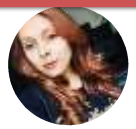
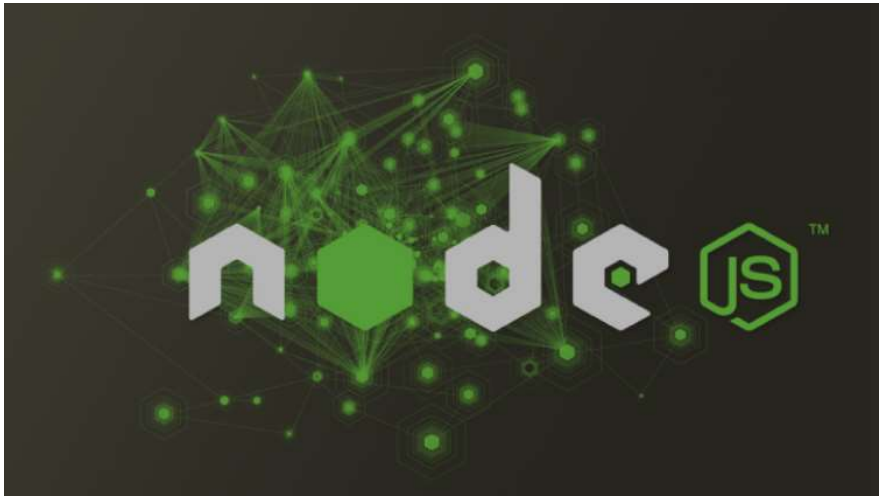


Jakelliny Gracielly Follow
Development Leader, Speaker, Decoding the Matrix, Nordic Culture, Community Manager at WordPress São Paulo, NerdZão and FC Tech
Sep 20 · 8 min read

Crie uma API RESTful em NodeJS para gerenciar seus crushs



1. Neste tutorial vamos usar uma API RESTful em NodeJS para criar um gerenciador (CRUD) de Crushs. Ao concluir, vamos conseguir

inserir um novo Crush com nome, apelido, whatsapp, observações, foto e uma nota de 0 a 5, editá-lo, deletá-lo e listar todos os contatinhos que inserirmos. Vamos lá então!

. . .

Repositório com o projeto criado: http://bit.ly/2D4gbJ

O que vamos usar?

- NodeJS
- NPM
- TypeScript
- MongoDB
- Postman

Dentro do terminal do NodeJS vamos iniciar nosso projeto criando uma pasta com o nome: **crush-management** e vamos entrar nessa pasta com o comando **cd**

```
mkdir crush-management && cd crush-management
```

Vamos iniciar o projeto com o comando:

```
npm init
```

Irão aparecer algumas perguntas:

name: (crush-management)
version: (1.0.0)
description:
entry point: (index.js) **server.ts**
test command:
git repository:
keywords:
author: Jakeliny Gracielly
license: (ISC)

1. *Os valores que aparecem entre parenteses são os valores default, se nada for digitado o valor entre parenteses será preenchido*
2. *na pergunta entry point, vamos colocar **server.ts** será nosso arquivo de partida*
3. *em author vamos colocar nosso nome*

Vamos instalar os pacotes necessários para nossa API funcionar:

```
npm install body-parser express http-status mongoose morgan
nodemon
```

Esse comando vai instalar as dependências do projeto e já gravar no package.json, não é necessário colocar -S ou --save, a partir da versão 5.0.0 do NodeJS os módulos são gravados automaticamente

Vamos instalar os pacotes necessários para desenvolvimento:

```
npm install @types/body-parser @types/express @types/http-
status @types/mongoose @types/morgan @types/node ts-node
typescript -D
```

O -D no final do comando é igual a --only=dev, que salva os pacotes no package.json como dependência para desenvolvimento

Abra sua IDE favorita e abra o arquivo que foi criado **package.json** observe que os dados inseridos nas perguntas e os pacotes instalados estão todos referenciados.

