1 Introdução

O desenvolvimento moderno de aplicações Web é comumente realizado utilizando-se diferentes tecnologias. Muitas vezes, sua combinação dá origem a algo que tem levado o nome de desenvolvimento “Full Stack”. Uma solução desenvolvida segundo esse paradigma possui, em geral, duas aplicações independentes que se comunicam por meio de uma interface bem definida. Uma delas oferece interfaces gráfica para o usuário e geralmente é chamada de aplicação Front End. A outra é responsável por disponibilizar as funcionalidades do sistema e leva o nome de aplicação Back End. Em geral, ela faz uso de um sistema independente que possui implementações eficientes para operações de acesso à memória secundária.

Além disso, seja no Front End ou no Back End, é comum o uso de diferentes *frameworks* que supostamente entregam um nível maior de abstração e promovem a produtividade dos desenvolvedores.

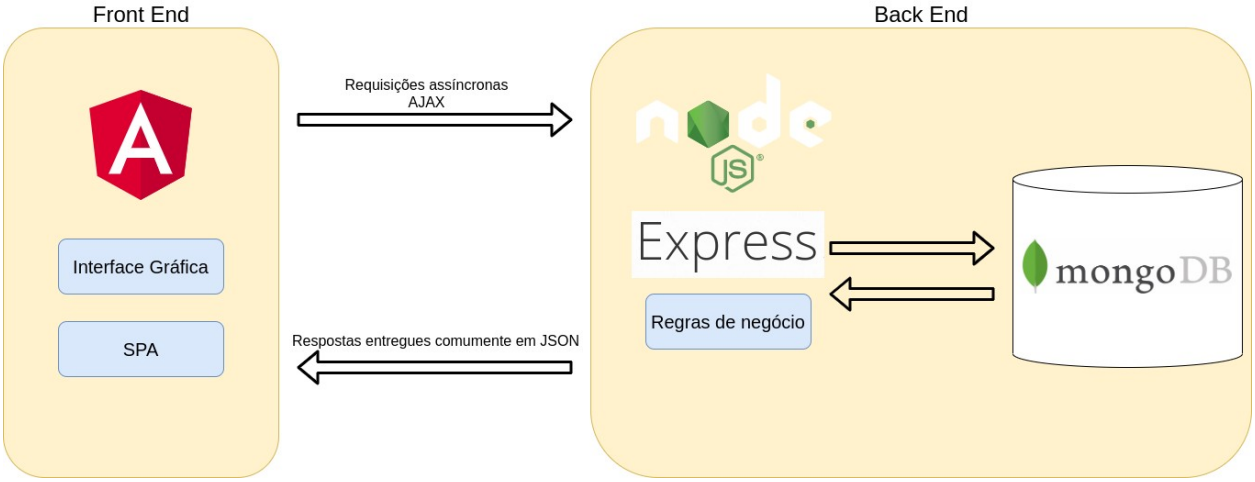
Nos dias atuais, uma das combinações mais utilizadas para esse fim tem a sigla “**MEAN**” associada, a qual deriva de “**MongoDB**”, “**Express**”, “**Angular**” e “**NodeJS**”.

Neste material desenvolveremos uma aplicação que faz uso da “pilha” MEAN.

2 Desenvolvimento

2.1 (Implementação do Back End com NodeJS e Express) O Back End de nossa aplicação será implementado utilizando o NodeJS como framework Express. Veja a Figura 2.1.1 para relembrar.

