

# **Aplicações para a Internet II**

## **Node.JS**

**2021/2022**

# <Express/>

---

- A utilização em **Node.js** da *framework Express* permite explorar funcionalidades do **lado do servidor (back-end)**;
  - Com recurso à linguagem **JavaScript (JS)**;
- O **Express** apresenta-se como:
  - Framework do lado do **servidor**;
  - *Template engine* - dois motores de modelos, o Pug e o EJS, que facilitam o fluxo de dados numa estrutura de site e permitem utilizar outros modelos;
  - Suporte a arquitetura **MVC**;
  - Multiplataforma;
  - Entre outros.

# <Express/>

---

- Características do Express:
  - Código minimalista;
  - Integrável com motores de template;
  - Trabalha com conceito *middleware*;
  - Foco em performance;
  - Utilização de padrões e boas práticas REST e RESTFul;
  - Entre outros.

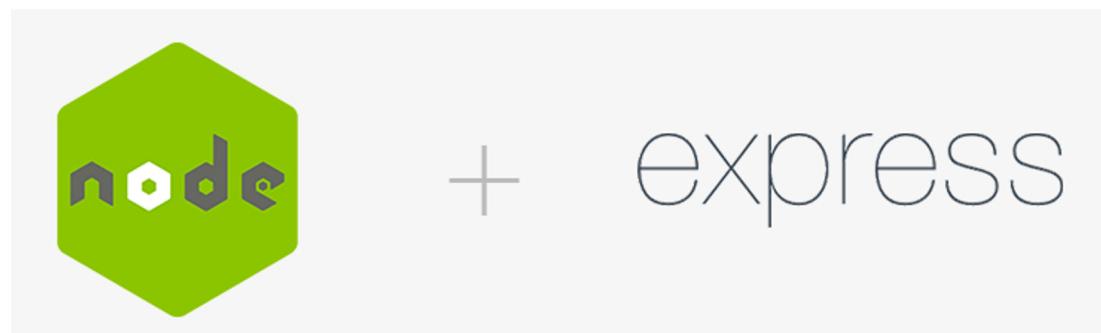


Imagen retirada do artigo <https://projectsplaza.com/create-simple-static-website-nodejsexpress-jade>.

# <Routing/>

---

- A função de uma **rota** (*routing*) no Express é o de decidir o que fazer quando determinada solicitação (*http*) chega;
  - Por exemplo, obtenção de dados de um modelo, renderização de uma view, etc.;
- A definição das rotas tem a seguinte **sintaxe**:

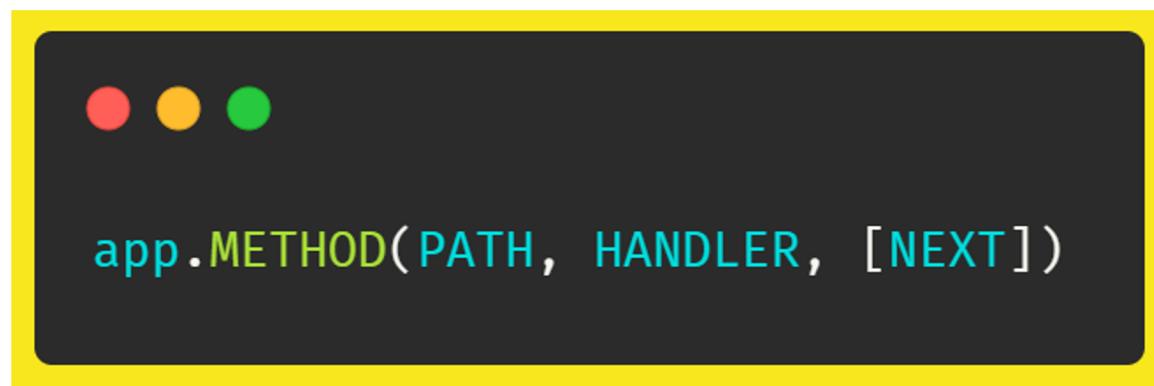


Figura 1 - **Sintaxe** de uma **rota** em Express.

# <Routing/>

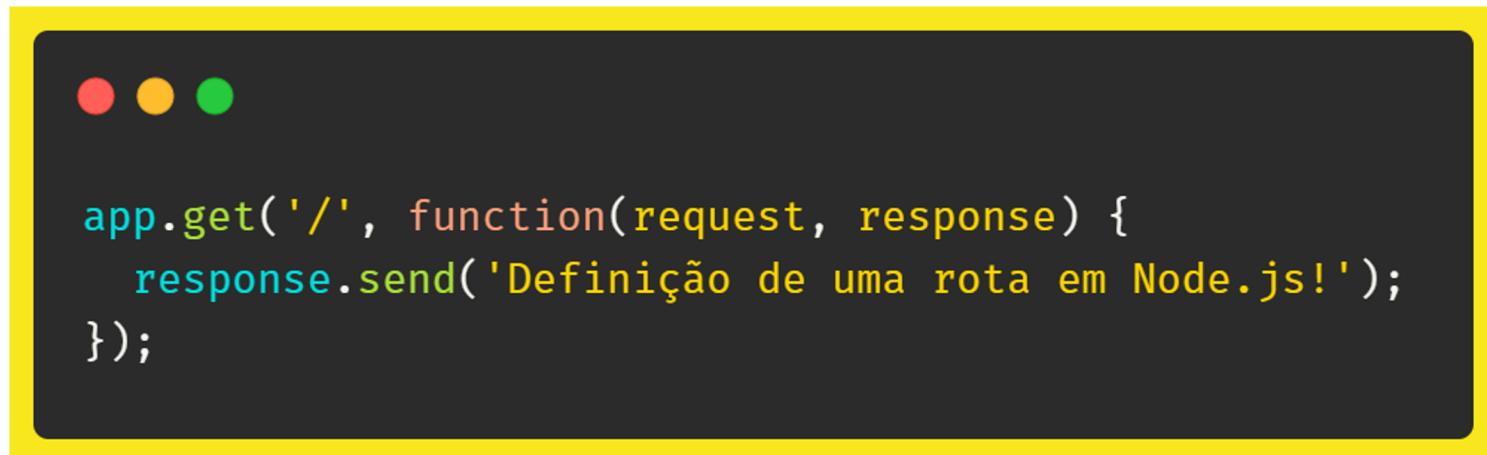
---

- Em suma:
  - *app* - é a instância do Express;
  - *METHOD* - um método de solicitação HTTP (p. ex., *get*, *post*, *put*, etc.);
  - *PATH* - um caminho (URI) no servidor;
  - *HANDLER* - função de *callback* que é executada quando a rota é correspondida, isto é, o evento ocorre;
  - *NEXT* - (*opcional*) utiliza-se apenas em operações com *middleware* porque o Express não tem maneira de saber quando é que o pedido é concluído.

