DigitalHouse> Express

DigitalHouse>

Introdução a MVC

“MVC é um **design pattern** que nos permite **organizar o código** e os arquivos de acordo com sua **responsabilidade**, em outras palavras, de acordo com a tarefa que devem realizar.”

O que é design pattern?

Um *design pattern* **é uma solução geral para um problema comum** encontrado pela comunidade de desenvolvedores. Ele pode ser adaptado para cada situação, sem perder sua estrutura. Na prática, **é um conjunto de regras que definem a arquitetura de um sistema**.

Ou seja, é uma forma de trabalhar, um esquema sugerido que nos permite simplificar o código, além de gerar uma **aplicação mais organizada e eficiente** para solucionar diferentes situações ao longo de todo o desenvolvimento.

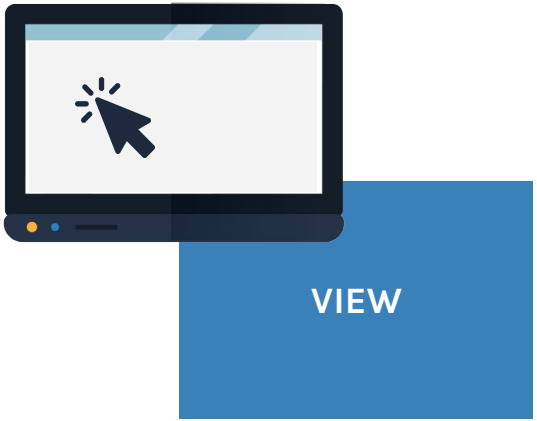
Um dos padrões mais populares é o **MVC**. Seu objetivo é criar aplicações modulares, dividindo a estrutura do projeto em **três componentes** principais:

- **M**odels
- **V**iews
- **C**ontrollers

As views

Compõem a **interface gráfica** da aplicação e contêm todos os elementos que são visíveis ao usuário. Através delas, o usuário interage enviando e solicitando informações para o servidor.

Sua responsabilidade é definir a **aparência** dos dados e mostra-los na tela.

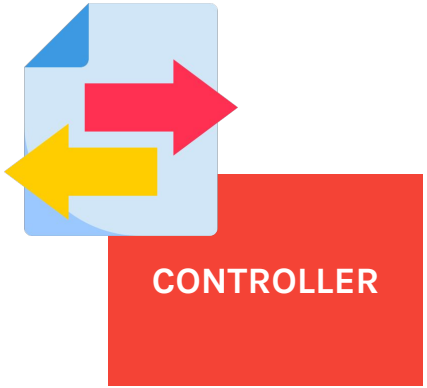


As views não se comunicam de forma direta com os models.

Os controllers

Compõem a camada intermediária entre as views e os models.

Sua responsabilidade é **processar** os dados que recebe dos *models* e escolher a *view* correspondente de acordo com esses dados.

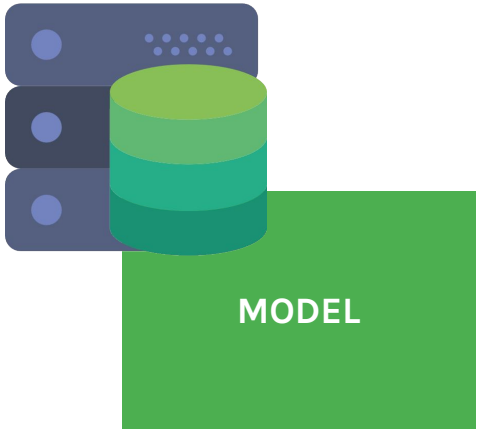


*Tem **relação direta** com as views e com os models e é um componente fundamental dentro do fluxo de padrão.*

Os models

Compõem e contêm a **lógica** da aplicação.

Suas responsabilidades são: se **conectar** com o **banco de dados**, realizar **consultas** e administrar o que se conhece como a regra de negócio.

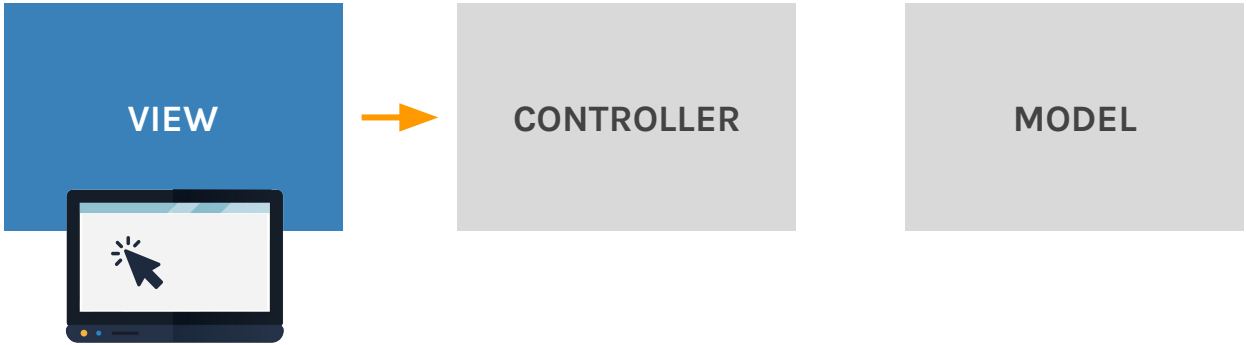


Os models não se comunicam de forma direta com as views.

Fluxo MVC

Um usuário percorre, através da interface gráfica, uma lista de produtos e quer solicitar mais informações sobre um produto.

A **view** então se **conecta** com o **controller** para solicitar esses dados.



Fluxo MVC

O **controller** recebe a requisição e **solicita** para o *model* o detalhe do produto solicitado.

Fluxo MVC

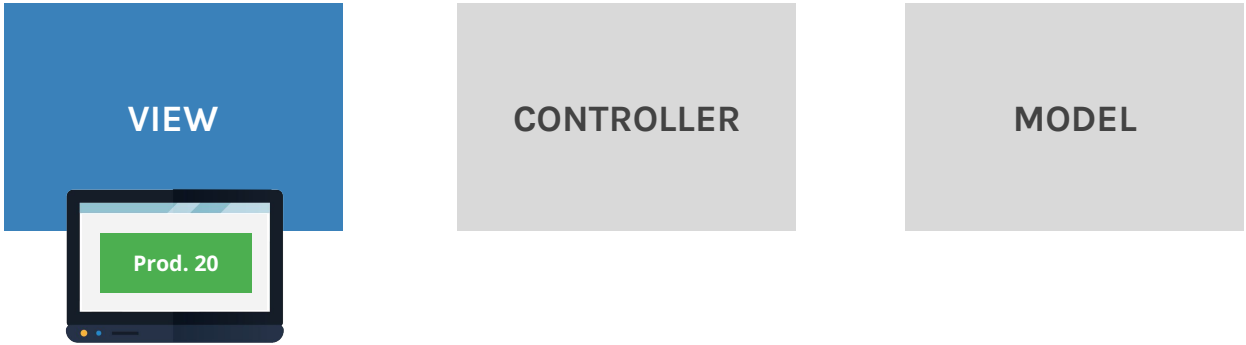
O **model** busca a informação solicitada e **envia** ao **controller**.

Fluxo MVC

O **controller** recebe a informação e **envia** os dados para a **view**.

Fluxo MVC

A **view** mostra ao **usuário** os dados que recebeu do *controller*.



“Os design patterns propõem uma forma de trabalho e não há nada por trás para regular se estas formas estão sendo cumpridas. São utilizadas como uma **boa prática** que facilita o **trabalho em equipe**, permitindo a unificação de critérios e abordagens.”

Padrão de desenho MVC

DigitalHouse>

Definindo uma rota

Usando o *express*, contamos com uma **estrutura básica** para definir cada uma das rotas da nossa aplicação e as respostas para cada uma dessas rotas

```
{}
app.get('/', function (req , res) {
    res.send('Olá, mundo!');
})
```


DigitalHouse> Coding School

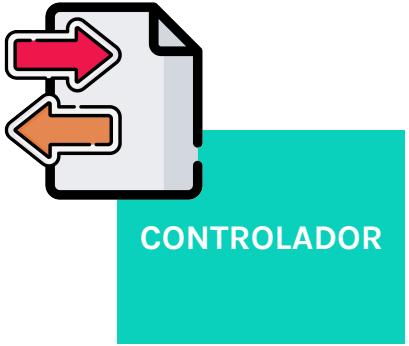
Rotas parametrizadas

Usando a mesma estrutura básica para definir uma rota, definimos o parâmetro fazendo uso de dois pontos `:`, seguido do nome que representa o dado que está chegando na URL.

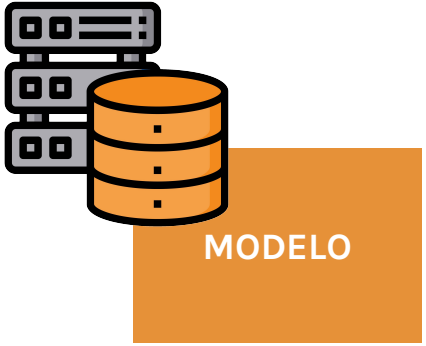
```
app.get('/produtos/:id', function (req,res) {
    // código
})
```

Model View Controller

Compreende a **interface gráfica** da aplicação.



Compreende a **camada intermediária** entre as visões (telas) e os modelos.



É a modelagem do **banco de dados**.

Parâmetros opcionais

Se queremos que um parâmetro seja opcional, adicionamos o símbolo de interrogação `?` depois do nome do dele.

Caso haja mais de um parâmetro, os opcionais devem ir sempre no final.

```
app.get('/produtos/:id?', function (req,res) {
    // código
})
```


DigitalHouse> Coding School

Sistema de roteamento

Definimos as rotas que consideramos necessárias para lidar com diferentes tipos de ***request***.

```
{}
// Rota raiz dos produtos / Início
router.get('/', (req, res) => {
    // código
});
// Rota que mostra os detalhes de um produto
router.get('/detalhe/:id', (req, res) => {
    // código
});
```


DigitalHouse> Coding School

Criando um Controlador

Removeremos a *callback* que definimos nas rotas e a escreveremos no método **index** do controlador de produtos.

```
const controlador = {
    index: (req, res) => {
        res.send('Olá, mundo!');
    },
};
```

Então vamos exportar a variável na última linha do arquivo.

```
module.exports = controlador;
```


DigitalHouse> Coding School

Implementando um Controlador

Para começar a usar os métodos que definimos, devemos solicitar (importar) o módulo correspondente de dentro do arquivo de roteamento do recurso, neste caso *products.js* (que está dentro da pasta de rotas).

```
{}  const prodController = require('../controllers/produtosController');
```

Vamos terminar de configurar aquela rota da qual removemos a *callback*, substituindo a *callback* anterior pelo método do controlador correspondente. Sendo uma *callback,* não escrevemos os parênteses.

```
{}  router.get('/', produtosController.index);
```


DigitalHouse> Coding School

Os controladores

DigitalHouse>

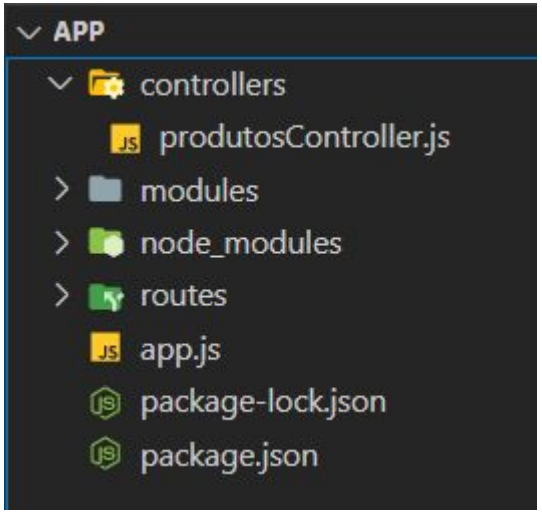
“Sua responsabilidade é a de **atender** os distintos **pedidos** do cliente e fazer **comunicação** entre as **views** e as **models.**”

Arquitetura dos arquivos

Na pasta raiz do projeto, criaremos a pasta **controllers**. Dentro dela, armazenaremos um controlador para cada recurso ou entidade que tenha na aplicação.

As entidades podem ser coisas como usuários, produtos, carrinhos de compras, etc.

Cada controlador será um **módulo** que exportaremos, para logo importá-lo onde precisarmos.



Criando um **controlador**

Criaremos um arquivo para manipular nosso recurso. Para nomear os arquivos, é habitual começar com o **nome** do **recurso**, seguido da palavra **Controller**, usando o estilo *camelCase*.

Criando um controlador

A primeira coisa a se fazer é definir uma variável no arquivo do controlador do recurso e atribuir um objeto literal.

```
{}
const controlador = {};
```

Dentro do objeto, vamos definindo os métodos que serão responsáveis por manipular cada pedido em particular.

```
{}
const controlador = {
    index: //mostrar lista de produtos,
    show: //mostrar detalhe de um produto,
    create: //enviar dados para adicionar um produto,
};
```

O jeito anterior

Até agora, a **lógica** de como responder cada request que chegava era escrita dentro do arquivo de rotas de cada recurso, onde definíamos uma **URL** e um **callback**, que se encarregava de manipular este pedido.

```
{}
// código do arquivo produtos.js, na pasta routes
router.get('/', (req, res) => {
    res.send('Index de produtos');
});
```

O jeito novo

Ao trabalhar com o padrão **MVC**, podemos nos apoiar nas suas regras e dividir as responsabilidades de nossos arquivos. Seguindo nessa linha, serão os **métodos** de cada controlador os que **receberão** esses dados e **enviarão** a informação correspondente.

Desse modo, vamos tirar o callback que havíamos definido nas rotas e o escreveremos no método index do controlador de produtos.

```
{}
const controlador = {
    index: (req, res) => {
        res.send('Index de produtos');
    },
};
```

“Como **boa** prática, é comum usar **nomes descritivos** e **lógicos** para nomear os **métodos** de um **controlador.**”

Implementar um controlador

Primeiro, temos que tornar visível todo o código que definimos no controlador. Para isto, exportaremos a variável na última linha do arquivo.

```
{}  module.exports = controlador;
```

Para começar a usar os métodos que definimos, devemos importar o módulo dentro do arquivo de rotas do recurso, nesse caso produtos.js, dentro da pasta routes.

```
{}  const produtosController = require('../controllers/produtosController');
```

“Para voltar uma **pasta** que está em um nível **anterior**, quando indicamos onde está hospedado um **arquivo**, devemos escrever dois pontos, seguidos de uma barra ../, no começo do **caminho.**”

Implementar um controlador

Com nosso módulo já visível no arquivo, é hora de terminar de configurar aquela rota, onde excluímos o callback.

A mesma vai continuar recebendo esse callback, mas agora não será mais sua responsabilidade definí-lo. Para isso, invocaremos o método **index** do controlador de produtos e passaremos para ele este método como segundo parâmetro. **Por ser um callback, não escrevemos os parênteses**.

```
{}  router.get('/', produtosController.index);
```

O código final

No arquivo do controlador, escrevemos os métodos de cada pedido e exportamos o objeto literal resultante.

```
{}
const controlador = {
    index: (req, res) => {
        res.send('Index de produtos');
    },
};
module.exports = controlador;
```

No arquivo de rotas, importamos o nosso controlador e em cada rota, executamos o método correspondente.

```
{}
const produtosController = require('../controllers/produtosController');
router.get('/', produtosController.index);
```

Revisão de rotas

DigitalHouse>

“Através do sistema de rotas do Express, podemos definir de maneira simples como nossa aplicação irá responder segundo o **método HTTP** e a **rota** que está chegando ao **servidor.**”

Definindo uma rota

Através do Express, contamos com uma **estrutura básica** para definir cada uma das rotas e suas respostas de nossa aplicação:

app

Variável que armazena a **execução** do **Express**.

```
app.get('/', function (req , res) {
    res.send('Olá mundo!');
})
```


DigitalHouse>

Definindo uma rota

Através do Express, contamos com uma **estrutura básica** para definir cada uma das rotas e suas respostas de nossa aplicação:

Método

É o **método HTTP** que queremos atender. Os métodos mais comuns são: *get*, *post*, *put*, *delete* ou *patch*.

```
app.get('/', function (req , res) {
    res.send('Olá mundo!');
})
```


DigitalHouse>

Definindo uma rota

Através do Express, contamos com uma **estrutura básica** para definir cada uma das rotas e suas respostas de nossa aplicação:

Path

String que fará referência à **rota em que o cliente entrou**.
Em outras palavras, a URL que será recebida por requisição.

```
app.get('/', function (req , res) {
    res.send('Olá mundo!');
})
```


DigitalHouse>

Definindo uma rota

Através do Express, contamos com uma **estrutura básica** para definir cada uma das rotas e suas respostas de nossa aplicação:

Handler

Callback que tem a função de definir qual ação será executada quando a rota em questão é acessada.

```
app.get('/', function (req , res) {
    res.send('Olá mundo!');
})
```

Definindo uma rota

Através do Express, contamos com uma **estrutura básica** para definir cada uma das rotas e suas respostas de nossa aplicação:

Request *(primeiro parâmetro do handler)*

É um objeto literal com múltiplos métodos e propriedades. Representa o **pedido** do cliente.

```
app.get('/', function (req , res) {
    res.send('Olá mundo!');
})
```

Definindo uma rota

Através do Express, contamos com uma **estrutura básica** para definir cada uma das rotas e suas respostas de nossa aplicação:

Response *(segundo parâmetro do handler)*

É um objeto literal com múltiplos métodos e propriedades. Representa a **resposta** que o servidor vai retornar.

```
app.get('/', function (req , res) {
    res.send('Olá mundo!');
})
```

Definindo uma rota

Através do Express, contamos com uma **estrutura básica** para definir cada uma das rotas e suas respostas de nossa aplicação:

```
app.get('/', function (req , res) {
    res.send('Olá mundo!');
})
```

Lógica da rota

Definimos a **lógica** que irá lidar com a rota definida. Em geral, aqui geramos a resposta que o cliente verá em seu navegador.

