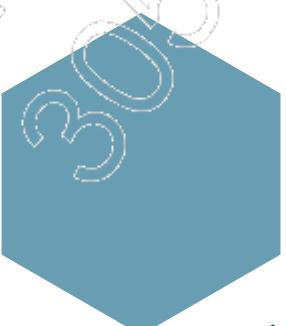




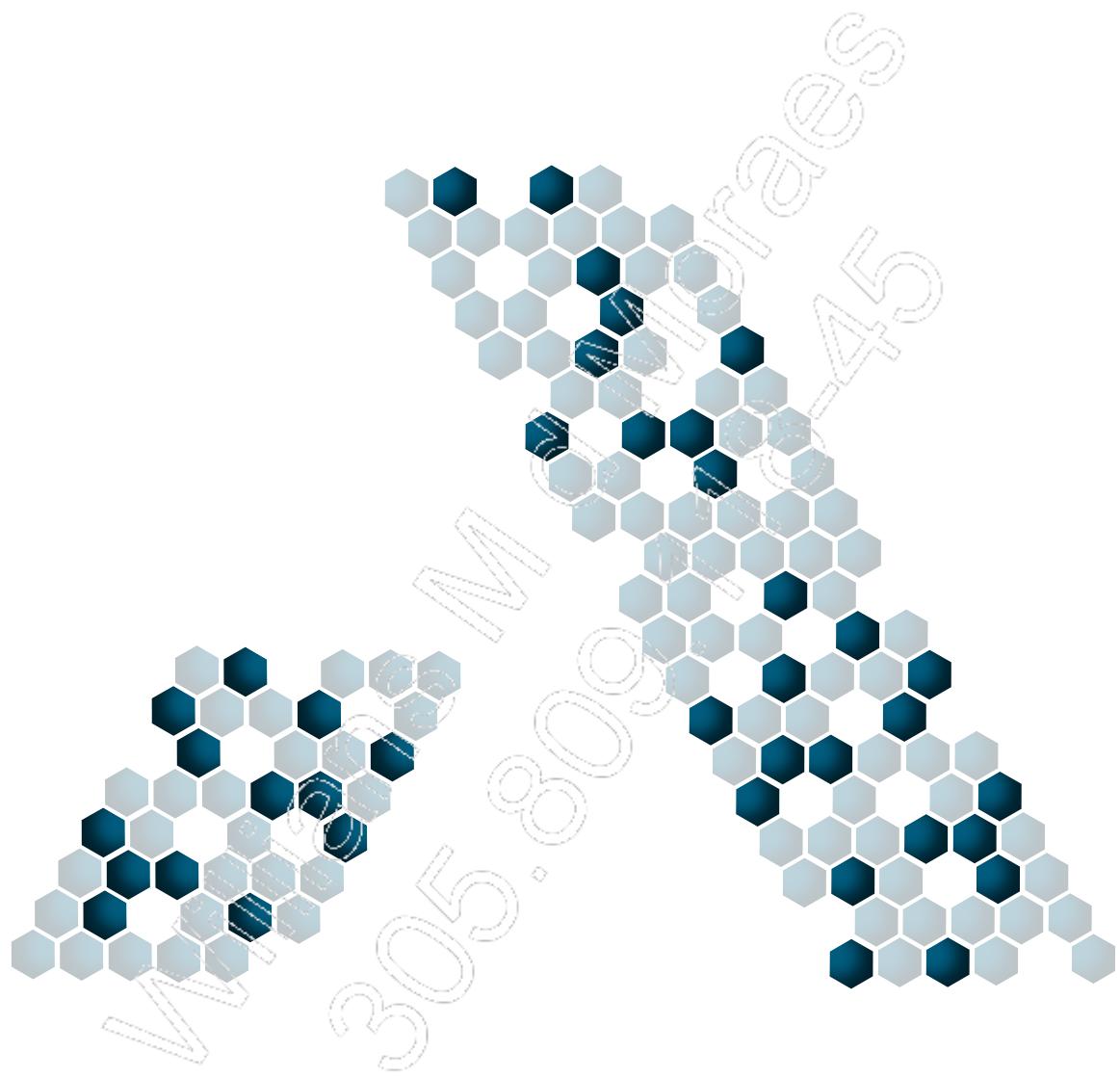
Desenvolvimento Web com Node.js, Angular e MongoDB

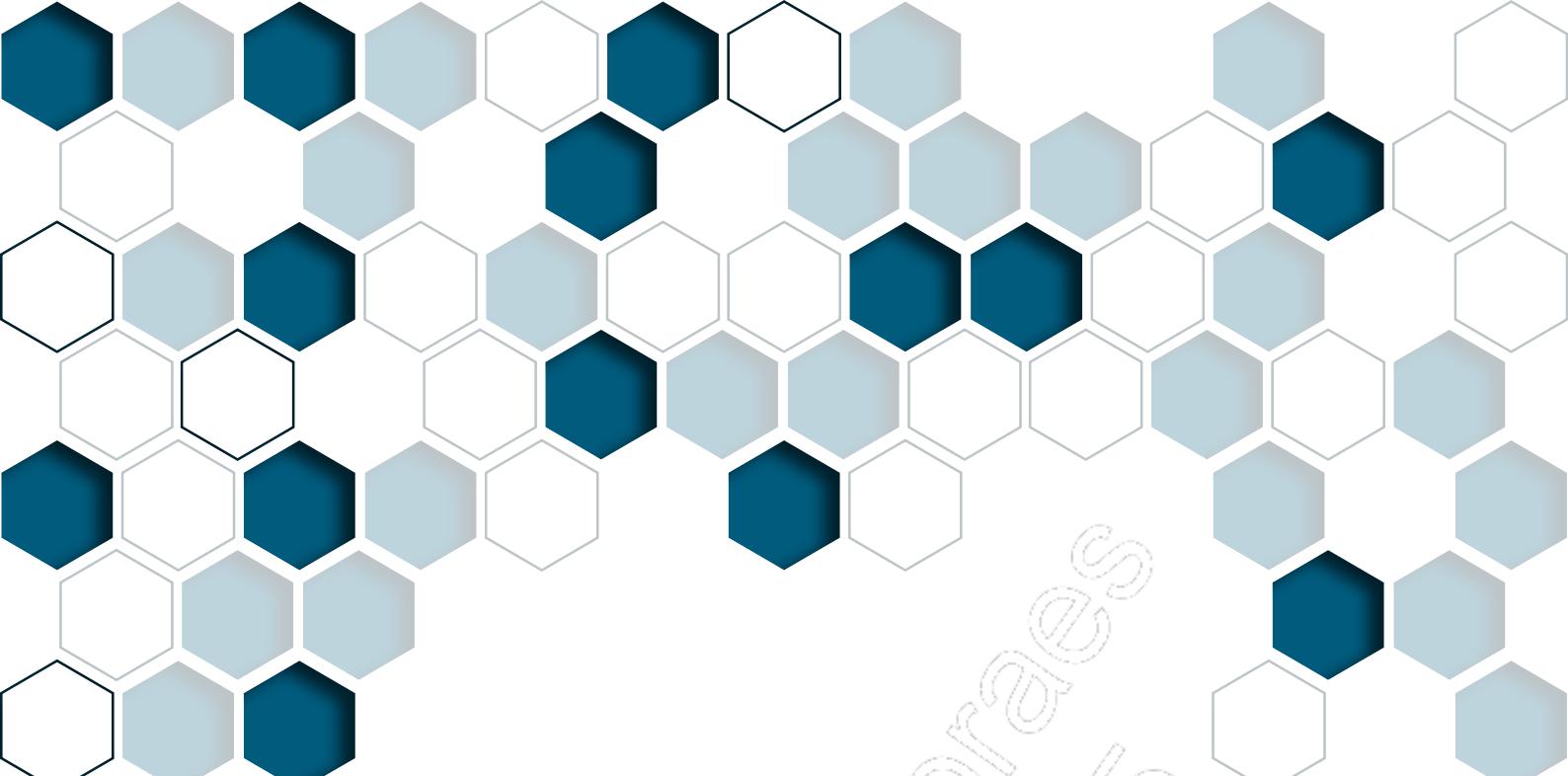
Millions
305.800



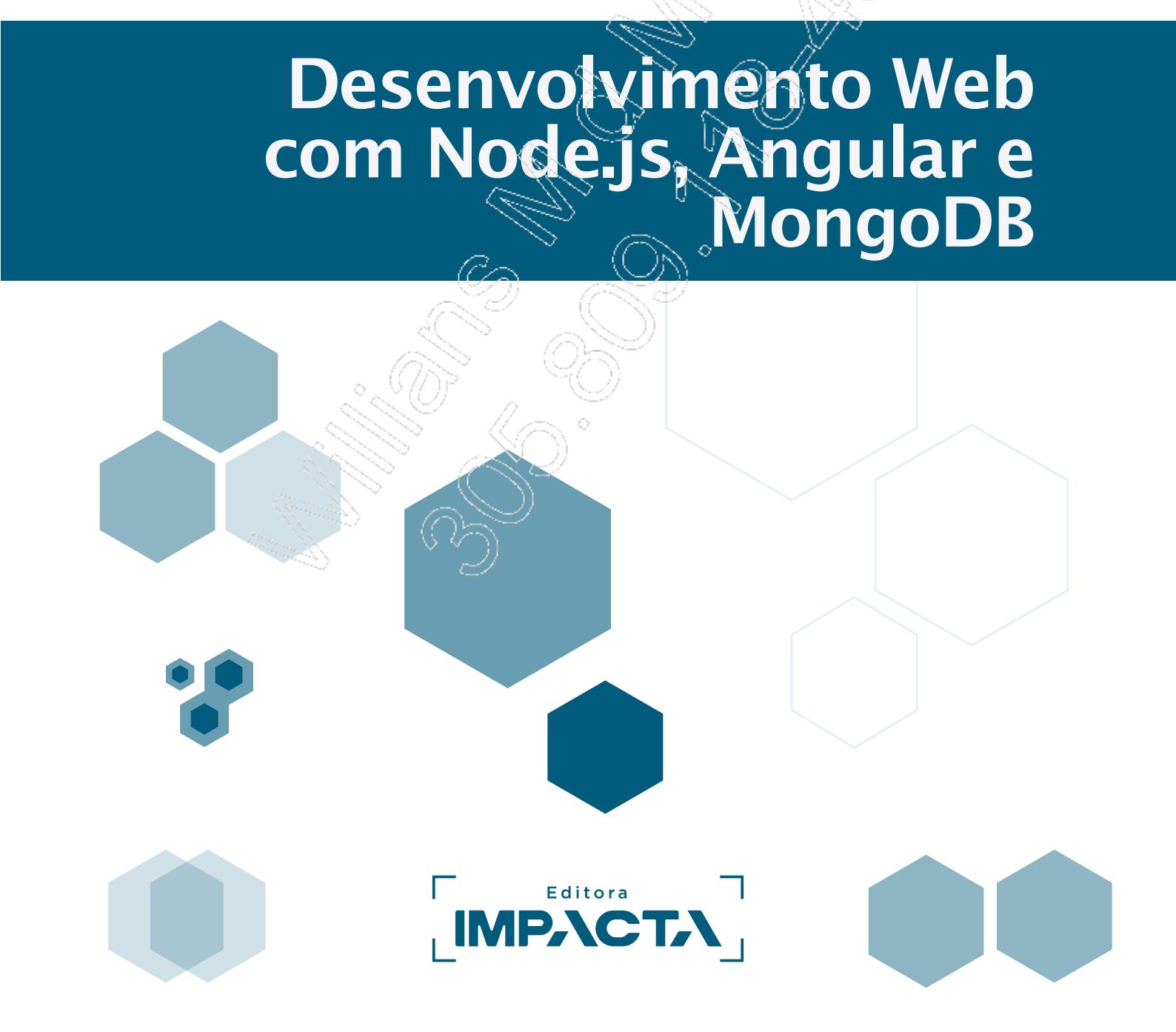
Editora
IMPACTA







Desenvolvimento Web com Node.js, Angular e MongoDB



Millions

305.800

0.800

45

M



Créditos

Copyright © Monte Everest Participações e Empreendimentos Ltda.

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da Monte Everest Participações e Empreendimentos Ltda., estando o contrafator sujeito a responder por crime de Violação de Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."

Desenvolvimento Web com Node.js, Angular e MongoDB

Coordenação Geral

Marcia M. Rosa

Coordenação Editorial

Henrique Thomaz Bruscagin

Autoria

Emilio Celso de Souza

Revisão Ortográfica e Gramatical

Fernanda Monteiro Laneri

Diagramação

Bruno de Oliveira Santos
Paloma da Silva Teixeira

Edição nº 1 | 1840_3

Maio/2019

Sumário

Conteúdo Programático	07
Introdução	13
1. Desenvolvimento com Node.js	19
2. O framework Express.js	29
3. Criando e consumindo banco de dados com MongoDB	53
4. Desenvolvimento de Web services com Node.js, Express e MongoDB	71
5. Desenvolvimento com AngularJs	81
6. Conhecendo o Angular 4	107
Apêndice - Visão geral do JavaScript.....	143
 Projeto	 165
1. Preparando o ambiente	167
2. Node.js e Express.js.....	169
3. Acesso a dados com Mongoose	185
4. Criando e consumindo Web services	195
5. Definindo uma aplicação com AngularJS – MEAN (Parte 1)	211
6. Definindo uma aplicação com Angular 4 – MEAN (Parte 2).....	219
Atividades do Apêndice - Conceitos do JavaScript.....	243
 Mãos à obra!.....	 251
1. Criando um projeto com Node.js	253
2. Criando um Web service com Node.js	255
3. Criando um projeto MEAN com AngularJS	257
4. Criando um projeto MEAN com Angular 4	259

Conteúdo Programático

305.800

Williams Mid Mergers & Acquisitions



Conteúdo programático - 1/7

1 – Desenvolvimento com Node.js

- O Node.js como servidor;
- Usando eventos, listeners e funções callback;
- Gerenciamento de arquivos e requisições;
- Implementando módulos;
- Criação de Web services.



Conteúdo programático - 2/7

2 – O framework Express.js

- Implementando o modelo MVC com Express;
- Modelos, rotas, controllers e views.



Conteúdo programático - 3/7

3 – Criando e consumindo banco de dados com MongoDB

- Conceitos de NoSQL;
- Instalando, criando e acessando banco de dados;
- Usando o Mongoose;
- Implementando acesso ao MongoDB no Node.js.



Conteúdo programático - 4/7

4 – Desenvolvimento de Web services com Node.js, Express e MongoDB

- Definindo Web services REST;
- Criando serviços.



Conteúdo programático – 5/7

5 – Desenvolvimento com AngularJS

- Conceitos;
- A arquitetura MVC;
- Controllers, Models e Views;
- Diretivas;
- Estrutura do AngularJS;
- Usando serviços;
- Rotas e modelos SPA;
- Acesso a Web services com AngularJS.



Conteúdo programático – 6/7

6 – Conhecendo o Angular 4°

- Conceitos;
- Criando um projeto;
- Módulos e componentes;
- Rotas no Angular 4°;
- Serviços de Injeção de Dependência;
- Acesso a Web services com Angular 4°;
- Binding unidirecional e bidirecional;
- Outras considerações.



Conteúdo programático - 7/7

Apêndice – Visão geral do JavaScript

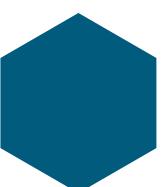
- Variáveis;
- Funções;
- Conhecendo o DOM;
- Eventos e listeners;
- Conhecendo o jQuery.



Treinamentos
IMPACTA

Introdução

305.800



Editora
IMPACTA



Introdução

Neste curso, será possível aplicar de forma bastante prática os conceitos associados ao desenvolvimento de aplicações Web, desde o back-end até o front-end, passando pelo banco de dados, utilizando como linguagem central o JavaScript. Este desenvolvimento é conhecido como **Full Stack**.

O desenvolvimento Full Stack engloba as tecnologias a seguir:

- **MongoDB**: Banco de dados;
- **Express.js**: Framework MVC para desenvolvimento Web;
- **Angular**: Front-end interativo;
- **Node.js**: Middleware para gerar o servidor de aplicações.



Introdução

O conjunto dessas tecnologias dá nome à plataforma: **MEAN Stack**.

A palavra **MEAN** é um acrônimo para as ferramentas usadas (**Mongo**, **Express**, **Angular** e **Node**).

A interação entre essas ferramentas se deve à uniformidade da linguagem que, como já mencionado, é o JavaScript.



Introdução



mongoDB®

Em MEAN Stack, o banco de dados é o **MongoDB**, devido ao formato dos dados, baseado em **documento** (e não composto por tabelas, como é o caso de bancos relacionais).

Este formato não considera linhas e colunas; apenas linhas. Cada linha é um elemento BSON (Binary JSON – um JSON serializado).

Link: <https://www.mongodb.com/>



Introdução

Vamos conhecer um pouco melhor cada uma dessas ferramentas:



Introdução express

Link: <http://expressjs.com/>

Express.js é uma plataforma (também conhecida como framework). Seu desenvolvimento segue a codificação do Node.js, e diversos componentes (pacotes) são adicionados via npm (mais sobre npm adiante).

Usado para desenvolvimento Web, também pode ser combinado com outros componentes de interface gráfica, como o Bootstrap, por exemplo.



Introdução



Links:

<https://angularjs.org/> (AngularJS)
<https://angular.io/> (Angular 4)

AngularJS, também conhecido como **Angular 1.x**, é usado para desenvolvimento do front-end da aplicação. Por meio de diretivas, o AngularJS permite iteração com partes do back-end.

O **Angular 4**, apesar do nome, é diferente do AngularJS, pois depende de uma estrutura de projeto adequada e usa o TypeScript como linguagem central (após compilado, ele se torna JavaScript).



Introdução



Node.js é o núcleo de todo desenvolvimento MEAN Stack.

Com ele é possível criar o próprio servidor Web, além de incluir pacotes para os outros itens (MongoDB, Angular 4 e Express).

Junto com o Node.js, temos o utilitário npm (node package manager), responsável pela instalação dos componentes.

Link: <https://nodejs.org/>



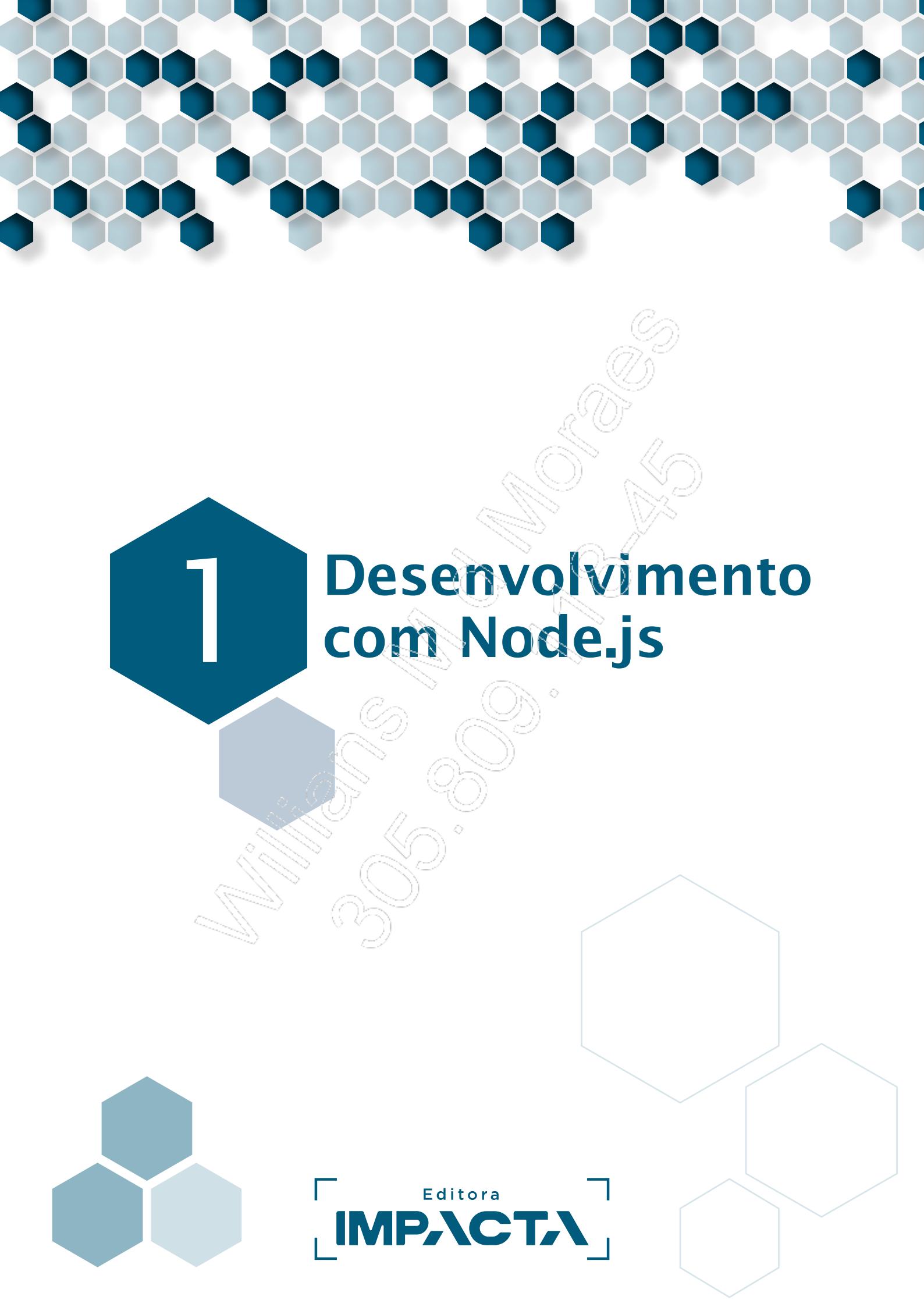
Introdução

Ao longo do curso, estudaremos cada uma dessas ferramentas e como elas interagem para compor uma aplicação real.

Os códigos desenvolvidos ao longo deste curso podem ser escritos em qualquer editor de textos, mas utilizaremos o **Visual Studio Code** (também conhecido como **VS Code**).

Podemos obtê-lo em <https://code.visualstudio.com/>.



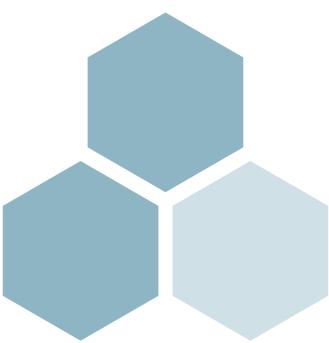


Desenvolvimento com Node.js

Wiliams Williams
305.800.
Moraes Moraes
45



Editora
IMPACTA



Contextualização

Em poucas palavras, **Node.js** é uma plataforma de software que permite a criação de um Web server e o desenvolvimento de aplicações “no seu topo”, ou seja, aplicações Web capazes de serem executadas pelo servidor criado pelo próprio Node.js.

O Node.js é composto por módulos, e um dos módulos disponíveis é o módulo HTTP. Esse módulo permite a criação do servidor, sem a necessidade de ferramentas adicionais – como é o caso do PHP, por exemplo, que necessita do servidor Apache, ou do ASP.NET, que é executado sobre o IIS.



Iniciando com o Node.js

O primeiro passo para trabalhar com o Node.js é obtê-lo e instalá-lo:

- Acesse o link <https://nodejs.org/en/>;
- Obtenha a versão mais recente e instale-a;
- Após a instalação no Windows, busque por **Variáveis de Ambiente** e configure:
`NODE_ENV='development'`

- Para testar, execute isto no prompt de comandos:

```
node -v
```



Iniciando com o Node.js

- Verifique o número da versão.
- Juntamente com o Node.js, é instalado o **npm** (node package manager). Para testá-lo, execute isto:

```
npm -v
```



Executando um código com Node.js

- Selecione uma pasta adequada para seus projetos. Use o VSCode para referencia-la.
- Todo código executado pelo Node.js deve possuir a extensão **.js** (JavaScript). Crie, então, um arquivo chamado **exemplo01_teste.js**.
- Nesse arquivo, escreva o seguinte código:

```
console.log('curso node.js');
```

- No terminal, execute o seguinte comando:

```
node exemplo01_teste.js
```



Executando um código com Node.js

- No VSCode ainda é possível acessar o prompt no próprio editor. Basta digitar **CTRL + '** e usá-lo.



Definindo módulos

O ponto forte do Node.js está nos módulos. É por meio deles que podemos criar novas aplicações, como aplicações Web com Express.js ou aplicações SPA com Angular 4.

O padrão utilizado no carregamento dos módulos é chamado **CommonJS**.

Um módulo é, na verdade, um código JavaScript. Para exemplificar, vamos definir três arquivos: dois representando os módulos e o terceiro representando a aplicação, utilizando o módulo em questão.



Definindo módulos

- Arquivo **mod1.js**:

```
module.exports = function (x) {
  console.log(x);
}
```

- Arquivo **mod2.js**:

```
exports.mensagem = function (x) {
  console.log(x);
}
```



Definindo módulos

Carregando os módulos:

- Arquivo **app.js**:

```
var m1 = require('./mod1');
var m2 = require('./mod2');

m1('Carregando uma única função modular');
m2.mensagem('Carregando objeto com funções modulares');
```

O símbolo ‘./’ indica o mesmo diretório. Para testar, execute isto:



Definindo um servidor Web

Da mesma forma que podemos definir nossos próprios módulos, existem diversos módulos prontos para serem usados. Para ilustrar, vamos utilizar dois módulos: **http** e **fs** (**fs** se refere a **file system**):

Definindo um servidor Web

- Arquivo `app_server.js`:

```
var http = require('http');
var requisicao = function (request, response) {
    response.writeHead(200, { "Content-Type": "text/html" });
    response.write("<h1>Texto a ser exibido no browser</h1>");
    response.end();
}

var server = http.createServer(requisicao);
var resultado = function () {
    console.log('Servidor em funcionamento!');
}
server.listen(3000, resultado);
```

Definindo um servidor Web

Vamos analisar o código anterior:

1. Obtemos uma referência ao módulo **http**:

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

2. Definimos uma função com dois parâmetros (requisição e resposta):

```
var requisicao = function (request, response) {
    response.writeHead(200, { "Content-Type": "text/html" });
    response.write("<h1>Texto a ser exibido no browser</h1>");
    response.end();
}
```



Definindo um servidor Web

3. Definimos uma variável responsável pela conexão com o servidor. Observe que a função **createServer()** espera uma função **callback** contendo dois parâmetros, que são configurados adequadamente:

```
var server = http.createServer(requisicao);
```

4. Esta função é opcional, porém importante. Ela será executada quando a conexão for estabelecida e o servidor estiver no ar:

```
var resultado = function () {
    console.log('Servidor em funcionamento!');
}
```

5. Finalmente, definimos o servidor atendendo a requisições na porta 3000:

```
server.listen(3000, resultado);
```



Definindo um servidor Web

Para executar:

1. No prompt, execute isto:

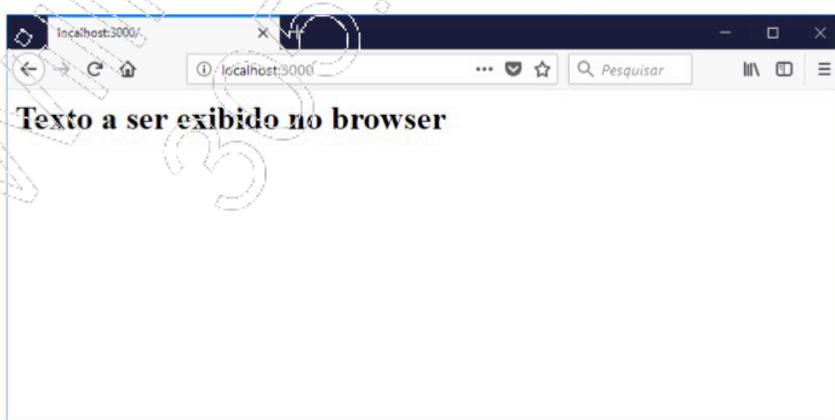
```
node app_server.js
```

2. Quando surgir a mensagem **Servidor em Funcionamento**, o servidor já estará no ar. Abra um browser da sua escolha e digite esta URL:

```
localhost:3000
```

Definindo um servidor Web

O resultado deverá ser algo semelhante ao apresentado abaixo:



Definindo um servidor Web

O exemplo anterior considera um conteúdo HTML escrito na própria função referenciada pela variável **requisicao**. Se for necessária uma página mais elaborada, esse procedimento se tornará inviável.

