

[Get started](#)[Open in app](#)

Fernando Evangelista

37 Followers · About [Follow](#)

Consumindo API REST com HttpClient no Angular 8



[Fernando Evangelista](#) Oct 19, 2019 · 7 min read



Neste tutorial passo a passo vamos aprender a consumir API REST no angular 8 usando o serviço HttpClient

Requisitos

Para exemplificar o uso do `HttpClient` vamos criar um projeto angular, o projeto será um CRUD para nos ajudar de forma simples e básica a manipular carros, também criaremos uma API REST fake, essa api fake vai simular nosso back-end, assim focaremos no uso do `HttpClient`, então para isso vamos precisar de algumas tecnologias

- Angular CLI (v8.15.0)
- Node.js (v10.9.0)
- Json-server

Angular 8 HttpClient

O `HttpClient` é usado para fazer a comunicação entre cliente e servidor usando o protocolo HTTP. Ou seja, se você está querendo consumir dados de uma API externa o `HttpClient` facilitará essa comunicação, através de muitos métodos disponíveis:

`post()`

`get()`

`put()`

`delete()`

`patch()`

`request()`

`head()`

`jsonp()`

`options()`

Benefícios do HttpClient

O `HttpClient` usa a interface `XMLHttpRequest` que também suporta a navegadores antigos, além de fácil de usar disponibiliza benefícios, como:

Solicitações de request e response interceptadas

Manipulação de erros simplificada

Suporte a api Observable

APIs e tratamentos de erros

Instalando o Angular CLI

O projeto será criado usando o `angular CLI`, então para seguirmos com o tutorial, temos que tê-lo instalado em nossa máquina. Levando em consideração que temos o `Node.js` instalado, basta executar o seguinte comando:

```
sudo npm install -g @angular/cli
```

Criando projeto angular 8

Com o `angular CLI` e o `Node.js` instalado, vamos criar o nosso projeto angular usando o CLI, para isso execute o comando abaixo no seu terminal:

```
ng new angular-http
```

Após executar o comando acima, o `angular CLI` fará algumas perguntas, na primeira digite "y" para criar o projeto com rota, em seguida escolha a opção CSS como formato de folha de estilo, isso criará o nosso projeto com os módulos NPM necessários.

Rodando o projeto angular 8

Após criar o projeto, precisamos rodar e verificar se tudo foi criado corretamente, então entre na pasta do projeto e digite:

```
ng serve --open
```

O comando acima rodará o nosso projeto, e o parâmetro `--open` abrirá automaticamente o browser com o nosso projeto angular 8 em execução, como na imagem abaixo.



Tela do angular 8 em execução

Criando uma API REST fake

Para simular o uso do `HttpClient`, precisamos de uma API REST, como o foco é o `HttpClient` não vamos nos preocupar em criar uma API REST, para isso podemos usar o `json-server`, que faz uma API REST fake, assim focaremos no `HttpClient`. Para mais detalhes sobre o `json-server`, podemos consultar seu [github](#).

Para instalar o `json-server`, basta executar o seguinte comando:

```
sudo npm install -g json-server
```

Dentro do nosso projeto, vamos criar uma pasta chamada “data” dentro de “assets”

```
/src/assets/data
```

Agora crie um arquivo chamado `db.json` e jogue dentro da pasta “data” que acabamos de criar:

```
/src/assets/data/db.json
```

Vamos abrir o arquivo `db.json` e incluir o seguinte json:

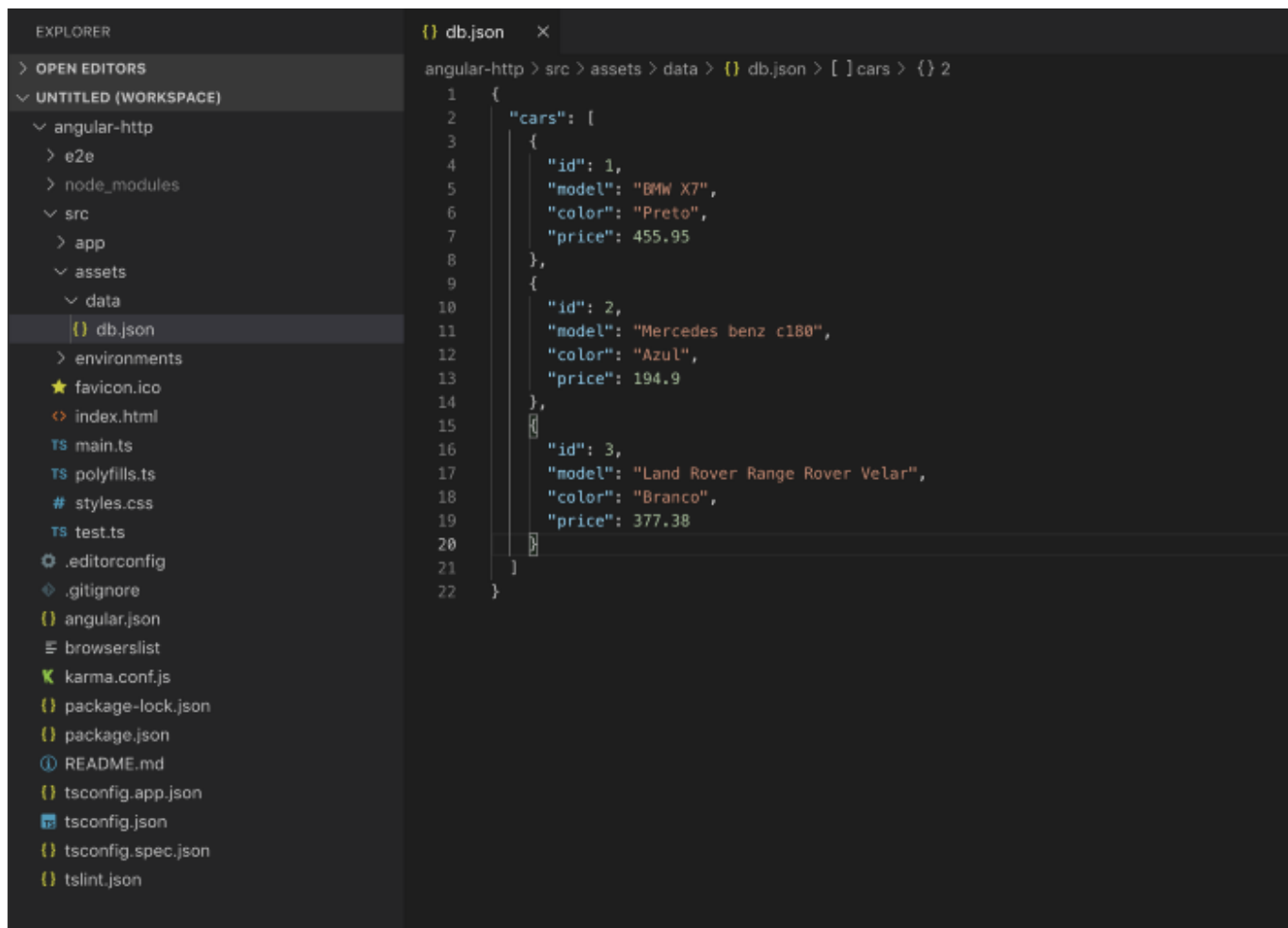
```
1  {  
2    "cars": [  
3      {  
4        "id": 1,  
5        "model": "BMW X7",  
6        "color": "Preto",  
7        "price": 455.95
```

```
8      },
9      {
10     "id": 2,
11     "model": "Mercedes benz c180",
12     "color": "Azul",
13     "price": 194.9
14   },
15   {
16     "id": 3,
17     "model": "Land Rover Range Rover Velar",
18     "color": "Branco",
19     "price": 377.38
20   }
21 ]
22 }
```

db.json hosted with  by GitHub[view raw](#)

Arquivo db.json para o json-server

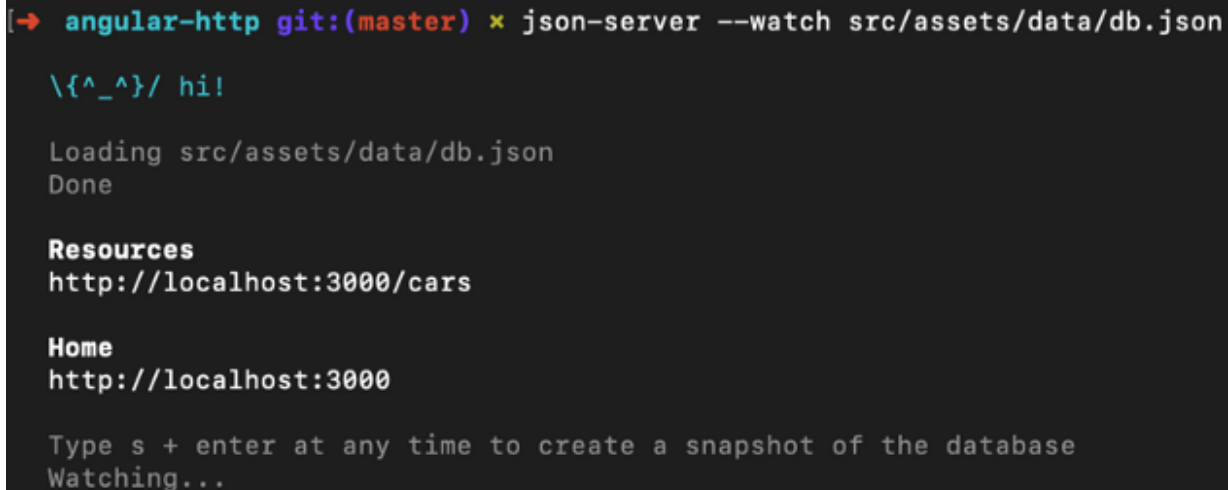
Nossa estrutura de pastas e o arquivo `db.json` ficará assim:



Estrutura de pastas com o arquivo db.json

Vamos rodar o `json-server` para simular nossa API REST, abra um novo terminal e na raiz do projeto execute o seguinte comando:

```
json-server --watch src/assets/data/db.json
```



```
[→ angular-http git:(master) ✕ json-server --watch src/assets/data/db.json  
  
\\{^_^}/ hi!  
  
Loading src/assets/data/db.json  
Done  
  
Resources  
http://localhost:3000/cars  
  
Home  
http://localhost:3000  
  
Type s + enter at any time to create a snapshot of the database  
Watching...
```

json-server em execução

Tudo funcionará como na imagem acima, observe que nossa API REST fake está exposta no endereço: <http://localhost:3000>

Configurando o HttpClient

Para usar o `HttpClient`, precisamos adicionar o modulo `HttpClientModule` no arquivo

```
app.module.ts.
```

Para fazer isso, vamos abrir o arquivo `app.module.ts`

```
src/app/app.module.ts
```

Dentro de imports do decorator `@NgModule`, adicione o modulo `HttpClientModule`

Vamos nos atentar para não usar o pacote `@angular/http`, esse pacote estava depreciado desde a versão 5 do angular, no angular 8 ele foi removido, o pacote que utilizaremos é

@angular/common/http.

Vamos aproveitar e adicionar o modulo `FormsModule`, esse modulo nos ajudará em nosso formulário para nossa tela de exemplo, porém não é necessário para o uso do `HttpClient`. O resultado do arquivo `app.module.ts` será:

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4  import { AppRoutingModule } from './app-routing.module';
5  import { AppComponent } from './app.component';
6  import { HttpClientModule } from '@angular/common/http';
7  import { FormsModule } from '@angular/forms';
8
9  @NgModule({
10   declarations: [
11     AppComponent
12   ],
13   imports: [
14     BrowserModule,
15     AppRoutingModule,
16     HttpClientModule,
17     FormsModule
18   ],
19   providers: [],
20   bootstrap: [AppComponent]
21 })
22 export class AppModule { }
```

app.module.ts hosted with ❤ by GitHub

[view raw](#)

Arquivo app.module.ts

Criando o model

Vamos criar uma interface de modelo para os dados dos carros. Na raiz do projeto vamos executando o seguinte comando:

```
ng g interface models/car
```

O parâmetro `g` é uma abreviação de `generate`.

Será criado 2 arquivos dentro de uma pasta com o nome `models`, o arquivo `car.spec.ts` é um arquivo de teste, o segundo é o `car.ts` o model que representará nosso carro.

Para não criar o arquivo de teste, é só usar o parâmetro `--skipTests=true`

Agora vamos adicionar o seguinte conteúdo dentro de nosso model `car.ts`

```
1  export interface Car {
2      id: number;
3      model: string;
4      color: string;
5      price: number;
6  }
```

car.ts hosted with ❤ by GitHub

[view raw](#)

Arquivo `car.ts`

Criando o serviço responsável pelas requisições http

Vamos criar os métodos responsáveis pelas requisições http que faremos no `json-server` usando o `HttpClient`, mas antes devemos criar um arquivo service, no angular é recomendado criar services para os métodos que faz chamadas http.

Para criar o serviço, digite o seguinte comando:

```
ng g service services/car
```

O comando acima, criará um arquivo com o nome `car.services.ts` dentro da pasta `services`. Após criar nosso service, já podemos começar a utilizar o `HttpClient`

Usando o HttpClient

Dentro do nosso service `car.services.ts`, vamos criar alguns métodos http utilizando o `HttpClient`.

Abra o arquivo `car.services.ts` dentro da pasta `services` e inclua o seguinte código:

```
1  import { Injectable } from '@angular/core';
2  import { HttpClient, HttpResponse, HttpHeaders } from '@angular/common/http';
3  import { Observable, throwError } from 'rxjs';
```



```
4 import { retry, catchError } from 'rxjs/operators';
5 import { Car } from '../models/car';
6
7 @Injectable({
8   providedIn: 'root'
9 })
10 export class CarService {
11
12   url = 'http://localhost:3000/cars'; // api rest fake
13
14   // injetando o HttpClient
15   constructor(private httpClient: HttpClient) { }
16
17   // Headers
18   httpOptions = {
19     headers: new HttpHeaders({ 'Content-Type': 'application/json' })
20   }
21
22   // Obtem todos os carros
23   getCars(): Observable<Car[]> {
24     return this.httpClient.get<Car[]>(this.url)
25       .pipe(
26         retry(2),
27         catchError(this.handleError))
28   }
29
30   // Obtem um carro pelo id
31   getCarById(id: number): Observable<Car> {
32     return this.httpClient.get<Car>(this.url + '/' + id)
33       .pipe(
34         retry(2),
35         catchError(this.handleError)
36       )
37   }
38
39   // salva um carro
40   saveCar(car: Car): Observable<Car> {
41     return this.httpClient.post<Car>(this.url, JSON.stringify(car), this.httpOptions)
42       .pipe(
43         retry(2),
44         catchError(this.handleError)
45       )
46   }
47 }
```

```
48 // atualiza um carro
49 updateCar(car: Car): Observable<Car> {
50     return this.httpClient.put<Car>(this.url + '/' + car.id, JSON.stringify(car), this.httpOptions)
51         .pipe(
52             retry(1),
53             catchError(this.handleError)
54         )
55 }
56
57 // deleta um carro
58 deleteCar(car: Car) {
59     return this.httpClient.delete<Car>(this.url + '/' + car.id, this.httpOptions)
60         .pipe(
61             retry(1),
62             catchError(this.handleError)
63         )
64 }
65
66 // Manipulação de erros
67 handleError(error: HttpResponse) {
68     let errorMessage = '';
69     if (error.error instanceof ErrorEvent) {
70         // Erro ocorreu no lado do client
71         errorMessage = error.error.message;
72     } else {
73         // Erro ocorreu no lado do servidor
74         errorMessage = `Código do erro: ${error.status}, ` + `mensagem: ${error.message}`;
75     }
76     console.log(errorMessage);
77     return throwError(errorMessage);
78 };
79
80 }
```

car.service.ts hosted with ❤ by GitHub

[view raw](#)

Arquivo car.service.ts

Entendendo nosso serviço

Injetamos o `HttpClient` e atribuímos a uma variável chamada `httpClient`, observe como é simples usar os métodos `get`, `post`, `put` e `delete` do nosso `httpClient`:

```
1 this.httpClient.get<Car[]>(this.url)
```

```
2 this.httpClient.post<Car>(this.url, JSON.stringify(car), this.httpOptions)
3 this.httpClient.put<Car>(this.url + '/' + car.id, JSON.stringify(car), this.httpOptions)
4 this.httpClient.delete<Car>(this.url + '/' + car.id, this.httpOptions)
```

car.services.ts hosted with ❤ by GitHub

[view raw](#)

Chamando métodos http com httpClient

Note que passamos uma variável chamada “**url**” essa conterá o endereço <http://localhost:3000/cars> disponibilizada pela API REST fake do `json-server`.

Adicionamos em nosso `HttpClient` um objeto do tipo `HttpOptions`, contendo nosso header através da classe `HttpHeaders`.

Alguns servidores podem exigir cabeçalhos para algumas operações como, post, put e delete, por exemplo um back-end onde as requisições dependem de um token de autorização.

Erros podem acontecer, por exemplo, um servidor back-end pode rejeitar nossa solicitação, ou um erro de rede no lado do cliente, esses erros produzem um objeto javascript do tipo `ErrorEvent`. Para manipularmos esses erros, criamos um método dentro do nosso serviço chamado `handleError`, esse método produzirá um `RxJS ErrorObservable`, caso nossas requisições contenha um erro, redirecionaremos para nossos serviços com uma mensagem mais amigável.

O angular recomenda que métodos de manipulação de erros deve ser nos services e não nos componentes.

Antes de chamar o método `handleError`, observe que adicionamos um operador chamado `retry`, esse é o mais simples da biblioteca `RxJS`, ele reexecutará a chamada em um numero específico de vezes caso aconteça um erro.

Após entendemos o uso do `httpClient`, concluímos que nosso serviço é responsável pelas seguintes ações:

- `getCars()` : recupera todos os carros.
- `getCarById()` : recupera um carro especifico pelo id.
- `saveCar()` : salva um carro.

- `updateCar()` : atualiza um carro.
- `deleteCar()` : exclui um carro específico pelo id.

Usando nosso serviço

Agora vamos chamar nosso serviço através do component `app.component.ts`. Para isso vamos editar o component `app.component.ts` e adicionar o seguinte conteúdo:

Arquivo `app.component.ts`

Na linha 16 injetamos via construtor nosso serviço `CarService` e atribuímos a uma variável chamada `carService`

Criamos um método `getCars()`, esse será responsável por chamar a listagem de carros disponíveis através do nosso serviço `CarService`, incluímos no `onOnInit()` para quando acessarmos `app.component.ts` ele nos trazer todos os carros, observe que logo após o método `getCars()` do `carService`, usamos o `subscribe`