Agora vamos configurar o compilador do typescript, crie um arquivo na raiz da pasta do nosso projeto com o nome tsconfig.json e adicione o conteúdo:

```
{
"compilerOptions":{
"target": "es5",
"module": "commonjs",
"outDir": "build",
"typeRoots": [
"../node_modules/@types"
],
"types": [
"node"
]
},
"include": [
"server/**/*.ts",
"server/*.ts"
],
```

```
"exclude": [
"node_modules"
],
"compileOnSave": true,
"buildOnSave": true
}
```

Nesse arquivo estamos:

- *colocando as opções de compilação do Type Script, queremos que ele transcreva todo o conteúdo para es5 usando o módulo commonjs e passe tudo para a pasta build.*
- *incluímos os arquivos da nossa futura pasta server e os arquivos de suas subpastas*
- *deixamos de fora da compilação os arquivos da node_modules*
- *e por último dizemos que quando salvarmos algum arquivo .ts o typescript ja compila e "builda" esses arquivos*

Vamos criar a pasta server

```
mkdir server
```

Vamos criar os arquivos app.ts e server.ts dentro da pasta server

```
touch server/{app,server}.ts
```

Vamos abrir o arquivo app.ts em nossa IDE e colocar o conteúdo

```
import * as express from 'express';
import * as morgan from 'morgan';
import * as bodyParser from 'body-parser';

class App{

public app: express.Application;
private morgan: morgan.Morgan;
private bodyParser;

constructor(){
  this.app = express();
  this.middleware();
  this.routes();
}

middleware(){
  this.app.use(morgan('dev'));
```

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

  ,routes(){
    this.app.route('/').get((req, res) =>
  res.status(200).json({ 'message': 'Hello world!' }));
  }
  }

  export default new App();
```

Agora no arquivo server.ts e configurar para nossa aplicação rodar na porta 4200

```
import App from './app';

App.app.listen(4200, () => console.log('servidor rodando,
porta: 4200'));
```

Agora nosso servidor está pronto, vamos fazer uma pequena configuração no **package.json**, vamos localizar a linha onde está o comando script: {…} e adicionamos dentro da chave comando “start”

```
"scripts": {
"test": "echo \"Error: no test specified\" && exit 1",
```

```
"start": "NODE_ENV=development nodemon
./node_modules/.bin/ts-node ./server/server.ts"
},
```

Agora no terminal do Node dentro da pasta do projeto vamos rodar o comando:

```
npm start
```

Se não der nenhum erro, abra o endereço http://localhost:4200 no seu navegador preferido, você deve conseguir ver o resultado a seguir:

UHUUU, nosso serve está respondendo e rodando sem erros, e agora qual o próximo passo?

Conectar com o MongoDB

Vamos criar uma pasta com o nome config dentro da pasta server:

```
mkdir /server/config
```

E um arquivo com o nome db.ts

```
touch /server/config/db.ts
```

Dentro desse arquivo vamos colocar os seguintes comandos:

```
import * as mongoose from 'mongoose';

class DataBase {

private DB_URI = 'mongodb://127.0.0.1/crush-manegement';

private DB_CONNECTION;

constructor() { }

createConnection() {

mongoose.connect(this.DB_URI);

this.logger(this.DB_URI);
```

```
}


closeConnection(message, callback) {

this.DB_CONNECTION.close(() => {

console.log('Mongoose foi desconectado pelo: ' + message);

callback();

})

}

logger(uri) {

this.DB_CONNECTION = mongoose.connection;

this.DB_CONNECTION.on('connected', () =>
console.log('Mongoose está conectado ao ' + uri));

this.DB_CONNECTION.on('error', error =>
console.error.bind(console, "Erro na conexão: " + error));

this.DB_CONNECTION.on('disconnected', () =>
console.log("Mongoose está desconectado do " + uri));

}

}

export default DataBases;
```

Com o nosso arquivo de conexão feito vamos chama-lo no app.ts, primeiro importamos o arquivo

```
import DataBase from './config/db';
```

Criamos o atributo database do tipo DataBase

```
private database: DataBase;
```

No método construtor instanciamos a classe DataBase

```
this.database = new DataBase();
```

Vamos criar um método de conexão

```
dataBaseConnection(){
this.database.createConnection();
}
```

Vamos criar um método para fechar a classe

```
closeDataBaseConnection(message, callback){
this.database.closeConnection(message, () => callback());
}
```

Por ultimo vamos chamar no método construtor o método que abre nossa conexão para quando a nossa API rodar uma das primeiras coisas que será feita é a abertura da conexão

```
this.dataBaseConnection();
```

Agora vamos no arquivo server.ts e vamos adicionar dois comandos que serão responsáveis por fechar nossa conexão com o banco de dados caso o processo da nossa API seja terminado inesperadamente, como por exemplo, quando executamos ctrl+C no terminal

