TRYBE
Modulo III – Back-end

Bloco 27: NodeJS - Camada de Serviço e Arquitetura Rest e Restful

Programa do bloco -

Intro - Arquitetura de software

MSC: padrão arquitetural para organizar aplicações Node.js e Express.
Acessaremos Bds MongoDB e MySQL e entenderemos o que é **arquitetura de cliente-servidor**.

"Arquitetura é um conhecimento compartilhado por desenvolvedores experientes sobre como organizar um sistema de software." Martin Fowler

Existem padrões de arquitetura específicos para problemas específicos e divisão de responsabilidades por camadas. Atenção particular também para as regras de negócio.

1) Arquitetura de Software - Camada de Model

Model

O model é onde nós manipulamos e definimos a **estrutura dos nossos dados.**
Responsável por abstrair completamente os **detalhes de acesso e armazenamento**, fornecendo somente uma **API que permita requisitar e manipular esses dados**.

Para boa manutenibilidade e reusabilidade de código, o model deve ser completamente **desacoplado das demais camadas**.

Model com MySQL

1. Criar e popular o banco de dados
CREATE TABLE / INSERT INTO etc no Workbench.

2. Estabelecer uma conexão com o banco
$ mkdir model-example – pasta para conter o projeto
$ cd model-example
$ npm init -y – iniciar um projeto node
$ npm install mysql2 – driver para se conectar com mysql

Pasta models, arquivo connection.js :

```
const mysql = require('mysql2/promise');
const connection = mysql.createPool({
  host: 'localhost',
  user: 'root',
  password: 'senha123',
  database: 'model_example'});
module.exports = connection;
```

O método createPool retorna um objeto Pool representando uma sessão com o banco, armazenado na variável connection.
/promise permite evitar trabalhar apenas com callbacks no mysql.

3. Criar o model

= Documento que vai servir de intermediário entre o connection e os endpoints.

```
const getAll = async () => {
  const [authors] = await connection.execute(
    'SELECT id, first_name, middle_name, last_name FROM model_example.authors;',
  );
  return authors.map(serialize).map(getNewAuthor);
};
```

Método **execute** retorna uma Promise que quando resolvida, fornece *um array com 2 campos, [rows, fields]*. Usar *connection.execute* para extrair dados retornados pelo BD.

Outra dica: converter snake_case → camelCase via **serialize**.

```
// Serializa o nome dos campos de snake_case para camelCase

const serialize = (authorData) => ({
  id: authorData.id,
  firstName: authorData.first_name,
  middleName: authorData.middle_name,
  lastName: authorData.last_name});
```

Com o model é criado a rota.

4. CRUDs com dados retornados
$ npm install express / Diversos endpoints no index.js / *$ npm install body-parser*

Model com MongoDB

1. Popular o banco de dados
use model_example / db.authors.insertMany no mongo.

2. Estabelecer uma conexão com o banco
npm install mongodb – driver

No connection.js :

```
const { MongoClient } = require('mongodb');
const MONGO_DB_URL = 'mongodb://127.0.0.1:27017';
const connection = () => {
return MongoClient
.connect(MONGO_DB_URL, {
useNewUrlParser: true,
useUnifiedTopology: true})
.then((conn) => conn.db('model_example'))
.catch((err) => {
console.error(err);
process.exit(1);
});
};
module.exports = connection;
```

Parâmetros useNewUrlParser e useUnifiedTopology dizem ao driver como se conectar ao banco.

3. CRUD

Operações com o BD semelhantes porém com sintaxe diferenciada, usando possibilidades do mongo.
Opcional: desinstalar mysql via *npm uninstall mysql2* .

Dicas diversas

```
database: 'live_lecture_27_1',
port: '3306',
```
// No connection.js. **Port 3306** corresponde com mysql.

_

Ver TOP 10 VULNERABILITIES.

