

# **Apostila de Introdução ao Node.JS**

**Versão 6.2**

***Profª Mª Denilce Veloso***

**Sorocaba/SP  
Setembro/2024**

## **ÍNDICE**

1.	O que é NODE.JS.....	7
1.1	Histórico.....	7
1.2	Definição.....	7
1.3	Aplicações do Node .....	8
1.4	Outras opções .....	9
1.5	Modelo Tradicional x Modelo Node.js .....	9
1.6	Elementos que compõem o Node.js.....	11
1.7	Outras opções de utilização do Node.js .....	14
2.	Instalação Node.js .....	14
3.	Exercícios.....	16
3.1	Primeiro exercício.....	16
3.2	Exercício 2 .....	18
3.3	Exercício 3 .....	19
3.4	Exercício 4 .....	19
3.5	Exercício 5 .....	24
4.	Respondendo requisições HTTP .....	26
5.	Respondendo requisições usando a URL .....	28
6.	Ferramentas para auxiliar no Desenvolvimento .....	29
6.1	NPM.....	30
6.2	Instalação do Express .....	32
6.3	Instalação do EJS .....	33
6.4	Instalação do Nodemon .....	34
7.	Utilizando as novas ferramentas.....	35
7.1	Guardando os exercícios anteriores.....	35
7.2	Utilizando o Express .....	36
7.3	Utilizando o EJS .....	39
7.4	Testando o Nodemon.....	43
7.5	Organizando melhor a aplicação .....	44
7.5.1	Criando pasta views (guardando views anterior).....	44
7.5.2	Módulos e CommonJS.....	47
7.5.3	Modularizando a aplicação .....	49
7.5.3.1	Separando as configurações do servidor .....	49
7.5.3.2	Separando as configurações das rotas.....	52

8. Banco de Dados.....	55
8.1 Instalação do SQL Server .....	55
8.2 Instalação do driver mssql.....	56
8.3 Usando o SQL Server no Management Studio .....	56
8.4 Usando o SQL Server no Prompt.....	58
8.5 Listando dados de uma tabela na página.....	59
8.6 Listando dados de uma tabela na página (através da view) .....	62
8.7 Alterando a forma de conexão com o banco SQL Server.....	64
9. Separando a aplicação em Camadas .....	66
9.1 MVC.....	66
9.1.1 Model ou Modelo.....	66
9.1.2 Controller ou Controlador.....	67
9.1.3 View ou Visão .....	67
10. Melhorando a organização das rotas.....	67
10.1 Consign .....	67
10.2 Instalação do Consign.....	68
10.3 Incluir o Consign no server.js.....	68
10.4 Restruturação da parte do banco de dados.....	71
10.5 Criando uma página para recuperar pelo ID.....	77
11 Implementando os Models .....	80
12 Criação do formulário de Inclusão .....	84
12.1 Body-Parser .....	87
12.1.1 Instalação do Body-Parser.....	87
12.1.2 Alteração do server e model para inserir no banco de dados .....	88
13 Crud completo da aplicação.....	93
14. Exercício de Criação de API Rest Local .....	106
15. Exercício de Consumo de API Rest através do pacote .....	111
16. Extras – Encoding .....	115
Referências.....	119

## **Índice de Figuras**

Figura 1: Modelo Tradicional x Modelo Node.Js.....	10
Figura 2: Arquitetura do Node.Js .....	11
Figura 3: Site nodejs.org.....	14
Figura 4: Instalação do arquivo do Node.Js.....	15
Figura 5: Versão do Node.Js .....	16
Figura 8: Primeiro.js .....	17
Figura 9: Executando Primeiro.Js .....	17
Figura 10: Executando segundoS.js.....	18
Figura 11: Executando segundoA.js .....	18
Figura 12: Execução do Exercício 3 .....	19
Figura 13: Arquivo file.txt.....	20
Figura 14: Execução do QuartoS .....	20
Figura 15: Execução do callback1.js .....	22
Figura 16: Execução do callback2.js .....	22
Figura 17: Execução do QuartoA.....	23
Figura 18: Execução do Exercício quinto.js .....	26
Figura 19: Execução do exercicio6.js.....	27
Figura 20: Execução do exercicio6.js (chamando o browser) .....	28
Figura 21: Execução do exercicio7.js (chamando no browser a opção historia) .....	29
Figura 22: Versão do NPM.....	30
Figura 23: Execução npm init .....	30
Figura 24: Arquivo package-json .....	31
Figura 25: Pasta node_modules .....	33
Figura 26: Pasta node_modules\ejs .....	34
Figura 27: Teste nodemon digitando npm start no terminal.....	35
Figura 28: Pasta ExerciciosAnteriores .....	35
Figura 29: exercicio7.js e app.js.....	36
Figura 30: Teste do servidor com express.....	36
Figura 31: Teste local do servidor .....	37
Figura 32: Teste local.....	38
Figura 33: Teste da opção Cursos.....	38
Figura 34: Pasta views e seção .....	40
Figura 35: Seção historia .....	42
Figura 36: Teste do Nodemon.....	43
Figura 37: Teste do site principal .....	43
Figura 38: Novo Teste do site principal.....	44
Figura 45: Nova Pasta views.....	45
Figura 46: Estrutura Módulo .....	47
Figura 47: Teste chamando o módulo.....	48
Figura 48: Teste chamando a página.....	49
Figura 49: Teste chamando o módulo usando nodemon .....	49
Figura 50: Pasta config .....	49
Figura 51: Cópia da pasta views .....	51

Figura 52: Tela console iniciando servidor .....	51
Figura 53: Tela console iniciando servidor .....	51
Figura 54: Pasta routes.....	52
Figura 55: Tela console iniciando servidor .....	54
Figura 56: Tela página professores .....	54
Figura 57: Site download do SQL Server .....	55
Figura 58: Módulos do SQL Server .....	56
Figura 59: Tela de Login do Management Studio .....	56
Figura 60: Interface do Management Studio .....	57
Figura 61: Testando o SqlServer no console .....	58
Figura 62: Testando a página dos professores (listar registros) .....	61
Figura 63: Testando a página dos professores (usando view dinâmica).....	64
Figura 64: Novo arquivo dbConnection para acesso ao banco.....	64
Figura 65: Estrutura MVC.....	66
Figura 66: node_modules com o Consign .....	68
Figura 67: Consign incluindo a pasta routes .....	69
Figura 68: Consign incluindo a pasta routes mesmo sem estarem no app.js.....	70
Figura 69: Página principal depois do Consign.....	70
Figura 70: Página história depois do Consign.....	71
Figura 71: Página professores depois do Consign (sem “startar” banco de dados) .....	71
Figura 71: Recarregando servidor com autoload carregando routes .....	75
Figura 72: Página principal com autoload carregando routes .....	75
Figura 73: Página história com autoload carregando routes .....	76
Figura 74: Rota detalhaprofessor.js .....	77
Figura 75: View detalhaprofessor.ejs.....	78
Figura 76: Servidor carregando rota detalhaprofessor.js .....	79
Figura 77: Teste página professores.....	79
Figura 78: Teste página detalhaprofessor.....	80
Figura 79: Criação pasta models .....	81
Figura 80: Servidor carregando a pasta models.....	83
Figura 81: Página dos professores com servidor carregando a pasta models .....	83
Figura 82: Página detalhaprofessor com servidor carregando a pasta models .....	84
Figura 83: Rota adicionar_professor.js.....	84
Figura 84: Arquivo (página) adicionar_professor.ejs .....	85
Figura 85: Página adicionar_professor.....	86
Figura 86: Retorno do Enviar (Salvar) professor .....	87
Figura 87: Body-Parser no node_modules .....	88
Figura 88: Carregando página adicionar_professor .....	89
Figura 89: Retorno do Salvar de adicionar_professor.....	89
Figura 90: Servidor recarregado .....	92
Figura 91: Página adicionar_professor .....	92
Figura 92: Página dos professores.....	93
Figura 93: Página admin/crud_professores .....	104
Figura 94: Executando server.js .....	107
Figura 95: Pesquisando End-Point.....	108
Figura 96: Executando testeapi.html .....	110
Figura 97: Atualizando End-Point.....	110
Figura 96: Executando consapi.js .....	112

Figura 39: Preferences > Settings.....	116
Figura 40: Indicação do encoding.....	116
Figura 41: Modificação das configurações do usuário.....	116
Figura 42: Salvando o arquivo no formato UTF-8 .....	117
Figura 43: Navegando em configurações .....	117
Figura 44: Indo na opção UTF-8 .....	118

## Introdução ao Node.Js

### 1. O que é NODE.JS

#### 1.1 Histórico

O Node.js foi desenvolvido por Ryan Dahl, em 2009, essencialmente como uma forma de rodar (interpretar) programas JavaScript fora do contexto de um browser, o Node.js foi desenvolvido em C++.

Essa nova abordagem fez nascer uma nova técnica para criação de aplicações web onde com apenas uma linguagem de programação, é possível criar tantos scripts para o *Front-End* quanto para o *Back-End*.

#### 1.2 Definição

O Node.js é *runtime* (refere-se ao ambiente de execução no qual um código é interpretado e executado), é uma plataforma construída sobre o motor JavaScript<sup>1</sup> do Google Chrome (V8) para facilmente construir aplicações de rede rápidas e escaláveis. Usa um modelo de I/O direcionada a evento não bloqueante (figura 1) que o torna leve e eficiente, ideal para aplicações em tempo real com troca intensa de dados através de dispositivos distribuídos.

O JavaScript é uma linguagem interpretada, o que o coloca em desvantagem quando comparado com linguagens compiladas, pois cada linha de código precisa ser interpretada enquanto o código é executado. O V8 compila o código para linguagem de máquina, além de otimizar drasticamente a execução usando heurísticas (procedimentos, estratégias), permitindo que a execução seja feita em cima do código compilado e não interpretado.

O Node.js é construído com as novas versões do V8. Mantendo-se em dia com as últimas atualizações desta *engine*, as novas funcionalidades da especificação JavaScript ECMA-262 são trazidas para os desenvolvedores Node.js em tempo hábil, assim como as melhorias contínuas de performance e estabilidade (novas versões do ES).<sup>2</sup>Todas as funcionalidades em lançamento (*shipping*), que o V8

---

<sup>1</sup> Um motor JavaScript é um componente fundamental em navegadores da web que interpreta e executa código JavaScript. Ele converte o código escrito em JavaScript em instruções que o computador pode entender e executar.

<sup>2</sup> <https://nodejs.org/pt-br/docs/es6/>

considera estáveis, são ativadas por padrão no Node.js e NÃO necessitam de nenhum tipo de flag de tempo de execução.

O Node.js é open-source e multiplataforma e permite aos desenvolvedores criarem todo tipo de aplicativos e ferramentas do lado do servidor (*Back-End*) em JavaScript. O Node.js é usado fora do contexto de um navegador, ou seja, executado diretamente no computador ou no servidor. Como tal, o ambiente omite APIs JavaScript específicas do navegador e adiciona suporte para APIs de sistema operacional mais tradicionais, incluindo bibliotecas de sistemas HTTP e arquivos.<sup>3</sup>

### 1.3 Aplicações do Node

O Node.js pode ser utilizado para:

- Criar servidores web (ex. Amazon, Mercado livre etc. usam para criar servidores);
- Criar APIs (Interfaces de Programação de Aplicações);
- Manipulação de Pacotes e Dependências (usa npm);
- Desenvolvimento de micro serviços (ex. cada serviço é responsável por uma funcionalidade específica e se comunica com os outros através de APIs);
- Desenvolvimento de aplicações em tempo real (ex. criar chats, jogos etc.);
- Alto fluxo input/output de dados via rede (ex. criar servidores de streaming e vídeos);
- Middleware<sup>4</sup> (ex. implementar autenticação e autorização de usuários);
- IoT;
- Streaming (ex. aplicativos que transmitem dados em tempo real – ex. financeiro).

Uma API (Interface de Programação de Aplicações) é um conjunto de regras e especificações que define como diferentes softwares podem se comunicar e interagir, é o contrato da comunicação.

---

<sup>3</sup> <https://nodejs.org/pt-br/about/>

<sup>4</sup> Middleware refere-se a software que atua como uma camada intermediária entre diferentes componentes de software ou sistemas, facilitando a comunicação e a troca de dados entre eles.



Os microserviços são uma abordagem arquitetural que divide uma aplicação monolítica em pequenos serviços independentes, cada um responsável por uma funcionalidade específica.

Cada microserviço geralmente expõe uma API: Essa API permite que outros serviços ou aplicações externas se conectem e utilizem as funcionalidades oferecidas por aquele microserviço.

APIs podem ser usadas para outros fins além de conectar microserviços: Elas podem ser utilizadas para integrar diferentes sistemas legados, fornecer acesso a dados para aplicativos móveis etc.

#### *1.4 Outras opções*

Deno (2018) - é um ambiente de execução para JavaScript e TypeScript baseado no mecanismo JavaScript V8 e na linguagem de programação Rust. Foi criado por Ryan Dahl, criador original do Node.js, e é focado na produtividade. O Deno ainda está em desenvolvimento, mas está ganhando popularidade entre os desenvolvedores. Uma das diferenças em relação ao Node.js é que ele dá suporte nativo aos módulos do TypeScript.

Bun (2021) - é um toolkit rápido e completo para executar, construir, testar e depurar JavaScript e TypeScript, seja apenas um único arquivo ou um aplicações full-stack.

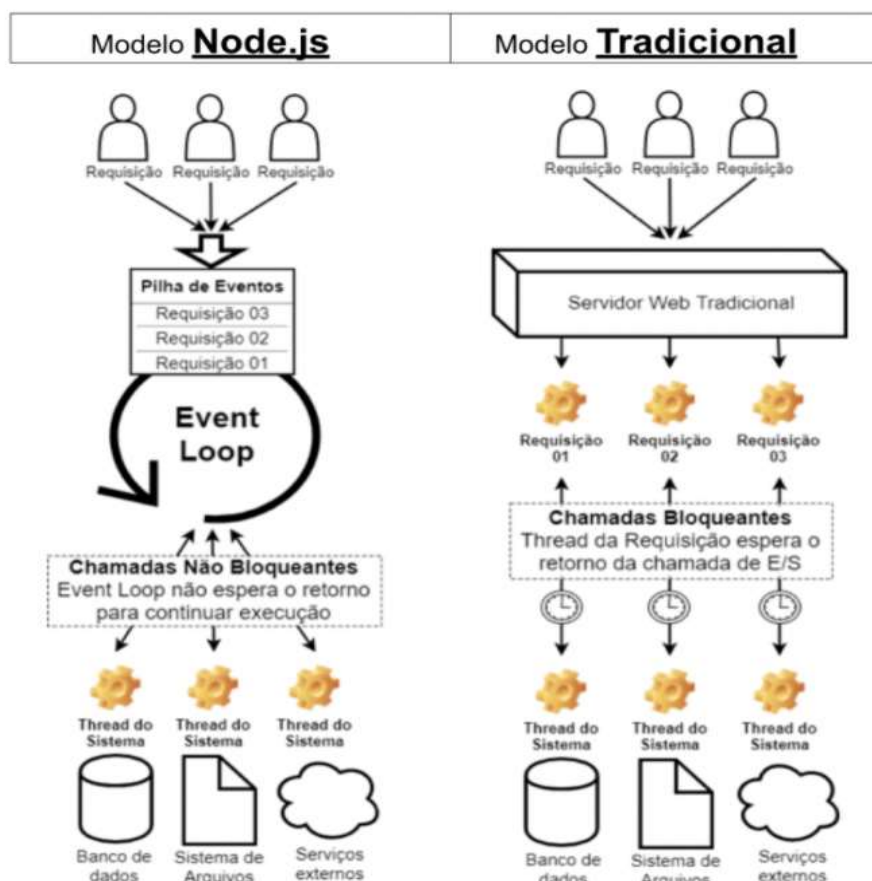
#### *1.5 Modelo Tradicional x Modelo Node.Js*

Em um servidor web utilizando linguagens tradicionais, para cada requisição recebida é criada uma nova thread para tratá-la (figura 1). A cada requisição, serão demandados recursos computacionais (memória RAM, por exemplo) para a criação dessa nova *thread*. Uma vez que esses recursos são limitados, as threads não serão criadas infinitamente, e quando esse limite for atingido, as novas requisições terão que esperar a liberação desses recursos alocados para serem tratadas.

O Node.Js foi pensado em ser escalável, e se destaca pelo fato de trabalhar de forma assíncrona utilizando todas as APIs nativas com I/O (input e output) não bloqueante afirmando mais uma vez o poder de desempenho da tecnologia. Todo processamento acontece através de um event loop e uma única thread que fica capturando as requisições e juntamente com a sua API não bloqueante

consegue processar diversas requisições ao mesmo tempo e respondendo com as que acabar em primeiro. Não existe um encadeamento pois nem sempre o primeiro a entrar será o primeiro a sair, tudo dependerá do nível de processamento da requisição.

Figura 1: Modelo Tradicional x Modelo Node.js<sup>5</sup>



*Thread* → é um pequeno programa que trabalha como um subsistema, sendo uma forma de um processo se autodividir em duas ou mais tarefas (ordem de execução).<sup>6</sup> Ela executa o *event loop*.

*Single Thread* -> Qual seria a vantagem de limitar a execução da aplicação em somente um *thread*? Linguagens como Java, PHP e Ruby seguem um modelo

<sup>5</sup> <https://www.opus-software.com.br/node-js/>

<sup>6</sup> Em programação cada thread possui seu próprio contador de programa, pilha de execução e contexto de registro. Elas compartilham recursos do processo, como memória, arquivos abertos e outros recursos do sistema operacional. O próprio Windows utiliza threads para várias finalidades, como executar operações de entrada e saída, lidar com eventos, responder a interações do usuário, executar cálculos em segundo plano e muito mais. Ao utilizar threads, o Windows pode realizar várias tarefas de forma eficiente e responsiva, distribuindo o tempo de CPU entre os processos e threads em execução.

onde cada nova requisição roda em um *thread* separada do sistema operacional. Esse modelo é eficiente, mas tem um custo de recursos muito alto e nem sempre é necessário todo o recurso computacional aplicado para executar um novo *thread*. O Node.js foi criado para solucionar esse problema, usar programação assíncrona e recursos compartilhados para tirar maior proveito de um *thread*.

*Call Stack* → A *Stack* (pilha) é um conceito bem comum no mundo das linguagens de programação, já ouviu “estouro de pilha”? No *Node.js* e no JavaScript, em geral, esse conceito não se difere muito de outras linguagens. Sempre que uma função é executada, ela entra na *stack*, que executa somente uma coisa por vez, ou seja, o código posterior ao que está rodando precisa esperar a função atual terminar de executar para seguir adiante.

*Multi Threading* → Na verdade, quem é *single thread* é o V8, o motor do Google utilizado para rodar o *Node.js*. Para que seja possível executar tarefas assíncronas, o *Node.js* conta com diversas outras APIs – algumas delas providas pelos próprios sistemas operacionais, como é o caso de eventos de disco, *sockets TCP* (*Transmission Control Protocol*) e *UDP* (*User Datagram Protocol*). Quem toma conta dessa parte de I/O assíncrono, de administrar múltiplas *threads* e enviar notificações é a *libuv*.

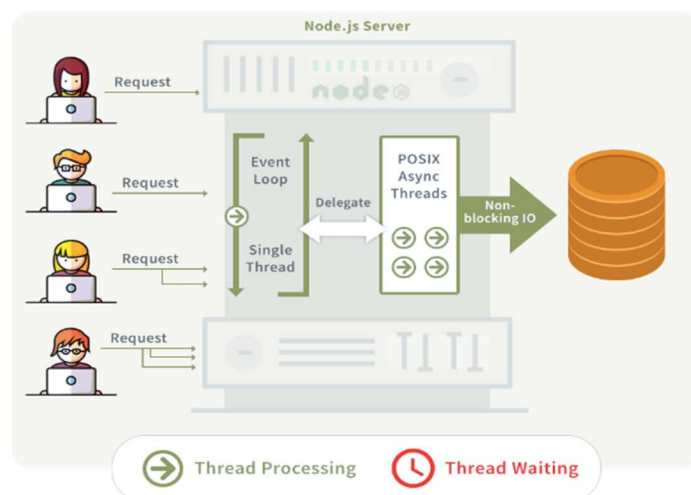
## 1.6 Elementos que compõem o Node.js

Na figura 2 estão os elementos que compõe o Node.js.

*Figura 2: Arquitetura do Node.js<sup>7</sup>*

---

<sup>7</sup> Fonte: <https://blog.schoolofnet.com/como-comecar-com-node-js/>



V8: É a engine de código aberto de alto desempenho para JavaScript e WebAssembly do Google, escrito em C ++. É usado no Chrome e no Node.js, entre outros. Ele implementa ECMAScript e WebAssembly e é executado no Windows 7 ou posterior, macOS 10.12+ e sistemas Linux que usam processadores x64, IA-32, ARM ou MIPS. O V8 pode ser executado de forma autônoma ou pode ser incorporado a qualquer aplicativo C ++.<sup>8</sup>

*Event Loop* → O Node.js é guiado (orientado) por eventos, termo também conhecido como *Event Driven*. Esse conceito já é bastante aplicado em interações com interface de usuário. O JavaScript possui diversas APIs baseadas em eventos para interações com o *DOM* (*onClick*, *onHide*, *onShow*, etc), que são muito comuns no mundo front-end com JavaScript. *Event driven* (“fica escutando os eventos”) se refere ao estilo de programação baseado em eventos e call-backs (função passada para outra função como argumento), onde as operações de I/O são tratadas de forma assíncrona, permitindo que o programa continue executando outras tarefas enquanto aguarda eventos ou operações assíncronas serem concluídas.

Libuv: É uma biblioteca de suporte multiplataforma com foco em entrada e saída assíncrona. Ele foi desenvolvido principalmente para uso por Node.js, mas também é usado por Luvit<sup>9</sup>, Julia, pyuv e outros.<sup>10</sup>

<sup>8</sup> <https://v8.dev>

<sup>9</sup> Plataformas de e-learning

<sup>10</sup> <http://docs.libuv.org/en/v1.x/>

HTTP-parser: É um analisador para mensagens HTTP (*Hypertext Transfer Protocol*) (usado para transferir dados na web, define a estrutura dos dados) escritas em C. Ele analisa solicitações e respostas. O analisador é projetado para ser usado em aplicativos HTTP de desempenho. Ele não faz chamadas nem alocações, não armazena dados em buffer, pode ser interrompido a qualquer momento. Dependendo da sua arquitetura, ele requer apenas cerca de 40 bytes de dados por fluxo de mensagens (em um servidor web por conexão).<sup>11</sup>

C-ares: É uma biblioteca C para solicitações assíncronas de DNS (*User Datagram Protocol*) (faz a tradução dos nomes de domínio em IPs e vice-versa). Compatibilidade com C89, licenciado pelo MIT, é construído e executado em POSIX, Windows, Netware, Android e muitos outros sistemas operacionais.<sup>12</sup>

OpenSSL: É um kit de ferramentas robusto, de nível comercial e completo para os protocolos TLS (*Layer Security*) e SSL (*Secure Sockets Layer*), protocolos de segurança e autenticação.<sup>13</sup>

zlib: O zlib foi projetado para ser uma biblioteca de compactação de dados sem perdas, gratuita e de uso geral, legalmente desimpedida (ou seja, não coberta por nenhuma patente) para uso em virtualmente qualquer hardware de computador e sistema operacional. O próprio formato de dados zlib é portátil entre plataformas. O método de compactação atualmente usado em zlib essencialmente nunca expande os dados. A área de cobertura da memória do zlib também é independente dos dados de entrada e pode ser reduzida, se necessário, com algum custo na compactação.<sup>14</sup>

Posix → A interface do sistema operacional portátil para ambientes de computação é um conjunto de padrões e especificações que definem maneiras

---

<sup>11</sup> <https://github.com/nodejs/http-parser/blob/master/README.md>

<sup>12</sup> <https://c-ares.haxx.se>

<sup>13</sup> <https://www.openssl.org>

<sup>14</sup> <https://zlib.net>

de os programas de computador interagirem com um sistema operacional, apesar da base dele ter sido o Unix.