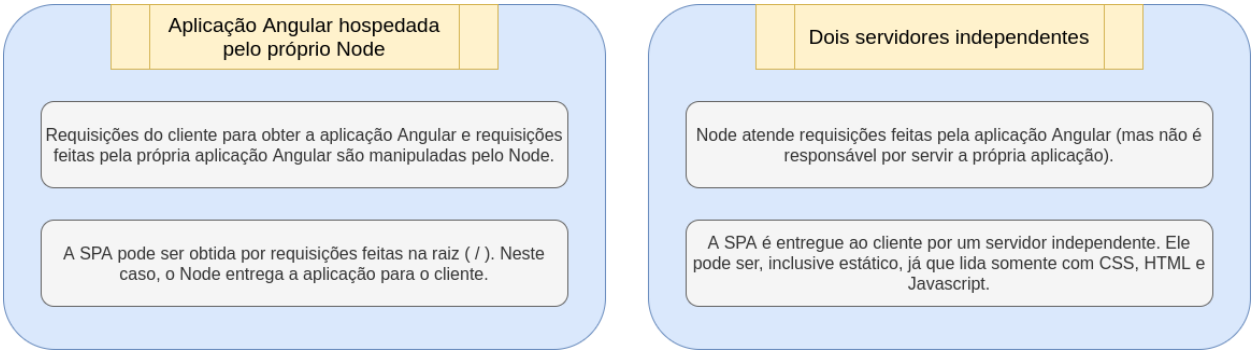
Figura 2.1.1



- Até o momento, utilizamos o comando **ng serve** para testar a nossa aplicação. Embora seja baseado no Node, trata-se de um servidor extremamente simples que certamente não pode ser colocado em produção. Sua finalidade é ser utilizado em tempo de desenvolvimento e testes.

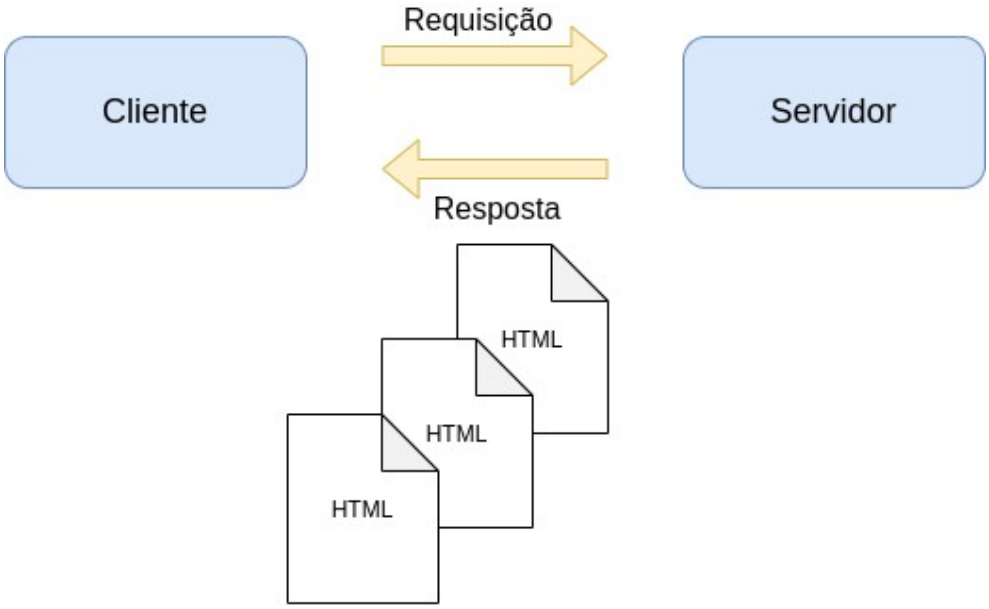
- Há duas formas para fazer com que uma aplicação Angular “converse” com uma aplicação Node. Veja a Figura 2.1.2.

