

**EAD**  
**UNISANTA**

## **PROGRAMAÇÃO PARA INTERNET**

Me. Fernando José Cesílio Branquinho

**GUIA DA  
DISCIPLINA**

## 1. DESENVOLVIMENTO WEB

### 1.1. Aplicações WEB

Segue um resumo (bem resumido) da evolução do conteúdo WEB ao longo do tempo:

- Páginas HTML Estáticas
- Java Script
- Pacotes reutilizáveis (Libraries)
- Frameworks Frontend e Backend

No início o conteúdo era disponibilizado basicamente através de Páginas Estáticas. De forma grosseira podemos comparar isso ao conteúdo de um livro. Alguém escrevia o conteúdo da página e disponibilizava para ser acessado a partir de um Browser de Internet (ex: Google Chrome, Microsoft Edge, Firefox etc). Para haver uma alteração na informação, alguém precisaria alterar o conteúdo da página.

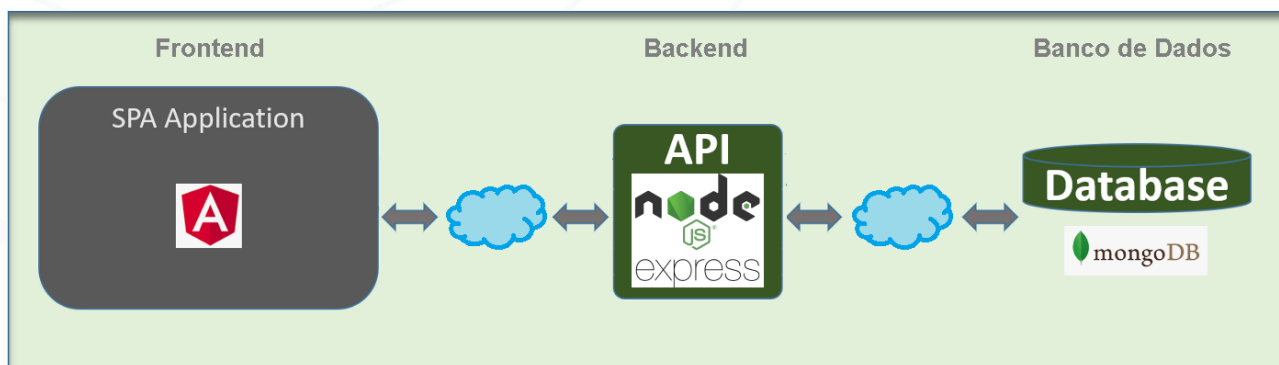
Com o passar do tempo, os browsers passaram a executar código escrito na linguagem **Java Script**. Isso permitiu que o conteúdo pudesse se comportar de modo mais dinâmico, pois era possível de ser alterado pelo programa que estava rodando dentro do browser.

Com o sucesso do **Java Script**, foram criadas bibliotecas (**Libraries**) que agrupavam código interessante para resolver vários problemas comuns.

Com a crescente disponibilidade da Internet e com o significativo aumento de possibilidade de aplicações, os sistemas se tornaram cada vez mais complexos e trabalhosos. Surgem então os **Frameworks** para facilitar a criação de aplicações através de formas padronizadas de desenvolvimento. Atualmente existem muitos Frameworks disponíveis, cada um com suas características específicas.

### 1.2. Exemplo de arquitetura de uma aplicação WEB

Atualmente existem muitas formas para a criação de aplicações WEB. Segue abaixo um diagrama simplificado da arquitetura escolhida para utilizarmos em nosso estudo:



Como podemos ver acima, nossa aplicação será composta por três partes principais:

- Frontend
- Backend
- Banco de Dados

Na esquerda temos o **Frontend**, que é a parte da aplicação acessada pelo usuário. Esta parte roda no browser do dispositivo do usuário (computador, smartphone, tablet etc). Para desenvolver esta parte, utilizaremos um **Framework** chamado de **Angular**. É comum também chamarmos a parte do Frontend de **Client-Side**.

Na parte do meio temos o **Backend**, que roda em um SERVIDOR na nuvem. Esta parte é que vai disponibilizar os serviços funcionando como uma API (Application Programming Interface). Para desenvolver esta parte, usaremos um **Framework** chamado de **Express**. É muito comum também chamarmos a parte do Backend de **Server-Side**.

Na direita temos o BANCO DE DADOS, que é a parte responsável por armazenar os dados manipulados pela aplicação. Em nossa aplicação, utilizaremos um Banco de Dados chamado de **MongoDB**.

### 1.3. Preparação do Ambiente

Para desenvolver nossa aplicação, vamos precisar instalar as seguintes ferramentas em um computador com o Sistema Operacional Windows:

- Visual Studio Code
- Git
- Node.js
- Angular CLI
- Postman

## 1.4. Instalação do Visual Studio Code

Este será o editor de texto utilizado para escrevermos os nossos programas. Considerando que todos os exemplos serão realizados com ele, é muito recomendado que você instale em seu computador.

Link para baixar o arquivo de instalação:

<https://code.visualstudio.com/download>

### 1.4.1. Testando o Visual Studio Code

Abrir o Prompt de Comando do Windows e executar o seguinte comando:

**code .**

O VsCode será executado

## 1.5. Instalação do git

O git é um software muito eficiente para gerenciar alterações em arquivos. Por mais que não façamos uso do mesmo em nossas aulas, é recomendado sua instalação para evitar mensagens de warning provenientes do Angular CLI.

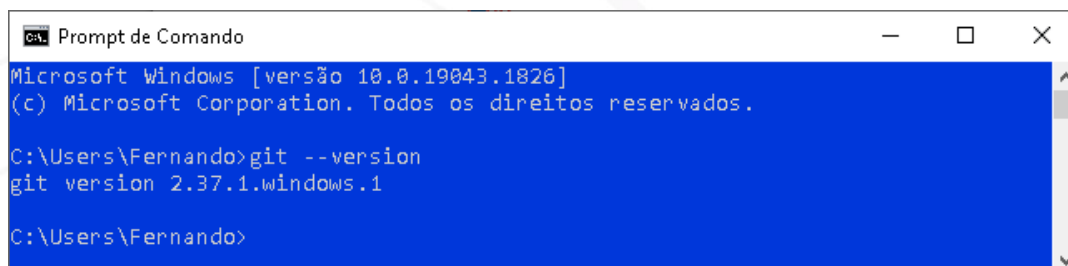
Link para baixar o arquivo de instalação:

<https://git-scm.com/downloads>

### 1.5.1. Testando o git

Abrir o Prompt de Comando do Windows e executar o seguinte comando:

**git --version**



```
Microsoft Windows [versão 10.0.19043.1826]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Fernando>git --version
git version 2.37.1.windows.1

C:\Users\Fernando>
```

## 1.6. Instalação do Node.js

O Node é um ambiente para execução de JavaScript. Ele será utilizado em todas as aulas, seja para oferecer o ambiente de desenvolvimento do App Angular como também para disponibilizar o servidor feito Express no Backend.

Link para baixar o arquivo de instalação:

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

### 1.6.1. Testando o Node

Abrir o Prompt de Comando do Windows e executar o seguinte comando:

**node --version**

O programa apresentará a versão instalada.

## 1.7. Instalação do Angular CLI

O Angular CLI oferece o ambiente de desenvolvimento do App Angular. Para ser instalado ele faz uso do gerenciador de pacotes (npm) oferecido pelo Node.js.

Abrir o Prompt de Comando do Windows e executar o seguinte comando:

**npm install -g @angular/cli**

### 1.7.1. Testando o Angular CLI

Abrir o Prompt de Comando do Windows e executar o seguinte comando:

**ng version**

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [versão 10.0.19043.1826]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Fernando>ng version

Angular CLI
14.0.6

Angular CLI: 14.0.6
Node: 16.16.0
Package Manager: npm 8.14.0
OS: win32 x64

Angular:
...

Package                                  Version
-----
@angular-devkit/architect                0.1400.6 (cli-only)
@angular-devkit/core                     14.0.6 (cli-only)
@angular-devkit/schematics               14.0.6 (cli-only)
@schematics/angular                     14.0.6 (cli-only)

C:\Users\Fernando>
```

## 1.8. Instalação do Postman

Vamos utilizar o Postman para enviar requisições HTTP de teste de nosso servidor no Backend.

Link para baixar o arquivo de instalação:

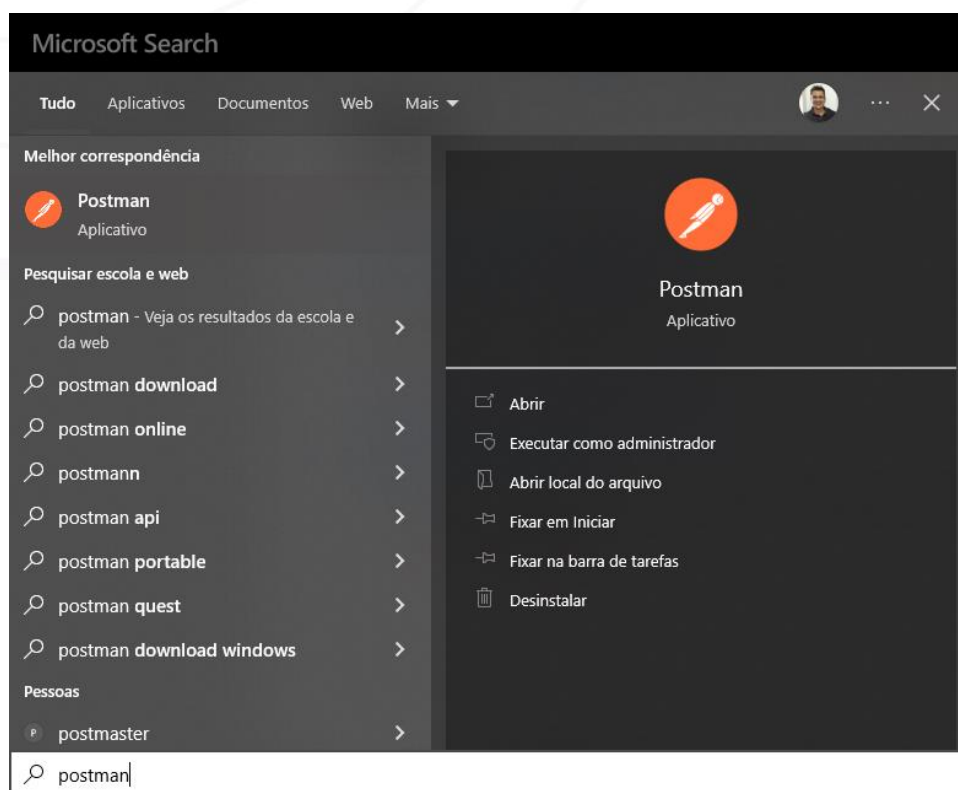
<https://www.postman.com/downloads/>

### 1.8.1. Testando o Postman

Pressione o botão iniciar do Windows e digite:

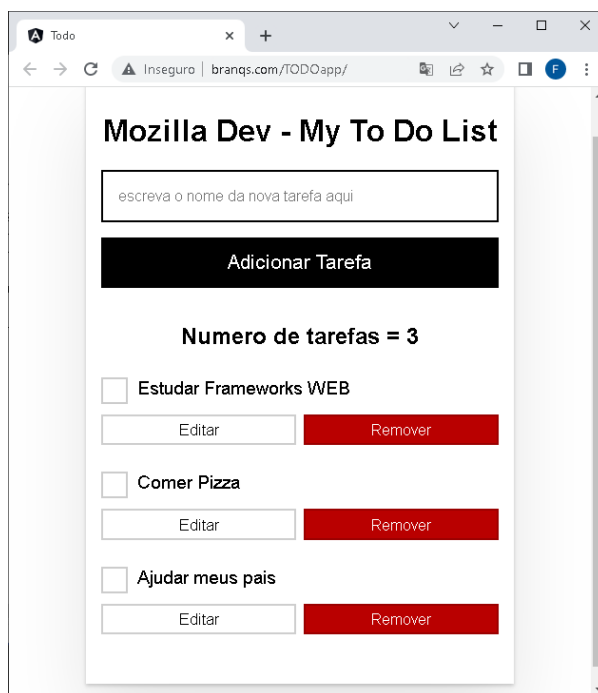
**postman**

Execute o programa a partir do ícone apresentado. Ex:



## 2. INICIANDO O FRONTEND

Vamos iniciar o desenvolvimento do nosso primeiro aplicativo Frontend, que será destinado para gerenciar uma lista de **Tarefas a Fazer**, permitindo Gravar, Consultar, Alterar e Excluir (CRUD) tarefas de uma lista. Segue o resultado que pretendemos atingir até o final desta aula e também um link que permite experimentar a utilização do App:



<http://branqs.com/TODOexemplo/>

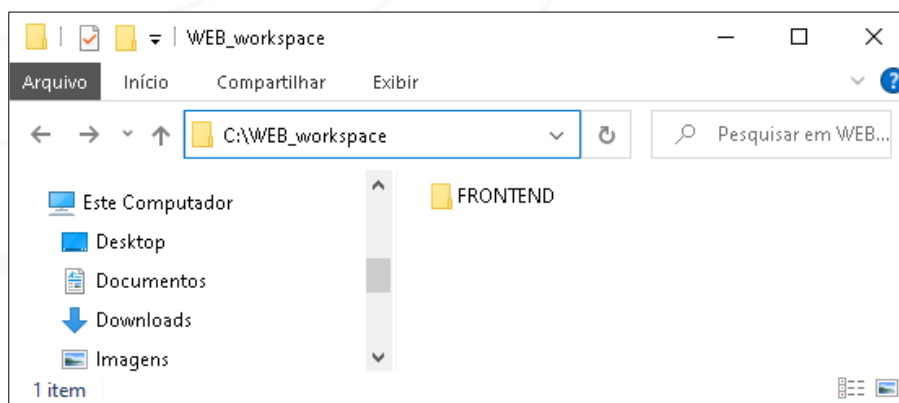
Nosso App será desenvolvido com ajuda do **Framework Angular**. A principal ferramenta de desenvolvimento do Angular é chamada de **Angular CLI** (já instalada em nossa aula anterior). Ela permite realizar várias atividades como: Criar projetos; Implementar código estrutural; Efetuar testes; Efetuar Deploy e etc.

### 2.1. Criando o workspace

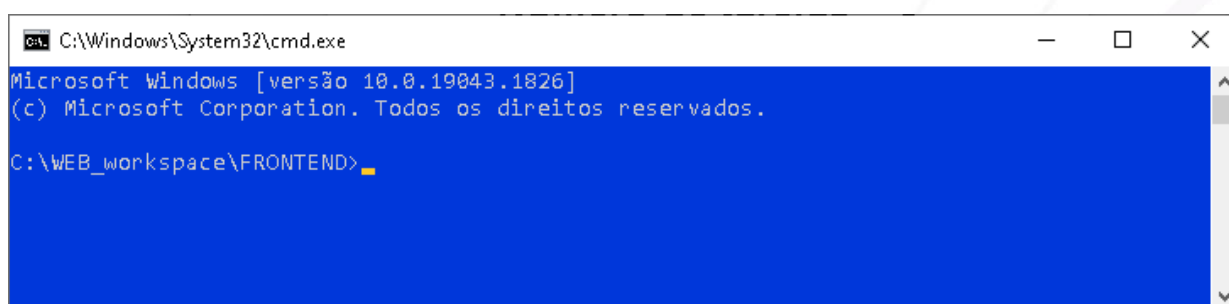
Para facilitar o acompanhamento das aulas, crie uma pasta chamada "WEB\_workspace" no raiz da unidade C: de seu computador.

Dentro dessa pasta crie uma nova pasta chamada de FRONTEND. Ex:





Acesse o prompt de comando do Windows dentro da pasta FRONTEND. É aqui onde você irá executar os primeiros comandos do Angular CLI para criação e teste de seu App:

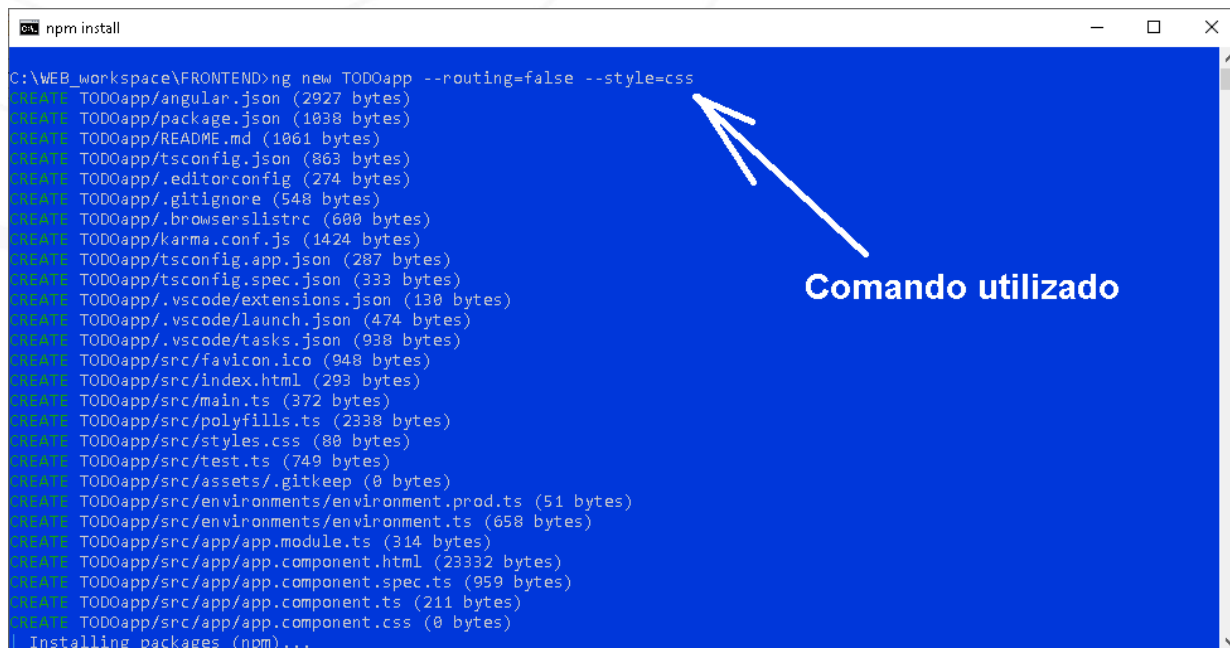


## 2.2. Criando o projeto TODOapp

Abra o prompt de comando dentro da pasta FRONTEND, e execute o seguinte comando para criar um App Angular com o nome "TODOapp":

**ng new TODOapp --routing=false --style=css**

Ao ser executado o comando acima, a ferramenta inicia a criação da estrutura de pastas do seu novo projeto. Ex:

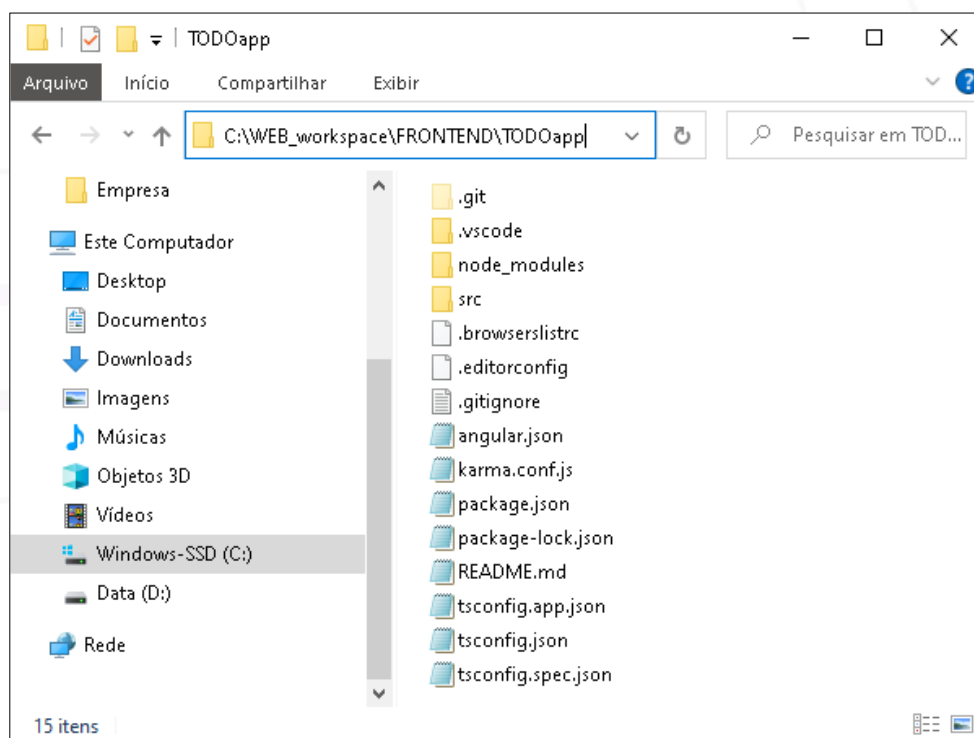