Vamos, então, apresentar um exemplo no qual o conteúdo HTML é exibido em uma página separada:



Definindo um servidor Web

- Arquivo **index.html**:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Olá este é o meu site pessoal!</title>
  </head>
  <body>
    <h1>Bem vindo ao meu site pessoal</h1>
  </body>
</html>
```



Definindo um servidor Web

- Arquivo **fs_server.js**:

```
var http = require('http');
var fs = require('fs');
var server = http.createServer(function (request, response) {
  fs.readFile(__dirname + '/index.html', function (erro, html) {
    response.writeHead(200, { 'Content-Type': 'text/html' });
    response.write(html);
    response.end();
  });
});
server.listen(3000, function () {
  console.log('Executando Site Pessoal');
});
```



Definindo um servidor Web

A constante `__dirname` retorna o diretório raiz da aplicação.

Verifique que nesse exemplo usamos funções callback anônimas, em vez de escrevê-las separadamente.

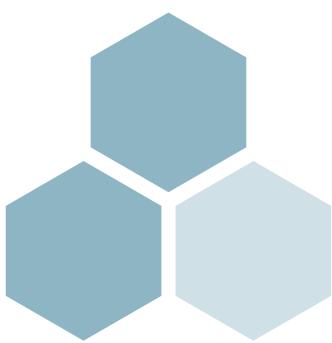




2

O framework Express.js

Williams
305.800.
Moraes
45



Conceitos do Express.js

Express (ou Express.js) é um framework para desenvolvimento Web baseado no Node.js.

Podemos dizer que, com exceção do banco de dados MongoDB, é possível obtermos uma aplicação completa baseada na arquitetura MVC usando o Node e o Express.

O Express, assim como qualquer outro framework baseado no Node.js, é composto por módulos. Cada módulo especifica uma funcionalidade da aplicação. Os módulos são incluídos na aplicação por meio do gerenciador de módulos, o npm, e o npm instala os módulos com base nas informações do arquivo **package.json**.

O arquivo package.json

A seguir, apresentaremos um exemplo do arquivo **package.json**:

```
{  
  "name": "app-name",  
  "version": "0.0.1",  
  "private": true,  
  "dependencies": {  
    "body-parser": "~1.15.1",  
    "cookie-parser": "~1.4.3",  
    "debug": "~2.2.0",  
    "ejs": "~2.4.1",  
    "express": "~4.13.4",  
    "morgan": "~1.7.0",  
    "serve-favicon": "~2.3.0",  
  }  
}
```

Metadados da aplicação

Dependências

O arquivo package.json

Observe que, na parte de dependências, que é o que o Node instala de fato no projeto, existem versões com símbolos especiais na frente do número da versão, como ~ ou ^. Esses símbolos possuem significados especiais, que serão descritos a seguir:



O arquivo package.json

versão: Exatamente a versão indicada

~versão: Aproximadamente

>= versão: Igual ou maior que

> versão: Maior que

<= versão: Igual ou menor que

< versão: Menor que

^versão: Compatível com

1.2.x: 1.2.0, 1.2.1, 1.2.2,..., mas não 1.3.0

A documentação do npm pode ser obtida em
<https://docs.npmjs.com/>.



Instalando as dependências

O arquivo **package.json** fica localizado na raiz da pasta do projeto. A instalação das dependências requeridas nesse arquivo é feita por meio deste comando:

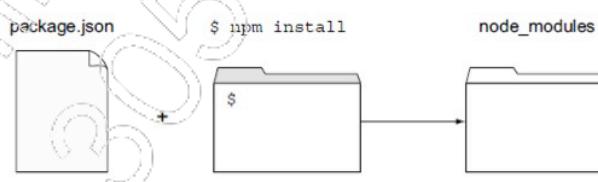
```
npm install
```

Esse procedimento instrui o npm a baixar todas as dependências e instalá-las em uma pasta chamada **node_modules**.

Treinamentos
IMPACTA

Instalando as dependências

A seguir, temos uma ilustração desse processo:



Treinamentos
IMPACTA

Instalando o Express

Com a pasta de instalação do projeto Express selecionada, execute o comando para baixar e instalar o framework Express.js:

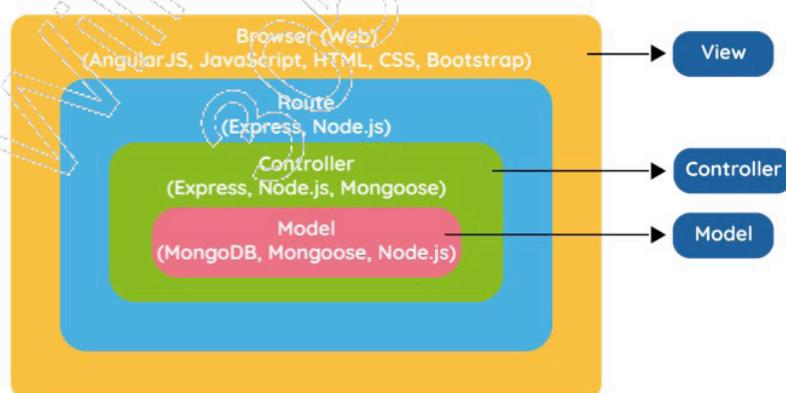
```
npm install -g express-generator
```

O atributo **-g** instala o Express globalmente. Se omitido, o Express é instalado para o usuário em atividade no momento.



Criando um projeto Express

Uma aplicação Express possui as partes indicadas nesta imagem:



Criando um projeto Express

Para verificar a versão instalada do Express, basta executar este comando:

```
express --version
```

Se a instalação procedeu corretamente, será possível ver a versão instalada.

Dependendo da utilização (se via prompt de comando ou pelo PowerShell do Windows), pode ser necessário reiniciar o terminal.



Criando um projeto Express

Algumas considerações importantes devem ser levadas em conta ao criar um projeto Express:

- Qual o mecanismo a ser usado para gerar as Views (Template HTML);
- Qual tipo de processamento CSS utilizar;
- Se é necessário adicionar suporte para gerenciamento de sessão.

Uma instalação default utilizará o mecanismo Jade como template HTML, porém ele não possui suporte a CSS ou suporte a gerenciamento de sessão. Sendo assim, é possível especificar diferentes mecanismos na ocasião da criação do projeto.



Criando um projeto Express

Alguns exemplos de configurações que podem ser utilizadas na criação do projeto:

Comando de configuração	Efeito
--css less stylus	Inclui um pré-processador CSS no projeto, tanto no formato LESS quanto Stylus.
--ejs	Altera o mecanismo de gerenciamento de views de Jade para EJS.
--jshtml	Altera o mecanismo de gerenciamento de views de Jade para JsHTML.
--hogan	Altera o mecanismo de gerenciamento de views de Jade para Hogan.



Criando um projeto Express

Uma primeira opção para criarmos um projeto usando o Node é:

- Definir uma pasta para o projeto;
- Entrar na pasta;
- Digitar o comando **npm init**;
- Fornecer as informações básicas do projeto.

O comando **npm init** criará um arquivo **package.json** com as informações mínimas para o projeto.



Criando um projeto Express

Outra opção para criação do projeto (a que utilizaremos aqui) é usar o comando **express** (após ter instalado o **express-generator**). Vamos, então, criar um projeto chamado **app_exemplo**, usando o mecanismo **ejs**:

```
express app_exemplo --ejs
```

A pasta **app_exemplo** é criada, já com algumas estruturas de pastas prontas e com o arquivo **package.json** presente. Para instalar os módulos, temos que entrar nessa pasta e executar o comando para instalar os pacotes via npm:

```
cd app_exemplo  
npm install
```



Criando um projeto Express

Quando o processo de instalação terminar, estaremos prontos para iniciar o desenvolvimento da nossa aplicação.

Ao longo do desenvolvimento, poderá ser necessário instalar novos pacotes (para gerenciamento de sessão, por exemplo). Quando isso ocorrer, basta executar o seguinte comando:

```
npm install --save package-name
```

O atributo **--save** é opcional. Ele inclui a dependência instalada de forma avulsa no arquivo **package.json** para futuras instalações (sem a necessidade de selecionar novamente todos os pacotes).



Executando um projeto Express

Uma vez concluída a instalação dos pacotes, por meio do comando **npm install**, podemos encontrar um arquivo chamado **app.js** na raiz do projeto.

Dificilmente usamos o código gerado para esse arquivo. Sempre incluímos nosso próprio código. É o que faremos em todos os projetos que criarmos com base no Express!



Executando um projeto Express

Vamos, então, apagar todo o conteúdo de **app.js** e incluir este conteúdo:

```
var express = require('express');
var app = express();
app.get('/exemplo', function (requisicao, resposta) {
  resposta.send("<h1>Exemplo do uso do Express</h1>");
});
app.listen(3000, function () {
  console.log("servidor no ar");
});
```



Executando um projeto Express

1. Referenciamos o módulo **express**:

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

2. Criamos uma referência ao **express**, já acessando o módulo:

```
var app = express();
```

3. Definimos uma URL '**/exemplo**' com um texto HTML:

```
app.get('/exemplo', function (requisicao, resposta) {  
    resposta.send("<h1>Exemplo do uso do Express</h1>");  
});
```



Executando um projeto Express

4. Criamos um servidor, desta vez usando o **express** em vez do módulo **http**:

```
app.listen(3000, function () {  
    console.log("servidor no ar");  
});
```

A execução é similar àquela realizada na definição do servidor via **http**. Executamos este comando:

```
node app.js
```



Executando um projeto Express

Quando recebermos a mensagem que o servidor está no ar, abriremos um browser e executaremos a seguinte URL:

localhost:3000/exemplo



Criando um projeto Express

Podemos perceber no exemplo anterior que, ao trabalhar com o **express**, o conteúdo HTML enviado por meio da instrução...

```
resposta.send("<h1>Exemplo do uso do Express</h1>");
```

...não é o mais adequado, uma vez que foi necessário escrever código HTML a ser enviado para o servidor.
E se fosse um conteúdo mais complexo?

O mecanismo EJS que adicionamos no projeto irá nos auxiliar nesta tarefa (afinal, é para o gerenciamento da camada de visualização que ele serve!)



Trabalhando com o EJS

Para utilizar o EJS, é necessário configurá-lo no **app.js**. Devemos acrescentar esta instrução:

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

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

O EJS considera a presença de arquivos na pasta **views**.

É importante que haja uma organização de pastas para cada grupo de views. No nosso exemplo, criaremos a seguinte estrutura de pastas:



Trabalhando com o EJS

/views/exemplos

Dentro dessa pasta, criaremos o arquivo **app.ejs**, com o seguinte conteúdo:



Trabalhando com o EJS

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Exemplo Ejs</title>
</head>
<body>
    <h1>Conteúdo de uma página EJS</h1>
    <p>Aqui pode ser incluído conteúdo CSS e Javascript</p>
</body>
</html>
```



Trabalhando com o EJS

O conteúdo do arquivo `app.js` deve ser atualizado para isto:

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

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

app.get('/exemplo', function (requisicao, resposta) {
    resposta.render("exemplos/app");
});

app.listen(3000, function () {
    console.log("servidor no ar");
});
```



Trabalhando com o EJS

Analise o seguinte código:

```
app.get('/exemplo', function (requisicao, resposta) {  
    resposta.render("exemplos/app");  
});
```

O parâmetro do método `render()` especifica o arquivo `app.ejs`, presente na pasta `exemplos`, abaixo de `views`. Este é o mecanismo aplicado pelo EJS.

Mais informações: <http://www.embeddedjs.com/>



Utilizando o nodemon

Cada vez que realizamos uma alteração no arquivo `app.js`, é necessário reiniciar o servidor. Uma vírgula que seja alterada requer essa tarefa.

O Node.js pode ser configurado para monitorar as alterações realizadas no código e proceder com a atualização do servidor, de forma mais dinâmica. Para isso, podemos usar o componente chamado `nodemon`.

Sua instalação pode ser realizada por esta instrução:

```
npm install -g nodemon
```



Utilizando o nodemon

A partir deste ponto, a execução da aplicação pode ser realizada por meio deste comando:

```
nodemon app.js
```

Verifique que, a cada alteração realizada no código, o servidor é reiniciado. Basta atualizar a página para ver as alterações refletidas.



Trabalhando com rotas, views e controllers

Em uma aplicação real, as partes de um aplicação se juntam para compor um funcionamento comum. Com o Express não é diferente.

Vamos entender como gerenciar rotas e controllers, com a finalidade de montar uma aplicação mais modular. Siga os passos adiante:

1. Instale o módulo **express-load** (usado para otimizar o relacionamento entre rotas, controllers e views):

```
npm install express-load --save
```



Trabalhando com rotas, views e controllers

2. Atualize o arquivo **app.js** (nossa aplicação):

```
var express = require('express');
var load = require('express-load');
var app = express();

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

app.get('/exemplo', function (requisicao, resposta) {
  console.log("monitorando a aplicação");
  resposta.render("exemplos/app");
});
```



Trabalhando com rotas, views e controllers

```
load('models')
.then('controllers')
.then('routes')
.into(app);

app.listen(3000, function () {
  console.log("servidor no ar");
});
```

3. Crie as pastas **models** e **controllers** no projeto:



Trabalhando com rotas, views e controllers

4. Na pasta **routes**, crie o arquivo **home.js**:

```
module.exports = function (app) {
  var home = app.controllers.home;
  app.get('/', home.index);
};
```

5. **app.controllers.home** se refere a **controllers/home.js**. Crie este arquivo:

Trabalhando com rotas, views e controllers

```
module.exports = function (app) {
  var HomeController = {
    index: function (requisicao, resposta) {
      resposta.render('home/index');
    }
  };
  return HomeController;
};
```

6. Na pasta **views**, crie a pasta **home** e, dentro dela, o arquivo **index.ejs**:

Trabalhando com rotas, views e controllers

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Controle de Acesso</title>
</head>
<body>
    <header>
        <h1>Controle de Acesso</h1>
        <h4>Fornecer o código de acesso</h4>
    </header>
```



Trabalhando com rotas, views e controllers

```
<section>
    <form action="/intro" method="post">
        Código:
        <br/>
        <input type="text" name="codigo" placeholder="Código">
        <br>
        <button type="submit">Enviar</button>
    </form>
</section>
<footer>
    <small>Sistema de Controle de acesso - Express</small>
</footer>
</body>
</html>
```



Trabalhando com rotas, views e controllers

Ao executar `app.js` e chamar a URL `localhost:3000`, o resultado é este:

Controle de Acesso

Fornecer o código de acesso

Código:

Sistema de Controle de acesso - Express



Trabalhando com rotas, views e controllers

Explicação:

1. A instrução...

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

...define uma referência ao módulo **express-load**, usado para estabelecer a sequência de componentes usados na arquitetura MVC.

2. A linha abaixo...

```
app.set('views', __dirname + '/views');
```



Trabalhando com rotas, views e controllers

...apesar de dispensável, define a pasta `/views` como sendo a raiz das páginas `.ejs`. É uma boa prática incluir essa instrução nos projetos;



Trabalhando com rotas, views e controllers

3. A instrução...

```
app.use(express.static(__dirname + '/public'));
```

...estabelece a pasta `/public` como sendo a pasta padrão onde os arquivos estáticos, como estilos `css` ou arquivos `javascript` auxiliares, devem ser colocados.



Trabalhando com rotas, views e controllers

4. O bloco de código a seguir...

```
load('models')
  .then('controllers')
  .then('routes')
  .into(app);
```

...define a sequência de pastas que deve ser acessada na execução do projeto:

- As entidades na pasta **models** são carregadas para a memória;
- Em seguida, os arquivos presentes na pasta **controllers** são carregados; e
- No final, os arquivos na pasta **routes** são carregados.



Trabalhando com rotas, views e controllers

Em outras palavras:

- As rotas são os primeiros itens a serem acessados, já que são usados na URL.
- As rotas executam os controllers.
- Quando presentes, os controllers manipulam os models.
- Por último, as views são renderizadas e apresentadas para o usuário.



Trabalhando com rotas, views e controllers

5. O arquivo **routes/index.js** define uma rota:

```
module.exports = function (app) {
    var home = app.controllers.home;
    app.get('/', home.index);
};
```

Aqui, o controller **home**, presente no arquivo **controllers/home.js**, é executado, e seu resultado é processado quando o usuário chamar a URL a partir da raiz. É o que estabelece a função **get()**, referenciando a rota **'/'**.

O segundo parâmetro da função **get()** - **home.index** - indica o action no controller a ser executado.



Trabalhando com rotas, views e controllers

6. O controller, no arquivo **controllers/home.js**...

```
module.exports = function (app) {
    var HomeController = {
        index: function (requisicao, resposta) {
            resposta.render('home/index');
        }
    };
    return HomeController;
};
```

...define um action chamado **index**, cuja função é renderizar a view **home/index**, presente no arquivo **home/index.ejs**.



Trabalhando com rotas, views e controllers

Opcionalmente, é possível enviar informações do controller para a view, por meio de objetos no formato JSON. Considere o exemplo modificado:

```
module.exports = function (app) {
  var HomeController = {
    index: function (requisicao, resposta) {
      resposta.render('home/index', { titulo: 'Exemplo Express' });
    }
  };
  return HomeController;
};
```



Trabalhando com rotas, views e controllers

Na view `index.ejs`, representada no controller, podemos acessar a propriedade `titulo` do objeto enviado no método `render()`. O acesso é realizado conforme o exemplo a seguir:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title><%= titulo %></title>
</head>
<body>
  ...
</body>
```



Trabalhando com rotas, views e controllers

Este mecanismo é o ponto de partida para uma aplicação. As rotas podem definir rotas tanto por meio do método `get()` como método `post()`, ou outros verbos que se fizerem necessários.

Na elaboração dos projetos do curso, teremos a oportunidade de aplicar outros recursos, como gerenciamento de sessão, acesso a componentes de formulários, dentre outras funcionalidades.

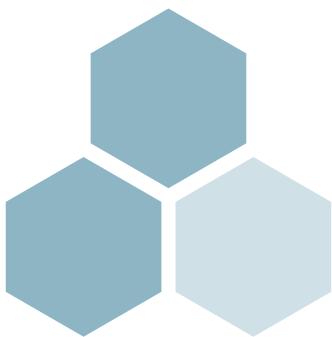
[Elaborar o Projeto 01](#)
[Elaborar o Projeto 02](#)



3

Criando e consumindo banco de dados com MongoDB

Willian Moraes
305.800



Conceitos de NoSQL

Um banco de dados NoSQL é essencialmente um banco de dados que não possui entidades relacionadas por meio de chaves primárias ou estrangeiras. Ou seja, este conceito é aplicado a banco de dados não relacional.

Não se aplicam instruções SQL tradicionais para estes tipos de banco de dados.

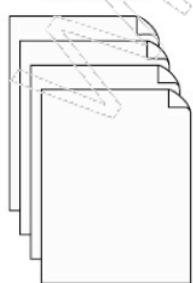
O armazenamento das informações também não é realizado em tabelas, e não são considerados campos ou registros.



Conceitos de NoSQL

A estrutura de um banco de dados NoSQL, especificamente o MongoDB, é a seguinte:

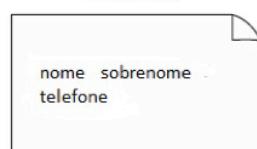
Collection



Document



Schema



Path

```
sobrenome: {  
  type: String,  
  required: true  
}
```



Conceitos de NoSQL

Na imagem anterior:

- **Collection:** Conjunto de documentos.
- **Document:** Contém as informações propriamente ditas. Essas informações seguem uma estrutura definida pelo Schema.
- **Schema:** Cada Schema é constituído por um conjunto de Paths (definições).
- **Path:** Definição dos dados presentes no documento.



Conceitos de NoSQL

Uma característica importante em bancos de dados NoSQL é que os documentos não precisam necessariamente possuir as mesmas informações, ou seja, cada linha (documento) pode ter um Schema distinto.

Exemplo de uma coleção com dois documentos:



Conceitos de NoSQL

```
{  
    nome: "Jose",  
    sobrenome: "Clemente",  
    telefone: "(11)3254-2200",  
    _id : ObjectId("52279effc62ca8b0c1000007")  
},  
{  
    nome: "Sofia",  
    email: "sofia@impacta.com",  
    celular: "(11)99599-9999",  
    _id : ObjectId("9674aacd55ff897420821409")  
}
```



Conceitos de NoSQL

Observe que o MongoDB define um atributo chamado `_id`, cujo valor é exclusivo para cada documento, e que os documentos não necessitam da mesma estrutura dentro de uma coleção.



Instalando o banco de dados

O primeiro passo é instalar o banco de dados. Ele pode ser obtido em <https://www.mongodb.com/>.

Por padrão (no Windows) os bancos de dados ficam armazenados na pasta **C:\data\db**. Devemos criar essa pasta antes de iniciar.

É possível definir outra pasta, desde que, ao iniciar o serviço, a especifiquemos.

Para iniciar o serviço, devemos digitar o comando abaixo na linha de comandos:

`mongod`



Instalando o banco de dados

Para iniciar o servidor referenciando uma pasta diferente, devemos executar este comando:

`mongod --dbpath <caminho escolhido>`

Exemplo:

`mongod --dbpath D:\Projetos\mongodb\dados`

Para o comando acima funcionar corretamente, é necessário que exista a pasta **D:\Projetos\mongodb\dados**.

Por padrão, o MongoDB é executado na porta **27017**.



Criando e acessando banco de dados

Com o servidor em execução, abrimos outra instância do terminal e executamos este comando:

```
mongo
```

Após essa execução, teremos um terminal cliente para executar instruções pertinentes ao banco de dados, como criar e manipular bancos de dados. O terminal do MongoDB possui o símbolo “>”.

Vamos executar algumas instruções e testar o banco de dados instalado.

Treinamentos
IMPACTA

Criando e acessando banco de dados

1. Exibindo todos os bancos de dados:

```
>show databases  
ou  
>show dbs
```

2. Criando um banco de dados: Na verdade, nós executamos o mesmo comando para criar e para tornar um banco de dados ativo. Se ele não existir, será criado:

```
>use db exemplo
```

O banco de dados **db exemplo** se tornará ativo, mas não existe de fato.

Treinamentos
IMPACTA

Criando e acessando banco de dados

3. Listando as coleções:

```
>show collections
```

Nada será mostrado, uma vez que o banco de dados ativo não possui nenhum documento. Quando criarmos um documento ou coleção, este será criado.

4. Inserindo um novo documento no banco:

```
>db.clientes.insert({ nome: "Impacta", telefone : "3254-2200" })
```

Após a execução dessa instrução, o banco de dados é criado, juntamente com a coleção **clientes**. Execute **show collections** para visualizar!



Criando e acessando banco de dados

5. Listando os documentos:

```
>db.clientes.find()
```

Exibe os documentos contidos na coleção **clientes**.

6. Listando os documentos de forma estruturada:

```
>db.clientes.find().pretty()
```

7. Atualizando um documento:

```
>db.clientes.update({nome: "Impacta"},{url: "impacta.com.br"})
```



Criando e acessando banco de dados

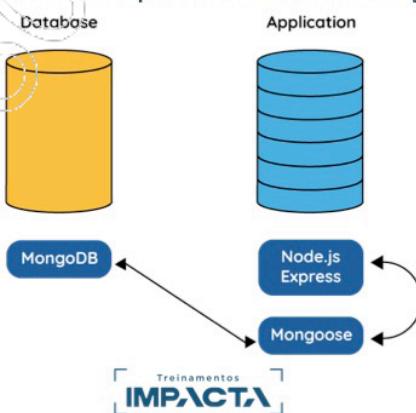
Substitui o atributo **nome** (quando encontrado) pelo atributo **url**.

Estes são alguns exemplos de utilização do banco de dados. Nos próximos passos, veremos como manipular o banco de dados por meio de uma aplicação baseada no Express.



Usando o Mongoose

O **Mongoose** é um módulo do Node.js que facilita as tarefas de acesso ao banco de dados. Em vez de acessá-lo diretamente, o Mongoose representa uma camada que abstrai toda a complexidade pertinente ao banco. Seu mecanismo é representado na imagem abaixo:



Usando o Mongoose

Tendo o MongoDB instalado e o serviço iniciado, o Mongoose é capaz de criar o banco de dados, caso não exista. Não há necessidade de interferir diretamente no banco de dados.

Para instalar o Mongoose na aplicação, usamos este comando:

```
npm install --save mongoose
```

Lembrando que o atributo **--save** inclui o módulo no arquivo **package.json** para futuras instalações, quando necessário.



Usando o Mongoose com Models

O elemento central no desenvolvimento de um banco de dados é modelo, ou **Schema**, um tipo especial de código JavaScript no qual definimos os elementos no banco de dados.

A forma geral para definição de um elemento no documento é esta:

```
elemento: {type: Tipo [, parametros adicionais ]}
```

Exemplo:

```
nome: {type: String}
```



Usando o Mongoose com Models

Os tipos usados na definição de um Schema são:

- **String**: Qualquer cadeia de caracteres no padrão UTF-8.
- **Number**: Qualquer tipo numérico. Não temos como especificar se é inteiro ou double, apenas Number.
- **Date**: Retornado como ISODate no MongoDB.
- **Boolean**: Retorna True ou False.



Usando o Mongoose com Models

Os tipos usados na definição de um Schema são:

- **Buffer**: Representa informações binárias, como imagens, por exemplo.
- **Mixed**: Qualquer tipo de dado.
- **Array**: Uma coleção de elementos.
- **ObjectId**: Qualquer identificador diferente de _id. Normalmente usado para referenciar o valor do _id em outros documentos.



Usando o Mongoose com Models

O exemplo a seguir ilustra a definição de um Schema:

```
var cliente = new mongoose.Schema({  
    nome: String,  
    endereco: String,  
    avaliacao: Number,  
    atividades: [String]  
});
```

Ou



Usando o Mongoose com Models

```
var Schema = require('mongoose').Schema;  
  
var cliente = Schema({  
    nome: String,  
    endereco: String,  
    avaliacao: Number,  
    atividades: [String] //array de String  
});
```



Usando o Mongoose com Models

É possível adicionarmos subdocumentos a um documento. Exemplo:

```
var schemaAtividade = new mongoose.Schema({  
    descricao: { type: String },  
    duracao: Number  
});  
  
var cliente = Schema({  
    nome: { type: String, required: true },  
    endereco: String,  
    avaliacao: { type: Number, "default": 0, min: 0, max: 5 },  
    atividades: [schemaAtividade] //array de SchemaAtividade  
});
```



Usando o Mongoose com Models

Na definição de subdocumentos, é importante que a definição do subdocumento ocorra antes do documento principal que o utiliza.



Acessando o MongoDB com Node.js

Considerando o projeto criado com base no Express e o arquivo `app.js`, vamos incluir as instruções destacadas para referenciar o Mongoose:

```
var express = require('express');
var load = require('express-load');
var app = express();

var mongoose = require('mongoose');
global.db = mongoose.connect('mongodb://localhost:27017/dbusuarios');

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));
```



Acessando o MongoDB com Node.js

```
load('models')
.then('controllers')
.then('routes')
.into(app);

app.listen(3000, function () {
  console.log("servidor no ar");
});
```



Acessando o MongoDB com Node.js

Observe que, na função `connect()`, estamos referenciando um banco de dados chamado `dbusuarios`. Este será criado pelo Mongoose.

Agora, na pasta `models`, crie o arquivo `usuarios.js`:

```
module.exports = function (app) {
    var Schema = require('mongoose').Schema;

    var usuario = Schema({
        nome: { type: String, required: true},
        email: { type: String, required: true }
    });
    return db.model('usuarios', usuario);
};
```



Acessando o MongoDB com Node.js

No controller `home.js`, realize as alterações destacadas:

```
module.exports = function (app) {
    var Usuario = app.models.usuario;
    var HomeController = {
        index: function (requisicao, resposta) {
            //definindo um novo usuário
            var nome = "usuario novo";
            var email = "email@impacta.com";
            var usuario = { "nome": nome, "email": email };

            //definindo a resposta
            var resultado;
```



Acessando o MongoDB com Node.js

```
Usuario.create(usuario, function (erro, usuario) {
    if (erro) {
        resultado = "Ocorreu um erro ao incluir usuário";
    }
    else {
        resultado = "Usuário incluído com sucesso";
    }
});

resposta.render('home/index', { titulo: 'Exemplo Express',
                                resultado: resultado});
}

return HomeController;
};
```



Acessando o MongoDB com Node.js

Ao iniciar a aplicação `app.js`, execute-a no browser para que o controller possa ser processado. Verifique no console do banco de dados se a coleção `usuários` foi criada e se o registro foi incluído corretamente.



Acessando o MongoDB com Node.js

No código, podemos escrever alguns manipuladores de eventos para monitorar o processo de conexão. Podemos considerar três eventos no ciclo de vida do Mongoose: **connected**, **error** e **disconnected**.

Sua utilização está apresentada no código a seguir:

Acessando o MongoDB com Node.js

```
var mongoose = require('mongoose');
global.db = mongoose.connect('mongodb://localhost:27017/dbusuarios');

mongoose.connection.on('connected', function () {
    console.log('=====Conexão estabelecida com sucesso=====');
});
mongoose.connection.on('error', function (err) {
    console.log('=====Ocorreu um erro: ' + err);
});
mongoose.connection.on('disconnected', function () {
    console.log('=====Conexão terminada=====');
});
```

Acessando o MongoDB com Node.js

Elaborar o Projeto 03

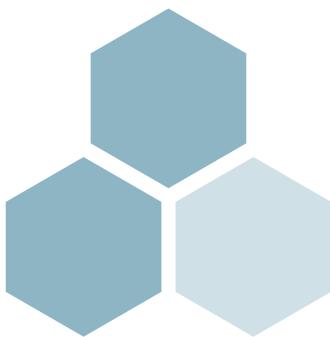


Treinamentos
IMPACTA



4

Desenvolvimento de Web services com Node.js, Express e MongoDB



Editora
IMPACTA

Definindo Web services REST

A sigla **REST** significa **Representational State Transfer**. REST é um padrão de Web services baseados em verbos HTTP (**GET**, **POST**, **PUT** e **DELETE**).

Mas o que é um Web service?

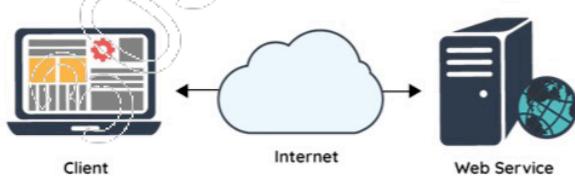
Vamos imaginar o seguinte cenário:

Um cliente efetua uma compra em um supermercado. O pagamento é realizado com seu cartão de crédito. O sistema do supermercado não conhece os dados do cartão, então encaminha a solicitação ao sistema da administradora de cartão de crédito. O sistema da administradora responde ao sistema do supermercado com um dos possíveis resultados: **Saldo insuficiente**, **Transação aceita**, entre outros.



Definindo Web services REST

A imagem abaixo ilustra esse mecanismo:



Nessa imagem, o sistema da administradora é o nosso Web service.



Definindo Web services REST

Um Web service é um sistema cujo acesso ocorre de forma agnóstica (ou seja, independe da plataforma usada para o desenvolvimento do sistema que o consome).

Com o Node.js, é possível tanto desenvolver um Web service para ser consumido por outras aplicações quanto consumir um Web service criado em outra linguagem.

Neste capítulo, veremos como criar serviços com Node.js.



Criando serviços

A criação de um serviço (Web service) com Node.js será ilustrada por meio de um exemplo.

Considere que um projeto chamado **apiRest** seja criado, e os módulos necessários, adicionados:

```
express apiRest --ejs  
cd apiRest  
npm install  
npm install body-parser -save  
npm install express-load --save  
npm install mongoose --save
```



Criando serviços

Criamos um arquivo chamado **contato.js**, na pasta **models** do projeto.