# <Routing/>

---

- Exemplo de uma rota para a **raiz** (root) do projeto:



```
● ● ●

app.get('/', function(request, response) {
  response.send('Definição de uma rota em Node.js!');
});
```

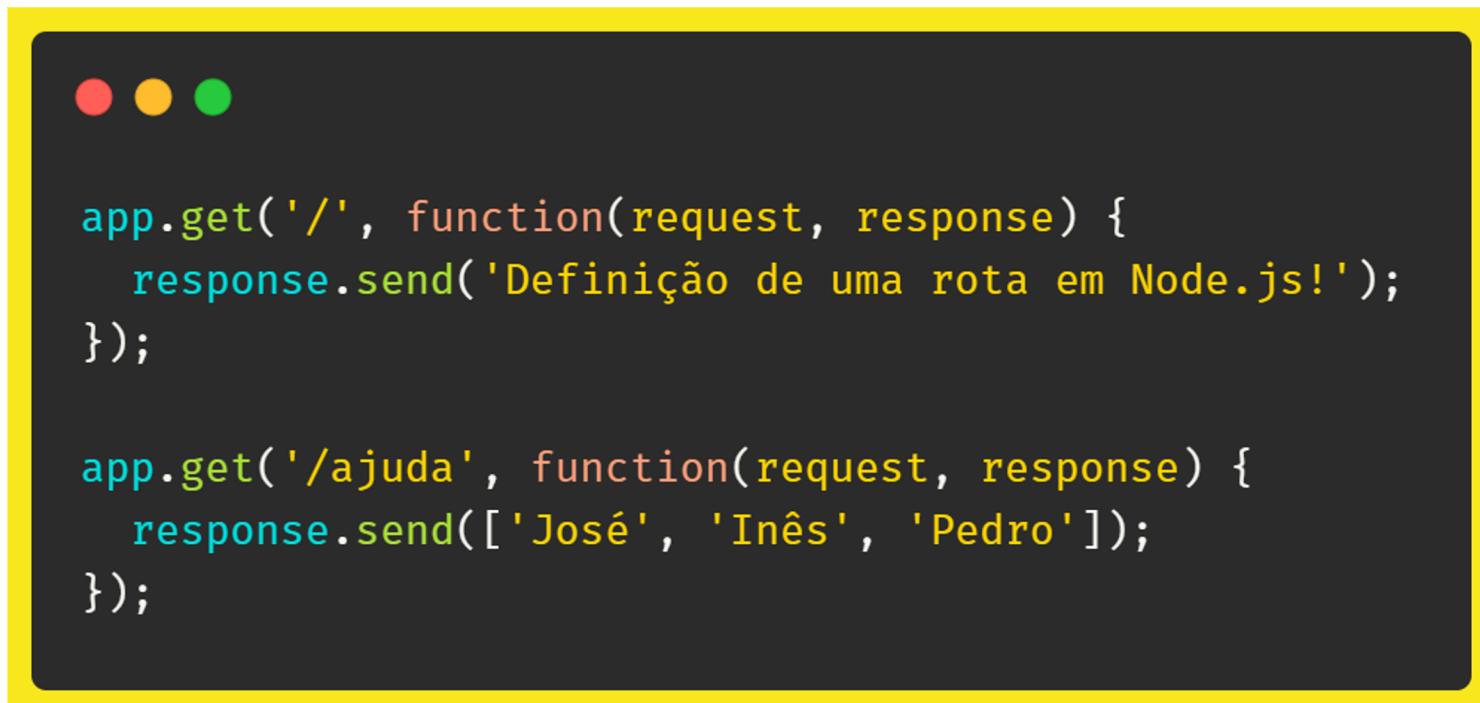
Figura 2 - Definição de uma **rota** em Express.

- O método **send()** da resposta permite enviar respostas com dados em diferentes formatos (*string, object, array, etc.*);

# <Routing/>

---

- Exemplo de mais do que uma rota numa aplicação *Express*:



The screenshot shows a code editor window with a dark theme. At the top left, there are three colored circular icons: red, yellow, and green. The main area contains the following code:

```
app.get('/', function(request, response) {
  response.send('Definição de uma rota em Node.js!');
});

app.get('/ajuda', function(request, response) {
  response.send(['José', 'Inês', 'Pedro']);
});
```

Figura 3 - Definição de várias **rotas** em Express.

# <Routing/>

---

- **Métodos de Resposta (*response*):**
  - **download()** - solicita que seja efetuado o download de um ficheiro;
  - **end()** - termina o processo de resposta;
  - **json()** - envia uma resposta JSON;
  - **jsonp()** - envia resposta JSON com suporte ao JSONP;
  - **redirect()** - redireciona uma solicitação;
  - **render()** - renderiza um modelo de visualização;
  - **send()** - envia uma resposta de vários tipos;
  - **sendFile()** - envia um ficheiro;
  - **sendStatus()** - configura o código do estado (status) de resposta e envia a sua representação em sequência de caracteres como corpo de resposta.

# <Routing/>

---

- Os métodos de resposta (*response*) podem enviar uma resposta ao cliente e finalizar o ciclo solicitação-resposta;
  - Se nenhum dos métodos listado anteriormente for invocado a partir de um manipulador de rota, a solicitação será deixada em suspenso;



```
app.get('/', function(request, response) {
  response.json({ nome: 'Pedro Silva' });
});
```

Figura 4 - Definição de uma **rota** com **resposta** no formato **JSON**.