```
npm install
C:\WEB_workspace\FRONTEND>ng new TODOapp --routing=false --style=css
CREATE TODOapp/angular.json (2927 bytes)
CREATE TODOapp/package.json (1038 bytes)
CREATE TODOapp/README.md (1061 bytes)
CREATE TODOapp/tsconfig.json (863 bytes)
CREATE TODOapp/.editorconfig (274 bytes)
CREATE TODOapp/.gitignore (548 bytes)
CREATE TODOapp/.browserslistrc (600 bytes)
CREATE TODOapp/karma.conf.js (1424 bytes)
CREATE TODOapp/tsconfig.app.json (287 bytes)
CREATE TODOapp/tsconfig.spec.json (333 bytes)
CREATE TODOapp/.vscode/extensions.json (130 bytes)
CREATE TODOapp/.vscode/launch.json (474 bytes)
CREATE TODOapp/.vscode/tasks.json (938 bytes)
CREATE TODOapp/src/favicon.ico (948 bytes)
CREATE TODOapp/src/index.html (293 bytes)
CREATE TODOapp/src/main.ts (372 bytes)
CREATE TODOapp/src/polyfills.ts (2338 bytes)
CREATE TODOapp/src/styles.css (80 bytes)
CREATE TODOapp/src/test.ts (749 bytes)
CREATE TODOapp/src/assets/.gitkeep (0 bytes)
CREATE TODOapp/src/environments/environment.prod.ts (51 bytes)
CREATE TODOapp/src/environments/environment.ts (658 bytes)
CREATE TODOapp/src/app/app.module.ts (314 bytes)
CREATE TODOapp/src/app/app.component.html (23332 bytes)
CREATE TODOapp/src/app/app.component.spec.ts (959 bytes)
CREATE TODOapp/src/app/app.component.ts (211 bytes)
CREATE TODOapp/src/app/app.component.css (0 bytes)
! Installing packages (npm)...
```

Comando utilizado

Não se preocupe se aparecerem vários warnings sobre "...LF will be replaced by CRLF the next time..." e também mensagens sobre a falta de identificação do usuário para o git".

O importante é perceber que todos os pacotes foram instalados e que foi criada a seguinte estrutura de pastas para o seu projeto:



A partir de agora, já podemos iniciar o ambiente de desenvolvimento do Angular, executando um servidor que permitirá apresentar o App em execução.

É importante lembrar que este servidor serve somente para efeito de testes de forma local em seu computador durante o desenvolvimento.

Ao final do desenvolvimento, o código final de seu App deverá ser transferido para o ambiente de um servidor HTTP, permitindo que possa ser carregado a partir de qualquer browser.

### 2.3. Executando o servidor

Para que possamos testar nosso App localmente, o Angular CLI oferece um servidor HTTP que disponibiliza o conteúdo de nosso App a partir da porta 4200. Entre na pasta criada para o projeto:

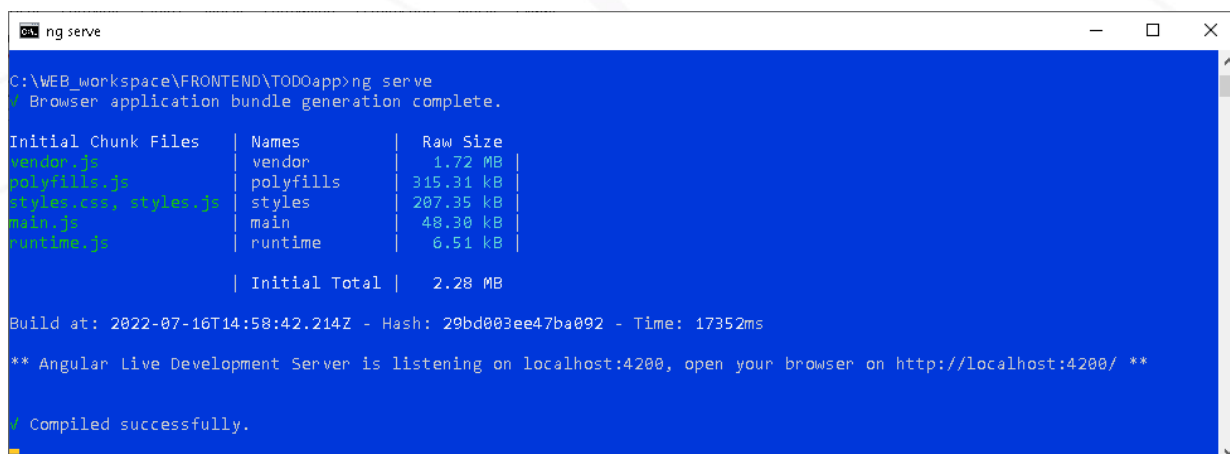
```
cd TODOapp
```

Execute o seguinte comando para iniciar o servidor local:

```
ng serve
```

O Angular então irá gerar um pacote da aplicação para poder ser executado no browser.

Ao final da compilação, a ferramenta vai informar que o App está disponível para uso a partir da porta 4200 de localhost:



```
ng serve
C:\WEB_workspace\FRONTEND\TODOapp>ng serve
✓ Browser application bundle generation complete.

Initial Chunk Files | Names          | Raw Size
vendor.js           | vendor         | 1.72 MB
polyfills.js        | polyfills      | 315.31 kB
styles.css, styles.js | styles         | 207.35 kB
main.js             | main           | 48.30 kB
runtime.js          | runtime        | 6.51 kB
                    | Initial Total  | 2.28 MB

Build at: 2022-07-16T14:58:42.214Z - Hash: 29bd003ee47ba092 - Time: 17352ms

** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **

✓ Compiled successfully.
```

A janela acima mostra o resultado apresentado pelo comando "ng serve".

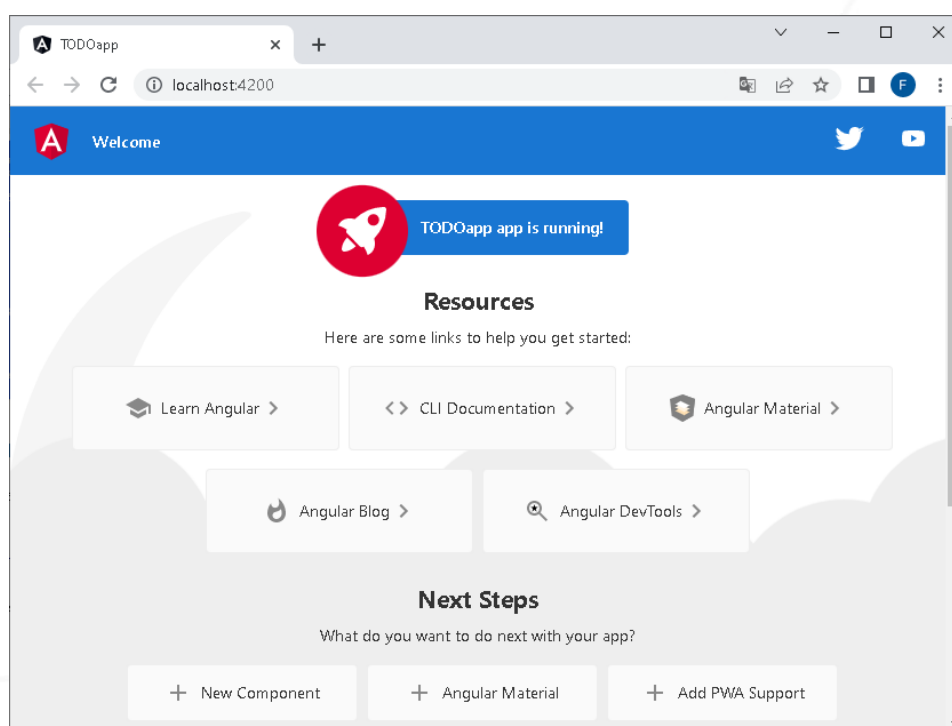
Deixe essa janela aberta enquanto estiver testando o seu App a partir do browser.

## 2.4. Testando o TODOapp localmente a partir do browser

Agora basta utilizar um browser de sua preferência para abrir o seguinte endereço:

**http://localhost:4200/**

O browser apresentará um App Angular **default** em execução. Ex:



O conteúdo acima corresponde a um esqueleto padrão de exemplo oferecido pelo Angular CLI durante a criação de um novo projeto.

Nosso objetivo agora é aproveitar a estrutura criada, e realizar as alterações necessárias para a construção de um App de acordo com as nossas necessidades.

## 2.5. Alterando a View do AppComponent

Agora vamos experimentar modificar o conteúdo apresentado pelo App. Abra um novo prompt de comando dentro da pasta TODOapp. Execute o VsCode utilizando o seguinte comando:

**code .**

Substitua todo o conteúdo do arquivo **app.component.html** pelas linhas abaixo:

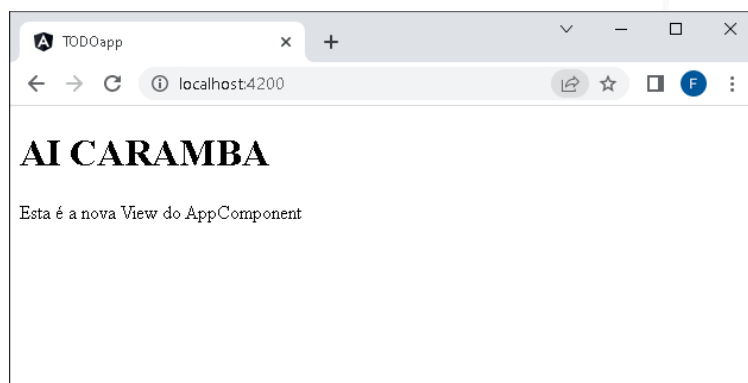
```
<h1>AI CARAMBA</h1>  
Esta é a nova View do AppComponent
```

## 2.6. Testando no Browser

Abra novamente o browser utilizando o mesmo endereço:

**http://localhost:4200/**

Resultado esperado:



### 3. MOSTRANDO UMA LISTA DE TAREFAS

Agora vamos preparar nosso App para mostrar uma lista de "Tarefas a Fazer". Cada tarefa deverá possuir duas informações importantes:

- Uma Descrição  
Tipo: string (Ex: "Jogar Futebol com os amigos")
- Um Status  
Tipo: boolean (Ex: "true" = "Realizada" e "false" = "Não Realizada")

Para começar, vamos então criar uma "classe" que determinará que uma "Tarefa" deva possuir as informações definidas acima. Para isso, vamos criar um arquivo chamado **"tarefa.ts"** dentro da pasta **"TODOapp/src/app"** com o seguinte conteúdo:

```
export class Tarefa {  
  descricao: string;  
  statusRealizada: boolean;  
  
  constructor(_descricao: string, _statusRealizada: boolean) {  
    this.descricao = _descricao;  
    this.statusRealizada = _statusRealizada;  
  }  
}
```

Perceba que o código acima define uma classe chamada de "Tarefa". Essa classe possui dois atributos: Um chamado de "descricao" do tipo "string" e outro chamado de "statusRealizada" do tipo boolean.

Adicionalmente a classe define um "método construtor" que recebe uma "descrição" e um "status" como parâmetros. Dentro do corpo desse método, os parâmetros são armazenados nos atributos da classe.

As definições acima permitirão que nosso App manipule facilmente instâncias (objetos) do tipo "Tarefa".

A seguir, vamos inserir as linhas abaixo destacadas em verde, no conteúdo do arquivo app.component.ts:

```
import { Component } from '@angular/core';
import { Tarefa } from "../tarefa";

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'TODOapp';

  arrayDeTarefas: Tarefa[] = [];

  constructor() {
    this.READ_tarefas();
  }

  READ_tarefas() {
    this.arrayDeTarefas = [
      new Tarefa("Estudar Frameworks WEB", false),
      new Tarefa("Comer Pizza", false),
      new Tarefa("Ajudar meus pais", false)
    ];
  }
}
```

O código adicionado (em verde), realiza as seguintes atividades:

- A linha do "import" permite que o AppComponent utilize as definições realizadas na classe "Tarefa".
- A variável "arrayDeTarefas" foi definida como um array vazio para armazenar objetos do tipo "Tarefa".
- O método READ\_tarefas armazena 3 tarefas dentro do array. O nome deste método pode parecer estranho neste momento, porém ele simula o processo de "Leitura" de tarefas armazenadas em algum tipo de repositório persistente.
- O construtor executa o método READ\_tarefas durante a inicialização do programa, garantido que o array seja populado com alguns objetos para a realização de testes.

Agora vamos permitir que as tarefas sejam apresentadas no browser. Para isso vamos definir o seguinte conteúdo para o arquivo **app.component.html**:

```
<div class="main">
  <h1>Mozilla Dev - My To Do List</h1>
  <ul>
    <li *ngFor="let tarefaDoLoop of arrayDeTarefas">
      {{tarefaDoLoop.descricao}}
    </li>
  </ul>
</div>
```

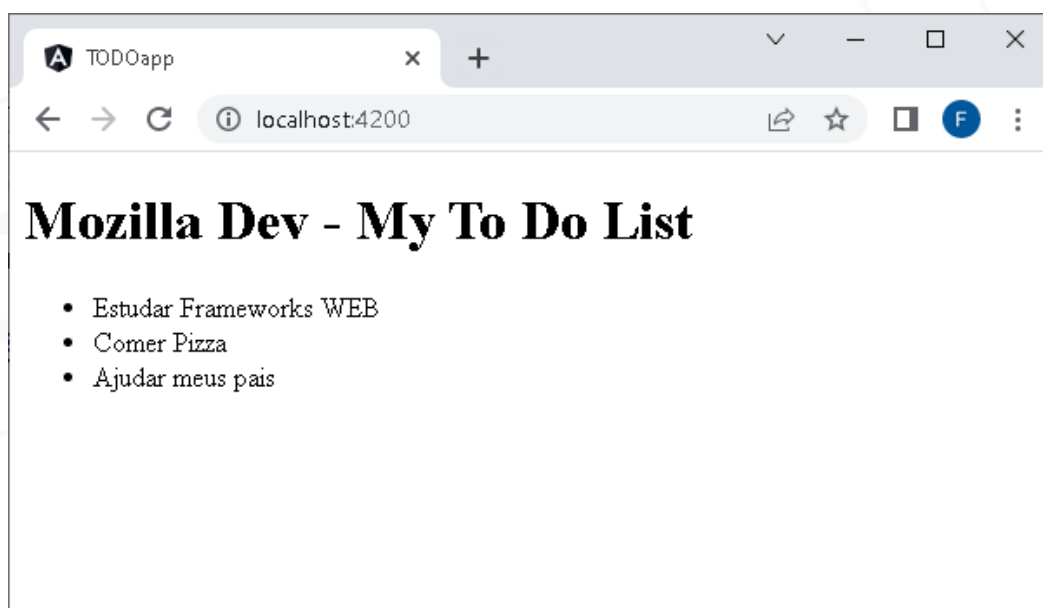
Perceba que o código acima faz uso de uma diretiva Angular de repetição chamada de "ngFor" juntamente com o código HTML. Esta diretiva irá criar um elemento HTML <li> para objeto do tipo "Tarefa" armazenado em "arrayDeTarefas". Adicionalmente, essa linha faz uso também de uma estrutura de "Data Binding" do Angular chamada de "interpolação", que faz uso das chaves duplas para apresentar na view um determinado valor proveniente da lógica do componente (neste caso, o atributo "descricao" do objeto representado pela variável "tarefaDoLoop" da diretiva "ngFor"). Esses detalhes serão explicados de forma mais apropriada na próxima aula.

### 3.1. Testando no Browser

Abra novamente o browser utilizando o mesmo endereço:

**http://localhost:4200/**

Resultado esperado:





### 3.2. Adicionando novas tarefas

Para permitir ao usuário inserir novas tarefas, vamos criar um novo método na lógica do AppComponent que permita armazenar uma nova tarefa no array. Para isso, adicione as linhas abaixo destacadas em verde, no conteúdo do arquivo **app.component.ts**:

```
import { Component } from '@angular/core';
import { Tarefa } from "../tarefa";

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'TODOapp';

  arrayDeTarefas: Tarefa[] = [];

  constructor() {
    this.READ_tarefas();
  }

  CREATE_tarefa(descricaoNovaTarefa: string) {
    var novaTarefa = new Tarefa(descricaoNovaTarefa, false);
    this.arrayDeTarefas.unshift(novaTarefa);
  }

  READ_tarefas() {
    this.arrayDeTarefas = [
      new Tarefa("Estudar Frameworks WEB", false),
      new Tarefa("Comer Pizza", false),
      new Tarefa("Ajudar meus pais", false)
    ];
  }
}
```

Perceba que o método adicionado recebe como parâmetro somente a descrição da tarefa. Na sequência ele cria um novo objeto do tipo "Tarefa", utilizando a descrição recebida como parâmetro e define automaticamente o status "false". A última linha do método insere o objeto criado no arrayDeTarefas.

### 3.3. Adicionando o campo da descrição e o botão na View

Adicionalmente, vamos precisar inserir dois novos elementos na view de AppComponent:

Um campo para a digitação da descrição da nova tarefa

Um botão para confirmar a adição da tarefa

Para isso, insira as linhas abaixo destacadas em verde, no conteúdo do arquivo **app.component.html**:

```
<div class="main">
  <h1>Mozilla Dev - My To Do List</h1>
  <input class="lg-text-input" #campoNovoItem />
  <button class="btn-primary"
(click)="CREATE_tarefa(campoNovoItem.value)">
    Adicionar Tarefa</button>
  <ul>
    <li *ngFor="let tarefaDoLoop of arrayDeTarefas">
      {{tarefaDoLoop.descricao}}
    </li>
  </ul>
</div>
```

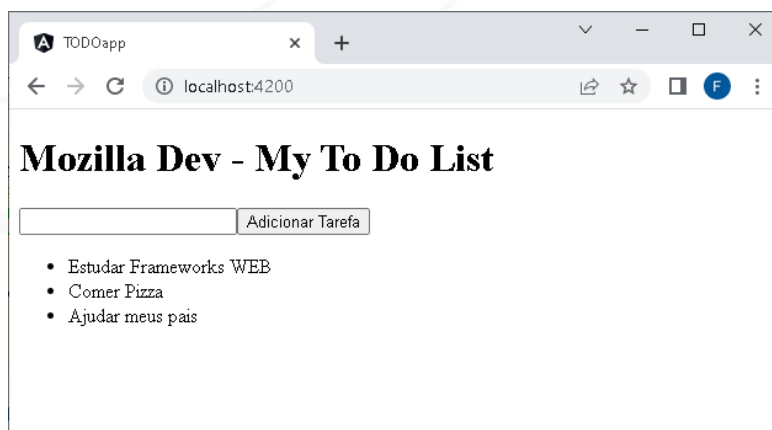
A primeira linha adicionada representa um campo de input. As próximas linhas adicionadas representam o botão. Perceba associado ao evento de "click" do botão, foi inserida a chamada ao método **CREATE\_tarefa** definido anteriormente, passando o texto digitado no campo de input.

### 3.4. Testando no Browser

Abra novamente o browser utilizando o mesmo endereço:

**http://localhost:4200/**

Resultado esperado:



Experimente inserir novas tarefas na lista a partir do browser.

### 3.5. Adicionando Estilos ao app

O Framework Angular permite originalmente a definição de dois escopos de estilo:

- Estilo da aplicação
- Estilo do componente

### 3.6. Estilo da aplicação

O estilos globais do App são definidos a partir do arquivo "**src/styles.css**"

Experimente definir os estilos de escopo geral do App, alterando o conteúdo do arquivo "**src/styles.css**" com as seguintes definições:

```
body {  
  font-family: Helvetica, Arial, sans-serif;  
}  
  
.btn-wrapper {  
  /* flexbox */  
  display: flex;  
  flex-wrap: nowrap;  
  justify-content: space-between;  
}  
  
.btn {  
  color: #000;  
  background-color: #fff;  
  border: 2px solid #cecece;  
  padding: .35rem 1rem .25rem 1rem;  
  font-size: 1rem;  
}
```

```
.btn:hover {  
  background-color: #ecf2fd;  
}  
  
.btn:active {  
  background-color: #d1e0fe;  
}  
  
.btn:focus {  
  outline: none;  
  border: black solid 2px;  
}  
  
.btn-primary {  
  color: #fff;  
  background-color: #000;  
  width: 100%;  
  padding: .75rem;  
  font-size: 1.3rem;  
  border: black solid 2px;  
  margin: 1rem 0;  
}  
  
.btn-primary:hover {  
  background-color: #444242;  
}  
  
.btn-primary:focus {  
  color: #000;  
  outline: none;  
  border: #000 solid 2px;  
  background-color: #d7ecff;  
}  
  
.btn-primary:active {  
  background-color: #212020;  
}
```

### 3.7. Estilo do Componente

É possível também definir estilos específicos para cada componente através do arquivo **XXX.component.css**, onde XXX é o nome do componente específico. Até este momento, nosso projeto possui somente um componente Angular, chamado por default de "app". Dessa forma, as definições de estilo desse componente devem ser declaradas dentro do arquivo **"app.component.css"**.

Experimente definir os estilos do AppComponent, alterando o conteúdo do arquivo **app.component.css** com as seguintes definições:

```
body {  
  color: #4d4d4d;  
  background-color: #f5f5f5;  
  color: #4d4d4d;  
}  
  
.main {  
  max-width: 500px;  
  width: 85%;  
  margin: 2rem auto;  
  padding: 1rem;  
  text-align: center;  
  box-shadow: 0 2px 4px 0 rgba(0, 0, 0, .2), 0 2.5rem 5rem 0 rgba(0, 0, 0, .1);  
}  
  
@media screen and (min-width: 600px) {  
  .main {  
    width: 70%;  
  }  
}  
  
label {  
  font-size: 1.5rem;  
  font-weight: bold;  
  display: block;  
  padding-bottom: 1rem;  
}  
  
.lg-text-input {  
  width: 100%;  
  padding: 1rem;  
  border: 2px solid #000;  
  display: block;  
  box-sizing: border-box;  
  font-size: 1rem;  
}  
  
.btn-wrapper {  
  margin-bottom: 2rem;  
}  
  
.btn-menu {  
  flex-basis: 32%;  
}  
  
.active {  
  color: green;  
}
```

```
ul {  
  padding-inline-start: 0;  
}  
  
ul li {  
  list-style: none;  
}
```

É importante saber que as definições de estilo acima foram obtidas a partir do tutorial de inspiração deste conteúdo, originalmente publicado em:

[https://developer.mozilla.org/en-US/docs/Learn/Tools\\_and\\_testing/Client-side\\_JavaScript\\_frameworks/Angular\\_styling](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Angular_styling)

Dessa forma, é possível que nem todas as definições de estilo presente, estejam efetivamente sendo aproveitadas pelos elementos HTML apresentados pelo nosso App.

### 3.8. Testando no Browser

Abra novamente o browser utilizando o mesmo endereço:

**http://localhost:4200/**

Resultado esperado:



## 4. CRIANDO UM NOVO COMPONENTE

Vamos agora criar um novo componente Angular para representar cada "Tarefa" apresentada. Este componente facilitará implementarmos os seguintes recursos:

Marcar ou desmarcar uma tarefa com o status "Realizada"

Alterar a descrição de uma tarefa existente

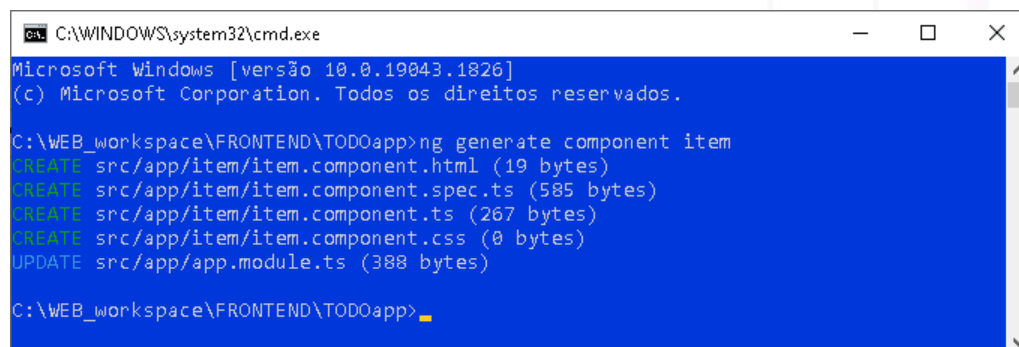
Remover uma tarefa

Por questões didáticas, vamos batizar este novo componente com o nome "item". É importante lembrar que ItemComponent irá definir os elementos que serão apresentados para cada tarefa, bem como suas funcionalidades e aparência. Vamos utilizar o nome "item" para não causar confusão com outras declarações utilizadas para representar as Tarefas.

Para criar o novo componente, execute o seguinte comando dentro da pasta TODOapp:

**ng generate component item**

O comando acima irá criar um componente chamado de ItemComponent, e será considerado um componente filho de AppComponent. A execução do comando apresentará o seguinte resultado:



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [versão 10.0.19043.1826]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\WEB_workspace\FRONTEND\TODOapp>ng generate component item
CREATE src/app/item/item.component.html (19 bytes)
CREATE src/app/item/item.component.spec.ts (585 bytes)
CREATE src/app/item/item.component.ts (267 bytes)
CREATE src/app/item/item.component.css (0 bytes)
UPDATE src/app/app.module.ts (388 bytes)

C:\WEB_workspace\FRONTEND\TODOapp>
```

#### 4.1. Referenciando ItemComponent

Agora vamos referenciar ItemComponent a partir da view de AppComponent. Vamos substituir a interpolação que apresentava a descrição da Tarefa, por uma referência a ItemComponent.

Para isso, basta substituir `{{tarefaDoLoop.descricao}}` por `<app-item></app-item>` dentro do arquivo **app.component.html**.

Após a alteração, o arquivo **app.component.html** ficará com o seguinte conteúdo:

```
<div class="main">
  <h1>Mozilla Dev - My To Do List</h1>
```

```

<input class="lg-text-input" #campoNovoItem />
<button class="btn-primary"
(click)="CREATE_tarefa(campoNovoItem.value)">
  Adicionar Tarefa</button>
<ul>
  <li *ngFor="let tarefaDoLoop of arrayDeTarefas">
    <app-item></app-item>
  </li>
</ul>
</div>

```

A linha em verde acima, indica que um ItemComponent (chamado de app-item) deve ser apresentado nessa posição.

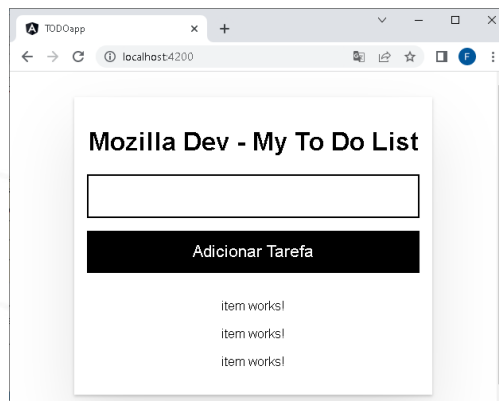
Consulte rapidamente o conteúdo do arquivo **item.component.html** e então visualize o resultado dessa modificação durante a execução do App.

#### 4.2. Testando no Browser

Abra novamente o browser utilizando o mesmo endereço:

**http://localhost:4200/**

Resultado esperado:



#### 4.3. Definindo as propriedades de ItemComponent

Substitua do código do arquivo **item.component.ts** por:

```

import { Component, Input, Output, EventEmitter } from '@angular/core';
import { Tarefa } from "../tarefa";

@Component({

```



```

selector: 'app-item',
templateUrl: './item.component.html',
styleUrls: ['./item.component.css']
})
export class ItemComponent {
  emEdicao = false;
  @Input() tarefa: Tarefa = new Tarefa("", false);
}

```

O código acima implementa as seguintes funcionalidades:

- Adiciona a linha do "import" permitindo que o ItemComponent também utilize as definições realizadas na classe "Tarefa".
- Define uma propriedade chamada "emEdicao" que permitirá sinalizar se uma Tarefa está em edição ou não.
- Define uma propriedade chamada "tarefa" que permitirá armazenar a Tarefa representada por ItemComponent.

#### 4.4. Definindo a view de ItemComponent

Substitua o código de **item.component.html** pelo seguinte conteúdo:

```

<div class="item">
  <input [id]="tarefa.descricao" type="checkbox"
[checked]="tarefa.statusRealizada"
  (change)="tarefa.statusRealizada = !tarefa.statusRealizada;" />
  <label [for]="tarefa.descricao">{{tarefa.descricao}}</label>

  <div *ngIf="!emEdicao" class="btn-wrapper">
    <button class="btn" (click)="emEdicao = true ">Editar</button>
    <button class="btn btn-warn">Remover</button>
  </div>
  <div *ngIf="emEdicao">
    <input class="sm-text-input" #editedItem placeholder="escreva o novo
nome da tarefa aqui"
    [value]="tarefa.descricao">
    <div class="btn-wrapper">
      <button class="btn" (click)="emEdicao = false ">Cancelar</button>
      <button class="btn btn-save"
(click)="tarefa.descricao=editedItem.value; emEdicao =
false;">Salvar</button>
    </div>
  </div>
</div>

```

O código acima, define os seguintes elementos para ItemComponent:

- Um **CHECKBOX** para informar se a tarefa foi feita ou não
- Um **LABEL** com a descrição da tarefa
- Quando a Tarefa não estiver sendo editada:
  - Um **BUTTON** para permitir editar a tarefa
  - Um **BUTTON** para permitir remover a tarefa
- Quando a Tarefa estiver sendo editada:
  - Um **INPUT** para definir a nova descrição da tarefa
  - Um **BUTTON** para permitir cancelar a edição
  - Um **BUTTON** para permitir salvar a alteração

No exemplo acima, a diretiva ngIf do Angular apresenta a <div> correspondente, dependendo do valor da variável "emEdicao"

#### 4.5. Transferindo os dados da Tarefa para ItemComponent

Considerando que agora cada Tarefa será representada por um `ItemComponent`, temos que transferir os dados da Tarefa armazenada em `AppComponent` (neste caso considerado como sendo o componente Pai) para o `ItemComponent` (neste caso considerado como sendo o componente Filho). Isso pode ser facilmente realizado através de um mecanismo chamado de "Property Binding". A transferência é feita durante a indicação do componente Filho dentro da view do componente Pai.

Em nosso caso, basta alterar o código de **app.component.html** acrescentando o trecho destacado abaixo na cor verde:

```
<div class="main">
  <h1>Mozilla Dev - My To Do List</h1>
  <input class="lg-text-input" #campoNovoItem />
  <button class="btn-primary"
(click)="CREATE_tarefa(campoNovoItem.value)">
    Adicionar Tarefa</button>
  <ul>
    <li *ngFor="let tarefaDoLoop of arrayDeTarefas">
      <app-item [tarefa]="tarefaDoLoop"></app-item>
    </li>
  </ul>
</div>
```

O código acrescentado acima, passa a Tarefa referenciada pela variável "tarefaDoLoop" para o atributo "tarefa" de `ItemComponent`.