Exemplificando a INJECTION:

```
const [result] = await connection.execute(
  `INSERT INTO characters (name, cartoon) VALUES ('${name}', '${cartoon}')`
);
```
// Vulnerabilidade pois alguém pode dar variaveis que formam comandos para deletar BD!

```
const [
  result,
] = await connection.execute(
  'INSERT INTO characters (name, cartoon) VALUES (?, ?)',
  [name, cartoon]
);
```
// forma correta

_

Sintaxe com [] no connection.execute:
const [user] = await connection.execute('SELECT * FROM users_crud.users WHERE id = ?', [id]);

---

2) Arquitetura de Software - Camada de Controller e Service

Objetivo: entender a estrutura de uma aplicação em *camadas*, saber delegar *responsabilidades específicas* para cada parte do app e elhorar manutenibilidade e reusabilidade do código. Arquiteturas focadas em APIs cada vez mais populares.

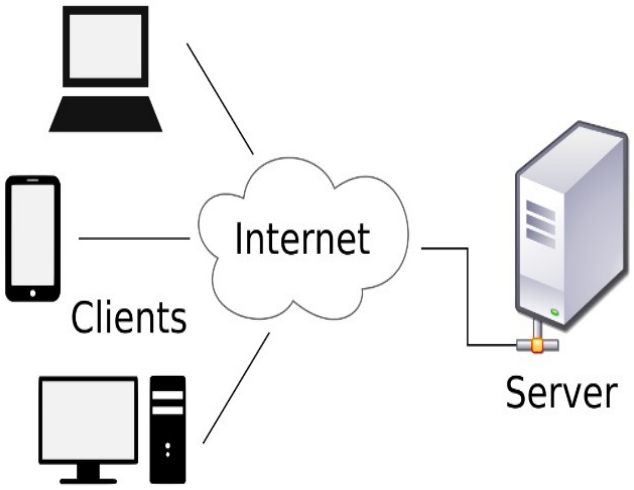
Arquitetura Cliente-Servidor

Arquitetura de aplicações distribuídas em que as **tarefas são distribuídas entre** módulos que fornecem algum recurso ou serviço, chamados de **servidores,** e módulos que requisitam os serviços, chamados de **clientes**. Podem estar no mesmo computador ou separados.

Arquitetura já integrada nas Bds que conhecemos, por exemplo mysqld é servidor do MySQL.

Hoje, back-end fornece APIs em vez de retornar HTML diretamente para cliente, e o browser cria o HTML além de renderiza-lo.

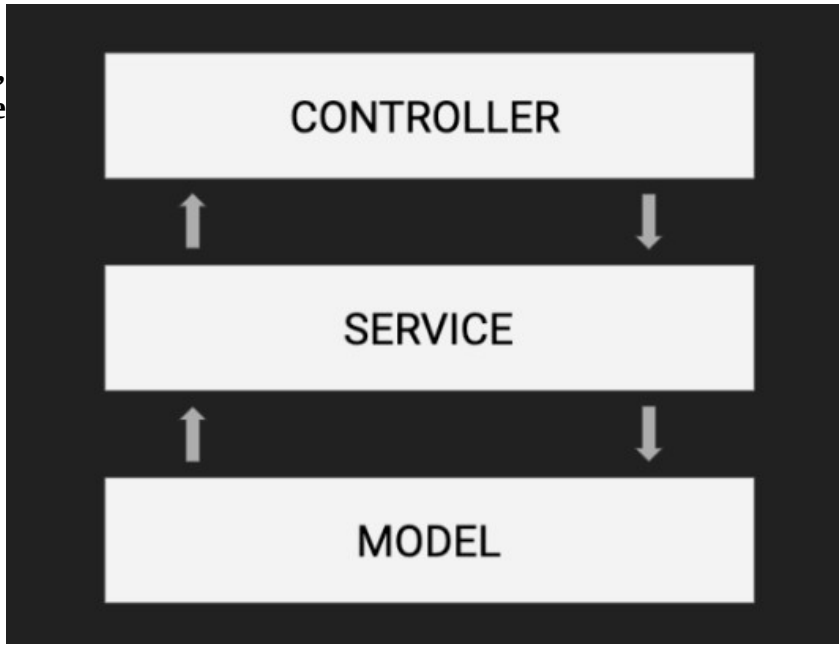
Vantagens: mesma API consumida por diversos clientes, carga do servidor menor.



