

reprograma

Hora da Revisão

START

Pela última vez, May

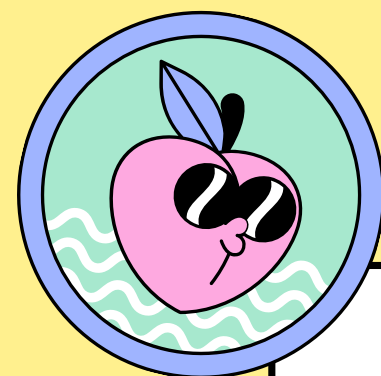


Eaaaaai, meninaaaaaaass!!!!

Eu sou a May! Sou Engenheira de Software Backend e, atualmente, trabalho (principalmente) com Golang na Wildlife Studios.

SAC da May

@mayhhara_: insta
@mayjinboo: twitter
Mayhhara Morais: linkedinho



Vamos relembrar?

01

Protocolo HTTP e Verbos

02

CRUD - Create, Read, Update e Delete

03

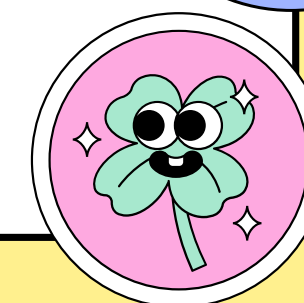
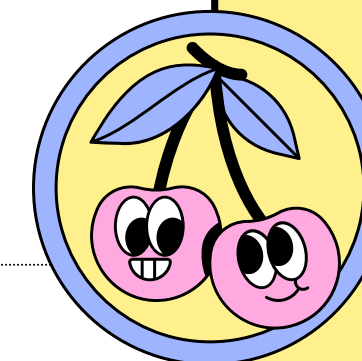
API - API Rest

04

GET, POST, PATCH, PUT, DELETE

05

Bora exercitar?



HTTP

Protocolo de Transferência de Hipertexto é um protocolo usado dentro do modelo Client/Server é baseado em pedidos (requests) e respostas (responses).

Ele é a forma em que o Cliente e o Servidor se comunicam.

Pensando em uniformizar a comunicação entre servidores e clientes foram criados **códigos** e **verbos** que são usados por ambas as partes, e essas requisições são feitas em **URLs** que possuem uma estrutura específica.



`<protocolo>://<servidor>:<porta>/<recurso>`

HTTP

Verbos

Os verbos HTTP são um conjunto de métodos de requisição responsáveis por indicar a ação a ser executada.

O **Client** manda um request solicitando um dos verbos e o **Server** deve estar preparado para receber e responde-lo com um **response**.

GET

POST

PATCH

PUT

DELETE

HTTP

Status Code

Os códigos de status das respostas HTTP indicam se uma requisição HTTP foi concluída. As respostas são agrupadas em cinco classes:

Respostas de **INFORMAÇÃO**

(100-199)

Respostas de **SUCESSO**

(200-299)

Respostas de **REDIRECIONAMENTO**

(300-399)

Erros do **CLIENTE**

(400-499)

Erros do **SERVIDOR**

(500-599)

É a desenvolvedora Back-end que coloca na construção do servidor quais serão as situações referentes a cada resposta.

CRUD

CRUD é a composição da primeira letra de quatro operações básicas de um banco de dados, e são o que a maioria das aplicações fazem.

C: Create (criar) - criar um novo registro

R: Read (ler) - exibir as informações de um registro

U: Update (atualizar) - atualizar os dados do registro

D: Delete (apagar) - apagar um registro

GET

READ

POST

CREATE

PATCH

UPDATE

PUT

UPDATE

DELETE

DELETE

ARRASOU!

API

Interface de Programação de Aplicativos

API busca criar formas e ferramentas de se usar uma funcionalidade ou uma informação sem realmente ter que "reinventar a tal função".

Ela não necessariamente está num link na Web, ela pode ser uma lib ou um framework, uma função já pronta em uma linguagem específica por exemplo.