# <Routing/>

---

- Quando se desenvolve aplicações Web é necessário obter informações (dados) das interações com os utilizadores, isto é, valores diretos, ou valores passados via formulários, etc.;
- O pedido pode ser feito de várias maneiras, sendo as mais comuns as seguintes:
  - GET - parâmetros de URL
    - Os dados são passados através da URL;
    - São recolhidos utilizando o comando  
*request.query.<nome do parâmetro>*

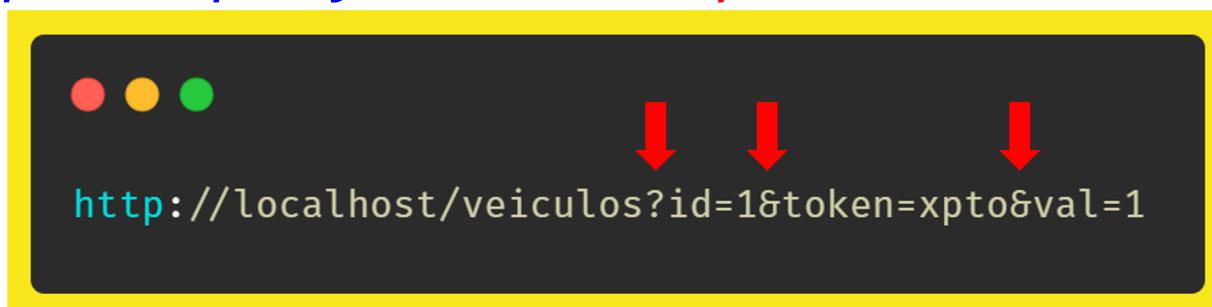


Figura 6 - Exemplo de uma URL com **parâmetros** passados via **GET**.

# <Routing/>

---

- GET - parâmetros de URL



```
app.get('/veiculo', function(request, response) {
  let car_id = request.query.id;
  let token = request.query.token;
  let valido = request.query.val;
  response.send(car_id + ' ' + token + ' ' + valido);
});
```

Figura 7 - Exemplo de uma rota com **parâmetros** (id, token, val) passados via **URL**.

# <Routing/>

---

- POST - parâmetros internos
  - Os dados provêm de formulários;
  - Os dados não são visíveis;
  - As informações são enviadas como *application/x-www-form-urlencoded*;
  - São recolhidos utilizando o comando  
*request.body.<nome do parâmetro>*;

# <Routing/>

---

- POST - parâmetros internos

```
● ● ●

var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: true }));

app.post('/veiculo', function(request, response) {
    let car_id = request.body.id;
    let token = request.body.token;
    let valido = request.body.val;
    response.send(car_id + ' ' + token + ' ' + valido);
});
```

Figura 8 - Exemplo de uma rota com parâmetros (id, token, val) passados internamente.

- Se pretendermos testar o envio dos dados com aplicações de terceiros (p. ex., POSTman, etc.), é necessário instalar um parser, como o módulo NPM ‘body-parser’;

# <Middleware/>

- Conjunto de funções que são invocadas pela camada de *routing* (rotas) do Express antes de serem manipuladas;

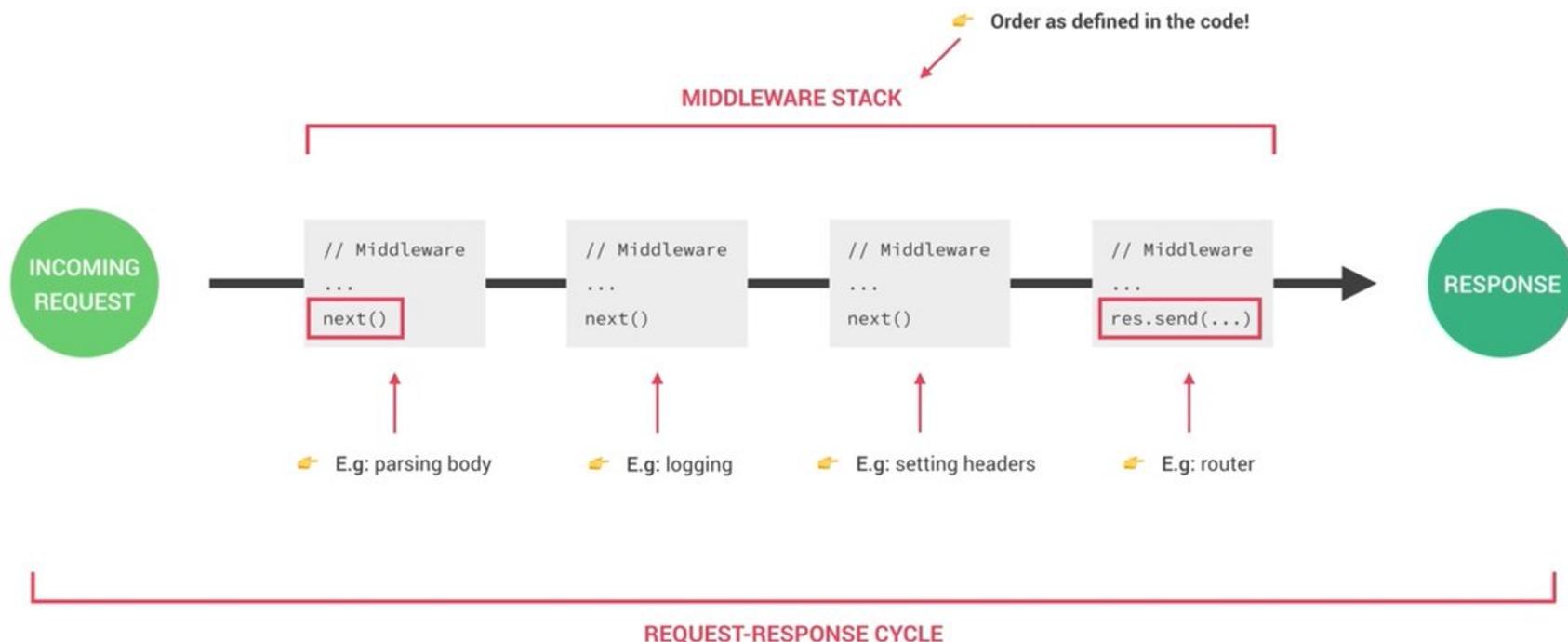


Figura 9 - Ciclo solicitação-resposta de uma *middleware*.

