



**BOSCH**

Invented for life

## Aula 4 - Node, NPM

### Docupedia Export

Author: Siqueira Joao (CtP/ETS)  
Date: 27-Jun-2022 13:31

## Table of Contents

<b>1</b>	<b>O que é o Node.js?</b>	<b>3</b>
<b>2</b>	<b>Como funciona?</b>	<b>4</b>
<b>3</b>	<b>Características</b>	<b>5</b>
<b>4</b>	<b>Node.js Module</b>	<b>6</b>
<b>5</b>	<b>Objeto exports</b>	<b>7</b>
<b>6</b>	<b>NPM - Node Packet Manager</b>	<b>10</b>
<b>7</b>	<b>Inicializando o NPM</b>	<b>11</b>
<b>8</b>	<b>Recursão Mútua e listagem de arquivos com FS</b>	<b>13</b>
<b>9</b>	<b>Lendo e escrevendo arquivos</b>	<b>14</b>
<b>10</b>	<b>Express</b>	<b>16</b>
<b>11</b>	<b>Conteúdos estáticos</b>	<b>23</b>

# 1 O que é o Node.js?



De acordo com sua definição oficial, o Node é um runtime, que nada mais é do que um conjunto de códigos, API's, ou seja, são bibliotecas responsáveis pelo tempo de execução (é o que faz o seu programa rodar) que funciona como um interpretador de JavaScript fora do ambiente do navegador web.

É importante frisar que o Node.JS é um ambiente de execução assíncrono, isto é, ele trabalha de modo a não bloquear no momento da execução da aplicação, delegando os processos demorados a um segundo plano.

## 2 Como funciona?

O Node é capaz de interpretar um código JavaScript, igual ao que o navegador faz. Sendo assim, quando o navegador recebe um comando em JavaScript, ele o interpreta e depois executa as instruções fornecidas.

Ele torna possível o envio de instruções (os nossos códigos) sem precisar de um navegador ativo, basta ter o Node.JS instalado e utilizar o terminal para executar um programa construído em JavaScript.

Além disso, você pode utilizar apenas uma linguagem de programação para tratar requisições entre cliente e servidor.

## 3 Características

- **Multiplataforma:** permite criar desde aplicativos desktop, aplicativos móveis e até sites SaaS;
- **Multi-paradigma:** é possível programar em diferentes paradigmas, como: Orientado a Objetos, funcional, imperativo e dirigido à eventos;
- **Open Source:** é uma plataforma de código aberto, isso significa que você pode ter acesso ao código fonte do Node.JS e realizar suas próprias customizações ou mesmo contribuir para a comunidade de forma direta;
- **Escalável:** Node.JS foi criado para construir aplicações web escaláveis, como podemos ver na sua [documentação oficial](#).

## 4 Node.js Module

Um *module* é uma coleção de funções e objetos do JavaScript que podem ser utilizados por aplicativos externos. Descrever um trecho de código como um módulo se refere menos ao que o código é do que aquilo que ele faz — qualquer arquivo Node.js pode ser considerado um módulo caso suas funções e dados sejam feitos para programas externos. O Node.js já possui um objeto padrão *module* pronto, que podemos visualizar utilizando o comando:

```
[Running] node "c:\Users\siqet\Documents\Node\tempCodeRunnerFile.js"
Module {
  id: '.',
  path: 'c:\\Users\\siqet\\Documents\\Node',
  exports: {},
  filename: 'c:\\Users\\siqet\\Documents\\Node\\tempCodeRunnerFile.js',
  loaded: false,
  children: [],
  paths: [
    'c:\\Users\\siqet\\Documents\\Node\\node_modules',
    'c:\\Users\\siqet\\Documents\\node_modules',
    'c:\\Users\\siqet\\node_modules',
    'c:\\node_modules'
  ]
}
```