Arquitetura cliente-servidor

Camada de controle

O **controller recebe as requisições do cliente, consulta o service, envia para o cliente o que o service retornar**, podendo ser uma mensagem de erro em caso de falha ou as informações requisitas em caso de sucesso.
Analogia: garçon de um restaurante.

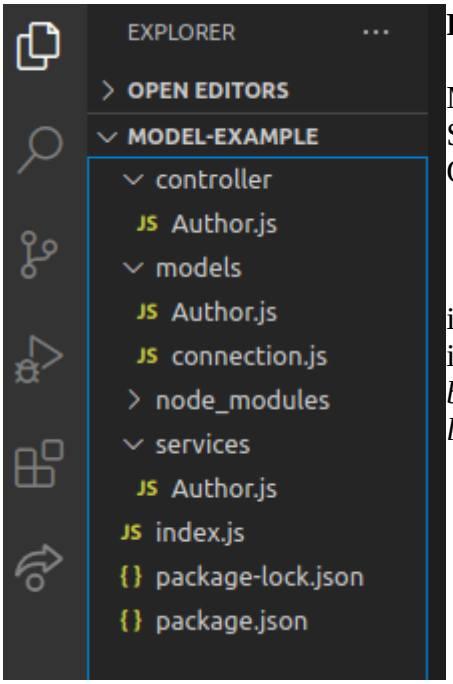


Camada de serviço

Fica situada entre as camadas de controller e model e é responsável pela **lógica de negócio**. Seria como uma extensão da camada de model e suas regras de negócios.
Analogia: chef de cozinha.

Boa quando tiver nada de BD, req, res, http.
Apenas service e acesso a dados e funções externas.

Na prática



Pastas models, services e controllers, cada um com nomeescolhido.js .

Models foca na conexão com o banco.
Services foca na lógica e funções de apoio.
Controller concentra os req e res.

index.js principal apenas para configurar e iniciar *(estrutura mínima de express, bodyparser, importações, endpoints e port listener).* →

```
const express = require('express');
const bodyParser = require('body-parser');

const Author = require('../controllers/Author');

const app = express();

app.use(bodyParser.json());

app.get('/authors', Author.getAll);

app.get('/authors/:id', Author.findById);

app.post('/authors', Author.create);

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Ouvindo a porta ${PORT}`);
});
```

Boas práticas em arquitetura de software

1. Pensar antes de codar;

2. Pensar em componentes;

3. Organizar pastas, que seja por tema ou papel técnico:

```
└── author
|   ├── authorController.js
|   ├── authorService.js
|   └── authorModel.js
└── book
|   └── bookController.js
|   └── bookService.js
|   └── bookModel.js
```

```
└── controllers
|   ├── authorController.js
|   └── bookController.js
└── services
|   ├── authorService.js
|   └── bookService.js
└── models
|   ├── authorModel.js
|   └── bookModel.js
```

4. Manter o Express longe, ou seja contido integralmente dentro das fronteiras do controller;

5. Manter configuração separada e segura:

→ Para poucas, podemos *setar variáveis no terminal.* Por exemplo:
DB_URL="mongodb://localhost:27017" node index.js
console.log(process.env.DB_URL) // mongodb://localhost:27017

→ Para mais, usar o ***env*** (objeto da variável global process do Node):
npm install dotenv