#### 4.6. Definindo o estilo de ItemComponent

Altere o conteúdo do arquivo **item.component.css** com as seguintes definições:

```
.item {
  padding: .5rem 0 .75rem 0;
  text-align: left;
  font-size: 1.2rem;
}

.btn-wrapper {
  margin-top: 1rem;
  margin-bottom: .5rem;
}
```

```
.btn {  
  /* menu buttons flexbox styles */  
  flex-basis: 49%;  
}  
  
.btn-save {  
  background-color: #000;  
  color: #fff;  
  border-color: #000;  
}  
  
.btn-save:hover {  
  background-color: #444242;  
}  
  
.btn-save:focus {  
  background-color: #fff;  
  color: #000;  
}  
  
.checkbox-wrapper {  
  margin: .5rem 0;  
}  
  
.btn-warn {  
  background-color: #b90000;  
  color: #fff;  
  border-color: #9a0000;  
}  
  
.btn-warn:hover {  
  background-color: #9a0000;  
}  
  
.btn-warn:active {  
  background-color: #e30000;  
  border-color: #000;  
}  
  
.sm-text-input {  
  width: 100%;  
  padding: .5rem;  
  border: 2px solid #555;  
  display: block;  
  box-sizing: border-box;  
  font-size: 1rem;  
  margin: 1rem 0;  
}  
  
[type="checkbox"]:not(:checked),  
[type="checkbox"]:checked {  
  position: absolute;
```

```
left: -9999px;
}

[type="checkbox"]:not(:checked)+label,
[type="checkbox"]:checked+label {
  position: relative;
  padding-left: 1.95em;
  cursor: pointer;
}

[type="checkbox"]:not(:checked)+label:before,
[type="checkbox"]:checked+label:before {
  content: "";
  position: absolute;
  left: 0;
  top: 0;
  width: 1.25em;
  height: 1.25em;
  border: 2px solid #ccc;
  background: #fff;
}

[type="checkbox"]:not(:checked)+label:after,
[type="checkbox"]:checked+label:after {
  content: '\2713\0020';
  position: absolute;
  top: .15em;
  left: .22em;
  font-size: 1.3em;
  line-height: 0.8;
  color: #0d8dee;
  transition: all .2s;
  font-family: 'Lucida Sans Unicode', 'Arial Unicode MS', Arial;
}

[type="checkbox"]:not(:checked)+label:after {
  opacity: 0;
  transform: scale(0);
}

[type="checkbox"]:checked+label:after {
  opacity: 1;
  transform: scale(1);
}

[type="checkbox"]:checked:focus+label:before,
[type="checkbox"]:not(:checked):focus+label:before {
  border: 2px dotted blue;
}
```

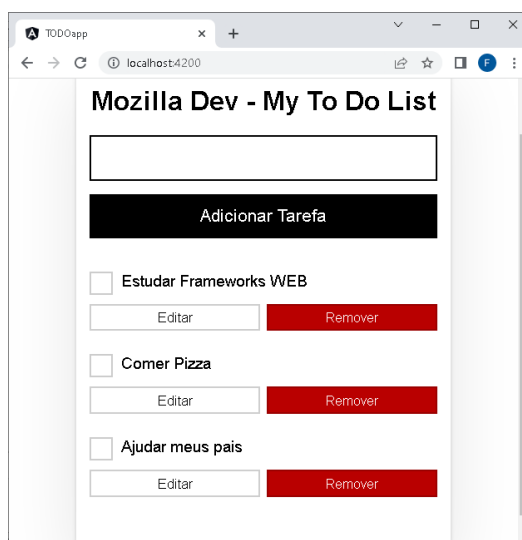
Pelo mesmo motivo explicado na definição do estilo de AppComponent, é possível que nem todas as definições de estilo acima, estejam efetivamente sendo aproveitadas pelos elementos HTML apresentados pelo componente correspondente.

#### 4.7. Testando no Browser

Abra novamente o browser utilizando o mesmo endereço:

**http://localhost:4200/**

Resultado esperado:



#### 4.8. Permitindo a remoção de Tarefas

Neste momento o único recurso que falta ser concluído em nosso App é permitir a "Remoção de Tarefas" da lista. É importante saber que, por mais que tenhamos incluído um botão para esta finalidade, o mesmo ainda não é capaz de realizar qualquer ação.

Para implementar essa solução, temos que notar que a lista de Tarefas está armazenada em um array existente dentro de AppComponent, e que o botão "Remove" está definido dentro de ItemComponent. Isso significa que o componente Filho (ItemComponent) deverá avisar o componente Pai (AppComponent) que o botão "Remove" foi pressionado.

Para resolver essa situação podemos fazer uso de um mecanismo chamado de "Event Binding", que permitirá que ItemComponent emita um evento para ser capturado por AppComponent.

#### 4.9. Declarando um emissor de evento

O primeiro passo necessário para implementar o Event Binding, é declarar uma propriedade no componente Filho utilizando a diretiva @Output e armazenando um objeto do tipo EventEmitter.

Adicione a seguinte linha destacada em verde ao código de **item.component.ts**:

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
import { Tarefa } from "../tarefa";

@Component({
  selector: 'app-item',
  templateUrl: './item.component.html',
  styleUrls: ['./item.component.css']
})
export class ItemComponent {
  emEdicao = false;
  @Input() tarefa: Tarefa = new Tarefa("", false);
  @Output() removeTarefa = new EventEmitter();
}
```

#### 4.10. Emitindo o evento

O próximo passo é avisar o componente Pai (AppComponent) sempre que o botão "Remover" for pressionado. Para isso vamos fazer o componente Filho emitir um evento no momento em que for detectado o "click" do botão "Remover". Segue destacado em verde, o código que deve ser adicionado em **item.component.html** com esse propósito:

```
<div class="item">
  <input [id]="tarefa.descricao" type="checkbox"
  [checked]="tarefa.statusRealizada"
  (change)="tarefa.statusRealizada = !tarefa.statusRealizada;" />
  <label [for]="tarefa.descricao">{{tarefa.descricao}}</label>

  <div *ngIf="!emEdicao" class="btn-wrapper">
    <button class="btn" (click)="emEdicao = true ">Editar</button>
    <button class="btn btn-warn" (click)="removeTarefa.emit()">Remover</button>
  </div>
  <div *ngIf="emEdicao">
    <input class="sm-text-input" #editedItem placeholder="escreva o novo nome da
    tarefa aqui"
    [value]="tarefa.descricao">
    <div class="btn-wrapper">
      <button class="btn" (click)="emEdicao = false ">Cancelar</button>
      <button class="btn btn-save" (click)="tarefa.descricao=editedItem.value;
      emEdicao = false;">Salvar</button>
    </div>
  </div>
</div>
```

&lt;/div&gt;

#### 4.11. Preparando AppComponent para eliminar a Tarefa

Considerando que a lista de Tarefas está armazenada no componente Pai (AppComponent), devemos implementar um método que seja capaz de eliminar uma Tarefa do arrayDeTarefas. Segue o código do método que deve ser adicionado em **app.component.ts** com esse propósito:

```
DELETE_tarefa(tarefaAserRemovida : Tarefa) {  
  this.arrayDeTarefas.splice(this.arrayDeTarefas.indexOf(tarefaAserRemovida),  
    1);  
}
```



#### 4.12. Detectando o evento em AppComponent

Insira o código destacado em verde em **app.component.html**. Esta construção executará o método DELETE\_tarefa no momento em que for detectado o evento "removeTarefa" emitido pelo componente Filho.

```
<div class="main">
  <h1>Mozilla Dev - My To Do List</h1>
  <input class="lg-text-input" #campoNovoItem />
  <button class="btn-primary"
(click)="CREATE_tarefa(campoNovoItem.value)">
    Adicionar Tarefa</button>
  <ul>
    <li *ngFor="let tarefaDoLoop of arrayDeTarefas">
      <app-item [tarefa]="tarefaDoLoop"
(removeTarefa)="DELETE_tarefa(tarefaDoLoop)">
    </app-item>
    </li>
  </ul>
</div>
```

#### 4.13. Testando no Browser

Abra novamente o browser utilizando o mesmo endereço:

**http://localhost:4200/**

Experimente testar todos os recursos do programa como Adicionar, Editar e Remover tarefas.

#### 4.14. Considerações finais

Ao chegarmos aqui finalizamos a primeira versão de nosso App Angular.

É importante notar que neste momento, nosso App não é capaz de persistir as informações das Tarefas cadastradas. Isso acontece porque todo o armazenamento dos dados está sendo realizado em memória.

Ao longo das aulas vamos aprender como armazenar as informações em um Banco de Dados, fazendo uso de uma arquitetura simplificada de micro-serviços disponibilizados a partir de uma REST API.

## 5. ESTRUTURA DE UM APP ANGULAR

Antes de partirmos para falar sobre Data Binding, é importante entender um pouco da estrutura definida pelo Framework Angular.

Um App Angular é formado por blocos de construção chamados de "Componentes". Um componente é constituído principalmente de:

- Lógica (arquivo com extensão **".ts"** também chamado de class)
- View (arquivo com extensão **".html"** também chamado de template)
- Estilos (arquivo com extensão **".css"** também chamado de style)

Ao criar uma nova aplicação Angular, automaticamente é criado um componente principal chamado de "app". É comum chamarmos esse componente de AppComponent.

Para cada componente existirá uma pasta com seu próprio nome. Dessa forma é possível encontrarmos os seguintes arquivos dentro da pasta "app":

### 5.1.1. *app.component.ts*

Contém a lógica do AppComponent. Considerando que ele é o componente principal do app, então este código define a lógica da página principal do App (class).

### 5.1.2. *app.component.html*

Conteúdo HTML do AppComponent que será inicialmente aberto no browser (template).

### 5.1.3. *app.component.css*

Contém o estilo do AppComponent (style)

Considerando que o componente "app" é o componente principal da aplicação, vamos encontrar também um outro arquivo chamado de **"app.module.ts"** que especifica todos os componentes usados pelo App.

Sempre que você cria um novo componente com o Angular CLI, automaticamente serão criados os arquivos conforme essa estrutura.

Exemplo:

Considere que você deseja criar um componente chamado de "**produto**". Para isso você poderá utilizar o seguinte comando do Angular CLI:

## ng generate component produto

Nesse caso é criada uma pasta chamada "**produto**" dentro da pasta "**app**", com os seguintes arquivos:

- produto.component.ts
- produto.component.html
- produto.component.css

Um arquivo adicional com o nome "produto.component.spec.ts" também é criado para o propósito de realização de testes.

### 5.2. Componentes Pai e Filho

Sempre que um componente utiliza outros componentes, é estabelecida uma relação de Pai para Filho entre eles, ou seja, o componente que **utiliza** outro componente é considerado o componente **Pai**, enquanto o componente utilizado é considerado o componente **Filho**.

No exemplo que estamos utilizando em nossas aulas, o componente do tipo "**app**" faz uso de componentes do tipo "**item**". Esse relacionamento é evidenciado no momento em que o componente "**app**" apresenta em sua view, uma referência a componentes do tipo "**item**".

Abaixo podemos ver o arquivo "**app.component.html**", destacando em verde a utilização do componente "**item**":

```
<div class="main">
  <h1>Mozilla Dev - My To Do List</h1>
  <input class="lg-text-input" #campoNovoItem />
  <button class="btn-primary"
(click)="CREATE_tarefa(campoNovoItem.value)">
    Adicionar Tarefa</button>
  <ul>
```