```
process.once('SIGUSR2', () =>
App.closeDataBaseConnection('nodemon restart', ()=>
process.kill(process.pid, 'SIGUSR2')));


process.on('SIGINT', () =>
App.closeDataBaseConnection('execução foi interrompida',
```

```
()=> process.exit(0)));
```

E pronto, terminamos de escrever os códigos necessários para fazer a conexão com o MongoDB, vamos testar?

Primeiro certifique-se que o mongo está rodando na sua maquina, se você está no ubuntu pode rodar no seu terminal o comando:

```
mongo
```

se der algum erro ou aviso, certifique-se que o MongoDB está instalado e rode o seguinte comando para iniciar o serviço do MongoDB:

```
sudo service mongod start
```

E agora podemos testar nosso projeto

```
npm start
```

Dentro do arquivo db.ts escrevemos algumas mensagens para determinados estados da nossa conexão, se foi bem sucedida deve aparecer a mensagem “Mongoose está conectado ao [link do seu mongo]”

Vamos gerenciar nossos Crushs

Agora que ja temos nosso web service funcionando e conectado ao nosso banco de dados vamos começar a escrever o módulo “Crush”

Vamos começar criando a seguinte estrutura de pasta

```
mkdir server/modules/crush
```

Dentro da pasta crush vamos criar os arquivos controller.ts, repository.ts, routes.ts e schema.ts

```
touch {controller,repository,routes,schema}.ts
```

Vamos começar pelo arquivo schema, onde vamos escrever os campos e tipos de campos que teremos para guardar as informações do nosso Crush

```
import * as mongoose from 'mongoose';

const CrushSchema = new mongoose.Schema({

nome: {type: String, required: true},
apelido: {type: String, unique: true, required: true},
whatsapp: {type: String, unique: true, required: true},
observacoes: {type: String, required: true},
foto: {type: String, required: true},
nota: {type: String, required: true},
createdAt: {type: Date, default: Date.now}
});

export default CrushSchema
```

Próximo passo é ir no nosso arquivo repository onde vamos passar nosso schema para o mongoose, sempre que quisermos executar algo com o mongo o mongoose tem a função de verificar se essa "tabela" existe, se não existir ele cria pra nós e a usa em seguida sem que o usuário perceba o que está acontecendo

```
import * as mongoose from 'mongoose';

import CrushSchema from './schema';

export default mongoose.model('Crush', CrushSchema);
```

Agora vamos ao nosso arquivo controller onde vamos escrever nossos metodos de crud usando as função do mongoose, por exemplo find, que serão importadas nesse arquivo a partir da importação do repository

```
import Crush from ‘./repository’;


class CrushController{

constructor(){}

getAll(){
  return Crush.find({});
}

getByID(id){
  return Crush.findById(id);
}

create(crush){
  return Crush.create(crush);
}

update(id, crush){
  const updateCrush = {
    nome: crush.nome,
    apelido: crush.apelido,
    whatsapp: crush.whatsapp,
    observacoes: crush.observacoes,
    foto: crush.foto,
    nota: crush.nota
  }
```

```
    return Crush.findByIdAndUpdate(id, updateCrush);
  }

  delete(id){
    return Crush.remove(id);
  }


  }

  export default new CrushController();
```

Com o nosso CRUD feito vamos em rotas, aqui o objetivo é criar as chamas para acessar os metodos de CRUD do arquivo controller

```
import CrushController from './controller';
import * as httpStatus from 'http-status';

const sendResponse = function(res, statusCode, data){
  res.status(statusCode).json({'result' : data});
}

class CrushRoutes{

  constructor(){}

  getAll(req, res){
    CrushController
    .getAll()
    .then(crushs => sendResponse(res, httpStatus.OK,
crushs))
    .catch(err => console.error.bind(console, 'Erro: '+
```

```
err));
  }

  getByID(req, res){
    const id = {_id: req.params.id}
    if(!id){
      sendResponse(res, httpStatus.OK, 'Crush não
encontrado');
    }
    CrushController
    .getByID(id)
    .then(crush => sendResponse(res, httpStatus.OK, crush))
    .catch(err => console.error.bind(console, 'Erro: '+
err));
  }

  create(req, res){

    const crush = req.body;

