Tutorial: API REST com Node.js, Express e Sequelize para Controle de Pedidos (Microsserviços)

Introdução

Este guia detalha o processo de transformação de uma aplicação monolítica em uma arquitetura de **microsserviços** utilizando Node.js, Express, Sequelize (ORM) e MySQL. O projeto de exemplo gerencia dois domínios distintos: **Clientes** e **Pedidos**, cada um implementado como um serviço independente, com seu próprio banco de dados e expondo suas próprias APIs REST. A comunicação entre os serviços é feita de forma síncrona via HTTP (REST), simulando um cenário real de microsserviços.

A arquitetura de microsserviços traz benefícios como escalabilidade independente, isolamento de falhas, facilidade de deploy e evolução tecnológica por serviço. Ao final deste tutorial, você terá dois serviços funcionando e se comunicando, prontos para serem estendidos com mais funcionalidades.

Objetivos

Os principais objetivos deste guia são:

1. **Separar domínios** em serviços independentes, cada um com seu próprio banco de dados.
2. **Implementar comunicação síncrona** via HTTP entre os serviços (Pedido consulta Cliente).
3. **Configurar bancos de dados MySQL** separados para cada microsserviço.
4. **Desenvolver CRUD completo** para as entidades Cliente e Pedido.

5. **Estruturar um projeto Node.js** seguindo boas práticas para microsserviços (separação de camadas: models, controllers, routes, config).
6. **Compreender os principais métodos do Sequelize** para persistência e consultas.

Tecnologias Utilizadas

- **Node.js** (versão 18 ou superior)
- **Express** – framework web para criação das APIs
- **Sequelize** – ORM para modelagem e interação com o banco de dados
- **MySQL** – banco de dados relacional
- **dotenv** – gerenciamento de variáveis de ambiente
- **node-fetch** – para requisições HTTP entre serviços (substituto do fetch nativo em versões antigas do Node; a partir da versão 18 pode-se usar o fetch global)
- **Postman** ou **Insomnia** – para testes das APIs

Estrutura do Projeto

A organização das pastas reflete a separação dos serviços:

```
nodejs-microservices/
│
├─ cliente-service/
│   ├─ package.json
│   ├─ .env
│   ├─ app.js
│   ├─ config/
│   │   └─ database.js
│   ├─ models/
│   │   └─ Cliente.js
│   ├─ controllers/
│   │   └─ cliente.controller.js
│   └─ routes/
│       └─ cliente.routes.js
│
```

```
├─ pedido-service/
│   ├─ package.json
│   ├─ .env
│   ├─ app.js
│   ├─ config/
│   │   └─ database.js
│   ├─ models/
│   │   └─ Pedido.js
│   ├─ controllers/
│   │   └─ pedido.controller.js
│   └─ routes/
│       └─ pedido.routes.js
│
└─ README.md
```

Cada serviço roda em uma porta diferente e possui seu próprio banco de dados (ou schema) no MySQL. O serviço de Pedidos consulta o serviço de Clientes para validar o `clienteId` antes de criar ou atualizar um pedido.

Passo 1: Configuração Inicial dos Serviços

1.1 Criar as pastas e os arquivos `package.json`

Abra o terminal e execute:

```
mkdir nodejs-microservices
cd nodejs-microservices
mkdir cliente-service pedido-service
```

Dentro de cada pasta, crie um arquivo `package.json` com o conteúdo adequado.

cliente-service/package.json

```
{
  "name": "cliente-service",
  "version": "1.0.0",
  "type": "module",
  "main": "app.js",
  "scripts": {
```

```
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "sequelize": "^6.32.1",
    "mysql2": "^3.4.0",
    "dotenv": "^16.1.0"
  }
}
```

pedido-service/package.json

```
{
  "name": "pedido-service",
  "version": "1.0.0",
  "type": "module",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "sequelize": "^6.32.1",
    "mysql2": "^3.4.0",
    "dotenv": "^16.1.0",
    "node-fetch": "^3.3.1"
  }
}
```

Nota: O `pedido-service` necessita do pacote `node-fetch` para fazer requisições HTTP ao `cliente-service`. Se você estiver usando Node.js 18+, pode utilizar o `fetch` nativo e remover essa dependência.