### *1.7 Outras opções de utilização do Node.js*

É possível utilizar o Node.js em um ambiente online: Existem várias plataformas online que oferecem ambientes de desenvolvimento integrados (IDEs) ou máquinas virtuais pré-configuradas para desenvolvimento Node.js. Alguns exemplos são o Replit (replit.com) e o Glitch (glitch.com).

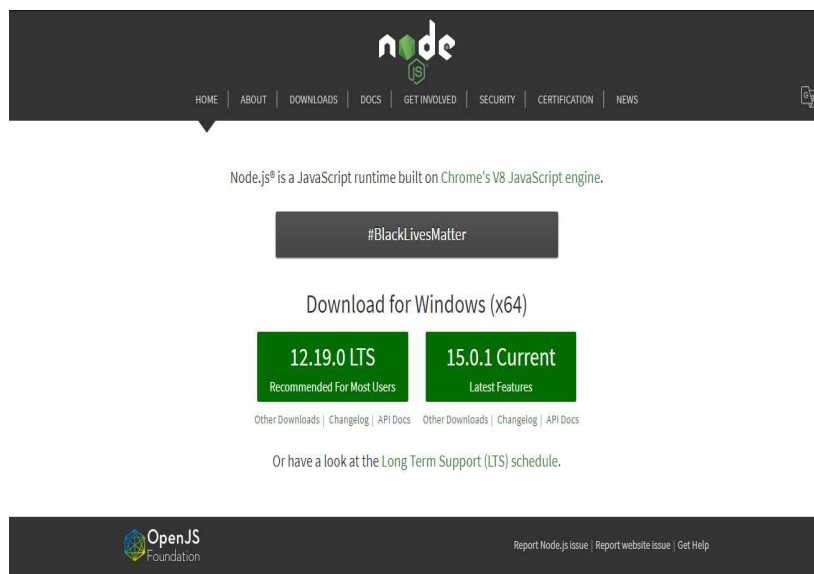
Um outra opção é utilizar serviços em nuvem, como o Amazon Web Services (AWS), Google Cloud Platform (GCP) ou Microsoft Azure. Esses serviços permitem que se criem máquinas virtuais ou ambientes de desenvolvimento na nuvem, onde é possível instalar e executar o Node.js sem depender da infraestrutura local.

É também é possível executar o Node.js em um ambiente portátil. Essas versões são executáveis autônomos que se pode copiar para uma pasta em ambiente de trabalho e executar diretamente. No entanto, algumas funcionalidades do Node.js podem depender de bibliotecas externas que podem não estar disponíveis nesse ambiente

## **2. Instalação Node.js**

Para efetuar a instalação do Node.js, o primeiro passo é acessar o site referente a tecnologia: [nodejs.org](https://nodejs.org)

*Figura 3: Site [nodejs.org](https://nodejs.org)*



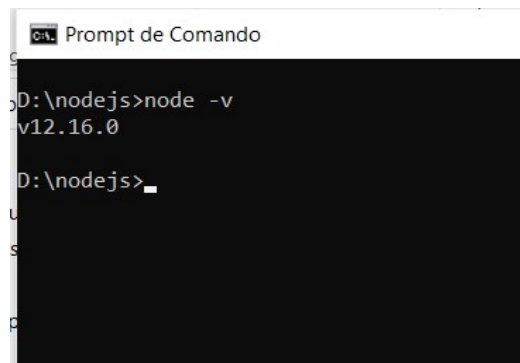
Com o site aberto, verificar a versão compatível com a máquina a ser instalada e selecionar a opção LTS, que como o próprio site indica é recomendável para a maior parte dos usuários por ser mais estável. Após o download, abra o arquivo baixado e execute-o.

*Figura 4: Instalação do arquivo do Node.js*



Após a instalação, pode-se confirmar se está tudo ok, basta abrir o prompt de comando (cmd) e digitar o comando: node -v.

*Figura 5: Versão do Node.Js*



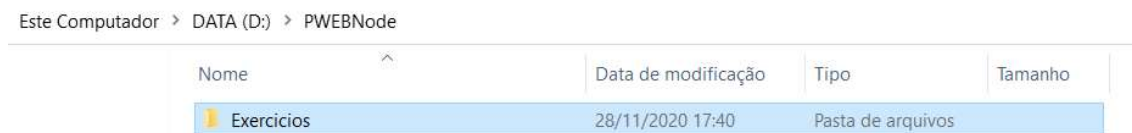
### **3. Exercícios**

A seguir são mostrados alguns exercícios no Node.Js:

#### *3.1 Primeiro exercício*

Antes dos exercícios, criar uma pasta chamada PWEBNode e dentro dela outra pasta chamada Exercícios, onde serão gravados os exercícios realizados.





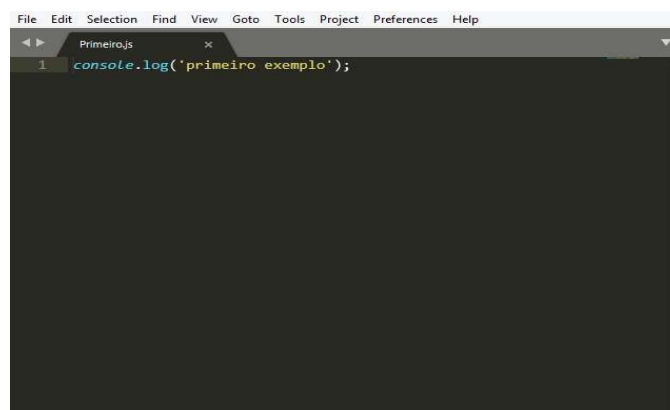
Com o Node.js e o editor de sua preferência instalado (pode ser Visual Studio Code, Sublime Text etc.), criar o primeiro exercício para mostrar uma mensagem no console.

Abrir o editor e nele escreva o seguinte comando:

```
console.log('primeiro exemplo');
```

Salvar como Primeiro.js na pasta PWebNode\Exercicios

*Figura 8: Primeiro.js*



Por questões práticas, pelo cmd acesse a pasta PWebNode\Exercícios. Para acessar o cmd, pode-se digitar cmd na barra do Explorer ou na barra de pesquisa, nessa última vai precisar de mais comandos para abrir a pasta desejada, como CD.

Quando estiver dentro da pasta exercícios, execute o comando: node Primeiro.js

*Figura 9: Executando Primeiro.js*

```
D:\PWEBNode\Exercicios>node primeiro.js  
primeiro exemplo  
  
D:\PWEBNode\Exercicios>
```

### 3.2 Exercício 2

Nesse exercício será testada a diferença entre um código **síncrono** e outro **assíncrono**.

Digitar o código para o arquivo segundoS.js (S de síncrono)

```
console.log('1');  
t();  
console.log('3');  
function t() {  
  console.log('2');  
}
```

Digitar o código para o arquivo segundoA.js (A de assíncrono)

```
console.log('1');  
t();  
console.log('3');  
function t() {  
  setTimeout(function() {  
    console.log('2');  
  }, 10);  
}
```

Testar os dois códigos no prompt.

*Figura 10: Executando segundoS.js*

```
C:\PWEBNode\Exercicios>node segundoS  
1  
2  
3
```

*Figura 11: Executando segundoA.js*

```
C:\PWEBNode\Exercicios>node segundoA  
1  
3  
2
```

### 3.3 Exercício 3

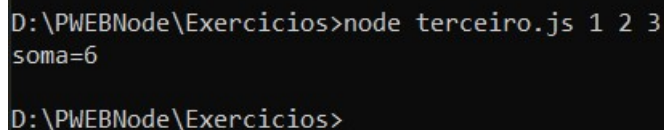
Nesse exercício será mostrado como acessar os argumentos da linha de comando através do objeto global process. O objeto process possui uma propriedade argv, que é um array contendo a linha de comando completa. Por exemplo: process.argv.

Digitar o código para somar os valores recebidos via argumentos (parâmetros) para o arquivo terceiro.js

```
var soma = 0;
for (var i=2; i<=process.argv.length-1; i++)
    soma=soma+Number(process.argv[i]);
console.log("soma="+soma);
```

Testar o arquivo terceiro.js passando parâmetros.

Figura 12: Execução do Exercício 3



```
D:\PWEBNode\Exercicios>node terceiro.js 1 2 3
soma=6

D:\PWEBNode\Exercicios>
```

### 3.4 Exercício 4

Nesse exercício será mostrado como utilizar o sistema de arquivos (*filesystem*) para ler e imprimir linhas.

Para realizar essa operação no sistema de arquivos (*filesystem*), será necessário incluir o módulo fs da *library* principal do Node.js. Para carregar esse tipo de módulo ou qualquer outro módulo "global", use o seguinte código:

```
var fs = require('fs');15
```

O comando require permite incorporar outros arquivos ao arquivo corrente, podendo importar bibliotecas, outras páginas etc.

---

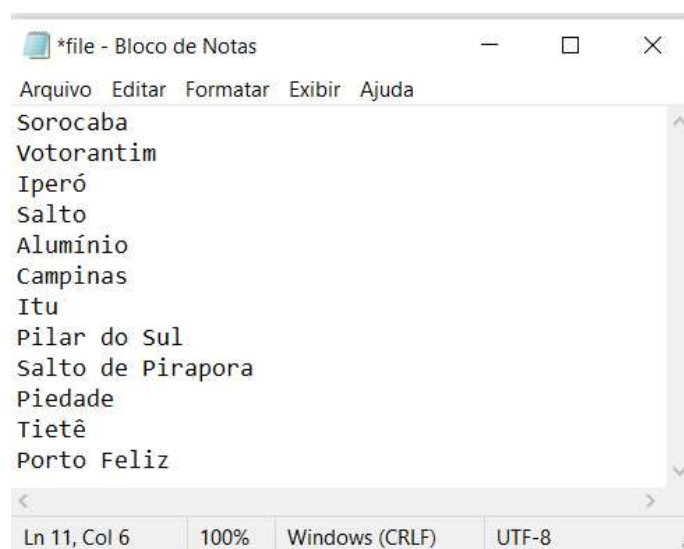
<sup>15</sup> <https://nodejs.org/api/fs.html>

Todos os métodos de sistema de arquivos síncronos (ou bloqueantes) no módulo fs terminam com 'Sync'. Para ler um arquivo, usar `fs.readFileSync('caminho/do/arquivo')`. Esse método irá retornar um objeto Buffer contendo o conteúdo completo do arquivo.

Objetos Buffer são a maneira do Node.js representar eficientemente arrays arbitrários de dados, sejam eles ascii, binários ou quaisquer outros formatos. Objetos Buffer podem ser convertidos em strings invocando o método `toString()` neles. Por exemplo: `var str = buf.toString()`.<sup>16</sup>

Criar um arquivo `file.txt` para ser lidos nos arquivos `quartoA.js` e `quartoS.js`.

*Figura 13: Arquivo file.txt*



Digitar o código `quartoS.js`

```
const fs = require('fs');
const data = fs.readFileSync('file.txt');
// a execução é bloqueada aqui até o arquivo ser lido
console.log(data.toString());
```

Testar o arquivo `quartoS.js`

*Figura 14: Execução do QuartoS*

<sup>16</sup> <https://nodejs.org/api/buffer.html>

```
D:\PWEBNode\Exercicios>node quartoS.js
Sorocaba
Votorantim
Iperó
Salto
Alumínio
Campinas
Itu
Pilar do Sul
Salto de Pirapora
Piedade
Tietê
Porto Feliz
```

Ao invés de `fs.readFileSync()` usar `fs.readFile()` e ao invés de usar o valor de retorno desse método, será coletado o valor de uma função de callback<sup>17</sup> que será passada como o segundo argumento.

Exemplos de funções callbacks:

Inicialmente, abra o terminal e instale o seguinte pacote:

```
npm install prompt-sync
```

Após instalar o pacote podemos executar o exemplo abaixo.

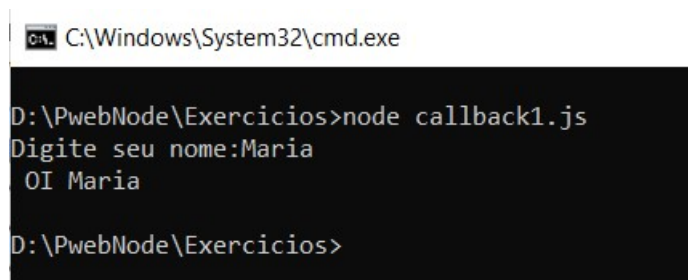
---

<sup>17</sup> callback é um tipo de função que só é executada após o processamento de outra função. Na linguagem JavaScript, quando uma função é passada como um argumento de outra, ela é, então, chamada de callback. Isso é importante porque uma característica dessa linguagem é não esperar o término de cada evento para a execução do próximo. Portanto, ela contribui para controlar melhor o fluxo de processamento assíncrono.

### Exemplo 1: callback1.js

```
const prompt = require('prompt-sync')();  
  
// parênteses indicam que estamos executando a função prompt-sync. Ao fazer //isso, a função retorna um valor, que é  
// uma nova função que pode ser usada //para criar prompts.  
// não esquecer de instalar  
// npm install prompt-sync  
function saudacao(nome) {  
  console.log(' Oi ' + nome);  
}  
  
function entradaNome(callback) {  
  var nome = prompt('Digite seu nome:');  
  callback(nome); // chamando a função callback (saudação)  
}  
  
entradaNome(saudacao);  
  
// obter o nome do usuário através de uma caixa de diálogo e, em seguida, chamar a função de retorno  
// (callback) fornecida como parâmetro, passando o nome digitado como argumento.
```

*Figura 15: Execução do callback1.js*

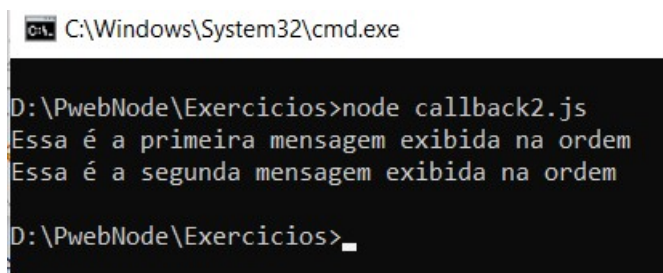


```
C:\Windows\System32\cmd.exe  
  
D:\PwebNode\Exercicios>node callback1.js  
Digite seu nome:Maria  
OI Maria  
  
D:\PwebNode\Exercicios>
```

### Exemplo 2: callback2.js

```
function exibeMensagensNaOrdem(mensagem, callback) {  
  console.log(mensagem);  
  callback();  
}  
  
//executando  
exibeMensagensNaOrdem('Essa é a primeira mensagem  
exibida na ordem', function() {  
  console.log('Essa é a segunda mensagem exibida na ordem');  
});
```

*Figura 16: Execução do callback2.js*



```
C:\Windows\System32\cmd.exe

D:\PwebNode\Exercicios>node callback2.js
Essa é a primeira mensagem exibida na ordem
Essa é a segunda mensagem exibida na ordem

D:\PwebNode\Exercicios>
```

As tradicionais callbacks do Node.js normalmente têm a assinatura:

```
function callback (err, data) { /* ... */ }
```

Será necessário **checar se um erro ocorreu** **checando se o primeiro argumento é verdadeiro**. Se não houver nenhum erro, pode se ter o objeto Buffer como segundo argumento.

Digitar o código no arquivo quartoA.js.

```
// Função para imprimir a primeira parte
function Parte1() {
  for (var i = 1; i <= 10; i++) {
    console.log("primeira parte:" + i);
  }
}

setTimeout(Parte1, 2000); // para atrasar
const fs = require('fs');
fs.readFile('file.txt', (err, data) => {
  if (err) throw err;
  const registros = data.toString().split('\n');18
  registros.forEach((registro, index) => {
    console.log("segunda parte:" + registro);
  });
});
```

Testar o arquivo quartoA.js

*Figura 17: Execução do QuartoA*

<sup>18</sup> A função `split()` divide uma string em um array de strings, usando o delimitador especificado como argumento. No caso deste código, o delimitador é `\n`, que é o caractere de nova linha. A função callback é anônima e recebe `err` e `data`. Depois que lê passa para a callback.

```
C:\PWEBNodeM\Exercicios>node quartoa
segunda parte:Sorocaba
segunda parte:Votorantim
segunda parte:Iperó
segunda parte:Salto
segunda parte:Alumínio
segunda parte:Campinas
segunda parte:Itu
segunda parte:Pilar do Sul
segunda parte:Salto de Pirapora
segunda parte:Piedade
segunda parte:Tietê
segunda parte:Porto Feliz
primeira parte:1
primeira parte:2
primeira parte:3
primeira parte:4
primeira parte:5
primeira parte:6
primeira parte:7
primeira parte:8
primeira parte:9
primeira parte:10
```

No exemplo `quartoS.js` apesar de ser mais enxuto tem um código bloqueando execução de qualquer JavaScript adicional até que todo o arquivo seja lido. Na versão `quartoA.js` assíncrona, a execução é *single threaded*. Então a concorrência é referente somente à capacidade do event loop de executar funções de callback depois de completar qualquer outro processamento. Qualquer código que pode rodar de maneira concorrente deve permitir que o event loop continue executando enquanto uma operação não-JavaScript, como I/O, está sendo executada.

### 3.5 Exercício 5

Nesse exercício será mostrado como são criados os eventos no lado do servidor. Os eventos do lado do servidor não têm nada a ver com os eventos Javascript que já são conhecidos e utilizados nas aplicações web do lado do cliente. Aqui os eventos são produzidos no servidor e podem ser de diversos tipos dependendo das bibliotecas ou classes que estejam trabalhando. Por exemplo, em um servidor HTTP teria o evento receber uma solicitação. Outros exemplos:



e-mail enviado, banco de dados conectado (ou erro), usuário autenticado, arquivo carregado etc.

Os eventos se encontram em um módulo independente events. Dentro desta biblioteca ou módulo há uma série de utilidades para trabalhar com eventos.

Para carregá-lo:

```
var eventos = require('events');
```

O emissor de eventos, encontra-se na propriedade EventEmitter.

```
var EmissorEventos = eventos.EventEmitter;
```

Em Node.js existe um loop de eventos, de modo que quando ocorre um evento, o sistema fica escutando no momento que se produz, para executar então uma função. Essa função é conhecida como "callback" ou como "manipulador de eventos" e contém o código que você quer que seja executado no momento que se produza o evento ao que a associamos.

Primeiro deve-se que "instanciar" um objeto da classe EventEmitter, através variável EmissorEventos.

```
var ee = new EmissorEventos();
```

O objeto Event Emitter tem vários métodos interessantes para emitir, cadastrar e processar eventos, por exemplo o método on( ), onde registra-se um evento, e emit( ) onde emite-se de fato o evento, mediante código Javascript.

Usa-se método on() para definir as funções manipuladoras de eventos, ou seu equivalente addEventListener().

No exemplo será utilizado, o método Date.now() que retorna o número de milissegundos decorridos desde 1 de janeiro de 1970 00:00:00 UTC.

Para aproveitar algumas das características mais interessantes de aplicações NodeJS será usado setInterval() para emitir dados a cada intervalo de tempo.

Na linha do método on, é definido um "ouvinte" de evento para o evento chamado 'dados' usando o método on() do EventEmitter. Quando o evento 'dados' for emitido, a função definida como argumento será executada. Essa função imprime o valor recebido como argumento (chamado de fecha) no console.

Em seguida, é criado um intervalo usando `setInterval()`, que executa uma função a cada 500 milissegundos. Dentro dessa função, o evento 'dados' é emitido usando o método `emit()` do `EventEmitter`. O valor passado para o evento é `Date.now()`, que retorna a data e a hora atuais em milissegundos.

Portanto, a cada 500 milissegundos, o evento 'dados' é emitido com o valor atual de data e hora em milissegundos, que é então impresso no console pelo ouvinte do evento.

Digitar o código no arquivo `quinto.js`

```
var eventos = require('events'); // Atribuição da classe EventEmitter a uma variável

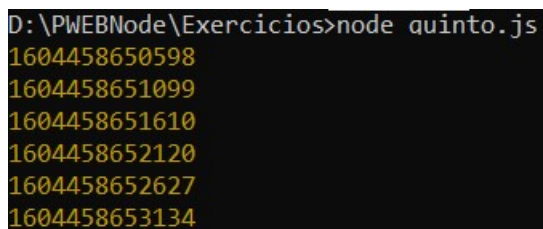
var EmissorEventos = eventos.EventEmitter; //Criação de uma instância do EventEmitter variável:

var ee = new EmissorEventos();
//ou criando direto sem a variável intermediária
//var ee = new eventos.EventEmitter();
//mas da forma anterior é uma boa prática
//É registrado um ouvinte (listener) para o evento 'dados'.
//Quando esse evento é emitido, a função passada como argumento será executada.
//A função neste caso //imprime no console o valor recebido.
ee.on('dados', function(fecha){
  console.log(fecha);
});

//Emissão do evento a cada 500 milissegundos:
setInterval(function(){
  ee.emit('dados', Date.now());
}, 500);
```

Testar o arquivo `quinto.js`

*Figura 18: Execução do Exercício quinto.js*



```
D:\PWEBNode\Exercicios>node quinto.js
1604458650598
1604458651099
1604458651610
1604458652120
1604458652627
1604458653134
```

Caso queira parar o programa digite o atalho `Ctrl + C` que o programa irá parar.

#### **4. Respondendo requisições HTTP**

Antes a interpretação do código JavaScript ficava a cargo dos navegadores, com o Node é possível trazer essa realização para aplicações desktop, bastando apenas submeter os scripts para que o Node.JS interprete o comando e converta tais comandos em linguagem de máquina, permitindo, portanto, o controle de recursos do sistema operacional. **Do ponto de vista do desenvolvimento web o Node.js implementa diversos recursos que tornam capaz de responder a diversos tipos de protocolos diferentes, dentre esses protocolos o HTTP.** Isso significa que, apesar do Node.js não ser um servidor HTTP propriamente dito, é possível implementar os recursos necessários para que a aplicação seja capaz de responder a requisições feitas a partir deste protocolo.

É necessário importar uma biblioteca chamada HTTP e a partir dessa biblioteca pode-se criar um servidor e com isso passar a “escutar” requisições que são feitas em uma porta específica.

Para criar (ou subir) um servidor usar o método: `http.createServer`

Esse método vai receber um atributo `function` com dois parâmetros: uma requisição e uma resposta.

A função vai devolver para o requisitante um código `html`.

Também é necessário informar para o servidor qual é a porta que ele deve “escutar”.

Digitar o código para criar o servidor no arquivo `exercicio6.js`

```
var http = require('http');
var server = http.createServer( function(req,res){
    res.end("<html><body>Site da Fatec Sorocaba</body></html>");
});
server.listen(3000);
```

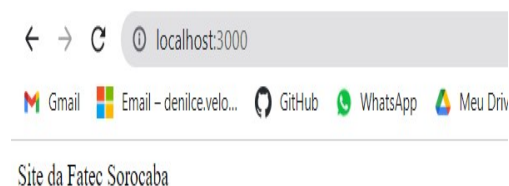
Executar o `exercicio6.js` e ele vai ficar aguardando (escutando) as requisições.

*Figura 19: Execução do `exercicio6.js`*

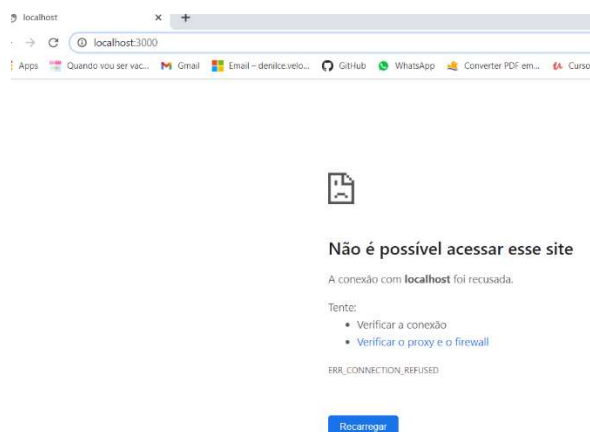
```
D:\PWEBNode\Exercicios>node exercicio6.js
```

Abrir o navegador e digitar localhost:3000, a tela abaixo será apresentada, porque encontrou o servidor.