### Visualização do module

1	<code>console.log(module)</code>
---	----------------------------------

## 5 Objeto *exports*

Utilizando o objeto *exports* dentro do módulo é possível exportar e importar uma informação entre módulos como nos exemplos abaixo, utilizando variáveis e funções:

### Exportando variável

```

1  const nome = "João";
2  const sobrenome = "Siqueira";
3
4  const nomeInteiro = () => {
5      console.log(nome, sobrenome);
6  };
7
8  module.exports.nome = nome;
9  module.exports.sobrenome = sobrenome;
10 module.exports.nomeInteiro = nomeInteiro;
11 console.log(module)

```

```

[Running] node "c:\Users\siqet\Documents\Node\tempCodeRunnerFile.js"
Module {
  id: '.',
  path: 'c:\Users\siqet\Documents\Node',
  exports: {
    nome: 'João',
    sobrenome: 'Siqueira',
    nomeInteiro: [Function: nomeInteiro]
  },
  filename: 'c:\Users\siqet\Documents\Node\tempCodeRunnerFile.js',
  loaded: false,
  children: [],
  paths: [
    'c:\Users\siqet\Documents\Node\node_modules',
    'c:\Users\siqet\Documents\node_modules',
    'c:\Users\siqet\node_modules',
    'c:\Users\node_modules',
    'c:\node_modules'
  ]
}

```

```

[Running] node "c:\Users\siqet\Documents\Node\app.js"
João
Siqueira
João Siqueira

```

A informação exportada pode ser recebida em outro arquivo pelo código abaixo:

### Importando

```

1  const mod1 = require('./mod1');
2  console.log(mod1.nome);
3  console.log(mod1.sobrenome);
4  mod1.nomeInteiro();
5
6  //const {nome, sobrenome, nomeInteiro}=
7  //require('./mod1')
8  //console.log(nome)
9  //console.log(sobrenome)
10 //nomeInteiro()

```

Como estamos utilizando o módulo padrão *module* podemos referenciá-lo utilizando apenas *exports* e também utilizar a palavra reservada *this* para substituir ambos os termos.

Entretanto, isso não nos permite criar diretamente um novo objeto com todos os dados a serem exportados, como pode ser visto nos exemplos abaixo:

```

1  const nome = "João";
2  const sobrenome = "Siqueira";
3
4  //module.exports = {
5  //  nome, sobrenome
6  //};
7
8  //This.qualquerCoisa = "Qualquer coisa"
9
10 exports.outraCoisa = "Outra coisa"

```

Também é possível exportar e importar classes:

```

1  class Pessoa{
2      constructor(nome){
3          this.nome = nome;
4      }
5  }
6
7  exports.Pessoa = Pessoa

```

```

1  const { Pessoa } =
    require('./mod1');
2  console.log(Pessoa);
3
4  const p1 = new Pessoa('João')
    ;
5  console.log(p1);

```

Assim como funções também:

```

1  module.exports = function(x,
2      y) {
3      return x * y;
4  };

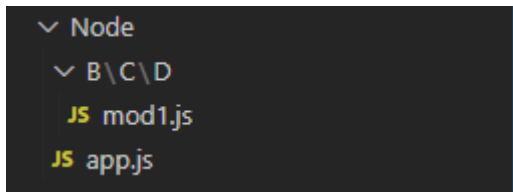
```

```

1  const multiplicacao =
    require('./mod1');
2
3  console.log(multiplicacao(2,
4      2));

```

É importante sempre nos atentarmos ao diretório do arquivo que estamos requerindo:



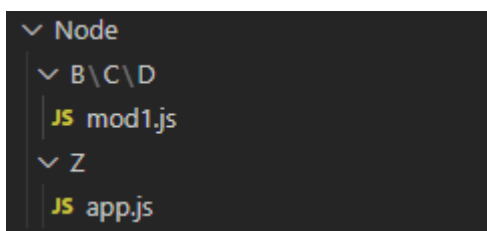
Caso ele esteja dentro de várias pastas deveremos endereçá-lo corretamente.

```

1  const multiplicacao = require('./B/C/D/mod1');
2
3  console.log(multiplicacao(2,2));