# <Middleware/>

---

- O *middleware* funciona como um **filtro das requisições efetuadas**;
  - No momento em que um pedido é recebido por uma aplicação do Express, este aciona várias funções referidas como *middleware*;
  - Logo os pedidos podem ser modificados antes de serem entregues ao processo seguinte;
- Permite criar aplicações mais fáceis de manter e com menos código;

# <Middleware/>

---

- Embora o comando `app.use()` seja invocado para cada método HTTP, o *middleware* é processado após receber a ordem enviada pela manipulação de rotas;
- É importante ter em atenção os seguintes aspetos:
  - **Ordem dos pedidos** - o *GET* é executado antes do *middleware* e o *POST* após o *middleware*;
  - **Esquecimento do `next()`** - se a solução não atualiza ou não responde, então o código não está a chamar o `next()`;
  - **Sobreposição de propriedades** - os argumentos `request` e `response` correspondem à mesma instância para todos os *middlewares* e rotas;

# <Middleware/>

---

- Estas alterações podem provocar erros, pelo que é importante estar-se atento às modificações que o *middleware* faz nas propriedades;



A screenshot of a terminal window with a yellow border. The window title bar has three colored dots (red, yellow, green). The terminal displays the following code:

```
app.use(function (request, response, next) {
  console.log('Data:', Date.now());
  next();
});
```

Figura 10 - Exemplificação de uma função *middleware* em Express.

# <Middleware/>

---



```
app.use(function(req, res, next) {
  console.log('Passei por aqui antes da rota.. middleware em funcionamento!');
  next();
}

app.get('/', function(req, res) {
  console.log('E agora passo aqui, após o término da middleware... ');
  res.send('Rota principal...');
});
```

Figura 11 – Exemplo de implementação de uma middleware.

# <Routing + Middleware/>

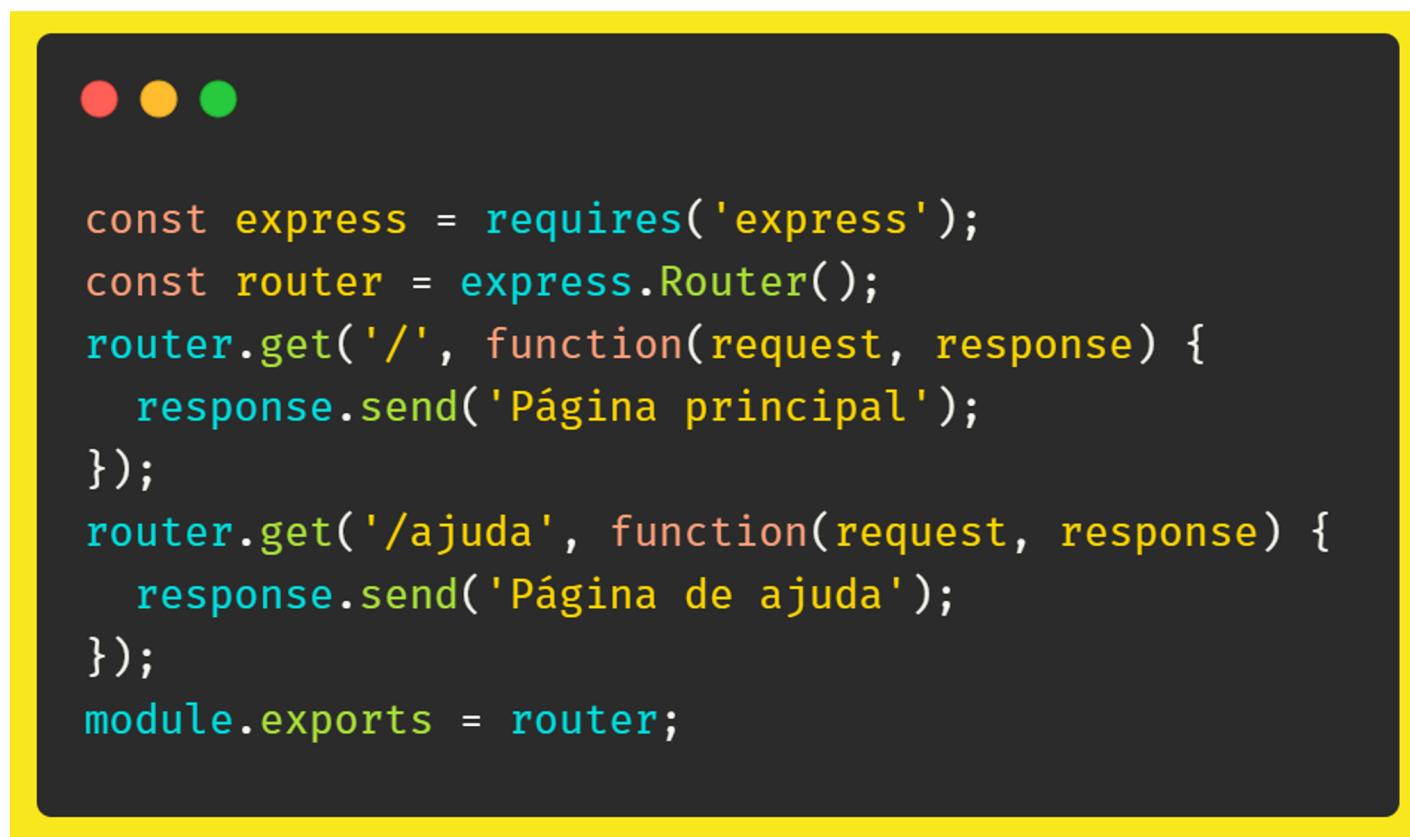
---

- Com o objetivo de simplificar o processo de gestão de rotas, utiliza-se o módulo *router*,
  - A classe `express.Router` permite criar manipuladores de rota;
- Uma instância do *router* é um sistema completo de *middleware* e roteamento que é facilmente exportado e usado numa aplicação;

# <Routing + Middleware/>

---

- Ficheiro: main.route.js



The image shows a terminal window with a yellow border. In the top-left corner of the terminal, there are three small colored circles: red, yellow, and green. The terminal itself has a black background and contains the following code:

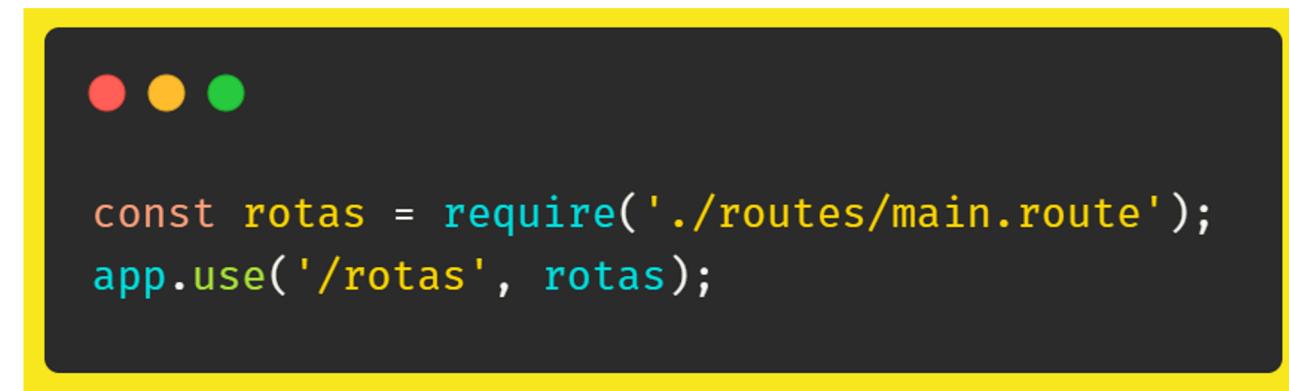
```
const express = require('express');
const router = express.Router();
router.get('/', function(request, response) {
  response.send('Página principal');
});
router.get('/ajuda', function(request, response) {
  response.send('Página de ajuda');
});
module.exports = router;
```

Figura 12 - Definição de rotas através do módulo *router* do Express.

# <Routing + Middleware/>

---

- Para utilizar o router, é necessário fazer o `require()` do ficheiro;
- No ficheiro da aplicação (`app.js`):



```
● ● ●

const rotas = require('./routes/main.route');
app.use('/rotas', rotas);
```

A screenshot of a terminal window with a yellow border. Inside, there are three small colored dots at the top (red, yellow, green). Below them is some code in a monospaced font. The code imports a module named 'rotas' from a file 'main.route' located in the 'routes' directory, and then uses it on the '/rotas' route.

Figura 13 - Importação do módulo router na aplicação.

# <MVC/>

---

- Conforme já foi abordado em aulas anteriores, o padrão **Model-View-Controller** (MVC) divide um sistema em três camadas:
  - **Model** (modelo) - a camada de domínio da aplicação que contém a lógica (e as regras) de negócio, a persistência dos dados, etc.;
  - **View** (vista) - camada que contém as interfaces de utilizador e é responsável por apresentar os dados;
  - **Controller** (controlador) - camada que processa e responde a eventos, recebe alterações no *model* e atualiza a *view*;

# <MVC/>

---

- Em Node.js o padrão **Model-View-Controller** (MVC) é implementado com recurso à *framework Express*;
  - A utilização desta *framework* permite criar uma nova componente na nossa aplicação, as **routes** (rotas);
- As boas práticas na implementação do padrão **MVC** definem o nome que damos às diretorias e respetivos ficheiros;
  - Após criarmos uma diretoria para o projeto/exercício, devemos criar uma diretoria ‘src’ onde ficará armazenado todo o conteúdo da aplicação;
    - A esta fase deverá executar o comando NPM que permite criar o ‘**package.json**’ da aplicação;

# <MVC/>

---

- A organização das diretórias e ficheiros deverá ficar igual à imagem que se segue:

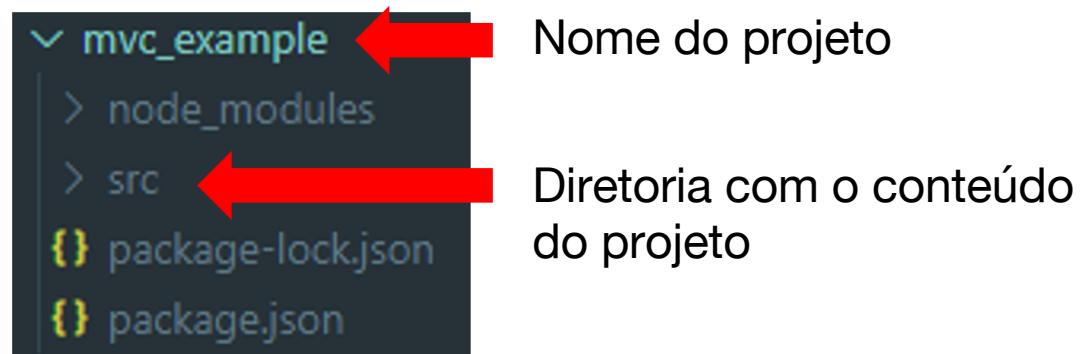


Figura 21 - Exemplificação da estrutura e organização das diretórias de um projeto em NodeJS.

- Dentro da diretoria ‘src’ devemos criar a diretoria ‘controllers’ para armazenar os controladores, uma diretoria ‘models’ para os modelos, uma diretoria ‘views’ para as UI e por último uma diretoria ‘routes’ para as rotas;

# <MVC/>

---

- Também é comum criar-se uma pasta ‘assets’ que inclui todos os ficheiros públicos (p. ex., imagens, ficheiros CSS, etc.) e a pasta ‘config’ com as configurações da aplicação (p. ex., ligação à base de dados, etc.);