*Figura 20: Execução do exercicio6.js (chamando o browser)*



Para “derrubar” o servidor basta dar ctrl+c, e é claro que a chamada no browser vai dar erro.



## 5. Respondendo requisições usando a URL

As requisições na web são feitas a uma URL (*Uniform Resource Locator*), que nada mais é do que o caminho do site. Um site pode ter muitas URLs.

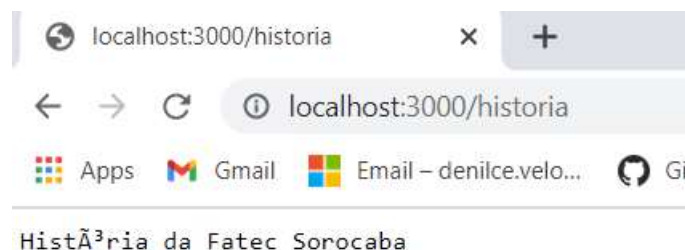
Digitar o código para criar mais algumas urls no arquivo exercicio7.js

```
var http = require('http');
var server = http.createServer( function(req,res){
  var opcao = req.url;
  if (opcao=='/historia') {
    res.end("<html><body>História da Fatec Sorocaba</body></html>");
  }
  else if (opcao=='/cursos') {
    res.end("<html><body>Cursos</body></html>");
  }
  else if (opcao=='/professores') {
    res.end("<html><body>Professores</body></html>");
  }
  else
    res.end("<html><body>Site da Fatec Sorocaba</body></html>");
});
server.listen(3000);
```

No prompt executar exercicio7.js e ele vai ficar aguardando as requisições.

Abrir o navegador e digitar localhost:3000/historia, a tela abaixo será apresentada, porque encontrou o servidor e a url desejada.

*Figura 21: Execução do exercicio7.js (chamando no browser a opção historia)*



Repetir para localhost:3000/cursos e localhost:3000/professores.

Observe que toda vez que fizer uma alteração no código será necessário reiniciar o servidor do Node. Mais para frente será utilizada a ferramenta Nodemon para auxiliar nesse problema.

## **6. Ferramentas para auxiliar no Desenvolvimento**

Serão citadas aqui algumas ferramentas que podem auxiliar no desenvolvimento com Node.js.

### 6.1 NPM

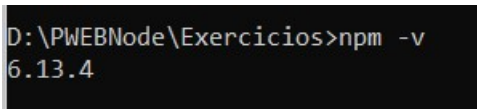
O Npm (*Node Package Manager*) é um serviço (grátis para pacotes públicos) que possibilita aos desenvolvedores criarem e compartilhar módulos com outros desenvolvedores, dessa forma facilitando a distribuição de códigos e ferramentas escritas em JavaScript. A vantagem do npm é que existem muitos módulos a disposição e a desvantagem é que como ele é open-source existem muitos módulos que muitas vezes realizam a mesma tarefa, e aí a necessidade de descobrir o que mais se adapta a necessidade do desenvolvedor. O Npm já foi instalado junto com o Node.js e será utilizado na instalação (inclusão) das demais ferramentas.<sup>19</sup>

Para verificar a versão do NPM digitar no prompt:

```
npm -v
```

A tela apresentada será parecida com essa:

*Figura 22: Versão do NPM*



```
D:\PWEBNode\Exercicios>npm -v
6.13.4
```

O próximo passo será customizar o npm, que no final vai gerar um arquivo package.json<sup>20</sup>, para isto digitar no prompt:

```
npm init
```

*Figura 23: Execução npm init*

---

<sup>19</sup> <https://docs.npmjs.com/about-npm>

<sup>20</sup> JSON (*JavaScript Object Notation*) é um modelo para armazenamento e transmissão de informações no formato texto. É bem simples, legível para as pessoas e tem sido bastante utilizado por aplicações Web devido a sua capacidade de estruturar informações de uma forma bem mais compacta do que o formato XML.

```
See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

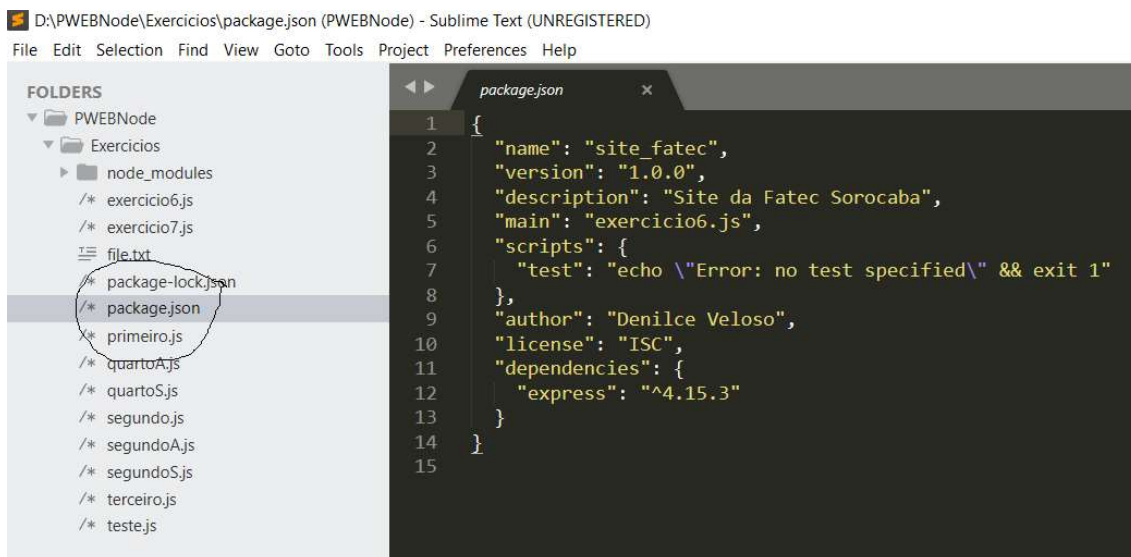
Press ^C at any time to quit.
package name: (exercicios) site_fatec
version: (1.0.0)
description: Site da Fatec Sorocaba
git repository:
keywords:
author: Denilce Veloso
license: (ISC)
About to write to D:\PWEBNode\Exercicios\package.json:

{
  "name": "site_fatec",
  "version": "1.0.0",
  "description": "Site da Fatec Sorocaba",
  "main": "exercicio6.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Denilce Veloso",
  "license": "ISC"
}
```

Se estiver tudo certo, ok.

Ele incluiu o arquivo package-json, que é um arquivo de metadados que descreve o projeto, incluindo informações como o nome do projeto, versão, dependências, scripts de inicialização, e outras configurações relevantes. importante pois se for instalar a aplicação em algum lugar, ele já sabe quais são as dependências necessárias e suas versões para baixar.

*Figura 24: Arquivo package-json*



**DICA:** Se você quiser saber como está o download de um item pelo npm, basta acessar <https://www.npmtrends.com>.

## 6.2 Instalação do Express

O Express é um popular *framework* web estruturado, escrito em JavaScript que roda sobre o ambiente node. Ele fornece **API para controle de rotas** e permite adicionar novos processos de requisição por meio de "middleware"<sup>21</sup> em qualquer ponto da "fila" de requisições.<sup>22</sup>

Para instalar o Express digitar no prompt:

npm install express --save

-save -> cria diretório e traz os arquivos de fatos (dependências) para dentro da nossa máquina, não precisa ficar procurando.

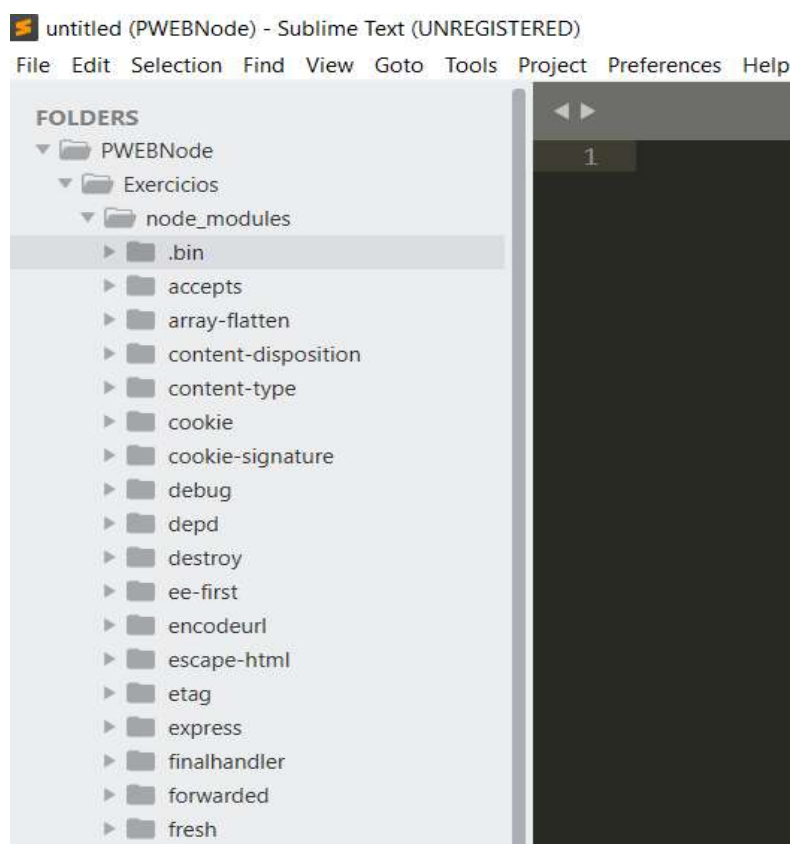
Ele cria uma pasta **node\_modules** (com todos os recursos que podem ser utilizados para otimizar o desenvolvimento) dentro do nosso projeto.

<sup>21</sup> O middleware é o software que se encontra entre o sistema operacional e os aplicativos nele executados.

<sup>22</sup> <https://expressjs.com/pt-br/>



*Figura 25: Pasta node\_modules*



### 6.3 Instalação do EJS

O EJS (*Embedded JavaScript Templating*) é uma engine de visualização, sendo possível transportar dados do back-end para o front-end, utilizando códigos em javascript no html das páginas.

O EJS permite criar HTML junto as instruções JavaScript.<sup>23</sup>

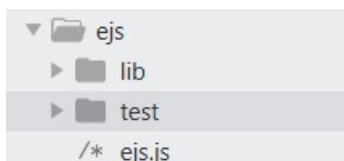
Para instalar o EJS digitar no prompt:

```
npm install ejs --save
```

Ele vai ficar disponível no node\_modules

<sup>23</sup> <https://medium.com/@pedrompinto/tutorial-node-js-como-usar-o-engine-ejs-12bcc688ebab>

*Figura 26: Pasta node\_modules\ejs*



#### **6.4 Instalação do Nodemon**

O Nodemon é um utilitário que reinicia automaticamente o servidor NodeJS quando houver qualquer alteração no código. Quando a aplicação for instalada em outro servidor o Nodemon vai ficar nesse servidor.

Uma das vantagens de linguagens interpretadas é só alterar o script e tentar executar sem compilar, o Nodemon vai facilitar isso no caso do Node para não necessitar subir o servidor toda vez que fizer alteração no código.

Como ele é um utilitário e não um módulo do projeto, ele pode ficar na área de desenvolvimento, quanto no servidor onde a aplicação de fato estiver instalada.

Para instalar o Nodemon digitar no prompt:

```
npm install nodemon --save-dev
```

Para iniciar o servidor usando Nodemon basta digitar no prompt:

```
nodemon app.js
```

Por enquanto não temos o arquivo, mas poderia testar por exemplo com o exercício7.

Atenção: Há uma restrição (por medidas de segurança) da execução do Nodemon no terminal do Visual Code, usar prompt do Windows ou digitar no terminal: Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

No arquivo package.json adicione o seguinte objeto script:

```
"scripts": {  
  "start": "nodemon exercicio7.js"  
},
```

*Figura 27: Teste nodemon digitando npm start no terminal*

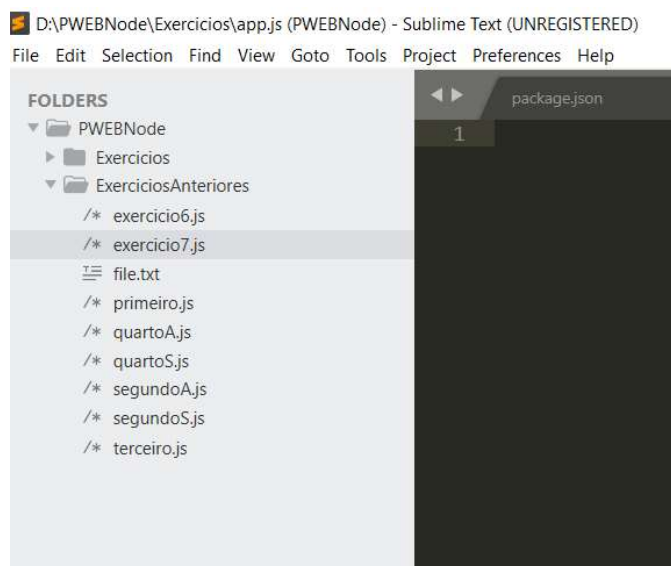
```
PS C:\Users\PWEBM\Desktop\PWEBNode\Exercicios> npm start  
  
> start  
> nodemon exercicio7.js  
  
[nodemon] 3.0.1  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,cjs,json  
[nodemon] starting `node exercicio7.js`
```

## 7. Utilizando as novas ferramentas

### 7.1 Guardando os exercícios anteriores

Com as novas ferramentas instaladas, será desenvolvido um novo exercício. Primeiro passo é criar uma pasta ExerciciosAnteriores para “guardar” os exercícios realizados (**copiar primeiro, segundo, ... exercicio7**) até agora (**somente os exercícios, tomar cuidado para não copiar node\_modules, package.json e etc**).

*Figura 28: Pasta ExerciciosAnteriores*



## 7.2 Utilizando o Express

Quem executa o código JavaScript é o Node e o *framework* Express é uma camada que fica acima do Node para fazer a interface entre os scripts e o Node, portanto é necessário que a aplicação seja feita com base na estrutura que o *framework* Express exige.

Criar um arquivo `app.js` dentro da pasta `exercícios`, observe `exercício7.js` para fins de comparação.

Figura 29: `exercício7.js` e `app.js`

```
1 var http = require('http');
2 var server = http.createServer( function(req,res){
3   var opcao = req.url;
4   if (opcao=='/historia') {
5     res.end("<html><body>História da Fatec Sorocaba</body></html>");
6   }
7   else if (opcao=='/cursos') {
8     res.end("<html><body>Cursos</body></html>");
9   }
10  else if (opcao=='/professores') {
11    res.end("<html><body>Professores</body></html>");
12  }
13  else
14    res.end("<html><body>Site da Fatec Sorocaba</body></html>");
15  } );
16 server.listen(3000);
17
```

```
var express = require('express');
var app=express(); // executando o express
app.listen(3000, function(){
  console.log("servidor com
express foi carregado");
});
```

Mudar o `require` para `express` e mudar também o nome da variável.

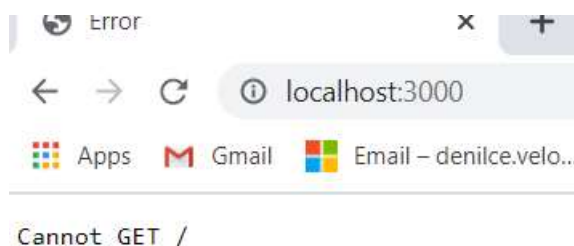
O `express` retorna uma função, precisa executar essa função. E o método `listen` fica “escutando” as requisições em uma determinada porta e precisa passar uma função de *callback* como anteriormente. Dentro da função fazer um teste, por exemplo, com `console.log(“servidor com express foi carregado”)`, só para ver se o servidor subiu.

Figura 30: Teste do servidor com `express`

```
D:\PWEBNode\Exercicios>node app.js
servidor com express foi carregado
```

Executar no browser

*Figura 31: Teste local do servidor*



O Express tratou e identificou que não existe nenhum caminho (página / resposta). Antes precisava tratar a url, agora pode usar a função get, incluir a chamada para a página principal (o /) e usar o método send por estar trabalhando direto com o Express, quando estava trabalhando direto com o node era end.

#### *Diferença res.send vs. res.end*

**res.send** - é um método do objeto res no Express que envia a resposta para o cliente e termina o processo de resposta.

Ele pode aceitar vários tipos de entrada: string, buffer, ou um objeto que será convertido para JSON.

Além de enviar o conteúdo, ele automaticamente define o cabeçalho Content-Type baseado no tipo de conteúdo que está sendo enviado. Por exemplo, se você enviar uma string HTML, ele definirá o Content-Type como text/html.

**res.end** - é um método do objeto res no Node.js (não específico do Express) que finaliza a resposta sem nenhum processamento adicional.

Ele envia o que for passado como argumento como corpo da resposta e encerra a comunicação, mas não define o Content-Type automaticamente. Portanto,

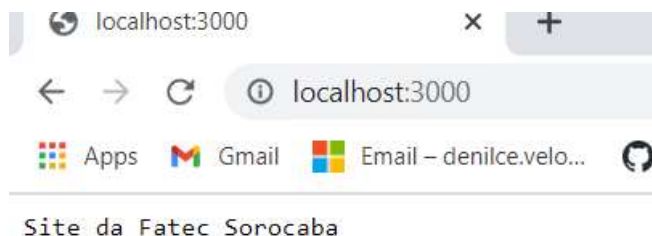
cabe ao desenvolvedor configurar os cabeçalhos adequados antes de usar `res.end` se necessário. Esse método é mais baixo nível comparado a `res.send`.

Alterar o arquivo `app.js` para que os caminhos sejam encontrados.

```
var express = require('express');
var app=express(); // executando o express
app.get('/', function(req,res){
    res.send("<html><body>Site da Fatec Sorocaba</body></html>");
});
app.get('/historia', function(req,res){
    res.send("<html><body>História da Fatec Sorocaba</body></html>");
});
app.get('/cursos', function(req,res){
    res.send("<html><body>Cursos da Fatec Sorocaba</body></html>");
});
app.get('/professores', function(req,res){
    res.send("<html><body>Professores da Fatec Sorocaba</body></html>");
});
app.listen(3000, function(){
    console.log("servidor com express foi carregado");
});
```

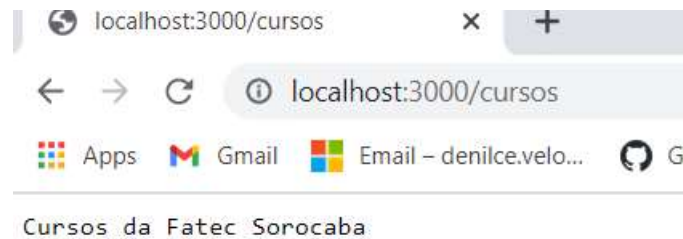
Para o servidor e tentar fazer a requisição novamente.

*Figura 32: Teste local*



Se tentar chamar a página de Cursos por exemplo.

*Figura 33: Teste da opção Cursos*



O código fica mais simples utilizando o Express, do que ficar utilizando aqueles ifs como no exemplo exercício7.js, fica mais fácil organizar as rotas.

### 7.3 Utilizando o EJS

Será utilizando o EJS (linguagem de modelagem), utilizando o Get junto com a função Render do EJS.

Alterar o arquivo app.js, o primeiro passo é informar no app que o engine de view do express passou a ser ejs → `app.set('view engine', 'ejs');`

Agora ao invés do método send, com o ejs pode ser utilizado o render e passar o arquivo que será renderizado, alterar as rotas.

```
var express = require('express');
var app=express(); // executando o express

app.set('view engine', 'ejs');

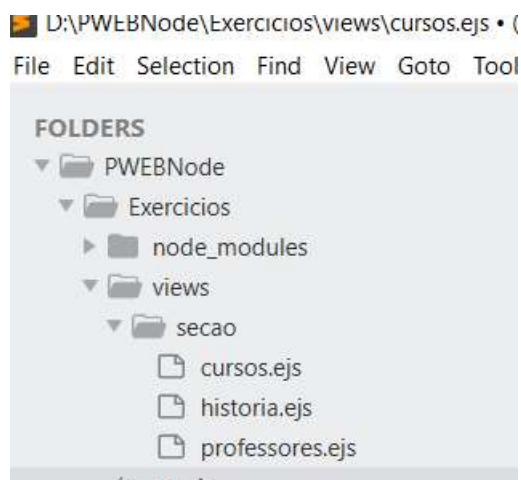
app.get('/', function(req,res){
  res.send("<html><body>Site da Fatec Sorocaba</body></html>");
});

app.get('/historia', function(req,res){
  res.render("secao/historia");
});
app.get('/cursos', function(req,res){
  res.render("secao/cursos");
});
app.get('/professores', function(req,res){
  res.render("secao/professores");
});
app.listen(3000, function(){
```

```
console.log("servidor com express foi carregado");  
});
```

Ainda não executar, o próximo passo arrumar a estrutura da aplicação, criando um new folder chamado views dentro da pasta Exercícios. E dentro dele serão criados os arquivos com extensão ejs. Criar um secao e dentro dela criar os arquivos (opções) historia, cursos e professores.