Rotas parametrizadas

DigitalHouse>

“O Express nos permite criar **rotas dinâmicas**, nas quais definimos qual parâmetro será utilizado.”

Rotas parametrizadas

Podemos utilizar a mesma estrutura básica para definir uma rota e adicionar o parâmetro que queremos no **path** (caminho). Para poder acessar esse parâmetro, precisamos dar um nome a ele.

Para adicionar um parâmetro a uma rota, utilizamos dois pontos `:` seguido do **nome** que representa o dado que vai ser recebido na url. Cada parâmetro deve estar separado do resto da rota por barras `/`.

```
{}
app.get('/produtos/:id', function (req,res) {
   // código
})
```

Parâmetros obrigatórios

Os parâmetros em geral são obrigatórios. Isso quer dizer que a rota só responderá se receber o parâmetro.

Rota

É a URL que será recebida por requisição.

Parâmetro obrigatório

Definimos uma parte dinâmica dentro da rota.
Nesse caso, será obrigatório inserir o valor do parâmetro.

```
app.get('/produtos/:id', function (req,res) {
   // código
})
```

Parâmetros opcionais

Se queremos que um parâmetro seja opcional, devemos adicionar um ponto de interrogação `?` ao final do seu nome.

Nesse caso, a rota responderá independente de receber o parâmetro ou não.

Rota

É a URL que será recebida por requisição.

Parâmetro opcional

Quando usamos ponto de interrogação no final, significa que o parâmetro não é obrigatório.

```
app.get('/produtos/:id/:nome?', function (req,res) {
   // código
})
```


DigitalHouse>

Acessando os parâmetros

Quando usamos a propriedade **params** do objeto literal **request**, podemos **capturar** os parâmetros da rota.

A propriedade **params** é um **objeto literal** que armazena os parâmetros recebidos pela URL com a estrutura *propriedade:valor*, na qual cada **propriedade** terá o **nome** do parâmetro que definimos no **path**.

```
app.get('/produtos/:id/:nome?', function (req,res) {
   let idProduto = req.params.id;
   let nomeProduto = req.params.nome || 'Sem nome';
})
```

Rotas parametrizadas

Como vemos no navegador:

```
app.get('/produtos/:id', function (req,res) {
    let idProduto = req.params.id; // 6
})
```

O **número** que colocarmos no navegador no lugar do `:id`, é o que obteremos do objeto **request** a partir da propriedade `params.id`.

DigitalHouse>

Sistema de roteamento

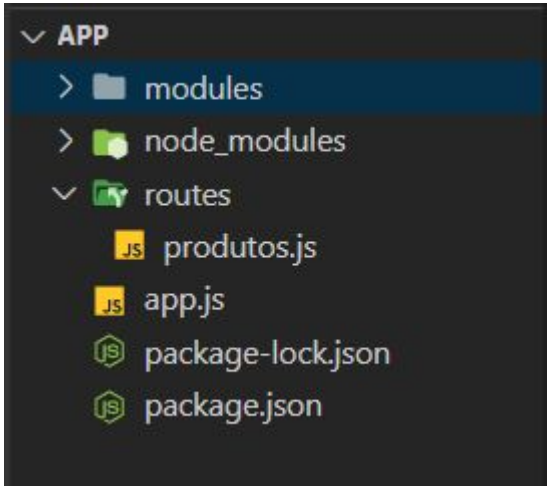
DigitalHouse>

“As **rotas** podem ser muito complexas e o Express nos permite **organizá-las** através de um **sistema de roteamento**.”

Arquitetura de arquivos

Na pasta **routes**, vamos guardar, para cada recurso, um arquivo *js* que administre os **pedidos** para as rotas que tenham a ver com esse recurso.

Cada arquivo *js* será um **módulo** a exportar, para depois importar dentro do entry point da aplicação (normalmente `app.js`).



O arquivo de rotas

Para começar a definir o sistema de roteamento de cada recurso, é necessário:

- Criar o **arquivo de rotas** do recurso.
- Importar o **módulo express**.
- Guardar a **execução** do método `router()` em uma variável nova.

```
{}
const express = require('express');
const router = express.Router();
```

Este método vai nos permitir **modularizar** por completo todo o **sistema** de **roteamento** de um jeito simples.

O arquivo de rotas

O próximo passo será definir todas as rotas que consideramos necessárias para cuidar dos distintos tipos de **pedidos** que este recurso receberá.

Para cada uma delas, vamos utilizar o objeto **router**, seguido do método que estamos processando.

```
{}
// Rota raiz dos produtos / Início
router.get('/', (req, res) => {
    // código
});
// Rota que mostra o detalhe de um produto
router.get('/detalhe/:id', (req, res) => {
    // código
});
```

O arquivo de rotas

Para que as rotas fiquem disponíveis, na última linha do arquivo, **exportamos** todo o conteúdo de **router**, para torná-lo visível.

```
{}
// Rota raiz dos produtos / Início
router.get('/', (req, res) => {
    // código
});
// Rota que mostra o detalhe de um produto
router.get('/detalle/:id', (req, res) => {
    // código
});

module.exports = router;
```

Implementando as rotas

A implementação é feita dentro do `app.js` , criando uma constante e importando o módulo que contém as rotas.

```
{}  const rotasProdutos = require('./routes/produtos');
```

Então, usaremos o método `use()`, colocando:

- Como primeiro parâmetro, uma string que será a rota base do recurso, neste caso: *produtos*. Por ser uma rota, deve começar com uma `/`.
- Como segundo parâmetro, a constante que definimos anteriormente.

```
{}  app.use('/produtos', rotasProdutos);
```

Implementando as rotas

É importante saber que a rota final será a concatenação da parte que está no app.js:

```
app.use('/produtos', rotasProdutos);
```

Com o que está definido no nosso arquivo de rotas:

```
router.get('/detalhe/:id', (req, res) => { // código });
```

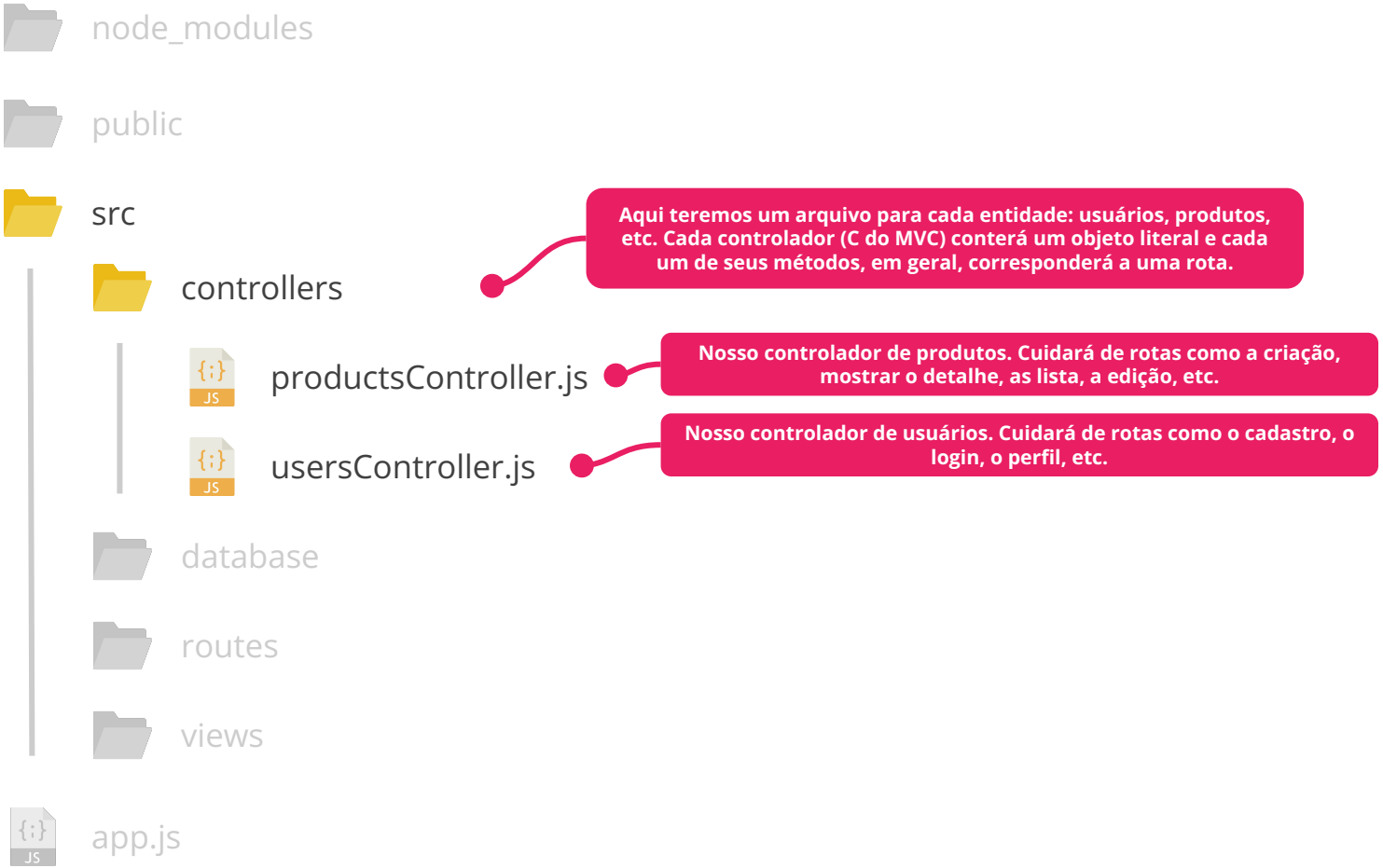
O resultado final será, então: `/produtos/detalhe/:id`

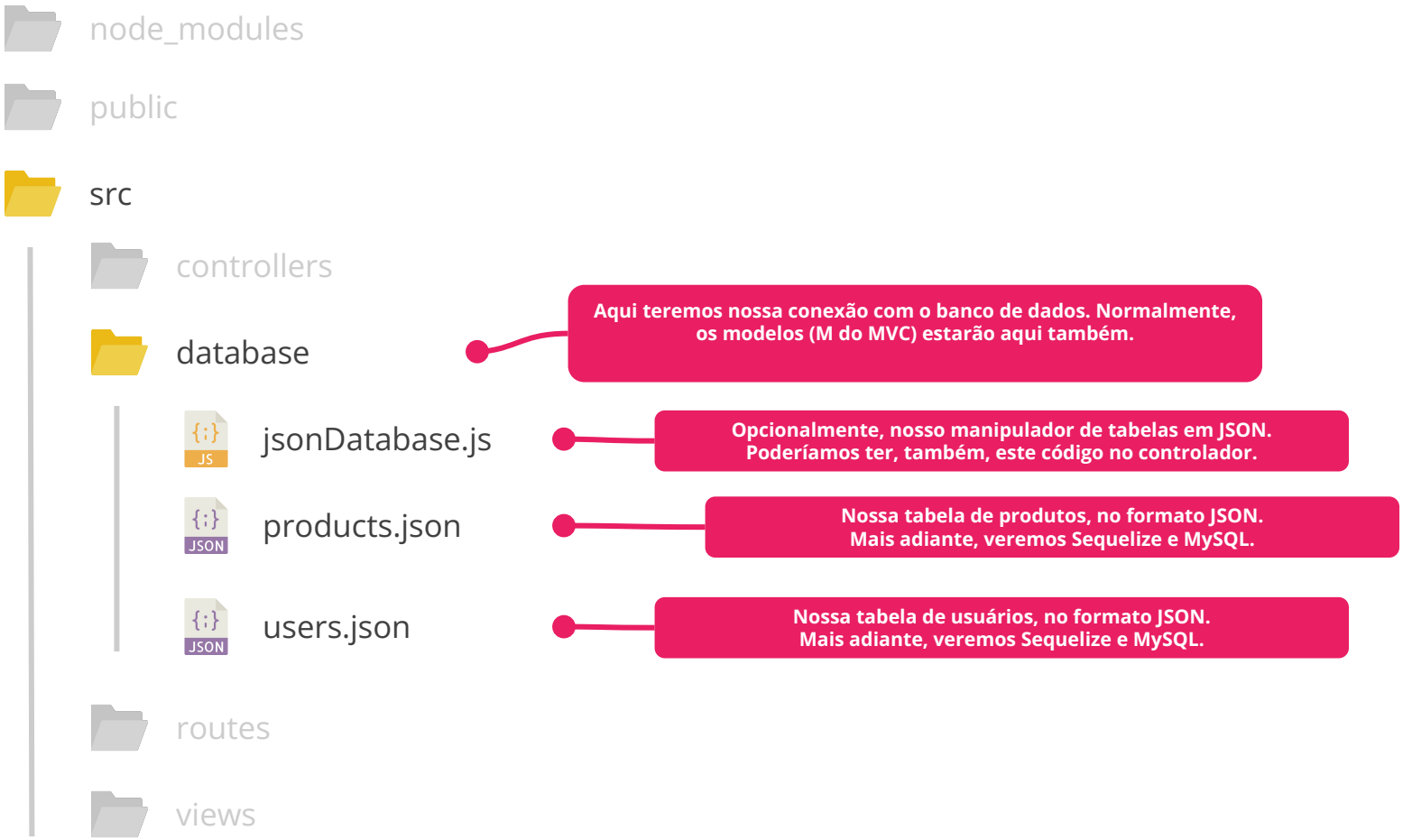
Como organizar seus arquivos

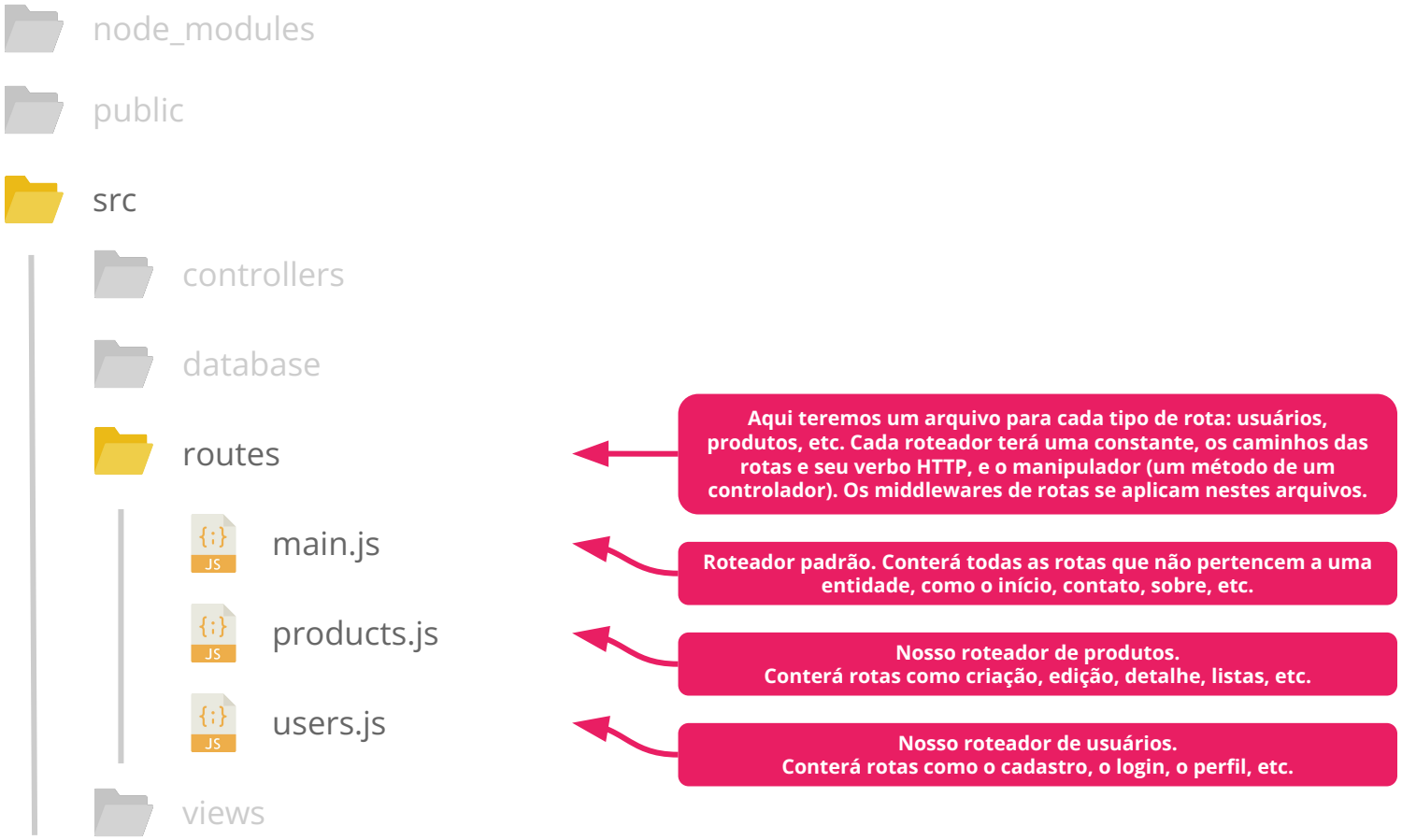
DigitalHouse>

- node_modules
  - Esta pasta é criada pelo Node e contém os módulos instalados e suas dependências.
- public
  - Aqui colocamos todos os recursos estáticos, que não passam pelo sistema de roteamento do Node: imagens, css e Javascript do front-end.
- src
  - Aqui deixaremos todo nosso projeto Node.
  - controllers
    - Aqui teremos um arquivo para cada entidade: usuários, produtos, etc. Cada controlador (C do MVC) conterá um objeto literal e cada um de seus métodos, em geral, corresponderá a uma rota.
  - database
    - Aqui teremos nossa conexão com o banco de dados. Normalmente, os modelos (M do MVC) estarão aqui também.
  - routes
    - Aqui teremos um arquivo para cada tipo de rota: usuários, produtos, etc. Cada roteador terá uma constante, os caminhos das rotas e seu verbo HTTP, e o manipulador (um método de um controlador).
    Os middlewares de rotas se aplicam nestes arquivos.
  - views
    - Aqui deixaremos todas nossas vistas (V do MVC). No nosso caso, serão arquivos ejs.
- app.js
  - Nosso ponto de entrada, conterá todos os módulos da aplicação.
- package-lock.json
  - Contém o registro das versões específicas dos pacotes instalados em nosso projeto.
- package.json
  - Contém o registro dos pacotes que o projeto precisa para funcionar e outras configurações do projeto.