    CrushController
    .create(crush)
    .then(crush => sendResponse(res, httpStatus.CREATED,
"Crush criado com amor!"))
    .catch(err => console.error.bind(console, 'Erro: '+
err));
  }

  update(req, res){

    const id = {_id: req.params.id}
    const crush = req.body;

    CrushController
    .update(id, crush)
    .then(crush => sendResponse(res, httpStatus.OK, "Crush
alterado!"))
    .catch(err => console.error.bind(console, 'Erro: '+
```

```
err));
  }

  delete(req, res){

    const id = {_id: req.params.id}

    CrushController
    .delete(id)
    .then(result => sendResponse(res, httpStatus.OK,
result))
    .catch(err => console.error.bind(console, 'Erro: '+
err));
  }

}

export default new CrushRoutes();
```

E pronto! escrevemos todo nosso módulo para gerenciar nosso Crushs, agora vamos chamar tudo isso no arquivo app.ts, começamos importando o arquivo routes.ts que é o responsável por chamar toda a estrutura do módulo

```
import CrushRoutes from './modules/user/routes';
```

No método rotas onde chamamos o "/" que é a nossa url base, vamos criar novas rotas para chamar cada função da nossa API que são:

- Listar todos os nossos Crushs
- Listar um Crush em especifico
- Editar um Crush
- Apagar umCrush

Vamos criar as rotas

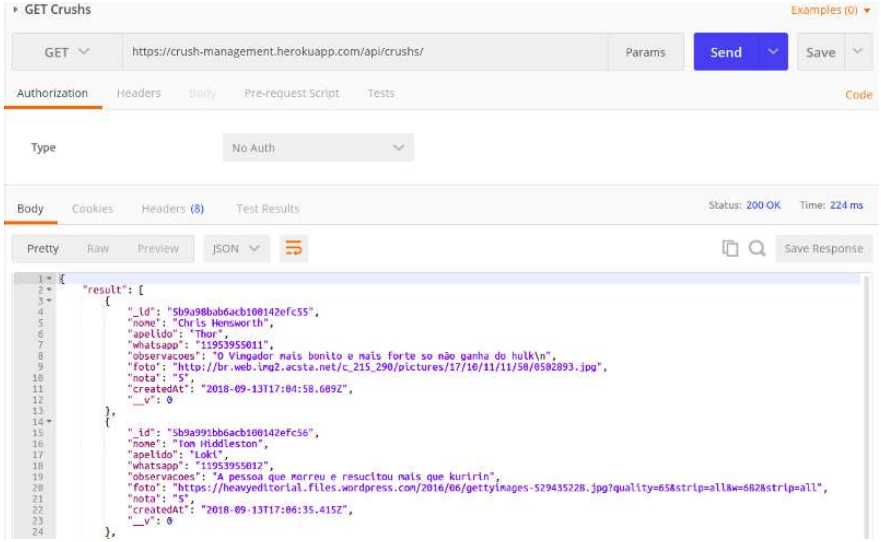
```
this.app.route('/api/crushs').get(CrushRoutes.getAll);
this.app.route('/api/crushs/:id').get(CrushRoutes.getByID);
this.app.route('/api/crushs').post(CrushRoutes.create);
this.app.route('/api/crushs/:id').put(CrushRoutes.update);
this.app.route('/api/crushs/:id').delete(CrushRoutes.delete)
;
```