*Figura 34: Pasta views e seção*



No arquivo historia.ejs digitar o código abaixo:

```
<!DOCTYPE html>  
<html lang="pt-br">  
  
<head>  
  <meta charset="UTF-8">  
  <title> História da Fatec Sorocaba</title>  
</head>  
  
<body>  
  <p> A Faculdade de Tecnologia de Sorocaba foi criada em 20/05/1970 pelo então Governador do Estado de São Paulo, Dr. Roberto Costa de Abreu Sodré. Foi a primeira escola pública de nível superior em Sorocaba.  
  </p>  
  <br>  
  <br>
```



```
<p>
    O primeiro dia letivo na Fatec Sorocaba ocorreu no dia 07/06/1971, nas instalações da atual Etec
    Rubens de Faria e Souza com 66 alunos que iniciavam seus estudos no então Curso Técnico Superior de
    Oficinas, atualmente Tecnologia em Fabricação Mecânica.
</p>
</body>

</html>
```

No arquivo cursos.ejs digitar o código:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title> Cursos Fatec Sorocaba</title>
</head>
<body>
    <ul> Os cursos da Fatec Sorocaba são:
    <li>Análise e Desenvolvimento de Sistemas</li>
    <li>Eletrônica Automotiva</li>
```

```
<li>Fabricação Mecânica</li>
<li>Gestão da Qualidade</li>
<li> Logística</li>
<li>Manufatura Avançada</li>
<li>Processos Metalúrgicos</li>
<li> Polímeros</li>
<li>Projetos Mecânicos</li>
<li>Sistemas Biomédicos</li>
<li> Gestão Empresarial - EAD</li>
</ul>
</body>
</html>
```

No arquivo professores.ejs digitar o código:

```
<!DOCTYPE html>
<html lang="pt-br">

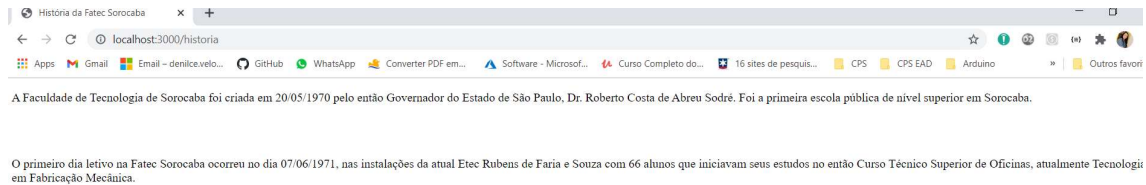
<head>
  <meta charset="UTF-8">
  <title> Professores Fatec Sorocaba</title>
</head>

<body>
  <h1>Está é a lista do corpo docente da Fatec Sorocaba</h1>
</ul>
</body>
</html>
```

Recarregue o servidor com o arquivo app.js.

Através do browser procurar a página historia por exemplo, ficará assim:

*Figura 35: Seção historia*



## 7.4 Testando o Nodemon

Sair do servidor e executar nodemon app.js

Vai no arquivo package.json e altere o parâmetro scripts:

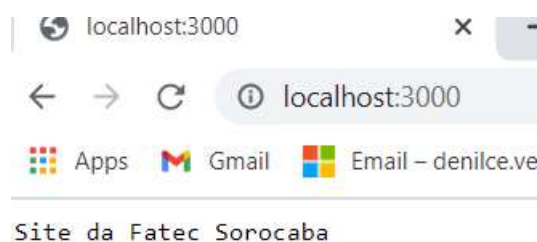
```
"scripts": {  
  "start": "nodemon app.js"  
},
```

Figura 36: Teste do Nodemon

```
PS C:\Users\PWEBM\Desktop\PWEBNode\Exercicios> npm start  
  
> start  
> nodemon app.js  
  
[nodemon] 3.0.1  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,cjs,json  
[nodemon] starting `node app.js`  
servidor com express foi carregado
```

Tentar executar a pasta principal

Figura 37: Teste do site principal

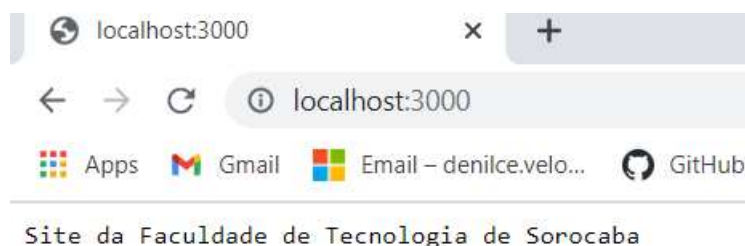


Se for feita qualquer alteração no arquivo app.js, automaticamente ele restarta o servidor considerando a alteração. Supondo que fosse alterada a linha 8 do arquivo app.js

```
res.send("<html><body>Site da Faculdade de Tecnologia de Sorocaba</body></html>");
```

Tentar executar a pasta principal novamente

*Figura 38: Novo Teste do site principal*



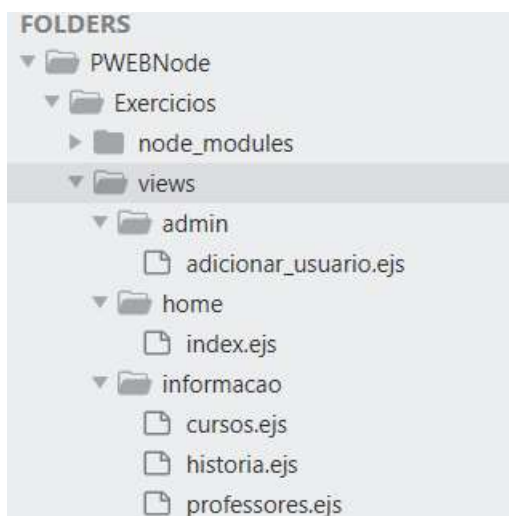
Observar que ele aceitou a mudança e não precisou parar e subir o servidor novamente.

## **7.5 Organizando melhor a aplicação**

### **7.5.1 Criando pasta views (guardando views anterior)**

Vamos melhorar mais a nossa aplicação. Renomear a pasta views para views anteriores e criar uma pasta views com outras pastas dentro dela (admin, home e informação), e dentro de cada pasta criar alguns arquivo .ejs. Na pasta informacao copiar os arquivos .ejs das views anteriores, na pasta home criar um arquivo index.ejs e na pasta admin criar um arquivo adicionar\_usuario.ejs.

*Figura 45: Nova Pasta views*



Veja sugestão para arquivo adicionar\_usuario.ejs da pasta admin.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <title> Adicionar usuário </title>
</head>
<body>
  <p> adicionar usuário</p>
</body>
</html>
```

Veja sugestão para arquivo index.ejs da pasta home.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <title> Página Principal </title>
```

```
</head>
<body>
  <p> Bem-vindo à página inicial da Fatec Sorocaba</p>
</body>
</html>
```

### Alterar o código do arquivo app.js

```
var express = require('express');
var app=express(); // executando o express
app.set('view engine', 'ejs');
app.get('/', function(req,res){
  res.render("home/index");
});
app.get('/formulario_adicionar_usuario', function(req,res){
  res.render("admin/adicionar_usuario");
});
app.get('/informacao/historia', function(req,res){
  res.render("informacao/historia");
});
app.get('/informacao/cursos', function(req,res){
  res.render("informacao/cursos");
});
app.get('/informacao/professores', function(req,res){
  res.render("informacao/professores");
});

app.listen(3000, function(){
  console.log("servidor express carregado");
});
```

Recarregar o servidor usando nodemon e testar os novos caminhos

[http://localhost:3000/formulario\\_adicionar\\_usuario](http://localhost:3000/formulario_adicionar_usuario)

<http://localhost:3000/>

<http://localhost:3000/informacao/historia>

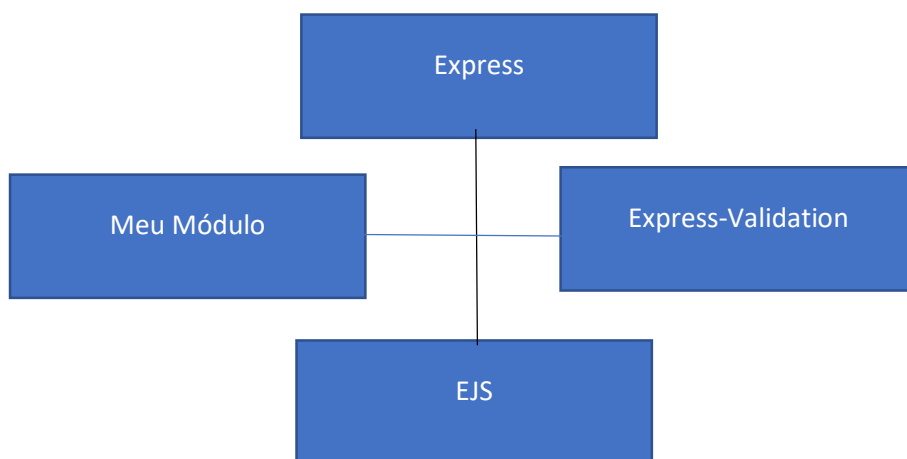
<http://localhost:3000/informacao/professores>

<http://localhost:3000/informacao/cursos>

### 7.5.2 Módulos e CommonJS

O que são esses módulos e como podem ser construídos?

Figura 46: Estrutura Módulo



No Node.js, um module é uma coleção de funções e objetos do JavaScript que podem ser utilizados por aplicativos externos. Um js pode ser considerado um módulo, caso suas funções e dados sejam feitos para programas externos. Serve para organizar melhor o código e facilitar a manutenção (por exemplo, o que fazemos com as classes). Um módulo pode ser disponibilizado na web para ser utilizado em outros projetos. Exemplos de Módulos: Express e EJS. O módulo pode retornar um objeto, uma função, uma string etc.

Para incluir um módulo, como já visto anteriormente, utiliza-se a função require. "Common JS", diz respeito ao formato de construção dos módulos. É uma API com o objetivo de agrupar as necessidades de diversas aplicações Javascript em uma única API, que funcione em diversos ambientes e interpretadores. Criando o conceito de módulos que façam essas funções.

Criar um arquivo novo chamado modulo1.js na pasta Exercicios.

```
var texto = "Observe que essa mensagem vem do módulo";
```

Alterar o código do arquivo app.js para utilizar um módulo:

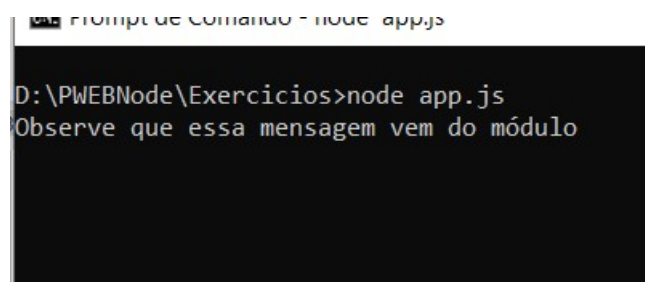
```
var express = require('express');  
var texto = require('./modulo1'); //não precisa colocar modulo1.js ele já entende que é js  
  
var app=express(); // executando o express  
  
app.set('view engine', 'ejs');  
  
app.get('/', function(req,res){  
    res.render("home/index");  
});  
  
app.get('/formulario_adicionar_usuario', function(req,res){  
    res.render("admin/adicionar_usuario");  
});  
  
app.get('/informacao/historia', function(req,res){  
    res.render("informacao/historia");  
});  
  
app.get('/informacao/cursos', function(req,res){  
    res.render("informacao/cursos")  
});  
  
app.get('/informacao/professores', function(req,res){  
    res.render("informacao/professores");  
});  
  
app.listen(3000, function(){  
    console.log(texto);  
});
```

Precisa usar exports no modulo1, senão ele não encontrará o texto, pois tentar executar no console não mostrará o texto.

```
var texto = "Observe que essa mensagem vem do módulo";  
module.exports = texto;
```

Testar inicialmente chamando o app.js

*Figura 47: Teste chamando o módulo*





*Figura 48: Teste chamando a página*



Se o teste for realizado usando o nodemon a tela apresentada será assim:

*Figura 49: Teste chamando o módulo usando nodemon*

```
D:\PWEBNode\Exercicios>nodemon app.js
[nodemon] 1.10.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
Observe que essa mensagem vem do módulo
```

Quando o módulo retornar uma função, precisa executar a função. Embora o módulo possa ter muitos códigos, o que será recebido será o que está no exports. Observe na pasta node\_modules por exemplo os arquivos do ejs aparece o exports.

### **7.5.3 Modularizando a aplicação**

**Separar código de infraestrutura e regras de negócios traz muitas vantagens como por exemplo facilitar manutenções.**

#### *7.5.3.1 Separando as configurações do servidor*

Para separar as configurações do servidor das demais, será necessário criar uma pasta app dentro da pasta exercícios e uma pasta config dentro de app. E dentro da pasta config criar um arquivo (módulo) server.js.

*Figura 50: Pasta config*



O código do arquivo **server.js** é o seguinte, observar que é parte do código que estava no arquivo app.js.

```
var express = require('express');

var app=express(); // executando o express

app.set('view engine', 'ejs'); // o mecanismo de engine a ser usado
app.set('views', './app/views'); // diretório onde os arquivos estão localizados

module.exports = app;
```

O arquivo **app.js** fica conforme a seguir, não esquecer de chamar o módulo servidor:

```
var app = require('./app/config/server'); // carregando o módulo do servidor

app.get('/', function(req,res){
  res.render("home/index");
});

app.get('/formulario_adicionar_usuario', function(req,res){
  res.render("admin/adicionar_usuario");
});

app.get('/informacao/historia', function(req,res){
  res.render("informacao/historia");
});

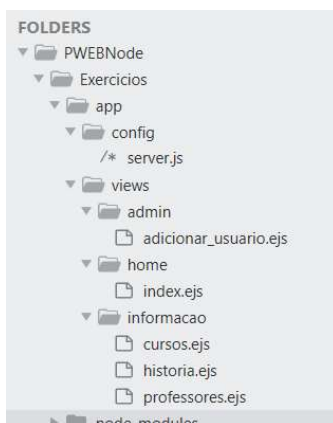
app.get('/informacao/cursos', function(req,res){
  res.render("informacao/cursos")
});

app.get('/informacao/professores', function(req,res){
  res.render("informacao/professores");
});

app.listen(3000, function(){
  console.log("servidor iniciado");
});
```

Antes do teste, mover a pasta views para dentro da pasta app. Deverá ficar assim:

*Figura 51: Cópia da pasta views*



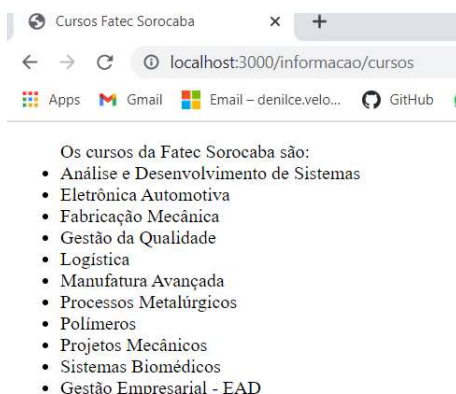
Fazer um teste se inicia o servidor.

*Figura 52: Tela console iniciando servidor*

```
cmd Prompt de Comando - nodemon app.js
D:\PWEBNode\Exercicios>nodemon app.js
[nodemon] 1.10.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
servidor iniciado
```

Se chamar <http://localhost:3000/> e as outras páginas estiverem funcionando normalmente, poderá testar a página iniciar e a de cursos por exemplo.

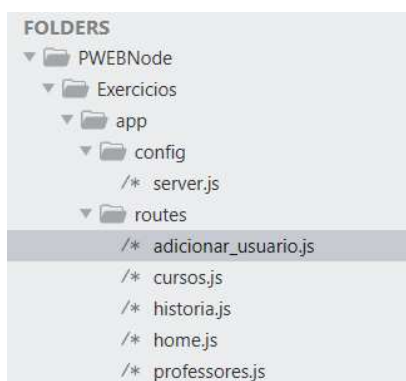
*Figura 53: Tela console iniciando servidor*



### 7.5.3.2 Separando as configurações das rotas

**Criar uma pasta routes dentro de app.** Na pasta routes criar 5 arquivos js, para as rotas que estão em app.js

*Figura 54: Pasta routes*



Por padrão o EJS procura os arquivos das rotas na pasta raiz, como os arquivos mudaram de pasta, **configurar a nova pasta no config/server e incluir os módulos dentro da aplicação.** Antes precisa deixar no formato EJS, colocando o exports. Nesse caso precisa exportar como função, porque precisa executar os comandos, não é possível simplesmente atribuir algo. Também precisa definir a variável app como parâmetro porque ele não reconhece a variável app que está dentro de app.js.

#### Arquivo adicionar\_usuario.js

```
module.exports = function(app) {  
  app.get('/admin/adicionar_usuario', function(req,res){  
    res.render("admin/adicionar_usuario");  
  });  
}
```

#### Arquivo cursos.js

```
module.exports = function(app) {  
  app.get('/informacao/cursos', function(req,res){  
    res.render('informacao/cursos');  
  });  
}
```

#### Arquivo historia.js

```
module.exports = function(app) {  
  app.get('/informacao/historia', function(req,res){  
    res.render('informacao/historia');  
  });  
}
```

#### Arquivo home.js

```
module.exports = function(app) {  
  app.get('/', function(req,res){  
    res.render("home/index");  
  });  
}
```

#### Arquivo professores.js

```
module.exports = function(app) {  
  app.get('/informacao/professores', function(req,res){  
    res.render('informacao/professores');  
  });  
}
```

No arquivo **app.js** deve incluir esses módulos das rotas.

```
var app = require('./app/config/server');  
  
var rotaHome = require('./app/routes/home'); // só está definindo  
rotaHome(app); // está executando  
  
var rotaAdicionarUsuario = require('./app/routes/adicionar_usuario');  
rotaAdicionarUsuario(app);  
  
var rotaHistoria = require('./app/routes/historia');  
rotaHistoria(app);  
  
var rotaCursos = require('./app/routes/cursos');  
rotaCursos(app); // está executando  
  
var rotaProfessores = require('./app/routes/professores'); // só está definindo
```

```
rotaProfessores(app); // está executando

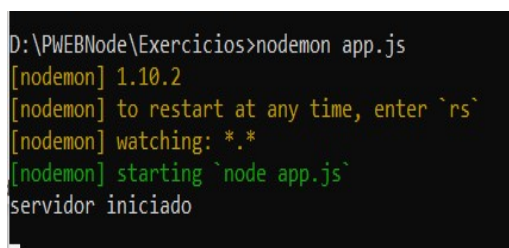
/* poderia executar assim também */
/*
var rotaAdicionarUsuario = require('./app/routes/adicionar_usuario')(app);

*/

app.listen(3000, function(){
    console.log("servidor iniciado");
});
```

Iniciar o servidor novamente e testar as páginas.

*Figura 55: Tela console iniciando servidor*



```
D:\PWEBNode\Exercicios>nodemon app.js
[nodemon] 1.10.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
servidor iniciado
```

Teste da página professores.

*Figura 56: Tela página professores*



**Está é a lista do corpo docente da Fatec Sorocaba**

## **8. Banco de Dados**

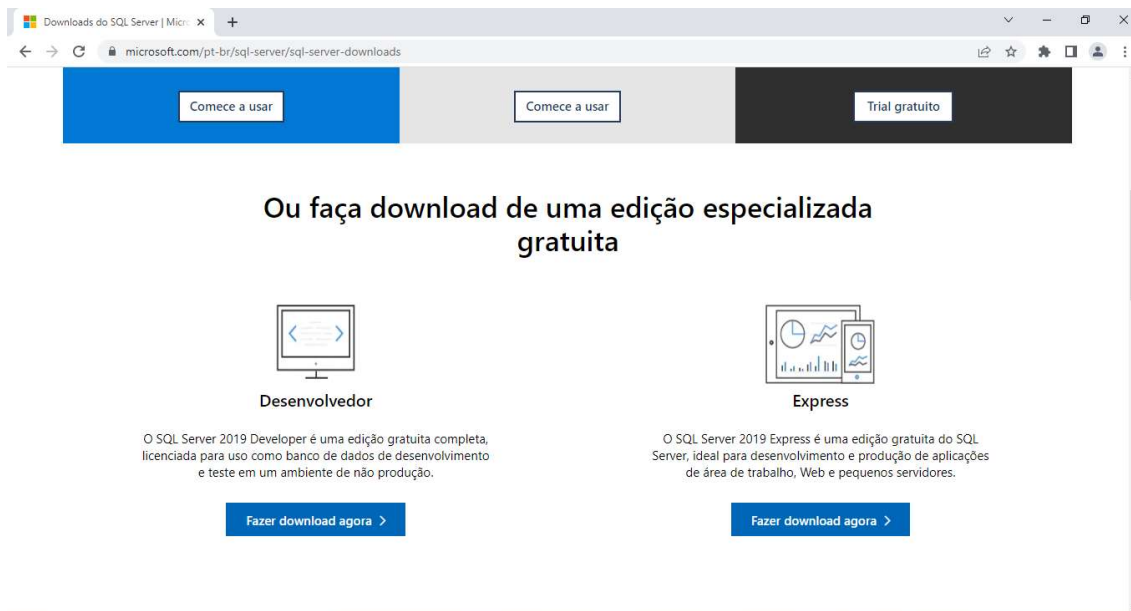
Nesse tópico será construída uma aplicação em NODE.JS que irá se conectar ao banco de dados Microsoft SQL Server e realizar algumas manipulações usando a linguagem DML (*Data Manipulation Language*) do SQL, ou seja, algumas operações de SELECT, INSERT, UPDATE e DELETE.

SqlServer é um sistema gerenciador de banco de dados, utiliza a linguagem SQL (*Structure Query Language* – Linguagem de Consulta Estruturada), que é a linguagem mais popular para inserir, acessar e gerenciar o conteúdo armazenado num banco de dados.<sup>24</sup>

### **8.1 Instalação do SQL Server**

Para instalar o SQL Server na sua máquina, fazer o download no site abaixo.

*Figura 57: Site download do SQL Server*



<sup>24</sup> <https://www.microsoft.com/pt-br/sql-server/sql-server-2019>

## 8.2 Instalação do driver mssql

Será necessário baixar um driver do SQL Server para o Node, que irá funcionar como um módulo para a aplicação de maneira mais fácil.

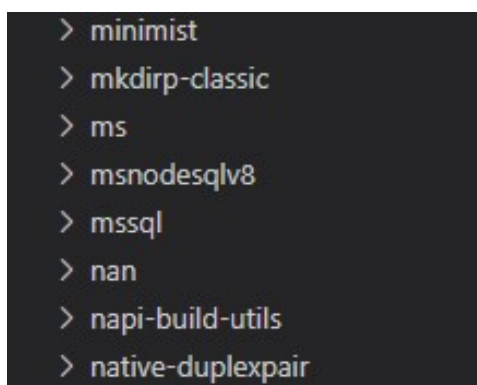
Comandos:

`npm install mssql --save` → para desenvolvimento na Fatec

`npm install msnodesqlv8 --save` → para desenvolvimento na sua própria máquina

Nos módulos deve aparecer o mssql.

*Figura 58: Módulos do SQL Server*



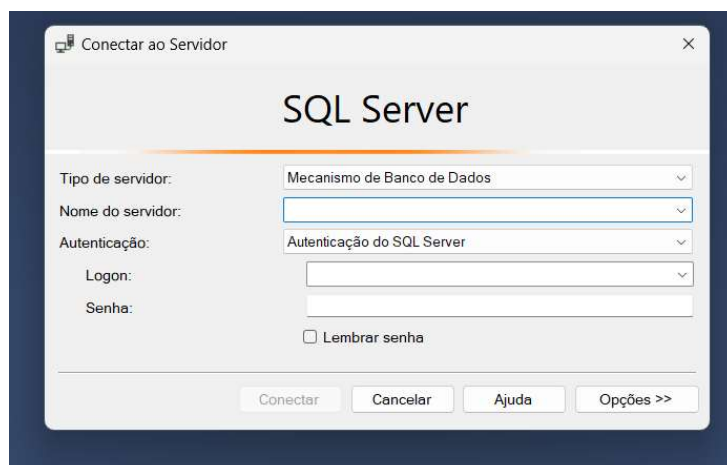
Para criar o banco de dados, será utilizada a ferramenta SQL Server Management Studio. **Atenção, a partir daqui você tem 2 opções.**

*Se for usar na máquina da faculdade utilizar o item 8.3, caso seja na sua máquina particular e não quiser utilizar o Management utilize o item 8.4, se for usar o Management a autenticação pode ser pelo Windows (opção escolhida na instalação).*

## 8.3 Usando o SQL Server no Management Studio

*Figura 59: Tela de Login do Management Studio*





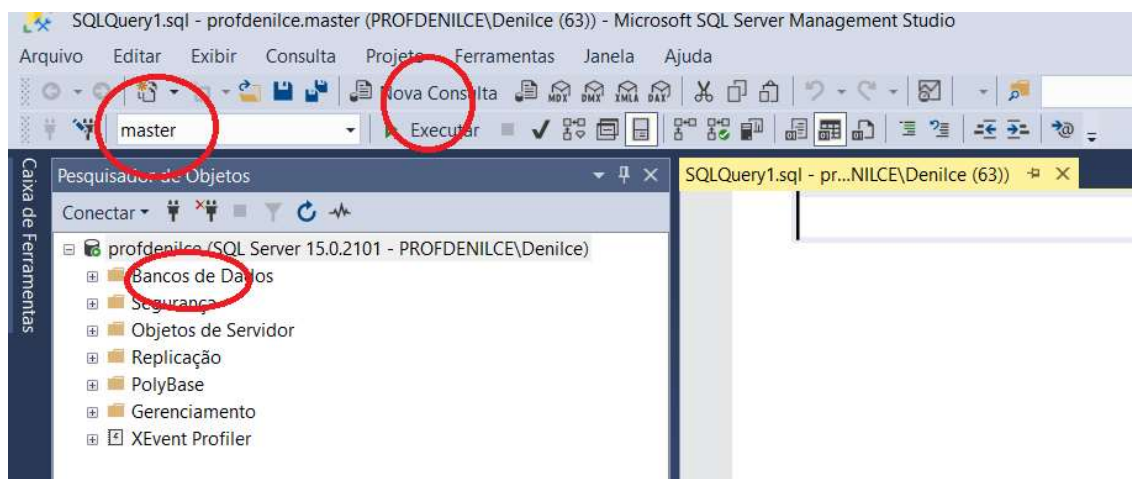
Servidor: Apolo

Logon: BD+últimos 7 dígitos do seu RA

Senha: A12345678a (se acaso for a primeira vez que você está logando, será necessário trocar a senha)

Através da opção Nova consulta, criar as seguintes tabelas professores e usuário, não esqueça de verificar qual Database você está setado, você precisa ter direito para criar as tabelas (veja se o database PWEB está disponível para você).

*Figura 60: Interface do Management Studio*



```
CREATE TABLE professores
(
  ID_PROFESSOR INT IDENTITY(1,1) PRIMARY KEY NOT NULL ,
  NOME_PROFESSOR VARCHAR(50) NOT NULL,
  EMAIL_PROFESSOR VARCHAR(100) NOT NULL,
);
```

```
CREATE TABLE usuario  
(  
ID_USUARIO INT IDENTITY(1,1) PRIMARY KEY NOT NULL ,  
NOME_USUARIO VARCHAR(50) NOT NULL,  
EMAIL_USUARIO VARCHAR(100) NOT NULL,  
SENHA_USUARIO VARCHAR(6) NOT NULL,  
);
```

Criar alguns registros na tabela de professores.