```
module.exports = function (app) {
  var Schema = require('mongoose').Schema;
  var contato = Schema({
    cpf: String,
    nome: String,
    telefone: String
  });
  return db.model('contatos', contato);
};
```



Criando serviços

No arquivo **app.js**, que é nossa aplicação, escrevemos os métodos a serem acessados (consumidos) pelo sistema externo. Ou seja, definimos as funções que serão acessadas em forma de serviço.

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

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

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```



Criando serviços

```
var mongoose = require('mongoose');
global.db = mongoose.connect('mongodb://localhost:27017/contatos');

load('models').into(app);

var Contato = app.models.contato;

app.listen(3000, function () {
  console.log('ok');
});
```



Criando serviços

As funções do serviço serão estas:

```
//método do serviço
app.get('/', function (request, response) {
response.send('Servidor no ar');
});
app.get('/contatos', function (request, response) { });

app.get('/contatos/:id', function (request, response) { });

app.post('/contatos', function (request, response) { });

app.put('/contatos', function (request, response) { });

app.delete('/contatos', function (request, response) { });
```



Criando serviços

Os nomes das funções correspondem aos verbos HTTP usados na sua chamada. Como exemplo, uma aplicação para cada método, tomando como base o Mongoose e o model **contatos.js**, é dada a seguir:



Criando serviços

- Função `get()`: Obtendo a lista de contatos no formato JSON:

```
app.get('/contatos', function(request, response) {
  Contato.find(function(erro, contatos) {
    if (erro) {
      response.json(erro);
    } else {
      response.json(contatos);
    }
  });
});
```



Criando serviços

- Função **get()**: Obtendo um contato cujo id é informado na URL:

```
app.get('/contatos/:id', function (request, response) {  
    var id = request.query.id;  
  
    Contato.findById(id, function (erro, contato) {  
        if (erro) {  
            response.json(erro);  
        }  
        else {  
            response.json(contato);  
        }  
    });  
});
```



Criando serviços

- Função **post()**: Incluindo um novo contato:

```
app.post('/contatos', function (request, response) {  
    var cpf = request.body.cpf;  
    var nome = request.body.nome;  
    var telefone = request.body.telefone;  
  
    var usuario = {  
        'cpf': cpf,  
        'nome': nome,  
        'telefone': telefone  
    };
```

Continua...



Criando serviços

```
Contato.create(usuario, function (erro, contato) {
  if (erro) {
    response.json(erro);
  }
  else {
    response.json(contato);
  }
});
```



Criando serviços

- Função `put()`: Alterando um contato pelo id:

```
app.put('/contatos/:id', function (request, response) {
  var id = request.query.id;

  Contato.findById(id, function (erro, contato) {
    if (erro) {
      response.json(erro);
    }
    else {
      var contato_upd = contato;
      contato_upd.cpf = request.params.cpf;
      contato_upd.nome = request.params.nome;
      contato_upd.telefone = request.params.telefone;
```

Continua...



Criando serviços

```
contato_upd.save(function (erro, contato) {
  if (erro) {
    response.json(erro);
  }
  else {
    response.json(contato);
  }
});
response.json(contato);
});
```



Criando serviços

- Função **delete()**: Removendo um contato a partir do id:

```
app.delete('/contatos/:id', function (request, response) {
  var id = request.query.id;

  Contato.remove(id, function (erro, contato) {
    if (erro) {
      response.json(erro);
    }
    else {
      response.send('removido');
    }
  });
});
```



Criando serviços

Elaborar o Projeto 04



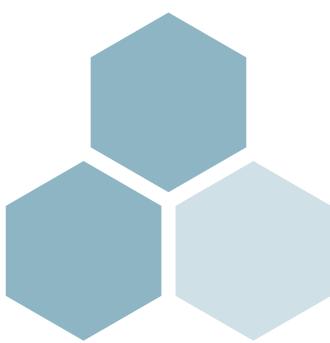
Treinamentos
IMPACTA



5

Desenvolvimento com AngularJs

Willians Moraes
305.800°



Conceitos

AngularJS é uma tecnologia para aprimorar no desenvolvimento do front-end da aplicação, incluindo funcionalidades AJAX, acesso a serviços, manipulação de componentes, entre outras funcionalidades, por meio da configuração de módulos disponíveis e prontos para uso.

A utilização do AngularJS consiste essencialmente na inclusão de uma referência, que pode ser obtida no site do AngularJS (<https://www.angularjs.org/>) ou on-line, via CDN (Content Delivery Network).

Neste capítulo, vamos entender o mecanismo do AngularJS e como ele pode ser adicionado a uma aplicação baseada no MEAN.



A arquitetura MVC

A arquitetura MVC consiste em um modelo de projetos no qual as camadas da aplicação são separadas, de forma a serem manipuladas de forma independente uma da outra.

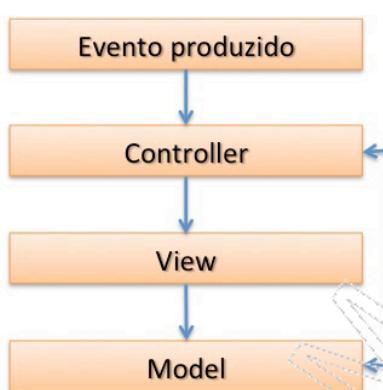
O termo **MVC** significa **Model-View-Controller**:

- **Model:** Camada responsável por manter os dados.
- **View:** Exibe dados para o usuário.
- **Controller:** Contém as regras aplicadas na interação entre o model e a view.



A arquitetura MVC

De uma forma consistente, podemos considerar o esquema abaixo:



A arquitetura MVC

Quando um evento é disparado na interface gráfica (um clique de botão, por exemplo), uma ação no controller é executada, buscando e/ou enviando dados da view para o model.

Em uma aplicação com AngularJS:

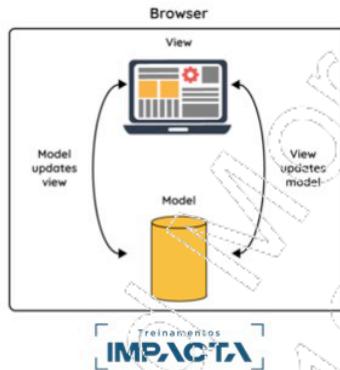
- A camada **View** representa o conjunto de elementos HTML, a parte que interage com o usuário.
- A camada **Controller** representa um conjunto de instruções JavaScript responsável por enviar e receber dados da View, de forma unidirecional (one-way data binding) ou bidirecional (two-way data binding).



A arquitetura MVC

- A camada **Model** corresponde aos objetos trocados entre o controller e a view. Esta camada também é formada por instruções JavaScript.

Neste sentido, a view atualiza o model por meio da intercepção do controller e vice-versa:



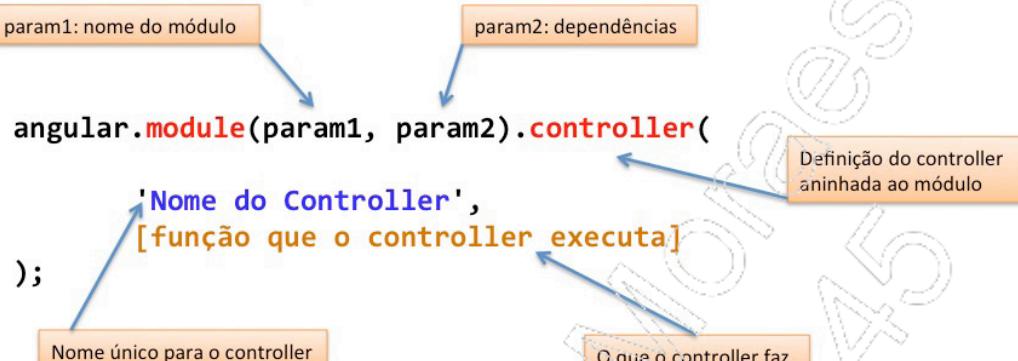
Controllers, Models e Views

Em linhas gerais, a estrutura do AngularJS consiste:

- Na declaração de um módulo.
- A partir do módulo, a definição de um controller.
- No controller, a implementação da função a ser executada (também chamada de **action**, como no controller do Express).

Controllers, Models e Views

Na estrutura abaixo, ilustramos essa definição:



Controllers, Models e Views

Para iniciar a utilização do Angular, consideraremos estas etapas:

- Referenciar a biblioteca do Angular:

```
<script src="js/angular.min.js"></script>
```

- Informar o Angular que a página em questão é uma aplicação (app), por meio da diretiva **ng-app**:

```
<html ng-app>
```

Este recurso diz ao Angular que tudo o que estiver aninhado a esta tag será parte da aplicação.

Controllers, Models e Views

Isso significa que não existe a necessidade de colocá-la apenas na tag HTML. Podemos considerar um bloco dentro de uma div como parte da aplicação, se não for importante que todo o conteúdo o seja.

- Para o vínculo unidirecional, considere como exemplo o model associado a um campo de entrada:

```
<input type="text" ng-model="entrada">
```

- Ao informar um texto, o resultado será refletido na view:

```
<h1>Você informou o texto: {{entrada}}</h1>
```

Treinamentos
IMPACTA

Controllers, Models e Views

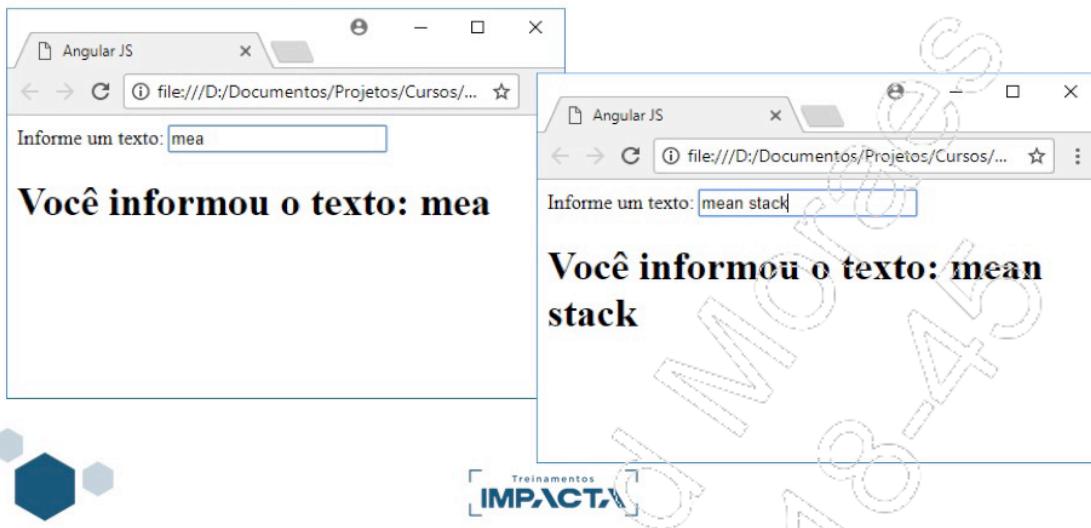
Nesse caso, não consideraremos nenhum controller, e sim um vínculo entre um campo de entrada e uma apresentação. O código completo será este:

```
<!DOCTYPE html>
<html ng-app>
<head>
  <title>Angular JS</title>
</head>
<body>
  Informe um texto:
  <input type="text" ng-model="entrada"><br/>
  <h1>Você informou o texto: {{entrada}}</h1>
  <script src="js/angular.min.js"></script>
</body>
</html>
```

Treinamentos
IMPACTA

Controllers, Models e Views

Como resultado, teremos isto:



Controllers, Models e Views

Temos, também, a possibilidade de usar a diretiva **ng-bind** para apresentar o valor do model:

```
<h1>Você informou o texto: {{entrada}}</h1>
<span ng-bind="entrada"></span>
```

A grande vantagem do uso dessa diretiva é que, caso ocorra alguma demora na carga da página, as chaves não serão exibidas (sim, elas são exibidas como se fossem conteúdo do HTML, até que o Angular entre em ação e resolva o vínculo).

O uso de **ng-bind** é, então, mais elegante comparado ao uso de chaves duplas.



Controllers, Models e Views

Vamos, agora, definir a aplicação como um módulo:

```
<html ng-app="appModule">
```

Nesse caso, definimos o módulo em um código JavaScript:

```
angular.module('appModule',[]);
```

Definiremos um controller a partir do elemento `<body>`:

```
<body ng-controller="appController">
```



Controllers, Models e Views

Com isso, complementamos nosso módulo para incluir o controller:

```
angular.module('appModule',[])
  .controller('appController', [function(){
    //conteúdo do controller
  }]);

```

Observe que incluímos a função do controller entre colchetes ([]), para viabilizar a minificação, quando houver. Esses colchetes são dispensáveis, caso não haja preocupação com a minificação.



Diretivas

As diretivas são as responsáveis por agregar funcionalidades baseadas no AngularJS à página HTML, tornando a página (ou partes dela) um componente.

Tivemos a oportunidade de ver algumas diretivas:

ng-app
ng-controller
ng-model
ng-bind

Estudaremos outras diretivas ao longo do capítulo. É possível, também, criarmos nossas próprias diretivas.



Estrutura do AngularJS

A partir de agora, teremos a oportunidade de conhecer várias estruturas do AngularJS, como funções, objetos, arrays e muitos outros.

Esses tópicos serão importantes na definição de uma aplicação mais robusta.

Para estes exemplos, consideraremos um módulo clamado **appModule**, ou seja, teremos a definição **ng-app="appModule"** pronta.

Além disso, teremos, também, o conteúdo HTML escrito em um elemento body com um controller: **<body ng-controller="Principal as ctrl">**.



Estrutura do AngularJS

1. Funções, variáveis e eventos (JavaScript)

```
angular.module('appAngular', [])
    .controller('Principal', [function () {
        var self = this;
        self.mensagem = 'Vindo do Controller';
        self.alterarMensagem = function () {
            self.mensagem = "Mensagem Alterada";
        };
    }]);

```

Nesse controller, definimos uma propriedade (**mensagem**) e uma função (**alterarMensagem()**), a ser executada assincronamente.



Estrutura do AngularJS

1. Funções, variáveis e eventos (HTML)

```
Visualizar Mensagem: {{ctl.mensagem}}
<br />
<button ng-click="ctl.alterarMensagem()">
    Alterar Mensagem
</button>
```

A função **alterarMensagem()** é chamada quando o botão é clicado (veja o uso da diretiva **ng-click**).



Estrutura do AngularJS

2. Trabalhando com arrays (JavaScript)

```
angular.module('appAngular', [])
    .controller('Principal', [function () {
        var self = this;
        self.nomes = [
            'Ademar Torres',
            'Cecilia Maria',
            'Adoniran Barbosa'
        ];
    }]);
});
```

A definição do array segue o padrão JavaScript,



Estrutura do AngularJS

2. Trabalhando com arrays (HTML)

```
<h2>Lista de Alunos</h2>
<div ng-repeat="nome in ctl.nomes">
    {{nome}}
</div>
```

Por meio da diretiva **ng-repeat**, os elementos do array são expostos um a um, com o conteúdo referenciado pela variável **nome**.



Estrutura do AngularJS

3. Arrays de objetos (JavaScript)

```
angular.module('appAngular', [])
    .controller('Principal', [function () {
        var self = this;
        self.alunos = [
            { nome: 'Ademar Torres', curso: 'NodeJS' },
            { nome: 'Cecilia Maria', curso: 'Psicologia' },
            { nome: 'Adoniran Barbosa', curso: 'Turismo' }
        ];
    }]);

```

Aqui, cada elemento do array é um objeto especificado no formato JSON.



Estrutura do AngularJS

3. Arrays de objetos (HTML)

```
<h2>Lista de Alunos</h2>
<div ng-repeat="aluno in ctl.alunos">
    Nome: {{aluno.nome}}
    <br/> Curso: {{aluno.curso}}
    <br />
</div>
```

Analogamente, cada elemento do array é exposto pela diretiva **ng-repeat**, e o elemento expõe, também, as propriedades presentes no objeto, aqui sendo o elemento do array.



Estrutura do AngularJS

4. Utilização de formulários (JavaScript)

```
angular.module('appAngular', [])
  .controller('Principal', [function () {
    var self = this;
    //modo implícito

    self.submit = function () {
      alert('Dados do formulário: \n' +
        self.aluno.nome + '\n' + self.aluno.curso);
    };
}]);
```

O objeto aluno (**self.aluno**) é criado no momento em que o formulário é submetido.



Estrutura do AngularJS

4. Utilização de formulários (HTML)

```
<h2>Formulários</h2>
<form ng-submit="ctl.submit()">
  <input type="text" ng-model="ctl.aluno.nome" />
  <br />
  <input type="text" ng-model="ctl.aluno.curso" />
  <br />
  <input type="submit" value="Submit" />
</form>
```

O termo **aluno**, repetido nos dois modelos (**ng-model**), é criado pelo Angular sob forma de um objeto e enviado para o controller.



Estrutura do AngularJS

Muitas vezes é necessário que os dados do formulário sejam devidamente validados visando sua consistência.

No AngularJS, temos diversos critérios de validação. Alguns serão apresentados a seguir:



Estrutura do AngularJS

Alguns validadores:

- **required**: Marca o campo como obrigatório (html5).
- **ng-required**: Marca um campo como obrigatório em função de um valor booleano no controlador.
- **ng-minlength**: Define o tamanho mínimo do campo de entrada.
- **ng-maxlength**: Define o tamanho máximo do campo de entrada.
- **ng-pattern**: Utiliza expressão regular para validação.



Estrutura do AngularJS

Alguns estados de formulários:

- **\$invalid**: Estado definido quando alguma validação falhar.
- **\$valid**: Estado definido quando o formulário for válido.
- **\$pristine**: Estado definido quando o formulário é carregado, antes de alguma interação.
- **\$dirty**: O inverso de **\$pristine**.



Estrutura do AngularJS

Alguns estados de formulários:

- **\$error**: Armazena o estado do formulário com erros.
- **\$invalid**: ng-invalid
- **\$valid**: ng-valid
- **\$pristine**: ng-pristine
- **\$dirty**: ng-dirty



Estrutura do AngularJS

5. Validação de formulários (CSS)

```
<style>
  .nome.ng-valid {
    color: green;
  }
  .nome.ng-dirty.ng-invalid-required {
    background-color: red;
  }
  .nome.ng-dirty.ng-invalid-minlength {
    color: orange;
  }
</style>
```

Este CSS define configurações de acordo com o estado de cada controle indicado.

Estrutura do AngularJS

5. Validação de formulários (HTML)

```
<form ng-submit="ctl.submit()" name="alunosForm">
  Nome:<br />
  <input type="text" class="nome" name="anome"
    ng-model="ctl.aluno.nome" required ng-minlength="4" />
  <span ng-show="alunosForm.anome.$error.required">
    Campo obrigatório
  </span><br/>
  Curso:<br />
  <input type="text" ng-model="ctl.aluno.curso" /><br />
  <input type="submit" value="Enviar" />
</form>
```

Estrutura do AngularJS

5. Validação de formulários (HTML)

Nesta página, apresentamos as regras de validação, e as configurações CSS serão aplicadas de acordo com o estado dos formulários.



Usando serviços

1. Uso de serviços comuns (JavaScript)

```
//a ordem da injeção de dependência é importante
angular.module('appAngular', [])
    .controller('Principal', ['$location', '$window',
        function ($location, $window) {
            var self = this;

            self.mostrarLog = function () {
                var url = $location.url();
                var absUrl = $location.absUrl();
                var protocol = $location.protocol();
```



Usando serviços

1. Uso de serviços comuns (JavaScript)

```
var mensagem = 'URL: ' + url + '\nURL Absoluto: ' +
               absUrl + '\nPROTOCOLO: ' +
               protocol;
$window.alert(mensagem);
});
```

Neste código, os serviços padrão do AngularJS foram adicionados como dependência do controller. Observe que a ordem deve ser a mesma usada como parâmetros da função do controller.



Usando serviços

2. Definindo serviços: factory (JavaScript)

```
appAngular.factory('MathFactory', function () {
  var factory = {};
  factory.multiplicar = function (a, b) {
    return a * b;
  }
  return factory;
});
```

Usamos o componente **factory** quando desejamos entregar um objeto para a aplicação, tanto para um serviço como para um controller. Neste exemplo, o factory será injetado no serviço que definiremos.



Usando serviços

2. Definindo serviços: service (JavaScript)

```
appAngular.service('MathService', function (MathFactory) {
    this.calcularQuadrado = function (a) {
        return MathFactory.multiplicar(a, a);
    }
});
```

Observe que aqui o **service** entrega uma função, ou um método, obtido como execução do componente **factory** (MathFactory) injetado pelo AngularJS.



Usando serviços

2. Definindo serviços: controller (JavaScript)

```
appAngular.controller('Principal', function ($scope, MathService) {
    $scope.calcularQuadrado = function () {
        $scope.resultado = MathService.calcularQuadrado($scope.valor);
    }
});
```

O controller tem por tarefa interagir com a camada de apresentação.

Quando desejarmos criar objetos reutilizáveis, usamos o **factory**

Se a necessidade for consumir o objeto repassado e gerar um action, criamos um **service**. Por último, o controller entrega a resposta para a view.



Rotas e modelos SPA

1. Configurando as rotas (JavaScript)

Para trabalharmos com rotas, devemos usar a api **angular-route.min.js**, além da api tradicional.

Esta api permite referenciar a dependência **ngRoute** no módulo e, além disso, referenciar o serviço **\$routeProvider**, cuja finalidade é configurar a estrutura de rotas.

Vamos analisar os exemplos:



Rotas e modelos SPA

1. Configurando as rotas (JavaScript)

```
angular.module('appAngular', ['ngRoute'])
    .config(['$routeProvider', function ($routeProvider) {
        $routeProvider
            .when('/', {
                template: '<h2>Página inicial</h2>'
            })
            .when('/lista', {
                templateUrl: 'checkbox02.html'
            })
            .otherwise({ redirectTo: '/' })
    }])
    .controller('Principal', [function () {
```



Rotas e modelos SPA

1. Configurando as rotas (JavaScript)

```
var self = this;  
  
self.cursos = [  
    { descricao: 'JAVA', selecionado: 'SIM' },  
    { descricao: 'PHP', selecionado: 'SIM' },  
    { descricao: '.NET', selecionado: 'NAO' }  
];  
})
```



Rotas e modelos SPA

1. Configurando as rotas (HTML)

```
<div>  
  <ul>  
    <li>  
      <a href="#">Rota padrão (inicial)</a></li>  
    <li>  
      <a href="#/lista">Cursos</a> </li>  
    <li>  
      <a href="#/desconhecida">Rota inexistente</a></li>  
  </ul>  
  <div class="borda" ng-view></div>  
</div>
```



Rotas e modelos SPA

A div assinalada com a diretiva **ng-view** é usada para interpolar o conteúdo da view de acordo com a rota escolhida.

Por exemplo, quando acionamos a rota...