.env (depois integrar no .gitignore)
PORT=3000
DB_URL=mongodb://localhost:27017
DB_NAME=model_example

index.js
require('dotenv').config();
// ...
const PORT = process.env.PORT;
app.listen(PORT, () => console.log(`Server listening on port ${PORT}`));

models.connection.js

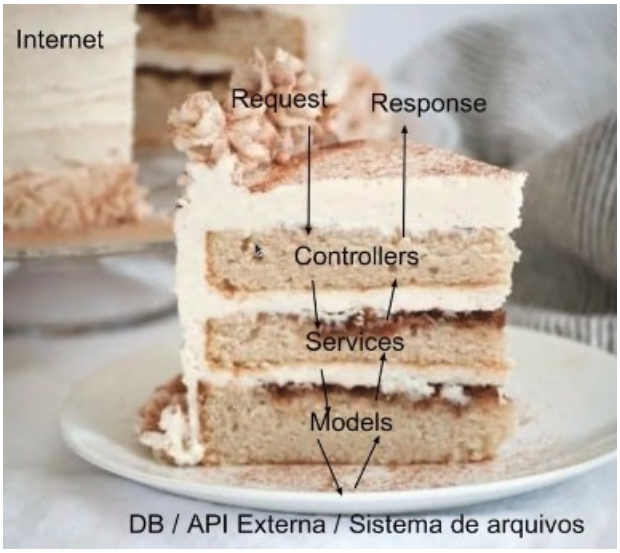
```
const mongoClient = require('mongodb').MongoClient;

const connection = () => {
  return mongoClient
    .connect(process.env.DB_URL, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    })
    .then((conn) => conn.db(process.env.DB_NAME))
    .catch((err) => {
      console.error(err);
      process.exit(1);
    });
};

module.exports = connection;
```

Dicas diversas

Relação controller-middlewares: a camada de controllers é formada por middlewares. Porém, ao organizar o código em arquivos, isso pode se refletir, ou não. Existe a possibilidade de criar uma pasta middlewares fora o controllers para os que não resolvem problema do cliente ou manipulam req e res diretamente, por exemplo autenticação.



Retornar objetos de erro

```
if (!song)
  return {
    error: true,
    code: 'not_found',
    message: `Song with id ${id} was not found`,
  };

return song;
```

---

3) Arquitetura web - Rest e Restful

REST

Representational State Transfer (REST), Transferência Representacional de Estado, é um estilo de arquitetura de software, controlado pelo W3C, que define um c**onjunto de restrições (boas práticas a serem usadas para a criação de APIs** - padrão que vai delimitar como a API deve se comportar ao se comunicar com o mundo.

As 6 restrições para ser RESTful

RESTful = aplicar todos os princípios REST.

1. Interface uniforme
Interface de comunicação entre servidor e clienteInclui o **endpoint , o tipo de retorno e o uso dos verbos HTTP**.
Sendo que:
- No endpoint deve aparecer o *recurso* (abstração da informação, porexemplo planets no https://swapi.dev/api/planets/:id);
- O tipo de retorno deve ser *consistente* e feito numa sintaxe universal (header Content-type de qualquer mime type como json, XML ou Javascript);
- O verbo deve especificar o *tipo de ação* e as respostas sempre com *status()*.

2. Arquitetura cliente-servidor
API e cliente devem estar desacoplados.
Para garantir a independência dos desenvolvimentos respetivos, as responsabilidades devem estar atribuidas assim:
- cliente: exibição dos dados, experiência do usuário;
- servidor: armazenamento e acesso dos dados, cache, log etc.

3. Sem estado (stateless)
API Stateless *para atender todos pedidos*. A *requisição deve ser autossuficiente* ou seja conter todas as informações necessárias para a API realizar a ação. Vantagens: *transparência e escalabilidade*.

4. Cacheable
Cache do lado do cliente (browser) para conservar informações, deve ser usado sabiamente.
Respostas dadas pela API devem e*xplicitar se podem ou não ser cacheadas e por quanto tempo*.
Se trata no HTTP de um header - *Cache-Control: max-age=120*

5. Sistema em camadas (layered system)
Diferente da organização do código - Abstrair do cliente as *camadas necessárias para responder* a uma requisição.