node_modules
public
Aqui colocamos todos os recursos estáticos, que não passam pelo sistema de roteamento do Node.
css
Nossa pasta de folhas de estilo. Aqui também colocaremos as bibliotecas CSS, se não estivermos usando a partir de uma CDN.
img
Nossa pasta de imagens. Podemos ter subpastas se precisarmos separar, por exemplo, imagens de produtos e de usuários.
js
Nossa pasta de Javascript de front. Aqui ficarão as bibliotecas e os scripts próprios que utilizamos nas vistas.
Também podemos criar subpastas se tivermos muitos arquivos.
src
controllers
database
routes
views


node_modules
public
src
controllers
productsController.js
usersController.js
database
routes
views
app.js
Aqui teremos um arquivo para cada entidade: usuários, produtos, etc. Cada controlador (C do MVC) conterá um objeto literal e cada um de seus métodos, em geral, corresponderá a uma rota.
Nosso controlador de produtos. Cuidará de rotas como a criação, mostrar o detalhe, as lista, a edição, etc.
Nosso controlador de usuários. Cuidará de rotas como o cadastro, o login, o perfil, etc.


node_modules
public
src
controllers
database
Aqui teremos nossa conexão com o banco de dados. Normalmente, os modelos (M do MVC) estarão aqui também.
jsonDatabase.js
Opcionalmente, nosso manipulador de tabelas em JSON. Poderíamos ter, também, este código no controlador.
products.json
Nossa tabela de produtos, no formato JSON. Mais adiante, veremos Sequelize e MySQL.
users.json
Nossa tabela de usuários, no formato JSON. Mais adiante, veremos Sequelize e MySQL.
routes
views


node_modules
public
src
controllers
database
routes
main.js
products.js
users.js
views
Aqui teremos um arquivo para cada tipo de rota: usuários, produtos, etc. Cada roteador terá uma constante, os caminhos das rotas e seu verbo HTTP, e o manipulador (um método de um controlador). Os middlewares de rotas se aplicam nestes arquivos.
Roteador padrão. Conterá todas as rotas que não pertencem a uma entidade, como o início, contato, sobre, etc.
Nosso roteador de produtos.
Conterá rotas como criação, edição, detalhe, listas, etc.
Nosso roteador de usuários.
Conterá rotas como o cadastro, o login, o perfil, etc.

- node_modules
- public
- src
  - controllers
  - database
  - routes
  - views
    - partials
    - products
    - users

Aqui deixaremos todas nossas vistas (V do MVC). No nosso caso, serão arquivos ejs.

Aqui ficarão arquivos parciais, ou seja, aqueles que se repetem em várias páginas como o header, o menu, o footer, etc.

Aqui estarão todas as vistas que tenham relação com os produtos como a listas, o detalhe, os formulários de criação, etc.

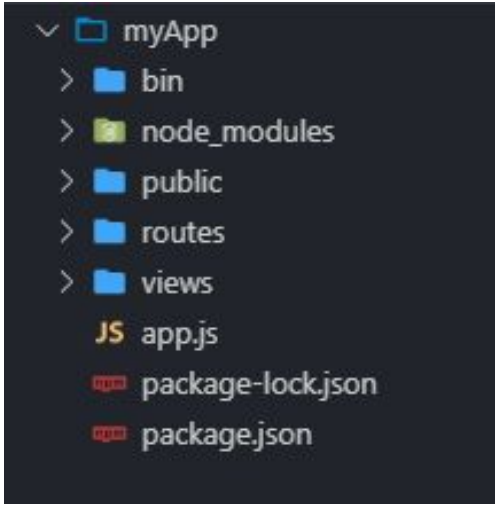
Aqui permanecerão todas as vistas que tenham relação com os usuários como o cadastro, o login, o perfil, etc.

Express-generator

DigitalHouse>

O que traz o express-generator?

Esta ferramenta apresenta uma grande vantagem na hora de iniciar um novo projeto, pois, ao instalá-la, traz consigo um esqueleto de pastas, arquivos e dependências que podem ser utilizadas para inicializar qualquer aplicação.



Estrutura de arquivos

- Na pasta raiz, encontramos o ponto de entrada `app.js` e o arquivo `package.json`.
- Dentro da pasta bin, encontramos o arquivo `www` sem extensão. Possui uma lógica interna definida e se encarregará de fazer rodar a aplicação.
- Dentro da pasta pública, podemos salvar todos os recursos estáticos de nosso aplicativo: CSS, imagens, JavaScript *front-end*.
- Dentro da pasta de rotas, gerenciamos o sistema de roteamento da aplicação. Encontramos os arquivos `index.js` e `users.js`.
- Dentro da pasta de views, encontramos duas *views* iniciais que o gerador traz: `index.ejs` e `error.ejs`.

“

Este gerador não inclui a pasta de *controllers* e seus arquivos! Portanto, teremos de criá-lo se quisermos respeitar a arquitetura com a qual temos trabalhado (MVC).

”

Instalando express-generator

Para poder trabalhar com o gerador de projetos, devemos instalá-lo globalmente para poder usá-lo no momento que necessitemos.

```
>_ npm install express-generator -g
```

Em seguida, usando o comando `express` seguido de um nome, poderemos criar um projeto de Node.js e Express. Também, podemos configurar a ***engine* de visualização*** que queremos usar.

```
>_ express myApp --ejs
```

* Se não indicamos uma engine de visualização (*--ejs*), se instalará por padrão o Jade.

Instalando express-generator

Por fim, dentro da pasta do projeto, temos de executar o comando para instalar todas as dependências que foram configuradas no `package.json` para que o projeto funcione.

```
>_  npm install
```

Engines de visualização

As *engines* de visualização, também conhecidas como motores de modelo ou *template engines*, permitem-nos criar uma estrutura dinâmica para as visualizações do nosso projeto. Ou seja, permite definir blocos de conteúdo que podem ser preenchidos com dados variáveis.

Entre os mais populares estão EJS, Jade, Pug e Handlebars.

EJS

<% Embedded JavaScript %>

pug

handlebars

O que é Nodemon?

Essa ferramenta nos permite **monitorar** constantemente o **servidor** Node.js.

Cada vez que fizermos modificações no código do nosso projeto, o **Nodemon** se encarregará de fazer as "*stops*" e "*reloads*" do nosso servidor em Node.js. Portanto, não teremos que fazer isso manualmente.

Iniciar o servidor

Dentro da **pasta do projeto** devemos rodar o seguinte comando, com o qual indicaremos o arquivo que queremos executar.

```
>_  nodemon bin/www
```

Por último, devemos entrar em **http://localhost:3000** para comprovar que o servidor subiu corretamente.

Views com EJS

DigitalHouse>

“

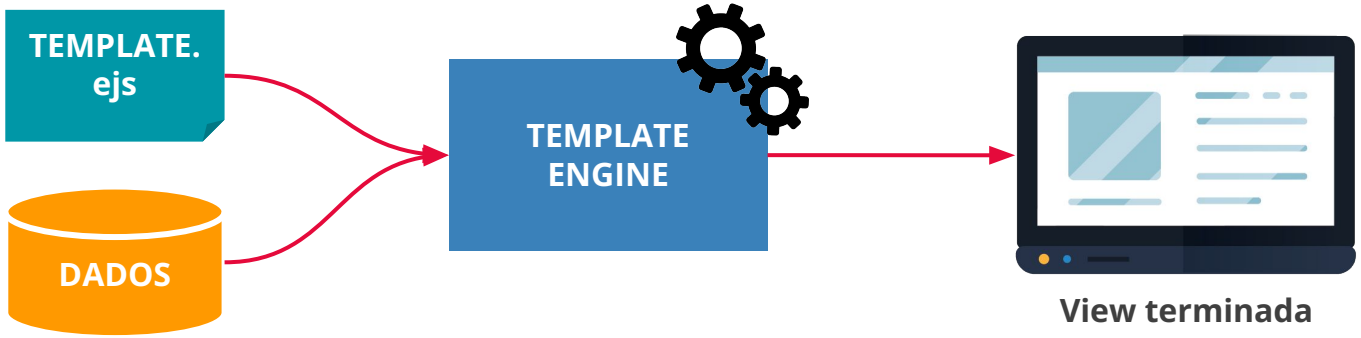
Um template engine é um mecanismo de modelo que nos permite gerar um arquivo que contém estruturas dinâmicas.

”

Como funcionam?

Os *template engines* podem ser usados quando precisamos exibir conteúdo dinâmico.

De um lado, temos os **dados** e do outro, o ***template***, onde podemos definir blocos de conteúdo que o *template engine* irá então **preencher** com **dados variáveis.**



Benefícios de usar *template*

- **Promove a organização** do nosso código-fonte em camadas, separando responsabilidades (MVC).
- **Nos permite reutilizar partes de *views* que se repetem** no projeto, como o *header*, o *footer*, os menus, os produtos, etc.
- Quando a informação pode mudar constantemente, usar *templates* **evita modificações recorrentes**.
- Podemos usar JavaScript incorporado no HTML mais facilmente.

Passo a passo

1. Instalar **EJS**.
2. Configurar **EJS** como o *template engine* da aplicação.
3. Configurar o acesso à pasta de recursos estáticos.
4. Usar `res.render()` para renderizar a *view*.

```
>_  npm i ejs
```

```
{}  app.set('view engine', 'ejs')
    app.use(express.static('public'))
```

```
{}  app.get('/', (req, res) => {
            res.render('index')
        }
    }
```

“

O EJS vem com um conjunto de *tags* que nos permite integrar as funcionalidades do JavaScript às nossas estruturas HTML.

”

<% %>

Usamos para integrar código JavaScript, como **condicionais**, **estruturas** de **controle** de **fluxo**, etc.

<%= %>

Usamos para integrar código e **imprimir** o **resultado** de uma expressão ou um valor.

Enviando dados às views

```
{}
const filmes = ['Deadpool', 'The Joker', 'Batman'];
const controller = {
  mostrarFilmes: (req, res) => {
    res.render('filmes', {lista: filmes, ativa: false, genero: ação })
  }
}
```

```
html
<% for(let filme of lista) { %>

      <% if (ativa) { %>
            <h2>O título do filme é <%= filme %></h2>
      <% } %>
<% } %>
```

Vistas parciales

html

```html
<!-- views/partials/head.ejs -->
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="/css/master.css">
  <title>Mi web</title>
</head>
```

html

```html
<!-- views/home.ejs -->
<html>
  <%- include('./partials/head') %>
    <body>
        <header>
            <div class="top-header">
                <img src="images/logo-mercado-livre.svg" alt="Logo de MercadoLivre">
            </div>
```

Introdução a template engines

DigitalHouse>

“Um template engine, ou mecanismo de modelos, nos permite gerar vistas (views) que contenham informação e estruturas dinâmicas.”

Onde estamos na arquitetura MVC?

Se lembrarmos o fluxo da arquitetura MVC, no caso dos templates engines, nos encontramos na parte da vista.

Em outras palavras, preparando a informação que o nosso cliente verá quando respondermos seu pedido.

Como funcionam?

Os mecanismos de modelo, ou template engines, podem ser utilizados quando precisamos mostrar conteúdo dinâmico.

Por um lado, temos os **dados** e por outro, o **template**, onde podemos definir blocos de conteúdo, em que o template engine irá **preencher** com **dados variáveis**.

Para saber mais

Como vimos, existem vários mecanismos de template para usar com Express.

Nós vamos ver EJS, mas te convidamos a pesquisar mais a fundo as outras possibilidades, com as quais talvez trabalhe no futuro.

Por enquanto, recomendamos seguir se aprofundando sobre todas as possibilidades que o **EJS** nos fornece.

Mecanismos de modelos

No nosso caso, iremos trabalhar com **EJS**, pela simplicidade com que pode ser aplicado em um projeto. Mas existem muitos outros mecanismos de modelos, como o Mustache, Handlebars e Pug.

Mustache

Handlebars

handlebars

EJS

EJS

<% Embedded JavaScript %>

PUG

pug

Benefícios de usar mecanismo de modelos

- **Promove a organização** do nosso código-fonte em camadas, separando as responsabilidades (MVC).
- **Nos permite reutilizar partes de vistas que se repetem** no projeto, como o header, o footer, os menus, os produtos, etc.
- Quando a informação pode mudar constantemente, usar templates **evita modificações recorrentes**.
- Podemos usar JavaScript embutido no HTML mais facilmente.

Instalação e implementação

DigitalHouse>

“Para instalar um **template engine** no nosso projeto, usaremos o **terminal**, através do gerenciador de dependências **NPM**.”

Compreendendo EJS

EJS, assim como outros mecanismos de modelos, nos permite ter vistas dinâmicas, onde a informação é processada antes de ser enviada para o cliente.

Uma das particularidades que o **EJS** tem é a extensão que possuem seus arquivos: em vez de utilizar **.html**, devemos criá-los com a extensão **.ejs**, para que o Express possa processá-los.

Os passos que devemos seguir para poder usar EJS em um projeto são:

1. **Instalar EJS.**
2. **Configurar a pasta que conterá as vistas .ejs (opcional).**
3. **Criar as vistas que precisamos com a extensão .ejs.**
4. **Renderizar as vistas.**

Instalar EJS

O primeiro passo **é instalá-lo** em nossa aplicação, ou seja, dentro da pasta onde temos o projeto. Para isso, executamos o seguinte comando no terminal:

```
>_ npm i ejs
```

Importante: se aplicação foi criada com **express-generator**, é provável que já tenhamos instalado EJS caso o comando tenha sido executado com a opção: `--view=ejs`. Caso não, para termos certeza disso, podemos verificar o arquivo **package.json** e procurar se aparece uma versão do EJS na aplicação.

Configurando EJS

Quando o **EJS** terminar de instalar, precisamos avisar para o **Express** qual é o mecanismo de modelos que vamos utilizar.

Ou seja, em nosso **ponto de entrada**, que normalmente será o **app.js**, devemos especificar o método `set()` com a propriedade `'view engine'` e o nome do motor, neste caso o **ejs**.

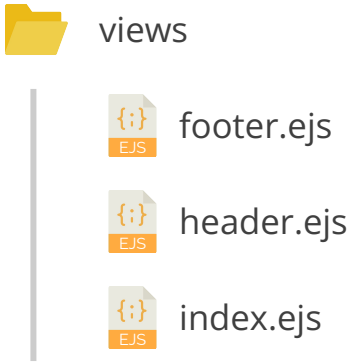
```
{}  app.set('view engine', 'ejs');
```

Configurando EJS

Ao implementar um mecanismo de modelos, todas as vistas que criarmos em nosso projeto deverão ser **armazenadas** em uma **pasta específica**.