```
<a href="#/lista">Cursos</a> </li>
```

...estamos requisitando a view **checkbox02.html**, conforme esta instrução:

```
.when('/lista', {  
    templateUrl: 'checkbox02.html'  
})
```



Acesso a Web services com AngularJS

Uma vez tendo um serviço disponível, consumi-lo com AngularJS é bastante simples, por meio do uso do serviço **\$http**.

Os exemplos a seguir ilustram o procedimento de acesso ao serviço por meio da sua URL.

Primeiro, realizaremos a busca por uma lista de produtos e, uma vez obtida, a armazenaremos na variável **items**.

Analise os exemplos:



Acesso a Web services com AngularJS

1. Usando o serviço \$http para acessar serviço (JavaScript)

```
//HTTP GET
angular.module("appAngular", [])
    .controller('Principal', ['$http', function ($http) {
        var self = this;
        self.items = [];
        $http.get('http://127.0.0.1:7628/api/produtos')
            .then(function (response) {
                self.items = response.data;
            }, function (error) {
                alert('Erro reportado: ' + error);
            });
    }]);
});
```



Acesso a Web services com AngularJS

1. Usando o serviço \$http para acessar serviço (HTML)

```
<table>
  <tr>
    <th>ID</th>
    <th>Descrição</th>
    <th>Data Criação</th>
    <th>Preço</th></tr>
  <tbody ng-repeat="produto in ctl.items">
    <tr>
      <td>{{produto.Id}}</td>
      <td>{{produto.Descricao}}</td>
      <td>{{produto.DataCriacao}}</td>
      <td>{{produto.Preco}}</td>
    </tr></tbody></table>
```



Acesso a Web services com AngularJS

Verifique que, uma vez obtido o resultado, seu tratamento segue as convenções tradicionais do AngularJS.

No próximo exemplo, geraremos um objeto no formulário (na verdade, o Angular realizará esta tarefa para nós) e o enviaremos para o serviço como forma de incluí-lo por meio do método **HTTP POST**:

Acesso a Web services com AngularJS

2. Usando o serviço \$http para enviar um objeto (JavaScript)

```
angular.module("appAngular", [])
    .controller('Principal', ['$http', function ($http) {
        var self = this;
        self.novoProduto = {};
        self.adicionar = function () {
            $http.post('http://localhost:7628/api/produtos/',
                self.novoProduto)
            .then(function (response) {
                self.novoProduto = {};
            });
        };
    }]);

```

Acesso a Web services com AngularJS

2. Usando o serviço \$http para enviar um objeto (HTML)

```
<div>
  <form name="incluirForm" ng-submit="ctl.adicionar()">
    <input type="text" ng-model="ctl.novoProduto.Descricao" />
    <br />
    <input type="date" ng-model="ctl.novoProduto.DataCriacao" />
    <br />
    <input type="text" ng-model="ctl.novoProduto.Preco" />
    <br />
    <input type="submit" value="Adicionar" />
  </form>
</div>
```



Acesso a Web services com AngularJS

Elaborar o Projeto 05

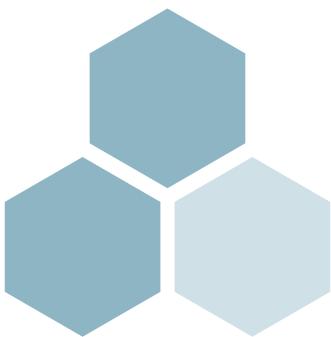




6

Conhecendo o Angular 4

Williams 305.800.
Moraes 45



Editora
IMPACTA

Conceitos

O Angular é um framework JavaScript mantido pelo Google® que permite trabalhar com aplicações SPA (Single Page App).

A versão 2 do Angular foi totalmente reestruturada e utiliza um supertipo do JavaScript, o **TypeScript**, desenvolvido pela Microsoft® e adotado amplamente pelo Angular 2.

Neste capítulo, abordaremos o Angular 4.

A diferença para o Angular 2 é que os módulos, diretivas e componentes foram reestruturados de forma a permitir uma melhor manutenção e adaptação.



Conceitos

No Angular 4, diferentemente do AngularJS, nós desenvolvemos componentes, e cada componente possui uma representação (uma view) em HTML. Nós não executamos a página HTML; é o componente que, quando carregado, gera a view com os dados definidos no componente.

Este padrão difere totalmente de uma aplicação baseada em páginas, como é o caso do AngularJS.



Criando um projeto

A criação de um projeto no Angular 4 é realizada por meio do Node.js.

Existe um projeto modelo disponível no repositório GitHub, chamado **QuickStart**. Este projeto é fornecido pela própria equipe do Angular 4 e está sempre passando por atualizações.

A boa notícia é que diversos módulos são instalados via npm, e o arquivo **package.json** pode ser atualizado de forma a permitir a geração de projetos em diferentes máquinas com as versões que usamos no nosso projeto (claro, com a possibilidade de atualizações).

Existe a possibilidade de usarmos, também, o **Angular CLI**. Neste curso, usaremos a opção do **QuickStart**.



Criando um projeto

Para criar um novo projeto Angular 4 (é necessário que o aplicativo Git esteja instalado no seu computador):

- Selecione uma pasta adequada para o projeto;
- Execute este comando:

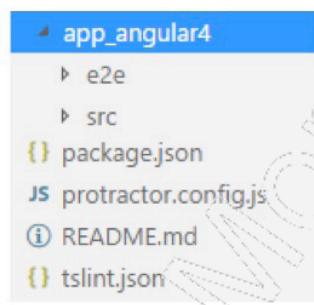
```
git clone https://github.com/angular/quickstart.git app_angular4
```

Em que **app_angular4** é o nome do projeto, escolhido arbitrariamente. Se nenhum nome for informado, o projeto se chamará **quickstart**.



Criando um projeto

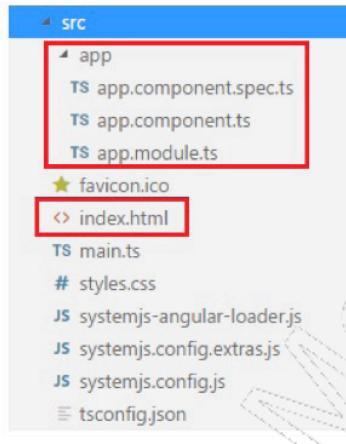
- Após clonado o projeto, teremos uma estrutura semelhante à apresentada a seguir (algumas pastas e arquivos foram omitidos):



Criando um projeto

- O arquivo criado **package.json** contém os módulos necessários para a aplicação.
- A pasta **src** é a pasta raiz da aplicação. Dentro dela temos a pasta **app** e o arquivo **index.html**.
- Temos, também, outros arquivos que serão apresentados adiante. A estrutura dessa pasta é dada a seguir:

Criando um projeto



Criando um projeto

- Verifique que, na pasta **app**, temos os arquivos **app.component.ts** e **app.module.ts**. Esses arquivos representam o componente principal e o módulo principal, respectivamente.
- Como template gerado a partir do QuickStart, eles podem ser alterados conforme nossa necessidade. Analisaremos sua utilidade na sequência.
- Para iniciar, entramos na pasta **app_angular4** que foi criada por meio do Git. Nessa pasta, executamos os comandos do Node.js para instalação dos módulos:

Criando um projeto

```
cd app_angular4  
npm install  
npm start
```

- O comando **npm start** executa um script definido no arquivo **package.json** (observe que, na propriedade **scripts**, temos um objeto com uma propriedade chamada **start**) com esse nome. A linha com tal informação é esta:

```
"start": "concurrently \"npm run build:watch\" \"npm run serve\"",
```

Treinamentos
IMPACTA

Criando um projeto

...que instrui o Angular a executar um servidor local chamado **lite-server**. Esse servidor, quando carregado, apresenta o browser com o seguinte conteúdo:



Treinamentos
IMPACTA

Criando um projeto

Como podemos ver, nada de muito atraente, mas é o ponto de partida para estudarmos o funcionamento do Angular 4.

Nos próximos tópicos, apresentaremos o fluxo de execução de uma aplicação Angular 4 e os conceitos de componentes, módulos, serviços e como eles interagem entre si.



Módulos e componentes

O ponto de partida da aplicação é o arquivo `index.html`.

Neste arquivo não definimos componentes HTML a serem renderizados na página, mas executamos um programa chamado `main.js`.

Se procurarmos este arquivo no projeto, não o encontraremos, mas é possível encontrar o arquivo `main.ts`. A extensão `.ts` se refere a **TypeScript**.

O Angular realiza um processo de **transpilação** (transformação e compilação) de um código typescript para o código javascript.

Isso porque o que é de fato executado é o javascript. A transpilação insere recursos de compatibilidade de browsers e de bibliotecas adicionais.



Módulos e componentes

Se esses componentes fossem escritos em JavaScript puro, certamente teríamos bem mais trabalho. Por isso escrevemos todas as instruções em TypeScript, bem mais simples e de alto nível.

Vamos analisar o processo de execução de uma aplicação Angular 4:

Módulos e componentes

1. Início da aplicação: arquivo index.html.

Neste arquivo, destacamos o elemento de partida, apresentado a seguir:

```
<script src="systemjs.config.js"></script>
<script>
    System.import('main.js').catch(function(err){
        console.error(err);
    });
</script>
```

Módulos e componentes

2. Execução de `main.js`.

Como já mencionado, este arquivo é uma transpilação do arquivo `main.ts`, cujo conteúdo é este:

```
import { platformBrowserDynamic } from
      '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```



Módulos e componentes

O comando `import` permite inserir elementos na aplicação, provenientes dos módulos instalados, ou a partir de componentes desenvolvidos pelo programador.

A instrução...

```
import { platformBrowserDynamic } from
      '@angular/platform-browser-dynamic';
```

...inclui o componente `platformBrowserDynamic` na aplicação para permitir a execução no browser:



Módulos e componentes

A instrução...

```
import { AppModule } from './app/app.module';
```

...insere o módulo principal definido no arquivo `app.module.ts`, disponível na pasta `app`.

3. Por sua vez, o conteúdo deste arquivo é o seguinte:



Módulos e componentes

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```



Módulos e componentes

Neste arquivo:

- Importamos os módulos **NgModule** e **BrowserModule**, ambos disponíveis na biblioteca inserida pelo Node.js.
- Importamos o componente **AppComponent** definido no arquivo **app.component.ts**.
- Configuramos o decorador **@NgModule** para:
 - Importar o módulo **BrowserModule**;
 - Declarar o componente **AppComponent**;
 - Definir **AppComponent** como o componente principal (inicial) da aplicação. O comando **bootstrap** realiza esta tarefa.



Módulos e componentes

4. O conteúdo de **app.component.ts** (novo componente inicial) é este:

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`,
})
export class AppComponent { name = 'Angular'; }
```



Módulos e componentes

Temos informações importantes neste arquivo.

O decorador **@Component** especifica dois atributos:

- **selector**: Componente a ser escrito na página HTML, em forma de elemento. Verifique que em **index.html** nós temos esta tag:

```
<my-app>Loading AppComponent content here ...</my-app>
```

- **template**: Conteúdo HTML a ser renderizado na ocasião da execução do nosso selector.

Módulos e componentes

No template, temos um elemento HTML e a interpolação de um model chamado **name**:

```
template: `<h1>Hello {{name}}</h1>`,
```

O valor desse model é definido na classe **AppComponent**:

```
export class AppComponent {
  name = 'Angular';
}
```

O resultado é visualizado no browser, como no início deste exemplo.

Módulos e componentes

5. Finalmente, a instrução...

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

...especifica o módulo **AppModule** como o módulo principal da aplicação.



Módulos e componentes

Resumindo:

1. O arquivo **index.html** é executado.
2. A partir de **index.html**, o código **main.js** é executado.
3. O código **main.js** especifica o módulo **AppModule** como o principal da aplicação.
4. O módulo **AppModule** define **AppComponent** como o componente inicial.
5. O componente **AppComponent** define o elemento **<my-app>**, devidamente sinalizado em **index.html**.



Módulos e componentes

6. O conteúdo HTML a ser renderizado em <my-app> é especificado no atributo **template**.

A partir desse princípio, detalharemos os componentes para enriquecer nossa aplicação.

Os passos serão descritos na sequência:



Módulos e componentes

- Inclua a biblioteca do Bootstrap para melhorar nossa apresentação visual (observe que escolhemos a versão do Bootstrap):

```
npm install bootstrap@3.3.7 --save
```

- No arquivo **index.html**, inclua esta configuração:

```
<link rel="stylesheet"  
      href="node_modules/bootstrap/dist/css/bootstrap.min.css">
```



Módulos e componentes

- Crie uma pasta chamada **menu** abaixo de **app**.
- Em seguida, crie a pasta **view** abaixo de **menu**.
- Nessa pasta, inclua o arquivo **menu.component.html** (verifique a nomenclatura) com o conteúdo:

```
<body>
  <h1>Página inicial</h1>
  <ul>
    <li><a href="#">Listar contatos</a></li>
  </ul>
</body>
```



Módulos e componentes

- Na pasta menu, defina o arquivo **menu.component.ts**:

```
import { Component } from '@angular/core';

@Component({
  moduleId: module.id,
  selector: 'menu',
  templateUrl: 'views/menu.component.html'
})
export class MenuComponent { }
```

O atributo **moduleId** faz parte da arquitetura do Angular 4 e o instrui a buscar o conteúdo de **templateUrl** a partir do local do componente. Se omitido, deveríamos indicar o caminho completo.



Módulos e componentes

- Abra o arquivo **app.component.ts** e realize as alterações em destaque:

```
import { Component } from '@angular/core';
import { MenuComponent } from './menu/menu.component';

@Component({
  moduleId: module.id,
  selector: 'my-app',
  template: '<menu></menu>'
})
export class AppComponent { }
```



Módulos e componentes

- Isso nos indica que o componente principal incluirá um novo componente, o **MenuComponent**.
- Esse novo componente deve ser registrado em **app.module.ts**:



Módulos e componentes

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent, MenuComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```



Módulos e componentes

- Em **menu.component.ts**, definimos o atributo selector com o nome **menu**.
- O elemento **templateUrl**, por sua vez, direcionou o componente para **views/menu.component.html**. Mas quem executou esse conteúdo?
- Ao atualizarmos o **app.component.ts**, seu atributo template (**<menu></menu>**) chamou o selector definido no componente menu.
- Dessa forma, executamos um componente a partir de outro.
- A visualização da aplicação em execução nos apresenta este resultado.



Rotas no Angular 4

No **menu.component.ts**, definimos o acesso a um código HTML que define um link para uma lista de contatos (que ainda não temos).

O objetivo é acessar o menu quando executarmos a URL a partir da raiz e a lista de contatos quando a indicarmos. Em outras palavras:

- <http://localhost:3000/> -> acessa o menu de opções.
- <http://localhost:3000/lista> -> acessa a lista de contatos.

Qualquer outra informação que for fornecida, acessamos uma página de erro.

Para essa tarefa, devemos definir um conjunto de **rotas**.



Rotas no Angular 4

Podemos definir uma estrutura de rotas implementando alguns componentes importantes, que descreveremos a seguir:

- É necessário definir, no elemento **<head>** do arquivo **index.html**, o elemento:

```
<base href="/">
```

Este elemento define a origem da estrutura de rotas.



Rotas no Angular 4

- No arquivo HTML que possuir as rotas, ou seja, a parte da aplicação que contém links permitindo acesso a outras rotas, devemos incluir este elemento:

```
<router-outlet></router-outlet>
```

- Devemos, também, configurar o link para executar a rota adequadamente. O arquivo **menu.component.html** deverá ficar assim:

Rotas no Angular 4

```
<body>
  <h1>Página inicial</h1>
  <ul>
    <li>
      <a [routerLink]="/lista">Listar contatos</a>
    </li>
  </ul>
</body>
<router-outlet></router-outlet>
```

Rotas no Angular 4

- Nossa próxima passo é definir o componente representando a página inexistente. Vamos chamá-la de **notfound.component.ts**, a ser criado na pasta **erro**:

```
import { Component } from '@angular/core';

@Component({
  template: `
    <div class="container margem">
      <h1>ERRO 404 - PÁGINA NÃO LOCALIZADA</h1>
    </div>
  `
})
export class NotFoundComponent {}
```

Rotas no Angular 4

- Observe que devemos colocar o conteúdo HTML no atributo **template**, quando ele tiver mais de uma linha, entre **crases**.
- Não será necessário, nesse exemplo, criarmos a pasta **views** nem nenhum arquivo HTML, pois seu conteúdo já foi incluído hard-coded no componente.
- Vamos criar, agora, a lista de contatos a ser exibida para o usuário. Criaremos uma interface contendo a estrutura de informações para um contato, definiremos uma pasta chamada **interfaces** e, nessa pasta, o arquivo **interface.contatos.ts**:

Rotas no Angular 4

```
export interface IContato {  
  codigo: number;  
  nome: string;  
  telefone: string,  
  idade: number;  
}
```

Usaremos essa interface em instantes.

- Agora, crie uma pasta **contatos**.
- Abaixo da pasta **contatos**, crie a pasta **views**.



Rotas no Angular 4

- Na pasta **contatos**, inclua o arquivo **lista.component.ts**:

```
import { Component } from '@angular/core';  
import { IContato } from '../interfaces/interface.contato';  
  
@Component({  
  moduleId: module.id,  
  templateUrl: 'views/lista.component.html'  
})
```



Rotas no Angular 4

```
export class ListaComponent{
  //definindo um array de contatos
  public listaContatos: IContato[] = [
    { codigo: 10, nome: 'Rafa', telefone: '99999-7724', idade: 20 },
    { codigo: 20, nome: 'Gerson', telefone: '2601-9930', idade: 32 },
    { codigo: 30, nome: 'Tatiana', telefone: '3395-1234', idade: 25 }
  ];
}
```



Rotas no Angular 4

- Na pasta **contatos/views**, inclua o arquivo **lista.component.html**:

```
<div class="container margem">
  <h1>LISTAGEM DE CONTATOS</h1>
  <div class="col-md-12">
    <ul class="list_group">
      <li *ngFor="let item of listaContatos"
          class="list-group-item">
        <span class="badge">{{item.codigo}}</span>
        <a href="#">{{item.nome}}</a>
        <span>{{item.telefone}}</span>
      </li>
    </ul>
  </div>
</div>
```



Rotas no Angular 4

- Vamos definir, agora, o sistema de rotas, contemplando as três possíveis rotas. Crie uma pasta chamada **rotas**. Nessa pasta, crie o arquivo **app.routes.ts**:

```
import { Routes } from '@angular/router';
import { ListaComponent } from '../contatos/lista.component';
import { NotFoundComponent } from '../erro/notfound.component';
import { AppComponent } from '../app.component';

export const appRoutes: Routes = [
  { path: "", component: AppComponent },
  { path: "lista", component: ListaComponent },
  { path: "**", component: NotFoundComponent }
];
```



Rotas no Angular 4

- O próximo passo é alterar o módulo da aplicação, arquivo **app.module.ts**. Analise as alterações realizadas:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';

import { appRoutes } from './rotas/app.routes'; //vem primeiro
import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';
```



Rotas no Angular 4

```
//usado na rota
import { ListaComponent } from './contatos/lista.component';
import { NotFoundComponent } from './erro/notfound.component';

@NgModule({
    imports: [ BrowserModule, RouterModule.forRoot(appRoutes) ],
    declarations: [AppComponent, MenuComponent,
        ListaComponent, NotFoundComponent],
    bootstrap: [ AppComponent ]
})
export class AppModule { }
```



Rotas no Angular 4

- Verifique a execução da página. Consegue identificar o que aconteceu?



Rotas no Angular 4

- Vamos resolver esse problema. A rota referente à página inicial aponta para o **AppComponent**, que, por sua vez, carrega o menu de opções. Dessa forma, o menu aparece duas vezes. Criaremos um componente para representar a página inicial.
- Crie uma pasta chamada **home**, e, nessa pasta, o arquivo **home.component.ts**:



Rotas no Angular 4

```
import { Component } from '@angular/core';

@Component({
  moduleId: module.id,
  template: 'PÁGINA INICIAL'
})
export class HomeComponent {}
```

- Configure esse novo componente no arquivo **app.module.ts**:



Rotas no Angular 4

```
//usado na rota
import { ListaComponent } from './contatos/lista.component';
import { NotFoundComponent } from './erro/notfound.component';
import { HomeComponent } from './home/home.component';

@NgModule({
  imports: [ BrowserModule, RouterModule.forRoot(appRoutes) ],
  declarations: [AppComponent, MenuComponent,
    ListaComponent, NotFoundComponent, HomeComponent],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```



Rotas no Angular 4

- Por fim, atualizaremos o arquivo **app.routes.ts**:

```
import { Routes } from '@angular/router';

import { ListaComponent } from '../contatos/lista.component';
import { NotFoundComponent } from '../erro/notfound.component';
import { AppComponent } from '../app.component';
import { MenuComponent } from '../menu/menu.component';
import { HomeComponent } from '../home/home.component';

export const appRoutes: Routes = [
  { path: "", component: HomeComponent },
  { path: "lista", component: ListaComponent },
  { path: "**", component: NotFoundComponent }
]
```

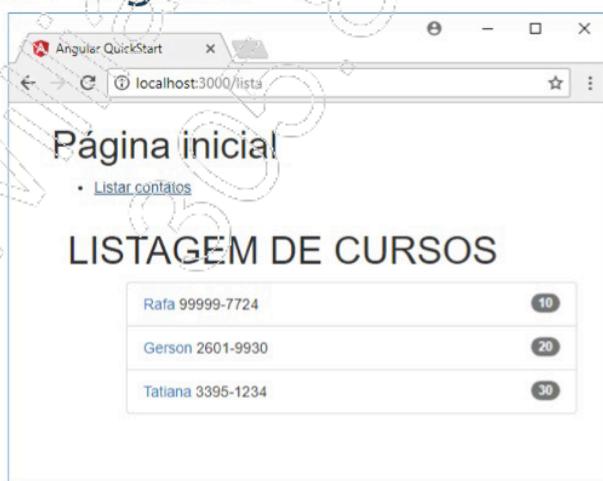


Rotas no Angular 4

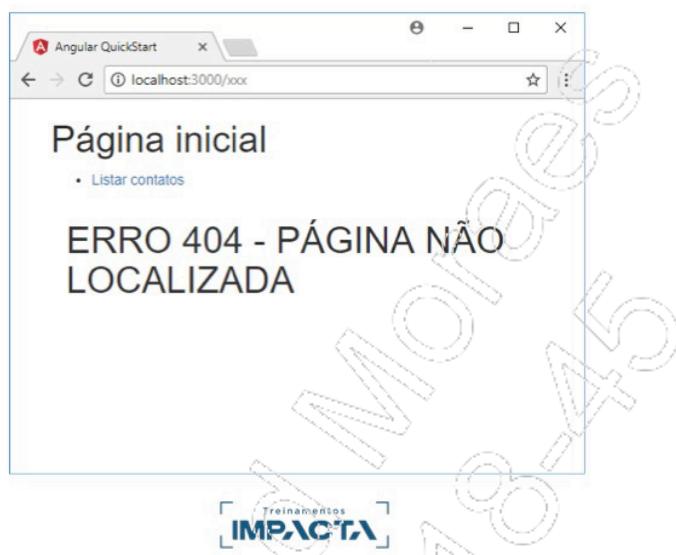
- Com isso, teremos os resultados para cada rota:



Rotas no Angular 4



Rotas no Angular 4



Serviços e Injeção de Dependência

Quando buscamos informações em um conjunto de informações, como é o caso do acesso a um banco de dados ou a um Web service, é importante que as responsabilidades sejam devidamente separadas.

Havendo mais de um componente necessitando desses dados, é conveniente que eles estejam em um único local e, quando necessário, são injetados no componente.

O procedimento de injeção de dependência consiste justamente na tarefa de incluir informações em um componente assim que requisitadas. O Angular 4 possui um recurso de injeção de dependência sofisticado, que nos poupa a tarefa de ter que lidar com ela manualmente.



Serviços e Injeção de Dependência

Para ilustrar, vamos realizar algumas alterações no nosso exemplo. O propósito é transferir a lista de contatos para um serviço e injetá-lo no componente `lista.component.ts`.

Veremos que essa lista poderá posteriormente acessar outra fonte de dados, sem a necessidade de alterar o componente.

- Crie uma pasta chamada `services`.
- Nessa pasta, crie o arquivo `contatos.services.ts`, com o seguinte conteúdo:

Serviços e Injeção de Dependência

```
import { IContato } from "../interfaces/interface.contato";
import { Injectable } from "@angular/core";

@Injectable()
export class ContatosService {
    public getContatos(): IContato[] {
        return [
            { codigo:10, nome:'Rafa', telefone:'99999-7724', idade:20 },
            { codigo:20, nome:'Gerson', telefone:'2601-9930', idade: 32 },
            { codigo:30, nome:'Tatiana', telefone:'3395-1234',idade: 25 }
        ];
    }
}
```



Serviços e Injeção de Dependência

Definimos uma classe representando um serviço “injetável” (`@Injectable`) e, nessa classe, uma função retornando a lista desejada.

O próximo passo é atualizar o arquivo `lista.component.ts` para receber essa coleção do serviço por meio de injeção de dependência.

- Realize as alterações a seguir no arquivo `lista.component.ts`.



Serviços e Injeção de Dependência

```
import { Component } from '@angular/core';
import { IContato } from '../interfaces/interface.contato';
import { ContatosService } from '../services/contatos.service';

@Component({
  moduleId: module.id,
  templateUrl: 'views/lista.component.html'
})
export class ListaComponent{
  public listaContatos: IContato[];

  constructor(contatosService: ContatosService) {
    this.listaContatos = contatosService.getContatos();
  }
}
```



Serviços e Injeção de Dependência

A dependência é introduzida no componente por meio do **construtor** da classe. O construtor é um tipo de método executado no momento em que o componente é criado.

Devemos, também, registrar o novo serviço no arquivo **app.module.ts**, desta vez em outra categoria: **providers**.

- Realize a seguinte alteração em **app.module.ts**:



Serviços e Injeção de Dependência

```
...
//services
import { ContatosService } from './services/contatos.service';

@NgModule({
    imports: [ BrowserModule, RouterModule.forRoot(appRoutes) ],
    declarations: [ AppComponent, MenuComponent,
        ListaComponent, NotFoundComponent, HomeComponent ],
    providers: [ ContatosService ],
    bootstrap: [ AppComponent ]
})
... continua o modulo
```



Serviços e Injeção de Dependência

Ao executar a aplicação, podemos ver que o resultado é o mesmo. A diferença está na forma de acesso, somente.



Acesso a Web services

Já sabemos que um Web service é um mecanismo de acesso a informações externas à aplicação. No Angular 4, o acesso a Web services é uma tarefa bastante simplificada.

Para ilustrar, vamos atualizar o serviço **ContatosService** para receber a lista de contatos de um Web service (este exemplo é apenas uma ilustração. No projeto que realizaremos, nós apresentaremos uma funcionalidade real).

- Considere a seguinte alteração no arquivo **contatos.service.ts**:



Acesso a Web services

```
import { IContato } from "../interfaces/interface.contato";
import { Injectable } from "@angular/core";

import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Rx';
import 'rxjs/Rx';
```



Acesso a Web services

```
@Injectable()
export class ContatosService {

    //acesso ao HTTP
    public constructor(private _http: Http) { }

    private url: string = "http://localhost:7628/api/contatos";

    public getContatos(): Observable<IContato[]> {

        return this._http.get(this.url)
            .map(res => res.json());
    }
}
```



Acesso a Web services

O módulo **Http** é injetado no componente, da mesma forma que o serviço que criamos no tópico anterior.

No componente **lista.component.ts**, devemos, também, realizar algumas alterações. Essas alterações serão realizadas apenas no construtor:

```
constructor(contatosService: ContatosService) {  
    this.listaContatos = contatosService.getContatos()  
        .subscribe(res => this.listaContatos = res,  
                  error => alert(error),  
                  () => console.log('finalizado'));  
}
```

Acesso a Web services

Nesse exemplo, consideramos a função **subscribe()**, necessária quando acessamos um serviço por meio de um Web service. Essa função recebe três funções callback, que executamos por meio do recurso **aerofuncion**, também chamado de **expressão lambda**, onde definimos a alteração, sem a necessidade de definir a função em si.

Binding unidirecional e bidirecional

É possível apresentarmos dados em uma lista e, ao selecionar o elemento em uma lista, apresentar seus detalhes em uma caixa de textos.

No momento em que alteramos alguma propriedade na caixa de textos, o item na lista é alterado também.

Estamos falando em um vínculo que ocorre em duas direções, ou seja, bidirecional.

Este funcionamento é bastante comum em projetos Angular 4.

Sua demonstração será realizada no projeto do curso.



Outras considerações

Temos uma série de funcionalidades disponíveis no Angular 4, como pudemos ver ao longo deste capítulo.

O desenvolvimento de um projeto ao longo do curso nos permitirá apresentar outras funcionalidades, além daquelas disponibilizadas neste capítulo.

Sabemos que o curso possui uma natureza bastante prática e, por isso, nada melhor que aplicar esses conceitos para ilustrá-las da melhor forma.

Elaborar o Projeto 06



Laboratórios

Elaborar os laboratórios:

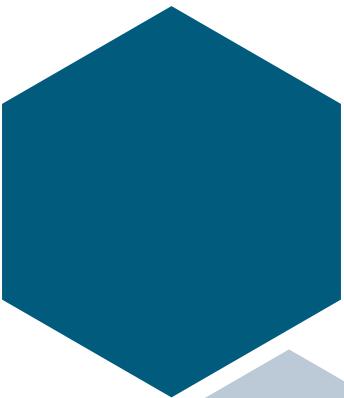
Laboratório 01

Laboratório 02

Laboratório 03

Laboratório 04

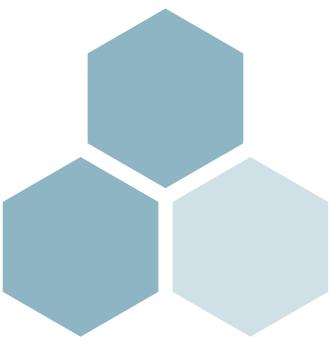
Trainamentos
IMPACTA



Visão geral do JavaScript

Apêndice

Willians Moreira
305.809-45



Editora
IMPACTA



Introdução ao JavaScript

- **JavaScript** é uma linguagem interpretada cuja execução ocorre no browser, ou seja, no cliente.
- O JavaScript deve ser usado com a finalidade de tornar as páginas dinâmicas.

No desenvolvimento Full Stack, o JavaScript se torna a linguagem adequada por ser utilizada pelas quatro ferramentas.

Pode haver desenvolvedores Full Stack, sem necessariamente ter o JavaScript como linguagem central. O preço a se pagar por isso é o conhecimento e o domínio de diferentes linguagens e plataformas, além da dependência por plataformas específicas (plataforma Java, por exemplo).



Introdução ao JavaScript

DOM (Document Object Model)

- O DOM é usado pelo navegador Web para representar seus elementos. É uma estrutura construída pelo navegador, em conformidade com o conteúdo da página e com a natureza dos componentes.
- Quando alteramos a estrutura dos elementos via JavaScript, alteramos, também, o DOM, ou a forma como os elementos são mostrados.



Introdução ao JavaScript

Estrutura do JavaScript

- Instruções são finalizadas com ponto e vírgula:

```
var variavel = 3;  
variavel = variavel + 1;  
document.write("Instruções longas \  
podem ser escritas em outra linha");
```



Introdução ao JavaScript

Estrutura do JavaScript

- Comentários:

```
document.write("Introdução ao JavaScript"); //apresenta  
mensagem  
/*  
Este bloco pode ser usado para múltiplas  
anotações  
*/
```



Introdução ao JavaScript

- Usamos **var** para declarar variáveis:

```
var nome = "Sebastian";
var idade = 35;
```



Introdução ao JavaScript

- Existem três tipos base no JavaScript:
 - String, Number e Boolean.
- Variáveis podem ser definidas como **null**.

