INSTITUTO FEDERAL
Paraná
Campus Paranaguá

Ministério da Educação

ENSINO MÉDIO INTEGRADO: INFORMÁTICA
Disciplina de Desenvolvimento Web
Aula 14: Node/Express - CRUD (Controller + Service + Repository)

Gil Eduardo de Andrade

Conceitos Preliminares

(https://nodejs.org/pt)
(https://expressjs.com/pt-br/)

Introdução

O Node.js e o Express.js são duas tecnologias amplamente utilizadas no desenvolvimento web, especialmente voltadas para criação do **back-end(*)** da aplicação. Embora estejam relacionadas, elas desempenham papéis distintos.

*__back-end:__ parte do sistema que o usuário não vê, responsável pela lógica de negócios, banco de dados, entre outros.

Node.js: Ambiente de execução JavaScript - Servidor

O Node.js não é uma linguagem de programação, nem um framework, e sim um ambiente de execução JavaScript de código aberto e multiplataforma que permite executar código JavaScript fora do navegador web.
Tradicionalmente, o JavaScript era usado apenas para scripts voltados ao lado do cliente (front-end), ou seja, dentro dos navegadores para interatividade em páginas web. Com a criação do Node.js tornou-se possível usar JavaScript para desenvolver aplicações voltadas ao lado do servidor, criar ferramentas de linha de comando, e até mesmo construir aplicações desktop.

Node.js: Principais Características

- JavaScript em todo lugar: permite que desenvolvedores usem a mesma linguagem (JS) para o front-end e o back-end, possibilitando otimizar o processo de desenvolvimento;

- Assíncrono e não-bloqueante: foi projetado para lidar com muitas conexões simultâneas de forma eficiente. Em vez de esperar uma operação (como uma consulta a um banco de dados) ser concluída, antes de passar para a

próxima, o Node.js inicia a operação e continua processando outras tarefas, sendo notificado quando a operação anterior termina. Isso o torna rápido e escalável.

- Orientado a eventos: utiliza um modelo de "loop de eventos" (permite que um programa lide com múltiplas tarefas simultâneas sem bloquear a execução principal) para processar requisições, o que contribui para sua alta performance.

- Gerenciador de pacotes NPM: é o maior ecossistema de bibliotecas de código aberto do mundo, o que facilita a instalação, o compartilhamento e o gerenciamento de pacotes (módulos de código) reutilizáveis.

Node.js: Utilização

- APIs RESTful (interfaces que permitem a comunicação entre diferentes softwares).
- Servidores web de alta performance.
- Aplicações em tempo real (como chats e jogos online).
- Microsserviços.
- Ferramentas de linha de comando.

Express.js: Framework web para Node.js

O Express.js (comumente chamado apenas de Express) é um framework web minimalista e flexível para Node.js, que fornece um conjunto robusto de recursos para construir aplicações web e APIs de forma rápida e organizada.

Ainda que o Node.js forneça a capacidade de executar JavaScript no lado do servidor, ele é bastante "cru". Construir um servidor web do zero com Node.js puro exigiria muito código para lidar com tarefas comuns como roteamento de URLs, manipulação de requisições e respostas HTTP, e gerenciamento de sessões. É neste contexto que o Express se mostra extremamente útil.

Express.js: Funcionalidades

O Express simplifica o desenvolvimento, oferecendo:

- Sistema de roteamento robusto: Permite definir facilmente como sua aplicação responde a diferentes URLs (endpoints) e métodos HTTP (GET, POST, PUT, DELETE, etc.).

- Middleware: Permite que você adicione funções que são executadas em uma sequência específica para cada requisição. Isso é útil para tarefas como autenticação, log, tratamento de erros, e muito mais.

- Manipulação de requisições e respostas: Facilita o acesso a dados da requisição (parâmetros, corpo, cabeçalhos) e o envio de respostas (JSON, HTML, arquivos).

- Integração com motores de template: Embora minimalista, ele se integra facilmente com vários "motores de template" (ferramentas de software que permitem separar a lógica de apresentação - o que o usuário vê - da lógica de negócios - a funcionalidade do sistema - em aplicações web) para renderizar páginas HTML dinamicamente.

