemplo

- **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.

Exemplo

```
// 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 porta 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.

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

mongoose.connect('mongodb://localhost/products', { useNewUrlParser: true, useUni
fiedTopology: 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).

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):

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

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

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

Exemplo de Comunicação Assíncrona (RabbitMQ):

```
// 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);
    });
});
```

Comunicação entre Micro Serviços

- **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

- **Event-Driven Architecture (Arquitetura Orientada a Eventos)**
- **Circuit Breaker Pattern**

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.

Event-Driven Architecture (Arquitetura Orientada a Eventos)

```
// 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);
    });
});
```

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.