```
insert into professores (NOME_PROFESSOR,EMAIL_PROFESSOR)  
values ('DENILCE','denilce@gmail.com');  
insert into professores values ( 'ANGELICA ','angelica@gmail.com');  
insert into professores values ( 'JEFFERSON','jefferson@gmail.com');  
insert into professores (nome_professor, email_professor) values  
('CRISTIANE','cris@gmail.com');
```

### *8.4 Usando o SQL Server no Prompt*

Abrir o prompt de comando e digitar o comando:

```
sqlcmd -S NOME_DO_SERVIDOR -U LOGON -P SENHA
```

Após isso, será exibido um contador de linha mostrando que você está conectado ao banco de dados.

*Figura 61: Testando o SqlServer no console*



A screenshot of a dark-themed console window. On the left side, there is a vertical list of numbers from 1 to 11, each followed by a greater-than sign (>). The numbers are displayed in a light blue or cyan color. The rest of the console area is empty and dark.

Se você instalar a ferramenta Microsoft SQL Server Management Studio será mais fácil criar o banco de dados e as tabelas através dessa ferramenta.

Para criar um banco pelo prompt, digitar:

```
create database site_fatec  
go
```

Para acessar o banco criado digitar:

```
use site_fatec;  
go
```

Criar as seguintes tabelas professores e usuario:

```
CREATE TABLE professores  
(  
ID_PROFESSOR INT IDENTITY(1,1) PRIMARY KEY NOT NULL ,  
NOME_PROFESSOR VARCHAR(50) NOT NULL,  
EMAIL_PROFESSOR VARCHAR(100) NOT NULL,  
);  
CREATE TABLE usuario  
(  
ID_USUARIO INT IDENTITY(1,1) PRIMARY KEY NOT NULL ,  
NOME_USUARIO VARCHAR(50) NOT NULL,  
EMAIL_USUARIO VARCHAR(100) NOT NULL,  
SENHA_USUARIO VARCHAR(6) NOT NULL,  
);  
go
```

Criar alguns registros na tabela de professores.

```
insert into professores (NOME_PROFESSOR,EMAIL_PROFESSOR) values ('DENILCE','denilce@gmail.com');  
insert into professores values ( 'ANGELICA ','angelica@gmail.com');  
insert into professores values ( 'JEFFERSON','jefferson@gmail.com');  
insert into professores (nome_professor, email_professor) values ('CRISTIANE','cris@gmail.com');  
go
```

### ***8.5 Listando dados de uma tabela na página***

A ideia é listar os professores cadastrados na página dos professores.

Alterar o arquivo **professores.js** conforme código abaixo.

Primeiro fazer a conexão com o banco de dados, incluir o módulo do SQL Server.

Quando a página dos professores for chamada irá testar se banco de dados está conectando e listar os registros dos professores. Os parâmetros da “conexão”

devem obedecer a uma estrutura JSON. O res.send envia o resultado da consulta.

```
module.exports = function(app){
  app.get('/informacao/professores', function(req,res){
    const sql = require('mssql');
    module.exports = function(){
      const sqlConfig = {
        user: 'BD22130**', // seu usuario BD+7 últimos dígitos do seu ra
        password: '*****', // sua senha
        database: 'BD',
        server: 'APOLO',
        options: {
          encrypt: false,
          trustServerCertificate: true
        }
      }
    }
    return sql.connect(sqlConfig);
  })
  async function getProfessores() {
    try {
      const pool = await sql.connect(sqlConfig);
      const results = await pool.request().query('SELECT * from PROFESSORES');
      res.send(results) // vai mostrar no formato JSON
    } catch (err) {
      console.log(err);
    }
  }
  getProfessores();
});
}
```

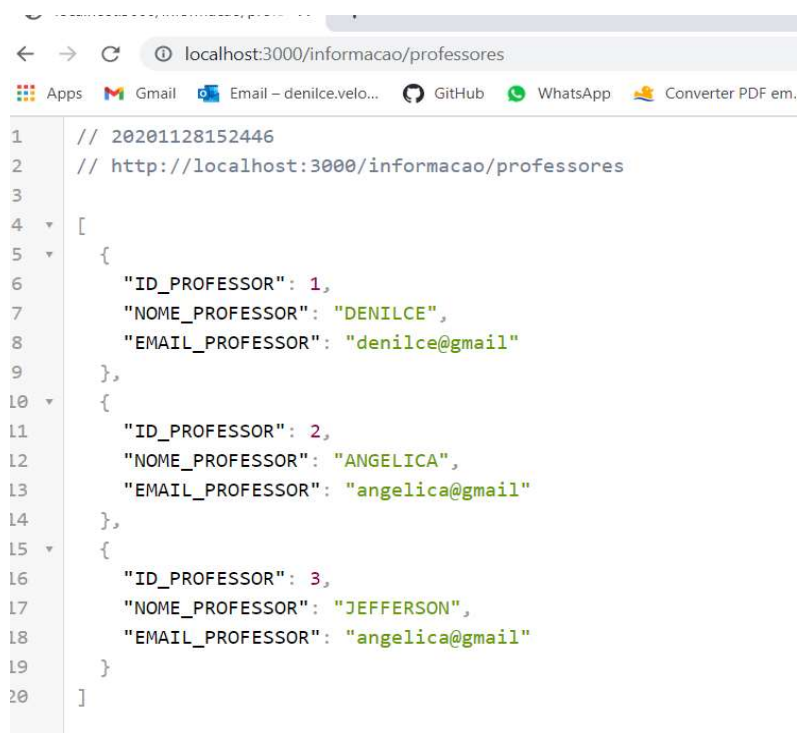
Tentar carregar a página dos professores

<http://localhost:3000/informacao/professores>

**\*\* se for na sua máquina particular ficaria um pouco diferente em alguns pontos**

```
const sql = require('mssql/msnodesqlv8');
const sqlConfig = {
  user: "xx",
  password: "xxx",
  database: 'BD',
  server: 'NOMESUAMAQUINA\\SQLEXPRESS',
  driver: 'msnodesqlv8',
  options: {
    encrypt: false,
    trustServerCertificate: true,
  }
}
```

*Figura 62: Testando a página dos professores (listar registros)*



```
1 // 20201128152446
2 // http://localhost:3000/informacao/professores
3
4 [
5   {
6     "ID_PROFESSOR": 1,
7     "NOME_PROFESSOR": "DENILCE",
8     "EMAIL_PROFESSOR": "denilce@gmail"
9   },
10  {
11    "ID_PROFESSOR": 2,
12    "NOME_PROFESSOR": "ANGELICA",
13    "EMAIL_PROFESSOR": "angelica@gmail"
14  },
15  {
16    "ID_PROFESSOR": 3,
17    "NOME_PROFESSOR": "JEFFERSON",
18    "EMAIL_PROFESSOR": "angelica@gmail"
19  }
20 ]
```

Observar que ele retornou os dados no formato JSON.

Observações:

1) Caso deseje criar um usuário para um banco que já existe, utilize:

CREATE LOGIN nomeusuario WITH PASSWORD = '1234';

CREATE USER nomeusuario FOR LOGIN nomeusuario;

GRANT SELECT, INSERT, UPDATE, DELETE

ON DATABASE::[PWEB]

TO nomeusuario;

2) Se acaso ocorrer algum erro, quando está usando na sua máquina, verifique se o SQLServer está aceitando conexões remotas e se está habilitado o TCP/IP (C:\Windows\SysWOW64\SQLServerManager15.msc para a versão 2019).

- 3) Na Fatec ao invés de usar o nome do servidor Apolo pode utilizar o IP dele.

### *8.6 Listando dados de uma tabela na página (através da view)*

Alterar o arquivo `professores.js` para que esses dados retornados sejam passados para a View e a ela deve exibi-los. No código substituir o `res.send` por `res.render` e nele colocar também um rótulo “profs” por exemplo e o JSON. O EJS permite recuperar as informações na view e lá dentro da view escrever também código JavaScript.

Dentro da view `professores` é possível recuperar as informações de “profs” como se fosse um array, sendo que cada abre e fecha chave representa um índice dentro do JSON.

```
module.exports = function(app){
  app.get('/informacao/professores', function(req,res){
    const sql = require('mssql');
    module.exports = function(){
      const sqlConfig = {
        user: 'BD22130**', // seu usuario
        password: '*****', // sua senha
        database: 'BD',
        server: 'APOLO',
        options: {
          encrypt: false,
          trustServerCertificate: true
        }
      }
    }
    return sql.connect(sqlConfig);
  })
  async function getProfessores() {
    try {
      const pool = await sql.connect(sqlConfig);
      const results = await pool.request().query('SELECT * from PROFESSORES');
      res.render('informacao/professores', { profs: results.recordset });
    } catch (err) {
      console.log(err);
    }
  }
  getProfessores();
};
}
```

Alterar a view **professores.ejs** conforme código abaixo para a impressão. É possível misturar o código HTML com JavaScript, basta usar os códigos `<%` e `%>`. Tudo que estiver dentro será interpretado.

O sinal de = é necessário quando um valor será impresso.

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <title> Professores Fatec Sorocaba</title>
</head>

<style>
  table#tabela1 tr td {
    /* Toda a tabela com fundo creme */
    background: #ffc;
  }

  table#tabela1 tr.cabeca td {
    background: #eee;
    /* Linhas com fundo cinza */
  }
</style>

<body>
  <table border="1px" cellpadding="5px" cellspacing="0"
    ID="tabela1">

    <tr class="cabeca">
      <td>Nome</td>
      <td>E-Mail</td>
    </tr>

    <% console.log('QUANTIDADE DE REGISTROS→' + profs.length); %>

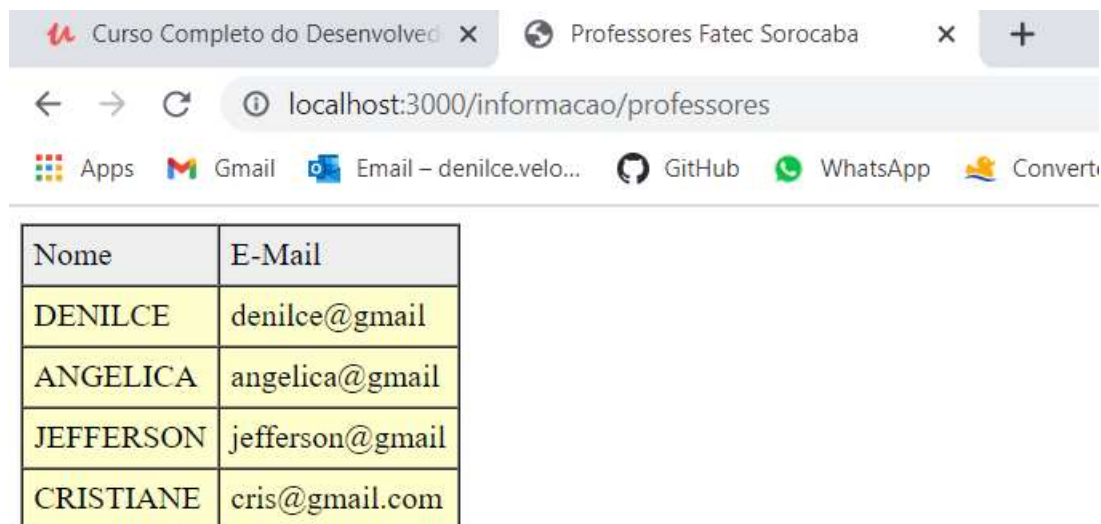
    <% for(var i = 0; i < profs.length; i++){ %>
      <tr>
        <td><%= profs[i].NOME_PROFESSOR %></td>
        <td><%= profs[i].EMAIL_PROFESSOR %></td>
      </tr>

      <% } %>

    </table>
  </body>
</html>
```

Testar a página dos professores, agora retornando os dados para a view.

*Figura 63: Testando a página dos professores (usando view dinâmica)*



The screenshot shows a web browser window with two tabs: 'Curso Completo do Desenvolvedor' and 'Professores Fatec Sorocaba'. The address bar shows 'localhost:3000/informacao/professores'. Below the browser window is a table with two columns: 'Nome' and 'E-Mail'. The table contains five rows of data.

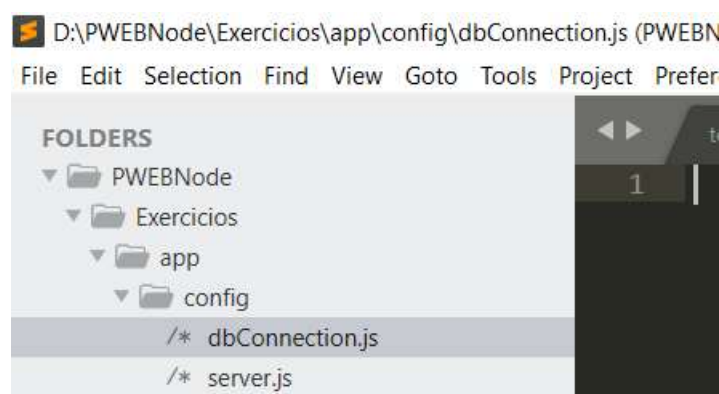
Nome	E-Mail
DENILCE	denilce@gmail
ANGELICA	angelica@gmail
JEFFERSON	jefferson@gmail
CRISTIANE	cris@gmail.com

### 8.7 Alterando a forma de conexão com o banco SQL Server

Esse exercício pretende modularizar a aplicação de forma a separar views do acesso de banco dados de forma a ficar mais parecido com o **MVC (Model View Control)**, a conexão do banco deve ficar separada das rotas.

**Criar um arquivo (dbConnection) dentro da pasta config.**

*Figura 64: Novo arquivo dbConnection para acesso ao banco*



Esse arquivo deve exportar o módulo de conexão do banco de dados (exports) e retornar como uma função.



```
var sql = require ('mssql');  
  
module.exports = function(){  
  const sqlConfig = {  
    user: 'BD22130**',  
    password: '*****',  
    database: 'BD',  
    server: 'APOLO',  
    options: {  
      encrypt: false,  
      trustServerCertificate: true  
    }  
  }  
  return sql.connect(sqlConfig);  
}
```

Acertar no arquivo **professores.js** para incluir o módulo de conexão ao banco de dados e executar a conexão através da função.

```
// para acessar o arquivo de config voltar 1 nivel  
  
var dbConnection = require('../config/dbConnection');  
  
module.exports = function(app){  
  app.get('/informacao/professores', function(req,res){  
    async function getProfessores() {  
      try {  
        const pool = await dbConnection(); // executando a funcao  
  
        const results = await pool.request().query('SELECT * from PROFESSORES');  
  
        res.render('informacao/professores',{profs : results.recordset});  
  
      } catch (err) {  
        console.log(err)  
      }  
    }  
  
    getProfessores();  
  });  
};
```

Testar novamente a página dos professores para ver se está ok.

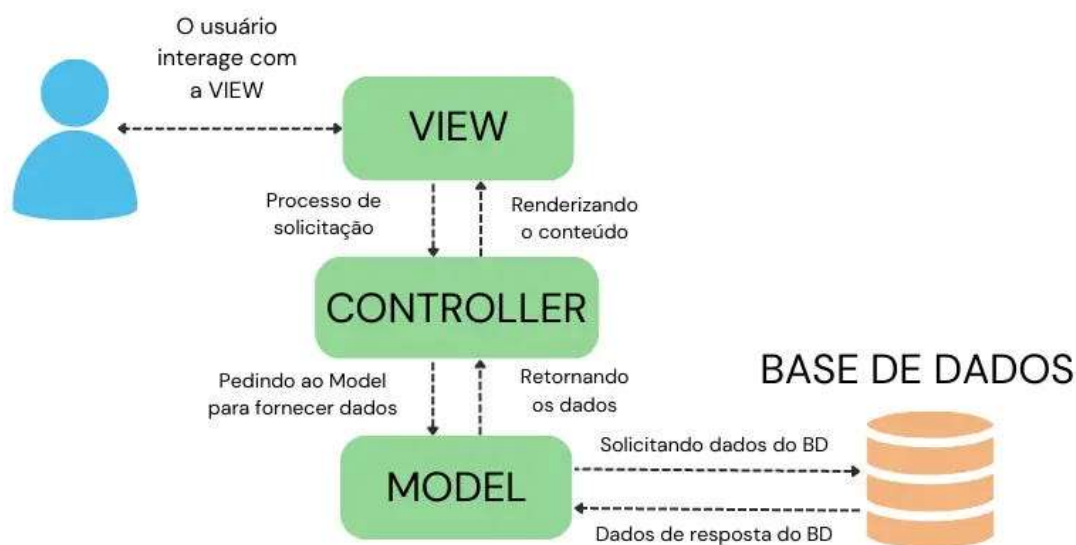
## 9. Separando a aplicação em Camadas

Você deve ter observado, que as separações ou reestruturações nos exercícios, apontam para o modelo MVC. A ideia é estruturar a aplicação de forma a aparecer mais com a estrutura MVC, que é um *Design Pattern* (modelo de projeto).

### 9.1 MVC

O MVC é uma sigla do termo em inglês Model (modelo), View (visão) e Controller (Controle) que facilita a troca de informações entre a interface do usuário aos dados no banco, fazendo com que as respostas sejam mais rápidas e dinâmicas.

Figura 65: Estrutura MVC



Fonte: <https://medium.com/@celionormando/arquitetura-mvc-e-princ%C3%ADpios-de-projeto-3d0b278ef910>

#### 9.1.1 Model ou Modelo

Essa camada também é conhecida como *Business Object Model* (objeto modelo de negócio), e é de sua responsabilidade gerenciar e controlar a forma como os dados se comportam por meio das funções, lógica e regras de negócios estabelecidas.

### **9.1.2 Controller ou Controlador**

A camada de controle é responsável por intermediar as requisições enviadas pelo View com as respostas fornecidas pelo Model, processando os dados que o usuário informou e repassando para outras camadas.

### **9.1.3 View ou Visão**

Essa camada é responsável por apresentar as informações de forma visual ao usuário, recursos ligados a aparência como mensagens, botões ou telas. O View está na linha de frente da comunicação com usuário e é responsável transmitir questionamentos ao controller e entregar as respostas obtidas ao usuário.

## **10. Melhorando a organização das rotas**

Para auxiliar na organização das rotas, serão utilizadas mais algumas bibliotecas.

### *10.1 Consign*

Essa biblioteca é um autoloader de scripts, aqui é sugerida **para facilitar o gerenciamento das rotas no Express**. Observe que uma biblioteca dessas pode ser muito útil quando se tem muitas telas, pois, quanto mais telas, mais rotas. **Sem Consign, as rotas precisariam ser carregadas no app.js uma a uma**, usando require.

Com o Consign, no arquivo server.js pode se fazer um require do módulo Consign e através da função include do Consign, incluir todas as rotas que estão dentro de app/routes, ou seja, no momento de carregar a aplicação ele “recupera” todas as rotas. Ele pode fazer também autoloader de views e arquivos de configuração, controllers, etc.

Na documentação do Express, existe também o Router, que permite criar manipuladores de rota da aplicação (<https://expressjs.com/pt-br/guide/routing.html>).

*DICA: Se você quiser saber como está o download de um item pelo npm, basta acessar <https://www.npmtrends.com>.*

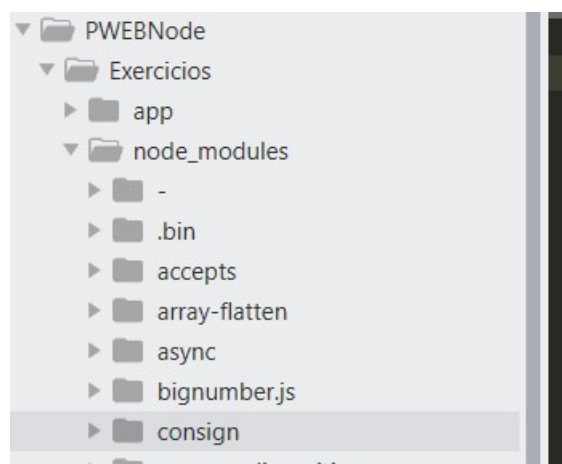
### *10.2 Instalação do Consign*

Para instalar o Consign usar:

```
npm install consign --save
```

Observe na pasta node\_modules que ele está aparecendo.

*Figura 66: node\_modules com o Consign*



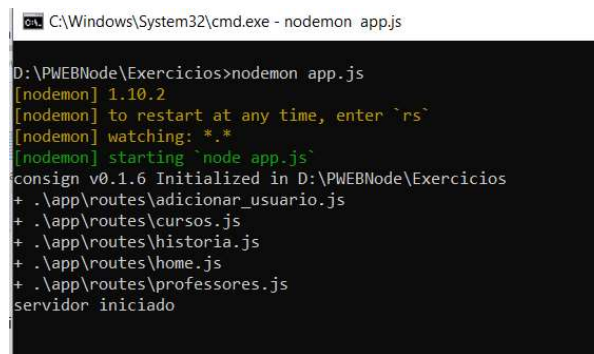
### *10.3 Inclusão do Consign no server.js*

Alterar o arquivo **server.js** para incluir o Consign, no futuro se tivermos novas rotas, não será necessário incluir todas as rotas no app.js pois o Consign carregará automaticamente.

```
var express = require('express');  
  
var consign = require('consign');  
  
var app = express();  
  
app.set('view engine', 'ejs');  
  
app.set('views', './app/views');  
  
consign().include('app/routes').into(app);  
  
module.exports = app;
```

Executar app.js → nodemon app.js

*Figura 67: Consign incluindo a pasta routes*



```
C:\Windows\System32\cmd.exe - nodemon app.js  
  
D:\PWEBNode\Exercicios>nodemon app.js  
[nodemon] 1.10.2  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching: *.*  
[nodemon] starting `node app.js`  
consign v0.1.6 Initialized in D:\PWEBNode\Exercicios  
+ .\app\routes\adicionar_usuario.js  
+ .\app\routes\cursos.js  
+ .\app\routes\historia.js  
+ .\app\routes\home.js  
+ .\app\routes\professores.js  
servidor iniciado
```

Alterar o arquivo **app.js**, agora não precisa colocar todas as rotas.

```
var app = require('./app/config/server');  
  
//COMENTAR  
/*var rotaHome = require('./app/routes/home');  
//rotaHome(app);  
  
var rotaAdicionarUsuario = require('./app/routes/adicionar_usuario');  
rotaAdicionarUsuario(app);  
  
var rotaHistoria = require('./app/routes/historia'); // só esta definindo  
rotaHistoria(app); // está executando
```

```
var rotaCursos = require('./app/routes/cursos'); // só esta definindo
rotaCursos(app); // está executando

var rotaProfessores = require('./app/routes/professores'); // só esta definindo
rotaProfessores(app); // está executando
*/
app.listen(3000, function(){
    console.log("servidor iniciado");
});
```

Executando o arquivo app.js

*Figura 68: Consign incluindo a pasta routes mesmo sem estarem no app.js*

```
D:\PWEBNode\Exercicios>nodemon app.js
[nodemon] 1.10.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
consign v0.1.6 Initialized in D:\PWEBNode\Exercicios
+ .\app\routes\adicionar_usuario.js
+ .\app\routes\cursos.js
+ .\app\routes\historia.js
+ .\app\routes\home.js
+ .\app\routes\professores.js
servidor iniciado
```

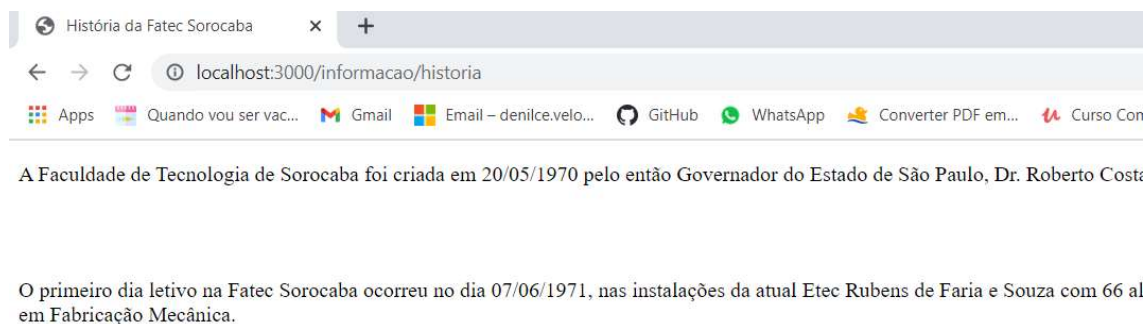
Testar a página principal

*Figura 69: Página principal depois do Consign*



Testar a página história

*Figura 70: Página história depois do Consign*



Tentar abrir a página dos professores, caso o seu banco SQL Server não estiver startado, vai apresentar erro.