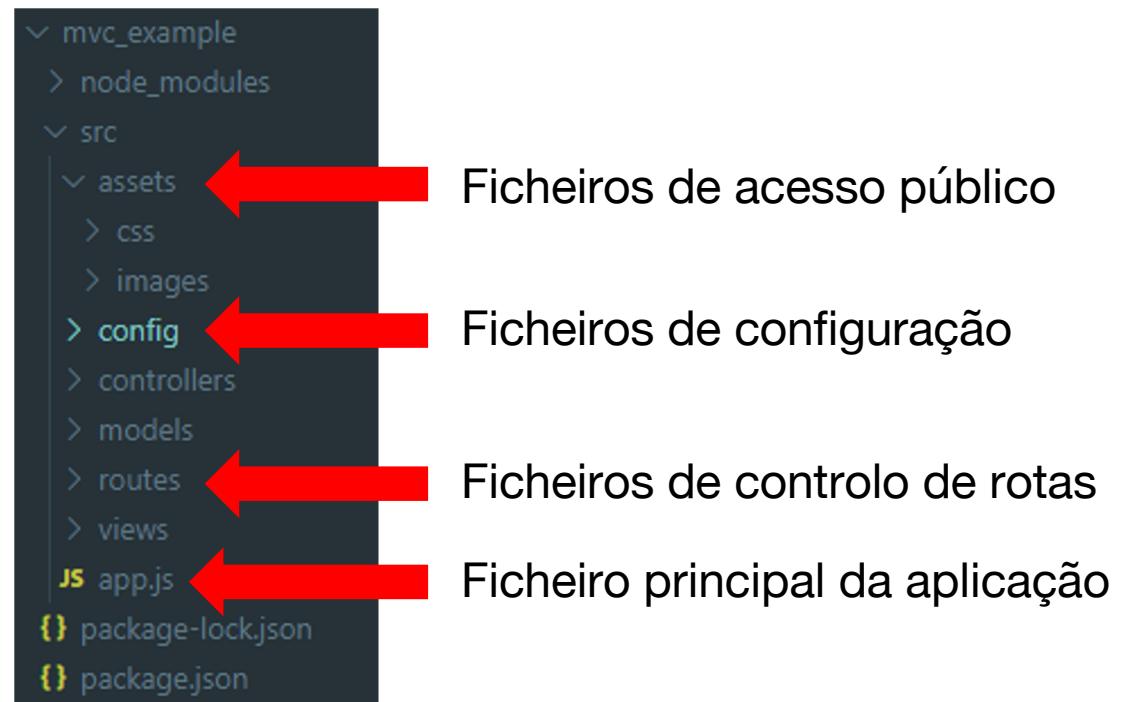


Figura 22 - Exemplificação da estrutura e organização das diretórias de um projeto de acordo com as boas práticas utilizadas na área.

# <MVC/>

---

- O nome dos ficheiros por norma incluem o nome do objeto, seguido do sufixo MVC (i.e., model, controller ou route), separados por um ponto (.);
  - Por exemplo: *pessoa.model.js*, *pessoa.controller.js*, etc;

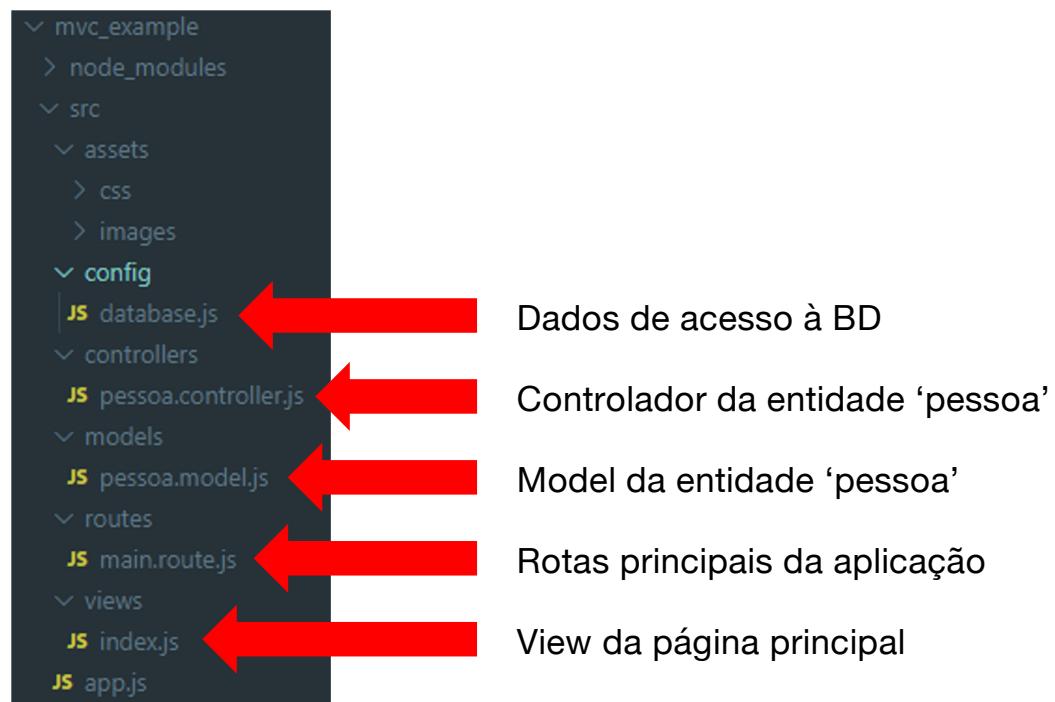


Figura 23 - Exemplificação da estrutura e organização dos ficheiros por diretoria de acordo com as boas práticas utilizadas na área.

# <CRUD/>

---

- O **paradigma CRUD** (*create, read, update e delete*) é fundamental para a construção de aplicações Web robustas;
  - Fornece uma estrutura bem definida e transversal a todos os programadores;
- Num cenário HTTP os pedidos CRUD correspondem respetivamente aos métodos *post, get, put e delete*;

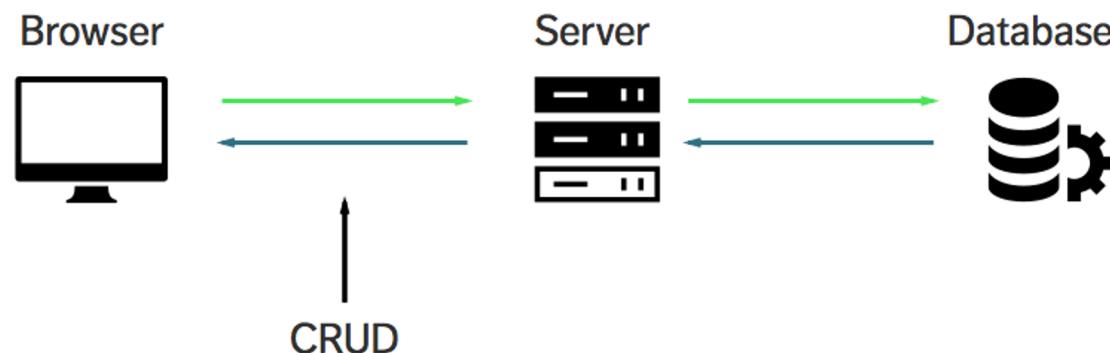


Figura 24 - Exemplificação de um pedido CRUD.  
(imagem retirada do artigo <https://zellwk.com/blog/crud-express-mongodb/>)