Express.js: Utilização

- Rapidez no desenvolvimento: abstrai muitas das complexidades de construir um servidor web com Node.js puro, permitindo que você se concentre na lógica da sua aplicação.

- Flexibilidade: por ser minimalista, não impõe muitas regras, permitindo que os desenvolvedores escolham as ferramentas e bibliotecas que melhor se adaptam às suas necessidades.

- Grande comunidade e documentação: como é um dos frameworks mais populares para Node.js, possui uma vasta comunidade, muitos recursos e excelente documentação.

Padrão: Conttroller / Service / Repository
(Controlador / Serviço / Repositório)
Controller, Service e Repository formam um padrão de arquitetura amplamente utilizado no desenvolvimento de software, especialmente em aplicações que seguem o padrão MVC(*) (Model-View-Controller) ou variantes dele. Este padrão promove uma separação das responsabilidades, tornando o código mais modular, testável e fácil de manter.

***MVC:** padrão de arquitetura de software que divide uma aplicação em três camadas interconectadas: Modelo (Model), Visão (View) e Controlador (Controller)

Resumidamente

- Controller: recebe as requisições do usuário, valida os dados de entrada e chama o serviço apropriado.

- Service: contém a lógica de negócio da aplicação, processando os dados e interagindo com o repositório.

- Repository: abstrai o acesso aos dados, permitindo que o serviço trabalhe com diferentes fontes de dados sem precisar conhecer os detalhes da sua implementação.

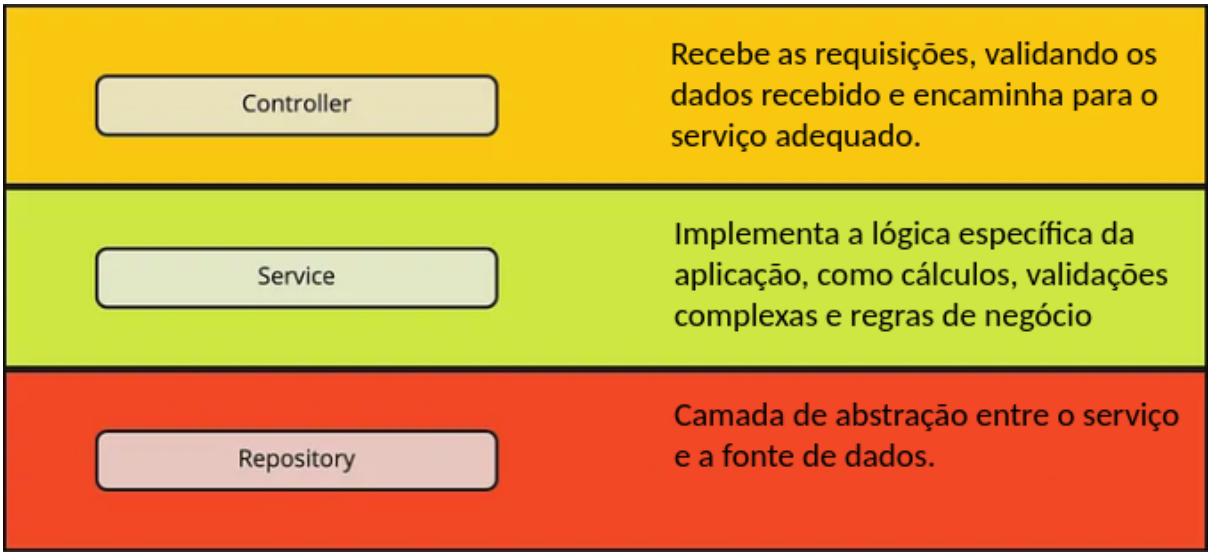


Figura 01: Padrão Controller / Service / Repository (Camadas).

Exemplificando

Controller

O controlador é a camada de entrada da aplicação, responsável por receber as requisições HTTP (ou de outro tipo), validando os dados recebidos e encaminhando-os para o serviço adequado. Ele não deve conter lógica de negócio, apenas a lógica de roteamento e manipulação da interface.