*Figura 71: Página professores depois do Consign (sem “startar” banco de dados)*



#### 10.4 Restruturação da parte do banco de dados

No projeto, por enquanto apenas uma página acessa o banco de dados, a dos professores (professores.ejs), mas supondo muitas páginas acessassem dados, em todas elas teria o carregamento da conexão no arquivo das rotas? Ela está assim:

```
var dbConnection = require('../config/dbConnection');

module.exports = function(app){
  app.get('/informacao/professores', function(req,res){
    async function getProfessores() {
      try {
        const pool = await dbConnection(); // executando a funcao

        const results = await pool.request().query('SELECT * from PROFESSORES');

        res.render('informacao/professores',{profs : results.recordset});

      } catch (err) {
        console.log(err)
      }
    }

    getProfessores();
  });
};
```

O ideal é **levar essa conexão para a parte do autoload (carregado inicialmente).**  
Vamos acertar para ficar mais fácil a manutenção.

Primeiro acertar o arquivo **server.js** incluindo a pasta config também.

```
var express = require('express');
var consign = require('consign');

var app = express();

app.set('view engine', 'ejs');
app.set('views', './app/views');

// especificado qual arquivo ele deve executar porque dentro do config tem o server
// ele iria ficar executando o servidor toda hora
// precisa da extensao senao ele pensa que é um subdiretorio

consign({cwd:'app'}) // para incluir a pasta app
  .include('routes')
```



```
.then('config/dbConnection.js')  
.into(app);  
  
module.exports = app;
```

**\*\* Caso ocorra algum problema na execução do app, execute npm init novamente (Main ->> deixar app)**

Depois comentar a linha da conexão no arquivo **professores.js**, e acertar a forma que será executada a conexão somente quando a rota for requisitada.

```
//var dbConnection = require('../config/dbConnection');  
  
module.exports = function(app){  
  app.get('/informacao/professores', function(req,res){  
    async function getProfessores() {  
      try {  
        // como está recebendo o app  
        // já foi carregada no autoload e está sendo requisitada somente quando essa rota for  
        // acessada  
  
        var connection = app.config.dbConnection;  
  
        const pool = await connection;  
  
        const results = await pool.request().query('SELECT * from PROFESSORES');  
  
        res.render('informacao/professores',{profs : results.recordset});  
  
      } catch (err) {  
        console.log(err)  
      }  
    }  
  
    getProfessores();  
  });  
};
```

Observe que no arquivo dbConnection.js anterior, estava sendo feita a conexão, o que faz que **com toda vez que carregar a aplicação ele fará a conexão**, isso não é ideal. Para resolver esse problema será utilizado wrap (“embrulhar”), e esse export **não retornará mais um método mais uma variável**, alterar o arquivo **dbConnection.js**

```
var sql = require('mssql');

var connSQLServer = function(){
  const sqlConfig = {
    user: 'BD22130**',
    password: '*****',
    database: 'BD',
    server: 'APOLO',
    options: {
      encrypt: false,
      trustServerCertificate: true
    }
  }
  return sql.connect(sqlConfig);
}

// exportando a função e quando chamar a
// página ele conecta
module.exports = function(){
  console.log('O autoloader carregou o módulo de conexão com o bd');
  return connSQLServer;
}
```

Alterar **professores.js**

```
module.exports = function(app){
  app.get('/informacao/professores', function(req, res){
    async function getProfessores() {
      try {

        var connection = app.config.dbConnection;
        const pool = await connection(); //executar

        const results = await pool.request().query('SELECT * from
PROFESSORES');

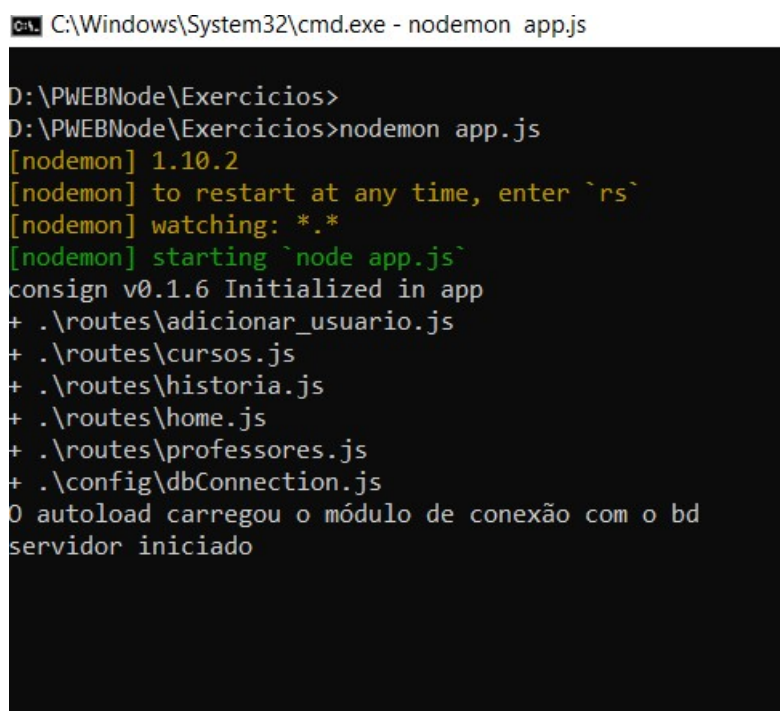
        res.render('informacao/professores', {profs : results.recordset});

      } catch (err) {
        console.log(err)
      }
    }
  })
}
```

```
    }  
  
    getProfessores();  
  });  
};
```

Recarregar o servidor com app.js

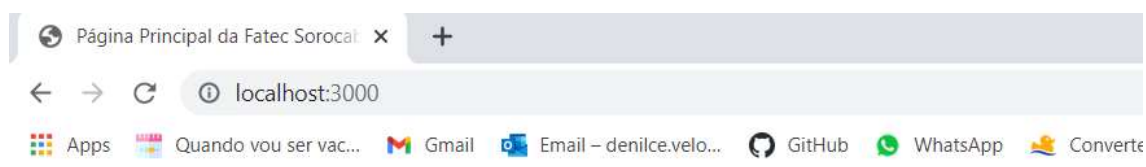
*Figura 71: Recarregando servidor com autoloader carregando routes*



```
C:\Windows\System32\cmd.exe - nodemon app.js  
  
D:\PWEBNode\Exercicios>  
D:\PWEBNode\Exercicios>nodemon app.js  
[nodemon] 1.10.2  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching: *.*  
[nodemon] starting `node app.js`  
consign v0.1.6 Initialized in app  
+ .\routes\adicionar_usuario.js  
+ .\routes\cursos.js  
+ .\routes\historia.js  
+ .\routes\home.js  
+ .\routes\professores.js  
+ .\config\dbConnection.js  
O autoloader carregou o módulo de conexão com o bd  
servidor iniciado
```

Carregar a página principal.

*Figura 72: Página principal com autoloader carregando routes*



## Bem vindo a página inicial da Fatec Sorocaba

Carregar a página dos professores.

*Figura 73: Página história com autoload carregando routes*

A screenshot of a web browser window showing a table of professors. The address bar shows 'localhost:3000/informacao/professores'. The browser's taskbar at the bottom shows icons for 'Apps', 'Quando vou ser vac...', 'Gmail', 'Email - denilce.velo...', and 'GitHub'.

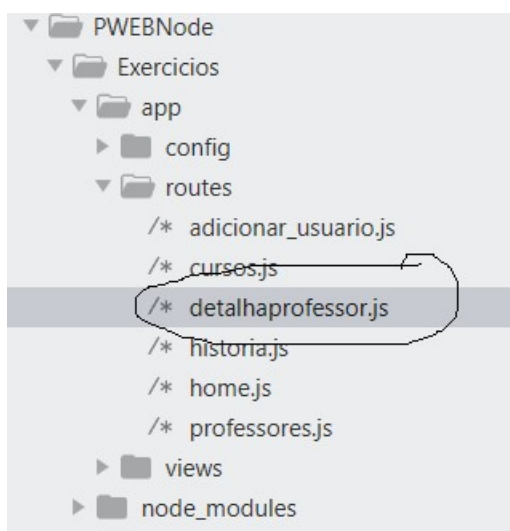
Nome	E-Mail
DENILCE	denilce@gmail
ANGELICA	angelica@gmail
JEFFERSON	jefferson@gmail
CRISTIANE	cris@gmail.com
CESAR	cesar@gmail.com
Levi	levi@gmail.com
DENILCe	denilce@gmail.com

### 10.5 Criando uma página para recuperar pelo ID

Nesse passo, será criada uma página para mostrar um professor pelo ID.

Primeiro criar uma rota **detalhaprofessor.js**, observar que não é necessário incluir essa rota no carregamento, pois ele fará isso no autoload da aplicação.

Figura 74: Rota detalhaprofessor.js



```
module.exports = function(app){
  app.get('/informacao/professores/detalhaprofessor', function(req,res){

    async function getProfessoresID() {
      try {
        var connection = app.config.dbConnection;

        const pool = await connection();

        const results = await pool.request().query('SELECT * FROM professores WHERE
id_professor = 1') // atenção para funcionar tem que existir o professor com esse ID

        res.render('informacao/professores/detalhaprofessor',{profs : results.recordset});

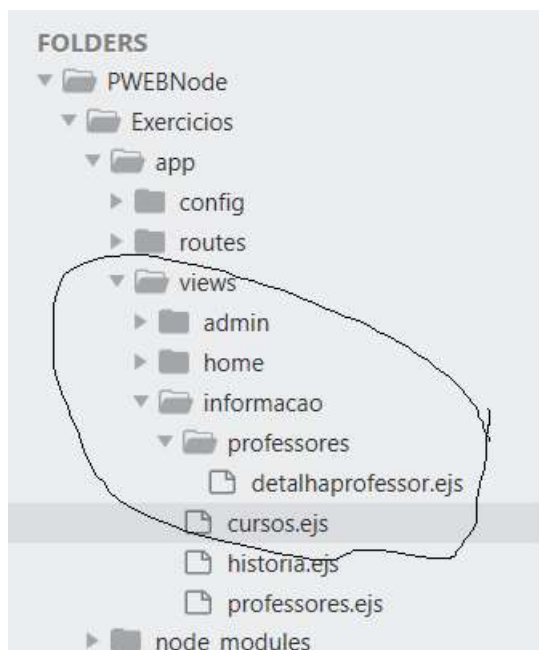
      } catch (err) {
        console.log(err)
      }
    }

    getProfessoresID();
  });
}
```

}

Criar a view `detalhaprofessor.ejs` dentro da pasta `informacao\professores`, observe que ele listará apenas um registro.

Figura 75: View `detalhaprofessor.ejs`



```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <title> Detalhe Professor Fatec Sorocaba</title>
</head>

<body>
  <table border="1px" cellpadding="5px" cellspacing="0" ID="tabela1">

    <tr class="cabeca">
      <td>ID</td>
      <td>Nome</td>
      <td>E-Mail</td>
    </tr>
    <tr>
      <td><%= profs[0].ID_PROFESSOR %></td>
      <td><%= profs[0].NOME_PROFESSOR %></td>
      <td><%= profs[0].EMAIL_PROFESSOR %></td>
```

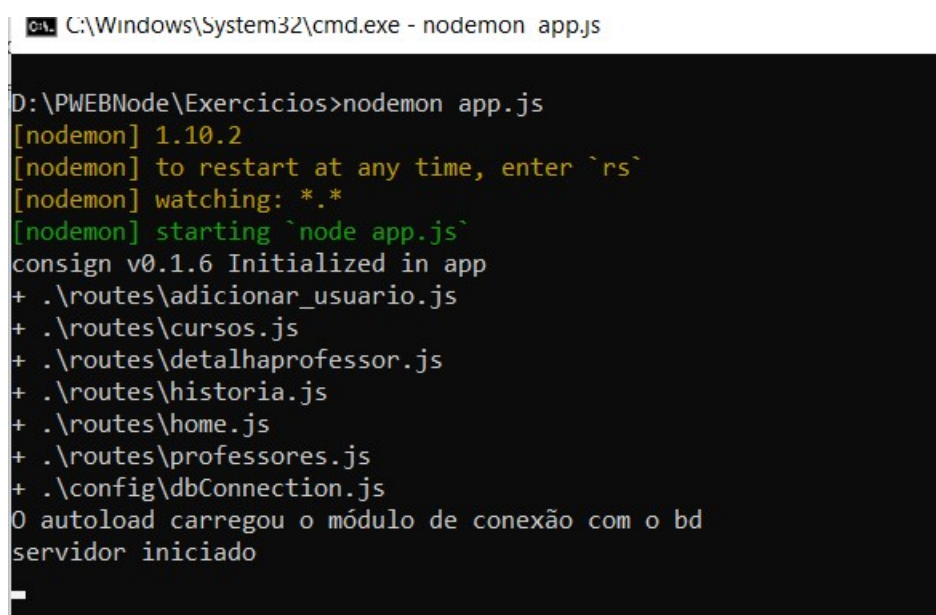
```
</tr>

</table>
</body>

</html>
```

Recarregar o servidor, observar que ele vai carregar a rota do detalhaprofessor.

*Figura 76: Servidor carregando rota detalhaprofessor.js*



```
C:\Windows\System32\cmd.exe - nodemon app.js

D:\PWEBNode\Exercicios>nodemon app.js
[nodemon] 1.10.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
consign v0.1.6 Initialized in app
+ .\routes\adicionar_usuario.js
+ .\routes\cursos.js
+ .\routes\detalhaprofessor.js
+ .\routes\historia.js
+ .\routes\home.js
+ .\routes\professores.js
+ .\config\dbConnection.js
O autoloader carregou o módulo de conexão com o bd
servidor iniciado
```

Testar a página informacao/professores

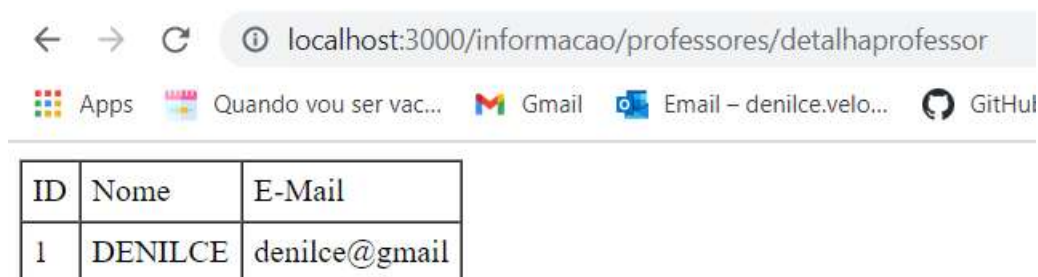
*Figura 77: Teste página professores*



Nome	E-Mail
DENILCE	denilce@gmail
ANGELICA	angelica@gmail
JEFFERSON	jefferson@gmail
CRISTIANE	cris@gmail.com
CESAR	cesar@gmail.com
Levi	levi@gmail.com
DENILCe	denilce@gmail.com

Testar a página [informacao/professores/detalhaprofessor](localhost:3000/informacao/professores/detalhaprofessor)

Figura 78: Teste página detalhaprofessor



ID	Nome	E-Mail
1	DENILCE	denilce@gmail

## 11 Implementando os Models

Continuando o objetivo de estruturar a aplicação como MVC, o próximo passo é implementar os Models, a parte relacionada ao banco de dados, é importante porque vamos separar a parte da lógica de dados da aplicação. As regras de negócio vão para dentro de Models.

Observe, por exemplo, que na rota de professores e detalha professor, está sempre conectando ao banco e listando os dados de acordo com o desejado,



pode ser que os comandos de recuperação dos dados sejam bem parecidos, mas estão sendo repetidos, podemos melhorar essa parte.

**Criar um diretório/pasta models dentro de app.**

*Figura 79: Criação pasta models*



Criar um arquivo **professormodel.js** dentro da pasta models, ele representar a entidade no banco de dados. Criar uma função dentro do contexto do módulo e usar o recurso do this para retornar os dados da função.

```
module.exports = function(){  
  
  this.getProfessores = function(connection, callback){  
    connection.query('select * from professores', callback);  
  }  
  
  this.getProfessor = function(connection, callback){  
    connection.query('select * from professores WHERE id_professor=1', callback);  
  }  
  
  return this;  
  
}
```

Alterar o arquivo **server.js** para incluir o Models no autoload.

```
var express = require('express');
var consign = require('consign');

var app = express();

app.set('view engine', 'ejs');
app.set('views', './app/views');

// especificado qual arquivo ele deve executar porque dentro do config tem o server
// ele iria ficar executando o server toda hora
// precisa da extensão, senao ele pensa que é um subdiretorio

consign({cwd:'app'}) // para incluir a pasta app
  .include('routes')
  .then('config/dbConnection.js')
  .then('models')
  .into(app);

module.exports = app;
```

Alterar a rota **professores.js** para carregar do Models

```
module.exports = function(app){
  app.get('/informacao/professores', function(req,res){
    async function getProf() {
      try {
        var connection = app.config.dbConnection;
        const pool = await connection();

        var professoresModel = app.models.professormodel; // variável que recupera a
função exporta

        //executar a função
        // tem passar a conexao e o callback
        professoresModel.getProfessores(pool, function(error, results){
          res.render('informacao/professores', { pros : results.recordset });
        });
      } catch (err) {
        console.log(err)
      }
    }
    getProf();
  });
}
```

Alterar a rota **detalhaprofessor.js** para carregar do Models, para não ficar precisando fazer require de todos os módulos, porque em uma aplicação normalmente tem várias tabelas (model).

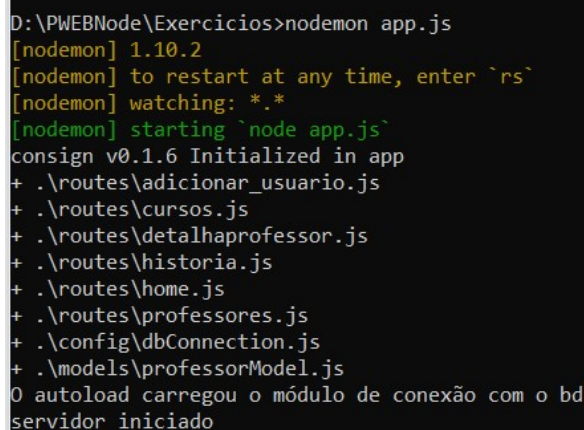
```
module.exports = function(app){
  app.get('/informacao/professores/detalhaprofessor', function(req,res){
    async function getProfessoresID() {
      try {
        var connection = app.config.dbConnection;
        const pool = await connection();

        var professoresModel = app.models.professormodel;

        professoresModel.getProfessor(pool, function(error, results){
          res.render('informacao/professores/detalhaprofessor', { pros : results.recordset });
        });
      } catch (err) {
        console.log(err)
      }
    }
    getProfessoresID();
  });
}
```

Carregar novamente o servidor para verificar se está carregando os models.

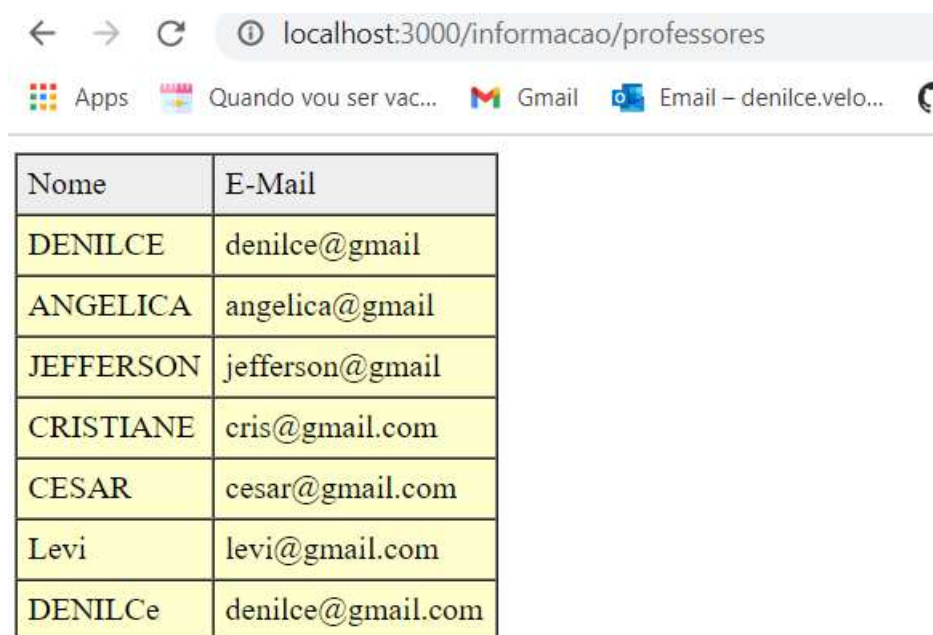
*Figura 80: Servidor carregando a pasta models*



```
D:\PWEBNode\Exercicios>nodemon app.js
[nodemon] 1.10.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
consign v0.1.6 Initialized in app
+ .\routes\adicionar_usuario.js
+ .\routes\cursos.js
+ .\routes\detalhaprofessor.js
+ .\routes\historia.js
+ .\routes\home.js
+ .\routes\professores.js
+ .\config\dbConnection.js
+ .\models\professorModel.js
O autoload carregou o módulo de conexão com o bd
servidor iniciado
```

Testar a página dos professores

*Figura 81: Página dos professores com servidor carregando a pasta models*



Nome	E-Mail
DENILCE	denilce@gmail
ANGELICA	angelica@gmail
JEFFERSON	jefferson@gmail
CRISTIANE	cris@gmail.com
CESAR	cesar@gmail.com
Levi	levi@gmail.com
DENILCe	denilce@gmail.com

Testar a página detalha professor

Figura 82: Página detalhaprofessor com servidor carregando a pasta models



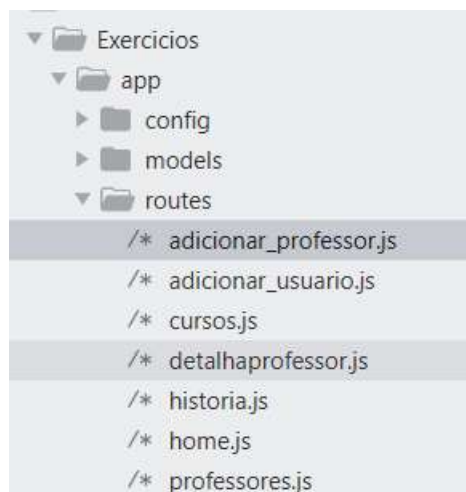
ID	Nome	E-Mail
1	DENILCE	denilce@gmail

## 12 Criação do formulário de Inclusão

Aqui será criado um formulário para a inclusão de um professor.