Por padrão, o Express vai procurar essas vistas em uma pasta chamada **views**, que temos de criar em nosso diretório.

Ali, guardaremos todos os arquivos de vistas com a **extensão correspondente**, baseados no mecanismo de modelos que temos instalado. Para nosso caso é: **.ejs**.



Configurando a pasta de vistas de EJS

Como já mencionamos, o Express tem uma pasta de vistas por padrão, a **views**, que se localiza no diretório raiz do projeto.

Caso haja a necessidade de mudar o nome ou a localização da pasta, novamente utilizaremos o método `.set()`.

Neste caso, devemos passar a string `'view'` e o caminho relativo à nova pasta.

Este passo é opcional, já que vamos usar a pasta padrão.

```
{}  app.set('view', './pasta-de-vistas');
```

O método .set()

O método `.set()` permite definir configurações do Express. É executado na variável que esteja associada à execução do Express, na maioria dos casos chamada `app`.

Recebe duas strings como parâmetros:

- O primeiro será o que queremos configurar.
- O segundo será o valor da configuração.

Neste exemplo, estamos configurando o mecanismo de modelos para que seja EJS.

```
{}  app.set('view engine', 'ejs');
```

Minha primeira view com EJS

DigitalHouse>

1. Instalamos EJS utilizando NPM.

```
>_ npm i ejs
```

2. Configuramos EJS como o template engine da aplicação.

```
{} app.set('view engine', 'ejs')
```

3 Configuramos o acesso à pasta de recursos estáticos (public).

```
app.use(express.static('public'))
```

4 Usamos res.render() para renderizar a view.

* Importante lembrar que não é necessário colocar o nome da pasta na extensão da view.

```
app.get('/', (req, res) => {
        res.render('index')
    }
}
```

EJS: renderizando views

DigitalHouse>

“No modelo **MVC**, as **views** se comunicam com os **controllers** e recebem a informação por eles, recebidas dos **models**.

Para poder **renderizar** uma **view** é importante deixar claro para o **controller qual view** queremos enviar para o navegador.”

O processo de renderização

As views **EJS** consistem basicamente em código HTML mesclado com **tags de EJS**.

As views são colocadas, em geral, na pasta **views** e levam a extensão **.ejs**.

O Express tem dois métodos para renderizar views:

- **app.render()**, usado para renderizar primeiro a view e depois passar o HTML para uma função callback que será encarregada de enviá-la para o browser.
- **res.render()**, usado para realizar os dois passos: renderizar a view e enviar o HTML resultante como resposta.

Renderizando com res.render()

Para indicar qual **view** vamos renderizar e enviar ao cliente, usamos o método `res.render()`.

O método render se encontra dentro do objeto **response** da requisição e recebe uma **string** como parâmetro: o nome do arquivo da view que queremos renderizar na tela.

Importante: quando passamos o nome do arquivo, **não há necessidade** de especificar a pasta onde essa view está armazenada - desde que tenhamos configurado corretamente o template engine com o **método use()** no arquivo **app.js**.

```
app.get('/', (req, res) => {
        res.render('index')
    }
}
```

Recursos estáticos

São os recursos públicos que gerenciamos em nosso aplicativo: imagens, arquivos css, vídeos, arquivos javascript, etc.

Para utilizá-los livremente em nosso projeto, é necessário **informar** ao **Express** onde vamos **armazenar** esses recursos.

Diretório/pasta

Path relativo à pasta onde armazenaremos os recursos estáticos.

Por convenção, é utilizado o nome `public`.

```
app.use(express.static('public');
```

Utilizando um recurso estático

Para acessar qualquer um desses recursos a partir de nossos arquivos, você só precisa declarar o caminho até esse recurso, sempre iniciando o path com uma barra `/`.

Por quê?

Porque o Express já sabe que nossos recursos estáticos estão na pasta `public` e então buscará os arquivos a partir dessa pasta.

Por tanto, ao escrever `/images/logo.png`, o path completo será na realidade `/public/images/logo.png`

```html
<img src="/images/logo.png">
```

Tags do EJS

DigitalHouse>

“EJS traz consigo um conjunto de **tags** que nos permitem integrar **funcionalidade** do JavaScript dentro de nossas estruturas HTML.”

EJS e HTML

Os arquivos do EJS seguem suportando **todas** as tags tradicionais do HTML e, portanto, poderemos desenvolver nossas vistas normalmente, só que agora adicionando **dinamismo**.

A parte dinâmica irá entre as tags do EJS e podemos usar toda a funcionalidade que nos é provido pelo JavaScript.

Em outras palavras, quando precisamos que as vistas renderizem conteúdo dinâmico, utilizamos as tags `<% %>`, que são fornecidas pelo **template engine** do EJS. Dentro delas, colocamos a parte dinâmica: `<%= user.firstname %>`, onde `user.firstname` é uma variável.

A tag <% %>

Esta tag nos permite incorporar código do JavaScript, como condicionais, estruturas de controle de fluxo, declaração de variáveis, etc.

Para utilizá-la, temos que fechar cada bloco de código JavaScript entre a tag **<%** de abertura e a tag **%>** de fechamento.

Se estamos usando uma estrutura, como uma condicional ou um ciclo, no meio podemos escrever qualquer conteúdo HTML que queremos que seja exibido ou que se repita.

```
<% if(4 < 5) { %>
    <h2>O 4 é menor que 5</h2>
<% } %>
```

A tag <%= %>

Esta tag nos permite imprimir um valor dinâmico e, desta forma, incorporá-lo na estrutura HTML.

Após o valor dinâmico que buscamos imprimir, temos que colocá-lo entre a tag **<%=** de abertura e a tag **%>** de fechamento.

Neste exemplo, combinamos impressão de tag com um bloco if.

```
<% if(nome) { %>
    <h2>Olá <%= nome %></h2>
<% } %>
```

<% %>

Usamos para embutir código do JavaScript, como **condicionais**, **estruturas** de **controle** de **fluxo**, etc.

<%= %>

Usamos para embutir código e **imprimir** o **resultado** de uma expressão ou um valor.

{código}

```
for(let i = 1; i < 5; i++) {


}
```

Como primeiro passo, podemos escrever a **sentença** do JavaScript que queremos **embutir** em nossa estrutura.

{código}

```
<div class="top-header">
<% for(let i = 1; i < 5; i++) { %>

<% } %>
</div>
```

Depois, devemos colocar cada linha **dentro** das tags de abertura e fechamento do **EJS**, que nos permitem embutir código JavaScript: **<%** e **%>**.

{código}

```
<div class="top-header">
<% for(let i = 1; i < 5; i++) { %>
    <p> Experimentando as tags do EJS </p>
<% } %>
</div>
```

```
Experimentando as tags do EJS
Experimentando as tags do EJS
Experimentando as tags do EJS
Experimentando as tags do EJS
```

Finalmente, escrevemos o **conteúdo** que queremos mostrar a cada iteração do laço declarado.

Aqui, podemos incorporar as tags HTML que quisermos.

{código}

```
<div class="top-header">
<% for(var i=1; i<5; i++) { %>
    <p> Vamos ver a volta <%= i %> </p>
<% } %>
</div>
```

```
Vamos ver a volta 1
Vamos ver a volta 2
Vamos ver a volta 3
Vamos ver a volta 4
```

Também podemos combinar as tags e dentro do ciclo imprimir conteúdo dinâmico.

Para isso, utilizamos a tag **<%= %>**.

Parâmetros compartilhados

DigitalHouse>

“A grande **vantagem** de utilizar um mecanismo de modelo (template engine) é que nos permite **compartilhar** uma informação **dos controladores** para as **views** e fazer o que quisermos com ela.”

Como mostrar uma **view?**

No objeto **response**, executamos o método `render()`, passando como argumento o nome da **view** que queremos renderizar.

Lembre-se de que não faz falta o diretório inicial, nem a extensão: isso já foi configurado anteriormente.

```
{}
const controller = {
    mostrarFilmes: (req, res) => {
        res.render('filmes')
    }
}
```

Como enviar dados para uma view?

O método `render()` pode receber um **objeto literal** como segundo parâmetro.

Nesse objeto, podemos armazenar a informação que queremos enviar junto com o nome da view que estamos renderizando.

Toda a informação que enviarmos estará disponível para ser usada na view.

```
{}
const controller = {
    mostrarFilmes: (req, res) => {
        res.render('filmes', { } )
    }
}
```

Enviando mais de um dado para a view

Enviar mais de um dado para a view é tão simples como adicionar mais propriedades ao objeto literal que havíamos preparado.

```
{}
const filmes = ['Deadpool', 'The Joker', 'Batman'];
const controller = {
  mostrarFilmes: (req, res) => {
    res.render('filmes', {
      lista: filmes,
      emOferta: true,
      genero: 'Ação'
    })
  }
}
```

O objeto de dados

No objeto, podemos criar tantas propriedades e valores quanto forem necessários. O nome de cada propriedade será o nome que usaremos para trabalhar com essa informação na view.

```
{}
const nomeDoFilme = 'Batman';
const controller = {
    mostrarFilme: (req, res) => {
        res.render('filme', { filme: nomeDoFilme })
    }
}
```

Como receber os dados na view?

Para poder renderizar a variável enviada pelo controlador, colocamos a mesma na tag de impressão: `<%= %>`.

Como `filme` é uma variável do JavaScript, podemos usar todos os métodos disponíveis. Sabendo que é uma string, podemos escrever algo como

`<%= filme.toUpperCase() %>` para que seja exibido **'BATMAN'**.

```html
<div>
    <h1>Filme Título: </h1>
    <span><%= filme %></span>
</div>
```

“Podemos criar a quantidade de propriedades que quisermos neste objeto.

A view, por sua vez, não receberá um objeto literal com todas essas propriedades, mas sim **receberá as propriedades soltas como variáveis**.”

Como receber os dados na view?

Da mesma forma que vimos antes, podemos acessar cada propriedade do objeto que criamos no controlador usando seus nomes diretamente, já que o Express se encarrega de transformá-las em variáveis dentro da view.

```html
<% for(let filme of lista) { %>
     <h2>O título do filme é <%= filme %> e o gênero é
<%= genero %></h2>
     <% if (emOferta) { %>
          <p>25% de desconto comprando este filme</p>
     <% } %>
<% } %>
```

Partial files

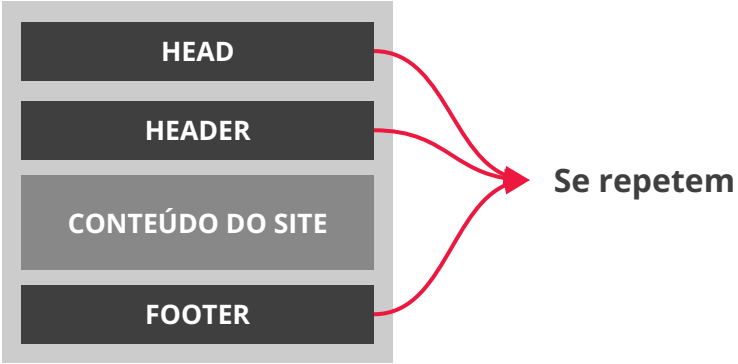
DigitalHouse>

“Existem formas de **reutilizar** o mesmo **HTML** em várias views, conseguindo assim simplificar muito o **desenvolvimento** do nosso projeto. Isso é alcançado com os arquivos parciais, conhecidos como **partial files**.”

Como modularizar?

A primeira coisa a se fazer é identificar aquelas partes do **código** que vemos sendo **repetidas** em todos os arquivos do projeto.

Em uma estrutura comum de HTML, podemos ver que todos os arquivos facilmente contam com: um head, uma barra de navegação e um footer, só para mencionar algumas.

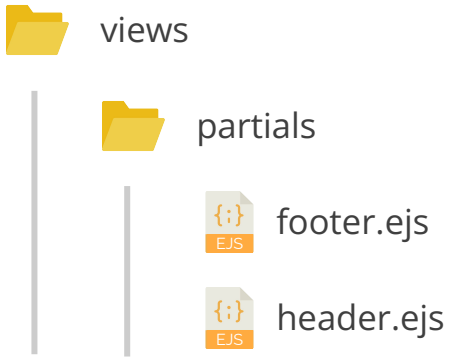


Como modularizar?

Cada uma destas partes irá se converter em um **arquivo novo**, que conterá **unicamente** essa **porção** de código. As chamamos **views parciais**.

Estes arquivos também são guardados dentro da pasta **views**, ou na que tivermos configurado em seu lugar.

Uma boa prática é ter uma pasta separada para as views parciais.



Views parciais

Uma vez identificadas as partes que se repetem dentro das nossas views:

- Tiramos cada parte e a colocamos em um novo arquivo.
- Damos um nome que represente a parte que este contém.
- Adicionamos a extensão correspondente, por exemplo: `head.ejs`.

```html
<!-- views/partials/head.ejs -->
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="/css/master.css">
  <title>Meu site</title>
</head>
```

Incluindo as views parciais

Para acessar o código que modularizamos, precisamos fazer uso da função `include()`, nativa do EJS. Ela recebe como parâmetro uma string, que será o caminho até o arquivo que queremos incluir.

Por último, devemos escrever a tag `<%- %>`, que imprimirá no documento o conteúdo exato retornado do `include()`.

```html
<!-- views/home.ejs -->
<!DOCTYPE html>
<html>
    <%- include('./partials/head') %>
    <body>
        <header>
            <div class="top-header">
```

“Modularizar nos oferece vários benefícios: **melhorar** nosso código, ser menos **repetitivo** e simplificar a **manutenção** do código.”

CRUD
Episódio I

DigitalHouse>

Índice

1. Métodos HTTP
2. Processamento GET
3. Processamento POST
4. Processamento PUT e DELETE
5. Erro 404

“

O protocolo **HTTP** define uma série de **regras** que devem ser seguidas para que as informações sejam **processadas.**

”

Métodos de transação

O fluxo de **transações** proposto pelo protocolo HTTP é acompanhado por diversos **métodos** que **definem** o que acontecerá a cada solicitação que um cliente fizer a um servidor.

No **Express** podemos implementar os métodos na execução da `aplicação` Express ou no sistema de roteamento do sistema do ***`router`***.

Como boa prática, sugere-se trabalhar os métodos diretamente na execução do **Express**, entendendo que é a arquitetura mais recomendada.

Tratamento das solicitações GET

```
// roteador
router.get("/filmes", (req, res) => { res.render('filmes')} );

// roteador com controller
router.get("/filmes", filmesController.todos );
const controller = {
todos: (req, res) => { res.render('filmes')}
};

// sobre a execução do express
app.get("/filmes", (req, res) => { res.render('filmes')} );
```

{}

CALLBACK

Tratamento das solicitações POST

Na solicitação do **request** encontramos a propriedade *body*, que é um objeto literal contendo **todas** as informações vindas do formulário:

- o nome de cada **chave** desse objeto será o nome do atributo *name* de cada *input* do formulário, e
- o **valor** será o dado que for inserido naquele campo.

html

```html
<form method="POST" action="/filme/criar">
    Título: <input type="text" name="titulo" value=""> // value="Batman" p.ex.
    ...
</form>
```

{}

```
router.post('/filme/criar', (req,res) => {
    console.log(req.body) // { titulo: Batman }
});
```

Capturar a informação

Para trabalhar com os dados que saem do formulário, é necessário **configurar** o ambiente da nossa aplicação para **capturar** esses dados.

Teremos que adicionar essas duas linhas de código no arquivo `app.js`.

```
{}
app.use(express.urlencoded({ extended: false }));
app.use(express.json());
```

req.body

Para **fechar** o ciclo do **request** e **response** do servidor, é necessário fazer um **redirecionamento** — após implementada a lógica — usando o método **redirect()** sobre o **response**:

```
{}  res.redirect('/filmes');
```

Lidar com solicitações PUT

Num contexto em que desejamos **modificar** os dados de um filme armazenado em nosso sistema, teríamos que criar duas **rotas**: uma para **mostrar** o formulário de **edição** e outra para **processar** a informação.