Service

O serviço é a camada onde a lógica de negócio da aplicação reside. Ele recebe os dados do controlador, efetua seu processamento, de acordo com as regras de negócio e, se necessário, interage com o repositório para acessar ou salvar dados. Os serviços são responsáveis por implementar a lógica específica da aplicação, como cálculos, validações complexas e regras de negócio.

Repository

O repositório atua como uma camada de abstração entre o serviço e a fonte de dados (banco de dados, API externa, etc.). Ele encapsula as operações de acesso a dados, como consultas, inserções, atualizações e exclusões, permitindo que o serviço trabalhe com uma interface padronizada, sem se preocupar com os detalhes de como os dados são armazenados ou recuperados.

Benefícios do padrão Controller, Service e Repository

- Separação das responsabilidades: cada camada tem uma responsabilidade específica, facilitando a manutenção e o desenvolvimento.
- Testabilidade: maior facilidade em testar a aplicação, visto que cada camada pode ser testada de forma isolada, já que são desacopladas;
- Reutilização de código: os serviços podem ser reutilizados por diferentes controllers ou outras partes da aplicação;
- Escalabilidade: a separação das camadas facilita a escalabilidade da aplicação;
- Manutenção: as alterações efetuadas numa camada têm menor impacto sobre as outras camadas.

CRIANDO UM CRUD SIMPLES

UTILIZANDO PADRÃO: Controlador / Serviço / Repositório

Pré-requisitos

- Node.js instalado (versão 14 ou superior)
- NPM (instalado automaticamente com o Node.js)
- Editor de código (VSCode, outros)
- Terminal ou prompt de comando

1ª ETAPA: Configuração Inicial do Projeto

1.1 Criando o diretório do projeto

(no terminal de comando)

```shell
mkdir my-crud-app
cd my-crud-app
```

1.2 Inicializando o projeto
(no terminal de comando)

```shell
npm init -y
```

1.3 Instalando as dependências
(no terminal de comando)

```shell
npm install express
npm install cors
```

CORS: Finalidade
CORS, ou Compartilhamento de Recursos entre Origens, é um mecanismo de segurança implementado em navegadores web que permite que um site faça solicitações para um domínio diferente daquele de onde ele foi servido. Sem o CORS, os navegadores restringiriam essas solicitações entre origens devido à política de mesma origem, que visa proteger os usuários de sites maliciosos. O CORS permite que os servidores especifiquem quais origens podem acessar seus recursos, adicionando cabeçalhos HTTP às respostas para indicar quais domínios são permitidos.

CORS: Funcionamento

1. Política de mesma origem: por padrão, navegadores web restringem solicitações feitas por um site para um domínio diferente do seu próprio, para evitar ataques de sites maliciosos.
2. Solicitações entre origens: quando um site precisa acessar recursos de outro domínio, o navegador envia uma solicitação com cabeçalhos adicionais que informam ao servidor de destino sobre a origem da solicitação.

3. Cabeçalhos CORS: o servidor responde com cabeçalhos CORS, como Access-Control-Allow-Origin, que especificam quais origens são permitidas para acessar seus recursos.
4. Permissão ou bloqueio: se a origem da solicitação estiver na lista de origens permitidas, o navegador permite o acesso aos recursos; caso contrário, a solicitação é bloqueada e um erro é exibido;

1.4 Criando a estrutura de diretórios

(no terminal de comando)

```shell
mkdir -p src/controllers src/services src/repositories src/routes
```

2ª ETAPA: Implementando a Camada de Acesso aos Dados

2.1 Criando o arquivo *"src/repositories/userRepository.js"*

```javascript
// Simula um banco de dados em memória
let users = [];
let currentId = 1;

class UserRepository {

    create(user) {
        const newUser = { id: currentId++, ...user };
        users.push(newUser);
        return newUser;
    }

    findAll() {
        return users;
    }

    findById(id) {
        return users.find(user => user.id === id);
    }
```

```
    update(id, updatedUser) {
        const index = users.findIndex(user => user.id === id);
        if (index !== -1) {
            users[index] = {
                   ...users[index],
                   ...updatedUser,
                   id: id
            };
            return users[index];
        }
        return null;
    }

    delete(id) {
        const initialLength = users.length;
        users = users.filter(user => user.id !== id);
        // Retorna true se um usuário foi deletado
        return users.length < initialLength;
    }
}

module.exports = new UserRepository();
```