Se eu digitar agora no meu navegador http://localhost:5000/api/crushs vai aparecer a lista (em json) de todos os meus Crushs e suas informações.

Como podemos testar nossa API?

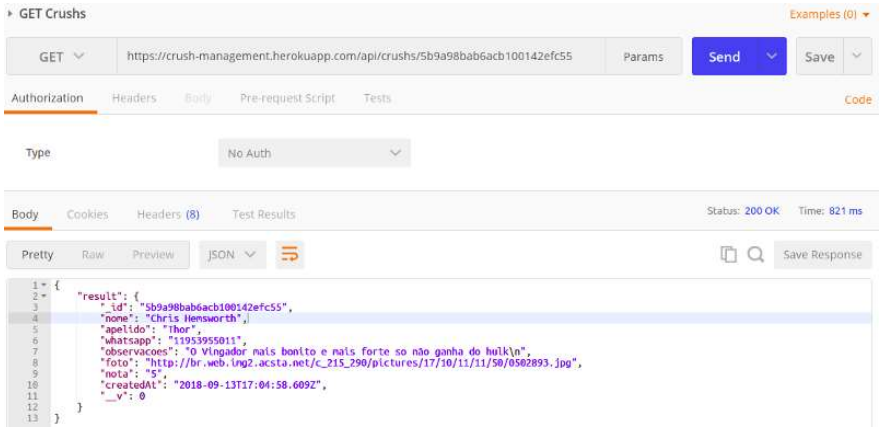
Vamos usar um programa para nos auxiliar nessa jornada, o Postman —https://www.getpostman.com/

Vou usar como exemplo a API que hospedei no heroku, link: https://crush-management.herokuapp.com/



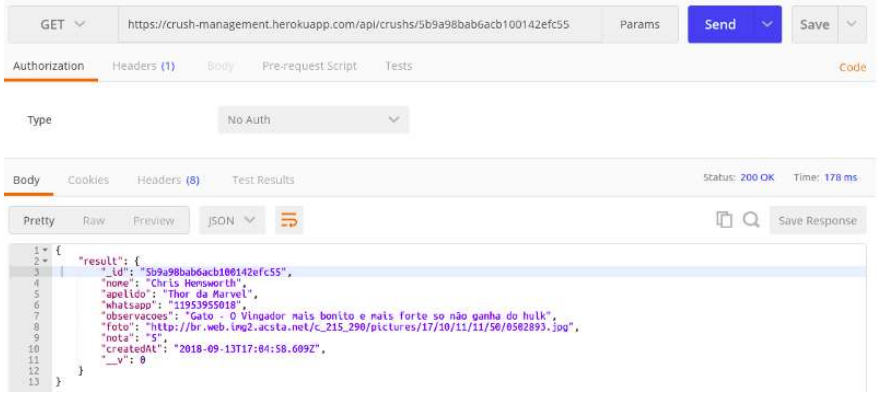
Eu coloco a rota de chamar todos os meus Chrushs e seleciono o metodo GET assim como está programado em nossas rotas

Vamos testar chamando um Crush em espefico passando o id 5b9a98bab6acb100142efc55 depois de crushs/ mantendo o metodo GET



Vamos testar o update, para isso temos que trocar o metodo para PUT e abrir a aba BODY passando os dados no form que surge ali no formato x-www-form-urlencoded

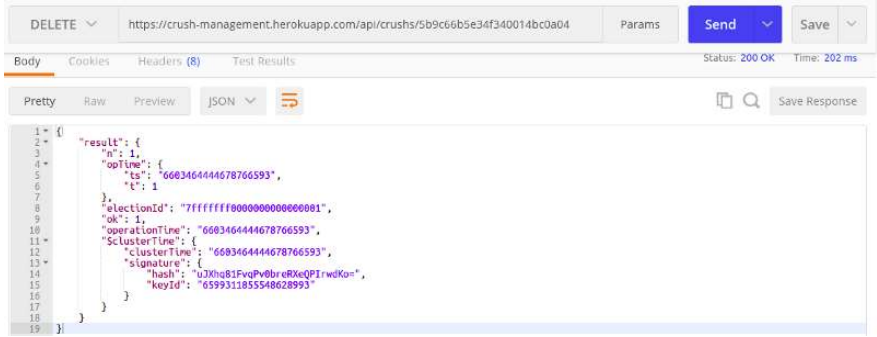
Mudei o “apelido”, “whatsapp” e o campo “observacoes”, depois de enviar vou listar novamente meu Crush no metodo GET e verificar se alterações foram feitas:



Vamos testar o método de excluir, vou usar o id 5b9c66b5e34f340014bc0a04 que é do Alisson Becker:


```
{
    "_id": "5b9c66b5e34f340014bc0a04",
    "nome": "Alisson Becker",
    "apelido": "Goleiro Lindo",
    "whatsapp": "11953955019",
    "observacoes": "Como goleiro é ruim, mas deixou essa copa maravilhosa",
    "foto": "http://pm1.narvii.com/6875/ea0632d285d5fc15ff89855a30eb2f45d2657318r1-594-396v2_00.jpg",
    "nota": "4",
    "createdAt": "2018-09-15T01:56:05.670Z",
    "__v": 0
},
```


E esse nosso resultado:



Quando executarmos o GET para trazer todos os Crushs Alisson Becker não estará mais na lista.

E agora você tem um API para gerenciar todos os seus Crushs o/