Figura 2.1.2



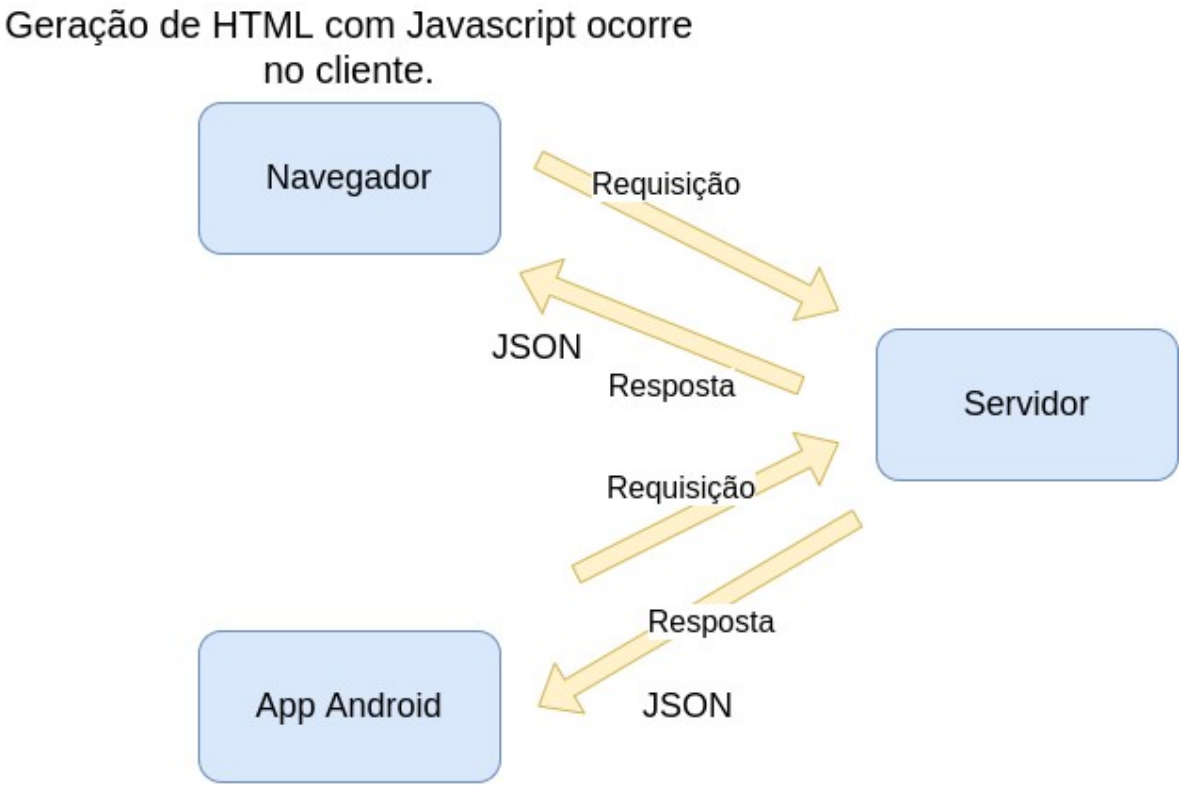
- Uma aplicação Web tradicional pode deixar como responsabilidade do servidor a geração de código HTML. Veja a Figura 2.1.3.

Figura 2.1.3



- Ocorre que, nos dias atuais, diferentes tipos de clientes se conectam a um mesmo servidor com objetivos e necessidades diferentes. Um navegador tradicional pode estar interessado em HTML. Porém, uma aplicação Android pode estar interessada somente em alguns dados armazenados no Back End e ela própria se encarregará de exibi-los sem utilizar HTML. Desta forma, é comum implementar serviços que devolvem somente dados representados de alguma maneira pré determinada (como JSON) e deixar que os clientes decidam o que fazer com eles. Veja a Figura 2.1.4.

Figura 2.1.4



- **REST** (Representational State Transfer) é uma arquitetura (uma coleção de princípios) utilizada para a integração de sistemas. Quando uma solução segue adequadamente os princípios REST, ela é chamada comumente de **Restful**.

Sistemas Restful lidam com a ideia de **recursos**. Um recurso pode ser uma página HTML, uma imagem, um arquivo de áudio. Cada recurso possui um identificador. Em um ambiente Web, é natural que o identificador de um recurso seja uma URL. Veja a Figura 2.1.5.