Em ambas as rotas vamos definir um parâmetro que nos ajude a identificar o recurso único que queremos modificar. Geralmente, o **id** é usado.

```
{}
// rota que envia um formulário de edição à view → GET
router.get('/filme/:id/editar', (req,res) => {res.render('editar')});

// rota que processa a informação vinda do formulário → PUT
router.put('/filme/:id/editar/', (req,res) => {...});
```

Lidar com solicitações DELETE

Em um contexto em que desejamos **excluir** um filme armazenado em nosso sistema, teríamos de criar uma rota que se encarregaria de buscar esse recurso e eliminá-lo. Na rota, devemos definir um parâmetro que nos ajude a identificar o recurso que queremos eliminar. Geralmente, o **id** é usado.

```
{}
// rota que processa a informação do formulário → DELETE
router.delete('/filme/:id/eliminar/', (req,res) => {...});
```

Habilitar métodos HTTP

Os métodos **PUT** e **DELETE** não são suportados por todos os navegadores. Para implementá-los em nosso formulário, é necessário instalar o pacote `method-override`:

```
>_  npm install method-override --save
```

Uma vez instalado, tem que configurar a aplicação em `app.js` para poder **sobrescrever o método original** e poder implementar os métodos **PUT** e **DELETE**:

```
{}  const methodOverride = require('method-override');
    app.use(methodOverride('_method'));
```

Configurar o formulário

Para terminar de habilitar o envio da informação através de qualquer um destes métodos, temos de adicionar uma **query string** ao `action` do formulário:

- ?_method=PUT
- ?_method=DELETE

Desta forma, estamos dizendo que, independente do método original que o formulário tenha definido, queremos enviar a informação usando PUT ou DELETE.

```html
<form method="POST" action="/filme/:id/editar?_method=PUT">
    ...
</form>
```

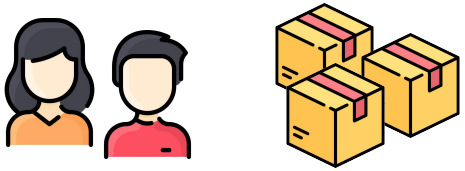
Erro 404

Dentro deste método, iremos definir como o sistema irá reagir cada vez que o usuário quiser acessar uma rota que não existe. No `response`, implementaremos o método `status ()`, seguido pelo método `render ()`. Isso vai renderizar a *tela* que o usuário verá cada vez que este cenário surgir.

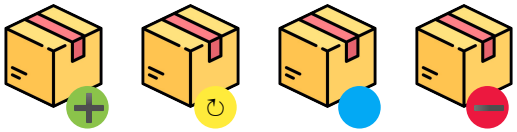
```
{}  app.use((req, res, next) => {
      res.status(404).render('not-found');
    })
```

O que é um CRUD?

DigitalHouse>



Na maior parte das aplicações, nós temos que manipular alguns registros como: produtos, usuários, categorias, etc.



Essa manipulação, em geral, consiste em criar, ler, listar, atualizar ou excluir registros.

CRUD

Em programação, chamamos esse conjunto de ações de **CRUD** (por suas siglas em inglês).

CRUD

- **C**reate / Criar
- **R**ead / Ler
- **U**pdate / Atualizar
- **D**elete / Excluir

Métodos HTTP

DigitalHouse>

GET → SOLICITAR DADOS

POST → ENVIAR DADOS / CRIAR REGISTROS

PUT → ATUALIZAR REGISTROS (o registro inteiro)

PATCH → ATUALIZAR REGISTROS (apenas algumas partes)

DELETE → EXCLUIR REGISTROS

ℹ️ → Os dados viajam na URL e a informação é visível para todos.

ℹ️ → Os dados requisitados podem ser guardados em cache e guardados em marcadores.

⚠️ → Não é utilizado para enviar informações sensíveis, como senhas.

👎 → Tem restrição de quantidade de informação.

👍 → Os dados trafegam ocultos no corpo da requisição.

ℹ️ → Os pedidos não podem ser guardados em cache e não podem ser guardados em marcadores.

👍 → É mais seguro que o GET, utilizado para manipular dados sensíveis, como senhas.

👍 → Não tem restrições com relação ao tamanho.

ℹ️ → São reenviados quando recarregam um site.

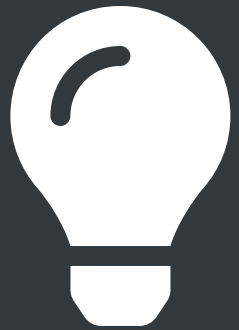
✔ Funciona com formulários HTML

✖ Não funciona com formulários HTML

Métodos HTTP

DigitalHouse>

“O protocolo **HTTP** define uma série de **regras** necessárias para que a informação possa ser **processada**.”



Métodos de transação

O fluxo de transações que o protocolo HTTP propõe vem acompanhado de vários métodos que definem o que acontecerá com cada requisição que um cliente enviar para um servidor.

No Express, podemos implementar os métodos sobre a instância do express - `app` - ou sobre o sistema de rotas - `router`.

Invocar os métodos diretamente na instância do Express é uma boa prática, pois entende-se que é a arquitetura mais recomendada.

O método GET

Com este método, é possível **solicitar** dados do servidor.

Ao acessar uma página através de uma **URL**, estou fazendo uma requisição do tipo **GET**.

O método POST

Com este método, é possível **enviar** dados do servidor.

Ao fazer um cadastro em um site novo com meus dados, estou fazendo uma requisição do tipo **POST**.

Diferenças entre GET e POST

As **requisições** do tipo **GET**...	As **requisições** do tipo **POST**...
viajam pela **URL** e man têm a informação **visível**	viajam **ocultas** e man têm a informação **encriptada**
podem ser colocadas em cache	**não podem** ser colocadas em cache
podem ser armazenadas em favoritos	**não podem** ser armazenadas em favoritos
têm restrições de tamanho	**não têm** restrições de tamanho
são utilizadas para manipular dados que **não sejam sensíveis**	são utilizadas para manipular dados que **sejam sensíveis**
não são reenviadas ao recarregar a página	**são reenviadas** ao recarregar a página

O método PUT

Com este método, podemos **substituir** uma informação existente. Ao mudar minha informação pessoal no Instagram estou fazendo uma requisição do tipo **PUT**.

O método DELETE

Com este método, podemos **apagar** um registro existente no servidor. Ao apagar uma foto no Facebook estou fazendo uma requisição do tipo **DELETE**.

“Os formulários HTML não suportam os métodos especiais **PUT**, **PATCH** e **DELETE**, de modo que alguns frameworks têm algumas funções para os simular.”

Método GET

DigitalHouse>

“As solicitações feitas por **GET** são todas aquelas que vêm diretamente na **URL** do navegador ou internamente na página a partir de um link.”

Aplicando solicitações GET

Geralmente, usamos o método GET para:

- Retornar **views**.
- Retornar **arquivos**.
- Retornar **dados**.

Quando definimos uma rota, podemos fazê-lo diretamente na **execução do express**, implementar um **sistema de roteamento** ou **incorporar controllers** responsáveis pelo gerenciamento das rotas. Em todos os casos, as rotas receberão dois parâmetros:

- Uma string com a rota que estamos processando;
- Uma callback onde escrevemos a lógica para lidar com a solicitação que está chegando.

Aplicando solicitações GET

```
// Roteador
router.get("/filmes", (req, res) => { res.render('filmes')} );

// Roteador com controller
router.get("/filmes", filmesController.todos);
...
const filmesController = {
    todos: (req, res) => { res.render('filmes')}
};

// Execução do express
app.get("/filmes", (req, res) => { res.render('filmes')} );
```

Aplicando solicitações GET

```
// Roteador
router.get("/filmes", (req, res) => { res.render('filmes')});

// Roteador com controller
router.get("/filmes", filmesController.todos);
...
const filmesController = {
  todos: (req, res) => { res.render('filmes')}
};

// Execução do express
app.get("/filmes", (req, res) => { res.render('filmes')});
```

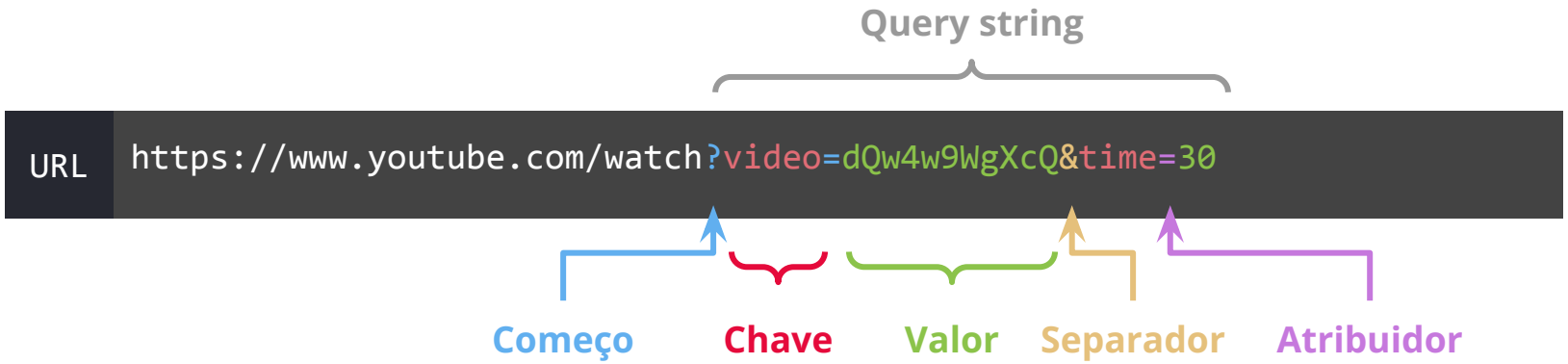
CALLBACK

DigitalHouse>

Query string

É uma cadeia de **texto** conhecida como string de consulta, que navega na URL ao fazer uma solicitação ao servidor mediante o método GET.

- Começa no final da rota com o sinal de interrogação `?`.
- É formada por um ou mais pares de `chave=valor`.
- No caso de haver mais de um par, eles são separados pelo caracter `&`.



Query string

Para **acessar** a query string dentro da callback que lida com a solicitação, usamos a propriedade **query** do objeto **request**.

Essa propriedade é um objeto literal, em que suas chaves e valores serão os mesmos que os da url.

É importante ter em conta que todos os dados que chegam serão do tipo string.

```
{}
// https://www.youtube.com/watch?video=dQw4w9WgXcQ&time=30
console.log(req.query.video)
// 'dQw4w9WgXcQ'
```

Método POST

DigitalHouse>

Aplicando Solicitações POST

Normalmente, usamos o método **POST** para:

- Enviar informações confidenciais para o servidor.
- Criar um novo recurso.

Quando definimos uma rota, podemos fazê-lo diretamente na execução do Express, implementar um sistema de roteamento e também incorporar controladores que se encarregam de tratá-los.

Em todos os casos, as rotas receberão dois parâmetros:

- Uma string com a rota que iremos processar.
- Um retorno de chamada onde definiremos qual lógica será executada quando o cliente solicitar essa rota.

Aplicando Solicitações POST

Num contexto em que queremos adicionar um novo filme ao nosso sistema, teríamos que criar **duas rotas**: uma que mostra o formulário de criação e outra que se encarrega de processar a informação.

```
{}
// rota que envia um formulário para a view → GET
router.get('/filme/criar', (req,res) => {res.render('criar')});

// rota que processa a informação do formulário → POST
router.post('/filme/criar', (req,res) => {...});
```

*Os nomes das rotas podem **ser iguais**, porque cada uma está implementando um método diferente.*

Configurar o formulário

Para enviar pedidos por **POST**, é necessário ter um **formulário**. Para que esse formulário envie os dados, devemos configurar duas propriedades:

- método → define o **método HTTP** que usaremos para enviar a informação, que neste caso é o POST. Se não configurarmos o método, ele será GET por padrão.
- ação → define a **rota** por onde essa informação irá viajar para ser processada. Se não configurarmos a ação, por padrão seria a mesma URL onde o formulário está localizado.

```html
<form method="POST" action="/filme/criar">
    ...
</form>
```

Capturar a informação

Para trabalhar com os dados enviados do formulário, é necessário **configurar** o ambiente de nossa aplicação para capturar essas informações.

Se estivermos trabalhando com `express-generator`, essa configuração será criada por padrão durante a instalação.

Caso contrário, teremos que adicionar estas duas linhas de código no arquivo `app.js`:

```
{}
app.use(express.urlencoded({ extended: false }));
app.use(express.json());
```

“Dessa forma, estamos especificando ao **aplicativo** que queremos capturar na forma de um **objeto literal** tudo o que vem de um **formulário**.

E, por sua vez, temos a possibilidade de converter essas informações em formato **JSON**, se necessário.”

req.body

No **request** da solicitação, encontramos a propriedade body, um objeto literal que conterá todas as informações do formulário:

- O nome de cada chave desse objeto será o nome do atributo *name* de cada entrada do formulário.
- O valor será os dados que foram inseridos naquele campo.

html

```html
<form method="POST" action="/filme/criar">
    Título: <input type="text" name="titulo" value="Batman">
    ...
</form>
```

{}

```
router.post('/filme/criar', (req,res) => {
    console.log(req.body) // { titulo: Batman }
});
```

res.redirect()

Para **fechar o ciclo** de requisição e resposta do **servidor**, é necessário fazer um **redirecionamento** — após a lógica implementada — utilizando o método **redirect()** na resposta.

No caso do filme que estávamos criando, após a criação, podemos levar o usuário para a lista de filmes.

```
router.post('/filme/criar', (req,res) => {
    // Procesamos la información enviada por el usuario
    // Si todo sale bien, lo redirigimos al listado de películas.
    res.redirect('/filme');
});
```


DigitalHouse>

Métodos PUT e DELETE

DigitalHouse>

“A informação que viaja como pedido **PUT** e **DELETE** deve ser feita através de um **formulário**.

Os métodos PUT e DELETE não são suportados nativamente pelos formulários, então teremos que utilizar métodos alternativos para enviar a informação.”

Aplicando solicitações PUT

Usamos o método PUT para:

- **Enviar informação** sensível para o servidor de maneira segura.
- **Modificar** um recurso existente.

Quando definimos uma rota, podemos fazê-la diretamente na **execução do express**, implementar um **sistema de roteamento** ou **incorporar controllers** responsáveis pelo gerenciamento dos routes. Em todos os casos, as rotas receberão dois parâmetros:

- Uma string com a rota que estamos processando;
- Um callback onde escrevemos a lógica para lidar com a solicitação que está chegando.

Aplicando solicitações PUT

Num contexto onde se quer **modificar** dados de um filme armazenado em nosso sistema, teríamos que criar duas **rotas**: uma que mostre o formulário de edição e outra responsável pelo processamento da informação.

Em ambas as rotas, devemos definir um parâmetro que nos ajude a identificar o recurso único que queremos modificar. Geralmente, usa-se **id**.

```
{}
// rota que envia um formulário de edição para a view → GET
router.get('/filme/:id/editar', (req,res) => {res.render('editar')});

// rota que processa a informação do formulário → PUT
router.put('/filme/:id/editar', (req,res) => {...});
```

Aplicando solicitações DELETE

Usamos o método DELETE para:

- **Excluir** um recurso existente.

Num contexto onde se quer excluir um filme armazenado em nosso sistema, teríamos que criar uma rota que se encarregue de buscar esse recurso e excluí-lo. Na rota, devemos definir um parâmetro que nos ajude a identificar o recurso que estamos querendo excluir. Geralmente, usa-se **id**.

```
{}
// rota que processa a informação do formulário → DELETE
router.delete('/filme/:id/excluir', (req,res) => {...});
```

Pedidos PUT e DELETE

No **callback** dos pedidos **DELETE** e **PUT**, teremos que definir a **lógica** do que queremos implementar com essa informação que recebemos.

Normalmente, ao terminar de processar o pedido, faremos um **redirecionamento** para outra página que mostre o resultado.

```
router.post('/filme/:id/excluir', (req,res) => {
    // Excluímos o filme solicitado
    // Se tudo der certo, redirecionamos à lista de filmes.
    res.redirect('/filmes');
});
```

Habilitar métodos HTTP

Os métodos PUT e DELETE não são suportados por todos os navegadores. Para poder implementá-los em nosso formulário, é necessário instalar o pacote `method-override`:

```
>_ npm install method-override --save
```

Uma vez instalado, teremos que configurar a aplicação em `app.js` para poder substituir o método original e implementar os métodos PUT ou DELETE:

```
{}
const methodOverride = require('method-override');
app.use(methodOverride('_method'));
```

Configurar o formulário

Para concluir a ativação de envio de informação por meio de um desses dois métodos, precisamos adicionar uma *query string* ao `action` do formulário de acordo com o método que queremos implementar:

- ?_method=PUT
- ?_method=DELETE

Desta forma, estamos esclarecendo que, independentemente do método original definido pelo formulário, queremos enviar a informação usando PUT ou DELETE.

```html
<form method="POST" action="/filme/:id/editar?_method=PUT">
    ...
</form>
```

“Embora os formulários ainda tenham o método POST configurado, a informação estará trafegando através do **método** que **definimos** em cada **rota**.

Dessa forma, é fácil identificar o que faz cada rota do nosso sistema, apenas vendo o método **implementado**.”

Erro 404

DigitalHouse>

“É um erro que o **servidor** envia quando **não encontra** o recurso que o **cliente** solicitou.”

Preparando o sistema

Todo sistema deve estar preparado para manipular as solicitações erradas.

O express traz consigo uma resposta automatizada para quando o usuário **solicita** um conteúdo que já **não existe** no servidor.

No entry point da aplicação - na maioria das vezes `app.js` - podemos configurar esta resposta, implementando o método `use()` na execução do Express.

```
{}
app.use((req, res, next) => {
    ...
})
```


DigitalHouse>

Preparando o sistema

Neste caso, `use()` deve receber uma callback com três argumentos: uma request, uma response e o próximo passo a ser executado depois da callback.