```
<li *ngFor="let tarefaDoLoop of arrayDeTarefas">  
  <app-item></app-item>  
</li>  
</ul>  
</div>
```

Neste caso, podemos dizer que o componente "**app**" é pai do componente "**item**".

Esse relacionamento é muito importante para entender como funciona o Data Binding.

### 5.3. Data Binding

O termo "Data Binding" em Angular representa os mecanismos utilizados para transferir informações entre diferentes partes do aplicativo. Os três principais métodos de Data Binding do Angular são:

- Interpolation
- Property Binding
- Event Binding

O diagrama abaixo resume uma boa explicação sobre o propósito de cada método:

### 5.4. Interpolation

Este método permite transferir informações da classe para a view de um componente.

São utilizadas chaves duplas `{{}}` para referenciar a informação.

Esse tipo de construção aparece sempre declarada dentro do arquivo `".html"` do componente.

### 5.5. Property Binding

Este método permite transferir informações de um componente Pai para a uma propriedade de um componente Filho.

São utilizados colchetes `[]` para referenciar a propriedade do componente Filho. Para fazer uso desse recurso, é importante lembrar que a propriedade do componente Filho precisa ser declarada utilizando o Decorator `@Input`.

Esse tipo de construção aparece sempre declarada dentro do arquivo `".html"` do componente Pai.

## 5.6. Event Binding

Este método permite transferir informações de um componente Filho para um componente Pai.

A transferência de informações é feita através da emissão de um evento do componente Filho para o componente Pai.

Para que um filho possa emitir um evento, ele precisa declarar uma propriedade utilizando o Decorator `@Output`.

A emissão do evento pelo Filho ocorre quando o método `"emit()"` é executado a partir de um objeto do tipo `EventEmitter`.

Já dentro da view do componente Pai, são utilizados parêntesis para capturar o evento e executar um código que funcionará como manipulador do mesmo.

## 6. PREPARANDO O BANCO DE DADOS

Nesta parte do tutorial vamos entender como configurar um Banco de Dados para permitir o armazenamento futuro das informações de nosso App.

### 6.1. Usando MongoDB e Mongoose

MongoDB é um banco de dados open source NoSQL organizado em **Collections** e **Documents**, que funcionam de certo modo análogo a **Tabelas** e **Linhas** em banco de dados relacionais.

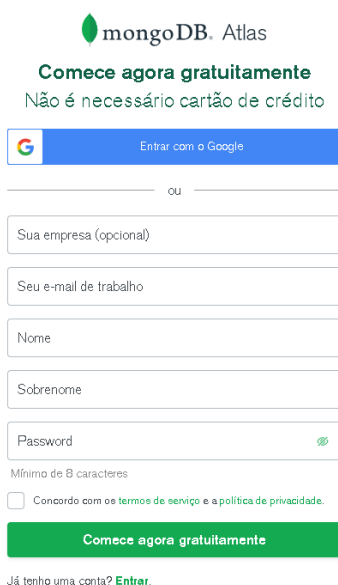
Mongoose funciona como uma interface para o MongoDB oferecendo a possibilidade de definição de Schemas e outras funcionalidades.

Como uma experiência adicional, ao contrário de utilizarmos um banco de dados local, vamos criar uma conta no **MongoDB Atlas** para utilizarmos um banco de dados hospedado na nuvem.

### 6.2. Criando uma conta no MongoDB Atlas

Acessar o seguinte endereço para se registrar no MongoDB Atlas:

<https://www.mongodb.com/pt-br/cloud/atlas/register>



The screenshot shows the MongoDB Atlas registration interface. At the top, it says 'mongoDB. Atlas' with a green leaf icon. Below that, it says 'Comece agora gratuitamente' and 'Não é necessário cartão de crédito'. There is a blue button with the Google logo and the text 'Entrar com o Google'. Below this, there is a line with 'ou' in the center. Then, there are several input fields: 'Sua empresa (opcional)', 'Seu e-mail de trabalho', 'Nome', 'Sobrenome', and 'Password'. Below the password field, it says 'Mínimo de 8 caracteres'. There is a checkbox with the text 'Concordo com os termos de serviço e a política de privacidade.'. At the bottom, there is a green button that says 'Comece agora gratuitamente'. Below the green button, it says 'Já tenho uma conta? Entrar.'

Em seguida, realize a verificação solicitada por email:



Email successfully verified!



Continue

Realize login em sua conta através do seguinte endereço:  
**<https://cloud.mongodb.com/user/login>**

### 6.3. Criando o banco de dados (Cluster)

Clique no botão "Build a database" para criar seu primeiro cluster.



Create a database

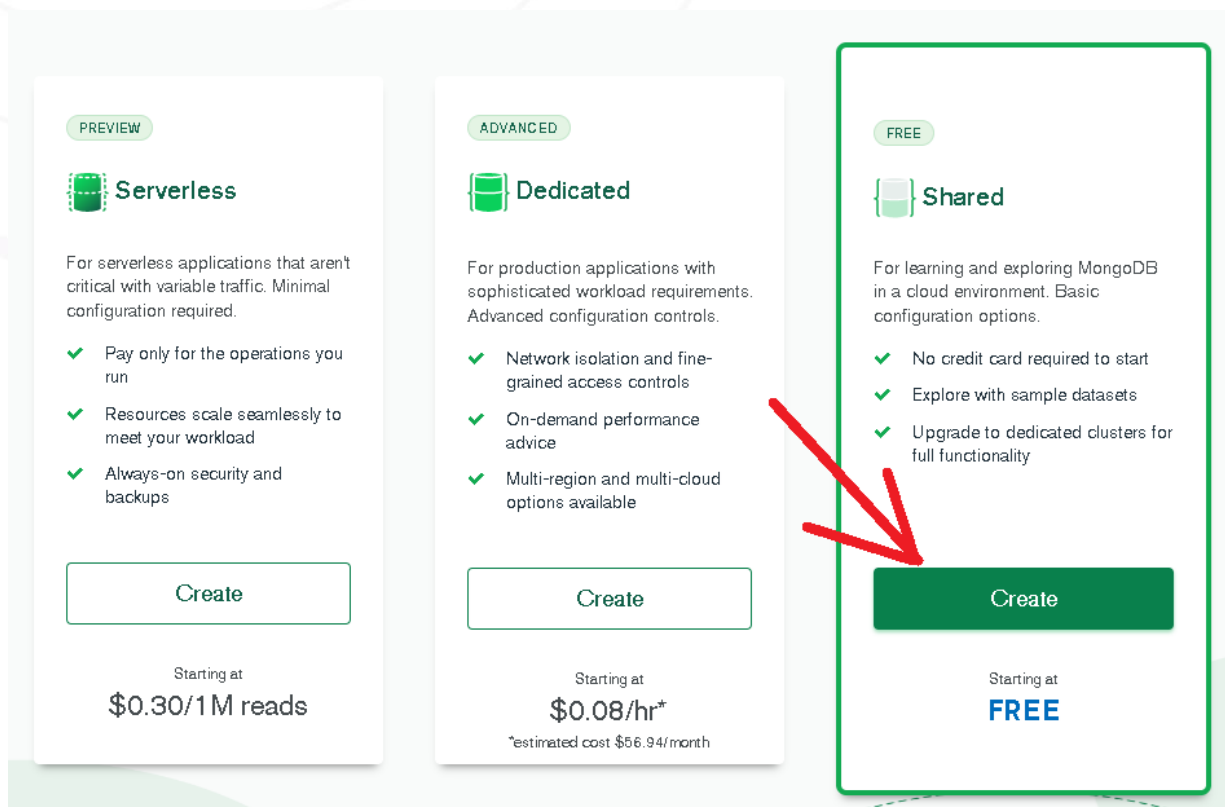
Choose your cloud provider, region, and specs.

Build a Database

Once your database is up and running, live migrate an existing MongoDB database into Atlas with our [Live Migration Service](#).

Selecione a opção "FREE" (Shared), pressionando o botão "CREATE"





PREVIEW	ADVANCED	FREE
<b>Serverless</b>	<b>Dedicated</b>	<b>Shared</b>
For serverless applications that aren't critical with variable traffic. Minimal configuration required.	For production applications with sophisticated workload requirements. Advanced configuration controls.	For learning and exploring MongoDB in a cloud environment. Basic configuration options.
<ul style="list-style-type: none"><li>✓ Pay only for the operations you run</li><li>✓ Resources scale seamlessly to meet your workload</li><li>✓ Always-on security and backups</li></ul>	<ul style="list-style-type: none"><li>✓ Network isolation and fine-grained access controls</li><li>✓ On-demand performance advice</li><li>✓ Multi-region and multi-cloud options available</li></ul>	<ul style="list-style-type: none"><li>✓ No credit card required to start</li><li>✓ Explore with sample datasets</li><li>✓ Upgrade to dedicated clusters for full functionality</li></ul>
<b>Create</b>	<b>Create</b>	<b>Create</b>
Starting at <b>\$0.30/1M reads</b>	Starting at <b>\$0.08/hr*</b> <small>*estimated cost \$56.94/month</small>	Starting at <b>FREE</b>

Certifique-se de manter selecionada a opção "M0 Sandbox" para Cluster Tier e clique no botão verde "Create Cluster".

### 6.3.1. Configurações de acesso seguro

Na sequência, vamos definir as opções no "Security Quickstart" para permitir o acesso futuro ao banco de dados, configurando:

- Usuário e senha para autenticar ao banco de dados
- Lista de endereços IP permitidos para conectar ao banco de dados

#### Autenticação

Vá em "How would you like to authenticate your connection" para definir o nome de usuário e senha.

Selecione a opção "**Username and password**".

Defina "**Username**" com: **joaoXXXXXX**, substituindo "joao" pelo seu primeiro nome e XXXXXX pelo seu RA.

Defina "**Password**" com uma **senha** de sua preferência

Clique no botão "**Create user**"

### Origem de acesso

Vá em "Where would you like to connect from" para definir as possíveis origens de acesso ao seu Banco de Dados.

Selecione a opção "**My local environment**"

Preencha o campo "**IP Address**" com: **0.0.0.0/0**

Ao definirmos o IP 0.0.0.0/0 permitiremos acesso a partir de qualquer endereço. Perceba que esta opção não é segura, porém facilitará nosso trabalho durante a elaboração do sistema de estudo.

Preencha o campo "**Description**" com: **Teste**

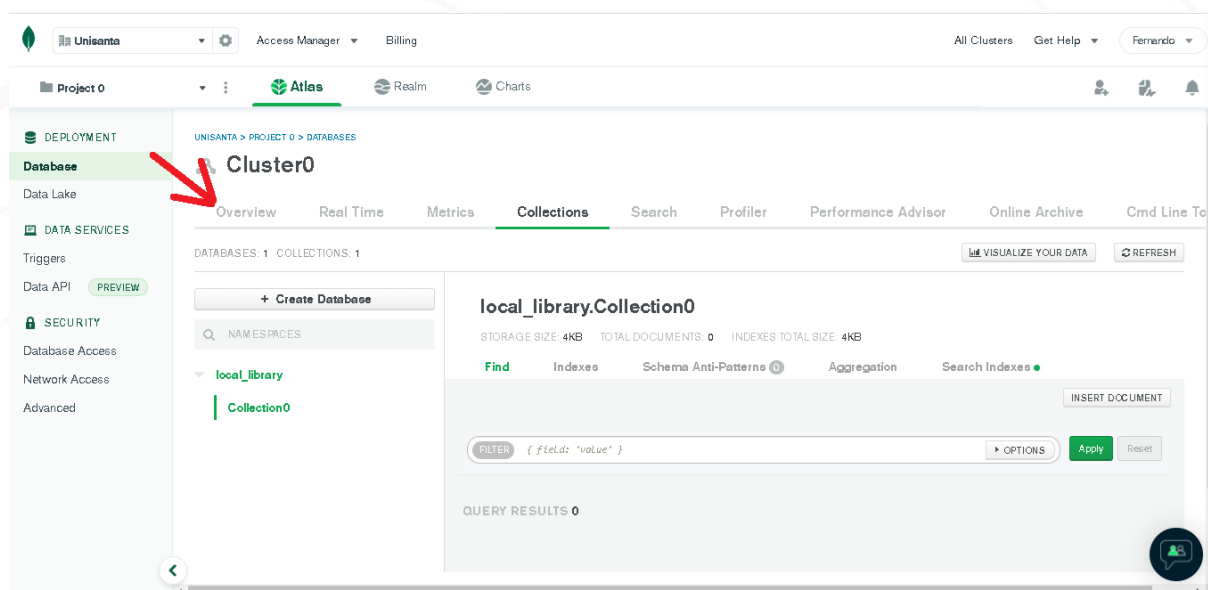
Pressione o botão "**Add Entry**"

Pressione o botão verde "**Finish and Close**"

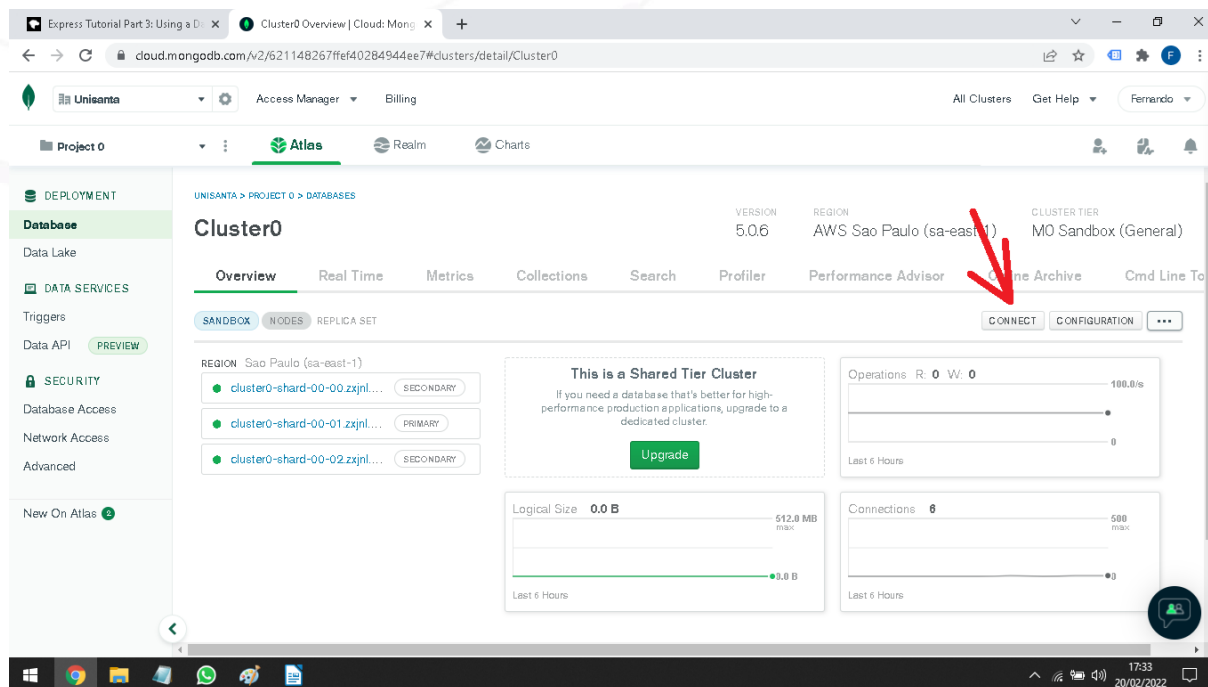
Pronto. Neste momento aparecerá uma página informando que seu cluster está sendo criado.

### Obtendo a URL de acesso ao Banco de Dados

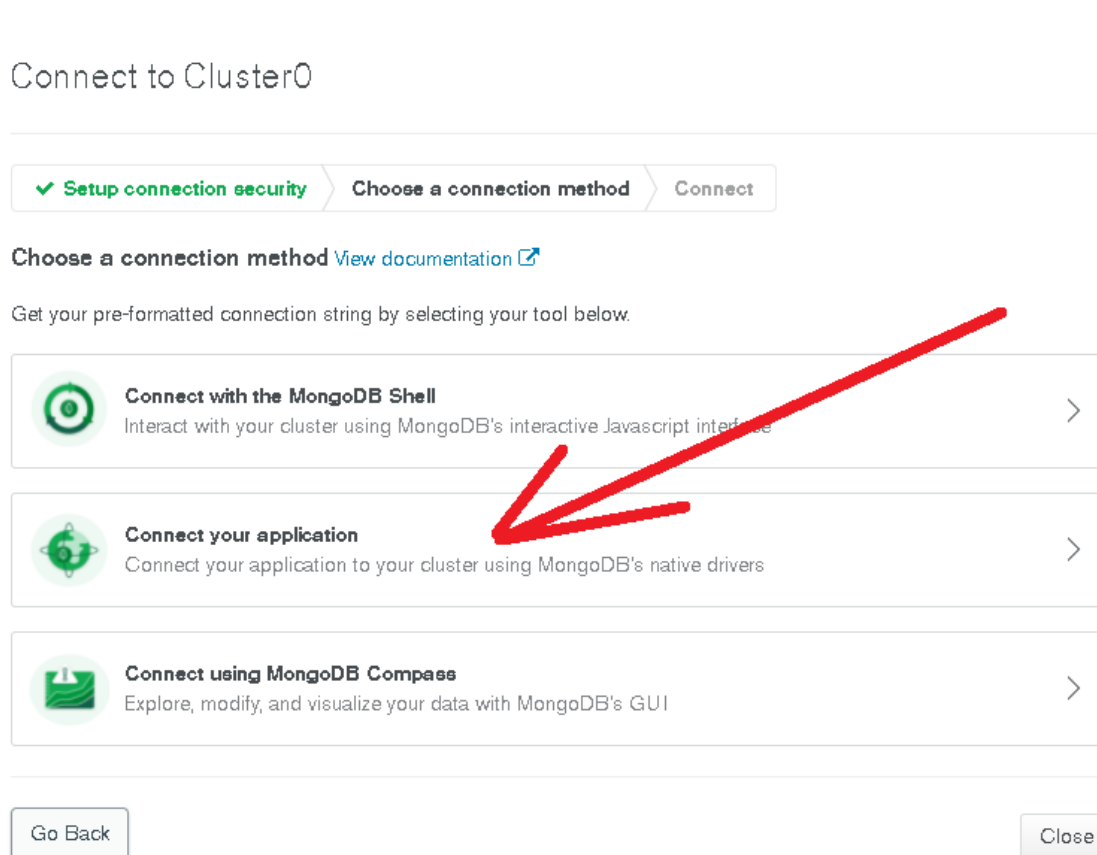
Clique em "**Overview**":



Clique em **"CONNECT"**:



Clique em **"Connect your application"**:



Copie a URL de conexão apresentada no destaque abaixo:

Substitua **<password>** pela sua senha

Substitua **"myFirstDatabase"** por **"tarefasDB"**

Salve essa URL como sendo a sua URL de conexão ao banco de dados.

#### 6.4. Testando a conexão com o Banco de Dados

Baixe o projeto "teste-banco-dados.zip" disponível em:

**brnqs.com/training/teste-banco-dados.zip**

Descompacte o conteúdo do arquivo e instale as dependências através do comando:

**npm install**

Execute o script através do seguinte comando, utilizando a sua URL de conexão:

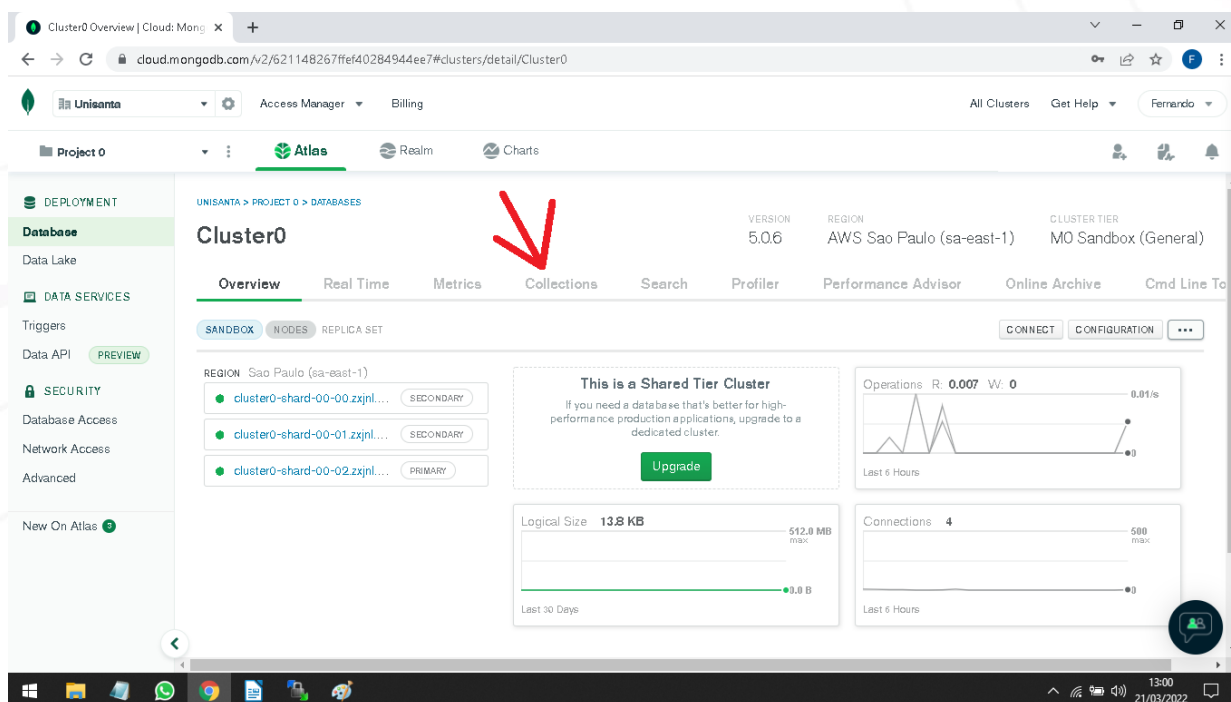
**node populateDB "coloque sua URL de conexão aqui"**

Obs: Coloque sua URL entre aspas

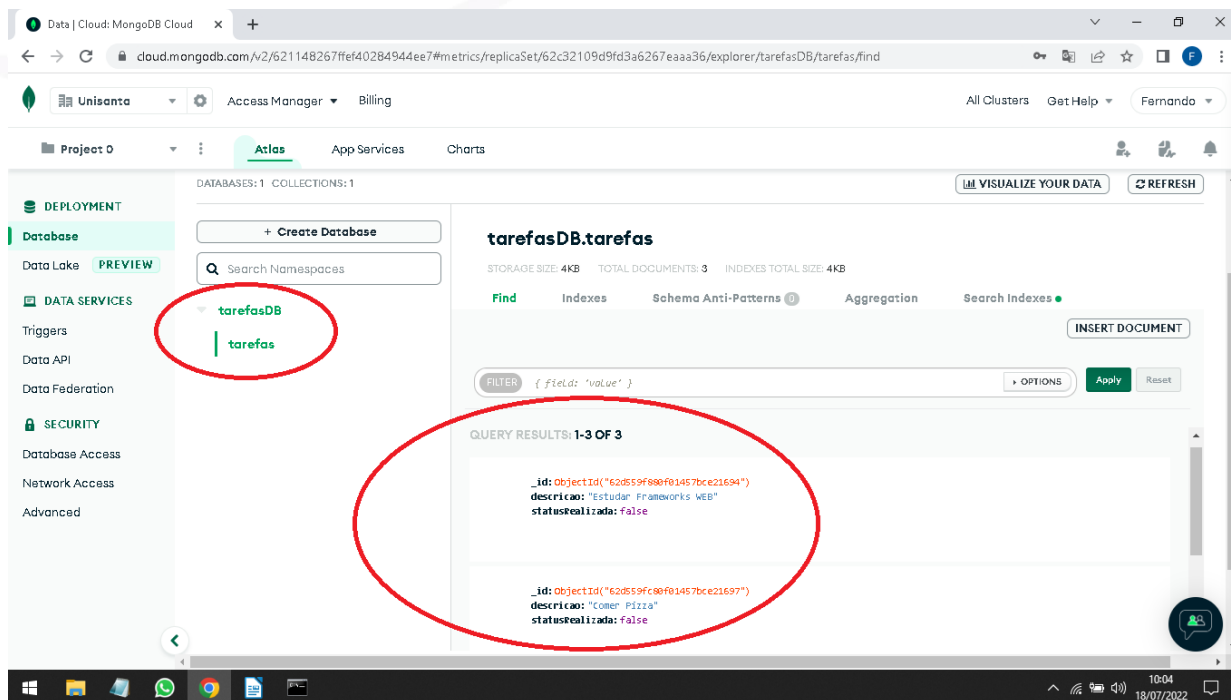
##### 6.4.1. Constatando que os dados foram gravados com sucesso

Volte à sua conta no MongoDB Atlas

Clique em **"Collections"**



Constate que 3 registros foram gravados com sucesso no banco de dados chamado **"tarefasDB"** dentro da Collection chamada de **"tarefas"**:



## 7. BACKEND COM EXPRESS

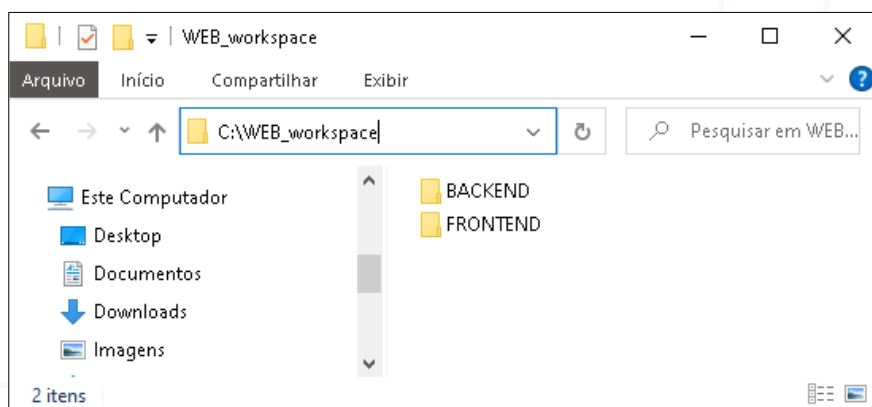
Nesta aula vamos desenvolver o programa que vai rodar no Backend, permitindo a manipulação de dados no Banco de Dados criado na aula anterior.

O Backend será desenvolvido no formato de uma WEB API, disponibilizando endpoints para a realização de operações CRUD (Create, Read, Update e Delete) no Banco de Dados.

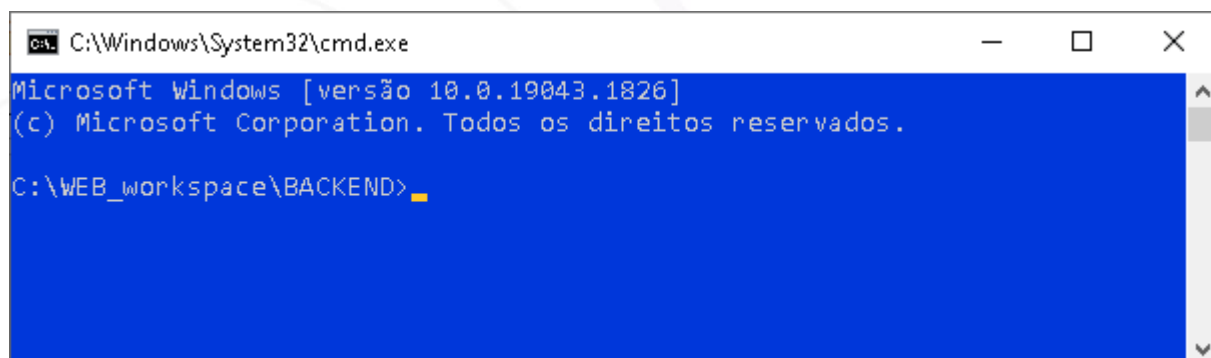
### 7.1. Criando o workspace

Acesse a pasta "C:\WEB\_workspace" de seu computador, criada no início de nossos estudos.

Dentro dessa pasta crie uma nova pasta chamada de BACKEND (no mesmo nível da pasta FRONTEND criada anteriormente). Ex:



Acesse o prompt de comando do Windows dentro da pasta BACKEND. É aqui onde você executará os primeiros comandos de criação da API:



## 7.2. Criando o projeto da API

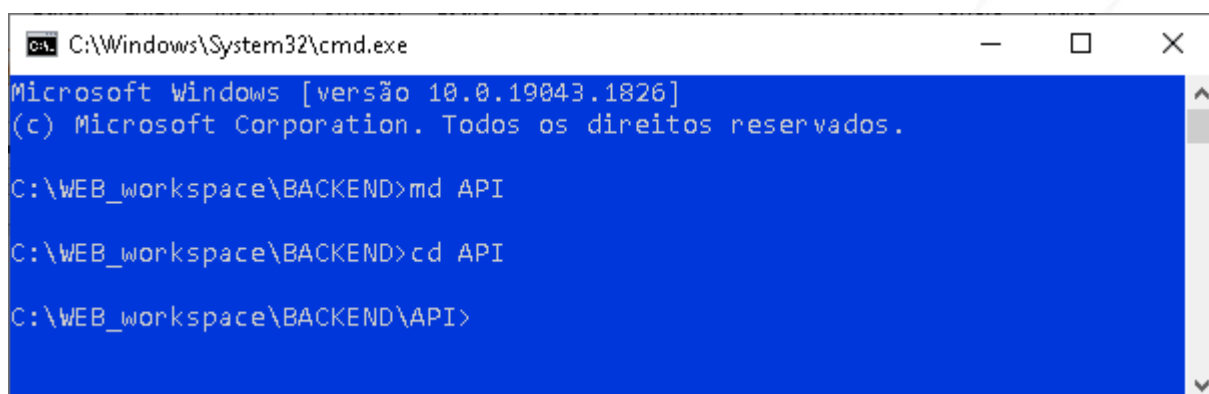
Abra o prompt de comando dentro da pasta BACKEND, e execute o seguinte comando para criar uma pasta chamada API para armazenar o código de nosso projeto:

**md API**

Entre na pasta API utilizando o seguinte comando:

**cd API**

Segue estado do prompt após executarmos os comandos acima:



```
C:\Windows\System32\cmd.exe
Microsoft Windows [versão 10.0.19043.1826]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\WEB_workspace\BACKEND>md API

C:\WEB_workspace\BACKEND>cd API

C:\WEB_workspace\BACKEND\API>
```

Execute o seguinte comando para criar a estrutura inicial do projeto (respondendo com ENTER para todas as perguntas realizadas):

**npm init**

Após a execução desse comando, o arquivo **package.json** é criado dentro da pasta API.

Agora vamos adicionar os módulos do framework "**express**" ao nosso projeto:

**npm i express --save**

Agora vamos adicionar o módulo "**nodemon**" para monitorar as alterações em nosso código fonte e reiniciar o servidor automaticamente sempre que for detectada uma alteração no código durante o desenvolvimento:

**npm i nodemon -g --save-dev**

Agora vamos adicionar o módulo "**mongoose**" para permitir acesso ao Banco de Dados:

### **npm i mongoose --save**

Execute o VsCode para permitir a edição do código dentro da pasta atual:

**code .**

Crie um arquivo chamado de "**index.js**" com o seguinte conteúdo:

```
const express = require('express');
const app = express();
app.use((req, res, next) => {
  res.setHeader("Access-Control-Allow-Origin", "*");
  res.setHeader('Access-Control-Allow-Methods', 'HEAD, GET, POST, PATCH, DELETE');
  res.header(
    "Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept"
  );
  next();
});
app.use(express.json());
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server Started at ${PORT}`)
})
// Obtendo os parametros passados pela linha de comando
var userArgs = process.argv.slice(2);
var mongoURL = userArgs[0];

//Configurando a conexao com o Banco de Dados
var mongoose = require('mongoose');
mongoose.connect(mongoURL, {
  useNewUrlParser: true, useUnifiedTopology:
    true
});
mongoose.Promise = global.Promise;
const db = mongoose.connection;
db.on('error', (error) => {
  console.log(error)
})
db.once('connected', () => {
  console.log('Database Connected');
})
```

O código acima vai inicializar a API respondendo pela porta 3000. Em seguida vai iniciar uma conexão com o Banco De Dados utilizando a URL de conexão passada como parâmetro através da linha de comando.

Caso a conexão com o Banco de Dados ocorra com sucesso, irá apresentar a mensagem "Database Connected", caso contrário apresentará uma mensagem indicando qual foi o erro ocorrido.

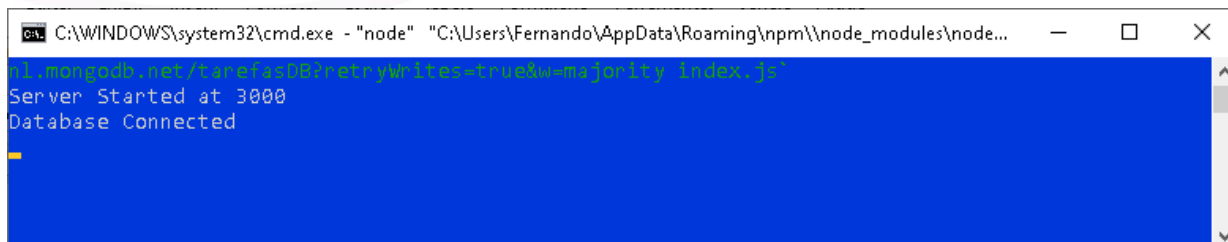
Execute o servidor através do seguinte comando:



**nodemon index** "coloque sua URL de conexão aqui"

Obs: Coloque sua URL de conexão com o banco de dados "entre aspas"

Deverá aparecer o seguinte resultado no console:



```
C:\WINDOWS\system32\cmd.exe - "node" "C:\Users\Fernando\AppData\Roaming\npm\node_modules\node...  
hl.mongodb.net/tarefas?retryWrites=true&w-majority index.js"  
Server Started at 3000  
Database Connected
```

### 7.3. Definindo o modelo de dados

Antes de criarmos as rotas dos endpoints, vamos definir o modelo dos dados a serem armazenados no MongoDB. Para isso vamos criar um arquivo chamado de "API/models/tarefa.js" com o seguinte conteúdo:

```
const mongoose = require('mongoose');  
const schemaTarefa = new mongoose.Schema({  
  descricao: {  
    required: true,  
    type: String  
  },  
  statusRealizada: {  
    required: true,  
    type: Boolean  
  },  
},  
{  
  versionKey: false  
})  
module.exports = mongoose.model('Tarefa', schemaTarefa)
```

O schema acima define o armazenamento dados na Collection "tarefas".  
Dentro da Collection, cada tarefa (Document) possuirá:

- Um campo "**descricao**" do tipo **string**
- Um campo "**statusRealizada**" do tipo **boolean**.

## 7.4. Criando a estrutura de rotas

A nossa API deverá ser capaz de disponibilizar micro-serviços para manipular informações no Banco de Dados. Esses serviços permitirão realizar as operações de Gravação (CREATE), Leitura (READ), Atualização (UPDATE) e Remoção (DELETE) de informações no Banco. Esse conjunto de operações costuma-se chamar de CRUD, que são as iniciais das siglas em Inglês.

Para conseguir isso, devemos especificar um endereço (endpoint) para cada serviço disponibilizado. Adicionalmente ao endpoint, devemos também associar o método esperado para cada tipo de requisição.

Segue tabela com os endpoints dos serviços a serem disponibilizados por nossa API:

Endpoint	Método HTTP	Operação CRUD	Objetivo
/api/post	POST	CREATE	Grava uma nova tarefa
/api/getAll	GET	READ	Obter todas as tarefas (itens)
/api/update/id	PATCH	UPDATE	Atualiza a tarefa identificada pelo id
/api/delete/id	DELETE	DELETE	Elimina a tarefa identificada pelo id

Para cada endpoint deverá ser criada uma rota para tratar as solicitações realizadas pelo cliente. Para isso, vamos criar um arquivo chamado de "**API/routes/routes.js**", com o seguinte conteúdo, para iniciar a preparação da estrutura de rotas.

```
const express = require('express');
const router = express.Router()
module.exports = router;
const modeloTarefa = require('../models/tarefa');
```

Agora vamos direcionar para o arquivo de rotas, o tratamento de todos os endpoints iniciados pelo prefixo "/api". Para isso adicione as linhas abaixo destacadas em verde no arquivo index.js:

```
c const express = require('express');
const app = express();
app.use(express.json());
const routes = require('./routes/routes');
app.use('/api', routes);
```

```
app.listen(3000, () => {
  console.log(`Server Started at ${3000}`)
})
// Obtendo os parametros passados pela linha de comando
var userArgs = process.argv.slice(2);
var mongoURL = userArgs[0];