6. Opcional: Código sob demanda (code on demand)
Possibilidade do servidor enviar código ao cliente, assim costumizando comportamento do cliente (exemplo: widget para ter chat).

Reflexões

→ Ser ou não ser RESTful
Depende do contexto, qualquer padrão pode ser evitado se a justificativa for relevante.

→ REST no Express
REST é padrão independentemente de ser com tecnologia Express ou outro framework.
Vantagem do express para o REST: organização de rotas, retornos por cliente, status HTTP.

<u>Dicas diversas</u>

1xx: Informação;

2xx: Sucesso;

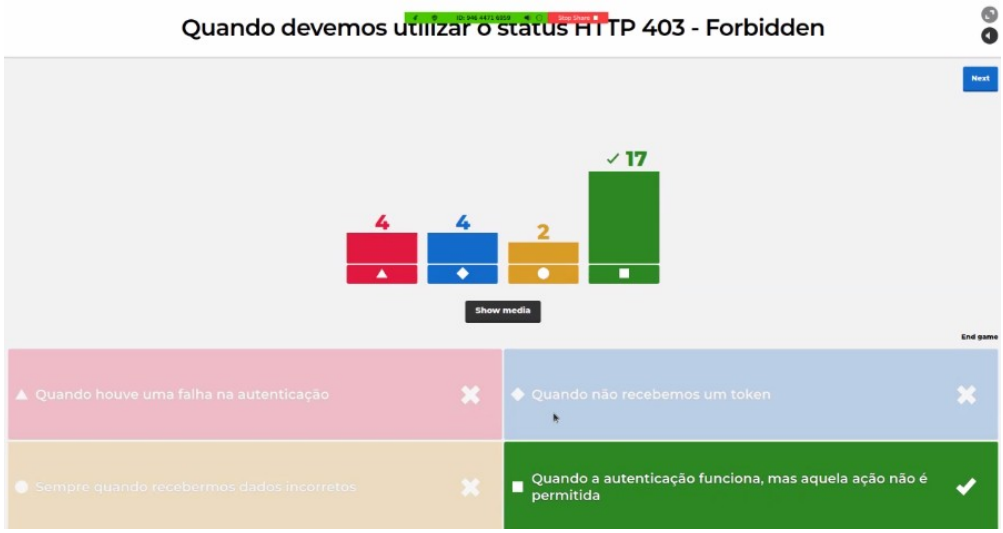
3xx: Redirecionamento;

4xx: Erro do cliente;

5xx: Erro no servidor.

Para lembrar mais facilmente dos status HTTP

CRUD: Post, Get, Put, Delete



Boa prática de Router no controller

const router = express.Router();
//Router é agrupador de middlewares

//Chamar no index.js com use, pode dar qual vai ser o começo da url sempre:
app.use('/products', require('./controllers/productController'));

Diferentes status de erro – o mais comum sendo 500

// como prever 404

res.json tem code e message, num app com mais de uma língua pode ser melhor usar o code e manipular no frontend.

Projeto – Store Manager

db.collection.findOneAndDelete()

Tratamento de erro manual:

No service

```
const getById = async (id) => {
  if (!ObjectId.isValid(id)) {
    throw {
      code: 'invalid_data',
      message: 'Wrong id format',
    };
  }
  const productById = await prodModel.getById(id);
  if (!productById) {
    throw {
      code: 'invalid_data',
      message: 'Wrong id format',
    };
  }
  return productById;
};
```

No controller

```
router.get('/:id', async (req, res) => {
  const { id } = req.params;
  try {
    const prodById = await prodService.getById(id);
    res.status(200).json(prodById);
  } catch (err) {
    if (err.code === 'invalid_data') {
      return res.status(422).json({ err });
    }
    console.error(err);
    res.status(500).json({ message: 'Erro interno aiaiai' });
  }
});
```