Dentro deste método, definiremos como o sistema vai reagir cada vez que o usuário quiser acessar uma rota que não existe.

Sobre a `response`, implementamos o método `status()` seguido do método `render()`, para poder renderizar a view que será apresentada ao usuário cada vez que este cenário ocorrer.

É importante que **esta rota seja a última**, já que só deve atuar se nenhuma das anteriores for executada.

```
app.use((req, res, next) => {
  res.status(404).render('not-found');
})
```

Lendo e escrevendo arquivos

DigitalHouse>

Índice

1. readFileSync()
2. writeFileSync()
3. appendFileSync()

1 readFileSync()

.readFileSync()

É um método que traz o pacote nativo **file system** que nos permite **recuperar dados** de um arquivo para poder usá-los.

Como **primeiro parâmetro** recebe a **rota** do arquivo que vamos ler.

```
{}
const fs = require('fs');
let sitcoms = fs.readFileSync('sitcoms.txt');
```

Para poder decodificar os dados que o método devolve é **fundamental** passar um **segundo parâmetro** para esclarecer o tipo de codificação.

```
{}
let sitcoms = fs.readFileSync('sitcoms.txt', 'utf-8');
```

“

Se estamos lendo um arquivo **JSON**, tem de converter essa string em um **objeto literal** para poder manipular os dados, usando o método **JSON.parse()**.

”

JSON.parse()

Passa os dados em formato JSON a um objeto literal para poder ser manipulado com JavaScript.

```
{}
let users = fs.readFileSync('users.json', 'utf-8');
let usersJson = JSON.parse(users);
```

2 | writeFileSync()

.writeFileSync()

É um método fornecido pelo pacote do sistema de arquivos nativo que nos permite escrever arquivos. Ela recebe dois parâmetros:

- O primeiro, **o arquivo** onde queremos escrever
- O segundo, **o conteúdo** que queremos escrever

Algo importante a ter em conta é que, se passarmos a ele o nome de um arquivo que ainda não existe, o mesmo método se encarregará de criá-lo.

Outro aspecto importante a ser observado é que este método **elimina todo o conteúdo existente** e o substitui por um novo conteúdo.

```
{}
const fs = require('fs');
fs.writeFileSync('estreias-2020.txt','Titanic 2');
```

Preparando o conteúdo

Os **métodos** de **escrita** de arquivo fornecidos pelo **file system** só podem receber conteúdo de string.

Se o que queremos salvar é outro tipo de dados JavaScript, podemos usar o formato JSON. Para convertê-lo, usamos o método `JSON.stringify()`.

```
{}
const fs = require('fs');
let filme = {
    titulo: 'Titanic',
    minutos: 560
};
let filmeJson = JSON.stringify(filme);
fs.writeFileSync('titanic.json', filmeJson);
```

3 appendFileSync()

.appendFileSync()

É um método que traz o pacote nativo **file system** que nos permite agregar conteúdo a arquivos. Recebe dois parâmetros:

- O primeiro, o **arquivo** onde queremos escrever.
- O segundo, o **conteúdo** que queremos escrever.

Como com **writeFileSync()**, se passamos o nome de um arquivo que ainda não existe, o método se encargará de criá-lo.

A diferença de **writeFileSync()**, é que este método acrescenta conteúdo ao final do conteúdo existente sem excluí-lo.

```
{}
const fs = require('fs');
fs.apppendFileSync('estreias-2021.txt','Titanic 3');
```

Multer

DigitalHouse>

“

O **multer** é um middleware que nos ajuda a simplificar o upload de arquivos para o servidor.

”

Passo a Passo

```
npm install multer
```

```html
<form action="/register" method="POST" enctype="multipart/form-data">
  <input type="file" name="avatarFile" id="file"  multiple>
  <button type="submit">Enviar</button>
</form>
```

Passo a Passo

```
const multer = require('multer');
var storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, '/uploads')
  },
  filename: function (req, file, cb) {
    cb(null, file.fieldname + '-' + Date.now())
  }
})
var upload = multer({ storage: storage })
app.post('/register', upload.single('avatarFile'), (req, res) => {
  console.log(req.file) // Nos devolve um objeto com a informação do arquivo
  res.send('O arquivo foi subido corretamente')
})
```

Tratando os erros

```
var multer = require('multer')
var upload = multer().single('avatarFile')
app.post("/register", (req, res) => {
  upload(req, res, (err) => {
   if(err) {
     res.status(400).send("Algo deu errado!");
   }
   res.send(req.file);
 });
});
```


DigitalHouse>

Validar o carregamento do arquivo

```
var upload = multer({ storage: storage })
app.post('/register', upload.single('avatarFile'), (req, res, next) => {
  const file = req.file
  if (!file) {
    const error = new Error('Por favor selecione um arquivo')
    error.httpStatusCode = 400
    return next(error)
  }
    res.send(file)
})
```

Introdução ao Multer

DigitalHouse>

“O **multer** é um middleware nos ajuda a simplificar o envio (upload) de arquivos para o servidor.”

Primeiros Passos

Para entender como vamos manipular o upload de arquivos para o servidor, vamos dividir o problema em diferentes passos. A primeira coisa que devemos fazer é instalar o **multer**, ou seja, devemos executar no projeto:

```
>_ npm install multer
```

Depois, devemos entender que configuração temos de ter do lado do cliente e o que devemos codificar do lado do servidor para poder processar os arquivos que faremos upload. Do lado do servidor, esta parte será simplificada pelo **Multer.**

Para agora, vejamos do lado do cliente o que precisamos ter.

multipart/form-data

Quando precisamos subir um arquivo para o servidor, no nosso arquivo html, devemos criar um formulário com os seguintes atributos:

method="POST"

POST, define o método HTTP através do qual será enviada a informação

multipart/form-data

Devemos especificá-lo para indicar que serão enviados arquivos e não só texto.

```html
<form action="/register" method="POST" enctype="multipart/form-data">
  <!-- campos do formulário -->
</form>
```

input type="file"

Além disso, no formulário, devemos adicionar um input do tipo file `<input type="file">`. Quando o formulário é enviado, este arquivo selecionado é enviado para o servidor, junto com seu nome e tipo.

input="file"

Nos permite selecionar um arquivo para enviá-lo ao servidor.

multiple

Caso seja preciso enviar mais de um arquivo de uma única vez.

```html
<form action="/register" method="POST" enctype="multipart/form-data">
  <input type="file" name="avatarFile" id="file"  multiple>
  <button type="submit">Enviar</button>
</form>
```

Para continuar investigando

Até aqui, temos visto do lado do cliente o que precisamos para subir arquivos no servidor. Posteriormente, veremos como, com a ajuda do **Multer**, podemos processar esse arquivo, mas do lado do servidor.

Além do **Multer**, existem outras bibliotecas que nos permitem fazer o mesmo, mas escolhemos mostrar o Multer por ser fácil de instalar e de usar.

Recomendamos que você continue investigando sobre **Multer**.

Subindo arquivos com multer

DigitalHouse>

“Para conseguir fazer upload de arquivos em nosso servidor, será necessário realizar alguns passos.”

Como funciona o Multer?

Até agora, temos o **Multer** instalado e também criamos o **HTML** para fazer o upload dos nossos arquivos. Vamos ver o que precisamos no lado do servidor:

require 'multer'

Importamos o multer para podemos usar seus recursos.

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

Como funciona Multer?

Multer oferece a opção de salvarmos arquivos em disco, para isso iremos usar o método `.diskStorage()`. Esse método recebe como parâmetro um **objeto literal** com duas propriedades: `destination` e `filename`.

```
{}
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, '/uploads')
  },
  filename: function (req, file, cb) {
    cb(null, file.fieldname + '-' + Date.now())
  }
})

const upload = multer({ storage: storage })
```

.diskStorage()

destination

Permite definir a pasta onde iremos armazenar os arquivos.

```
{}
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    //...
  },
  filename: function (req, file, cb) {
    //...
  }
})
```

filename

Permite indicar qual nome o arquivo deverá ter em nosso servidor.

destination

Na propriedade destination, iremos usar uma **callback** (**cb**) para definir qual pasta iremos salvar os arquivos. O primeiro parâmetro deverá ser `null`, e o segundo, o caminho da pasta em nosso servidor. Caso não seja informado o destino, o servidor irá utilizar a pasta de arquivos temporários, geralmente chamada /temp.

```
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'public/img/uploads')
  },
  filename: function (req, file, cb) {
    cb(null, file.fieldname + '-' + Date.now())
  }
})
const upload = multer({ storage: storage })
```

filename

Na propriedade filename, usaremos a **callback** (**cb**) para definir com qual nome iremos salvar os nossos arquivos. O primeiro parâmetro será `null`, o segundo, o nome do arquivo. Por exemplo, podemos usar a variável `file` em conjunto com a `path` para criar o nome do nosso arquivo, junto com o método `extname()` passando como parâmetro o nome original do arquivo.

```
{}
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'public/img/uploads')
  },
  filename: function (req, file, cb) {
   cb(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname))
  }
})
const upload = multer({ storage: storage })
```

Configurando a rota

Já vimos como indicar onde o arquivo que estamos enviando para o servidor será armazenado e como será chamado. Agora vamos ver como configuramos a rota que tratará a solicitação de upload de arquivo, ou seja, a rota que indicamos em nosso **action** do formulário.

Importante: Devemos implementar todo o código de configuração em cada arquivo, onde existe um caminho que está implementando o upload do arquivo.

```
{}
const upload = multer({ storage: storage });
app.post('/register', upload.single('avatarFile'), (req, res) => {
  console.log(req.file) // Irá devolver um objeto com as informações do arquivo
  res.send('Upload feito com sucesso!')
})
```

Subindo um arquivo

Para indicar que vamos subir um arquivo, usamos `.single("name")`, onde name deve ser o mesmo nome do atributo name do input do formulário.

```
{}
const upload = multer({ storage: storage });
app.post('/register', upload.single('avatarFile'), (req, res) => {
  console.log(req.file) // Irá devolver um objeto com as informações do arquivo
  res.send('Upload feito com sucesso!')
})
```

Subindo múltiplos arquivos

Para indicar que vamos carregar mais de um arquivo, usamos `.array("name")`, onde o `name` deve corresponder ao atributo name de input do formulário, sem esquecer que o input deve ter a propriedade `multiple`.

html

```html
<input type="file" name="avatarFiles" id="file"  multiple>
```

{}

```
var upload = multer({ storage: storage });
app.post('/register', upload.array('avatarFiles'), function (req, res, next) {
  console.log(req.files)
  res.send('Upload  de arquivos feito com sucesso')
})
```

“Na documentação oficial da biblioteca, podemos encontrar toda essa configuração já definida para copiar e incorporar ao nosso projeto.”

Validando arquivos com Multer

DigitalHouse>

“

Quando **subimos arquivos** para um servidor, é importante **validá-los** antes de permitir que os mesmos sejam armazenados.

”

Tratando os erros

Quando ocorre um erro, o Multer delega esse erro ao Express, e podemos mostrá-lo ao usuário da seguinte maneira:

```
var multer = require('multer')
var upload = multer().single('avatarFile')
app.post("/register", (req, res) => {
  upload(req, res, (err) => {
   if(err) {
     res.status(400).send("Algo deu errado!");
   }
   res.send(req.file);
 });
});
```

Validar o *upload* de arquivo

Para saber se um arquivo foi enviado ou não, podemos analisar o objeto `req.file`:

```
var upload = multer({ storage: storage })
app.post('/register', upload.single('avatarFile'), (req, res, next) => {
  const file = req.file
  if (!file) {
    const error = new Error('Por favor selecione um arquivo')
    error.httpStatusCode = 400
    return next(error)
  }
    res.send(file)
})
```

Introdução a Middlewares

DigitalHouse>

1

"Express trata o **request** e o **response** como *objetos*.

Recebe um pedido, o *processa* e *devolve* um **objeto** como **resposta**."

2

“Todo pedido exige uma série de passos para ser processado. O **Express** divide essas responsabilidades em **funções** denominadas **middlewares**.”

3

“Os middlewares são **funções** executadas entre a *requisição* do cliente e a *resposta* do mesmo.”

4

"A rotas que precisam de middlewares - *além do request e do response* - devem receber um terceiro parâmetro: a função **next**."

5

"A função **next** permite a execução em cadeia de *todas as funções necessárias* que devem ser executadas **antes** de enviar a resposta ao cliente."

6

“Com os *middlewares*, podemos **isolar** código que nos permita resolver um cenário particular e **implementá-lo** onde seja necessário, evitando assim *repetir código*.”

7

“Essas funções adicionam um nível de **segurança**. Se a função que definimos como middleware *não retorna* o esperado, ele **interrompe** a execução em cadeia e devolverá a resposta definida para esses casos.”

Como funciona

REQUEST

APP

Um usuário faz um pedido e **solicita** acesso à view de edição de perfil.

Como funciona

Entre o pedido e a resposta tem um **middleware**.

Como funciona

Ele se encarrega de **verificar** que o usuário está **logado**

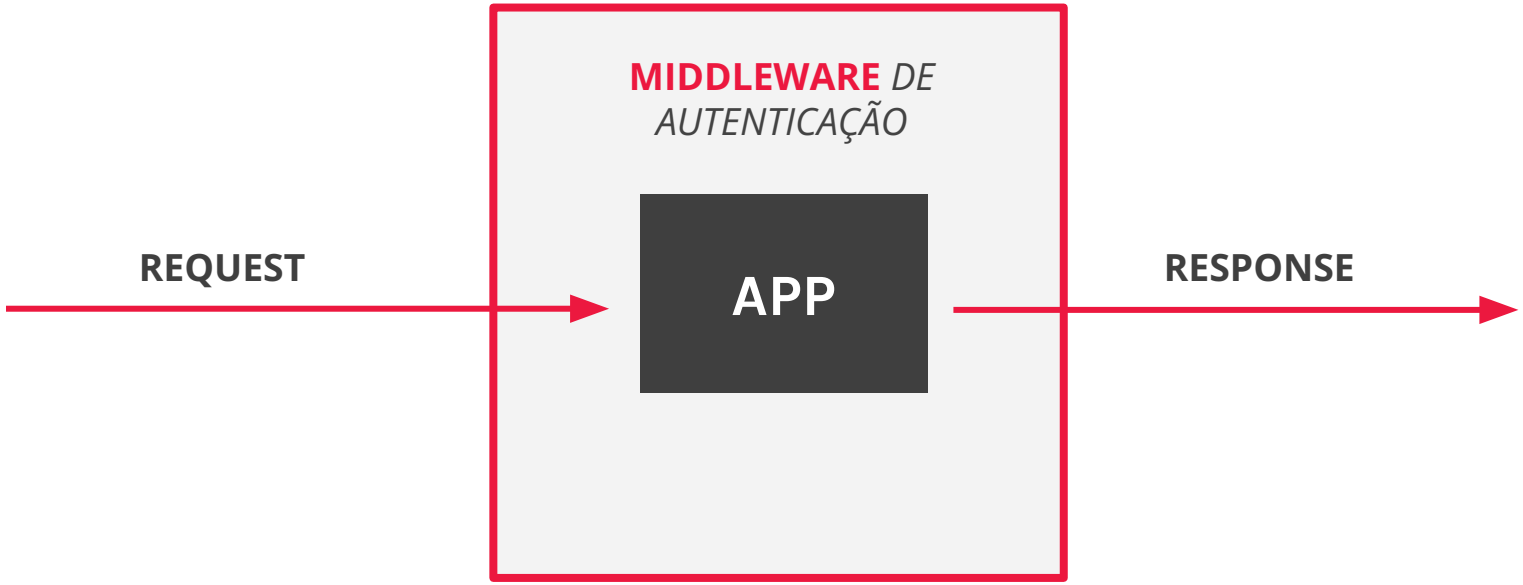
Como funciona

Se **estiver logado**, será executada a tarefa seguinte, que neste caso é **permitir** o **acesso** à view de edição de perfil.

Como funciona

Se **não estiver logado**, a execução é interrompida, e o middleware devolve a resposta que esteja definida para este caso.