1.2 Instalar as dependências

Em cada pasta, execute:

```
npm install
```

Passo 2: Configuração do Banco de Dados e Variáveis de Ambiente

Crie um arquivo `.env` na raiz de cada serviço com as seguintes variáveis:

cliente-service/.env

```
DB_NAME=cliente_db
DB_USER=root
DB_PASS=sua_senha
DB_HOST=localhost
PORT=3001
```

pedido-service/.env

```
DB_NAME=pedido_db
DB_USER=root
DB_PASS=sua_senha
DB_HOST=localhost
CLIENTE_SERVICE_URL=http://localhost:3001/clientes
PORT=3002
```

> **Atenção:** Substitua `sua_senha` pela senha do seu MySQL. O banco de dados `cliente_db` e `pedido_db` devem ser criados manualmente no MySQL antes de executar os serviços (ou você pode configurar o Sequelize para criá-los automaticamente, conforme mostraremos adiante).

2.1 Arquivo de configuração do Sequelize

Crie o arquivo `config/database.js` em ambos os serviços com o mesmo conteúdo (apenas importa as variáveis do `.env`):

```
import { Sequelize } from 'sequelize';
import dotenv from 'dotenv';

dotenv.config();

const sequelize = new Sequelize(
  process.env.DB_NAME,
  process.env.DB_USER,
```

```
  process.env.DB_PASS,
  {
    host: process.env.DB_HOST,
    dialect: 'mysql'
  }
);

export default sequelize;
```

Passo 3: Criação dos Modelos

3.1 Modelo Cliente (`cliente-service/models/Cliente.js`)

```
import { DataTypes } from 'sequelize';
import sequelize from '../config/database.js';

const Cliente = sequelize.define('Cliente', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  nome: {
    type: DataTypes.STRING,
    allowNull: false
  },
  email: {
    type: DataTypes.STRING,
    allowNull: false,
    unique: true
  }
}, {
  timestamps: true // adiciona createdAt e updatedAt automa
ticamente
});

export default Cliente;
```

3.2 Modelo Pedido (`pedido-service/models/Pedido.js`)

```
import { DataTypes } from 'sequelize';
import sequelize from '../config/database.js';

const Pedido = sequelize.define('Pedido', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  descricao: {
    type: DataTypes.STRING,
    allowNull: false
  },
  valor: {
    type: DataTypes.DOUBLE,
    allowNull: false
  },
  clienteId: {
    type: DataTypes.INTEGER,
    allowNull: false
    // Não há FK real no banco — a integridade é garantida
pela chamada HTTP ao cliente-service
  }
}, {
  timestamps: true
});

export default Pedido;
```

Passo 4: Criação dos Controladores

4.1 Controlador de Clientes (`cliente-service/controllers/cliente.controller.js`)

```
import Cliente from '../models/Cliente.js';

// Criar um novo cliente
export const createCliente = async (req, res) => {
  try {
    const cliente = await Cliente.create(req.body);
    res.status(201).json(cliente);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

// Listar todos os clientes
export const getClientes = async (req, res) => {
  try {
    const clientes = await Cliente.findAll();
    res.json(clientes);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

// Buscar um cliente por ID
export const getClienteById = async (req, res) => {
  try {
    const cliente = await Cliente.findByPk(req.params.id);
    if (!cliente) {
      return res.status(404).json({ error: 'Cliente não enc
ontrado' });
    }
    res.json(cliente);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

// Atualizar um cliente
export const updateCliente = async (req, res) => {
```

```
  try {
    const cliente = await Cliente.findByPk(req.params.id);
    if (!cliente) {
      return res.status(404).json({ error: 'Cliente não enc
ontrado' });
    }
    await cliente.update(req.body);
    res.json(cliente);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

// Deletar um cliente
export const deleteCliente = async (req, res) => {
  try {
    const cliente = await Cliente.findByPk(req.params.id);
    if (!cliente) {
      return res.status(404).json({ error: 'Cliente não enc
ontrado' });
    }
    await cliente.destroy();
    res.json({ message: 'Cliente deletado com sucesso' });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};
```