Short words: API são instruções sobre como se comunicar com um serviço

Muitos serviços possuem API's

- YouTube possui uma API para listar vídeos, buscar, ver comentários...
- Instagram possui uma API para ver e enviar fotos...
- Uber possui uma API para chamar um motorista...
- Mercado Livre possui uma API para pesquisar produtos, fazer compras e rastrear pedidos...
- As alunas da turma On16 da {reprograma} possui uma API para pesquisar, favoritar, adicionar e alterar músicas/podcasts.

API REST - RESTFUL

REST significa Representational State Transfer. Trata-se de uma abstração da arquitetura da Web. Resumidamente, o **REST** consiste em princípios/regras/constraints que, quando seguidas, permitem a criação de um projeto com interfaces bem definidas. Desta forma, permitindo, por exemplo, que aplicações se comuniquem.

Short-words: **A API Rest é uma forma padronizada de criar API's baseada no HTTP.**

Existe uma certa confusão quanto aos termos **REST** e **RESTful**. Entretanto, ambos representam os mesmos princípios. A diferença é apenas gramatical. Em outras palavras, sistemas que utilizam os princípios **REST** são chamados de **RESTful**.

REST: conjunto de princípios de arquitetura

RESTful: capacidade de determinado sistema aplicar os princípios de REST.



Como se organiza uma API Rest?

1

2

3

Coleções de Recursos

Por exemplo: O {reprograma}fy

Na API temos uma coleção de músicas. “Musicas” é um recurso nessa API.

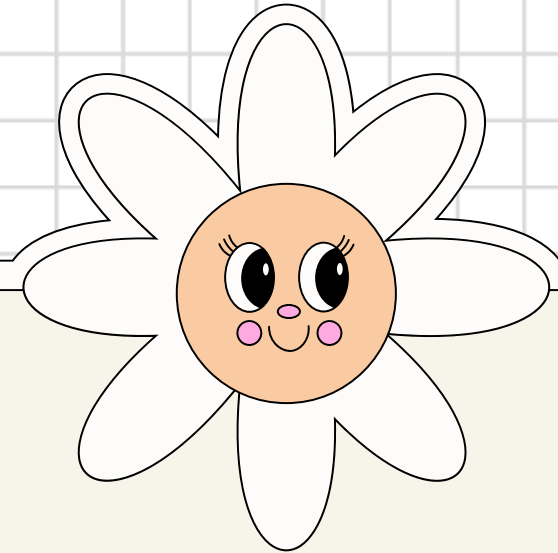
Nós também temos uma coleção de podcasts. “Podcasts” é um recurso nessa API.

Músicas

Título
Artista
Lançamento
Favorita

Podcast

Nome
Podcaster
Topico
Nota



Como se organiza uma API Rest?

1

2

3

Recursos possuem identificadores

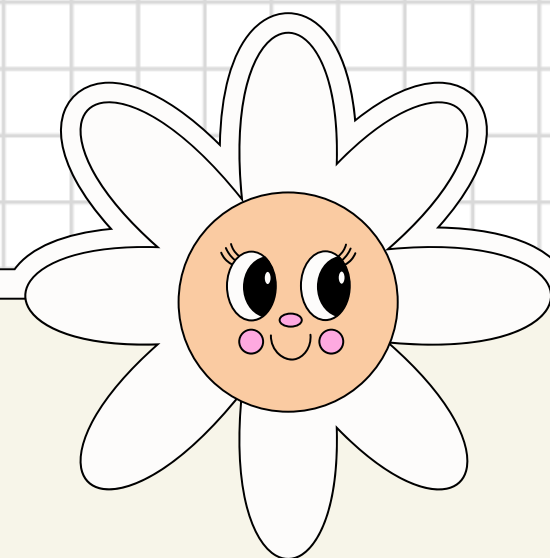
Também conhecidos como “id”.

Ex: Uma pessoa pode ser identificada pelo seu CPF.