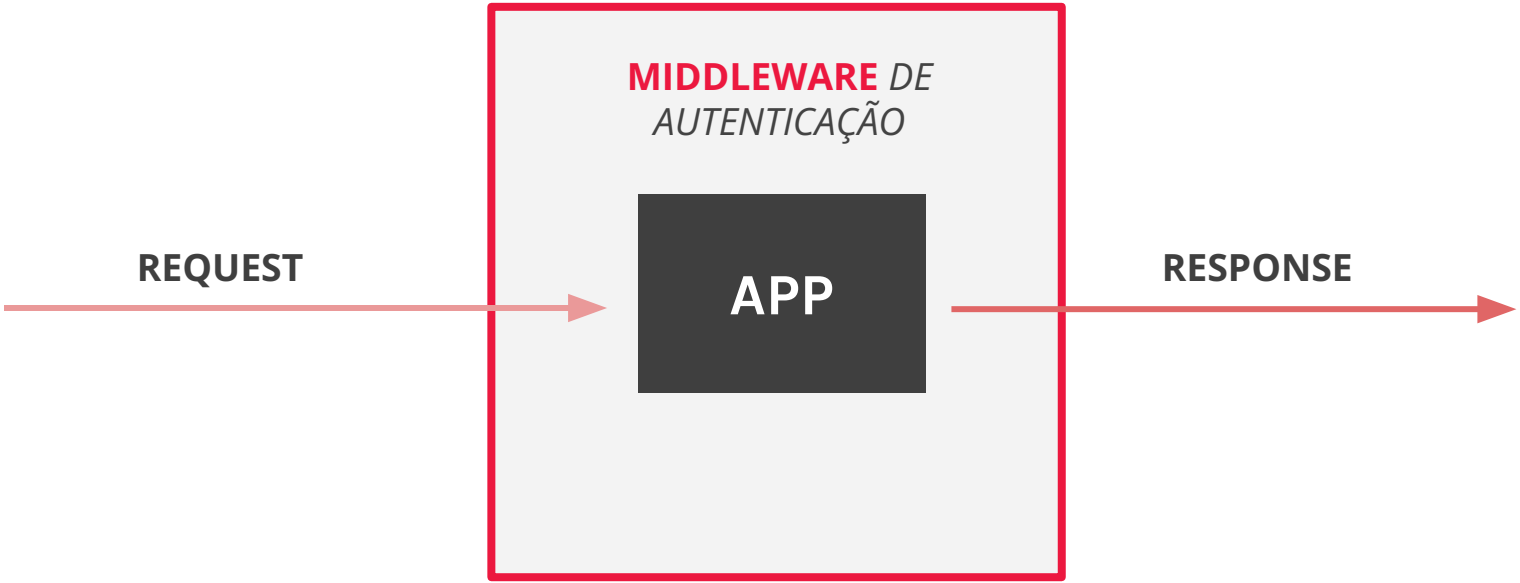
Como funciona



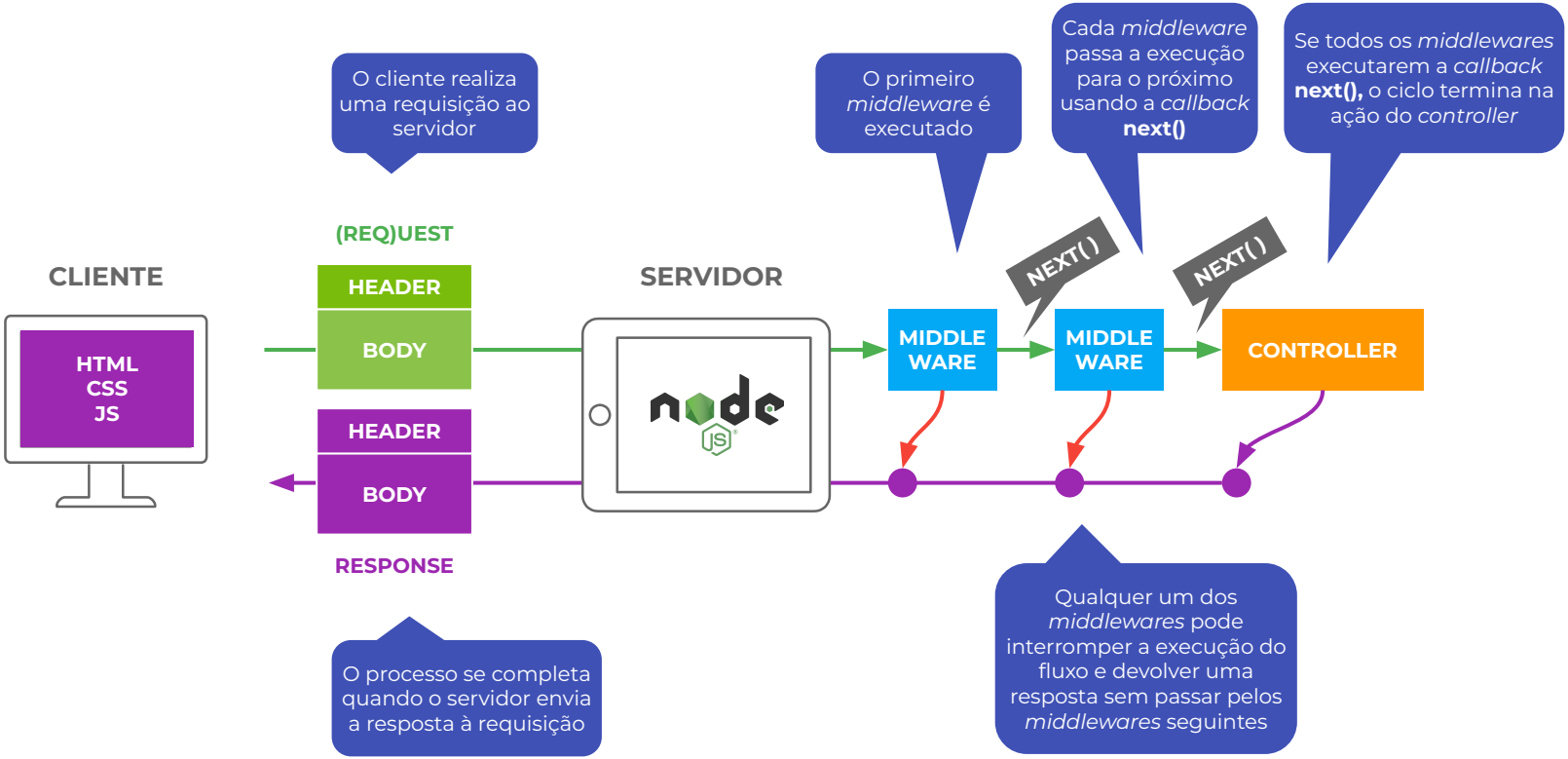
Middlewares

DigitalHouse>

Como funciona



O *request*, o *response* e os *middlewares*



O que é next()

`next` é uma *callback* que se encarregará de empilhar todos os *middlewares* que se aplicam ao mesmo pedido e irá **executá-los** um após o outro.

Quando chegar ao último *middleware*, e se foram executados corretamente, irá para o próximo passo, que é chegar ao *controller* que trata essa rota.

É por isso que sempre no final de cada *middleware* executamos ***next.***

```
{}
app.use(function(req, res, next) {
    ...
    next();
})
```

{ código }

```
app.use((req,res, next) => {
    res.status(404).render('404-page');
    next();
});
```

Alguns dos *middlewares* no **nível da aplicação** que já vínhamos utilizando são:

```
{}
 // configuração de pasta de arquivos estáticos
 app.use(express.static(__dirname + '/public'));

// configuração de roteamento
const rotasProdutos = require('./routes/products');
app.use('/', rotasProdutos);
```

Middleware no nível de rotas

Quando utilizamos um sistema de roteamento, ou rotas definidas diretamente na execução do Express, para aplicar o *middleware* a eles devemos passar uma *callback* para a rota entre o *request* e o *response*.

```
{}  router.get('/usuario/:id, callback, usersController.list);
```

O que é essa *callback*?

É o nosso *middleware*. É a funcionalidade que queremos implementar antes de que se execute o *response* da solicitação.

Instalando Express Validator

A primeira ação que devemos fazer **é instalar o pacote** em nossa aplicação, quer dizer, dentro da pasta onde temos o projeto.

Para isso executamos o seguinte comando:

```
>_ npm i express-validator
```

Tipos de validações

Existem vários tipos de validações, vamos ver as mais comuns:

```
{}
check('campo')
    .notEmpty() // Verifica se o campo não está vazio
    .isLength({ min: 5, max: 10 }) // Verifica o comprimento dos dados
    .isEmail() // Verifica se é um email válido
    .isInt() // Verifica se é um número inteiro
```

Como podem ver, podemos ter mais de uma validação para o mesmo campo. Quando é assim, simplesmente executamos um método seguido do outro.

A lista completa de validações pode ser visto aqui 📌

Mensagens de erro

```
{}
check('name')
   .notEmpty().withMessage('Deve completar o nome')
   .isLength({ min: 5 }).withMessage('O nome deve ter pelo menos 5
caracteres')
```

Cortando a cadeia de validação

```
{}
check('email')
   .notEmpty().withMessage('Deve completar o email').bail()
    // Caso a primeira validação falhe,
    // As seguintes não se executam para esse campo.
   .isEmail().withMessage('Deve completar com um email válido')
```

Agregando as validações nas rotas

Terminadas nossas validações é hora de implementá-las. Para isso, as adicionaremos à rota que processa o formulário sendo validado.

Este *middleware* se localiza entre a rota e a ação do *controller*.

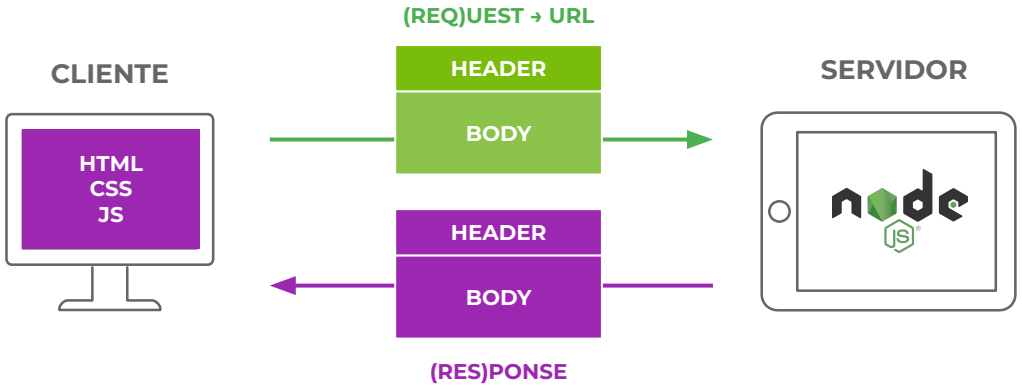
```
{}
const validateRegister = [ ... ];

// Processamento do formulário de criação
router.post('/', validateRegister, userController.processRegister);
```

Middlewares

DigitalHouse>

Request e response



REQ.PARAMS → URL

REQ.QUERY → QUERY STRING

REQ.BODY → FORMULÁRIOS

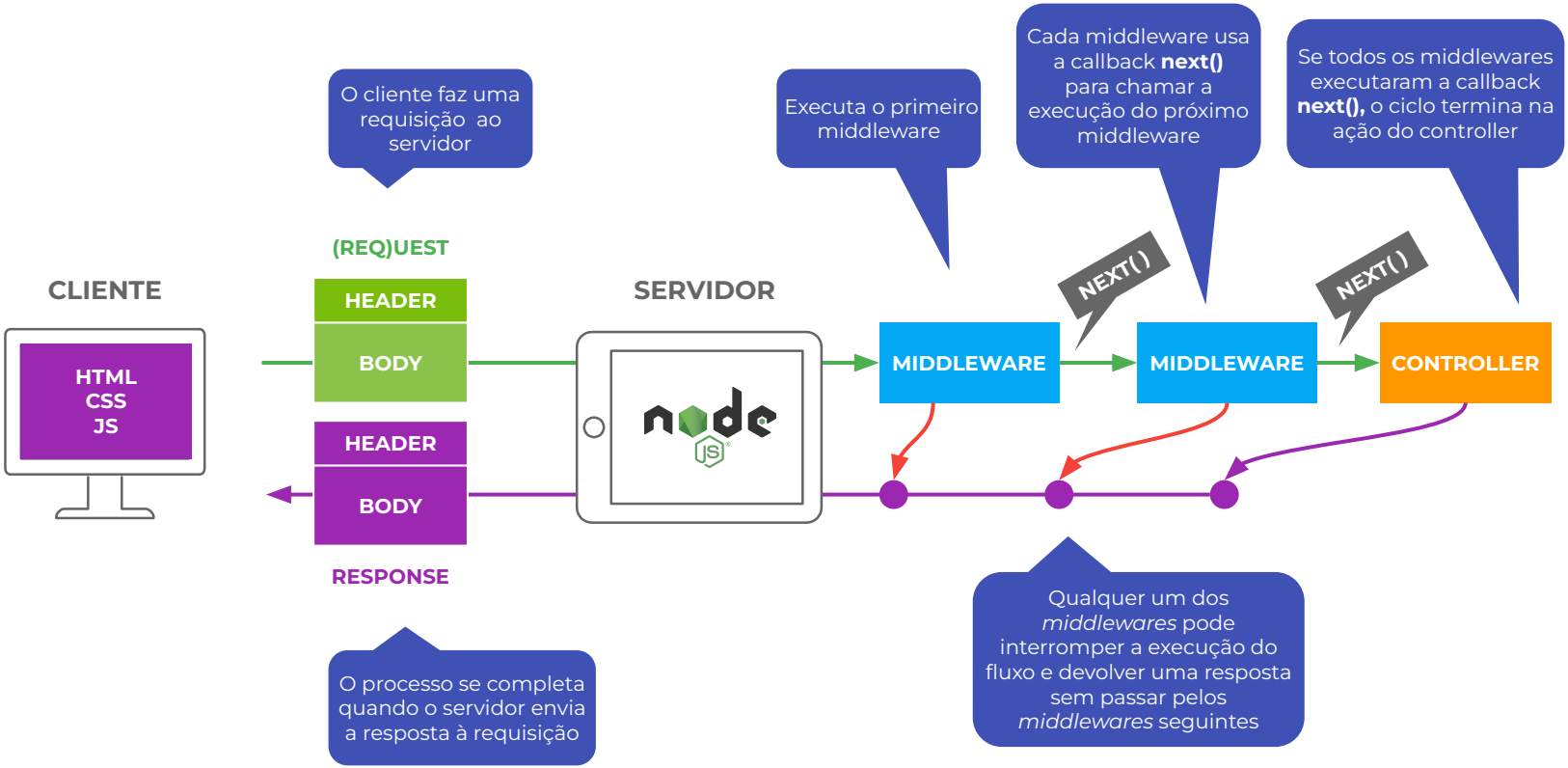
RES.SEND()

RES.SENDFILE()

RES.RENDER()

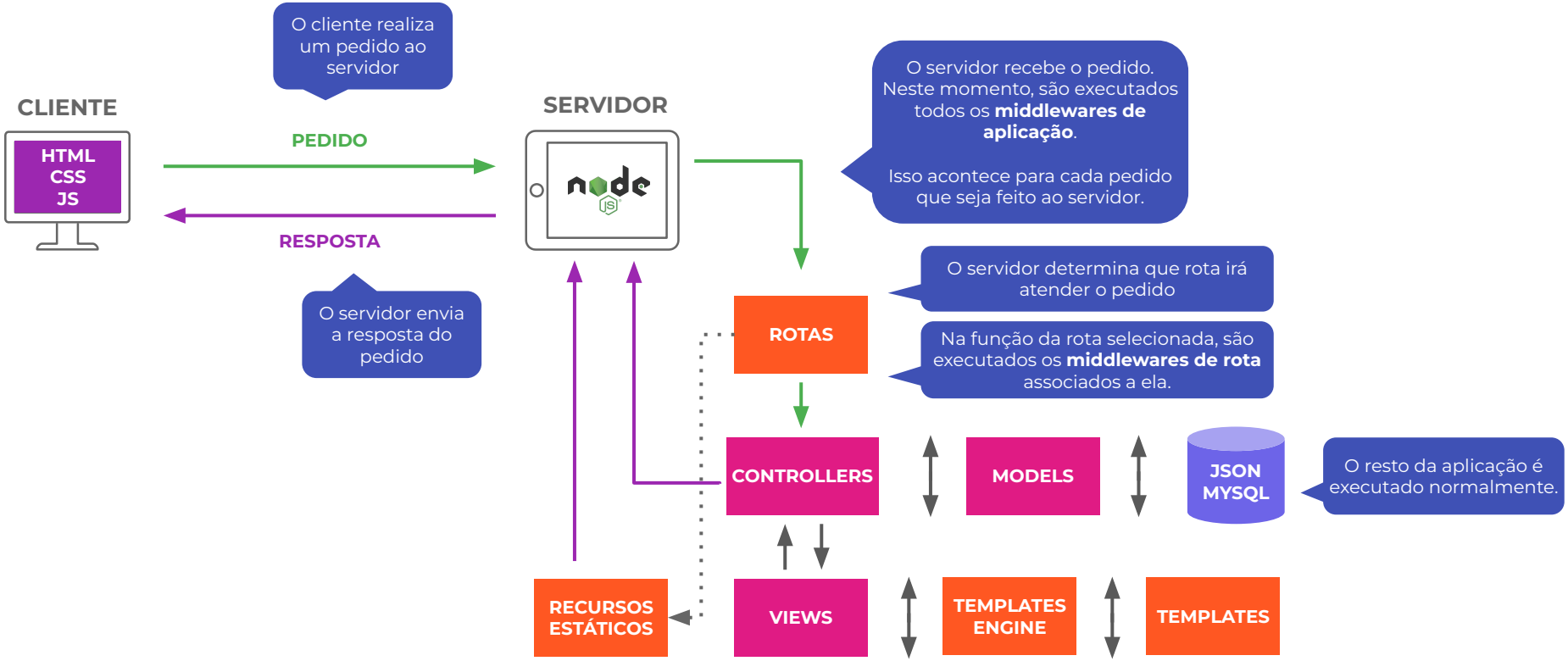
RES.REDIRECT()

Request, response e middlewares



DigitalHouse>

Middlewares e MVC no Node



DigitalHouse>

Express Validator Introdução

DigitalHouse>

“O **Express Validator** permite-nos tratar, de forma relativamente simples, todas as validações dos nossos formulários no back-end.”