```

Para voltar em uma pasta anterior podemos utilizar "../", como no caso abaixo:



```

1  const multiplicacao = require('../B/C/D/mod1');
2
3  console.log(multiplicacao(2,2));
4
5  // ./ Pasta atual
6  // ../ Pasta anterior

```

Para não nos confundirmos podemos usar os comandos que nos retornam o caminho do diretório ou arquivo em questão. Também podemos usar o diretório atual para referenciar algum outro.

```

1  console.log(__filename);
2  console.log(__dirname);

```



```
3
4  const path = require('path')
5  console.log(path.resolve(__dirname));
6
7  console.log(path.resolve(__dirname, "..", ".."));
8  console.log(path.resolve(__dirname, "..", "..", "Downloads"));
```

## 6 NPM - Node Packet Manager

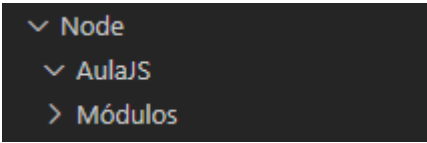
O NPM, que significa *Node Package Manager* é o gerenciador de pacotes padrão para o Node.js e consiste em um cliente de linha de comando, também chamado de npm, e um banco de dados online de pacotes públicos e privados pagos, chamado de registro npm. O registro é acessado por meio do cliente e os pacotes disponíveis podem ser navegados e pesquisados no site do npm.

O NPM permite, para além da criação de módulos, executar instruções ou conjuntos de instruções através de um comando criado pelo utilizador conforme a sua necessidade.

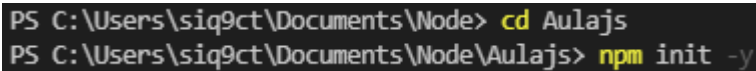


## 7 Inicializando o NPM

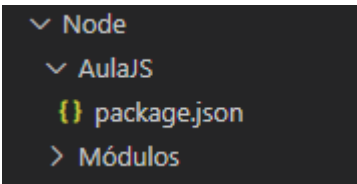
Para utilizar o NPM pela primeira vez, vamos criar uma nova pasta, acessá-la através do terminal de comando e inicializar o gerador com o comando *npm init -y*



```
▼ Node
  ▼ AulaJS
    > Módulos
```



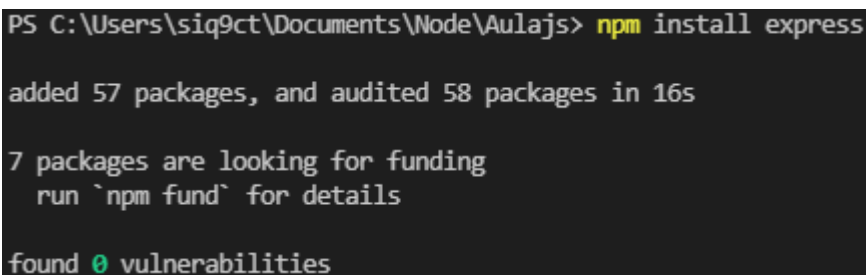
```
PS C:\Users\siq9ct\Documents\Node> cd AulaJS
PS C:\Users\siq9ct\Documents\Node\AulaJS> npm init -y
```



```
▼ Node
  ▼ AulaJS
    {} package.json
      > Módulos
```

### Instalando um pacote

Para instalar um pacote através do npm utilizamos o comando *npm install <nomedopacote>*, como por exemplo o framework web express.



```
PS C:\Users\siq9ct\Documents\Node\AulaJS> npm install express

added 57 packages, and audited 58 packages in 16s

7 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Acessando o arquivo package.json podemos verificar a versão do pacote instalado.