//Configurando a conexao com o Banco de Dados
var mongoose = require('mongoose');
mongoose.connect(mongoURL, {
  useNewUrlParser: true, useUnifiedTopology:
    true
});
mongoose.Promise = global.Promise;
const db = mongoose.connection;
db.on('error', (error) => {
  console.log(error)
})
db.once('connected', () => {
  console.log('Database Connected');
})
})
```

### 7.5. Criando a rota para /api/post

Agora sim podemos inserir a rota para o método "POST" ao final do arquivo "routes.js":

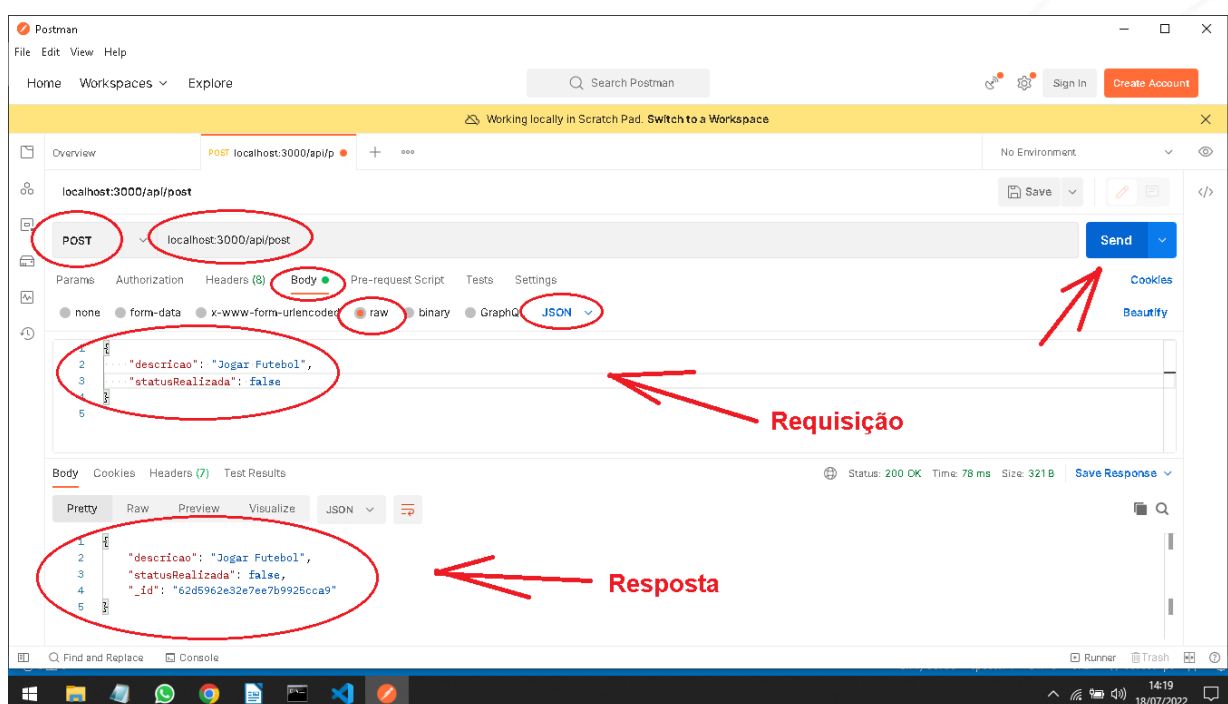
```
router.post('/post', async (req, res) => {
  const objetoTarefa = new modeloTarefa({
    descricao: req.body.descricao,
    statusRealizada: req.body.statusRealizada
  })
  try {
    const tarefaSalva = await objetoTarefa.save();
    res.status(200).json(tarefaSalva)
  }
  catch (error) {
    res.status(400).json({ message: error.message })
  }
})
```

A rota acima vai tratar as requisições HTTP realizadas pelo método POST para o endpoint "/api/post". O código cria um objeto do tipo modeloTarefa, utilizando os parâmetros de "descricao" e "statusRealizada" passados através do body da requisição HTTP. Em seguida o código solicita a gravação do objeto no Banco de Dados e retorna a resposta para o cliente.

## 7.6. Testando o endpoint /api/post

Para testar o endpoint, podemos realizar a requisição HTTP utilizando o programa **Postman**.

Transmita uma requisição utilizando o método **POST** para o endpoint **"localhost:3000/api/post"**, passando um JSON com a **"descricao"** e **"statusRealizada"** da tarefa no corpo (body) da requisição. Ex:



Entre em sua conta no **MongoDB Atlas** para constatar que o dado foi corretamente gravado.

### 7.7. Criando a rota para /api/getAll

Insira a seguinte rota ao final de "routes.js":

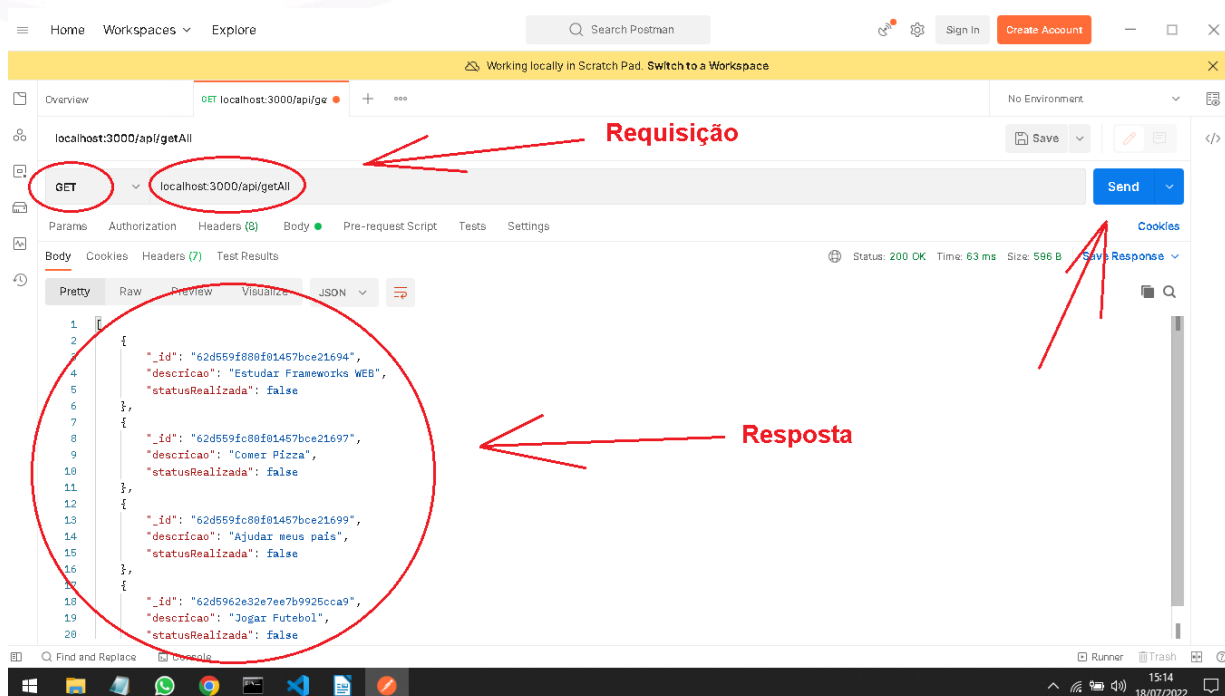
```
router.get('/getAll', async (req, res) => {  
  try {  
    const resultados = await modeloTarefa.find();  
    res.json(resultados)  
  }  
  catch (error) {  
    res.status(500).json({ message: error.message })  
  }  
})
```

A rota acima vai tratar as requisições HTTP realizadas pelo método GET para o endpoint "/api/getAll".

O código acima utiliza o método "find" para ler todos os registros do Banco de Dados e retorna como resposta no formato de um json para o cliente.

## 7.8. Testando o endpoint /api/getAll

Utilizando o Postman, transmita uma requisição utilizando o método GET para o endpoint "localhost:3000/api/getAll". Ex:



## 7.9. Criando a rota para /api/delete/id

Insira a seguinte rota ao final de "routes.js":

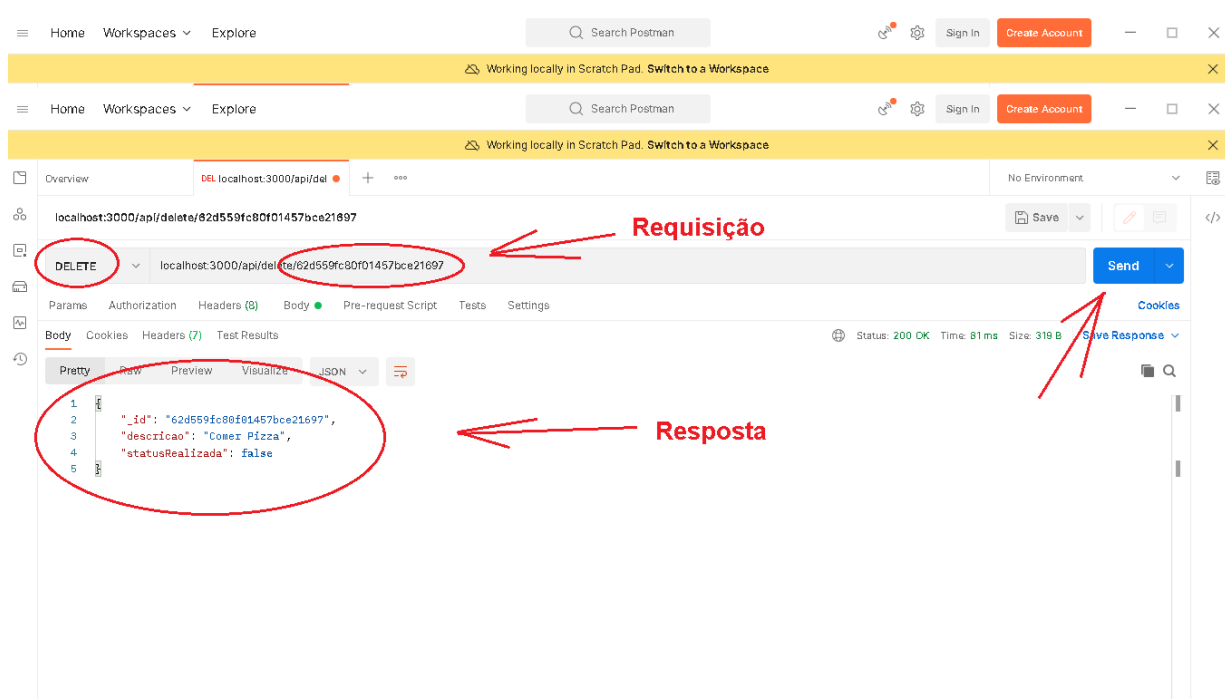
```
router.delete('/delete/:id', async (req, res) => {  
  try {  
    const resultado = await modeloTarefa.findByIdAndDelete(req.params.id)  
    res.json(resultado)  
  }  
  catch (error) {  
    res.status(400).json({ message: error.message })  
  }  
})
```

A rota acima vai tratar as requisições HTTP realizadas pelo método DELETE para o endpoint "/api/delete/id".

O código acima utiliza o método "findByIdAndDelete" para localizar e excluir o Document cujo "id" foi especificado como parte do endpoint.

## 7.10. Testando o endpoint `/api/delete/id`

Utilizando o Postman, transmita uma requisição utilizando o método DELETE para o endpoint "`localhost:3000/api/delete/id`", substituindo o "id" pelo código do "id" associado à tarefa a ser excluída no Bando de Dados. Ex:



Repare que o "id" utilizado na requisição foi o "id" da tarefa "Comer Pizza".

Após realizar a requisição, entre em sua conta no **MongoDB Atlas** para constatar que a tarefa foi corretamente eliminada.



### 7.11. Criando a rota para /api/update/id

Insira a seguinte rota ao final de "routes.js":

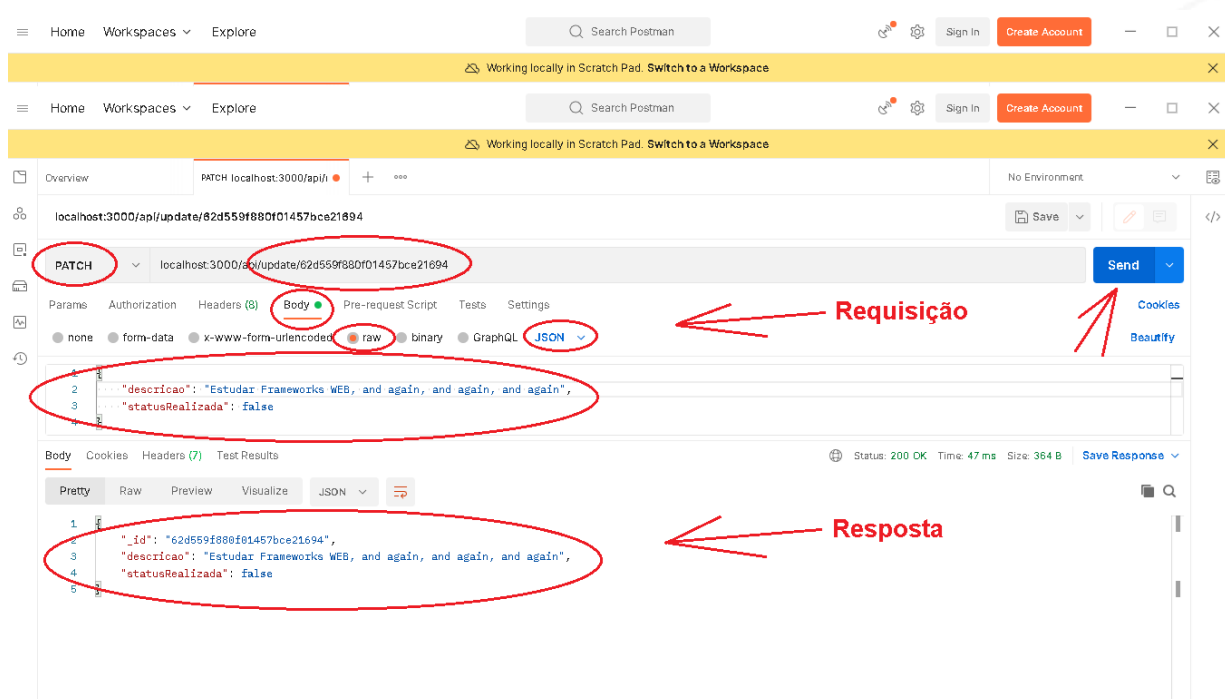
```
router.patch('/update/:id', async (req, res) => {  
  try {  
    const id = req.params.id;  
    const novaTarefa = req.body;  
    const options = { new: true };  
    const result = await modeloTarefa.findByIdAndUpdate(  
      id, novaTarefa, options  
    )  
    res.json(result)  
  }  
  catch (error) {  
    res.status(400).json({ message: error.message })  
  }  
})
```

A rota acima vai tratar as requisições HTTP realizadas pelo método PATCH para o endpoint "/api/update/id".

O código acima utiliza o método "findByIdAndUpdate" para localizar e atualizar o Document cujo "id" foi especificado como parte do endpoint. Ele então atualiza o registro encontrado com a "novaTarefa" recebida como parâmetro no body da requisição.

## 7.12. Testando o endpoint /api/update/id

Utilizando o Postman, transmita uma requisição utilizando o método PATCH para o endpoint "**localhost:3000/api/update/id**", substituindo o "id" pelo código do "id" associado à tarefa a ser alterada no Bando de Dados e passando um JSON com a nova "descricao" e "statusRealizada" da tarefa no corpo (body) da requisição. Ex:



Repare que o "id" utilizado na requisição é o "id" da tarefa "Estudar Frameworks WEB".

Perceba que a resposta apresenta uma nova descrição para a tarefa identificada com o mesmo "id" anterior.

Após realizar a requisição, entre em sua conta no **MongoDB Atlas** para constatar que a tarefa foi corretamente alterada.

## 8. INTEGRANDO FRONTEND E BACKEND

Agora vamos modificar nosso App Frontend para acessar os serviços disponíveis em nossa API desenvolvida para rodar no Backend.

Basicamente vamos alterar as operações de manipulação do array por requisições HTTP que acessam os endpoints da API. Dessa forma, nosso Frontend será capaz de utilizar dados armazenados no Banco De Dados armazenado na nuvem.

### 8.1. Preparando a comunicação

Agora vamos modificar nosso Frontend para conseguir emitir requisições HTTP para a API rodando no Backend.

### 8.2. Importando o módulo HttpClient

Modifique o arquivo `/src/app/app.module.ts` inserindo o código destacado em verde abaixo:

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

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

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

Obs: Cuidado para não esquecer de acrescentar a vírgula após a declaração **BrowserModule**. Perceba que ela também está destacada em verde.

Modifique o arquivo **"app.component.ts"** inserindo o código destacado em verde abaixo:

```
import { Component } from '@angular/core';
import { Tarefa } from "../tarefa";
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'TODOapp';

  arrayDeTarefas: Tarefa[] = [];
  apiURL : string;

  constructor(private http: HttpClient) {
    this.apiURL = 'http://localhost:3000';
    this.READ_tarefas();
  }

  CREATE_tarefa(descricaoNovaTarefa: string) {
    var novaTarefa = new Tarefa(descricaoNovaTarefa, false);
    this.arrayDeTarefas.unshift(novaTarefa);
  }

  READ_tarefas() {
    this.arrayDeTarefas = [
      new Tarefa("Estudar Frameworks WEB", false),
      new Tarefa("Comer Pizza", false),
      new Tarefa("Ajudar meus pais", false)
    ];
  }

  DELETE_tarefa(tarefaAserRemovida: Tarefa) {
    this.arrayDeTarefas.splice(this.arrayDeTarefas.indexOf(tarefaAserRemovida),
    1);
  }
}
```

Seu projeto agora está pronto para realizar requisições HTTP.

### 8.3. Executando o Backend

Acesse a pasta do Backend e execute o servidor utilizando o seguinte comando:

**nodemon index** "sua URL de conexão com o BD"

#### 8.4. Executando o Frontend

Acesse a pasta do Frontend e execute o programa utilizando o seguinte comando:

**ng serve**

#### 8.5. Lendo os dados a partir da API

Agora vamos fazer nosso App acessar a API para ler as tarefas armazenadas em nosso Banco de Dados. Para isso vamos enviar uma requisição HTTP com o método GET para o endpoint **"/api/getAll"**

Substitua o **corpo** do método **READ\_tarefas** dentro de **app.component.ts** pelo seguinte conteúdo destacado em verde:

```
READ_tarefas() {  
  this.http.get<Tarefa[]>(`${this.apiUrl}/api/getAll`).subscribe(  
    resultado => this.arrayDeTarefas=resultado;  
  }  
}
```

Perceba que agora o método **READ\_tarefas** alimenta o array com os dados obtidos a partir da API, ao contrário de popular o array com tarefas fixas.

#### 8.6. Testando no Browser

Abra o browser utilizando o endereço:

**http://localhost:4200/**

Se tudo estiver correto, você verá que as tarefas obtidas são aquelas que estão armazenadas em seu Banco de Dados armazenado na nuvem.

## 8.7. Salvando dados a partir da API

Agora vamos modificar nosso App para acessar a API para gravar novas Tarefas em nosso Bando de Dados. Para isso vamos enviar uma requisição HTTP com o método POST para o endpoint **`/api/post`** enviando a nova tarefa como parâmetro.

Modifique o **corpo** do método **CREATE\_tarefa** dentro de **app.component.ts**, conforme o novo conteúdo abaixo. Basicamente você vai substituir a linha **this.arrayDeTarefas.unshift(novaTarefa);** pelo código em verde abaixo:

```
CREATE_tarefa(descricaoNovaTarefa: string) {  
  var novaTarefa = new Tarefa(descricaoNovaTarefa, false);  
  this.http.post<Tarefa>(`${this.apiUrl}/api/post`, novaTarefa).subscribe(  
    resultado => { console.log(resultado); this.READ_tarefas(); });  
}
```

## 8.8. Testando no Browser

Abra o browser utilizando o endereço:

**`http://localhost:4200/`**

Experimente adicionar uma nova tarefa à lista de tarefas existentes.

Acesse sua conta do MongoDB Atlas e constate que a nova tarefa foi gravada com sucesso.

## 8.9. Removendo dados a partir da API

Agora vamos modificar nosso App para acessar a API para remover Tarefas em nosso Bando de Dados. Para isso vamos enviar uma requisição HTTP com o método DELETE para o endpoint **`/api/delete/id`** onde o id especifica qual item desejamos remover do banco de dados. Considerando a necessidade de utilização do parâmetro "id" a partir deste momento, então vamos também adicioná-lo como atributo da classe Tarefa, modificando o arquivo **`/src/app/tarefa.ts`** conforme destacado em verde abaixo:

```
export class Tarefa {  
  _id : string | undefined ;  
  descricao: string;  
  statusRealizada: boolean;  
  
  constructor(_descricao: string, _statusRealizada: boolean) {  
    this.descricao = _descricao;  
    this.statusRealizada = _statusRealizada;  
  }  
}
```

```
}
```

Substitua o **corpo** do método **DELETE\_tarefa** dentro de **app.component.ts** pelo seguinte conteúdo destacado em verde:

```
DELETE_tarefa(tarefaAserRemovida: Tarefa) {
  var indice = this.arrayDeTarefas.indexOf(tarefaAserRemovida);
  var id = this.arrayDeTarefas[indice]._id;
  this.http.delete<Tarefa>(`${this.apiUrl}/api/delete/${id}`).subscribe(
    resultado => { console.log(resultado); this.READ_tarefas(); });
}
```

### 8.10. Testando no Browser

Abra o browser utilizando o endereço e experimente remover uma tarefa existente:

**http://localhost:4200/**

### 8.11. Alterando dados a partir da API

Agora vamos modificar nosso App para acessar a API para alterar Tarefas em nosso Bando de Dados. Para isso vamos enviar uma requisição HTTP com o método PATCH para o endpoint **"/api/update/id"** enviando a tarefa modificada como parâmetro. Lembrando que o parâmetro o "id" especificará qual item desejamos alterar no banco de dados.

Esta alteração será um pouco maior, pois o App original fazia as alterações diretamente a partir da referência dos objetos armazenados no array. Como agora os dados precisam ser atualizados no Bando de Dados, precisaremos adicionar um método de UPDATE em **AppComponent** e executá-lo a partir das ações realizadas em **ItemComponent**.

Para isso, adicione o método **UPDATE\_tarefa** abaixo dentro de **app.component.ts**.