Figura 2.1.5

http://mercedesbenz.com/carros

http://mercedesbenz.com/carros/1

- Nosso Back End com Node será implementado como uma **API Restful.** Utilizaremos a alternativa de **dois servidores independentes**. Ou seja, teremos o servidor Node para o Back End e iremos manter o servidor de testes para colocar em execução a aplicação Angular.

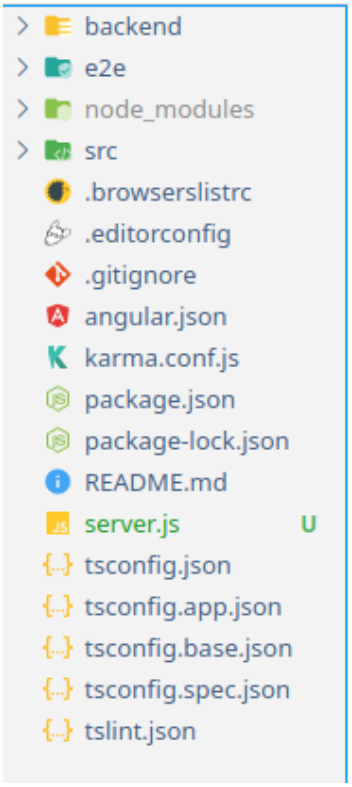
- Comece colocando a aplicação Angular em execução com

ng serve --open

- Vamos criar o Back End em uma pasta no mesmo diretório do projeto apenas por simplicidade. Crie uma pasta chamada **backend** lado a lado com a pasta **src** do seu projeto.

- A seguir, crie um arquivo chamado **server.js** na raiz do projeto (fora de qualquer pasta, inclusive da pasta backend). Faremos isso para ficar mais simples a sua execução neste momento. Veja a estrutura que temos até então na Figura 2.1.6.

Figura 2.1.6



- Abra um novo terminal no VS Code e execute o arquivo **server.js** com

node server.js

Adicione (no arquivo server.js) o código da Listagem 2.1.1 para fazer um pequeno teste.

Listagem 2.1.1

```
console.log("Hello, NodeJS");
```

- O nosso objetivo é colocar um servidor HTTP no ar. O Node possui um pacote próprio para isso chamado http. Uma vez que ele tenha sido trazido para o contexto, podemos usar o método **createServer** para colocar o servidor em execução. Ele recebe dois objetos que representam a requisição e a resposta. Eles contém, por exemplo, as informações dos cabeçalhos definidos pelo protocolo HTTP. Por fim, especificamos a porta em que o servidor ficará esperando as requisições. Veja a Listagem 2.1.2. Note que utilizamos **process.env.PORT** para acessar um valor de porta do ambiente em que o Node está em execução, caso ele exista. Caso contrário, ele usará o valor que especificamos.

Listagem 2.1.2

```
const http = require ('http');
const server = http.createServer((req, res) => {
  res.end("Hello from the Back End");
});
server.listen(process.env.PORT || 3000)
```

- Abra um navegador e faça uma requisição em localhost:3000 para ver o resultado.

- A Listagem 2.1.2 mostra código Node “puro”, sem utilizar nenhum framework. A manipulação das requisições pode se tornar trabalhosa e dar origem a código “boilerplate”. Em geral, utilizamos um framework para abstrair detalhes e ganhar em produtividade. É neste contexto que entra o **Express**. Trata-se de um framework cuja finalidade é fornecer um nível mais alto de abstração para a manipulação de requisições HTTP. Para utilizá-lo, vamos fazer a sua instalação com o npm, veja:

npm install --save express

- Agora passaremos a utilizar o Express. Crie um arquivo chamado **app.js** na pasta **backend**.