# <CRUD/>

---

- Em Express o paradigma CRUD caracteriza-se por utilizar um **método HTTP** através de uma **rota específica** e fazer uma determinada operação na base de dados;

PEDIDO	ROTA DE PEDIDO	OPERAÇÃO NA BD	RESPOSTA (body)	RESPOSTA (http code)
create	POST / data	Inserir os dados enviados no pedido (body)	{"success": "Registo criado com sucesso" }	201
read (todos os dados)	GET / data	Nenhuma (apenas lê)	{"success": [array contendo todos os registos] }	200
read (de um ID específico)	GET / data / :id	Nenhuma (apenas lê)	{"success": [array contendo todos os registos de um determinado ID] }	200

# <CRUD/>

---

PEDIDO	ROTA DE PEDIDO	OPERAÇÃO NA BD	RESPOSTA (body)	RESPOSTA (http code)
update	PUT / classes / :id	Alterar dados enviados no pedido (body) para determinado ID	{"success": "Registo atualizado com sucesso" }	200
delete	DELETE / classes / :id	Remover dados de determinado ID	Nenhuma	204

Tabela 1 - Exemplificação do comportamento CRUD em Express.

- Em projetos Web as tabelas das BD podem ser acedidas via SQL puro (através do módulo ‘mysql’) ou recorrendo a um *Object-Relational Mapping* (ORM);

# <ORM/>

---



- O ORM é uma técnica de mapeamento de objetos relacionais que permite fazer uma relação dos objetos com os dados que estes representam;
  - Em suma, permite mapear tabelas SQL em objetos relacionais;
- Em Express o ORM mais utilizado é o módulo ‘sequelize’ (<https://www.npmjs.com/package/sequelize>), que suporta MySQL e está preparado para realizar todas as operações de SQL como um objeto;
  - Os resultados das operações de BD são retornados em formato JSON;

# <API/>

---

- *Application Programming Interface* (API)
  - De uma forma geral uma API é composta por uma **série de funções**, acessíveis apenas por programação, que permitem utilizar características/funcionalidades de uma aplicação;
    - Evita a necessidade de mostrar detalhes da sua implementação interna;
  - As API privilegiam a **interface máquina-máquina** e permitem aos programadores aceder a funcionalidades de outras aplicações/serviços, através de uma estrutura de dados bem definida;

# <API/>

---

- Os autores das aplicações ou serviços criam, por norma, uma API específica para os outros programadores;
  - Permitindo criar plugins, estendendo-lhes, assim, as funcionalidades do programa;

# <Web API/>

---

- É uma API aplicada ao contexto Web;
- Ao desenvolver um site, o programador pode ter acesso, via API, a diversas outras aplicações/serviços;
- Conjunto definido de mensagens de pedido e resposta HTTP;
  - Podem ser expressos no formato XML ou JSON (preferencial);
  - Atualmente os serviços web tem vindo abandonar o modelo de serviços SOAP em detrimento do **Representational State Transfer (REST)**;

# <Web API + REST/>

---

- Vantagens do REST numa Web API:
  - Eficiência;
  - Diversidade de aplicações;
  - Gestão de processos;
  - Automatização de procedimentos;
  - Interoperabilidade e integração;
  - Personalização;
  - Entre outros.

# <REST/>

---

- REST = **Representational State Transfer**;
- O REST consiste em princípios/regras/constraints que utiliza padrões entre sistemas Web de modo a facilitar a comunicação entre si;
  - Os sistemas compatíveis com REST são geralmente denominados de **RESTful**;
- Caracterizam-se pelo facto de não terem estado (*stateless*) e de separarem as camadas do cliente e do servidor;
- Esta arquitetura pressupõe que as implementações do cliente e do servidor sejam efetuadas de forma independente, sem que um tenha conhecimento do outro;
  - O código do lado do cliente pode ser alterado a qualquer momento sem que isso afete a operação do lado do servidor, e vice-versa;

# <REST/>

---

- Para que a comunicação seja realizada com sucesso, apenas é necessário saber o **formato das mensagens** usado para cada um dos lados;
- Exemplo de um endpoint da API REST de Projetos do GitHub:
  - <https://developer.github.com/v3/projects/>

List organization projects ⓘ

Lists the projects in an organization. Returns a `404 Not Found` status if projects are disabled in the organization. If you do not have sufficient privileges to perform this action, a `401 Unauthorized` or `410 Gone` status is returned.

GET /orgs/:org/projects

Name	Type	Description
<code>state</code>	<code>string</code>	Indicates the state of the projects to return. Can be either <code>open</code> , <code>closed</code> , or <code>all</code> . Default: <code>open</code>

Response

```
Status: 200 OK
Link: <https://api.github.com/resource?page=2>; rel="next",
      <https://api.github.com/resource?page=5>; rel="last"
[{"id": 1002683,
 "node_id": "MDQ6UHJvamVjdDw0MDI2NDM=",
 "name": "My Projects",
 "body": "A board to manage my personal projects.",
 "number": 1,
 "state": "open",
 "creator": {
   "login": "octocat",
   "id": 1,
   "node_id": "MDQ6XXNlcjE=",
   "avatar_url": "https://github.com/images/error/octocat_happy.gif",
   "gravatar_id": "",
   "url": "https://api.github.com/users/octocat",
   "html_url": "https://github.com/octocat",
   "followers_url": "https://api.github.com/users/octocat/followers",
   "following_url": "https://api.github.com/users/octocat/following{/other_user}",
   "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
   "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",
   "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
   "organizations_url": "https://api.github.com/users/octocat/orgs",
   "repos_url": "https://api.github.com/users/octocat/repos",
   "events_url": "https://api.github.com/users/octocat/events{/privacy}",
   "received_events_url": "https://api.github.com/users/octocat/received_events",
   "type": "User",
   "site_admin": false
 },
 "created_at": "2011-04-10T20:09:31Z",
 "updated_at": "2014-03-03T18:58:10Z"}]
```