Na pasta app\routes incluir arquivo (rota) **adicionar\_professor.js**

Figura 83: Rota adicionar\_professor.js



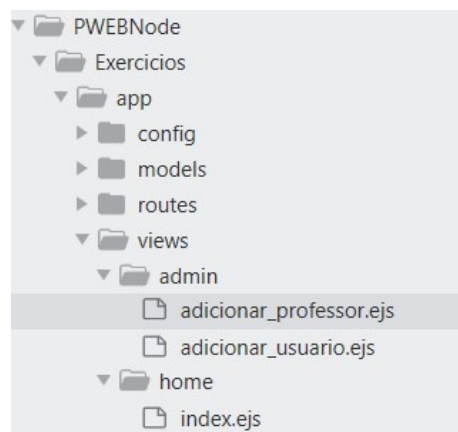
Alterar o **adicionar\_professor.js**. Observe que foi colocado um post para quando o formulário for submetido.

```
module.exports = function(application){
  application.get('/admin/adicionar_professor', function(req,res){
    res.render('admin/adicionar_professor');
  });

  application.post('/professor/salvar', function(req,res){
    res.send("Salvo!!!!");
  });
}
```

Na pasta view\admin incluir arquivo (página) adicionar\_professor.ejs

*Figura 84: Arquivo (página) adicionar\_professor.ejs*



Alterar arquivo **adicionar\_professor.ejs**

```
<!DOCTYPE html>
  <html lang="pt-br">
    <head>
      <meta charset="utf-8"/>
      <title>Cadastro de Professores</title>
    </head>
    <body>

      <h1>Adicionar Professor</h1>

      <form action="/professor/salvar" method="post">
        <label>Nome</label>
        <input type="text" id="nome" name="nome_professor"
placeholder="Nome do Professor" />
        <br/>
        <br/>
        <label>E-mail</label>
        <input type="email" id="email" name="email_professor" placeholder="E-mail do
Professor" />

        <br/>
        <br/>

        <input type="submit" value="Enviar" />
      </form>

    </body>
  </html>
```

Teste a página adicionar\_professor e tente salvar.

Figura 85: Página adicionar\_professor



← → ↻ ⓘ localhost:3000/admin/adicionar\_professor

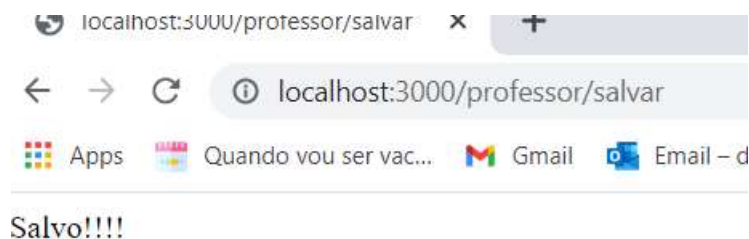
Apps Quando vou ser vac... Gmail Email – denilce.velo...

## Adicionar Professor

Nome

E-mail

*Figura 86: Retorno do Enviar (Salvar) professor*



**ATENÇÃO:** Observe que ainda não está incluindo no dado no banco de dados (teste informacao/professores), para fazer a efetiva inclusão no banco de dados serão necessários mais alguns ajustes.

### 12.1 Body-Parser

É um módulo que extrai a parte do corpo inteiro de um fluxo de solicitação de entrada e o expõe em req.body. Ele analisa os dados codificados JSON, buffer, string e URL enviados usando a solicitação HTTP POST. É possível fazer isso sem utilizar esse módulo, mas usá-lo poupará trabalho.

O middleware era parte do Express.js anterior, mas agora você precisa instalá-lo separadamente.

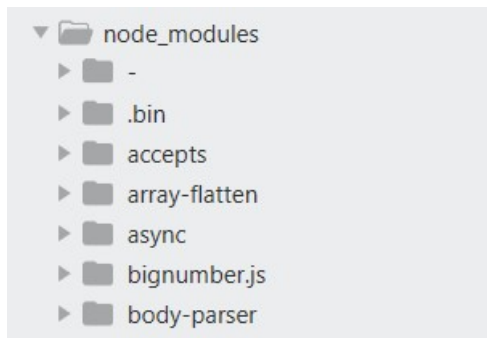
#### 12.1.1 Instalação do *Body-Parser*

Para instalar o Body-Parser digitar o comando:

```
npm install body-parser --save
```

Verifique se ele está aparecendo no node\_modules.

*Figura 87: Body-Parser no node\_modules*



### **12.1.2 Alteração do server e model para inserir no banco de dados**

Alterar o arquivo **server.js**. Primeiro incluir o módulo do Body-Parser. Como o Body-Parser é um middleware (software que se encontra entre o sistema operacional e os aplicativos nele executados) e vai atuar nos objetos de requisição e resposta, então ele precisa ficar antes do carregamento do Consign.

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

var app = express();

app.set('view engine', 'ejs');
app.set('views', './app/views');

// para ele entender o formato da URL
app.use(bodyParser.urlencoded({extended: true}));

consign({cwd:'app'}) // para incluir a pasta app
  .include('routes')
  .then('config/dbConnection.js')
  .then('models')
  .into(app);

module.exports = app;
```

Alterar o arquivo **adicionar\_professor.js** para mostrar os dados enviados pelo html.

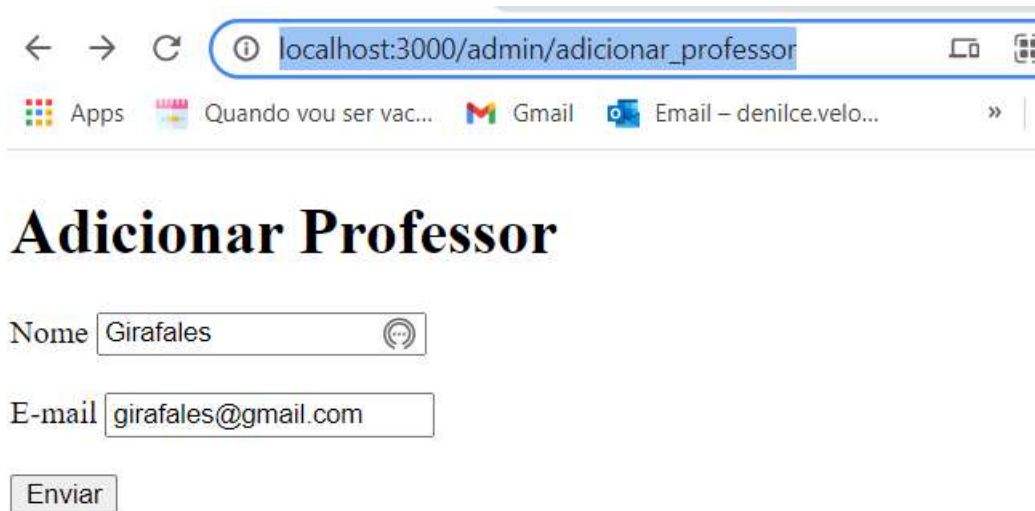
```
module.exports = function(application){
  application.get('/admin/adicionar_professor', function(req,res){
```



```
res.render('admin/adicionar_professor');  
});  
  
application.post('/professor/salvar', function(req,res){  
    res.send(req.body);  
    console.log(req.body) // só para verificar o que está voltando  
});  
}
```

Execute a página adicionar\_professor e tentar salvar (enviar) os dados.

*Figura 88: Carregando página adicionar\_professor*



← → ↻ ⓘ localhost:3000/admin/adicionar\_professor 📱 ☰

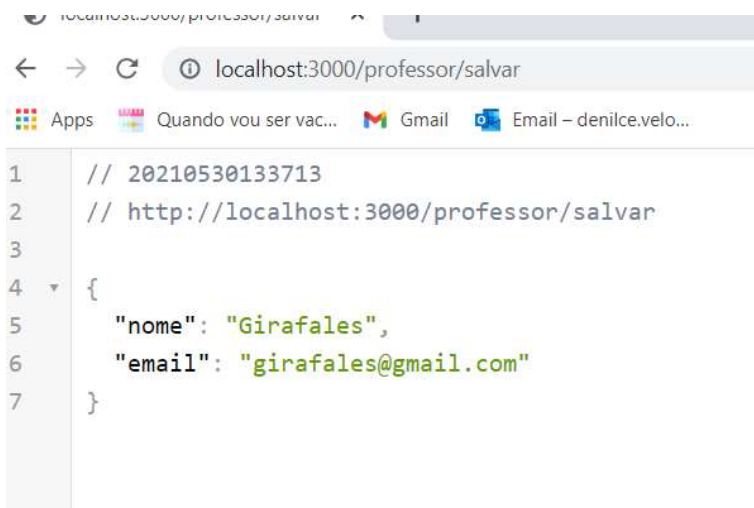
Apps Quando vou ser vac... Gmail Email – denilce.velo... »

## Adicionar Professor

Nome

E-mail

*Figura 89: Retorno do Salvar de adicionar\_professor*



```
1 // 20210530133713
2 // http://localhost:3000/professor/salvar
3
4 {
5   "nome": "Girafales",
6   "email": "girafales@gmail.com"
7 }
```

Embora ele esteja retornando o JSON dos dados, agora são precisos alguns ajustes para enviar ao banco. O próximo passo é alterar a rota `adicionar_professor.js` e usar `"req.body"`, requisição que está sendo recuperada e inserir um registro no banco de dados.

Alterar o arquivo `adicionar_professor.js`

```
module.exports = function(application){
  application.get('/admin/adicionar_professor', function(req,res){
    res.render('admin/adicionar_professor');
  });
  application.post('/professor/salvar', function(req,res){

    async function getAdcProfessor(){
      try {
        var professor = req.body;

        var connection = application.config.dbConnection;
        const pool = await connection();

        var professoresModel = application.models.professormodel;

        //usando uma funcao de callback e informar quem devemos salvar, no caso professor

        professoresModel.salvarProfessor(professor,pool, (error, results)=>{
          // após inserir redireciona o navegador para outra página
          // se der erro na inclusao criar um erro 500 --> nao sabe o que significa

          if(error){
            console.log('Erro ao inserir no banco:' + error);
            res.status(500).send(error);
          } else {
            console.log('professor criado!!!');
            res.redirect('/informacao/professores');
          }
        })
      }
    }
  })
}
```

```
    });  
    } catch (error) {  
        console.log(error);  
    }  
    }  
    getAdcProfessor();  
    });  
}
```

Implementar a função **salvarProfessor** no Model **professormodel.js**:

```
module.exports = function(){  
  
    this.getProfessores = function(connection, callback){  
        connection.query('select * from professores', callback);  
    }  
  
    this.getProfessor = function(connection, callback){  
        connection.query('select * from professores WHERE id_professor=1', callback);  
    }  
  
    this.salvarProfessor = function(professor, connection, callback){  
        connection.query("INSERT INTO professores  
(NOME_PROFESSOR,EMAIL_PROFESSOR) VALUES ('"+ professor.nome_professor  
+"','"+ professor.email_professor+"')", callback);  
    }  
  
    return this;  
}
```

Recarregar o servidor

*Figura 90: Servidor recarregado*

```
C:\Windows\System32\cmd.exe - node app.js

D:\PWEBNode\Exercicios>node app.js
consign v0.1.6 Initialized in app
+ .\routes\adicionar_professor.js
+ .\routes\adicionar_usuario.js
+ .\routes\cursos.js
+ .\routes\detalhaprofessor.js
+ .\routes\historia.js
+ .\routes\home.js
+ .\routes\professores.js
+ .\config\dbConnection.js
+ .\models\professorModel.js
O autoload carregou o módulo de conexão com o bd
servidor iniciado
```

Testar a página adicionar\_professor e salvar (enviar) os dados.

*Figura 91: Página adicionar\_professor*



← → ↻ ⓘ localhost:3000/admin/adicionar\_professor

Apps Quando vou ser vac... Gmail Email – denilce.ve

## Adicionar Professor

Nome

E-mail

Deve aparecer um resultado assim, pois após gravar ele está chamando a página professores.

Figura 92: Página dos professores



Nome	E-Mail
DENILCE	denilce@gmail
ANGELICA	angelica@gmail
JEFFERSON	jefferson@gmail
CRISTIANE	cris@gmail.com
CESAR	cesar@gmail.com
Levi	levi@gmail.com
DENILCe	denilce@gmail.com
NOVO PROFESSOR	NOVO@TESTE.COM
SEILA	SEILA@GMAIL.COM
Pardal	pardal@gmail.com

### 13 Crud completo da aplicação

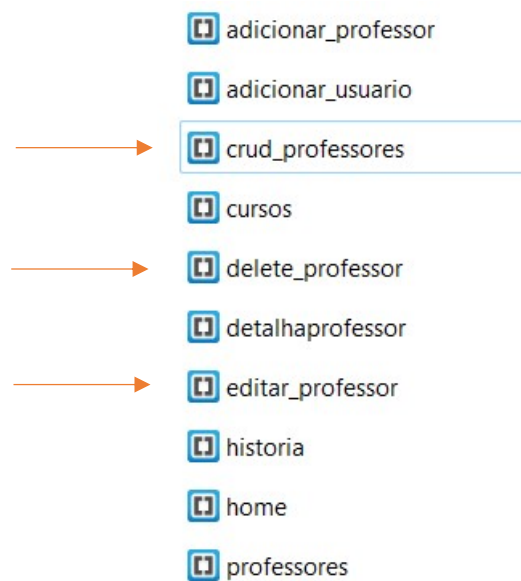
Os próximos passos são para terminar de fazer o CRUD completo da aplicação:

Alterar arquivo **professormodel.js**

```
module.exports = function(){  
  
  this.getProfessores = function(connection, callback){  
    connection.query('select * from professores', callback);  
  }  
  
  this.getProfessor = function(connection, callback){  
    connection.query('select * from professores WHERE id_professor=1', callback);  
  }  
}
```

```
this.getProfessorPorId = function(id_professor, connection, callback){  
    connection.query(`select * from professores WHERE id_professor=${id_professor}`,  
callback);  
}  
  
this.salvarProfessor = function(professor, connection, callback){  
    connection.query(`INSERT INTO professores (NOME_PROFESSOR,EMAIL_PROFESSOR)  
VALUES ('${professor.nome_professor}','${professor.email_professor}')`, callback);  
}  
  
this.deletarProfessor = function(professor, connection, callback){  
    connection.query(`DELETE FROM professores WHERE  
ID_PROFESSOR=${professor.id_professor}`, callback);  
}  
  
this.editarProfessor = function(professor, connection, callback){  
    connection.query(`UPDATE professores SET NOME_PROFESSOR =  
'${professor.nome_professor}', EMAIL_PROFESSOR = '${professor.email_professor}' WHERE  
ID_PROFESSOR=${professor.id_professor}`, callback);  
}  
  
return this;  
}
```

Criar as rotas:



**Criar** crud\_professores.js

```
module.exports = function(app){  
  app.get('/admin/crud_professores', function(req,res){  
  
    async function getProf() {  
      try {  
        var connection = app.config.dbConnection;  
        const pool = await connection();  
  
        var professoresModel = app.models.professormodel;// variável que recupera a função  
        exporta  
  
        professoresModel.getProfessores(pool, function(error, results){  
          res.render('admin/crud_professores', { profs : results.recordset });  
        });  
  
      } catch (err) {  
        console.log(err)  
      }  
    }  
  })  
}
```

```
}  
  
getProf();  
});  
}
```

**Criar** delete\_professor.js

```
module.exports = function(app){  
  app.post('/professor/deletar', function(req,res){  
  
    async function deleteProf(){  
      try {  
        var professor = req.body;  
  
        var connection = app.config.dbConnection;  
        const pool = await connection();  
  
        var professoresModel = app.models.professormodel;  
  
        professoresModel.deletarProfessor(professor,pool, (error, results)=>{  
  
          if(error){  
            console.log('Erro ao deletar do banco:' + error);  
            res.status(500).send(error);  
          } else {  
            console.log('professor deletado!!!');  
            res.redirect('/admin/crud_professores');  
          }  
        });  
      } catch (error) {  
        console.log(error);  
      }  
    }  
  });  
}
```



```
    }  
  }  
  deleteProf();  
});  
}
```

#### **Criar** **editar\_Professor.js**

```
module.exports = function(application){  
  application.get('/admin/editar_professor', function(req,res){  
  
    async function getProfessorPorId() {  
      try {  
        var id_professor = req.query.id;  
  
        var connection = application.config.dbConnection;  
        const pool = await connection();  
  
        var professoresModel = application.models.professormodel;  
  
        professoresModel.getProfessorPorId(id_professor, pool, function(error, results){  
          res.render('admin/editar_professor', { profs : results.recordset });  
        });  
  
      } catch (err) {  
        console.log(err)  
      }  
    }  
  
    getProfessorPorId();  
  });  
}
```

```
application.post('/professor/editar', function(req,res){

    async function editarProfessor(){
        try {
            var professor = req.body;

            var connection = application.config.dbConnection;
            const pool = await connection();

            var professoresModel = application.models.professormodel;

            // //usando uma funcao de callback e informar quem devemos salvar, no caso
professor

            professoresModel.editarProfessor(professor,pool, (error, results)=>{
                // após inserir redireciona o navegador para outra página
                // se der erro na inclusao criar um erro 500 --> nao sabe o que significa

                if(error){
                    console.log('Erro ao inserir no banco:' + error);
                    res.status(500).send(error);
                } else {
                    console.log('professor editado!!!');
                    res.redirect('/admin/crud_professores');
                }
            });
        } catch (error) {
            console.log(error);
        }
    }
}
```

```
    }  
  }  
  editarProfessor();  
});  
}
```

**Alterar** o arquivo **adicionar\_professor.js**

```
module.exports = function(application){  
  application.get('/admin/adicionar_professor', function(req,res){  
    res.render('admin/adicionar_professor');  
  });  
  
  application.post('/professor/salvar', function(req,res){  
  
    async function getAdcProfessor(){  
      try {  
        var professor = req.body;  
  
        var connection = application.config.dbConnection;  
        const pool = await connection();  
  
        var professoresModel = application.models.professormodel;  
  
        professoresModel.salvarProfessor(professor,pool, (error, results)=>{  
          if(error){  
            console.log('Erro ao inserir no banco:' + error);  
            res.status(500).send(error);  
          } else {  
            console.log('professor criado!!!');  
            res.redirect('/admin/crud_professores');  
          }  
        })  
      }  
    }  
  })  
}
```

```
    });  
    } catch (error) {  
        console.log(error);  
    }  
}  
getAdcProfessor();  
});  
}
```

**Alterar** professormodel.js

```
module.exports = function(){  
    this.getProfessores = function(connection, callback){  
        connection.query('select * from professores', callback);  
    }  
    this.getProfessor = function(connection, callback){  
        connection.query('select * from professores WHERE id_professor=1', callback);  
    }  
  
    this.getProfessorPorId = function(id_professor, connection, callback){  
        connection.query('select * from professores WHERE id_professor=${id_professor}',  
        callback);  
    }  
  
    this.salvarProfessor = function(professor, connection, callback){  
        connection.query(`INSERT INTO professores (NOME_PROFESSOR,EMAIL_PROFESSOR)  
VALUES ('${professor.nome_professor}','${professor.email_professor}')`, callback);  
    }  
  
    this.deletarProfessor = function(professor, connection, callback){  
        connection.query(`DELETE FROM professores WHERE  
ID_PROFESSOR=${professor.id_professor}`, callback);  
    }  
}
```

```
this.editarProfessor = function(professor, connection, callback){  
    connection.query(`UPDATE professores SET NOME_PROFESSOR =  
'${professor.nome_professor}', EMAIL_PROFESSOR = '${professor.email_professor}' WHERE  
ID_PROFESSOR=${professor.id_professor} `, callback);  
}  
  
return this;  
  
}
```

**Criar** os seguintes arquivos na pasta admin:

**crud\_professores.ejs**

```
<!DOCTYPE html>  
<html lang="pt-br">  
  
<head>  
    <meta charset="UTF-8">  
    <title> Professores Fatec Sorocaba</title>  
</head>  
  
<style>  
    table#tabela1 tr td {  
        /* Toda a tabela com fundo creme */  
        background: #ffc;  
    }  
  
    table#tabela1 tr.cabeca td {  
        background: #eee;  
        /* Linhas com fundo cinza */  
    }  
</style>
```

```
<body>

<h1>Lista de Professores</h1>

<button style="margin-bottom: 15px; padding: 5px 15px;">
  <a style="text-decoration: none;"
href="http://localhost:3000/admin/adicionar_professor">Adicionar professor</a>
</button>

<table border="1px" cellpadding="5px" cellspacing="0" ID="tabela1">

  <tr class="cabeca">
    <td>Nome</td>
    <td>E-Mail</td>
    <td>Operações</td>
  </tr>

  <% for(var i = 0; i < profs.length; i++){ %>
  <tr>
    <td><%= profs[i].NOME_PROFESSOR %></td>
    <td><%= profs[i].EMAIL_PROFESSOR %></td>
    <td>
      <button style="padding: 5px 15px; width: 100%; margin-bottom: 5px;">
        <a style="text-decoration: none; color: inherit;"
href="http://localhost:3000/admin/editar_professor?id=<%= profs[i].ID_PROFESSOR
%>">Editar</a>
      </button>

      <form action="/professor/deletar" method="post">
        <input type="hidden" id="id_professor" name="id_professor" value="<%=
profs[i].ID_PROFESSOR %>" />
        <button style="padding: 5px 15px; width: 100%;" type="submit">Deletar</button>
```

```
        </form>

    </td>
</tr>

<% } %>

</table>
</body>
```

**editar\_professor.ejs**

```
<!DOCTYPE html>

<html lang="pt-br">
    <head>
        <meta charset="utf-8"/>
        <title>Edição de Professor</title>
    </head>
    <body>

        <h1>Editar Professor</h1>

        <p>ID: <%= profs[0].ID_PROFESSOR %></p>
        <p>NOME: <%= profs[0].NOME_PROFESSOR %></p>
        <p>EMAIL: <%= profs[0].EMAIL_PROFESSOR %></p>

        <form action="/professor/editar" method="post">
            <input type="hidden" id="id_professor" name="id_professor" value="<%=
profs[0].ID_PROFESSOR %>" />
            <label>Nome</label>
            <input type="text" id="nome" name="nome_professor" placeholder="Nome do
Professor" />
            <br/>
```

```
<br/>

<label>E-mail</label>

<input type="email" id="email" name="email_professor" placeholder="E-mail do
Professor" />


<br/>

<br/>

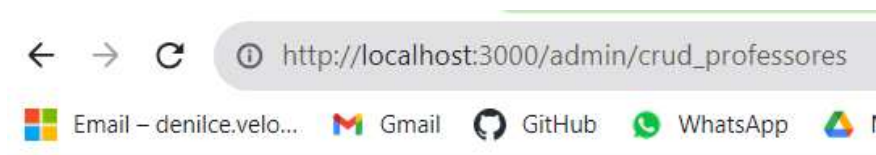
<input type="submit" value="Enviar" />
</form>

</body>
</html>
```

Executar a aplicação e testar essas novas rotas.

*Figura 93: Página admin/crud\_professores*





## Lista de Professores

[Adicionar professor](#)

Nome	E-Mail	Operações
DENILCE	denilce@gmail.com	<div>Editar</div> <div>Deletar</div>
ANGELICA	angelica@gmail.com	<div>Editar</div> <div>Deletar</div>
JEFFERSON	jefferson@gmail.com	<div>Editar</div> <div>Deletar</div>
CRISTIANE	cris@gmail.com	<div>Editar</div> <div>Deletar</div>
Dimas	dimas@gmail.com	<div>Editar</div> <div>Deletar</div>

**Desafio:** Chegou até aqui? Parabéns, então agora tente fazer o cadastro do usuário.

## 14. Exercício de Criação de API Rest Local

Criar uma pasta APIS dentro de PWEBNode.

Para fazer esse exercício serão necessários o Express e o Cors.

### CORS

O cross-origin é uma especificação que define meios para que um recurso do servidor seja acessado remotamente via web. É responsável por fornecer ao Express um middleware que permite lidar com requisições externas.

Instalar os pacotes do Cors