- No arquivo **app.js**, o primeiro passo é importar o express. A seguir, construímos um objeto com a função express. O funcionamento, em seguida, ocorre como uma cadeia de chamadas de função. Usamos a função **use** para especificar funções que desejamos que sejam executadas mediantes requisições. Note que cada função recebe três parâmetros:

- A requisição
- A resposta
- Um parâmetro next, que representa uma próxima função a ser executada na cadeia. Cabe à função atual decidir chamá-la ou não.

- Cada modulo tem uma propriedade chamada **exports**. Podemos atribuir a ela valores que desejamos que fiquem disponíveis para outros módulos. Neste caso vamos atribuir o objeto app.

Veja a Listagem 2.1.3.

Listagem 2.1.3

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

app.use((req, res, next) => {
  console.log("Chegou uma requisição");
  next();
});

app.use((req, res, next) => {
  res.send("Hello from the Back end");
});

module.exports = app;
```

- Com isso, podemos importar o objeto **app** no arquivo **server.js**. Uma vez importado, ele deve ser entregue à função createServer, substituindo o valor anterior. Além disso, há um ajuste a ser feito na configuração da porta. Veja a Listagem 2.1.4.

Listagem 2.1.4

```
const http = require ('http');
const app = require ('./backend/app');

const port = process.env.PORT || 3000;
app.set('port', port);
const server = http.createServer(app);
server.listen(port);
```

- No seu navegador, faça uma nova requisição em localhost:3000 e veja a resposta recebida, além da mensagem no console.

- Faça o teste (arquivo **backend/app.js**) da Listagem 2.1.5 (deixar de chamar a função next) e veja o que acontece.

Listagem 2.1.5

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

app.use((req, res, next) => {
  console.log("Chegou uma requisição");
  // next();
});

app.use((req, res, next) => {
  res.send("Hello from the Back end");
});

module.exports = app;
```

- Note que, no momento, a cada alteração realizada, precisamos reiniciar o servidor. Vamos instalar o pacote **nodemon** que, entre outras coisas, oferece a funcionalidade de **live reload.** Ele pode ser instalado com

npm install --save-dev nodemon

- Para utilizá-lo, vamos especificar um novo script no arquivo **package.json**. O nome pode ser qualquer um, o importante é o comando. Veja a Listagem 2.1.6.

Listagem 2.1.6

```
"scripts": {
   "ng": "ng",
   "start": "ng serve",
   "build": "ng build",
   "test": "ng test",
   "lint": "ng lint",
   "e2e": "ng e2e",
   "start:server": "nodemon server.js"
 },
```

- O script pode ser executado da seguinte forma

npm run start:server

2.2 (Busca por clientes) No momento, a aplicação Angular utiliza uma coleção local de clientes armazenada em memória volátil. Evidentemente, esperamos que a lista seja armazenada no Back End, em memória persistente. Vamos implementar um primeiro endpoint que permite a busca por clientes.

- No arquivo **app.js**, podemos eliminar a primeira função entregue para **use**. Veja a Listagem 2.2.1.

Listagem 2.2.1

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

app.use((req, res, next) => {
  res.send("Hello from the Back end");
});

module.exports = app;
```

- A função **use** pode receber outros argumentos, como o “path” que o usuário deve digitar no navegador para disparar determinada função. Faça o ajuste ilustrado na Listagem 2.2.2 e, então, faça uma requisição em localhost:3000/api/clientes no seu navegador.

Listagem 2.2.2

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

app.use('/api/clientes', (req, res, next) => {
  res.send("Hello from the Back end");
});

module.exports = app;
```

- Nossos endpoints irão devolver dados no formato JSON. Um dos benefícios obtidos por conta do uso do Express é a facilidade de manipulação desse formato. A seguir, vamos criar uma coleção fictícia com dados de alguns clientes. Para entregá-lo ao cliente em formato JSON, usamos o método json do objeto res. Veja a Listagem 2.2.3.