# <REST/>

---

- Uma pedido REST necessita de:
  - Um **método HTTP** para definir a operação a ser executada;
  - Um **cabeçalho**, onde o cliente passa as informações sobre o pedido;
  - Um **URL (*endpoint*)** com o caminho para um recurso REST;

# <REST/>

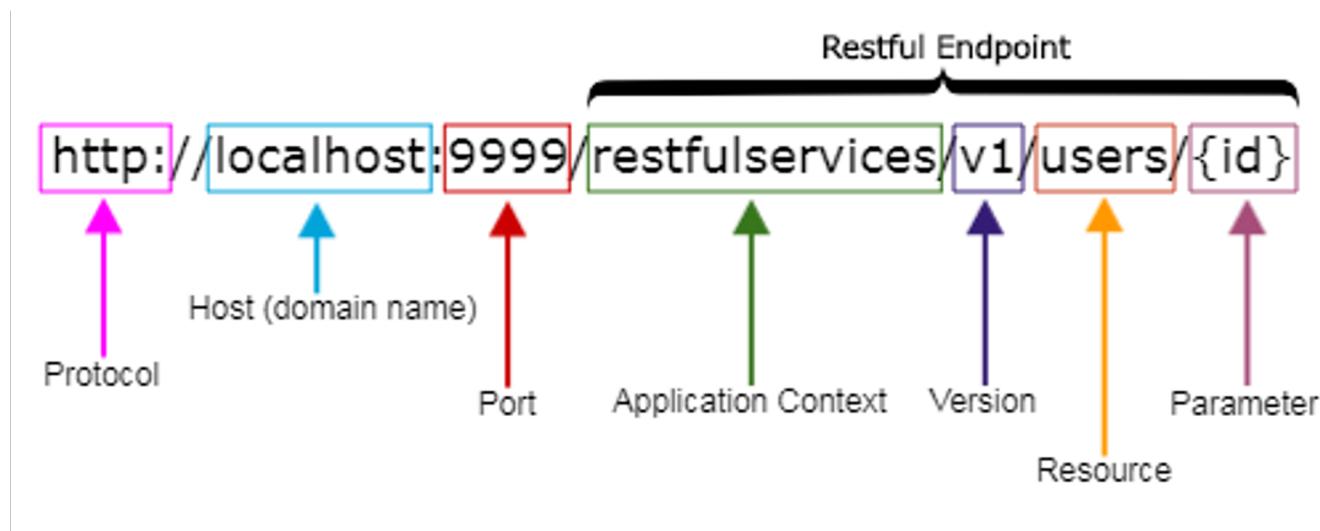
---

- Uma aplicação RESTful utiliza métodos HTTP para fornecer serviços (identificados por um URI), que por sua vez fornecem representações (dados);
- Existem quatro métodos HTTP básicos para interagir com coleções/recursos num sistema REST:
  - *get* - lê um dado específico (por ID) ou um conjunto de dados;
  - *post* - cria um novo dado;
  - *put* - atualiza um dado específico (por ID);
  - *delete* - remove um dado específico (por ID);
- As coleções (*collections*) são um conjunto de recursos;

# <REST/>

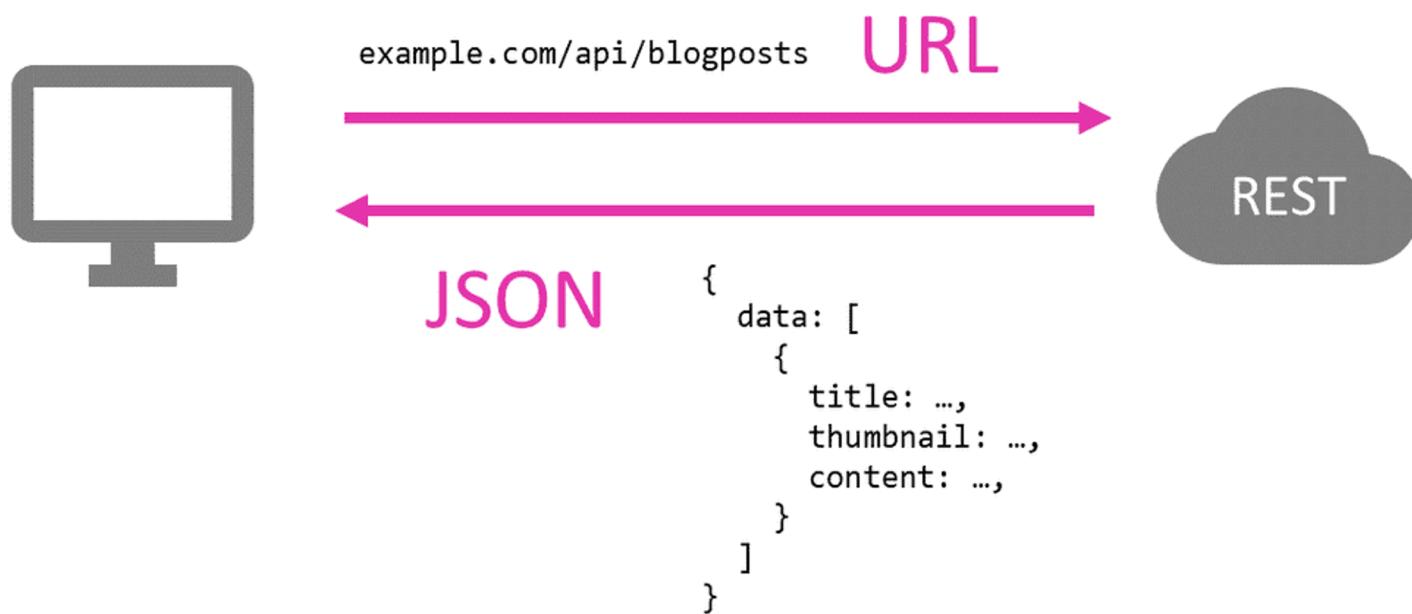
---

- O URI é o caminho (*path*) através do qual um recurso pode ser localizado e onde nesse recurso as ações podem ser efetuadas;
  - Todos os recursos são acedidos através do URI;



# <REST/>

---



---

Figura 25 - Exemplificação de um pedido REST a um endpoint específico. A resposta é uma coleção no formato JSON.