```
{
  "name": "aulajs",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.1"
  }
}
```

Podemos separar as dependências específicas de desenvolvimento como devDependencies com o comando *npm install <nomedopacote> --save-prod*.

Dependências de desenvolvimento são dependências que não são necessárias para nossa aplicação na versão final no servidor, apenas durante o processo de desenvolvimento. Alguns exemplos são os módulos que você instala para minificar, concatenar arquivos e até mesmo rodar alguns pré-processadores de css.

```
PS C:\Users\siq9ct\Documents\Node\Aulajs> npm install express --save-dev
npm WARN idealTree Removing dependencies.express in favor of devDependencies.express

up to date, audited 58 packages in 4s

7 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

```
"devDependencies": {
  "express": "^4.18.1"
```

Caso essa dependência seja uma dependência de produção ela pode ser movida para o dependencies com o comando `npm install <nomedopacote> --save-prod`

Como os pacotes possuem múltiplas versões também podemos especificar a versão a ser instalada, como no exemplo:

```
PS C:\Users\siq9ct\Documents\Node\Aulajs> npm install express@2.1.0
```

Os números de versões de pacote geralmente possuem três dígitos, como o 2.1.0 no exemplo. O 2 indica a versão *major*, o 1 a versão *minor* e o 0 o que chamamos de *path*. Se algum bug ou erro dessa versão fosse corrigido, haveria um incremento no path. Caso novas aplicações fossem lançadas, mas elas fossem compatíveis na anterior, a versão passaria a ser 2.2.0. E por último, em uma mudança mais radical, em que as novas aplicações não fossem compatíveis com a versão anterior, a mudança no major tornaria a versão como 3.0.0

## 8 Recursão Mútua e listagem de arquivos com FS

O FS (File System) é uma API que tem no Node usada para manipular arquivos e pastas, tendo uma [documentação](#) bem extensa, já a recursão mútua é uma função que depende de uma segunda função e essa segunda função depende da primeira

```
1  const fs = require('fs').promises;
2  const path = require('path');
3
4  async function readDir(roodDir) {
5      rootDir = roodDir || path.resolve(__dirname); // Caso seja mandado um
6      // diretório ele será usado, caso contrario usara o diretório de onde o
7      // arquivo JS está
8      const files = await fs.readdir(rootDir); // Retorna um array com todos
9      // os arquivos desse diretório
10     walk(files, roodDir);
11 }
12
13 async function walk(files, roodDir) {
14     for (let file of files) {
15         const fileFullPath = path.resolve(roodDir, file); // Retorna o
16         // caminho do arquivo mais o nome do arquivo
17         const stats = await fs.stat(fileFullPath);
18
19         if(stats.isDirectory()) { // Retorna se o arquivo é um diretório
20             // (pasta) ou não
21             readDir(fileFullPath); // Volta a chamar a função que chamou
22             // essa função, aplicando a recursão mútua no sistema
23             continue;
24         }
25
26         if (!/\.css$/g.test(fileFullPath)) continue; // Filtra apenas os
27         // arquivos css
28
29         console.log(fileFullPath);
30     }
31 }
32
33 readDir('C:/Users/liq1ct/Desktop');
```

[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"

C:\Users\liq1ct\Desktop\Aula JS\style.css

[Done] exited with code=0 in 0.549 seconds

## 9 Lendo e escrevendo arquivos

### Criando .txt

```
1  const fs = require('fs');
2  const path = require('path');
3  const caminhoArquivo = path.resolve(__dirname, '..', 'teste.txt'); // cria
   o caminho, com o diretório do arquivo, volta uma pasta e cria um txt
4
5  fs.writeFile(caminhoArquivo, 'Frase 1', { flag: 'w', encoding: 'utf-8' }); /
   / passamos o diretório, depois a mensagem e um objeto, dentro do objeto
   passamos o flag, e a codificação que será usada
```

para o flag podemos passar outras letras também, que mudaram a forma como o programa modifica o arquivo, o "w" limpa o arquivo antes de escrever, o "a" adiciona a frase no final do arquivo no final entre outros