3ª ETAPA: Implementando a Camada de Serviço

3.1 Criando o arquivo *"src/services/userService.js"*

JavaScript

```
const userRepository = require("../repositories/userRepository");

class UserService {
    createUser(userData) {
        // Aqui poderia haver validações de negócio, como verificar se o email já existe
        if (!userData.name || !userData.email) {
            throw new Error("Nome e email são obrigatórios.");
        }
```

```
        return userRepository.create(userData);
    }

    getAllUsers() {
        return userRepository.findAll();
    }

    getUserById(id) {
        const user = userRepository.findById(parseInt(id));
        if (!user) {
            throw new Error("Usuário não encontrado.");
        }
        return user;
    }

    updateUser(id, userData) {
        // Validações de negócio antes de atualizar
        if (!userData.name && !userData.email) {
            throw new Error("Pelo menos um campo (nome ou email) deve ser fornecido para atualização.");
        }
        const updatedUser = userRepository.update(parseInt(id), userData);
        if (!updatedUser) {
            throw new Error("Usuário não encontrado para atualização.");
        }
        return updatedUser;
    }

    deleteUser(id) {
        const deleted = userRepository.delete(parseInt(id));
        if (!deleted) {
            throw new Error("Usuário não encontrado para exclusão.");
        }
        return { message: "Usuário deletado com sucesso." };
    }
}

module.exports = new UserService();
```

4ª ETAPA: Implementando a Camada de Controle

4.1 Criando o arquivo *"src/controllers/userController.js"*

JavaScript

```
const userService = require("../services/userService");

class UserController {
  async createUser(req, res) {
    try {
      const newUser = await userService.createUser(req.body);
      res.status(201).json(newUser);
    } catch (error) {
      res.status(400).json({ message: error.message });
    }
  }

  async getAllUsers(req, res) {
    try {
      const users = await userService.getAllUsers();
      res.status(200).json(users);
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
  }

  async getUserById(req, res) {
    try {
      const user = await userService.getUserById(req.params.id);
      res.status(200).json(user);
    } catch (error) {
      res.status(404).json({ message: error.message });
    }
  }

  async updateUser(req, res) {
    try {
      const updatedUser = await userService.updateUser(
        req.params.id,
        req.body
      );
      res.status(200).json(updatedUser);
    } catch (error) {
```

```
      res.status(400).json({ message: error.message });
    }
  }

  async deleteUser(req, res) {
    try {
      const result = await userService.deleteUser(req.params.id);
      res.status(200).json(result);
    } catch (error) {
      res.status(404).json({ message: error.message });
    }
  }
}

module.exports = new UserController();
```

5ª ETAPA: Definindo as Rotas

5.1 Criando o arquivo *"src/routes/userRoutes.js"*

JavaScript

```
const express = require("express");
const userController = require("../controllers/userController");

const router = express.Router();

router.post("/users", userController.createUser);
router.get("/users", userController.getAllUsers);
router.get("/users/:id", userController.getUserById);
router.put("/users/:id", userController.updateUser);
router.delete("/users/:id", userController.deleteUser);

module.exports = router;
```

6ª ETAPA: Configurando a Aplicação Express

6.1 Criando o arquivo *"src/app.js"*

JavaScript

```
const express = require("express");
const userRoutes = require("./routes/userRoutes");

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

// Habilita suporte ao CORS
app.use(cors());
// Middleware para passar JSON no corpo das requisições
app.use(express.json());

// Monta as rotas de usuário sob o prefixo /api
app.use("/api", userRoutes);

//  Middleware de tratamento de erros genérico (opcional, mas recomendado)
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send("Algo deu errado!");
});

module.exports = app;
```

7ª ETAPA: Criando Ponto de Entrada

7.1 Criando o arquivo (na raiz) *"index.js"*

JavaScript

```
const app = require("./src/app");

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

app.listen(PORT, () => {
  console.log(`Servidor rodando na porta ${PORT}`);
```

```
  console.log(`Acesse: http://localhost:${PORT}/api/users`);
});
```