```
UPDATE_tarefa(tarefaAserModificada: Tarefa) {
  var indice = this.arrayDeTarefas.indexOf(tarefaAserModificada);
  var id = this.arrayDeTarefas[indice]._id;
  this.http.patch<Tarefa>(`${this.apiUrl}/api/update/${id}`,
    tarefaAserModificada).subscribe(
    resultado => { console.log(resultado); this.READ_tarefas(); });
}
```

Agora vamos adicionar um atributo para armazenar um objeto com o nome **modificaTarefa** do tipo **EventEmitter** dentro de **item.component.ts**, conforme exemplo

destacado

em

verde

abaixo:

```
...
export class ItemComponent {
  emEdicao = false;
  @Input() tarefa: Tarefa = new Tarefa("", false);
  @Output() removeTarefa = new EventEmitter();
  @Output() modificaTarefa = new EventEmitter();
}
```

Agora, vamos emitir esse evento nas duas possíveis situações de alteração de dados da tarefa, ou seja:

- Quando o usuário altera o status da tarefa
- Quando o usuário altera a descrição da tarefa

Para isso, seguem destacadas em verde, as alterações que devem ser realizadas em `item.component.html`:

```
<div class="item">
  <input [id]="tarefa.descricao" type="checkbox"
[checked]="tarefa.statusRealizada"
  (change)="tarefa.statusRealizada = !tarefa.statusRealizada;
modificaTarefa.emit()" />
  <label [for]="tarefa.descricao">{{tarefa.descricao}}</label>

  <div *ngIf="!emEdicao" class="btn-wrapper">
    <button class="btn" (click)="emEdicao = true ">Editar</button>
    <button class="btn btn-warn"
(click)="removeTarefa.emit()">Remover</button>
  </div>
  <div *ngIf="emEdicao">
    <input class="sm-text-input" #editedItem placeholder="escreva o novo
nome da tarefa aqui"
      [value]="tarefa.descricao">
    <div class="btn-wrapper">
      <button class="btn" (click)="emEdicao = false ">Cancelar</button>
      <button class="btn btn-save"
(click)="tarefa.descricao=editedItem.value; emEdicao = false;
modificaTarefa.emit()">Salvar</button>
    </div>
  </div>
</div>
```



Por fim, temos que solicitar a execução do método **UPDATE\_tarefa** assim que **AppComponent** detectar a ocorrência do evento "**modificaTarefa**".

Para isso, adicione a modificação destacada abaixo em verde, no código de **app.component.html**:

```
<div class="main">
  <h1>Mozilla Dev - My To Do List</h1>
  <input class="lg-text-input" #campoNovoItem />
  <button class="btn-primary"
(click)="CREATE_tarefa(campoNovoItem.value)">
    Adicionar Tarefa</button>
  <ul>
    <li *ngFor="let tarefaDoLoop of arrayDeTarefas">
      <app-item [tarefa]="tarefaDoLoop"
(removeTarefa)="DELETE_tarefa(tarefaDoLoop)"
(modificaTarefa)="UPDATE_tarefa(tarefaDoLoop)"></app-item>
    </li>
  </ul>
</div>
```

## 8.12. Testando no Browser

Abra o browser utilizando o endereço:

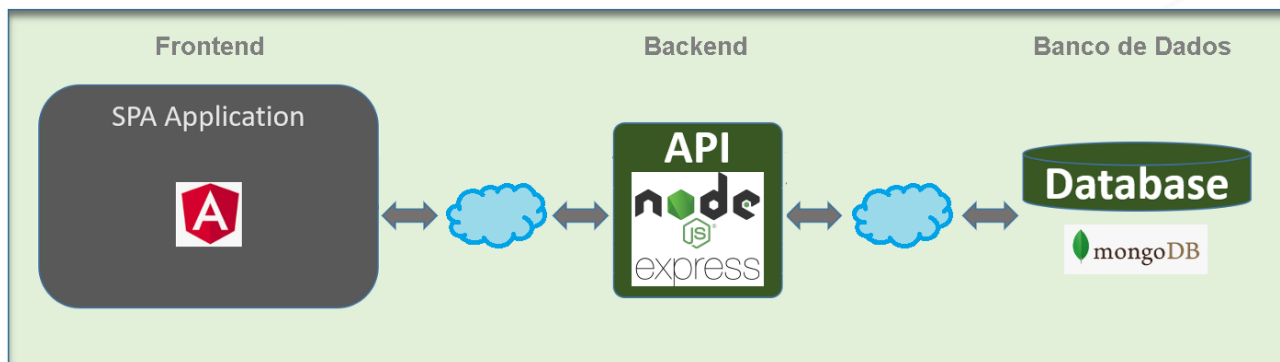
**http://localhost:4200/**

Experimente alterar o status e a descrição de uma tarefa existente.

## 9. HOSPEDANDO NA NUVEM

Chegou o grande momento onde vamos executar nossa solução em um ambiente de produção (Até o momento estávamos executando nossa solução em um ambiente de desenvolvimento).

Dessa forma é importante revermos a arquitetura definida originalmente:



Como podemos ver acima, nossa solução está dividida em três partes básicas:

- Frontend  
Representado pelo nosso TODOapp desenvolvido com o Framework Angular
- Backend  
Representado pela nossa API desenvolvida com o Framework Express
- Banco de Dados  
Utilizando o banco MongoDB

### 9.1. Definindo a hospedagem

Segue a definição de onde será hospedada cada parte da solução:

- Frontend  
Será hospedado em um servidor HTTP simples da Hostinger. Servirá simplesmente para baixar a SPA (Single Page Application) para o browser do cliente.
- Backend  
Será hospedado no Heroku, que oferece um servidor de aplicações, rodando o nosso código da API.
- Banco de Dados  
Já está hospedado no MongoDB Atlas. Será mantida a mesma conta.

### 9.2. Hospedagem do Backend

Considerando que nosso Banco de Dados já está hospedado, vamos partir para hospedar o nosso Backend. É importante saber que esta hospedagem será feita somente para efeito de demonstração e entendimento deste tipo de processo. As ações para realização deste processo podem variar muito ao longo do tempo e dependem exclusivamente dos procedimentos definidos por cada provedor. Em nosso exemplo, usaremos o Heroku, que é uma plataforma de cloud de bastante sucesso e amplamente conhecida pela comunidade. Neste momento, diversas outras plataformas existem no mercado para a mesma finalidade como: Amazon Web Services (AWS), Google Cloud Platform, Microsoft Azure, IBM Cloud, Oracle Cloud etc.

### 9.3. Preparando a API para rodar no Heroku

Ao colocar nossa API em um ambiente cloud, temos que entender como o mesmo fará para executar nossa aplicação sob demanda. No caso do Heroku, ao identificar que nossa aplicação é desenvolvida em Node Express, o Heroku possui a configuração padrão de executar nossa aplicação utilizando o comando:

#### **npm start**

O comando acima executa um script chamado de "**start**", declarado dentro do arquivo "**package.json**" de nosso projeto.

Dessa forma, vamos modificar esse arquivo, inserindo no conteúdo do script, o

comando de inicialização que utilizávamos para executar nossa API no ambiente de desenvolvimento.

Cabe, porém, observar duas pequenas diferenças:

- Por estarmos executando no ambiente de desenvolvimento, utilizávamos o comando **"nodemon"**. No ambiente de produção vamos substituir esse comando por **"node"**.
- Como o script necessita ficar entre aspas duplas, vamos substituir as aspas originais da URL por aspas simples.

Segue destacado em verde, a alteração que será necessária realizar no arquivo **"package.json"**:

```
{
  "name": "api",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index 'sua URL aqui'"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.1",
    "mongoose": "^6.4.6"
  }
}
```

Repare que será necessário incluir uma vírgula ao final do script "test"

#### 9.4. Abrindo sua conta no Heroku

O Heroku oferece diversos planos conforme sua necessidade de uso. Uma grande vantagem é que ele oferece também um plano básico gratuito que atenderá com muita folga as nossas necessidades durante aprendizado. Dessa forma, entre no site: <https://www.heroku.com/> e crie uma conta FREE pra você.

#### 9.5. Criando um novo App no Heroku

Clique no logo do Heroku para aparecer uma página chamada "Personal".

No canto direito você verá um botão chamado "NEW". Ao pressionar este botão selecione a opção "Create New App".

O Heroku irá perguntar qual o nome do App. Entre com o seguinte nome:

**apitarefasNNNNxxxxxx**

onde: **NNNN** será seu nome e **xxxxxx** será seu RA (tudo minúsculo). Utilize este mesmo procedimento até o final deste tutorial.

Caso surja alguma mensagem dizendo que o nome já existe, então acrescente seu sobrenome ou outras informações que permitam criar um endereço exclusivo.

Em seguida, pressione o botão "**Create App**".

Escolha a opção "**Heroku git**" como sendo o método de **Deployment**. Lembrando que instalamos o Git em nosso computador na primeira aula deste curso.

Abra o prompt de comando, dentro da pasta de nossa API, e execute o seguinte comando para instalar o Heroku CLI (caso ele nunca tenha sido instalado antes).

**npm install -g heroku**

Verifique se o Heroku foi instalado com sucesso através do seguinte comando:

**heroku -version**

Deve aparecer o número da versão instalada.

Realize o login pela linha de comando:

**heroku login**

Pressione ENTER para ele redirecioná-lo para uma página de login via browser. Ao concluir o login, volte ao prompt de comandos e perceba que o login foi reconhecido.

Defina um novo repositório **git** local, utilizando o seguinte comando:

**git init**

Garanta que seu email e nome de usuário estejam definidos para o git, executando os seguintes comandos:

**git config --global user.email "seu email"**

**git config --global user.name "seu nome"**

Associe o seu projeto local com o App criado remotamente no Heroku:

**heroku git:remote -a apitarefaNNNNxxxxxx**

Defina que todos os arquivos fazem parte do seu repositório:

**git add .**

Defina um commit da versão inicial do seu sistema:

**git commit -am "versao inicial"**

Transfira seu projeto para o repositório remoto:

**git push heroku master**

## 9.6. Testando no Browser

Abra o browser utilizando o endereço:

<https://apitarefaNNNNxxxxxx.herokuapp.com/api/getAll>

Se tudo estiver funcionando perfeitamente, o browser apresentará um json de todas as tarefas gravadas em seu banco de dados.

## 9.7. Hospedagem do Frontend

Por último, resta fazermos a hospedagem do Frontend para que nossa solução esteja disponibilizada 100% na nuvem.

## 9.8. Preparando o TODOapp para comunicar com a API

Agora que temos nossa API hospedada na nuvem, basta atualizar sua nova localização dentro do TODOapp.

Durante o desenvolvimento nossa API estava disponível no endereço:

<http://localhost:3000>

Ao hospedarmos a API no Heroku, seu novo endereço passou a ser:

<https://apitarefasNNNNxxxxxx.herokuapp.com>

Dessa forma, segue destacado em verde a alteração que deve ser feita no arquivo "app.component.ts" de nosso App:

```
...
export class AppComponent {
  title = 'TODOapp';
  arrayDeTarefas: Tarefa[] = [];
  apiURL: string;
  constructor(private http: HttpClient) {
    this.apiURL = 'https://apitarefasNNNNxxxxxx.herokuapp.com'
    this.READ_tarefas();
  }

  CREATE_tarefa(descricaoNovaTarefa: string) {
    ...
  }
}
```

### 9.9. Realizando o Build do App

Abra o prompt de comando dentro da pasta TODOapp.

Execute o seguinte comando para realizar o Build da aplicação:

**ng build**

O angular irá compilar seu projeto apresentando o seguinte resultado:

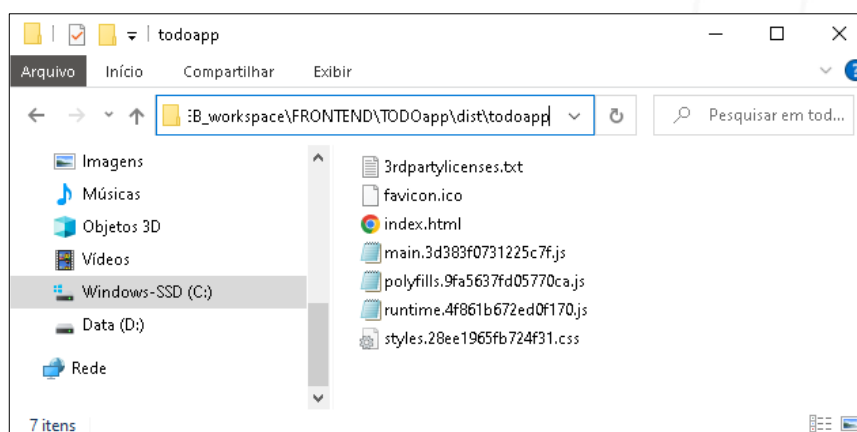
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [versão 10.0.19043.1826]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\WEB_workspace\FRONTEND\TODOapp>ng build
✓ Browser application bundle generation complete.
✓ Copying assets complete.
✓ Index html generation complete.

Initial Chunk Files | Names | Raw Size | Estimated Transfer Size
main.3d383f0731225c7f.js | main | 125.57 kB | 35.95 kB
polyfills.9fa5637fd05770ca.js | polyfills | 33.07 kB | 10.63 kB
runtime.4f861b672ed0f170.js | runtime | 1.04 kB | 596 bytes
styles.28ee1965fb724f31.css | styles | 648 bytes | 243 bytes
| Initial Total | 160.31 kB | 47.39 kB

Build at: 2022-07-21T14:16:51.739Z - Hash: 56dc8a2451dd7dae - Time: 6348ms
C:\WEB_workspace\FRONTEND\TODOapp>
```

Os arquivos finais da construção o seu projeto vão estar disponíveis dentro da pasta "**C:\WEB\_workspace\FRONTEND\TODOapp\dist\todoapp**". Ex:



Esses são os arquivos que deveremos hospedar no servidor do Frontend.



### 9.10. Modificando o local base do App

Considerando que o Angular prepara o App para rodar na pasta raiz de um site, vamos fazer uma pequena mudança para que possa ser colocado em uma sub-pasta. Para isso, edite o arquivo "**index.html**" modificando o elemento:

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

Por:

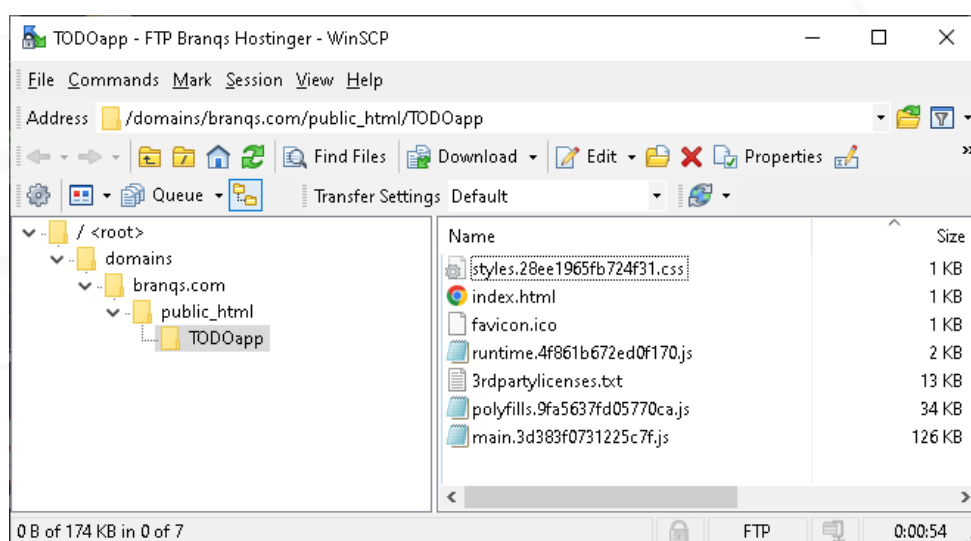
```
<base href="./">
```

Ou seja, basta inserir um ponto antes da barra.

### 9.11. Transferindo para o Hostinger

Diferentemente do Backend, o Frontend não necessita de uma configuração especial de recursos, bastando realizar uma simples transferência de arquivos para o servidor.

Os diversos servidores do mercado oferecem diferentes métodos para transferência desses arquivos. Em nosso exemplo, vou estabelecer a conexão FTP com uma conta que possuo no Hostinger para hospedar o domínio "branqs.com". Em seguida vou criar uma sub-pasta do domínio chamada de "TODOapp" e arrastar para dentro dela os arquivos criados durante o build do App. Ex:



Pronto, neste exemplo, nosso App já está pronto para uso a partir da URL:

<http://branqs.com/TODOapp/>

### 9.12. Considerações finais

Se você chegou até aqui... PARABÉNS !!!

Você conseguiu publicar uma solução WEB contemplando FRONTEND, BACKEND e BANCO DE DADOS, ou seja, o famoso FULL STACK.

Seu App agora pode ser executado rapidamente a partir do browser de qualquer tipo de dispositivo como: Computadores, Smartphones, Tablets etc. Tudo isso sem necessitar nem mesmo de instalação prévia.

Espero muito que você tenha gostado do conteúdo e que sirva de inspiração para você pesquisar detalhes de funcionamento e melhorias que podem ser implementadas, como por exemplo Autenticação de usuários e Autorizações de Acesso.

Muito obrigado por todos os momentos em que passamos juntos nesta matéria e parabéns por você se dedicar aos estudos para obter aprimoramento profissional. São pessoas assim que ajudam a criar um mundo cada vez melhor.

SUCESSO !!!!!!!!!!!