### Criando JSON

```
1  const fs = require('fs').promises;
2  const path = require('path');
3  const caminhoArquivo = path.resolve(__dirname, '..', 'teste.json');
4
5  const pessoas = [
6    {nome: "João"},
7    {nome: "Thiago"},
8    {nome: "Carlos"},
9    {nome: "Maria"},
10   {nome: "Luiza"},
11 ];
12
13 const json = JSON.stringify(pessoas, ' ', 2);
14
15 fs.writeFile(caminhoArquivo, json, { flag: 'w' });
```

Agora vamos fazer um mini app que consiga ler e escrever json de objetos do JavaScript

### escrever.js

```
1  const fs = require('fs').promises;
2
3  module.exports = (caminho, dados) => {
4    fs.writeFile(caminho, dados, { flag: 'w' });
5  }
```

**ler.js**

```
1  const fs = require('fs').promises;
2
3
4  module.exports = (caminho) => fs.readFile(caminho, 'utf-8');
```

**app.js**

```
1  const path = require('path');
2  const caminhoArquivo = path.resolve(__dirname, 'teste.json');
3  const escreve = require('./modules/escrever');
4  const ler = require('./modules/ler');
5
6  // const pessoas = [
7  //     {nome: "João"},
8  //     {nome: "Thiago"},
9  //     {nome: "Carlos"},
10 //     {nome: "Maria"},
11 //     {nome: "Luiza"},
12 // ];
13
14 // const json = JSON.stringify(pessoas, '', 2);
15
16 // escreve(caminhoArquivo, json);
17
18 async function lerArquivo(caminho) {
19     const dados = await ler(caminho);
20     transformaEmObjeto(dados);
21 }
22
23 function transformaEmObjeto(dados) {
24     dados = JSON.parse(dados);
25     dados.forEach(val => console.log(val));
26 }
27
28 lerArquivo(caminhoArquivo);
```

como já temos o json criado deixamos comentado a parte de criar ele para só ter que ler

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\EscriverLerArquivos\app.js"
```

```
{ nome: 'João' }
{ nome: 'Thiago' }
{ nome: 'Carlos' }
{ nome: 'Maria' }
{ nome: 'Luiza' }
```

```
[Done] exited with code=0 in 0.574 seconds
```

# 10 Express

Primeiro iniciamos o npm e instalamos o express como visto anteriormente

```
PS C:\Users\liq1ct\Desktop\Aula JS\Express> npm init -y
Wrote to C:\Users\liq1ct\Desktop\Aula JS\Express\package.json:
```

```
{
  "name": "express",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

```
PS C:\Users\liq1ct\Desktop\Aula JS\Express> npm install express
```

```
added 57 packages, and audited 58 packages in 14s
```

```
7 packages are looking for funding
run `npm fund` for details
```

```
found 0 vulnerabilities
```

```
PS C:\Users\liq1ct\Desktop\Aula JS\Express> []
```

Agora já podemos fazer uma página simples.

```
1  const express = require('express');
2  const app = express();
3
4  app.get('/', (requisicao, resposta) => {
5    resposta.send(`
6      <form action="/" method="POST">
7        Nome do cliente: <input type="text" name="nome">
8        <button>Enviar</button>
9      </form>
10    `);
11  });
12
13  app.post('/', (requisicao, resposta) => {
14    resposta.send('Formulário recebido');
15  });
16
17  app.listen(3000, () => console.log("Acessar http://localhost:3000")); //
    Muito importante usar uma porta que não tenha nenhum processo rodando
```



Para iniciar o servidor vamos no terminal e digitamos "node (nome do arquivo)" e ele começara a rodar, para parar também no terminal digamos CTRL + C

```
PS C:\Users\liq1ct\Desktop\Aula JS\Express> node server.js
Acessar http://localhost:3000
█
```