```
var semValor; //undefined
var nuloValor = null; //null, diferente de undefined
```



Introdução ao JavaScript

- Funções são usadas para que um conjunto de instruções possa ser reaproveitado.

```
function nomeFuncao(parametro1, parametro2)
{
    //instruções
}
```



Introdução ao JavaScript

- Parâmetros são acessíveis somente dentro das funções.
- Funções podem retornar valores.
- Variáveis locais podem ser declaradas dentro das funções.
- Variáveis declaradas fora das funções possuem acesso global.



Introdução ao JavaScript

- Comando **if**:

```
if (nuloValor == null) {  
    document.write("Valor nulo");  
}  
  
if (idade < 18) {  
    document.write("Menor de idade");  
}  
else {  
    document.write("Maior de idade");  
}
```



Introdução ao JavaScript

- Comando **switch**:

```
var taxa;  
switch (tipoQuarto) {  
    case "Suite":  
        taxa = 500;  
        break;  
    case "King":  
        taxa = 400;  
        break;  
    default:  
        taxa = 300;  
}
```



Introdução ao JavaScript

- Estruturas de repetição: **while**, **do...while**, **for**:

```
while (valor > 0) {  
    saldo = saldo +  
    valor;  
    valor--;  
}
```

```
do {  
    if (valor <= 0) {  
        break;  
    }  
} while (true);
```

```
for (var i = 0; i < 10; i++) {  
    executar(i);  
}
```



Introdução ao JavaScript

Objetos implícitos do JavaScript:

- String
- Date
- Array
- RegExp



Introdução ao JavaScript

```
//array  
var diasUteis = ["Segunda", "Terça", "Quarta", "Quinta", "Sexta"];  
  
//Date  
var hoje = new Date();  
//RegExp  
var reg = new RegExp("a.c");  
if (reg.test("abc")) {  
    //instrução  
}
```



Introdução ao JavaScript

Objetos

- O formato JSON é o mais adequado para representação de objetos:

```
//JSON  
var aluno = { "nome": "Carlos", "curso": "HTML5" };  
  
var alunos = [  
    { "nome": "Gerson", "curso": ".NET" },  
    { "nome": "Talita", "curso": "Java" },  
    { "nome": "Zé", "curso": "Manutenção" }  
];
```



Introdução ao JavaScript

- JavaScript possui APIs para serializar e realizar o parsing de objetos JSON:
 - `JSON.parse()`
 - `JSON.stringify()`



Integrando com JavaScript

O JavaScript possui uma API de programação que permite manipular elementos da página Web de diversas formas:

- Alterando elementos;
- Executando eventos;
- Modificando estilos, quando aplicáveis;
- Validando e atualizando páginas.

A seguir, apresentaremos exemplos de uso dos recursos mencionados.



Integrando com JavaScript

Considere este formulário:

```
<form name="LoginUsuario">
  <label for="uname">Nome do usuário:</label>
  <input id="uname" name="username" type="text" />
  <input type="submit" value="Enviar" />
</form>
```



Integrando com JavaScript

Podemos referenciar o form assim:

```
document.forms[0];
document.forms["LoginUsuario"];
document.forms.LoginUsuario;
document.LoginUsuario;
```



Integrando com JavaScript

Podemos, também, referenciar um elemento do form (elemento **username**):

```
document.forms.LoginUsuario.elements[0];
document.forms.LoginUsuario.elements["username"];
document.forms.LoginUsuario.username;
document.LoginUsuario.username;
document.getElementById("uname");
```



Integrando com JavaScript

A manipulação de objetos no DOM consiste nos seguintes passos:

- Criar um novo objeto com o dado;
- Buscar o elemento pai que deve conter o novo dado; e
- Manipular o elemento com o novo dado.

Para remover um elemento ou atributo:

- Buscar o elemento; e
- Usar **removeChild** ou **removeAttribute**, conforme o caso.



Integrando com JavaScript

Exemplo de aplicação:

```
<body>
<div id="conteudo">

</div>
<input type="button" value="Alterar conteúdo da div"
      onclick="inserir(); " />
<script>
function inserir() {
    var elemento = document.getElementById("conteudo");
    elemento.innerHTML = "TEXTO INSERIDO NO ELEMENTO";
}
</script>
</body>
```



Eventos em campos de formulário

- Nós tivemos a oportunidade de ver como uma função JavaScript pode ser chamada para ser executada pelo evento click de um botão.
- Existem outras alternativas para executarmos eventos, sempre visando uma melhor estruturação do código.
- Muitos elementos HTML possuem eventos embutidos que podem ser chamados conforme surge a necessidade. É o caso do evento **onclick** do elemento **button**, que vimos no exemplo anterior.



Eventos em campos de formulário

- Para separar a chamada ao evento do formulário, podemos adicionar o evento via JavaScript e incluí-lo no DOM. Veja o exemplo:

```
<body>
<input type="button" id="btn" value="Apresentar mensagem" />

<script>
    var botao = document.getElementById("btn");

    botao.addEventListener("click", function () {
        alert("O evento click foi acionado");
    });
</script>
</body>
```



Eventos em campos de formulário

- Nesse exemplo, usamos uma função anônima, pois sua funcionalidade é exclusiva para este evento.
- É possível definir funções e associá-las a eventos, bastando indicar seu nome no lugar da definição da função.
- Este procedimento é útil, especialmente se o evento associado a um controle depende de uma condição especial, além de poder ser usado para múltiplos eventos.



Introdução ao jQuery

- O jQuery é uma biblioteca JavaScript que fornece portabilidade de código JavaScript, além de permitir a construção de aplicações cross-browser.
- Para trabalharmos com jQuery, primeiro temos que ter disponível a API (obtida de diversas formas; dentre elas, por meio do site jquery.com ou do CDN).
- As versões mudam bastante de implementação para implementação.
- Um exemplo é dado a seguir:



Introdução ao jQuery

```
<head>
<title></title>
<meta charset="utf-8" />
<script src="../Scripts/jquery-3.1.1.js"></script>
</head>
```



Introdução ao jQuery

Os principais recursos do jQuery incluem:

- Seleções de elementos HTML;
- Manipulação de elementos HTML;
- Manipulação CSS;
- Eventos HTML;
- Efeitos e animações JavaScript;
- HTML DOM;
- AJAX.



Introdução ao jQuery

A sintaxe jQuery é orientada para selecionar elementos HTML e efetuar ações neles.

A sintaxe básica é `$(seletorHTML).ação()`.

O símbolo `$` define o elemento `(seletorHTML)`, onde fica o elemento, e `ação()` representa a tarefa a ser executada.

Exemplos:

```
$(this).hide()      // Esconde o elemento atual
$("p").hide()       // Esconde todos os parágrafos
$("p.teste").hide() // Esconde todos os parágrafos com a
                  classe="teste"
$("#test").hide()   // Esconde o elemento com o id="teste"
```



Introdução ao jQuery

Os seletores permitem selecionar e manipular elementos HTML. É possível selecionar por nome, atributo ou conteúdo.

Seletores de elementos:

```
$("p")          // seleciona os elementos <p>  
$("p.intro")    // seleciona todos os elementos <p> com class="intro".  
$("p#demo")     // seleciona o primeiro elemento <p> com id="demo".
```



Introdução ao jQuery

Seletores de atributos ou conteúdos:

```
$("[href]")  
// Seleciona todos os elementos com um atributo href  
  
$("[href='#']")  
// Seleciona todos os elementos com um valor href igual a "#"  
  
$("[href!='#']")  
// Seleciona todos os elementos com um valor href não igual a "#"  
  
$("[href$='.jpg']")  
// Seleciona todos os elementos com um atributo href que termine com ".jpg"
```



Introdução ao jQuery

Os eventos jQuery são as funções que lidam com as ações no HTML. Veja o exemplo a seguir:

```
<head>
    <title></title>
    <meta charset="utf-8" />
    <script src="../Scripts/jquery-3.1.1.js"></script>
    <script type="text/javascript">
        $(document).ready(function () {
            $("button").click(function () {
                $("p").hide();
            });
        });
    </script>
</head>
```



Introdução ao jQuery

```
<body>
<h2>Isto é um título</h2>
<p>Isto é um parágrafo.</p>
<p>Isto é mais um parágrafo.</p>
<button>Clica-me</button>
</body>
```

Quando o usuário clica no botão, todos os elementos `<p>` são escondidos.



Introdução ao jQuery

Mostrar e esconder elementos (**hide()**, **show()**):

```
$("#hide").click(function(){
    $("p").hide();
});

$("#show").click(function(){
    $("p").show();
});
```



Introdução ao jQuery

Ambos podem funcionar em conjunto com parâmetros opcionais: **speed** e **callback**.

```
$(selector).hide(speed,callback)
$(selector).show(speed,callback)
```

O parâmetro **speed** especifica a velocidade de mostrar/esconder e pode ter os valores **slow**, **normal**, **fast** ou em milissegundos.

```
$("#botao").click(function(){
    $("p").hide(800);
});
```



Introdução ao jQuery

O parâmetro **callback** é o nome de uma função a ser executada depois que a função hide/show estiver completa.

Alternar (**toggle()**):

O método **toggle()** permite alterar a visibilidade de elementos HTML que usam a função show/hide. Os elementos escondidos são mostrados, e os elementos visíveis são escondidos.



Introdução ao jQuery

```
$(selector).toggle(speed,callback)
```

```
$("#botao").click(function(){
    $("p").toggle(850);
});
```



Introdução ao jQuery

Deslizar (`slideDown()`, `slideUp()`, `slideToggle()`):

Os métodos de deslizamento do jQuery alteram gradualmente a altura dos elementos selecionados, por meio dos seguintes métodos:

```
$(selector).slideDown(speed,callback)
$(selector).slideUp(speed,callback)
$(selector).slideToggle(speed,callback)
```

Introdução ao jQuery

Exemplos:

```
//slideDown()
$("flip").click(function () {
    $(".panel").slideDown();
});

//slideUp()
$("flip").click(function () {
    $(".panel").slideUp();
};

//slideToggle()
$("flip").click(function () {
    $(".panel").slideToggle();
});
```

Introdução ao jQuery

Desvanecer (**fadeIn()**, **fadeOut()**, **fadeTo()**):

Os métodos de desvanecer alteram gradualmente a opacidade dos elementos selecionados. O jQuery tem os seguintes métodos de desvanecimento:

```
$(selector).fadeIn(speed,callback)
$(selector).fadeOut(speed,callback)
$(selector).fadeTo(speed,opacity,callback)
```



Introdução ao jQuery

Exemplos:

```
//fadeIn()
$("botão").click(function(){
    $("div").fadeIn(2000);
});

//fadeOut()
$("botão").click(function(){
    $("div").fadeIn(2000);
});

//fadeTo()
$("botão").click(function(){
    $("div").fadeTo("slow",0.30);
});
```



Introdução ao jQuery

Elaborar o projeto deste Apêndice

Trinamentos
IMPACTA

Projeto

305.800

Williams M d Moraes



Apresentando o projeto

Ao longo do curso, desenvolveremos um projeto contendo diversas vertentes. Nesse projeto, agregaremos os recursos do JavaScript, do Node.js com o Express.js e do banco de dados MongoDB, os quais aproveitaremos para ampliar o projeto com AngularJS e com Angular 4, contemplando, assim, os recursos aprendidos no treinamento.

Trata-se de um sistema de cadastro de eventos. Nessa aplicação, teremos os seguintes recursos a considerar:

- Pessoas devidamente logadas realizam o cadastro de eventos. Os eventos podem ser: palestra, festas, participação em concursos etc. Alguns eventos podem ser pagos;
- Se o evento for pago, o interessado poderá efetuar o pagamento por meio de cartão de crédito, cujos dados são encaminhados para um Web service, simulando uma administradora de cartões.

A aplicação deverá ter uma aparência agradável, com boa usabilidade e facilidade de navegação. O projeto será desenvolvido com o VSCode (Visual Studio Code), previamente instalado.

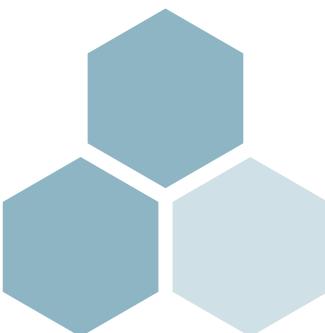


1

Preparando o ambiente

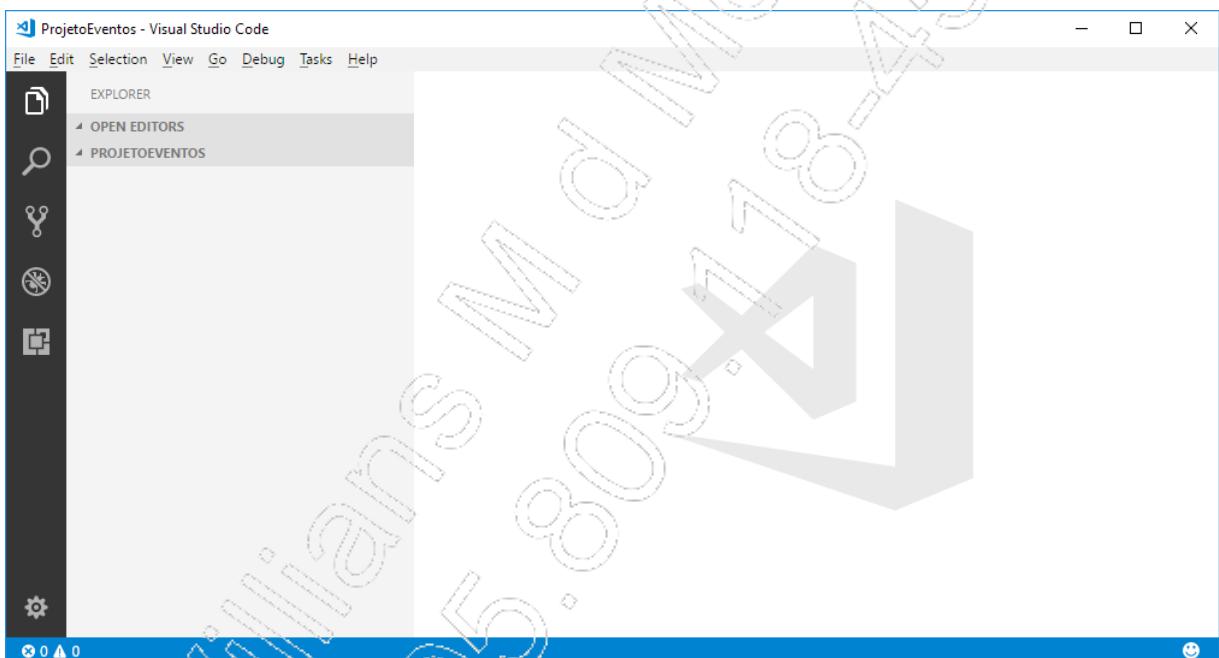


Atividade



A – Definindo a estrutura inicial para o projeto

1. Em um local de sua preferência (unidade C, pasta Documentos etc.), defina uma pasta chamada **ProjetoEventos** (outras pastas serão adicionadas ao longo do curso);
2. Abra o VSCode, apontando para essa pasta:



3. A partir deste ponto, podemos adicionar, alterar ou remover pastas e arquivos para nossa aplicação.

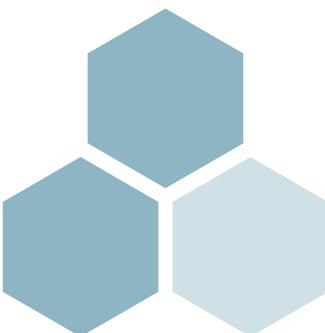


2

Node.js e Express.js



Atividade



A – Definindo um projeto baseado no Node.js

1. Na pasta **ProjetoEventos**, inclua uma nova pasta chamada **Nodejs**. Esta será a pasta que conterá a aplicação baseada no Node.js, no Express.js e no MongoDB;
2. Acesse a pasta **Nodejs** a partir do VSCode;
3. Use o prompt de comandos, ou o próprio VSCode, para executar os comandos para a criação do projeto. Um atalho no VSCode para abrir o comando é **CTRL + .**. Estando na pasta **Nodejs**, execute a sequência de comandos a seguir:

```
npm install -g express-generator  
express nodeEventos --ejs  
cd nodeEventos  
npm install
```

4. O processo anterior permitiu a instalação do Express.js a partir do comando **npm** do Node.js. Em seguida, criou o projeto **nodeEventos** usando o engine EJS para a camada de visualização. Mudamos para a pasta **nodeEventos** e executamos o comando **install**. Quando este processo estiver finalizado, mude para a pasta do projeto no VSCode;

5. Abra o arquivo **app.js** criado. Remova todo o seu conteúdo e acrescente o código adiante:

```
var express = require('express');  
app = express();  
  
app.set('views', __dirname + '/views');  
app.set('view engine', 'ejs');  
app.use(express.static(__dirname + '/public'));  
  
app.listen(3000, function () {  
    console.log("Aplicação no ar.");  
});
```

6. Execute a aplicação com o comando **node app.js**. Este procedimento é só para constatar que a aplicação está funcionando corretamente;

7. Crie as pastas **models** e **controllers**;

8. Instale, via **npm**, o módulo **express-load**. Para isso, execute o comando adiante:

```
npm install express-load --save
```

9. Atualize o arquivo **app.js**:

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

app = express();

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

load('models')
  .then('controllers')
  .then('routes')
  .into(app);

app.listen(3000, function () {
  console.log("Aplicação no ar.");
});
```

10. Exclua os arquivos **routes/users.js** e **routes/index.js**:

11. Crie o arquivo **routes/home.js** com o conteúdo a seguir:

```
module.exports = function (app) {
  var home = app.controllers.home;
  app.get('/', home.index);
};
```

12. A instrução **app.controllers.home** se refere a **controllers/home.js**. Crie este arquivo:

```
module.exports = function (app) {
  var HomeController = {
    index: function (req, res) {
      res.render('home/index');
    }
  };
  return HomeController;
};
```

13. Exclua o arquivo `views/index.ejs`. Inclua o arquivo `views/home/index.ejs` (tela de login):

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Cadastro de Eventos e Convidados - Login</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
</head>

<body>
    <header>
        <h1>Login</h1>
        <h4>Entre com suas credenciais</h4>
    </header>
    <section>
        <form action="/login" method="post">
            Nome: <br/>
            <input type="text" name="usuario[nome]" />
            <br> Senha: <br/>
            <input type="password" name="usuario[senha]" />
            <br>
            <button type="submit">Login</button>
        </form>
    </section>
    <footer>
        <small>Sistema de Cadastro de Eventos e Convidados</small>
    </footer>
</body>

</html>
```

14. Execute a aplicação e verifique se está tudo correto;

15. Vamos, agora, incluir um recurso para armazenar os dados do usuário em sessão. A ideia é que somente usuários logados (na sessão) possam acessar as partes da aplicação. Deveremos, portanto, instalar o módulo **express-session** via npm:

```
npm install express-session --save
```

16. Após essa instalação, crie duas novas rotas. No arquivo `routes/home.js`, realize as alterações:

```
module.exports = function (app) {
    var home = app.controllers.home;
    app.get('/', home.index);
    app.post('/login', home.login);
    app.get('/logout', home.logout);
};
```

17. Observe que criamos duas novas rotas. Essas rotas devem ser de conhecimento do controller. Vamos alterar o arquivo **controllers/home.js** para incluir estas duas actions:

```
module.exports = function (app) {
  var HomeController = {
    index: function (request, response) {
      response.render('home/index');
    },
    login: function (request, response) {
      var nome = request.body.usuario.nome;
      var senha = request.body.usuario.senha;

      if (nome == 'admin' && senha == 'admin') {
        var usuario = request.body.usuario;

        request.session.usuario = usuario;
        response.redirect('/menu');
      } else {
        response.redirect('/');
      }
    },
    logout: function (request, response) {
      request.session.destroy();
      response.redirect('/');
    }
  };
  return HomeController;
};
```

Observe que acrescentamos uma vírgula ao final do action index existente. Neste caso, estamos simulando um login considerando usuário = 'admin' e senha = 'admin'.

18. Se o usuário for validado, a aplicação o redireciona para a rota '/menu'. Ela se refere a um controller que ainda não criamos. Crie o arquivo **controllers/eventos.js**:

```
module.exports = function (app) {
  var EventosController = {
    menu: function (request, response) {

      var usuario = request.session.usuario,
          params = { usuario: usuario };
      response.render('eventos/menu', params);
    }
  };
  return EventosController;
};
```

19. Defina uma rota para o menu de opções. Crie o arquivo **routes/eventos.js**:

```
module.exports = function (app) {
  var eventos = app.controllers.eventos;
  app.get('/menu', eventos.menu);

};
```

20. De acordo com a função render, a view referente ao menu deverá ser criada na pasta **views/eventos**, e seu nome será **menu.ejs**. Simplificadamente, teremos: (**views/eventos/menu.ejs**):

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Cadastro de Eventos e Convidados - Menu de Opções</title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <header>
    <p>Bem-vindo,
      <%= usuario.nome %>
    </p>
    <h1>Menu de Opções</h1>
    <h4>Escolha sua opção</h4>
  </header>
  <section>
    <ul>
      <li>
        <a href="/cadUsuario">Cadastro de Usuários</a>
      </li>
      <li>
        <a href="/cadEvento">Cadastro de Eventos</a>
      </li>
      <li>
        <a href="/listaEventos">Lista de Eventos</a>
      </li>
    </ul>
  </section>
  <section>
    <a href='/logout'>Logout</a>
  </section>
  <footer>
    <small>Sistema de Cadastro de Eventos e Convidados</small>
  </footer>
</body>
</html>
```

21. Antes de executar a aplicação, deveremos habilitar os componentes (middlewares) instalados **body-parser**, **express-session** e **cookie-parser** no arquivo **app.js**:

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

var bodyParser = require('body-parser');
var cookieParser = require('cookie-parser');
var expressSession = require('express-session');

app = express();

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

app.use(cookieParser('nodeEventos'));
app.use(expressSession());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.use(express.static(__dirname + '/public'));

load('models')
  .then('controllers')
  .then('routes')
  .into(app);

app.listen(3000, function () {
  console.log("Aplicação no ar.");
});
```

cookieParser deve vir primeiro, para que o **expressSession()** utilize o mesmo **sessionId** mantido no cookie.

22. Execute a aplicação;

23. A partir do menu de opções (`views/eventos/menu.ejs`), temos links apontando para outras funcionalidades do sistema. Vamos definir suas rotas em `routes/eventos.js`:

```
module.exports = function (app) {
  var eventos = app.controllers.eventos;
  app.get('/menu', eventos.menu);

  app.get('/cadUsuario', eventos.cadastroUsuario);
  app.get('/cadEvento', eventos.cadastroEvento);
  app.get('/listaEventos', eventos.listaEventos);
};
```

24. No controller (`controllers/eventos.js`), devemos adicionar os actions solicitados na rota:

```
module.exports = function (app) {
  var EventosController = {
    menu: function (request, response) {
      var usuario = request.session.usuario,
          params = { usuario: usuario };
      response.render('eventos/menu', params);
    },

    cadastroUsuario: function (request, response) {
      var usuario = request.session.usuario,
          params = { usuario: usuario };
      response.render('eventos/cadUsuario', params);
    },

    cadastroEvento: function (request, response) {
      var usuario = request.session.usuario,
          params = { usuario: usuario };
      response.render('eventos/cadEvento', params);
    },

    listaEventos: function (request, response) {
      var usuario = request.session.usuario,
          params = { usuario: usuario };
      response.render('eventos/listaEventos', params);
    }
  };

  return EventosController;
};
```

25. Com essa alteração, na pasta **views/eventos** devemos adicionar as views **cadUsuario.ejs**, **cadEvento.ejs** e **listaEventos.ejs**;

- **cadUsuario.ejs**

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Cadastro de Eventos e Convidados</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
    <header>
        <p>Bem-vindo, <%= usuario.nome %>
    </p>
        <h1>Cadastro de Usuários</h1>
        <h4>Forneça os dados do usuário para seu cadastro</h4>
    </header>
    <section>
        <form action="/novoUsuario" method="post">
            Nome:
            <br />
            <input type="text" name="usuario[nome]" />
            <br> Senha:
            <br/>
            <input type="password" name="usuario[senha]" />
            <br> Confirma:
            <br />
            <input type="password" name="usuario[confirma]" />
            <br>
            <button type="submit">Login</button>
        </form>
    </section>
    <footer>
        <small>Sistema de Cadastro de Eventos e Convidados</small>
    </footer>
</body>

</html>
```

- **cadEvento.ejs**

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>Cadastro de Eventos e Convidados</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
</head>

<body>
    <header>
        <p>Bem-vindo, <%= usuario.nome %>
        </p>
        <h1>Cadastro de Eventos</h1>
        <h4>Forneça os dados para o cadastro</h4>
    </header>
    <section>
        <form action="/novoEvento" method="post">
            Descrição:
            <br />
            <input type="text" name="evento[descricao]" >
            <br> Data:
            <br />
            <input type="date" name="evento[data]" >
            <br> Preço:
            <br />
            <input type="text" name="evento[preco]" >
            <br>
            <button type="submit">Enviar</button>
        </form>
    </section>
    <footer>
        <small>Sistema de Cadastro de Eventos e Convidados</small>
    </footer>
</body>