Listagem 2.2.3

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

const clientes = [
  {
    id: '1',
    nome: 'Jose',
    fone: '11223344',
    email: 'jose@email.com'
  },
  {
    id: '2',
    nome: 'Jaqueline',
    fone: '22112211',
    email: 'jaqueline@email.com'
  }
]

app.use('/api/clientes', (req, res, next) => {
  res.json(clientes);
});

module.exports = app;
```

- Caso deseje, você também pode devolver um objeto JSON com mais dados, talvez com uma mensagem indicando o resultado da operação. Veja a Listagem 2.2.4.

Listagem 2.2.4

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

const clientes = [
  {
    id: '1',
    nome: 'Jose',
    fone: '11223344',
    email: 'jose@email.com'
  },
  {
    id: '2',
    nome: 'Jaqueline',
    fone: '22112211',
    email: 'jaqueline@email.com'
  }
]

app.use('/api/clientes', (req, res, next) => {
  res.json({
    mensagem: "Tudo OK",
    clientes: clientes
  });
});

module.exports = app;
```

- Também podemos encadear chamadas de métodos para especificar configurações diversas. Por exemplo, o método **status** permite especificar o código de status definido pelo protocolo HTTP. Veja a Listagem 2.2.5.

Listagem 2.2.5

```
app.use('/api/clientes', (req, res, next) => {
  res.status(200).json({
    mensagem: "Tudo OK",
    clientes: clientes
  });
});
```

2.3 (Consumindo o Web Service com a aplicação Angular) Agora podemos ajustar a aplicação Angular para que seu serviço de manipulação de clientes interaja diretamente com o Back End. O Angular possui um módulo que implementa o protocolo HTTP. O primeiro passo é importá-lo no módulo principal (**app.module.ts**) da aplicação. Veja a Listagem 2.3.1.

Listagem 2.3.1

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { HttpClientModule} from '@angular/common/http'

import { MatInputModule } from '@angular/material/input';
import { MatCardModule } from '@angular/material/card';
import { MatButtonModule } from '@angular/material/button';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatExpansionModule } from '@angular/material/expansion';

import { AppComponent } from './app.component';
import { ClienteInserirComponent } from './clientes/cliente-inserir/cliente-inserir.component';
import { CabecalhoComponent } from './cabecalho/cabecalho.component';
import { ClienteListaComponent } from './clientes/cliente-lista/cliente-lista.component';

import { ClienteService } from './clientes/clientes.service';

@NgModule({
  declarations: [
    AppComponent,
    ClienteInserirComponent,
    CabecalhoComponent,
    ClienteListaComponent,
  ],
  imports: [
```

```
    BrowserModule,
    FormsModule,
    BrowserAnimationsModule,
    MatInputModule,
    MatCardModule,
    MatButtonModule,
    MatToolbarModule,
    MatExpansionModule,
    HttpClientModule,
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

- Intuitivamente, cabe ao serviço de manipulação de clientes fazer as requisições HTTP que envolvem clientes. Por isso, vamos injetar um objeto do tipo **HttpClient** em seu construtor, no arquivo **clientes.service.ts**. Veja a Listagem 2.3.2.

Listagem 2.3.2

```
import { Cliente } from './cliente.model';
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';
import { HttpClient } from '@angular/common/http';

@Injectable({ providedIn: 'root' })
export class ClienteService {

  private clientes: Cliente[] = [];
  private listaClientesAtualizada = new Subject<Cliente[]>();

  constructor (private httpClient: HttpClient){

  }

  getClientes(): Cliente[] {
    return [...this.clientes];
  }
  getListaDeClientesAtualizadaObservable() {
    return this.listaClientesAtualizada.asObservable();
  }

  adicionarCliente(nome: string, fone: string, email: string) {
    const cliente: Cliente = {
      nome: nome,
```

```
    fone: fone,
    email: email,
   };
   this.clientes.push(cliente);
   this.listaClientesAtualizada.next([...this.clientes]);
  }
}
```

- Agora podemos utilizar o cliente HTTP no método **getClientes**. Enviaremos uma requisição **GET** (do protocolo HTTP) para começar. No futuro, veremos como especificar, para cada endpoint, quais tipos de requisição são suportadas. Veja a Listagem 2.3.3.