Mas isso faz com que todas as vezes que mudemos algo no arquivo tenhamos que parar o servidor e iniciá-lo novamente

Então podemos instalar o nodemon

```
PS C:\Users\liq1ct\Desktop\Aula JS\Express> npm install nodemon --save-dev

added 116 packages, and audited 174 packages in 41s

23 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\liq1ct\Desktop\Aula JS\Express> █
```

Após a instalação vamos no arquivo package.json, e modificamos o start

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node server.js"
},
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "nodemon server.js"
},
```

-->

Depois disso podemos voltar para o terminal e digitar npm start

```
PS C:\Users\liq1ct\Desktop\Aula JS\Express> npm start

> express@1.0.0 start
> nodemon server.js

[nodemon] 2.0.16
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Acessar http://localhost:3000
█
```

Existem 3 formas de recebermos informações o nosso site

.params é para enviarmos junto de mais uma barra

```
1 const express = require('express');
2 const app = express();
3
4 app.get('/:numero?', (requisicao, resposta) => { // "?" serve para o dado
5   resposta.send(`Número recebido: ${requisicao.params.numero}`);
6 });
7 app.listen(3000, () => console.log("Acessar http://localhost:3000"));
```

← → ↻ 🏠 ⓘ localhost:3000

Número recebido: undefined

← → ↻ 🏠 ⓘ localhost:3000/12

Número recebido: 12

.query é para enviarmos dados mais simples pela url

```
1 const express = require('express');
2 const app = express();
3
4 app.get('/', (requisicao, resposta) => {
5   resposta.send(`Número recebido: ${requisicao.query.numero}`);
6 });
7 app.listen(3000, () => console.log("Acessar http://localhost:3000"));
```

← → ↻ 🏠 ⓘ localhost:3000/?numero=1234

Número recebido: 1234

.body é para enviar dados que vieram de formulários

```
1 const express = require('express');
2 const app = express();
3
4 app.use(
5   express.urlencoded(
6     { extended: true }
7   )
8 ); // Importante colocar essa parte, caso contrario não ira funcionar o
   envio do formulário
9
10 app.get('/', (requisicao, resposta) => {
11   resposta.send(`
12     <form action="/" method="POST">
13       Nome do cliente: <input type="text" name="nome">
14       <button>Enviar</button>
15     </form>
```

```

16     `);
17   });
18
19   app.post('/', (requisicao, resposta) => {
20     resposta.send(`Nome: ${requisicao.body.nome}`);
21   });
22
23   app.listen(3000, () => console.log("Acessar http://localhost:3000/"));

```

← → ↻ 🏠 🌐 localhost:3000/

Nome do cliente:

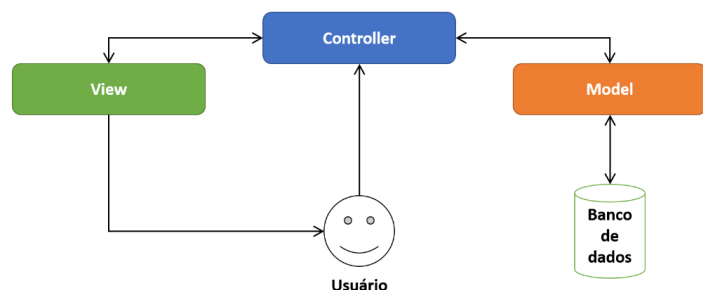
← → ↻ 🏠 ⓘ localhost:3000

Nome: Thiago

## Criando Rotas

Anteriormente a criação de rotas e as chamadas das funções estavam sendo escritas em sequência e num único arquivo, mas isso se torna uma prática inviável no mercado. Existem modelos que estruturam a arquitetura de software de forma que as construções de interfaces e execuções back-end sejam organizadas e divididos em camadas.

O modelo que utilizaremos é o Full MVC, um padrão de arquitetura de software. O MVC sugere uma maneira para você pensar na divisão de responsabilidades, principalmente dentro de um software web. O princípio básico do MVC é a divisão da aplicação em três camadas: a camada de interação do usuário (**view**), a camada de manipulação dos dados (**model**) e a camada de controle (**controller**).



Para implementar esse padrão utilizaremos as rotas do express, que comunicam as camadas de forma a responder as requisições dos clientes. Chamando um controlador de acordo com essa rota.

Para definirmos essas rotas, crie o arquivo `routes.js`, onde através do `express.Router` teremos as rotas tratadas.

Crie também uma pasta chamada `controllers`, onde estarão nossos arquivos de controle. Crie um arquivo chamado `homeControllers.js` que terá a mesma execução do

Já na nossa página principal apenas importamos as rotas e definimos que o express utilizará essas rotas com `app.use(routes)`.

exercício anterior porém dentro do padrão