</html>
```

- **listaEventos.ejs**

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Cadastro de Eventos e Convidados - Login</title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <header>
    <p>Bem-vindo, <%= usuario.nome %>
    </p>
    <h1>Lista de Eventos</h1>
    <h4>Selecione uma opção na lista para executá-la</h4>
  </header>
  <section>
    <!--será implementado em breve-->
  </section>
  <footer>
    <small>Sistema de Cadastro de Eventos e Convidados</small>
  </footer>
</body>

</html>
```

26. A view **cadUsuario.ejs** define um formulário como o action **novoUsuario**. Este action deve estar presente na rota **routes/home.js**. Vamos alterá-la:

```
module.exports = function (app) {

  var home = app.controllers.home;

  app.get('/', home.index);
  app.post('/login', home.login);
  app.get('/logout', home.logout);

  app.post('/novoUsuario', home.novoUsuario);
};
```

27. Execute e teste a aplicação. Sem fazer login, execute a rota **/menu**. O que você consegue observar?

28. Crie a pasta **middlewares** na raiz do projeto. Nessa pasta, inclua o arquivo **valida.js**:

```
module.exports = function (request, response, next) {
    if (!request.session.usuario) {
        return response.redirect('/');
    }
    return next();
};
```

29. Inclua este middleware em todas as rotas que apresentam views com o nome do usuário (arquivo **routes/eventos.js**):

```
module.exports = function (app) {

    var valida = require('../middlewares/valida');
    var eventos = app.controllers.eventos;

    app.get('/menu', valida, eventos.menu);

    app.get('/cadUsuario', valida, eventos.cadastroUsuario);
    app.get('/cadEvento', valida, eventos.cadastroEvento);
    app.get('/listaEventos', valida, eventos.listaEventos);

};
```

30. Teste a aplicação;

31. Na pasta **views**, inclua dois arquivos: **erro404.ejs** e **erroServidor.ejs**:

- **erro404.ejs**

```
<!DOCTYPE html>
<html>

    <head>
        <meta charset="utf-8">
        <title>Cadastro de Eventos e Convidados</title>
        <link rel='stylesheet' href='/stylesheets/style.css' />
    </head>

    <body>
        <header>
            <h1>Página não encontrada.</h1>
        </header>
        <section>
            <p>
                <a href="/">Página inicial</a>
            </p>
        </section>
        <footer>
            <small>Sistema de Cadastro de Eventos e Convidados</small>
        </footer>
    </body>

</html>
```

- **erroServidor.ejs**

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>Cadastro de Eventos e Convidados</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
</head>

<body>
    <header>
        <h1>Ocorreu um erro em tempo de execução.</h1>
    </header>
    <section>
        <p>
            <strong>Detalhes:</strong>
            <%= error.message %>
            <br/>
            <a href="/">Página inicial</a>
        </p>
    </section>
    <footer>
        <small>Sistema de Cadastro de Eventos e Convidados</small>
    </footer>
</body>

</html>
```

32. Defina o middleware de erro, em **middlewares/error.js**:

```
exports.notFound = function (request, response, next) {
    response.status(404);
    response.render('erro404');
};

exports.serverError = function (error, request, response, next) {
    response.status(500);
    response.render('erroServidor', { error: error });
};
```

33. Inclua estes dois middlewares no arquivo **app.js**, por último!

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

var bodyParser = require('body-parser');
var cookieParser = require('cookie-parser');
var expressSession = require('express-session');

var error = require('./middlewares/error');

app = express();

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

app.use(cookieParser('nodeEventos'));
app.use(expressSession());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.use(express.static(__dirname + '/public'));

load('models')
  .then('controllers')
  .then('routes')
  .into(app);

//middlewares
app.use(error.notFound);
app.use(error.serverError);

app.listen(3000, function () {
  console.log("Aplicação no ar.");
});
```

34. No controller **controllers/eventos.js**, prepare o action:

```
//cadastro de eventos
novoEvento: function (request, response) {

    //código a ser implementado
    response.redirect('/menu');
}
```

35. No arquivo **routes/eventos.js**, acrescente a rota:

```
module.exports = function (app) {

    var valida = require('../middlewares/valida');
    var eventos = app.controllers.eventos;

    app.get('/menu', valida, eventos.menu);

    app.get('/cadUsuario', valida, eventos.cadastroUsuario);
    app.get('/cadEvento', valida, eventos.cadastroEvento);
    app.get('/listaEventos', valida, eventos.listaEventos);

    app.post('/novoEvento', eventos.novoEvento);
};
```

36. No controller **controllers/home.js**, prepare o action:

```
//cadastro de usuários
novoUsuario: function (request, response) {
    var nome = request.body.usuario.nome;
    var senha = request.body.usuario.senha;
    var confirma = request.body.usuario.confirma;

    //código a ser implementado
    response.redirect('/menu');
}
```

37. Teste a aplicação.

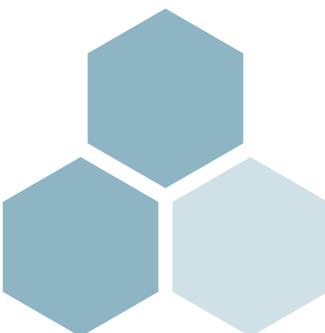


3

Acesso a dados com Mongoose



Atividade



A – Atualizando o projeto nodeEventos para incluir o banco de dados MongoDB

1. Crie uma pasta adequada para seu banco de dados. Por exemplo, **C:\ProjetoEventos\Dados** (ou outra de sua escolha). Se preferir manter o padrão, crie a pasta **C:\data\db**. Lembre-se de que, ao iniciar o serviço do MongoDB, é necessário especificar a pasta escolhida, se for diferente da padrão:

```
mongod --dbpath C:\ProjetoEventos\Dados
```

2. Instale o Mongoose via npm:

```
npm install mongoose --save
```

3. Atualize o arquivo **app.js** para contemplar o banco de dados (observe que a variável **db** é global). Será criado um banco de dados chamado **neventos**. Observe que incluímos, também, os eventos do ciclo de vida da conexão:

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

var bodyParser = require('body-parser');
var cookieParser = require('cookie-parser');
var expressSession = require('express-session');

var error = require('./middlewares/error');

app = express();

var mongoose = require('mongoose');
global.db = mongoose.connect('mongodb://localhost:27017/neventos');

mongoose.connection.on('connected', function () {
    console.log('=====Conexão estabelecida com sucesso=====');
});
mongoose.connection.on('error', function (err) {
    console.log('=====Ocorreu um erro: ' + err);
});
mongoose.connection.on('disconnected', function () {
    console.log('=====Conexão finalizada=====');
});

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

app.use(cookieParser('nodeEventos'));
app.use(expressSession());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

Acesso a dados com Mongoose

```

app.use(express.static(__dirname + '/public'));

load('models')
  .then('controllers')
  .then('routes')
  .into(app);

//middlewares
app.use(error.notFound);
app.use(error.serverError);

app.listen(3000, function () {
  console.log("Aplicação no ar.");
});

```

4. Inicie o banco de dados, executando **mongo** a partir da pasta bin do banco (necessitaremos de dois terminais). Neste ponto, será necessário alterar para o banco de dados referenciado pela URL de conexão. Execute, neste novo terminal, o comando adiante:

```
use neventos
```

5. Crie o modelo para cadastro e busca de usuários em **models/usuarios.js**:

```

module.exports = function (app) {

  var mongoose = require('mongoose');
  var Schema = mongoose.Schema;

  var usuario = Schema({
    nome: { type: String, required: true, index: { unique: true } },
    senha: { type: String, required: true }
  });
  return mongoose.model('usuarios', usuario);
};

```

6. Inclua um usuário no banco de dados. Os dados sugeridos são: **nome='admin'** e **senha='admin'**. Para tanto, use o comando adiante:

```
db.usuarios.insert({ 'nome': 'admin', 'senha': 'admin' })
```

7. Em **controllers/home.js**, altere a validação para buscar o usuário no banco de dados e a inclusão para efetivar a criação de um novo usuário:

```
module.exports = function (app) {  
  
    var mongoose = require('mongoose');  
    var Usuario = mongoose.model('usuarios');  
  
    var HomeController = {  
        index: function (request, response) {  
            response.render('home/index');  
        },  
  
        login: function (request, response) {  
  
            var nome = request.body.usuario.nome;  
            var senha = request.body.usuario.senha;  
  
            var query = { 'nome': nome, 'senha': senha };  
  
            Usuario.findOne(query).select('nome senha')  
                .exec(function (erro, usuario) {  
                    if (erro) {  
                        response.redirect('/');  
                    }  
                    else {  
                        request.session.usuario = usuario;  
                        response.redirect('/menu');  
                    }  
                });  
        },  
  
        logout: function (request, response) {  
            request.session.destroy();  
            response.redirect('/');  
        },  
    };  
};
```

```
//cadastro de usuários
novoUsuario: function (request, response) {
    var nome = request.body.usuario.nome;
    var senha = request.body.usuario.senha;
    var confirma = request.body.usuario.confirma;

    if ((senha != confirma) || nome.trim().length == 0) {
        response.redirect('/');
    }
    else {

        var usuario = request.body.usuario;
        Usuario.create(usuario, function (erro, usuario) {
            if (erro) {
                response.redirect('/');
            }
            else {
                response.redirect('/menu');
            }
        });
    }
};

return HomeController;
};
```

8. Defina o novo model para cadastro de eventos (**models/eventos.js**):

```
module.exports = function (app){

    var mongoose = require('mongoose');
    var Schema = mongoose.Schema;

    var evento = Schema({
        descricao: { type: String, required: true },
        data: { type: Date },
        preco: { type: Number }
    });
    return mongoose.model('eventos', evento);
};
```

9. Em **controllers/eventos.js**, realize as alterações para incluir um novo evento e para listar eventos:

```
module.exports = function (app) {  
  
    var Evento = app.models.eventos;  
  
    var EventosController = {  
        menu: function (request, response) {  
  
            var usuario = request.session.usuario,  
                params = { usuario: usuario };  
            response.render('eventos/menu', params);  
        },  
  
        cadastroUsuario: function (request, response) {  
            var usuario = request.session.usuario,  
                params = { usuario: usuario };  
            response.render('eventos/cadUsuario', params);  
        },  
  
        cadastroEvento: function (request, response) {  
            var usuario = request.session.usuario,  
                params = { usuario: usuario };  
            response.render('eventos/cadEvento', params);  
        },  
  
        listaEventos: function (request, response) {  
            Evento.find(function (erro, eventos) {  
                if (erro) {  
                    response.render('/menu');  
                }  
                else {  
                    var usuario = request.session.usuario,  
                        params = { usuario: usuario, eventos: eventos };  
                    response.render('eventos/listaEventos', params);  
                }  
            });  
        },  
    };  
};
```

```
//cadastro de eventos
novoEvento: function (request, response) {

    var evento = request.body.evento;
    if (evento.descricao.trim().length == 0 || evento.data == 'undefined' || evento.preco.trim().length == 0) {
        response.redirect('/cadEvento');
    }
    else {
        Evento.create(evento, function (erro, evento) {
            if (erro) {
                response.redirect('/cadEvento');
            }
            else {
                response.redirect('/menu');
            }
        });
    }
};

return EventosController;
};
```

10. Atualize o arquivo `views/eventos/listaEventos.ejs` para apresentar a lista de eventos real:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Cadastro de Eventos</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
    <header>
        <p>Bem-vindo,
           <%= usuario.nome %>
        </p>
        <h1>Lista de Eventos</h1>
        <h4>Selecione uma opção na lista para executá-la</h4>
    </header>
    <section>
        <!--será implementado em breve-->
        <table>
            <thead>
                <tr>
                    <th>Descrição</th>
                    <th>Data</th>
                    <th>Preço</th>
                </tr>
            </thead>
            <tbody>
                <% eventos.forEach(function(evento, index) { %>
                    <tr>
                        <td>
                            <%= evento.descricao %>
                        </td>
                        <td>
                            <%= evento.data %>
                        </td>
                        <td>
                            <%= evento.preco %>
                        </td>
                    </tr>
                <% }) %>
            </tbody>
        </table>
    </section>
    <section>
        <a href='/menu'>Voltar ao menu</a>
    </section>
    <footer>
        <small>Sistema de Cadastro de Eventos e Convidados</small>
    </footer>
</body>
</html>
```

Observe que, no trecho da instrução `eventos.forEach()`, a coleção `eventos` foi passada para a view no momento da sua renderização:

```
listaEventos: function (request, response) {
  Evento.find(function (erro, eventos) {
    if (erro) {
      response.render('/menu');
    }
    else {
      var usuario = request.session.usuario,
          params = { usuario: usuario, eventos: eventos };
      response.render('eventos/listaEventos', params);
    }
  });
},
```

11. Como opção, pesquise sobre formatação de datas e números, para melhorar a aparência da lista de eventos;

12. Teste a aplicação.

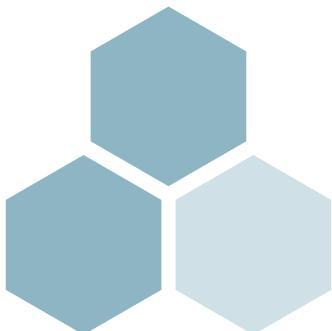


4

Criando e consumindo Web services

Atividade

Willian Moreira
305º



A – Criando um Web service para inclusão e consulta de eventos

Nesta fase do projeto, vamos ver como definir Web services para acesso a partir de outras aplicações ao nosso cadastro de eventos. Para tanto, criaremos um novo projeto exclusivo para o serviço.

1. Acesse a pasta **ProjetoEventos**. Nesta pasta, crie uma nova pasta chamada **WebServices**;

2. Estando na pasta **WebServices**, usando o prompt de comandos (ou o console que preferir, como o do VSCode, por exemplo), crie um novo projeto, com os comandos adiante:

```
express apiEventos --ejs
cd apiEventos
npm install
npm install body-parser --save
npm install express-load --save
npm install mongoose --save
```

3. Abra o VSCode apontando para a pasta **apiEventos**;

4. Crie a pasta **models**;

5. Inclua o arquivo **eventos.js**:

```
module.exports = function (app) {

    var mongoose = require('mongoose');
    var Schema = mongoose.Schema;

    var evento = Schema({
        descricao: { type: String, required: true },
        data: { type: Date },
        preco: { type: Number }
    });
    return mongoose.model('eventos', evento);
};
```

6. Substitua todo o conteúdo de **app.js** por este:

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

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

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

var mongoose = require('mongoose');
global.db = mongoose.connect('mongodb://localhost:27017/neventos');

load('models').into(app);

var Evento = app.models.eventos;

app.listen(3200, function () {
  console.log('ok');
});
```

7. Em **app.js**, acrescente os métodos de serviço:

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

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

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

var mongoose = require('mongoose');
global.db = mongoose.connect('mongodb://localhost:27017/neventos');

load('models').into(app);

var Evento = app.models.eventos;
```

```
//método do serviço
app.get('/', function (request, response) {
  response.send('Servidor no ar');
});
app.get('/eventos', function (request, response) {

});
app.get('/eventos/:id', function (request, response) {

});
app.post('/eventos', function (request, response) {

});
app.put('/eventos/:id', function (request, response) {

});
app.delete('/eventos/:id', function (request, response) {

});

app.listen(3200, function () {
  console.log('ok');
});
```

8. Vamos começar a codificar as funções para os serviços. Iniciaremos pela consulta aos eventos. Atualize os métodos `app.get()`:

```
//método do serviço
app.get('/', function (request, response) {
  response.send('Servidor no ar');
});
app.get('/eventos', function (request, response) {
  Evento.find(function (erro, eventos) {
    if (erro) {
      response.json(erro);
    }
    else {
      response.json(eventos);
    }
  });
});
```

```
app.get('/eventos/:id', function (request, response) {
  var id = request.params.id;

  Evento.findById(id, function (erro, evento) {
    if (erro) {
      response.json(erro);
    } else {
      response.json(evento);
    }
  });
});
```

9. Execute o comando a seguir:

```
node app.js
```

10. Teste o serviço, chamando a URL no browser (observe que colocamos uma porta diferente da aplicação. Isso é necessário porque podemos acessar este serviço em uma aplicação acessível por uma porta diferente, se estiverem no mesmo servidor):

```
localhost:3200/eventos
```

11. Verifique o id gerado para cada evento cadastrado. Copie o id de um evento desejado e execute a seguinte URL:

```
localhost:3200/eventos/5a6a0239dc740f18243a8294
```

Nesse exemplo, usamos o id **5a6a0239dc740f18243a8294**, mas isso não significa que ela exista quando você a executar, pois são valores diferentes para cada registro cadastrado.

12. Codifique os demais métodos:

```
app.post('/eventos', function (request, response) {
  var descricao = request.body.descricao;
  var data = request.body.data;
  var preco = request.body.preco;

  var evento = {
    'descricao': descricao,
    'data': data,
    'preco': preco
  };

  Evento.create(evento, function (erro, evento) {
    if (erro) {
      response.json(erro);
    }
  });
});
```

```
        else {
          response.json(evento);
        }
      );
    });

app.put('/eventos/:id', function (request, response) {
  var id = request.params.id;

  Evento.findById(id, function (erro, evento) {
    if (erro) {
      response.json(erro);
    }
    else {

      var evento_upd = evento;
      evento_upd.descricao = request.body.descricao;
      evento_upd.data = request.body.data;
      evento_upd.preco = request.body.preco;

      evento_upd.save(function (erro, evento) {
        if (erro) {
          response.json(erro);
        }
        else {
          response.json(evento);
        }
      });
      response.json(evento);
    }
  });
});

app.delete('/eventos/:id', function (request, response) {
  var id = request.params.id;

  Evento.findById(id, function (erro, evento) {
    if (erro) {
      response.json(erro);
    }
    else {
      Evento.remove(evento, function (erro, evento) {
        if (erro) {
          response.json(erro);
        }
        else {
          response.send('removido');
        }
      });
    }
  });
});
```

B – Consumindo o Web service a partir da aplicação

Para demonstrar o consumo de Web services, vamos completar nossa aplicação de cadastro de eventos para contemplar esta funcionalidade.

1. No VSCode, abra o projeto **nodeEventos**;
2. Acrescente um item no menu de opções, em **views/eventos/menu.ejs**:

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="utf-8">
    <title>Cadastro de Eventos</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>

  <body>
    <header>
      <p>Bem-vindo,<br/>
         <%= usuario.nome %></p>
      <h1>Menu de Opções</h1>
      <h4>Escolha sua opção</h4>
    </header>
    <section>
      <ul>
        <li>
          <a href="/cadUsuario">Cadastro de Usuários</a>
        </li>
        <li>
          <a href="/cadEvento">Cadastro de Eventos</a>
        </li>
        <li>
          <a href="/listaEventos">Lista de Eventos</a>
        </li>
        <li>
          <a href="/listaEventosWS">Lista de Eventos<br/>(WebService)</a>
        </li>
      </ul>
    </section>
    <section>
      <a href='/logout'>Logout</a>
    </section>
    <footer>
      <small>Sistema de Cadastro de Eventos e Convidados</small>
    </footer>
  </body>

</html>
```

3. No arquivo **routes/eventos.js**, acrescente a rota indicada:

```
module.exports = function (app) {

    var valida = require('../middlewares/valida');
    var eventos = app.controllers.eventos;

    app.get('/menu', valida, eventos.menu);

    app.get('/cadUsuario', valida, eventos.cadastroUsuario);
    app.get('/cadEvento', valida, eventos.cadastroEvento);
    app.get('/listaEventos', valida, eventos.listaEventos);

    app.get('/listaEventosWS', valida, eventos.listaEventosWS); // Nova rota

    app.post('/novoEvento', eventos.novoEvento);
};
```

4. Atualize o arquivo **controllers/eventos.js** para contemplar a nova rota, responsável pelo consumo do serviço. Para tanto, adicione o action (cuidado com as vírgulas!):

```
listaEventosWS: function (request, response) {
    //array para conter os eventos
    var eventos = [];

    //informações da requisição GET
    var info = {
        host: 'localhost',
        port: '3200',
        path: '/eventos',
        method: 'GET'
    };

    //chamando o serviço
    http.request(info, function (res) {
        res.setEncoding('utf8');
        res.on('data', function (data) {
            eventos = JSON.parse(data);

            var usuario = request.session.usuario,
                params = { usuario: usuario, eventos: eventos };
            response.render('eventos/listaEventosWS', params);
        });
    }).end();
},
```

Criando e consumindo Web services

5. Crie uma nova view, a ser renderizada pelo novo action no controller. A view deve ser criada em `views/eventos/listaEventosWS.ejs`. (Observe que se trata de uma cópia do arquivo `listaEventos.ejs`. Mudamos apenas o título!):

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>Cadastro de Eventos</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
</head>

<body>
    <header>
        <p>Bem-vindo, <%= usuario.nome %>
        </p>
        <h1>Lista de Eventos do Webservice</h1>
    </header>
    <section>
        <!--será implementado em breve-->
        <table>
            <thead>
                <tr>
                    <th>Descrição</th>
                    <th>Data</th>
                    <th>Preço</th>
                </tr>
            </thead>
            <tbody>
                <% eventos.forEach(function(evento, index) { %>
                    <tr>
                        <td>
                            <%= evento.descricao %>
                        </td>
                        <td>
                            <%= evento.data %>
                        </td>
                        <td>
                            <%= evento.preco %>
                        </td>
                    </tr>
                <% }) %>
            </tbody>
        </table>
    </section>
    <section>
        <a href='/menu'>Voltar ao menu</a>
    </section>
    <footer>
        <small>Sistema de Cadastro de Eventos e Convidados</small>
    </footer>
</body>

</html>
```

6. Mantenha o Web service em execução e teste a aplicação.

C – Adicionando recurso para pagamento do evento via Web service

Nesta parte da aplicação, criaremos, no Web service, um recurso para pagamento de um evento com cartão de crédito. Adicionaremos um link na lista de eventos para realizar o pagamento. O processo consiste em captar a descrição do evento, o preço do evento, passá-los como parâmetro via URL para uma nova view (**pagamentos.ejs**) e, a partir desta view, informaremos os dados do cartão e enviaremos todos os dados para o Web service.

1. Altere o arquivo **listarEventosWS.ejs** para acrescentar o link para a nova página. Este link informará a descrição e o preço do evento:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Cadastro de Eventos</title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>

<body>
  <header>
    <p>Bem-vindo, <%= usuario.nome %>
    </p>
    <h1>Lista de Eventos do Webservice</h1>
  </header>
  <section>
    <!--será implementado em breve-->
    <table>
      <thead>
        <tr>
          <th>Descrição</th>
          <th>Data</th>
          <th>Preço</th>
        </tr>
      </thead>
      <tbody>
        <% eventos.forEach(function(evento, index) { %>
          <tr>
            <td>
              <%= evento.descricao %>
            </td>
            <td>
              <%= evento.data %>
            </td>
            <td>
```

```

        <%= evento.preco %>
    </td>
    <td>
        <a href=
        "/pagamento/<%= evento.descricao %>/<%= evento.preco %>">
            comprar este evento</a>
        </td>
    </tr>
    <% }) %>
</tbody>
</table>
</section>
<section>
    <a href='/menu'>Voltar ao menu</a>
</section>
<footer>
    <small>Sistema de Cadastro de Eventos e Convidados</small>
</footer>
</body>

</html>

```

2. Altere o arquivo `routes/eventos.js` para contemplar essa nova rota. Adicionaremos, também, a rota a ser usada via método POST para efetivação do pagamento. Observe a implementação dos parâmetros na rota (GET):

```

module.exports = function (app) {

    var valida = require('../middlewares/valida');
    var eventos = app.controllers.eventos;

    app.get('/menu', valida, eventos.menu);

    app.get('/cadUsuario', valida, eventos.cadastroUsuario);
    app.get('/cadEvento', valida, eventos.cadastroEvento);
    app.get('/listaEventos', valida, eventos.listaEventos);

    app.get('/listaEventosWS', valida, eventos.listaEventosWS);

    app.post('/novoEvento', eventos.novoEvento);

    app.get('/pagamento/:evento/:preco', valida, eventos.pagamento);
    app.post('/novoPagamento', eventos.novoPagamento);
};


```

3. Atualize o **controller/eventos.js** para incluir os dois actions para essa tarefa:

```
pagamento: function (request, response) {
    var evento = request.params.evento,
        preco = request.params.preco,
        usuario = request.session.usuario,
        params = { usuario: usuario, evento: evento,
                   preco: preco };

    response.render('eventos/pagamento', params);
},
novoPagamento: function (request, response) {
    //a ser implementado
},
```

4. Crie a view **pagamento.ejs** (**views/eventos/pagamento.ejs**):

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>Pagamento de Eventos</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
</head>

<body>
    <header>
        <p>Bem-vindo, <%= usuario.nome %></p>
        <h1>Compra de Eventos</h1>
        <h4>Preencha os dados com seu cartão</h4>
    </header>
    <section>
        <form action="/novoPagamento" method="post">
            Evento:<br />
            <input type="text" name="cartao[evento]" value="<%= evento %>" />
            <br />Preço:<br />
            <input type="text" name="cartao[preco]" value="<%= preco %>" />
            <br> Num. Cartão:<br/>
            <input type="text" name="cartao[numcartao]" />
            <br> CVV: <br />
            <input type="text" name="cartao[cvv]" />
            <br>
            <button type="submit">Efetuar Pagamento</button>
        </form>
    </section>
    <footer>
        <small>Sistema de Cadastro de Eventos e Convidados</small>
    </footer>
</body>

</html>
```

Criando e consumindo Web services

5. Execute a aplicação, teste o link e verifique o valor sendo passado como parâmetro para **pagamento.ejs**:

6. Agora, vamos preparar o Web service. Crie o model **pagamentos.js**:

```
module.exports = function (app) {

  var mongoose = require('mongoose');
  var Schema = mongoose.Schema;

  var pagamento = Schema({
    evento: { type: String },
    preco: { type: Number },
    numcartao: { type: String },
    cvv: { type: String }
  });
  return mongoose.model('pagamentos', pagamento);
};
```

7. Atualize o action **novoPagamento**, em **controllers/eventos.js**. Neste action utilizaremos um novo serviço, a ser criado na próxima etapa;

```
novoPagamento: function (request, response) {
  //a ser implementado
  var cartao = request.body.cartao;

  var cartaoPost = JSON.stringify({
    'evento': cartao.evento,
    'preco': cartao.preco,
    'numcartao': cartao.numcartao,
    'cvv': cartao.cvv
  });

  //informações da requisição POST
  var info = {
    host: 'localhost',
    port: '3200',
    path: '/pagamentos',
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'Content-Length': cartaoPost.length
    }
};
```

```
//definição do pobjeto para requisição POST
var reqPost = http.request(info, function (res) {
    res.on('data', function (data) {
        console.log('Incluindo registros:\n');
        process.stdout.write(data);
        console.log('\n\nHTTP POST Concluído');
    });
});

//Gravação dos dados
reqPost.write(cartaoPost);
response.redirect('/menu');
reqPost.end();
reqPost.on('error', function (e) {
    console.error(e);
});
},

```

8. No projeto **apiEventos**, adicione, na pasta **models**, o model **pagamentos.js** (o mesmo usado no projeto **nodeEventos**):

```
module.exports = function (app) {

    var mongoose = require('mongoose');
    var Schema = mongoose.Schema;

    var pagamento = Schema({
        evento: { type: String },
        preco: { type: Number },
        numcartao: { type: String },
        cvv: { type: String }
    });
    return mongoose.model('pagamentos', pagamento);
};
```

9. Atualize o arquivo **app.js**, incluindo uma referência a este model, além das funções para incluir um novo pagamento e para listar pagamentos:

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

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

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

var mongoose = require('mongoose');
global.db = mongoose.connect('mongodb://localhost:27017/neventos');

load('models').into(app);

var Evento = app.models.eventos;
var Pagamento = app.models.pagamentos;

//método do serviço

//pagamentos
app.get('/pagamentos', function (request, response) {
  Pagamento.find(function (erro, pagamento) {
    if (erro) {
      response.json(erro);
    } else {
      response.json(pagamento);
    }
  });
});

app.post('/pagamentos', function (request, response) {

  var evento = request.body.evento;
  var preco = request.body.preco;
  var numcartao = request.body.numcartao;
  var cvv = request.body.cvv;

  var pagamento = {
    'evento': evento,
    'preco': preco,
    'numcartao': numcartao,
    'cvv': cvv
  };
});
```

```
Pagamento.create(pagamento, function (erro, pagto) {  
  if (erro) {  
    response.json(erro);  
  }  
  else {  
    response.json(pagto);  
  }  
});  
});  
  
app.listen(3200, function () {  
  console.log('ok');  
});
```

10. Teste a aplicação;

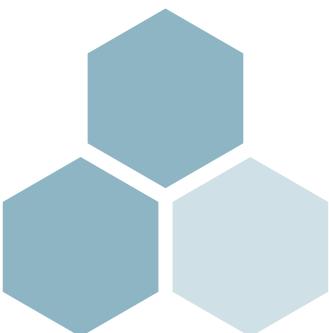
11. Como tarefa, implemente a lista de pagamentos no projeto **nodeEventos**.