Express Validator passo a passo

O Express Validator nos permitirá validar se as informações que chegam dos formulários são as que esperamos.

As etapas que devemos seguir para implementá-lo são:

1. **Instale o Express Validator.**
2. **Crie uma array com as validações de cada formulário.**
3. **Adicione-o como middleware da rota que processa o formulário.**
4. **Verifique se houve erros na validação do controlador.**
5. **Envie os erros para a view.**

Então, vamos ver as **três primeiras etapas!**

Instalando o Express Validator

A primeira ação que devemos tomar é instalar o pacote em nossa aplicação, ou seja, dentro da pasta onde está o projeto.

Para fazer isso, executamos o seguinte comando:

```
>_ npm i express-validator
```

Nosso formulário

Antes de começar com as validações, é importante considerar como nossos formulários são montados. Ao escrever as validações, tomaremos como referência a propriedade **name** de cada campo.

```html
<form action="/registro" method="post">
    <label for="name">Nome:</label>
    <input type="text" name="name" id="name">

    <label for="email">Email:</label>
    <input type="email" name="email" id="email">

    <label for="password">Senha:</label>
    <input type="password" name="password" id="password">

    <button type="submit">Cadastrar</button>
</form>
```

As validações

Agora que temos o módulo instalado, vamos solicitá-lo (importá-lo) onde faremos as validações. Podemos fazer isso diretamente no arquivo de rotas ou criar nossas validações em um arquivo separado.

Em ambos os casos, o primeiro passo será solicitar o módulo e, usando a desestruturação, solicitar o método **check**.

```
{}  const { check } = require('express-validator');
```

O segundo passo será criar uma variável onde armazenaremos o conjunto de validações que realizaremos no formulário.

```
{}  let validateRegister = [] // validações aqui
```

O método check()

O método **check()** nos permite adicionar validações para qualquer um dos campos do formulário. Como parâmetro, recebe o nome do campo a validar. Se, por exemplo, quisermos validar o campo **name**, o método ficaria assim:

```
{}  const validateRegister = [ check('name') ]
```

Assumindo que queremos validar se o campo não está vazio, no método anterior executamos o método **notEmpty()** da seguinte maneira:

```
{}  const validateRegister = [
        check('name').notEmpty()
    ]
```

Tipos de validações

Existem vários tipos de validações. Vejamos as mais comuns:

```
{}
check('campo')
    .notEmpty() // Verifica se o campo no está vazio
    .isLength({ min: 5, max: 10 }) // Verifica o tamanho dos dados
    .isEmail() // Verifica se é um email válido
    .isInt() // Verifica se é um número inteiro
```

Como você pode ver, podemos ter mais de uma validação para o mesmo campo. Se for esse o caso, apenas executamos um método seguido do outro.

A lista completa de validações pode ser vista aqui 📌

Mensagens de erro

Além das validações, o Express Validator permite definir a mensagem que o usuário receberá a cada validação que falhar.

Para implementar as mensagens, usamos o método **withMessage()** após cada validação.

```
{}
check('name')
    .notEmpty().withMessage('Deve preencher o nome')
    .isLength({ min: 5 }).withMessage('O nome deve ter pelo menos 5
caracteres')
```

Interrompendo a cadeia de validação

Em alguns casos, desejaremos interromper a validação, pois se, por exemplo, um campo estiver vazio, não faz sentido verificar se é um email válido.

Se não cortarmos a validação, o usuário obterá todos os erros juntos, em vez de apenas aquele que se aplica.

Para esses casos, podemos implementar o método **bail()**.

```
{}
check('email')
   .notEmpty().withMessage('Deve preencher o email').bail()
    // No caso da primeira validação falhar, as validações
    // seguintes não serão executadas para este campo.
   .isEmail().withMessage('Deve preencher com um email válido')
```

A array de validações completa

Em resumo, quando terminarmos de escrever nossas validações, teremos uma *array*, com um elemento por campo, com todas as suas validações.

```
const validateRegister = [
    check('name')
        .notEmpty().withMessage('Deve preencher o nome').bail()
        .isLength({ min: 5 }).withMessage('O nome deve ser maior'),
    check('email')
       .notEmpty().withMessage('Deve preencher o email').bail()
       .isEmail().withMessage('Deve preencher com um email válido'),
    check('password')
       .notEmpty().withMessage('Deve preencher a senha').bail()
       .isLength({ min: 8 }).withMessage('A senha deve ser maior')
]
```

Adicionando as validações às rotas

Uma vez finalizadas nossas validações, é hora de implementá-las. Para isso, vamos adicioná-las ao caminho que processa o formulário que queremos validar.

Este *middleware* está localizado entre a rota e a ação do controlador.

```
{}
const validateRegister = [ ... ];

// Processamento do formulário de registro
router.post('/', validateRegister, userController.processRegister);
```

Express Validator Avançado

DigitalHouse>

“O **Express Validator** nos permite gerar mensagens para cada erro de validação (que pode ser enviado para a visualização do usuário).”

Express Validator passo a passo

As primeiras etapas garantiram que ocorresse o processo de validação do formulário. Agora temos que trabalhar com o **resultado dessa validação.**

1. **Instale o Express Validator.**
2. **Crie um array com as validações de cada formulário.**
3. **Adicione-o como middleware da rota que processa o formulário.**
4. **Verifique se houve erros na validação do controlador.**
5. **Envie os erros para visualização.**

Então, vamos ver as **duas últimas etapas!**

As validações no controller

Novamente, a primeira etapa será importar o módulo e, usando a desestruturação, solicitar o método **validationResult**.

```
{}  const { validationResult } = require('express-validator');
```

A segunda etapa ocorrerá dentro do método do *controller* responsável pelo processamento do formulário. Lá salvaremos, na variável *errors*, a execução do método validationResult, passando o objeto *request* como parâmetro.

```
{}  let errors = validationResult(req);
```

O método isEmpty()

O método **isEmpty()** nos permite saber se há erros de validação ou não.

Se não houver erros, podemos prosseguir com segurança com a ação a ser executada por esse controlador, por exemplo, criar um novo usuário.

Caso contrário, retornaremos ao formulário com os erros para o usuário.

```
{}
register: (req, res) => {
    let errors = validationResult(req);

    if (errors.isEmpty()) {
        // Não existem erros, seguimos em frente
    } else {
        // Existem erros, voltaremos ao formulário com as mensagens
    }
},
```

O método mapped()

O método **mapped()** nos permite enviar os ***erros para visualização como um objeto***. Esse objeto conterá uma propriedade com o primeiro erro em cada campo.

Para enviar os erros para a visualização, simplesmente os adicionamos como um segundo parâmetro ao método **render()**.

É importante enviar também o conteúdo de *req.body*, pois queremos preservar os dados preenchidos pelo usuário ao retornar o formulário.

```
{}
if (errors.isEmpty()) {
    // Não existem erros, seguimos em frente
} else {
    // Existem erros, voltaremos ao formulário com as mensagens
    res.render('register', { errors: errors.mapped(), old: req.body });
}
```

O objeto de erros

Para cada campo com erro, o objeto terá uma propriedade, cujo nome será igual ao atributo *name* do campo.

Cada erro conterá: a mensagem (**msg**), o nome do campo (**param**), o valor inserido pelo usuário (**valor**) e de onde veio (**body** para formulários).

```
{}
{
    email: {
        msg: 'Deve preencher um email válido',
        param: 'email',
        value: 'umEmail',
        location: 'body'
    },
    password: {
        msg: 'A senha deve ser maior',
        param: 'password',
        value: '1234',
        location: 'body'
    }
}
```

Mostrando os erros na view

Usando EJS, podemos perguntar se um determinado campo contém erros. Se for esse o caso, podemos exibir a mensagem de erro.

É importante observar que, na primeira vez que o formulário for carregado, não haverá erros e, portanto, essa variável estará vazia. Para evitar problemas, devemos sempre perguntar se a variável de erro existe antes de tentar exibir um erro.

O código pode ser assim:

```
{}
<label for="email">Email:</label>
<input type="email" name="email" id="email">
<% if (locals.errors && errors.name) { %>
    <p class="feedback"><%= errors.name %></p>
<% } %>
```

Inserindo os dados anteriores no formulário

Outro ponto importante é que, se o usuário já preencheu o formulário, mas colocou informações inválidas em algum campo, não queremos que ele complete tudo novamente.

Por esse motivo, na etapa anterior, reenviamos os dados do formulário original no objeto antigo.

Novamente com EJS, podemos carregar esse valor em cada campo correspondente.

```
{}
<label for="email">Correo electrónico:</label>
<input type="email" name="email" id="email"
       value="<%= locals.old && old.email %>">
```

Criação do login

DigitalHouse >
Coding School

Implementar *session*

- Instalar o módulo express-session com npm:

```
>_ npm i express-session
```

- Importá-lo no *entry point* da aplicação:

```
{} const session = require('express-session');
```

- Configurá-lo como *middleware* em nível aplicação. Executamos `session()` passando como argumento um objeto literal com a propriedade `secret` com um texto único aleatório, que servirá para identificar nosso web-site.

```
{} app.use(session( {secret: "Nossa mensagem secreta"}));
```


DigitalHouse> Coding School

Implementar *session*

- Quando queremos **definir** e **armazenar** informações, chamamos a propriedade *session* do objeto *request (req)*:

```
req.session.corDeFundo = 'Violeta';
```

- Para **ler** informação de *session*:

```
let corDeFundo = req.session.corDeFundo;
```

Toda a informação que armazenamos na variável `session` estará disponível para usar em **todas** as *views* do *site*.

DigitalHouse> Coding School

Implementar *cookies*

- Instalar o módulo `cookie-parser` com npm. (No express-generator já vem incluído este módulo)

```
>_ npm i cookie-parser
```

- Para **criar** um cookie e **guardar** informação nele, executamos o método `cookie()` sobre o objeto `response` `(res)`, passando dois argumentos:
  - O **nome** que queremos dar a esse *cookie*.
  - O **valor** que terá.

```
{} res.cookie('club', 'C. A. Tigre');
```

O nome do *cookie* que definimos será uma propriedade de `cookies`.

DigitalHouse> Coding School

Lendo os *cookies*

- Para **ler** informação de um *cookie* usamos o objeto `request`, chamando o objeto `cookies`, seguido do nome do *cookie* que definimos anteriormente:

```
{}  console.log(req.cookies.club);
```


DigitalHouse> Coding School

.hashSync()

É um **método** do pacote `bcryptjs` que nos permite criptografar os dados. Recebe dois parâmetros:

- O **dado** que queremos criptografar.
- O **salt** que queremos acrescentar à criptografia.

O que é o salt?

É um pequeno dado acrescentado que faz com que as *hash* sejam significativamente mais difíceis de quebrar. Neste contexto geralmente passamos 10 ou 12 como valor.

```
{}
const bcrypt = require('bcryptjs');
let senhacriptografada = bcrypt.hashSync('doguinho123', 10);
```


DigitalHouse> Coding School

.compareSync()

É um **método** do pacote `bcryptjs` que nos permite **comparar** um texto simples com uma *hash* para saber se eles correspondem ou não.

Este método **retorna** um **booleano** e recebe dois parâmetros:

- O primeiro, **texto simples**.
- O segundo, a **hash** com o qual queremos compará-lo.

```
{}
let check = bcrypt.compareSync('doguinho123', senhaCriptografada);
console.log(check); // true
```


DigitalHouse> Coding School

Construindo login: sessão

DigitalHouse >
Coding School

“Uma **sessão** permite armazenar informações de maneira segura no servidor durante a visita do usuário ao *site*, e compartilhar essas informações com as *views*.”

Sessão em Express

Quando trabalhamos com `sessão`, armazenamos no lado do servidor os dados do usuário que são relevantes para permitir que se navegue sem problemas em nosso *site*. Podem ser informações pessoais, que servem para o *login*, ou alguma característica mais global, como idioma, moeda ou cor de fundo.

Por sua vez, no lado do **cliente**, será gerado um identificador único que associará aquele usuário a todas essas informações.

Algo importante a se ter em mente sobre a `sessão` é que, quando o usuário **fecha o navegador**, todas as informações são **excluídas**. Ou seja, os dados de uma `sessão` só existem enquanto o navegador estiver aberto.

Implementando sessão

- Instalamos o módulo `express-session` com npm:

```
>_ npm i express-session
```

- Queremos inseri-lo no *entry-point* da aplicação:

```
{} const session = require('express-session');
```

- Nós o configuramos como *middleware* no nível do aplicativo. Executamos `session()`, passando como argumento um objeto literal com a propriedade `secret` com um texto aleatório único, que servirá para identificar nosso *website*.

```
{} app.use(session( {secret: "Nossa mensagem secreta"}));
```

Manipulando os dados de sessão

- Quando queremos **definir** e **armazenar** informações, chamamos a propriedade `session` do objeto de solicitação (*req*):

```
req.session.corDeFundo = 'Violeta';
```

- Para **ler** informação de sessão:

```
let corDeFundo = req.session.corDeFundo;
```

Construindo o login: cookies

DigitalHouse >
Coding School

“ **Cookies** são **arquivos** que podemos salvar no lado do **cliente**, ou seja, no **navegador** do usuário.

Os *cookies* serão sempre associados ao site para o qual foram criados. ”

Cookies no Express

Ao contrário da sessão, os *cookies* podem ser configurados com uma "vida útil". Ou seja, um *cookie* deixará de existir quando esse tempo expirar e não quando o usuário fechar o navegador.

Justamente porque eles ficam no lado do usuário e suas informações podem ser lidas, é importante observar que não devemos armazenar nenhum dado sensível em um *cookie*.

Outro fator a ser levado em conta ao trabalhar com *cookies* é que temos um limite de espaço (4kb).

“Para manter um usuário logado após fechar seu navegador, podemos usar um ***cookie*** para identificá-lo e registrá-lo automaticamente na próxima vez em que entrar no site.”

Implementando *cookies*

- Instalamos o módulo `cookie-parser` com npm:

```
>_ npm i cookie-parser
```

- Para **criar** um cookie e **guardar** informação nele, executamos o método `cookie()` sobre o objeto *`response`*, passando dois argumentos:
  - O **nome** que queremos dar para esse cookie.
  - O **valor** que terá.

```
{} res.cookie('club', 'C. A. Tigre');
```

Lendo os *cookies*

- Para **ler** informações de um *cookie*, usamos o objeto `request`, chamando o objeto `cookies` seguido do nome do *cookie* que definimos anteriormente:

```
{}  console.log(req.cookies.club);
```

Cada vez que o cliente faz um pedido em nosso *site*, ele envia todos os *cookies* que lhe estão associados, por isso faz sentido que os encontremos no objeto de solicitação.

Criação de login: hashing

DigitalHouse >
Coding School

“Quando trabalhamos **com dados sigilosos**, é fundamental armazená-los **criptografados**, para **preservar** a informação caso um terceiro os acesse.”

O que é *hash*?

Na computação, as funções de *hashing* nos permitem **criptografar** dados. Ou seja, **transformar** um texto simples em uma nova série de caracteres - com comprimento fixo - impossível de decifrar pelo olho humano.

Neste tipo de criptografia (unilateral), os dados criptografados não podem ser descriptografados novamente.

É por isso que essas funções são acompanhadas por dois recursos principais:

- ➔ A opção de **criptografar um dado**.
- ➔ A opção de **comparar** um dado de entrada com um dado ***hash*** para verificar se ele corresponde ou não.

O pacote **bcryptjs** nos permite incorporar essas funções em nosso projeto Node.js.

Para usá-lo, você deve instalá-lo através do npm.

```
>_ npm i bcrypt
```

.hashSync()

É um **método** que o pacote bcryptjs traz que nos permite criptografar os dados. Ele recebe dois parâmetros:

- O **dado** que queremos criptografar.
- O **salt** que queremos dar para a criptografia.

O que é esse salt?

Um pequeno detalhe que torna os *hashes* significativamente mais difíceis de quebrar. Neste contexto geralmente passamos 10 ou 12 como valor.

```
{}
const bcrypt = require('bcrypt');
let senhaCriptografada = bcrypt.hashSync('doguinho123', 10);
```

.compareSync()

É um **método** que o pacote `bcryptjs` traz que nos permitirá **comparar** um texto simples com um *hash* para saber se eles correspondem ou não.

Este método **retorna** um **booleano** e recebe dois parâmetros:

- O primeiro, **texto simples**.
- O segundo, o **hash** com o qual queremos compará-lo.

```
{}
let check = bcrypt.compareSync('doguinho123', senhaCriptografada);
console.log(check); // true
```

DigitalHouse>