4.2 Controlador de Pedidos (`pedido-service/controllers/pedido.controller.js`)

Aqui utilizamos `fetch` para validar a existência do cliente antes de qualquer operação que envolva `clienteId`.

```
import Pedido from '../models/Pedido.js';
import fetch from 'node-fetch';

// Função auxiliar para validar cliente
const validarCliente = async (clienteId) => {
```

```
  const url = `${process.env.CLIENTE_SERVICE_URL}/${cliente
Id}`;
  const res = await fetch(url);
  if (!res.ok) {
    throw new Error('Cliente não encontrado');
  }
  return res.json();
};

// Criar um novo pedido
export const createPedido = async (req, res) => {
  try {
    await validarCliente(req.body.clienteId);
    const pedido = await Pedido.create(req.body);
    res.status(201).json(pedido);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
};

// Listar todos os pedidos
export const getPedidos = async (req, res) => {
  try {
    const pedidos = await Pedido.findAll();
    res.json(pedidos);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

// Buscar um pedido por ID
export const getPedidoById = async (req, res) => {
  try {
    const pedido = await Pedido.findByPk(req.params.id);
    if (!pedido) {
      return res.status(404).json({ error: 'Pedido não enco
ntrado' });
    }
```

```
    res.json(pedido);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

// Atualizar um pedido
export const updatePedido = async (req, res) => {
  try {
    const pedido = await Pedido.findByPk(req.params.id);
    if (!pedido) {
      return res.status(404).json({ error: 'Pedido não enco
ntrado' });
    }
    // Se o clienteId foi alterado, valida o novo cliente
    if (req.body.clienteId && req.body.clienteId !== pedid
o.clienteId) {
      await validarCliente(req.body.clienteId);
    }
    await pedido.update(req.body);
    res.json(pedido);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
};

// Deletar um pedido
export const deletePedido = async (req, res) => {
  try {
    const pedido = await Pedido.findByPk(req.params.id);
    if (!pedido) {
      return res.status(404).json({ error: 'Pedido não enco
ntrado' });
    }
    await pedido.destroy();
    res.json({ message: 'Pedido deletado com sucesso' });
  } catch (err) {
    res.status(500).json({ error: err.message });
```

```
  }
};
```

Passo 5: Criação das Rotas

5.1 Rotas de Cliente (`cliente-service/routes/cliente.routes.js`)

```
import { Router } from 'express';
import {
  createCliente,
  getClientes,
  getClienteById,
  updateCliente,
  deleteCliente
} from '../controllers/cliente.controller.js';

const router = Router();

router.post('/', createCliente);
router.get('/', getClientes);
router.get('/:id', getClienteById);
router.put('/:id', updateCliente);
router.delete('/:id', deleteCliente);

export default router;
```

5.2 Rotas de Pedido (`pedido-service/routes/pedido.routes.js`)

```
import { Router } from 'express';
import {
  createPedido,
  getPedidos,
  getPedidoById,
  updatePedido,
  deletePedido
} from '../controllers/pedido.controller.js';
```

```
const router = Router();

router.post('/', createPedido);
router.get('/', getPedidos);
router.get('/:id', getPedidoById);
router.put('/:id', updatePedido);
router.delete('/:id', deletePedido);

export default router;
```

Passo 6: Inicialização dos Serviços (app.js)

6.1 `cliente-service/app.js`

```
import express from 'express';
import dotenv from 'dotenv';
import sequelize from './config/database.js';
import clienteRoutes from './routes/cliente.routes.js';

dotenv.config();
const app = express();
app.use(express.json());
app.use('/clientes', clienteRoutes);

(async () => {
  try {
    // Sincroniza o modelo com o banco de dados (cria a tab
ela se não existir)
    await sequelize.sync({ alter: true }); // use { force:
true } apenas em desenvolvimento para recriar a tabela
    console.log('Cliente DB sincronizado!');
    app.listen(process.env.PORT, () => {
      console.log(`Cliente Service rodando na porta ${proce
ss.env.PORT}`);
    });
  } catch (err) {
```

```
    console.error('Erro ao iniciar o Cliente Service:', er
r);
  }
})();
```

6.2 `pedido-service/app.js`

```
import express from 'express';
import dotenv from 'dotenv';
import sequelize from './config/database.js';
import pedidoRoutes from './routes/pedido.routes.js';

dotenv.config();
const app = express();
app.use(express.json());
app.use('/pedidos', pedidoRoutes);

(async () => {
  try {
    await sequelize.sync({ alter: true });
    console.log('Pedido DB sincronizado!');
    app.listen(process.env.PORT, () => {
      console.log(`Pedido Service rodando na porta ${proces
s.env.PORT}`);
    });
  } catch (err) {
    console.error('Erro ao iniciar o Pedido Service:', er
r);
  }
})();
```