5

Definindo uma aplicação com AngularJS – MEAN (Parte 1)

Atividade



A – Criando uma aplicação com Node.js e Express.js, com a view implementando o AngularJS

Neste projeto, trabalharemos na camada de visualização. Definiremos um novo projeto baseado no Node.js com Express.js, acessando um Web service capaz de acessar dados do banco de dados MongoDB. A interação com o usuário será realizada por meio de uma única página, contemplando o recurso SPA (Single Page App) com AngularJS. Este projeto, por usar o Mongo, o Express, o Angular e o Node, será considerado uma aplicação MEAN Stack.

1. Na pasta **ProjetoEventos**, crie uma nova pasta chamada **Mean_AngularJS**;
2. Usando o Express, crie um novo projeto chamado **appAngularJS**. Use a sequência de comandos adiante:

```
express appAngularJS --ejs  
cd appAngularJS  
npm install  
npm install body-parser --save  
npm install express-load --save
```

3. Abra o VSCode na pasta **appAngularJS**;
4. Crie as pasta **controllers**;
5. Apague o conteúdo das pastas **routes** e **views**;
6. Na pasta **routes**, crie o arquivo **eventos.js**:

```
module.exports = function (app) {  
    var evento = app.controllers.eventos;  
    app.get('/', evento.index);  
};
```

7. Na pasta **controllers**, crie o arquivo **eventos.js**:

```
module.exports = function (app) {  
    var EventosController = {  
        index: function (req, res) {  
            res.render('eventos/index');  
        }  
    };  
    return EventosController;  
};
```

8. Atualize o arquivo **app.js**:

```
var express = require('express');
var load = require('express-load');
app = express();

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

load('controllers')
  .then('routes')
  .into(app);

app.listen(3000, function () {
  console.log("Aplicação no ar.");
});
```

9. Crie a view **views/eventos/index.ejs**:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel='stylesheet' href='/stylesheets/style.css' />
  <title>Aplicação AngularJS</title>
</head>
<body>
  <h1>Cadastro e Consulta de Eventos</h1>

</body>
</html>
```

10. Teste a aplicação até este ponto, para checar se está tudo bem. Execute, no prompt de comandos, **node app.js** e, no browser, **localhost:3000**;

11. Vamos implementar as funcionalidades do AngularJS. Na pasta **public/javascripts**, inclua a referência à biblioteca do AngularJS (arquivo **angular.js**). Faça o download desta biblioteca no link <https://angularjs.org/>. Inclua esta referência no arquivo **index.ejs**:

```
<!DOCTYPE html>
<html ng-app>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <title>Aplicação AngularJS</title>
</head>
<body>
    <h1>Cadastro e Consulta de Eventos</h1>

    <script src="/javascripts/angular.js"></script>
</body>
</html>
```

12. Esta aplicação utilizará o Web service desenvolvido no projeto anterior. Para que o Web service seja consumido por outra plataforma diferente daquela na qual foi desenvolvido, é necessário habilitar o recurso **Cross-Origin**, conhecido como **CORS**. Antes de prosseguir com a aplicação, abra o projeto **apiEventos**:

13. No prompt de comandos, execute o seguinte comando:

```
npm install cors --save
```

14. No arquivo **app.js** do Web service, realize as seguintes alterações:

```
var express = require('express');
var load = require('express-load');
var cors = require('cors');

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

app.use(cors());

app.use(function (req, res, next) {
    res.header("Access-Control-Allow-Origin", "*");
    res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With,
Content-Type, Accept");
    next();
});
```

15. O Web service deve ser reiniciado;

16. Retornando ao nosso projeto AngularJS, vamos, agora, incluir uma tabela na nossa view. O objetivo é usar o Web service desenvolvido no projeto anterior e já atualizado com o módulo **cors**, por meio do módulo **http** do AngularJS. Altere o arquivo **index.ejs**, incluindo as implementações do módulo e do controller do Angular:

```
<!DOCTYPE html>
<html ng-app="appAngular">

  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <style>
      table,
      tr,
      td,
      th {
        border: 1px solid #ccc;
        padding: 10px;
        margin: 10px;
      }
    </style>
    <title>Aplicação AngularJS</title>
  </head>

  <body ng-controller="Principal as ctl">
    <h1>Cadastro e Consulta de Eventos</h1>
    <h2>Lista de eventos</h2>

    <table>
      <thead>
        <tr>
          <th>Descrição</th>
          <th>Data</th>
          <th>Preço</th>
        </tr>
      </thead>

      <tbody>
        <tr ng-repeat="evento in ctl.items">
          <td>{{evento.descricao}}</td>
          <td>{{evento.data}}</td>
          <td>{{evento.preco}}</td>
        </tr>
      </tbody>
    </table>
  </body>
```

```
<script src="/javascripts/angular.js"></script>
<script type="text/javascript">

    angular.module("appAngular", [])
        .controller('Principal', ['$http', function ($http) {

            var self = this;

            self.items = [];

            var listarTodos = function () {
                return $http.get('http://localhost:3200/eventos/')
            .then(function (response) {
                self.items = response.data;
            }, function (error) {
                alert('Erro reportado: ' + error);
            });
        };

        listarTodos();
    }]);
</script>
</body>

</html>
```

17. Ao executar a aplicação, podemos visualizar a lista de eventos na página. A chamada foi executada na carga da página, mas sua execução foi realizada de forma assíncrona;

18. O próximo passo é definir um formulário que permita a inclusão de um novo evento. O formulário será desenvolvido na mesma página que a listagem, caracterizando, assim, uma aplicação SPA. Adicione o formulário e a função JavaScript, conforme modelo a seguir:

```
<!DOCTYPE html>
<html ng-app="appAngular">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <style>
        table,
        tr,
        td,
        th {
            border: 1px solid #ccc;
            padding: 10px;
            margin: 10px;
        }
    </style>
```

```

</style>
<title>Aplicação AngularJS</title>
</head>

<body ng-controller="Principal as ctl">
    <h1>Cadastro e Consulta de Eventos</h1>
    <h2>Lista de eventos</h2>

        <table>
            <thead>
                <tr>
                    <th>Descrição</th>
                    <th>Data</th>
                    <th>Preço</th>
                </tr>
            </thead>
            <tbody>
                <tr ng-repeat="evento in ctl.items">
                    <td>{{evento.descricao}}</td>
                    <td>{{evento.data | date:'dd/MM/yyyy'}}</td>
                    <td>{{evento.preco | currency}}</td>
                </tr>
            </tbody>
        </table>

        <h2>Inclusão de um novo evento</h2>
        <div>
            <form name="incluirForm" ng-submit="ctl.adicionar()">
                Descrição do evento:<br/>
                <input type="text" ng-model="ctl.novoEvento.descricao" />
                <br/>Data (dd/mm/yyyy):<br />
                <input type="date" ng-model="ctl.novoEvento.data" />
                <br/>Preço:<br />
                <input type="text" ng-model="ctl.novoEvento.preco" />
                <br />
                <input type="submit" value="Adicionar" />
            </form>
        </div>

<script src="/javasCript/angular.js"></script>
<script type="text/javasCript">

    angular.module("appAngular", [])
        .controller('Principal', ['$http', function ($http) {

            var self = this;

            self.items = [];
            self.novoEvento = {};
        }]);

```

```
var listarTodos = function () {
    return $http.get('http://localhost:3200/eventos/')
        .then(function (response) {
            self.items = response.data;
        }, function (error) {
            alert('Erro reportado: ' + error);
        });
};

listarTodos();

self.adicionar = function () {
    $http({
        url: 'http://localhost:3200/eventos/',
        method: 'POST',
        data: self.novoEvento,
        headers: { 'Content-Type': 'application/json' }
    }).then(function (response) {
        self.novoEvento = {};
    }, function (error) {
        alert('Erro reportado: ' + error);
    }).then(listarTodos);
};

}]);
</script>
</body>

</html>
```

19. Execute a aplicação e inclua alguns eventos. Observe que a lista é atualizada tão logo o evento seja incluído;

20. Para completar, vamos apresentar uma formatação mais adequada para a data e para a moeda. Para tanto, aplicaremos o conceito de filtros. Vamos alterar a tabela de valores para contemplar os filtros:

```
<table>
    <thead>
        <tr>
            <th>Descrição</th>
            <th>Data</th>
            <th>Preço</th>
        </tr>
    </thead>
    <tbody>
        <tr ng-repeat="evento in ctl.items">
            <td>{{evento.descricao}}</td>
            <td>{{evento.data | date:'dd/MM/yyyy'}}</td>
            <td>{{evento.preco | currency }}</td>
        </tr>
    </tbody>
</table>
```

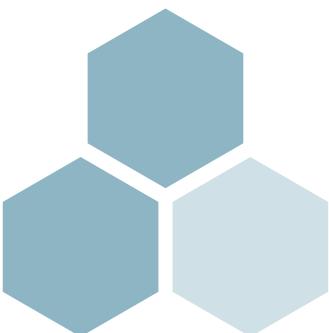
21. Visualize.



6

Definindo uma aplicação com Angular 4 – MEAN (Parte 2)

Atividade



A – Criando uma aplicação Angular 4 com Node.js

Vamos continuar aplicando o conceito do MEAN, só que, desta vez, implementaremos um projeto Angular 4. Este projeto também trata do cadastro de eventos. A diferença é que tanto a inclusão quanto o cadastro serão realizados por meio do Web service que desenvolvemos.

1. Na pasta **ProjetoEventos**, crie uma nova pasta chamada **Mean_Angular4**;
2. No prompt de comandos, acesse a pasta recém-criada;
3. Vamos criar um projeto chamado **appAngular4**, por meio do template **quickstart** disponível no GitHub. Execute o seguinte comando:

```
git clone https://github.com/angular/quickstart.git appAngular4
```

4. Abra o VSCode na pasta do projeto. No prompt de comandos, entre nessa pasta também, pois executaremos a aplicação a partir dela. Lembrando que é possível usar o próprio VSCode para essa finalidade;
5. No prompt, execute isto (somente se o projeto foi gerado via quickstart):

```
npm install
```

6. Instale a biblioteca do Bootstrap e do jQuery. Estas serão úteis na camada de visualização:

```
npm install bootstrap@3.3.7 --save  
npm install jquery --save
```

7. Abra o arquivo **index.html** (nossa página inicial) e configure as bibliotecas Bootstrap e jQuery recém-inseridas;

```
<!DOCTYPE html>  
<html>  
  
<head>  
  <title>Angular QuickStart</title>  
  <base href="/">  
  <meta charset="UTF-8">  
  <meta name="viewport" content="width=device-width, initial-scale=1">  
  
  <link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.min.css">  
  <script src="node_modules/jquery/dist/jquery.min.js"></script>  
  <script src="node_modules/bootstrap/dist/js/bootstrap.min.js"></script>
```

```
<link rel="stylesheet" href="styles.css">

<!-- Polyfill(s) for older browsers -->
<script src="node_modules/core-js/client/shim.min.js"></script>

<script src="node_modules/zone.js/dist/zone.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>

<script src="systemjs.config.js"></script>
<script>
  System.import('main.js').catch(function (err) { console.error(err); });
</script>
</head>

<body>
  <my-app>Loading AppComponent content here ...</my-app>
</body>

</html>
```

8. Nossa componente principal (**AppComponent**) define um texto, inserido no **index.html**. Vamos acrescentar dois componentes: um representando o menu de opções (**MenuComponent**) e outro, a página inicial de fato na nossa aplicação (**HomeComponent**), uma espécie de logotipo e apresentação da empresa. Para isso, crie pasta **menu**, abaixo de **app** (nossa raiz);

9. Na pasta **menu/views**, crie **menu.component.html** com o código adiante:

```
<div class="navbar navbar-default navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle"
        data-toggle="collapse" data-target=".navbar-collapse">
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="#">Impacta - Mean Stack</a>
    </div>
    <div class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li>
          <a href="#">Home</a>
        </li>
        <li>
          <a href="#">Gerenciar Eventos</a>
        </li>
      </ul>
    </div>
  </div>
</div>
```

10. Na pasta **menu**, defina o componente **menu.component.ts**:

```
import { Component } from '@angular/core';

@Component({
  moduleId: module.id,
  selector: 'menu',
  templateUrl: 'views/menu.component.html'
})
export class MenuComponent { }
```

11. Abra o arquivo **app.component.ts** e realize as alterações:

```
import { Component } from '@angular/core';
import { MenuComponent } from './menu/menu.component';

@Component({
  selector: 'my-app',
  template: '<menu></menu>'
})
export class AppComponent { }
```

12. Devemos, agora, configurar o novo componente em **app.module.ts**. Neste arquivo, configuraremos todos os componentes, serviços e módulos da aplicação:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, MenuComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

O componente principal incluirá outro componente, no caso, o **MenuComponent**. Para que isso seja possível, é necessário incluí-lo no metadata **declarations**.

13. Execute a aplicação. No prompt (na pasta do projeto), execute este comando (somente se criou o projeto pelo quickstart):

```
npm start
```

As alterações realizadas no projeto não requerem que esse comando seja executado novamente: um dos componentes do Angular 4 é o **BrowserLink**, usado para recarregar a página automaticamente. Esse componente monitora as alterações no código e recarrega o componente.

14. Até agora, nossas classes não possuíam nenhum código. No nosso exemplo, o conteúdo dos links será obtido a partir de variáveis definidas na classe. Veja as alterações realizadas na classe **MenuComponent** e no arquivo **menu.component.html**:

```
import { Component } from '@angular/core';

@Component({
  moduleId: module.id,
  selector: 'menu',
  templateUrl: 'views/menu.component.html'
})
export class MenuComponent {
  titulo_empresa: string = "Impacta Treinamentos";
  titulo_home: string = "Home";
  titulo_principal: string = "Gestão de Eventos";
}

<div class="navbar navbar-default navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle"
data-toggle="collapse" data-target=".navbar-collapse">
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand"
        [routerLink]=["'/'"]>{{titulo_empresa}}</a>
    </div>
    <div class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li>
          <a [routerLink]=["'/home']>{{titulo_home}}</a>
        </li>
        <li>
          <a [routerLink]=["'/eventos']>{{titulo_principal}}</a>
        </li>
      </ul>
    </div>
  </div>
</div>
<router-outlet></router-outlet>
```

15. Altere o conteúdo do arquivo **styles.css**:

```
h1 {  
    color: #369;  
    font-family: Arial, Helvetica, sans-serif;  
    font-size: 250%;  
}  
  
.margem {  
    margin-top: 50px;  
}
```

16. Vamos definir a página inicial da aplicação, o **home**. Crie uma pasta chamada **home**, abaixo de **app**. Nessa pasta, defina o arquivo **home.component.ts**:

```
import { Component } from '@angular/core';  
  
@Component({  
    template: `  
        <div class="container margem">  
            <h1>PÁGINA INICIAL</h1>  
        </div>  
    `)  
export class HomeComponent {}
```

17. Crie a pasta **cadastro** e, nela, o arquivo **cadastro.component.ts**:

```
import { Component } from '@angular/core';  
  
@Component({  
    template: `  
        <div class="container margem">  
            <h1>CADASTRO DE EVENTOS</h1>  
        </div>  
    `)  
export class CadastroComponent {}
```

18. Inclua um componente para representar uma URL inválida (erro 404). Crie uma pasta chamada **erro** e, nela, o arquivo **notfound.component.ts**:

```
import { Component } from '@angular/core';  
  
@Component({  
    template: `  
        <div class="container margem">  
            <h1>ERRO 404 - PÁGINA NÃO LOCALIZADA</h1>  
        </div>  
    `)  
export class NotFoundComponent {}
```

19. Vamos definir as rotas da aplicação. Crie uma pasta chamada **rotas** e, nesta pasta, o arquivo **app.routes.ts**. Analise o conteúdo:

```
import { Routes } from '@angular/router';
import { HomeComponent } from '../home/home.component';
import { CadastroComponent } from '../cadastro/cadastro.component';
import { NotFoundComponent } from '../erro/notFound.component';

export const appRoutes: Routes = [
  { path: "", component: HomeComponent },
  { path: "eventos", component: CadastroComponent },
  { path: "home", component: HomeComponent },
  { path: "**", component: NotFoundComponent }
];
```

20. O próximo passo é alterar o módulo da aplicação, arquivo **app.module.ts**. Analise as alterações realizadas:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';

import { appRoutes } from './rotas/app.routes' //deve vir primeiro

import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';

//usado nas rotas
import { HomeComponent } from './home/home.component';
import { CadastroComponent } from './cadastro/cadastro.component';
import { NotFoundComponent } from './erro/notFound.component';

@NgModule({
  //lista os modulos que a aplicação necessitara
  imports: [BrowserModule, RouterModule.forRoot(appRoutes)],

  //lista os componentes que nossa aplicação utilizará
  declarations: [AppComponent,
    MenuComponent,
    HomeComponent,
    CadastroComponent,
    NotFoundComponent],

  //este é o componente inicial, incluído no index.html
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

21. Teste a aplicação até este ponto, executando cada uma das rotas. O que é possível verificar?

22. Para começar a manipular os dados, vamos criar uma estrutura para armazenar as informações do evento. Para tanto, vamos criar uma interface que se chamará **IEvento**. Portanto, crie o arquivo **interface.evento.ts** na pasta **interfaces**:

```
export interface IEvento {  
    descricao: string;  
    data: string;  
    preco: number;  
}
```

23. Na pasta **cadastro**, crie uma subpasta chamada **views**. Nesta pasta, crie o arquivo **cadastro.component.html**:

```
<div class="container margem">  
    <h1>Cadastro de Eventos</h1>  
    <hr/>  
    <div class="row">  
        <div class="col-md-6">  
            <ul class="list_group">  
                <li *ngFor="let item of listaEventos"  
                    class="list-group-item">  
                    <a href="#">{{item.descricao}}</a>  
                </li>  
            </ul>  
        </div>  
  
        <div class="col-md-6">  
            <div class="form-group">  
                <label for="descricao">Descrição:</label>  
                <input type="text" class="form-control" id="descricao" name="descricao">  
            </div>  
            <div class="form-group">  
                <label for="data">Data:</label>  
                <input type="text" class="form-control" id="data" name="data">  
            </div>  
            <div class="form-group">  
                <label for="preco">Preço:</label>  
                <input type="number" class="form-control" id="preco" name="preco">  
            </div>  
  
            <div class="form-group">  
                <button type="button" class="btn btn-info">  
                    <span class="glyphicon glyphicon-pencil"></span>  
                    Incluir  
                </button>  
            </div>  
        </div>  
    </div>  
</div>
```

24. No arquivo **cadastro.component.ts**, crie uma lista de eventos provisória. Faça as alterações sugeridas no código a seguir:

```
import { Component } from '@angular/core';
import { IEvento } from './../../../interfaces/interface.evento';

@Component({
  moduleId: module.id,
  templateUrl: 'views/cadastro.component.html'
})

export class CadastroComponent {
  //definindo um array de eventos

  public listaEventos: IEvento[] = [
    { descricao:'Avaliação Angular', data: '23/10/2018', preco:0 },
    { descricao: 'Formatura', data: '02/05/2020', preco: 140 },
    { descricao: 'Torneio de Tenis', data: '10/07/2018', preco: 210 },
    { descricao: 'Congresso de TI', data: '16/01/2019', preco: 400 }
  ];
}
```

25. Execute a aplicação e acesse **Gestão de Eventos**. É possível visualizar a lista de eventos do lado esquerdo e um formulário do lado direito;

26. Tente acessar, na URL, um link inexistente. Verifique se o componente **NotFoundComponent** é renderizado;

27. Vamos transferir os dados da lista de eventos para um serviço. Para tanto, crie uma pasta chamada **services**. Nesta pasta, crie o arquivo **eventos.service.ts**:

```
import { Injectable } from '@angular/core';
import { IEvento } from './../../../interfaces/interface.evento';

@Injectable()
export class EventosService {
  public getEventos(): IEvento[] {
    return [
      { descricao: 'Avaliação Angular', data: '23/10/2018', preco: 0 },
      { descricao: 'Formatura', data: '02/05/2020', preco: 140 },
      { descricao: 'Torneio de Tenis', data: '10/07/2018', preco: 210 },
      { descricao: 'Congresso de TI', data: '16/01/2019', preco: 400 }
    ];
  }
}
```

28. No arquivo `cadastro.component.ts`, faça as alterações indicadas:

```
import { Component } from '@angular/core';
import { IEvento } from './../interfaces/interface.evento';
import { EventosService } from '../services/eventos.service';

@Component({
  moduleId: module.id,
  templateUrl: 'views/cadastro.component.html'
})

export class CadastroComponent {
  public listaEventos: IEvento[];
  constructor(eventosService: EventosService) {
    this.listaEventos = eventosService.getEventos();
  }
}
```

29. No arquivo `app.module.ts`, inclua o serviço com provider:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule }  from '@angular/router';

import { appRoutes } from './rotas/app.routes'; //deve vir primeiro

import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';

//usado nas rotas
import { HomeComponent } from './home/home.component';
import { CadastroComponent } from './cadastro/cadastro.component';
import { NotFoundComponent } from './erro/notFound.component';
import { EventosService } from './services/eventos.service';

@NgModule({
  //lista os modulos que a aplicação necessitara
  imports: [BrowserModule, RouterModule.forRoot(appRoutes)],

  //lista os componentes que nossa aplicação utilizará
  declarations: [AppComponent,
    MenuComponent,
    HomeComponent,
    CadastroComponent,
    NotFoundComponent],


  providers : [ EventosService ],
  //este é o componente inicial, incluído no index.html
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

30. Agora, vamos selecionar um item na lista de eventos, vê-lo no formulário e, na medida em que alterarmos alguma função no formulário, vê-la na lista. Esse processo caracteriza um vínculo bidirecional, ou um binding bidirecional. Para usar o binding bidirecional, é necessário habilitar o uso do atributo **ngModel**. Para tanto, realize as alterações no arquivo **app.module.ts**:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';
import { FormsModule } from '@angular/forms';

import { appRoutes } from './rotas/app.routes'; //deve vir primeiro

import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';

//usado nas rotas
import { HomeComponent } from './home/home.component';
import { CadastroComponent } from './cadastro/cadastro.component';
import { NotFoundComponent } from './erro/notFound.component';
import { EventosService } from './services/eventos.service';

@NgModule({
  //lista os modulos que a aplicação necessitara
  imports: [BrowserModule, RouterModule.forRoot(appRoutes), FormsModule],
  //lista os componentes que nossa aplicação utilizará
  declarations: [AppComponent,
    MenuComponent,
    HomeComponent,
    CadastroComponent,
    NotFoundComponent],
  providers : [ EventosService ],
  //este é o componente inicial, incluído no index.html
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

31. Realize, agora, as alterações em **cadastro.component.ts**:

```
import { Component } from '@angular/core';
import { IEvento } from './../../../interfaces/interface.evento';
import { EventosService } from '../services/eventos.service';

@Component({
  moduleId: module.id,
  templateUrl: 'views/cadastro.component.html'
})

export class CadastroComponent {
  //para um evento selecionado
  public eventoSelecionado: IEvento;

  public selecionar(item: IEvento): void {
    this.eventoSelecionado = item;
  }

  //lista de eventos
  public listaEventos: IEvento[];
  constructor(eventosService: EventosService) {
    this.listaEventos = eventosService.getEventos();
  }
}
```

32. Em seguida, providencie as alterações no arquivo **cadastro.component.html**:

```
<div class="container margem">
  <h1>Cadastro de Eventos</h1>
  <hr/>
  <div class="row">
    <!-- incluindo evento click na lista -->
    <div class="col-md-6">
      <ul class="list_group">
        <li *ngFor="let item of listaEventos"
            (click)="selecionar(item)" class="list-group-item">
          <a [class.selecionado]="item === eventoSelecionado"
             [routerLink]=["'/eventos']>{{item.descricao}}</a>
        </li>
      </ul>
    </div>
```

```

<!-- incluindo o formulário somente se houver elemento selecionado -->
<!-- nova div envolvendo <div class="col-md-6"> -->
<div *ngIf="eventoSelecionado">
    <div class="col-md-6">
        <div class="form-group">
            <label for="descricao">Descrição:</label>
            <input type="text" [(ngModel)]="eventoSelecionado.descricao"
                   class="form-control" id="descricao" name="descricao">
        </div>
        <div class="form-group">
            <label for="data">Data:</label>
            <input type="text" [(ngModel)]="eventoSelecionado.data"
                   class="form-control" id="data" name="data">
        </div>
        <div class="form-group">
            <label for="preco">Preço:</label>
            <input type="number" [(ngModel)]="eventoSelecionado.preco"
                   class="form-control" id="preco" name="preco">
        </div>

        <div class="form-group">
            <button type="button" class="btn btn-info">
                <span class="glyphicon glyphicon-pencil"></span>
                Incluir
            </button>
        </div>
    </div>
</div>

</div>
</div>

```

33. É interessante que o usuário tenha uma pista de qual item foi selecionado. Vamos tornar negrito o item selecionado. Adicione em **styles.css** este código:

```

h1 {
    color: #369;
    font-family: Arial, Helvetica, sans-serif;
    font-size: 250%;
}

.margem {
    margin-top: 50px;
}

.selecionado {
    font-weight: bold;
}

```

34. Teste esta nova funcionalidade;

35. Vamos, agora, permitir a inclusão de um novo item a ser cadastrado (pois só podemos visualizar uma lista pronta!). No arquivo **cadastro.component.ts**, realize a seguinte alteração:

```
import { Component } from '@angular/core';
import { IEvento } from './interfaces/interface.evento';
import { EventosService } from '../services/eventos.service';

@Component({
  moduleId: module.id,
  templateUrl: 'views/cadastro.component.html'
})

export class CadastroComponent {
  //para um evento selecionado
  public eventoSelecionado: IEvento;

  private novoEvento: IEvento;

  //para a inclusão de um novo evento
  public novo() {
    this.novoEvento = { descricao: '', data: '', preco: 0 };
    this.eventoSelecionado = this.novoEvento;
  }

  public incluir(evento: IEvento) {
    this.listaEventos.push(evento);
    alert('Evento incluído com sucesso');
  }

  public selecionar(item: IEvento): void {
    this.eventoSelecionado = item;
  }

  //lista de eventos
  public listaEventos: IEvento[];
  constructor(eventosService: EventosService) {
    this.listaEventos = eventosService.getEventos();
  }
}
```

36. No arquivo **cadastro.component.html**, realize as alterações (observe a inclusão de um novo botão e da chamada aos eventos):

```

<div class="container margem">
    <h1>Cadastro de Eventos</h1>
    <hr/>
    <div class="row">
        <!-- incluindo evento click na lista -->
        <div class="col-md-6">
            <ul class="list_group">
                <li *ngFor="let item of listaEventos"
                    (click)="selecionar(item)" class="list-group-item">
                    <a [class.selecionado]="item === eventoSelecionado"
                       [routerLink]=["'/eventos']>{{item.descricao}}</a>
                </li>
            </ul>

            <button type="button" (click)="novo()" class="btn btn-info">
                <span class="glyphicon glyphicon-plus"></span>
                Novo
            </button>
        </div>