8ª ETAPA: Executando e Testando a Aplicação

8.1 Iniciando o Servidor

(no terminal de comando)

```
Shell
node index.js
```

8.2 Testando os Endpoints

Criar Usuário

(no terminal de comando)

```
Shell
curl -X POST http://localhost:3000/api/users \
-H "Content-Type: application/json" \
-d '{"name": "João Silva", "email": "joao@email.com"}'
```

Listar Usuários

(no terminal de comando)

```
Shell
curl http://localhost:3000/api/users
```

```
g1l3du4rd0@g1l3du4rd0-K46CB /media/g1l3du4rd0/GEA/IFPR/
 % curl http://localhost:3000/api/users
[{"id":1,"name":"João Silva","email":"joao@email.com"}]
g1l3du4rd0@g1l3du4rd0-K46CB /media/g1l3du4rd0/GEA/IFPR/
 %
```

Buscar Usuário por ID
(no terminal de comando)

Shell
```
curl http://localhost:3000/api/users/1
```

```
g1l3du4rd0@g1l3du4rd0-K46CB /media/g1l3du4rd0/GEA/IFP
 % curl http://localhost:3000/api/users/1
{"id":1,"name":"João Silva","email":"joao@email.com"}
g1l3du4rd0@g1l3du4rd0-K46CB /media/g1l3du4rd0/GEA/IFP
 %
```

Atualizar Usuário
(no terminal de comando)

Shell
```
curl -X PUT http://localhost:3000/api/users/1 \
-H "Content-Type: application/json" \
-d '{"name": "João Santos", "email": "joao.santos@email.com"}'
```

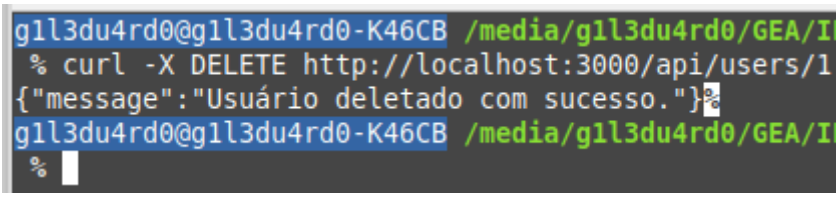
```
g1l3du4rd0@g1l3du4rd0-K46CB /media/g1l3du4rd0/GEA/IFPR/reposi
 % curl -X PUT http://localhost:3000/api/users/1 \
-H "Content-Type: application/json" \
-d '{"name": "João Santos", "email": "joao.santos@email.com"}
{"id":1,"name":"João Santos","email":"joao.santos@email.com"}
g1l3du4rd0@g1l3du4rd0-K46CB /media/g1l3du4rd0/GEA/IFPR/reposi
 %
```

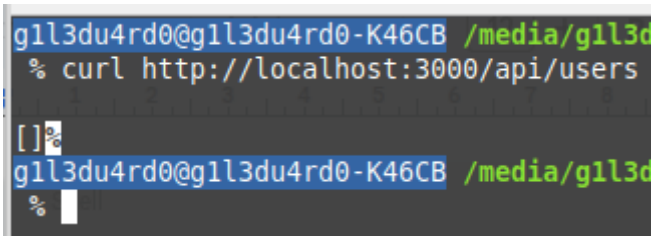
```
g1l3du4rd0@g1l3du4rd0-K46CB /media/g1l3du4rd0/GEA/IFPR/reposito
 % curl http://localhost:3000/api/users

[{"id":1,"name":"João Santos","email":"joao.santos@email.com"}]
g1l3du4rd0@g1l3du4rd0-K46CB /media/g1l3du4rd0/GEA/IFPR/reposito
 %
```

Remover Usuário
(no terminal de comando)

Shell
```
curl -X DELETE http://localhost:3000/api/users/1
```





Estrutura Final do Projeto

Textproto
```
my-crud-app/
├── src/
│   ├── controllers/
│   │   └── userController.js
│   ├── services/
│   │   └── userService.js
│   ├── repositories/
│   │   └── userRepository.js
│   ├── routes/
│   │   └── userRoutes.js
│   └── app.js
├── index.js
├── package.json
└── package-lock.json
```