```

routes.js
1  const express
   =
   require(
     'express');
2  const route =
   express.Route
   r();
3  const
   homeControlle
   r =
   require('./
   controllers/
   homeControlle
   r');
4
5  route.get('/',
   homeControlle
   r.paginaInici
   al);
6
7  module.export
   s = route;

```

```

homeCrontroller.js
1  exports.pagi
   naInicial =
   (req, res)
   => {
2
3     res.send(`
       <form
       action="/"
4       Nome do
       cliente:
       <input
       type="text"
5       name=
       "qualquercoi
       sa"><br>
       Outro
       campo:
       <input
       type="text"
6
7       name=
       "aquioutroca
       mpo">
       <button>Olá
       mundo</
       button>
7       </form>
8       `);
9   };

```

```

server.js
1  const express
   =
   require(
     'express');
2  const app =
   express();
3  const routes
   = require('./
   routes');
4
5  app.use(expre
   ss.urlencoded
   ({ extended:
     true }));
6  app.use(route
   s);
7
8  app.listen(30
   00, () =>
   console.log(
     "Acessar
     http://
     localhost:300
     0"));

```

Dessa forma o homeController como controlador da página home, para cada página dentro do nosso projeto criamos um controlador que determinará suas exibições e respostas.

## Views

Agora que já vimos como a camada de controle é implementada vamos criar a camada de visualizações. Para trabalhar com views vamos instalar o EJS. O EJS é uma engine de visualização, com ele conseguimos de uma maneira fácil e simples transportar dados do back-end para o front-end, basicamente conseguimos utilizar códigos em javascript no html de nossas páginas. Assim na pasta do seu projeto do o comando 'npm i ejs'.

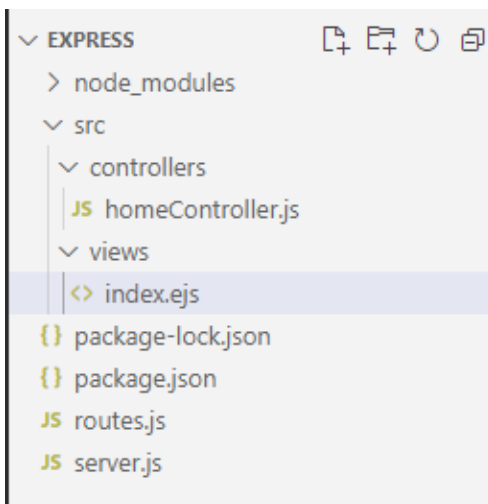
```
PS C:\Users\liq1ct\Desktop\Aula JS\Express> npm i ejs

added 6 packages, and audited 180 packages in 14s

23 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\liq1ct\Desktop\Aula JS\Express> 
```

Em seguida vamos organizar os arquivos de forma a combinar com o padrão, assim, crie uma pasta views onde colocaremos as páginas ejs(html renderizado), junto da pasta controllers mova para pasta src. Por fim crie o arquivo index.ejs na pasta views.



Feito isso vamos definir no express a utilização das views e do EJS como ferramenta de renderização.

No server.js modifique:

#### server.js