Listagem 2.3.3

```
getClientes(): void {
   this.httpClient.get <{mensagem: string, clientes:
Cliente[]}>('http://localhost:3000/api/clientes').subscribe(
     (dados) => {
       this.clientes = dados.clientes;
       this.listaClientesAtualizada.next([...this.clientes]);
     }
   )
  }
```

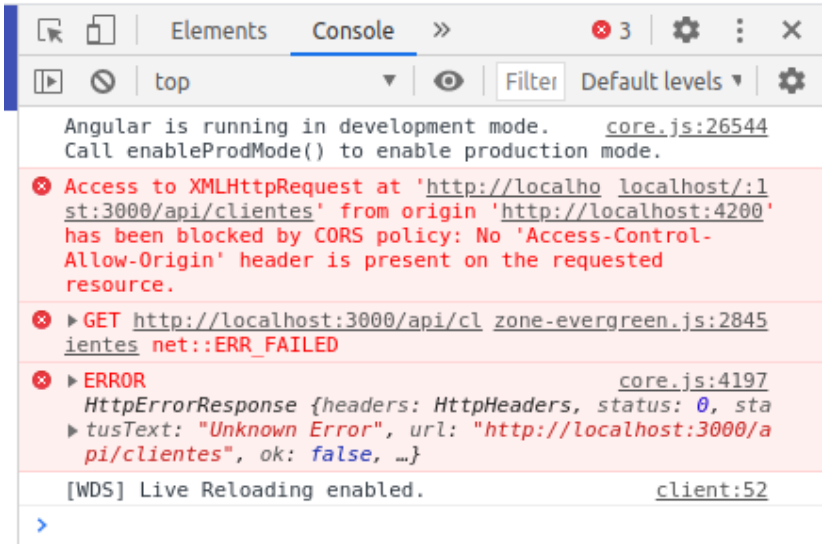
- Note que o componente **ClienteListaComponent** ainda lida com o método **getClientes** como se ele devolvesse uma lista de clientes, o que já não é o caso. Na verdade, basta que ele chame esse método para que a lista seja atualizada. Como ele se registrou como observador da lista de clientes do serviço, assim que ela for atualizada ele será avisado. Veja a Listagem 2.3.4.

Listagem 2.3.4

```
ngOnInit(): void {
   this.clienteService.getClientes();
   this.clientesSubscription = this.clienteService
     .getListaDeClientesAtualizadaObservable()
     .subscribe((clientes: Cliente[]) => {
       this.clientes = clientes;
     });
  }
```

- Perceba, contudo, que a lista de clientes ainda não aparece no navegador. Abra o Chrome Dev Tools (CTRL + SHIFT + I) para ver o erro da Figura 2.3.1.

Figura 2.3.1



- CORS significa **C**ross-**O**rigin **R**esource **S**haring. Note que o nome é um tanto sugestivo. Em nosso ambiente temos, de fato, conteúdo que desejamos compartilhar entre aplicações que estão em servidores diferentes. Quando escrevemos um servidor podemos especificar se requisições vindas de outros servidores são permitidas ou não. Por padrão, elas são bloqueadas. No arquivo **backend/app.js**, vamos especificar uma função que executa antes de a requisição ser atendida. Ela se encarrega de ajustar os cabeçalhos da resposta. Veja a Listagem 2.3.5.