        <!-- incluindo o formulário somente se houver elemento selecionado -->
        <!-- nova div envolvendo <div class="col-md-6"> -->
        <div *ngIf="eventoSelecionado">
            <div class="col-md-6">
                <div class="form-group">
                    <label for="descricao">Descrição:</label>
                    <input type="text" [(ngModel)]="eventoSelecionado.descricao"
                           class="form-control" id="descricao" name="descricao">
                </div>
                <div class="form-group">
                    <label for="data">Data:</label>
                    <input type="text" [(ngModel)]="eventoSelecionado.data"
                           class="form-control" id="data" name="data">
                </div>
                <div class="form-group">
                    <label for="preco">Preço:</label>
                    <input type="number" [(ngModel)]="eventoSelecionado.preco"
                           class="form-control" id="preco" name="preco">
                </div>

                <div class="form-group">
                    <button type="button" (click)="incluir(eventoSelecionado)"
                           class="btn btn-info">
                        <span class="glyphicon glyphicon-pencil"></span>
                        Incluir
                    </button>
                </div>
            </div>
        </div>
    </div>
</div>

```

B – Aplicando filtros nos componentes

Filtros são elementos que auxiliam na apresentação da view para o usuário. Por meio de filtros, podemos formatar datas, horas, apresentar subconjuntos, retornar textos em maiúsculo etc. Podemos usar filtros padrão, ou criar os nossos próprios. Os principais filtros padrão são estes:

- **uppercase**: Retorna a propriedade em maiúsculo;
- **lowercase**: Retorna a propriedade em minúsculo;
- **percent**: Apresenta um valor decimal com símbolo de porcentagem (%);
- **currency**: Especifica o valor da propriedade com moeda local;
- **date**: Apresenta um dado no formato de data (se compatível).

1. No arquivo **cadastro.component.html**, aplique o filtro **uppercase** na descrição:

```
<!-- incluindo evento click na lista -->
<div class="col-md-6">
    <ul class="list_group">
        <li *ngFor="let item of listaEventos"
            (click)="selecionar(item)" class="list-group-item">
            <a [class.selecionado]="item == eventoSelecionado"
               [routerLink]=["/eventos"]>
                {{item.descricao | uppercase}}</a>
        </li>
    </ul>

    <button type="button" (click)="novo()" class="btn btn-info">
        <span class="glyphicon glyphicon-plus"></span>
        Novo
    </button>
</div>
```

2. Visualize;

3. Vamos criar nosso próprio filtro. O objetivo é obter uma sublistas de eventos, com base em um texto digitado em um campo de textos. Crie uma pasta chamada **filters**. Na pasta **filters**, crie o arquivo **sublista.filter.ts**:

```
import { Pipe, PipeTransform } from '@angular/core';
import { IEvento } from '../interfaces/interface.evento';
@Pipe({
  name: 'sublista'
})
export class SubLista implements PipeTransform {
  transform(eventos: IEvento[], input: string): IEvento[] {
    //usando arrow function (similar a delegates do c#)
    return eventos.filter(
      evento =>
      evento.descricao.toLowerCase().includes(input.toLowerCase()));
  }
}
```

4. Altere o arquivo **app.module.ts**:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';
import { FormsModule } from '@angular/forms';

import { appRoutes } from './rotas/app.routes' //deve vir primeiro

import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';
import { SubLista } from './filters/sublista.filter';

//usado nas rotas
import { HomeComponent } from './home/home.component';
import { CadastroComponent } from './cadastro/cadastro.component';
import { NotFoundComponent } from './erro/notFound.component';
import { EventosService } from './services/eventos.service';

@NgModule({
  //lista os modulos que a aplicação necessitara
  imports: [BrowserModule, RouterModule.forRoot(appRoutes), FormsModule],
  //lista os componentes que nossa aplicação utilizará
  declarations: [AppComponent,
    MenuComponent,
    HomeComponent,
    CadastroComponent,
    NotFoundComponent, SubLista],
  providers : [ EventosService ],
  //este é o componente inicial, incluído no index.html
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

5. No arquivo **cadastro.component.html**, realize as alterações a seguir (inclusa a caixa de textos indicada):

```
<div class="container margem">
    <h1>Cadastro de Eventos</h1>
    <hr/>
    <div class="row">
        <!-- incluindo evento click na lista -->
        <div class="col-md-6">

            <div class="form-group">
                <label for="codigo">Informe a descrição para filtrar:</label>
                <input type="text" #busca (keyup)="0" class="form-control" >
            </div>

            <ul class="list_group">
                <li *ngFor="let item of listaEventos | sublista:busca.value"
                    (click)="selecionar(item)" class="list-group-item">
                    <a [class.selecionado]="item == eventoSelecionado"
                       [routerLink]=["'/eventos']>{{item.descricao |
                        uppercase}}</a>
                </li>
            </ul>

            <button type="button" (click)="novo()" class="btn btn-info">
                <span class="glyphicon glyphicon-plus"></span>
                Novo
            </button>
        </div>
    //continua....
```

6. Teste a aplicação com filtros.

C – Obtendo a lista de eventos do Web service appEventos

1. Em `app.module.ts`, importe e configure o módulo `HttpModule`:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';
import { FormsModule } from '@angular/forms';

import { HttpClientModule } from '@angular/http'; // Importado aqui

import { appRoutes } from './rotas/app.routes'; // deve vir primeiro

import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';
import { SubLista } from './filters/sublista.filter';

// usado nas rotas
import { HomeComponent } from './home/home.component';
import { CadastroComponent } from './cadastro/cadastro.component';
import { NotFoundComponent } from './erro/notFound.component';
import { EventosService } from './services/eventos.service';

@NgModule({
  // lista os modulos que a aplicação necessitara
  imports: [BrowserModule, RouterModule.forRoot(appRoutes),
            FormsModule, HttpClientModule], // Importado aqui

  // lista os componentes que nossa aplicação utilizará
  declarations: [AppComponent,
                 MenuComponent,
                 HomeComponent,
                 CadastroComponent,
                 NotFoundComponent, SubLista],

  providers : [ EventosService ],
  // este é o componente inicial, incluído no index.html
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

2. Em `eventos.service.ts`, realize as alterações:

```
import { Injectable } from '@angular/core';
import { IEvento } from '../interfaces/interface.evento';

import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Rx';
import 'rxjs/Rx';

@Injectable()
export class EventosService {

    //acesso ao HTTP
    public constructor(private _http: Http) {}
    private url: string = "http://localhost:3200/eventos";

    public getEventosWS(): Observable<IEvento[]> {
        return this._http.get(this.url)
            .map(res => res.json());
    }

    public getEventos(): IEvento[] {
        return [
            { descricao: 'Avaliação Angular', data: '23/10/2018', preco: 0 },
            { descricao: 'Formatura', data: '02/05/2020', preco: 140 },
            { descricao: 'Torneio de Tenis', data: '10/07/2018', preco: 210 },
            { descricao: 'Congresso de TI', data: '16/01/2019', preco: 400 }
        ];
    }
}
```

3. Em **cadastro.component.ts**, realize as alterações:

```
import { Component } from '@angular/core';
import { IEvento } from './../../../interfaces/interface.evento';
import { EventosService } from '../services/eventos.service';

@Component({
  moduleId: module.id,
  templateUrl: 'views/cadastro.component.html'
})

export class CadastroComponent {
  //para um evento selecionado
  public eventoSelecionado: IEvento;

  private novoEvento: IEvento;

  //para a inclusão de um novo evento
  public novo() {
    this.novoEvento = { descricao: '', data: '', preco: 0 };
    this.eventoSelecionado = this.novoEvento;
  }

  public incluir(evento: IEvento) {
    this.listaEventos.push(evento);
    alert('Evento incluído com sucesso');
  }

  public selecionar(item: IEvento): void {
    this.eventoSelecionado = item;
  }

  //lista de eventos
  public listaEventos: IEvento[];
  constructor(eventosService: EventosService) {
    //this.listaEventos = eventosService.getEventos();

    eventosService.getEventosWS()
      .subscribe(res => this.listaEventos = res,
        error => alert(error),
        () => console.log('finalizado'));
  }
}
```

4. Verifique, na execução da aplicação, que agora a lista contempla os eventos do Web service, e não mais da lista fictícia;

5. Nossa tarefa, agora, é enviar dados para o Web service com Angular 4. No arquivo `eventos.service.ts`, realize as alterações:

```
import { Injectable } from '@angular/core';
import { IEvento } from '../interfaces/interface.evento';

import { Http, Response, Headers, RequestOptions } from '@angular/http';
import { Observable } from 'rxjs/Rx';
import 'rxjs/Rx';

@Injectable()
export class EventosService {

    //acesso ao HTTP
    public constructor(private _http: Http) { }

    private url: string = "http://localhost:3200/eventos";

    public getEventosWS(): Observable<IEvento[]> {
        return this._http.get(this.url)
            .map(res => res.json());
    }

    public setEventoWS(evento: IEvento): Observable<IEvento> {
        let header = new Headers({ 'Content-Type': 'application/json' });
        let options = new RequestOptions({ headers: header });

        let json = JSON.stringify(
            {
                descricao: evento.descricao,
                data: evento.data,
                preco: evento.preco
            });
        return this._http.post(this.url, json, options)
            .map(res => res.json());
    }

    public getEventos(): IEvento[] {
        return [
            { descricao: 'Avaliação Angular', data: '23/10/2018', preco: 0 },
            { descricao: 'Formatura', data: '02/05/2020', preco: 140 },
            { descricao: 'Torneio de Tenis', data: '10/07/2018', preco: 210 },
            { descricao: 'Congresso de TI', data: '16/01/2019', preco: 400 }
        ];
    }
}
```

6. Em `cadastro.component.ts`, altere o método `incluir()` para acessar o método do Web service. Verifique que existem outras alterações a serem feitas:

```
import { Component } from '@angular/core';
import { IEvento } from '../interfaces/interface.evento';
import { EventosService } from '../services/eventos.service';

@Component({
  moduleId: module.id,
  templateUrl: 'views/cadastro.component.html'
})

export class CadastroComponent {
  //para um evento selecionado
  public eventoSelecionado: IEvento;
  private novoEvento: IEvento;

  //para a inclusão de um novo evento
  public novo() {
    this.novoEvento = { descricao: '', data:'',preco:0 }
    this.eventoSelecionado = this.novoEvento;
  }

  public incluir(evento: IEvento) {
    //this.listaEventos.push(evento);
    this.eventosService.setEventosWS(evento)
      .subscribe(res => JSON.stringify(res),
        error => alert(error),
        () => this.listar());
    alert('Evento incluído com sucesso');
  }

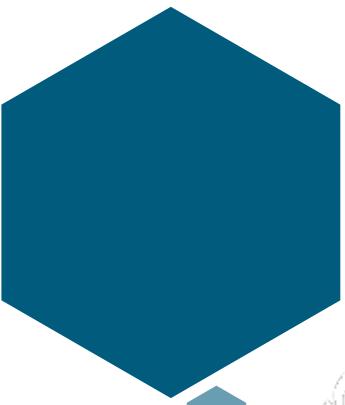
  public selecionar(item: IEvento): void {
    this.eventoSelecionado = item;
  }

  //lista de eventos
  public listaEventos: IEvento[];

  constructor(private eventosService: EventosService) {
    //this.listaEventos = eventosService.getEventos();
    this.listar();
  }

  public listar(): void{
    this.eventosService.getEventosWS()
      .subscribe(res => this.listaEventos = res,
        error => alert(error),
        () => console.log('finalizado'));
  }
}
```

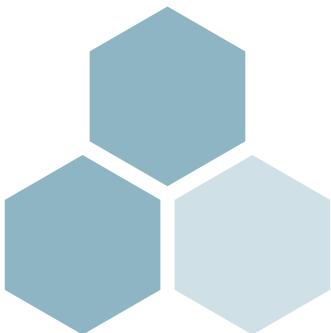
7. Teste a aplicação.



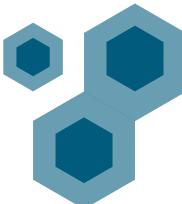
Conceitos do JavaScript

Atividades do Apêndice

Willians
305.890-4588
Cidade Moraes



Editora
IMPACTA



A – Criando a estrutura para uma aplicação prévia com JavaScript

Antes de iniciar com o projeto baseado no Node.js, desenvolveremos um protótipo com JavaScript. O propósito é praticá-lo, visando um melhor entendimento ao longo do treinamento. Vamos, então, seguir os passos adiante:

1. Na pasta **ProjetoEventos**, crie uma pasta chamada **PrototipoJavascript**. Essa pasta pode ser criada tanto pelo VSCode quanto na estrutura de pastas do sistema;
2. Mude o VSCode para essa nova pasta (trabalharemos nela agora);
3. Crie, na pasta **PrototipoJavascript**, a pasta **scripts**. Nessa pasta, colocaremos o código JavaScript do projeto.

B – Definindo a página HTML

1. Na pasta recém-criada, crie um arquivo chamado **index.html**, com um template inicial elaborado pelo VSCode. Deixe com o conteúdo indicado a seguir:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Página com conteúdo Javascript</title>
</head>
<body>

</body>
</html>
```

2. Inclua uma referência à biblioteca **Bootstrap**, para melhorar a aparência da página. Usaremos a versão CDN (versão on-line da biblioteca), disponível em <http://getbootstrap.com/>:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Página com conteúdo Javascript</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/
css/bootstrap.min.css">
</head>
<body>

</body>
</html>
```

3. Crie uma pasta chamada **scripts**. Nessa pasta, crie um arquivo com o nome **funcoes.js** (sem acentuação). Referencie esse arquivo como última instrução no elemento **<body>**:

```
<body>
    <script src="scripts/funcoes.js"></script>
</body>
```

4. No elemento **<body>**, escreva o conteúdo para criar um formulário, cuja funcionalidade é simular o cadastro de um evento. O conteúdo proposto é dado a seguir (observe as classes CSS usadas com base no Bootstrap):

```
<body>
    <div class="container">
        <h1>Cadastro de Eventos</h1>
        <form>
            <div class="form-group">
                <label for="descricao">Descrição do evento:</label>
                <input type="text" class="form-control col-md-6" id="descricao">
            </div>

            <div class="form-group">
                <label for="data">Data do evento:</label>
                <input type="date" class="form-control col-md-3" id="data">
            </div>

            <fieldset class="form-group">
                <legend>Local do evento</legend>
                <div class="form-group">
                    <label for="estado">Estado:</label>
                    <select class="form-control col-md-3" id="estado"></select>
                </div>
                <div class="form-group">
                    <label for="cidade">Cidade:</label>
                    <select class="form-control col-md-3" id="cidade"></select>
                </div>
            </fieldset>

            <div class="form-group">
                <label for="preco">Preço</label>
                <input type="text" class="form-control col-md-6" id="preco">
            </div>

            <button class="btn btn-info" type="button" id="btnEnviar">Enviar</button>
        </form>
    </div>

    <script src="scripts/funcoes.js"></script>
</body>
```

5. Execute a página até este momento. O resultado deve ser algo similar à imagem adiante:

Cadastro de Eventos

Descrição do evento:

Data do evento:

 dd/mm/aaaa

Local do evento

Estado:

Cidade:

Preço

Enviar

C – Definindo o conteúdo do arquivo JavaScript

As funcionalidades desta página serão codificadas em JavaScript. Cada parte deste conjunto de funcionalidades será elaborada em cada passo. Vamos criar um conjunto de funcionalidades para listar os estados assim que a página for carregada. A lista de estados e cidades será definida manualmente.

1. Abra o arquivo **funcoes.js** e escreva o código a seguir:

```
//Array de Estados  
var estados = [  
    { "id": "1", "estado": "SP" },  
    { "id": "2", "estado": "RJ" },  
    { "id": "3", "estado": "MG" },  
    { "id": "4", "estado": "BA" },  
];
```

```
var cidades = [
    { "id": "1", "ideestado": "1", "cidade": "CAMPINAS" },
    { "id": "2", "ideestado": "1", "cidade": "SOROCABA" },
    { "id": "3", "ideestado": "2", "cidade": "NITEROI" },
    { "id": "4", "ideestado": "2", "cidade": "CABO FRIO" },
    { "id": "5", "ideestado": "2", "cidade": "ANGRA" },
    { "id": "6", "ideestado": "3", "cidade": "BELO HORIZONTE" },
    { "id": "7", "ideestado": "3", "cidade": "BETIM" },
    { "id": "8", "ideestado": "3", "cidade": "EXTREMA" },
    { "id": "9", "ideestado": "4", "cidade": "SALVADOR" },
    { "id": "10", "ideestado": "4", "cidade": "PORTO SEGURO" },
];
```

2. No arquivo **index.html**, adicione a biblioteca do jQuery antes da referência ao arquivo criado neste projeto. Podemos obter a biblioteca on-line em <http://code.jquery.com/>;

```
<script src="http://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="scripts/funcoes.js"></script>
```

3. Escreva a função principal jQuery, a que certifica que a execução do seu conteúdo será realizada quando a página estiver totalmente carregada:

```
$(document).ready(function () {
});
```

4. Escreva a função para carregar a lista de estados, quando a página for executada:

```
$(document).ready(function () {

    //lista de estados
    $("#estado").html("<option>Selecione</option>");
    $.each(estados, function (i, estado) {
        $("#estado").append($('<option>', {
            value: estado.id,
            text: estado.estado
        }));
    });
});
```

5. Execute a página. Verifique que a lista de estados foi carregada;

6. Vamos, agora, carregar a lista de cidades, quando um estado for selecionado. Este processo ocorrerá de forma assíncrona, ou seja, o jQuery executará esta rotina em AJAX. Escreva o código, na sequência do código escrito anteriormente:

```
$(document).ready(function () {
    //lista de estados
    $("#estado").html("<option>Selecione</option>");
    $.each(estados, function (i, estado) {
        $("#estado").append($('<option>', {
            value: estado.id,
            text: estado.estado
        }));
    });

    //lista de cidades
    $("#estado").change(function () {
        var idestado = $(this).val();

        var cidadesFiltradas = $.grep(cidades, function (e) {
            return e.idestado == idestado;
        });

        $("#cidade").html("<option>Selecione</option>");
        $.each(cidadesFiltradas, function (i, cidade) {
            $("#cidade").append($('<option>', {
                value: cidade.id,
                text: cidade.cidade
            }));
        });
    });
});
```

7. Atualize a página e selecione um estado. Verifique as cidades deste estado;

8. Vamos, agora, escrever o conjunto de instruções para o evento click do botão. Quando o usuário clicar no botão, as seguintes tarefas serão processadas:

- O conteúdo de cada componente do formulário será lido e armazenado em variáveis;
- Se a descrição do evento não tiver sido fornecida, apresentaremos uma mensagem de erro, informando que o campo é obrigatório;
- Uma vez que tudo tenha sido fornecido, apresentaremos um resumo dessas informações.

Começaremos definindo o evento click do botão. Esta instrução deverá ser escrita dentro da função **ready()**:

```
//evento click do botão
$("#btnEnviar").click(function () {
});
```

9. Complemente esta função para obter os dados do formulário a partir do id de cada componente. Neste exercício, não consideraremos nem o estado nem a cidade:

```
//evento click do botão
$("#btnEnviar").click(function () {
    var descricao = $("#descricao").val();
    var data = $("#data").val();
    var preco = $("#preco").val();
});
```

10. No arquivo **index.html**, acrescente o elemento a seguir logo abaixo do título da página:

```
<h1>Cadastro de Eventos</h1>
<div id="mensagem"></div>
```

11. Complete o arquivo **funcoes.js** com o conteúdo adiante, no evento click do botão:

```
//evento click do botão
$("#btnEnviar").click(function () {
    var descricao = $("#descricao").val();
    var data = $("#data").val();
    var preco = $("#preco").val();

    if (descricao == "") {
        $("#mensagem").html("<div class='alert alert-danger' role='alert'>Campo descrição obrigatório</div>");
    }
});
```

12. Execute e, sem preencher o campo **Descrição**, clique no botão. A mensagem adiante deve aparecer no browser;

Campo descrição obrigatório

Descrição do evento:

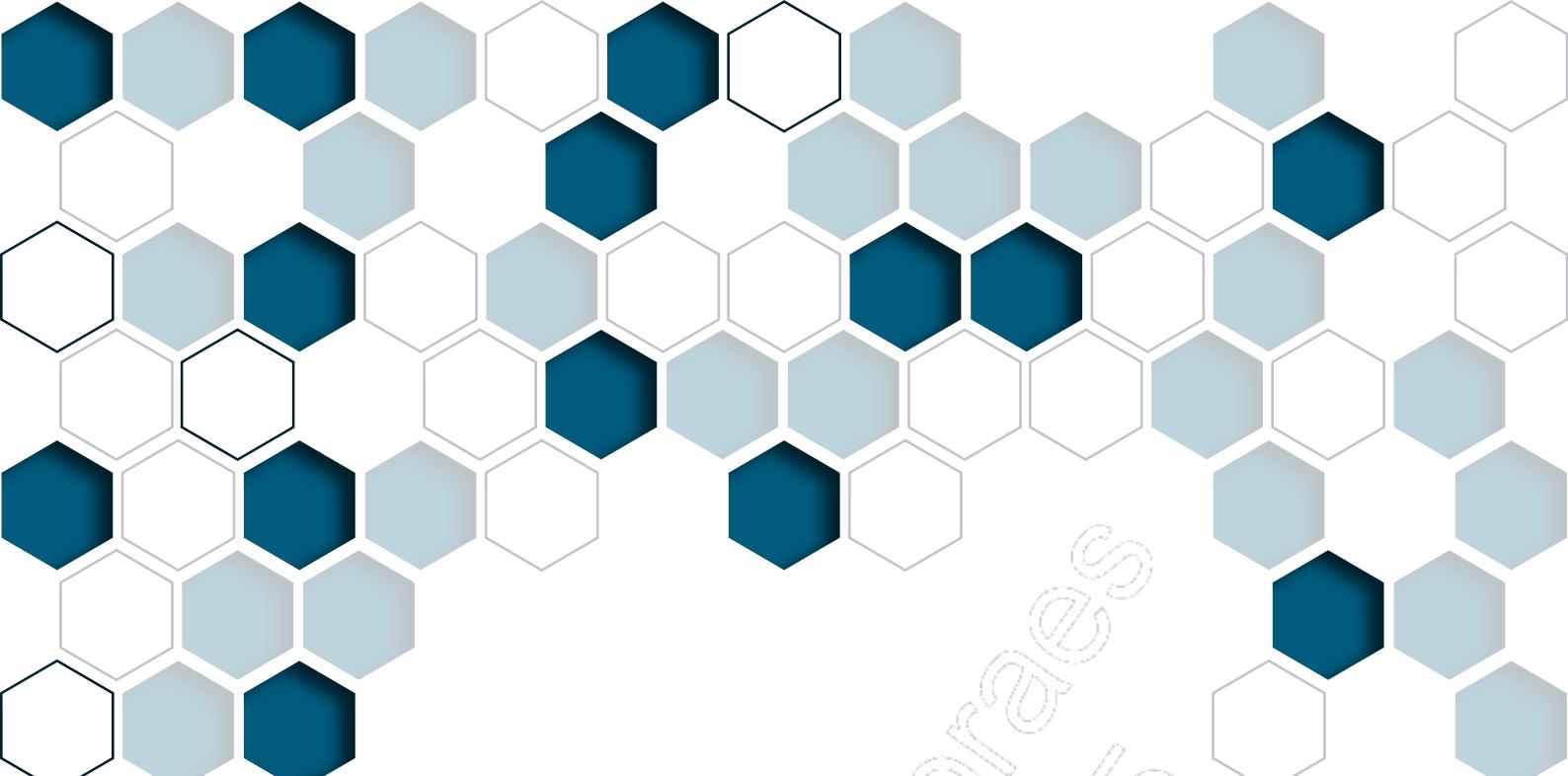
13. Vamos, agora, complementar o código para apresentar a mensagem desejada. Escreva as instruções complementares a seguir:

```
//evento click do botão
$("#btnEnviar").click(function () {
    var descricao = $("#descricao").val();
    var data = $("#data").val();
    var preco = $("#preco").val();

    if (descricao == "") {
        $("#mensagem").html("<div class='alert alert-danger' role='alert'>Campo descrição obrigatório</div>");
    } else {
        var resultado = "<strong>Dados do evento</strong><br/>";
        resultado += "Descrição: " + descricao;
        resultado += "<br/>Data: " + data;
        resultado += "<br/>Preço: " + preco;

        $("#mensagem").html("<div class='alert alert-success' role='alert'>" + resultado + "</div>");
    }
});
```

14. Execute a página e preencha todos os campos.



Mãos à obra!



305.800

Williams Moraes

45.770

305.800

Williams Moraes



1

Criando um projeto com Node.js



Mãos à obra!



A – Criando uma agenda de contatos com Node.js, Express.js e MongoDB

1. Crie a pasta **Lab01_Nodejs** como subpasta de **Laboratorios**;
2. Abra essa pasta no VSCode;
3. No prompt de comandos, acesse a pasta **Lab01_Nodejs**;
4. Instale o Express.js;
5. Usando o Express, defina um projeto chamado **nodeContatos**;
6. Esse projeto deverá ter a seguinte funcionalidade:
 - Uma página inicial apresenta um menu com duas opções: **Incluir Contato** e **Listar Contatos**;
 - Acessando **Incluir Contato**, o usuário deve ser direcionado para um formulário contendo campos de entrada para inserir um novo contato. Os dados do contato devem ser: **CPF**, **NOME**, **EMAIL**, **DATA NASCIMENTO** e **TELEFONE**;
 - Acessando **Listar Contatos**, o usuário deverá ver uma página com a lista de todos os contatos cadastrados;
 - As informações do contato deverão ser armazenadas em um banco de dados MongoDB;
 - Tanto a página de cadastro quanto a listagem deverão possuir links para o menu de opções, e a página de listagem deve ter, também, um link para um novo cadastro.



2

Criando um Web service com Node.js



Mãos à obra!

Willian Moreira
305º



Editora
IMPACTA



A – Criando um Web service com Node.js, Express.js e MongoDB

1. Crie a pasta **Lab02_WebServices** como subpasta de **Laboratorios**;
2. Abra essa pasta no VSCode;
3. No prompt de comandos, acesse a pasta **Lab02_WebServices**;
4. Instale o Express.js;
5. Usando o Express, defina um projeto chamado **apiContatos**;
6. Inclua o componente **cors**, pois este Web service será consumido por outras aplicações;
7. No arquivo **app.js** deste projeto, defina as funções do Web service para:
 - Buscar todos os contatos;
 - Buscar um contato pelo CPF;
 - Incluir um novo contato.
8. Copie o projeto **nodeContatos** do Lab01 para dentro da pasta **Lab02_WebServices**;
9. Refaça as funcionalidades de inclusão e listagem dos contatos para acessar o Web service.



3

Criando um projeto MEAN com AngularJS



Mãos à obra!

Willian Moreira
305º



A – Criando uma aplicação com AngularJS

1. Crie a pasta **Lab03_AngularJS** como subpasta de **Laboratorios**;
2. Abra essa pasta no VSCode;
3. No prompt de comandos, acesse a pasta **Lab03_AngularJS**;
4. Instale o Express.js;
5. Usando o Express, defina um projeto chamado **contatosAngularJS**;
6. Defina um controller, uma rota e uma view. A view deve se chamar **index.ejs**;
7. Nessa view, inclua a referência ao AngularJS. Copie essa referência para a pasta **javascrips** do projeto;
8. Defina um formulário para inclusão de contatos e uma listagem desses contatos, acessando-os por meio do Web service criado no Lab02;
9. Use recursos do Bootstrap para uma melhor visualização.



4

Criando um projeto MEAN com Angular 4

Mãos à obra!



A – Criando uma aplicação com Angular 4

1. Crie a pasta **Lab04_Angular4** como subpasta de **Laboratorios**;
2. Abra essa pasta no VSCode;
3. No prompt de comandos, acesse a pasta **Lab04_Angular4**;
4. Clone o projeto QuickStart com o nome **contatosAngular4**;
5. Usando o npm, instale os módulos do pacote;
6. Crie o componente **CadastroComponent** para permitir a inclusão e a listagem dos contatos;
7. Tanto o cadastro como a listagem devem acessar o Web service do Lab02;
8. Use recursos do Bootstrap para uma melhor visualização.