```
getCars() {  
  this.carService.getCars().subscribe((cars: Car[]) => {  
    this.cars = cars;  
  });  
}
```

O `subscribe` é um dos operadores mais importantes do `Observable` da biblioteca `RxJS`, ele notificará assim que a resposta vier e for transformada em `Json`, nos retornando um array de Carros.

No nosso exemplo, não esperamos retorno nas chamadas aos métodos `saveCar()`, `updateCar()` e `deleteCar()`, sendo assim, quando entrar no `subscribe` estamos limpando nosso formulário e consultando novamente o método `getCars` para listar todos os carros.

Testando a aplicação

Modifiquei o HTML e CSS para ficar parecido com um projeto real, apenas para mostrar o uso do `HttpClient` em ação de forma simples e visual através de interações de tela. Para isso, vamos incluir alguns trechos de código em nossa aplicação, primeiro vamos abrir o arquivo `index.html` e adicionamos o link CSS externo do `bootstrap` :

```
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">
```

O código ficará:

Arquivo `index.html`

Agora vamos apagar todo o conteúdo do arquivo `app.component.html` e inserir o seguinte código:

No arquivo `app.component.css` vamos adicionar o seguinte CSS:

Agora teremos nossa aplicação como a imagem abaixo:



Tela da aplicação

Já podemos notar o funcionamento do `HttpClient` através da listagem de carros que está no arquivo `db.json`. Assim que acessamos a tela da aplicação o método `onOnInit()` foi disparado chamado o método `getCars()` que chama a listagem de carros do nosso serviço `CarService`.

Vamos adicionar um novo carro, uma Lamborghini Aventador



Adicionando um novo carro

Se abrir o terminal do `json-server` podemos verificar que requisições do tipo GET e POST foram realizadas:

```
POST /cars 201 45.079 ms - 97
GET /cars 200 4.980 ms - 407
```

A primeira foi uma requisição do tipo GET que nos retornou uma listagem contendo os carros, essa foi executada assim que acessamos a tela, ela foi chamada através do método `getCars()`, dentro do `onOnInit()`, a segunda requisição foi do tipo POST, realizada através do método `saveCars()`, invocada após clicar no botão “*Salvar*”.

Se editarmos o modelo do carro para Lamborghini Huracan e sua cor para azul, podemos notar a requisição do tipo PUT e logo em seguida outra do tipo GET que é a listagem dos carros

```
PUT /cars/4 200 5.340 ms - 99
GET /cars 200 3.185 ms - 409
```

E se excluirmos nossa Lamborghini Huracan, notamos outras requisições do tipo DELETE e outra do tipo GET .

```
DELETE /cars/4 200 5.155 ms - 2
GET /cars 200 5.871 ms - 296
```

Conclusão

Através de uma aplicação modelo mostramos como consumir uma API REST usando o `HttpClient` , além de aprendemos como fazer requisições HTTP com os métodos GET, POST, PUT E DELETE, também aprendemos de forma básica como manipular erros e usar uma API REST fake com o `json-server` .

O código fonte do projeto está disponível no meu [github](#)

[Angular8](#)[Angular HttpClient](#)[Angular Cli](#)[Json Server](#)[Restful Api](#)[About](#) [Help](#) [Legal](#)

Get the Medium app