Passo 7: Comunicação entre Serviços

A comunicação síncrona ocorre quando o `pedido-service` precisa validar se um `clienteId` existe. Para isso, ele faz uma requisição HTTP GET para o endpoint `/clientes/:id` do `cliente-service`. Essa abordagem mantém os serviços

desacoplados em termos de banco de dados, mas introduz uma dependência de rede.

Em um ambiente de produção, considere usar:

- **Service Discovery** (como Consul ou Eureka) para localizar dinamicamente os serviços.
- **API Gateway** para rotear requisições e centralizar preocupações comuns (autenticação, logging, etc.).
- **Mensageria assíncrona** (RabbitMQ, Kafka) para comunicação mais resiliente e desacoplada.

Testando a Aplicação

7.1 Executar os serviços

Em dois terminais separados, execute:

```
# Terminal 1
cd cliente-service
npm start

# Terminal 2
cd pedido-service
npm start
```

7.2 Exemplos de requisições com Postman

Criar um cliente (POST)

`http://localhost:3001/clientes`

Body (JSON):

```
{
  "nome": "João Silva",
  "email": "joao@email.com"
}
```

Criar um pedido (POST)

`http://localhost:3002/pedidos`

Body (JSON):

```
{
  "descricao": "Pedido de notebook",
  "valor": 3500.00,
  "clienteId": 1
}
```

> O serviço de pedidos validará se o cliente com `id=1` existe antes de criar o pedido.

Listar pedidos (GET)

`http://localhost:3002/pedidos`

Buscar cliente por ID (GET)

`http://localhost:3001/clientes/1`

Atualizar pedido (PUT)

`http://localhost:3002/pedidos/1`

Body com os campos a alterar.

Deletar pedido (DELETE)

`http://localhost:3002/pedidos/1`

Consultas Personalizadas com Sequelize

Além dos métodos básicos, o Sequelize permite criar consultas complexas usando a sintaxe de objetos ou SQL bruto.

Exemplo com `where` e operadores:

```
import { Op } from 'sequelize';

const pedidosCaros = await Pedido.findAll({
  where: {
    valor: {
      [Op.gt]: 1000  // maior que 1000
    }
  }
});
```

Exemplo com SQL nativo:

```
const [results, metadata] = await sequelize.query("SELECT *
FROM Pedidos WHERE valor > 1000");
```

Desafio

Agora que você tem dois microsserviços funcionando e se comunicando, tente estender o sistema com o seguinte desafio:

1. **Adicionar um novo serviço** (ex.: `produto-service`) com seu próprio banco de dados e CRUD. Modifique `pedido-service` para incluir uma lista de produtos no pedido, validando cada produto via HTTP.

Conclusão

Neste tutorial, você aprendeu a construir uma arquitetura de microsserviços utilizando Node.js, Express, Sequelize e MySQL. Os domínios de Clientes e Pedidos foram separados em serviços independentes, cada um com seu próprio banco de dados e API. A comunicação entre eles foi realizada de forma síncrona via HTTP, garantindo a integridade referencial sem acoplamento de banco de dados.

Essa base permite escalar cada serviço de forma independente, escolher tecnologias diferentes por serviço (se necessário) e evoluir o sistema com mais facilidade. Os conceitos apresentados – separação por domínio, uso de ORM, comunicação entre serviços – são fundamentais para o desenvolvimento de sistemas modernos e distribuídos.

Referências Bibliográficas

- Node.js. **Node.js Documentation**. Disponível em: https://nodejs.org/en/docs/
- Express. **Express Guide**. Disponível em: https://expressjs.com/en/guide/routing.html
- Sequelize. **Sequelize Documentation**. Disponível em: https://sequelize.org/docs/v6/

- MySQL. **MySQL Reference Manual**. Disponível em: https://dev.mysql.com/doc/
- MDN Web Docs. **Fetch API**. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/API/Fetch_API
- Martin Fowler. **Microservices**. Disponível em: https://martinfowler.com/articles/microservices.html