```
npm install express
```

```
npm install cors
```

Criar um arquivo chamado "server.js" e digite o código abaixo:

```
const express = require('express');  
const app = express();  
const cors = require('cors');  
const port = 3000;  
// Middleware para permitir CORS  
app.use(cors());  
// Middleware para permitir JSON  
app.use(express.json());  
// Lista de tarefas simples  
let tarefas = [];  
  
// Rota para obter todas as tarefas  
app.get('/tarefas', (req, res) => {
```

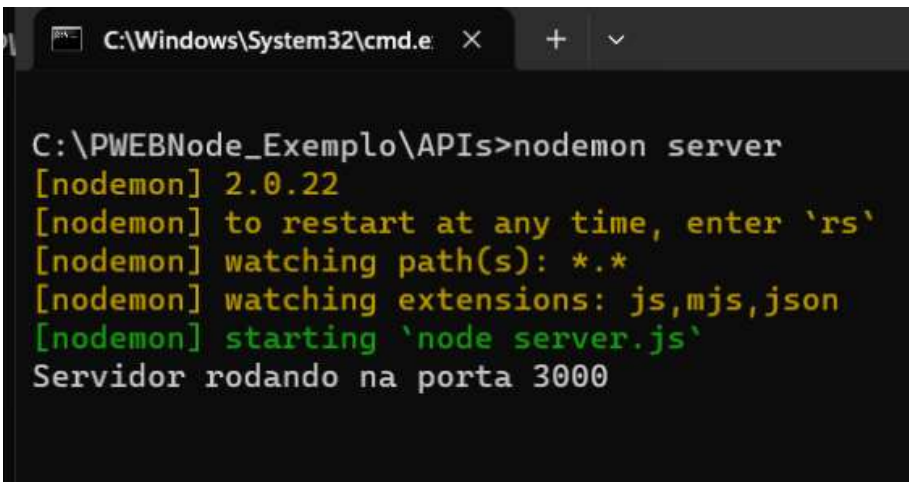
```
    res.json(tarefas);
  });

  // Rota para adicionar uma nova tarefa
  app.post('/tarefas', (req, res) => {
    const { tarefa } = req.body;
    tarefas.push(tarefa);
    res.status(201).json({ message: 'Tarefa adicionada com sucesso' });
  });

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

Testar o arquivo server.js

*Figura 94: Executando server.js*



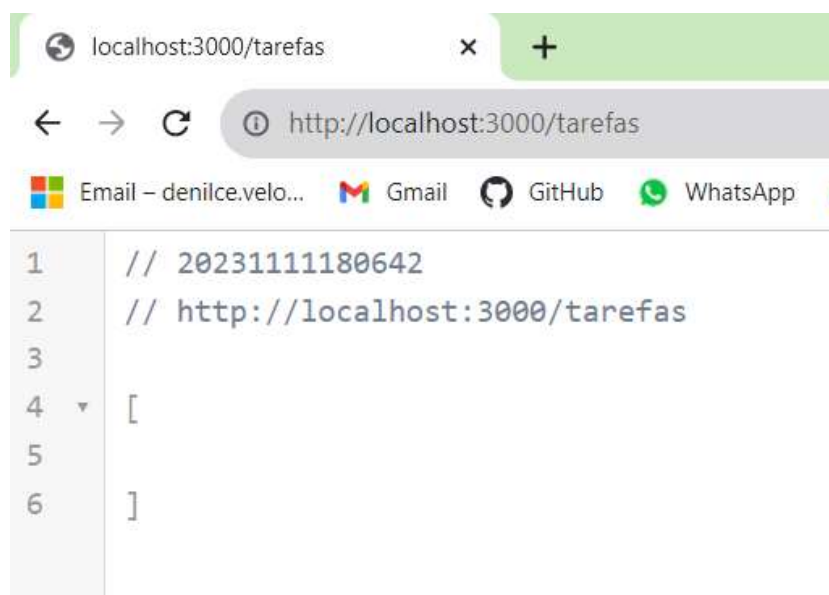
```
C:\Windows\System32\cmd.e X + v

C:\PWEBNode_Exemplo\APIs>nodemon server
[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Servidor rodando na porta 3000
```

Ir até o navegador e pesquise pelo seguinte End-Point:

<http://localhost:3000/tarefas>

*Figura 95: Pesquisando End-Point*



Se aparecer um [] ou JSON vazio, é porque a API foi criada com sucesso, para testar você pode criar um arquivo HTML, **digitar o código abaixo (executar a página clicando nela)** e tentar enviar através do formulário uma ou mais tarefas, toda vez que executar o botão de envio um método POST vai ser executado na API, agora basta dar um F5 na página da API e as tarefas serão visualizadas no formato JSON.

Criar um arquivo chamado "testeapi.html" e digite o código abaixo:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="utf-8">
    <title>Adicionar Tarefa</title>
  </head>

  <body>
    <h1>Adicionar Tarefa</h1>
    <form id="tarefa-form">
```

```
<label for="tarefa">Tarefa:</label>

<input type="text" id="tarefa" name="tarefa" placeholder="Digite uma
tarefa" required>

<button type="submit">Adicionar</button>

</form>

<script>
    document.addEventListener('DOMContentLoaded', function () {
        const form = document.getElementById('tarefa-form');
        const endpoint = 'http://localhost:3000/tarefas'; // Endpoint
da AP

        form.addEventListener('submit', function (e) {
            e.preventDefault();
            const tarefaInput =
document.getElementById('tarefa');

            const enviaTarefa = tarefaInput.value;

            // Realizar uma solicitação POST para a API
            fetch(endpoint, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({ tarefa: enviaTarefa }),
            })

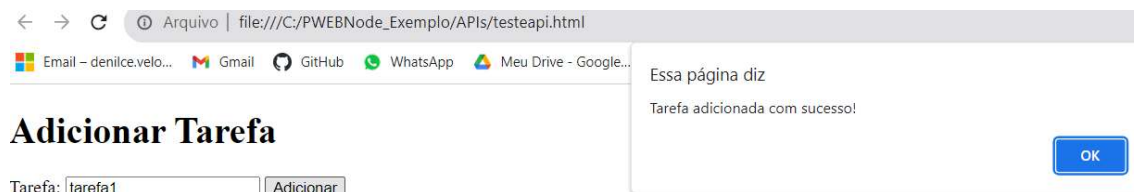
                .then((response) => {
                    if (response.ok) {
                        alert('Tarefa adicionada com
sucesso!');

                        tarefaInput.value = "";
                    } else {
                        alert('Erro ao adicionar a
tarefa.');
```

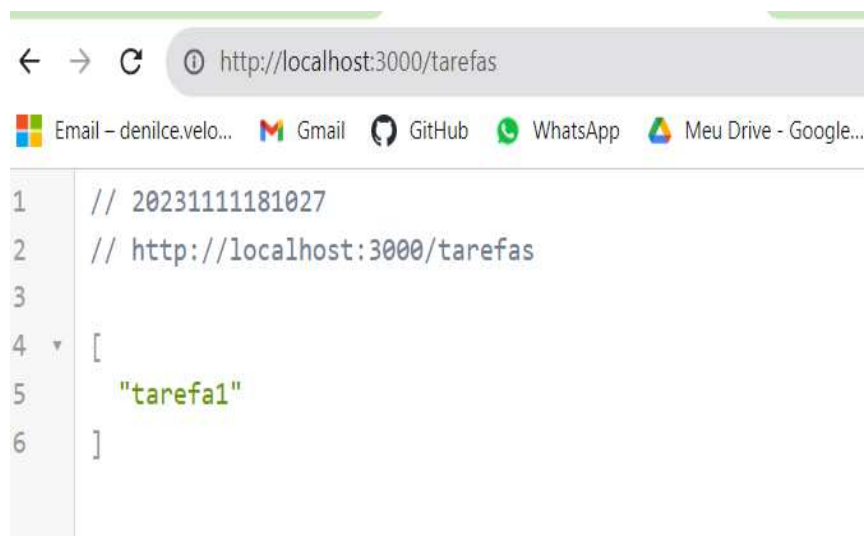
```
        }  
    })  
    .catch((error) => {  
        console.error('Erro:', error);  
        alert('Erro ao adicionar a tarefa.');
```

```
    });  
});  
});  
</script>  
</body>  
</html>
```

*Figura 96: Executando testeapi.html*



*Figura 97: Atualizando End-Point*



Caso deseje parar a aplicação basta dar um Ctrl + C dentro do terminal e a API vai parar a execução. Neste exemplo utilizamos o método HTTP POST, porém existem outros que são muito utilizados, dentre eles o GET, o PUT, o DELETE etc.

## **15. Exercício de Consumo de API Rest através do pacote**

Para fazer esse exercício serão necessários o Express e o Axios.

```
npm install axios --save
```

### **AXIOS**

Axios é um cliente HTTP baseado-em-promessas para o node.js e para o navegador. É isomórfico (= pode rodar no navegador e no node.js com a mesma base de código). No lado do servidor usa o código nativo do node.js - o modulo http, enquanto no lado do cliente (navegador) usa XMLHttpRequests.

que precisar de dados falsos . Pode ser em um README no GitHub, para uma demonstração no CodeSandbox, em exemplos de código no Stack Overflow, ou simplesmente para testar coisas localmente.

Neste exemplo, está sendo feita uma solicitação GET para a URL “https://jsonplaceholder.typicode.com/posts/1” da API JSONPlaceholder e, em seguida, manipulando a resposta e os erros potenciais. Para parar o servidor basta dar um Ctrl + C dentro do terminal.

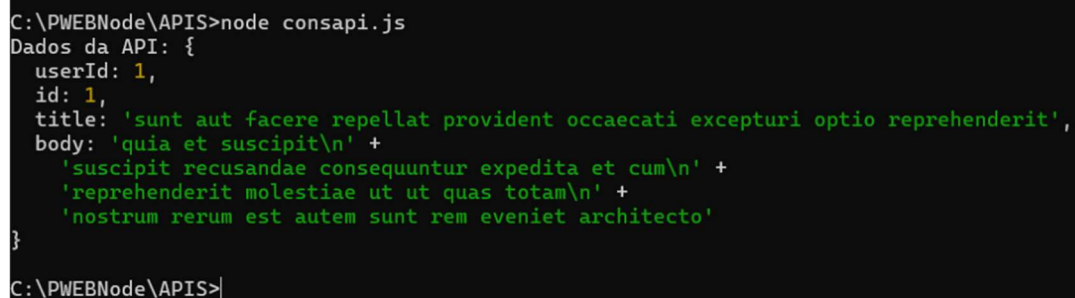
**Criar o arquivo ConsAPI.js** e digite o código abaixo:

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

// URL da API que queremos consumir
const apiUrl = 'https://jsonplaceholder.typicode.com/posts/1';

// Fazendo uma solicitação GET para a API
axios.get(apiUrl)
  .then((response) => {
    // Lida com a resposta da API
    const data = response.data;
    console.log('Dados da API:', data);
  })
  .catch((error) => {
    // Lida com erros caso ocorram
    console.error('Erro ao consumir a API:', error);
  });
```

*Figura 98: Executando consapi.js*



```
C:\PWEBNode\APIS>node consapi.js
Dados da API: {
  userId: 1,
  id: 1,
  title: 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit',
  body: 'quia et suscipit\n' +
    'suscipit recusandae consequuntur expedita et cum\n' +
    'reprehenderit molestiae ut ut quas totam\n' +
    'nostrum rerum est autem sunt rem eveniet architecto'
}
```

Nesse outro exemplo estamos consumindo a API ViaCep que volta os dados sobre um cep.



**Criar o arquivo ConsViaCep.js** e digite o código abaixo:

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

// CEP que deseja consultar
const cep = '18013-280';

// URL da API ViaCEP para consulta de CEP
const viacepUrl = `https://viacep.com.br/ws/${cep}/json`;
console.log(viacepUrl)

// Fazendo uma solicitação GET para a API ViaCEP
axios.get(viacepUrl)
  .then((response) => {
    // Verifica se a consulta foi bem sucedida (status 200)
    if (response.status === 200) {
      // Extrai os dados do endereço da resposta
      const endereco = response.data;

      // Exibe as informações do endereço no console
      console.log('Endereço completo:');
      console.log(` CEP: ${endereco.cep}`);
      console.log(` Logradouro: ${endereco.logradouro}`);
      console.log(` Complemento: ${endereco.complemento}`);
      console.log(` Bairro: ${endereco.bairro}`);
      console.log(` Cidade: ${endereco.localidade}`);
      console.log(` UF: ${endereco.uf}`);
    } else {
      // Erro na consulta (status diferente de 200)
      console.error('Erro na consulta do CEP:', response.status);
    }
  })
```

```
    })  
    .catch((error) => {  
        // Erro durante a comunicação com a API  
        console.error('Erro geral:', error);  
    });
```

*Figura 99: Executando consviacep.js*

```
C:\PWEBNode\APIS>node consviacep.js  
https://viacep.com.br/ws/18013-280/json  
Endereço completo:  
  CEP: 18013-280  
  Logradouro: Avenida Engenheiro Carlos Reinaldo Mendes  
  Complemento:  
  Bairro: Além Ponte  
  Cidade: Sorocaba  
  UF: SP  
  
C:\PWEBNode\APIS>|
```

## 16. Extras – Encoding

*Encoding* é o mecanismo que define como são apresentados os diversos símbolos e letras de diferentes alfabetos de maneira binária. Não está diretamente ligado ao idioma, só com os símbolos. O mesmo caractere á pode ser representado de diversas maneiras, com 1, 2 ou mais bytes, dependendo do encoding utilizado. E existem diversas maneiras de armazenar esses caracteres em disco/memória. Por exemplo, a palavra olá, convertida usando encoding diferentes:

ISO-8859-1

Letras	o	l	á
Hexadecimal	6f	6c	e1
Binário	01101111	01101100	11100001

UTF-8

Letras	o	l	á	
Hexadecimal	6f	6c	c3	a1
Binário	01101111	01101100	11000011	10100001

A questão é que alguns idiomas existem vários caracteres, por exemplo no português tem vários acentos e pode acabar aparecendo de forma errada na tela.

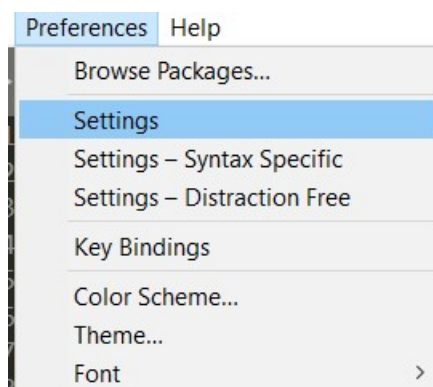
Para resolver isso, em páginas web, temos que nos atentar ao cabeçalho `<meta>`<sup>25</sup>, usado para informar aos navegadores o encoding da página e usar o encoding correto para salvar o arquivo. É mais fácil configurar no editor, em qual o encoding que o arquivo deve ser gravado.

**No caso do editor Sublime Text 3 ir em Preferences\Settings**

---

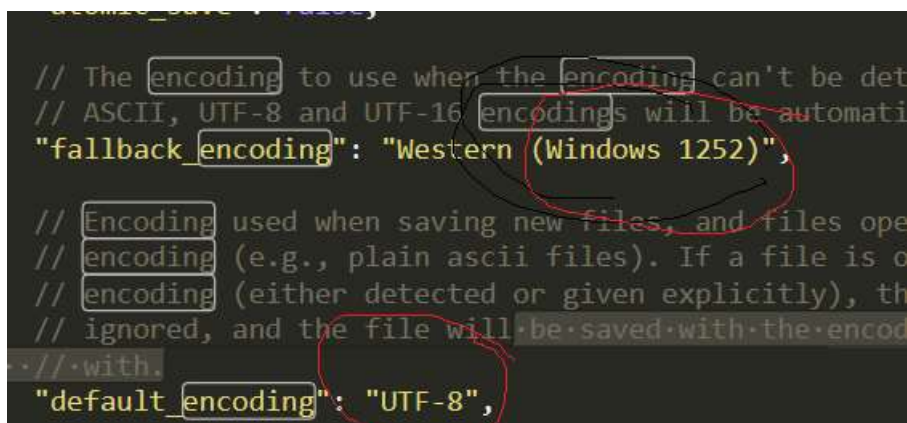
<sup>25</sup>     `<head>`  
          `<meta charset="utf-8"/>`  
      `</head>`

*Figura 39: Preferences > Settings*



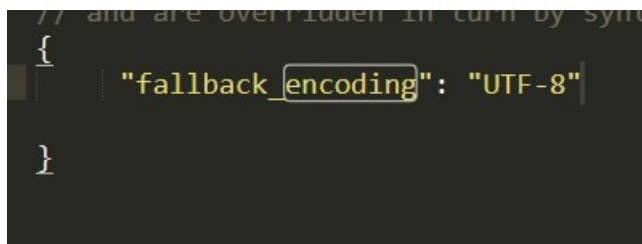
Procurar com Ctrl+F → encoding

*Figura 40: Indicação do encoding*



As configurações default (lado esquerdo) podem ser vistas, mas não podem ser modificadas. Para modificar uma configuração default do Sublime Text 3 você precisará aplicar essa modificação nas configurações do usuário (lado direito), para fazer isso basta informar o JSON com os valores atualizados separados por vírgula.

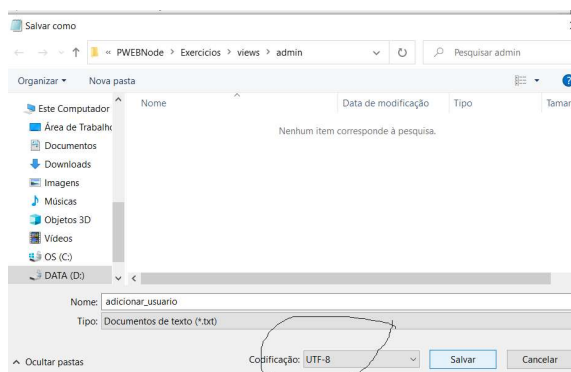
*Figura 41: Modificação das configurações do usuário*



## **No bloco de notas.**

É possível salvar o arquivo no formato utf8 no bloco de notas. Basta abrir o arquivo no bloco de notas e mudar a opção codificação e salvar novamente o arquivo.

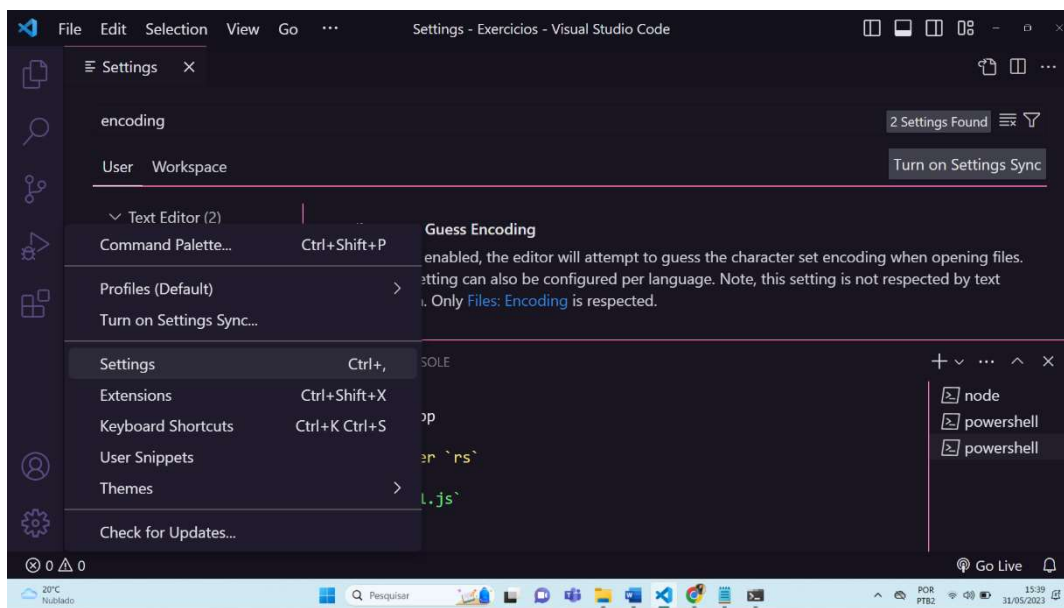
*Figura 42: Salvando o arquivo no formato UTF-8*



## **No Visual Code**

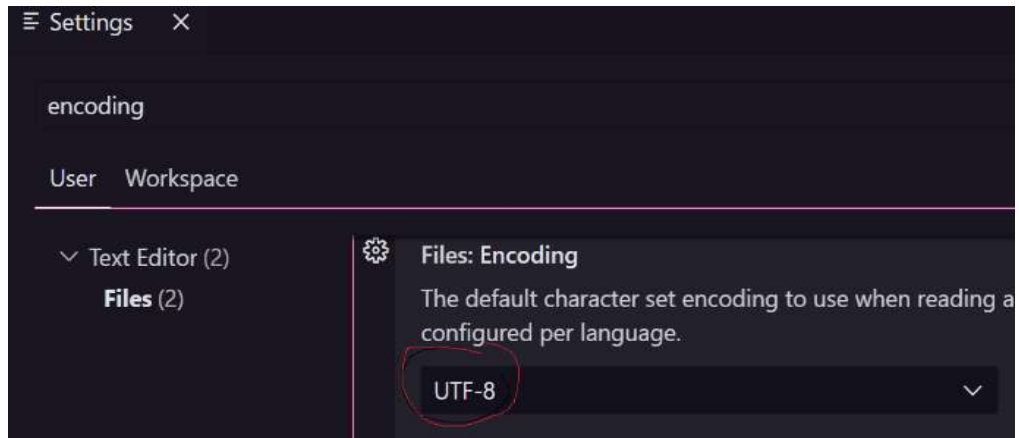
Clicar na engrenagem \ opção Settings e digitar encoding

*Figura 43: Navegando em configurações*



E do lado direito escolher UTF-8.

*Figura 44: Indo na opção UTF-8*



## **Referências**

BodyParser. O que o body-parser faz com o express? Disponível em: <https://www.tienxame.com/pt/node.js/o-que-o-body-parser-faz-com-o-express/825575205/>. Acesso: 30.Mai.2021.

CriarWeb. Disponível em: <http://www.criarweb.com/artigos/eventos-nodjs.html>. Acesso: 04.Jun.2020.

Moraes. William Bruno. Construindo Aplicações com NodeJS. Novatec Editora; 1ª edição (22 setembro 2015).

Mozilla developer. Disponível em: [https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Express\\_Nodejs/Introdução](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Express_Nodejs/Introdução). Acesso em: 15.Fev.2020.

MVC. O que é padrão MVC? Entenda arquitetura de softwares! Disponível em: <https://www.lewagon.com/pt-BR/blog/o-que-e-padrao-mvc>. Acesso em: 25.Mai.2020.

DIFERENÇA ENTRE OS PADRÕES DE DESIGN MVC E MVT. Disponível em: <https://acervolima.com/diferenca-entre-os-padroes-de-design-mvc-e-mvt/>. Acesso em: 18.Set.2023

Udemy Curso de Node. Disponível em: [www.udemy.com](http://www.udemy.com). Acesso em: 07.Dez.2019.

UFRGS Cliente/Servidor. Disponível em: [http://www.penta.ufrgs.br/redes296/cliente\\_ser/tutorial.htm](http://www.penta.ufrgs.br/redes296/cliente_ser/tutorial.htm). Acesso em: 15.Fev.2020.

Dica para saber que aplicação está utilizando a porta  
netstat -ano | findstr :3000