Listagem 2.3.5

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

const clientes = [
  {
    id: '1',
    nome: 'Jose',
    fone: '11223344',
    email: 'jose@email.com'
  },
  {
    id: '2',
    nome: 'Jaqueline',
    fone: '22112211',
    email: 'jaqueline@email.com'
  }
]
app.use ((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', "*");
  res.setHeader('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type,
Accept');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PATCH, DELETE,
OPTIONS');

  next();
});


app.use('/api/clientes', (req, res, next) => {
  res.status(200).json({
    mensagem: "Tudo OK",
    clientes: clientes
  });
});

module.exports = app;
```

2.4 (Endpoint para a inserção de novos clientes) O método mais adequado para a criação de novos recursos (a inserção de um novo cliente, por exemplo) do protocolo HTTP é o post. O objeto **app** permite que especifiquemos o método desejado trocando o método **use** pelo seu nome. Veja a Listagem 2.4.1. Certifique-se de escrever esse trecho abaixo do código que configura os cabeçalhos para tratamento CORS.

Listagem 2.4.1

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

const clientes = [
  {
    id: '1',
    nome: 'Jose',
    fone: '11223344',
    email: 'jose@email.com'
  },
  {
    id: '2',
    nome: 'Jaqueline',
    fone: '22112211',
    email: 'jaqueline@email.com'
  }
]
app.use ((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', "*");
  res.setHeader('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type,
Accept');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PATCH, DELETE,
OPTIONS');

  next();
});

app.post ('/api/clientes', (req, res, next) => {

});


app.use('/api/clientes', (req, res, next) => {
  res.status(200).json({
    mensagem: "Tudo OK",
    clientes: clientes
  });
});
```

```
module.exports = app;
```

- Vamos utilizar um novo pacote do Node para lidar com os dados da requisição. Precisamos de uma maneira eficiente de extrair o objeto JSON que a aplicação Angular enviará. O pacote **body-parser** é uma excelente opção. Para instalá-lo, use o comando

npm install --save body-parser

- No arquivo **app.js**, importe o pacote recém-instalado como mostra a Listagem 2.4.2. Além disso, chame a função use entregando a ela o resultado da chamada à função json do bodyParser. Ela devolve uma função apropriada para a manipulação de dados em formato JSON.

Listagem 2.4.2

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

app.use (bodyParser.json());
...
```

- Uma vez aplicada essa função, podemos fazer uso de um campo chamado **body** que agora existe no objeto que representa a requisição. Veja a Listagem 2.4.3.

Listagem 2.4.3

```
app.post ('/api/clientes', (req, res, next) => {
  const cliente = req.body;
  console.log(cliente);
  res.status(201).json({mensagem: 'Cliente inserido'})
});
```

2.5 (Inserindo clientes a partir da aplicação Angular) Agora ajustaremos o serviço de manipulação de clientes na aplicação Angular para que ele faça a inserção de clientes no Back End. Abra o arquivo **clientes.service.ts** e encontre o método adicionarCliente. A requisição HTTP será feita pelo mesmo objeto que utilizamos para obter a lista de clientes. Veja a Listagem 2.5.1.

Listagem 2.5.1

```
adicionarCliente(nome: string, fone: string, email: string) {
    const cliente: Cliente = {
      nome: nome,
      fone: fone,
      email: email,
    };
    this.httpClient.post<{mensagem: string}> ('http://localhost:3000/api/clientes',
cliente).subscribe(
      (dados) => {
        console.log(dados.mensagem);
        this.clientes.push(cliente);
        this.listaClientesAtualizada.next([...this.clientes]);
      }
    )

  }
```

- Faça uma nova requisição a partir da aplicação Angular para inserir um novo cliente e veja o resultado.

Referências

Angular. 2020. Disponível em <https://angular.io>. Acesso em agosto de 2020.

Angular Material UI component library. 2020. Disponível em <https://material.angular.io>. Acesso em agosto de 2020

Express - Node.js web application framework. 2020. Disponível em <https://expressjs.com>. Acesso em agosto de 2020.

Node.js. 2020. Disponível em <https://nodejs.org/en/>. Acesso em agosto de 2020.

The most popular database for modern apps | MongoDB. 2020. Disponível em <https://www.mongodb.com>. Acesso em agosto de 2020.