```
1  const express = require('express');
2  const app = express();
3  const routes = require('./routes');
4
5  app.use(express.urlencoded({ extended: true }));
6  app.set('views', './src/views');
7  app.set('view engine', 'ejs');
8
9  app.use(routes);
10
11 app.listen(3000, () => console.log("Acessar http://localhost:3000"));
```

Agora não vamos mais definir o "html" ou a resposta no controller e sim nas views, dessa forma no arquivo homeController.js vamos trocar a função paginaInicial e está passará a renderizar o arquivo index.ejs

#### homeCrontroller.js

```
1  exports.paginaInicial = (req, res) => {
2    res.render('index');
```

3

};

# 11 Conteúdos estáticos

São arquivos como folhas de estilo, imagens etc que também tem seu lugar nas aplicações, vamos criar esse lugar. Crie uma pasta chamada public.

Em seguida vamos definir que nela estão nossos conteúdos estáticos. No arquivo server.js inclua:

## server.js

```
1  const express = require('express');
2  const app = express();
3  const routes = require('./routes');
4
5  app.use(express.urlencoded({ extended: true }));
6  app.use(express.static('./public'))
7
8  app.set('views', './src/views');
9  app.set('view engine', 'ejs');
10
11 app.use(routes);
12
13 app.listen(3000, () => console.log("Acessar http://localhost:3000"));
```

Existem as variáveis de ambiente são aquelas que definimos fora de um programa, geralmente por um provedor da nuvem ou um sistema operacional. No Node, as variáveis de ambiente são uma ótima maneira de configurar com segurança e conveniência questões que não se alteram com frequência, como URLs, chaves de autenticação e senhas. DotEnv é um pacote leve do npm que carrega automaticamente as variáveis de ambiente de um arquivo `.env` no objeto `process.env`.

```
PS U:\JS\express> npm install dotenv
```

Para conectar com o banco de dados também vamos precisar instalar o mssql e o msnodesqlv8

```
PS U:\JS\express> npm install mssql      PS U:\JS\express> npm install msnodesqlv8
```

Para criar a conexão inclua:

## server.js

```
1  const pool = new sql.ConnectionPool({
2    database: "master",
3    server: "JVLPC0480",
4    driver: 'msnodesqlv8',
5    options: {
6      trustedConnection: true
7    }
8  })
9
10 pool.connect().then(() => {
11
12   pool.request().query(`SELECT * FROM teste`, (err, result) => {
```

```
13     if(err) console.log(err);
14
15     else {
16         console.dir(result.recordset);
17         app.emit('pronto');
18     }
19 });
20 });
```

Porém, dessa forma o aplicativo pode começar a escutar antes de conectar com a base de dados  
Então vamos emitir um sinal para que o aplicativo só começa a escutar após fazer a conexão

#### server.js

```
1  const express = require('express');
2  const app = express();
3
4  const sql = require('mssql/msnodesqlv8')
5
6  const pool = new sql.ConnectionPool({
7      database: "master",
8      server: "JVLPC0480",
9      driver: 'msnodesqlv8',
10     options: {
11         trustedConnection: true
12     }
13 })
14
15 pool.connect().then(() => {
16
17     pool.request().query(`INSERT INTO teste (id, nome) VALUES (17, 'teste2')`
18     , (err, result) => {
19         if(err) console.log(err);
20
21         else {
22             console.dir(result.recordset);
23             app.emit('pronto');
24         }
25     });
26
27     const routes = require('./routes');
28
29     app.use(express.urlencoded({ extended: true }));
30     app.use(express.static('./public'))
31
32     app.set('views', './views');
33     app.set('view engine', 'ejs');
34     app.use(routes);
35
36     app.on('pronto', () => {
37         app.listen(3000, () => {
38             console.log("Acessar http://localhost:3000");
39             console.log("Servidor executando na porta 3000");
40         });
41     });
```