Ex: Um produto em uma loja é identificado pelo seu código de barras.

Ex: Um carro é identificado pela sua placa.

Podem ser qualquer coisa, desde que sejam únicos e imutáveis.



Como se organiza uma API Rest?

1

2

3

Recursos representados como JSON

JSON é uma forma de representar dados em trânsito.

Suporta números, texto, objetos, listas, true/false e null.

```
{  
  "id": 2,  
  "title": "Flawless",  
  "launchYear": "2014",  
  "favorited": true,  
  "artists": [  
    "Beyonce",  
    "Nicki Minaj"  
  ]  
},
```

Tipos de Parametros na requisicao

Tanto o **body** quanto, o **query** e o **params** são parâmetros enviados na requisição e podem ser acessados pelo servidos afim de definir a requisição e as ações.

request.query * * * *

NÃO faz parte do url e é passado no formato **key=value**. Esses parâmetros devem ser definidos pela desenvolvedora da API.

Quando queremos criar **filtros para fazer consultas** na nossa aplicação, o ideal é sempre usar o **req.query**. Quero filtrar por ano? Quero filtrar por cor? Por tipo? Por diretor? Vamos usar a Query.

EX.: GET /musicas/findByArtista?artist=beyonce

request.params * * * *

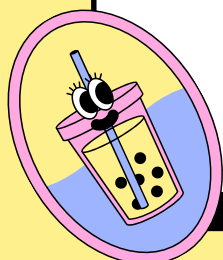
São partes variáveis de um caminho de URI. Eles são tipicamente usados para apontar para um recurso específico dentro de uma coleção. Um URL pode ter **vários parâmetros** de caminho, cada um denotado com chaves { } OU **dois pontos** . Quando quero filtrar/deletar/atualizar usando um **identificador único** (username, cpf, ID) usamos o **req.params**;

EX.: GET /musicas/:id

request.body * * * *

É usado para **enviar dados** que serão cadastrados no banco, podem ser combinados com query ou path params.

EX.: { "favorited": true }



Bora Lembrar dos verbos?

SIMBORA!

HTTP - GET & POST

GET

Usamos **GET** para ler ou recuperar um recurso. Um **GET** bem-sucedido retorna uma resposta contendo as informações solicitadas.

Em nossa **API**, uma das vezes que usamos o **GET** foi para recuperar músicas de um artista específico.

GET /musicas/artists?artists=beyonce

POST

Usamos **POST** para criar um novo recurso. Uma solicitação **POST** requer um corpo no qual você define os dados da entidade a ser criada.

Uma solicitação **POST** bem-sucedida seria um código de resposta 201. Em nossa API, utilizamos o método **POST** para adicionar uma música nova.

POST /musicas

```
{  
  "title": "Meu Nome É Bond",  
  "artista": ["Danny Bond"],  
  "Lançamento": "2017"  
}
```

HTTP - PATCH & PUT & DELETE

PUT

O **PUT** substitui todos os atuais dados do recurso de destino pelos dados passados na requisição. Perceba, que estamos falando de uma **atualização integral**. Existe, então, a possibilidade de atualizar todo o recurso em apenas uma requisição.

PUT /podcast/:idDoPodcast

```
{ "id": idDoPodcast,  
  "title": "tituloDoPod",  
  "podcaster": "podApresentador",  
  "topico": "topicoDoPod" }
```

PATCH

O **Patch** aplica modificações parciais em um recurso. Logo, é possível modificar apenas uma parte do recurso. Perceba que **NÃO** estamos falando de um subconjunto de uma atualização completa (PUT), estamos falando de uma **atualização parcial**, o que torna as coisas mais flexíveis.

PATCH /podcast/:idDoPodcast

```
{ "title": "novoTituloDoPod" }
```

DELETE

Usamos o método **DELETE** para **remover** um **recurso** ou uma coleção de recursos.

Quando em um formulário você clica no botão de “Excluir”, o evento que está sendo disparado passa pelos recursos do método **DELETE**.

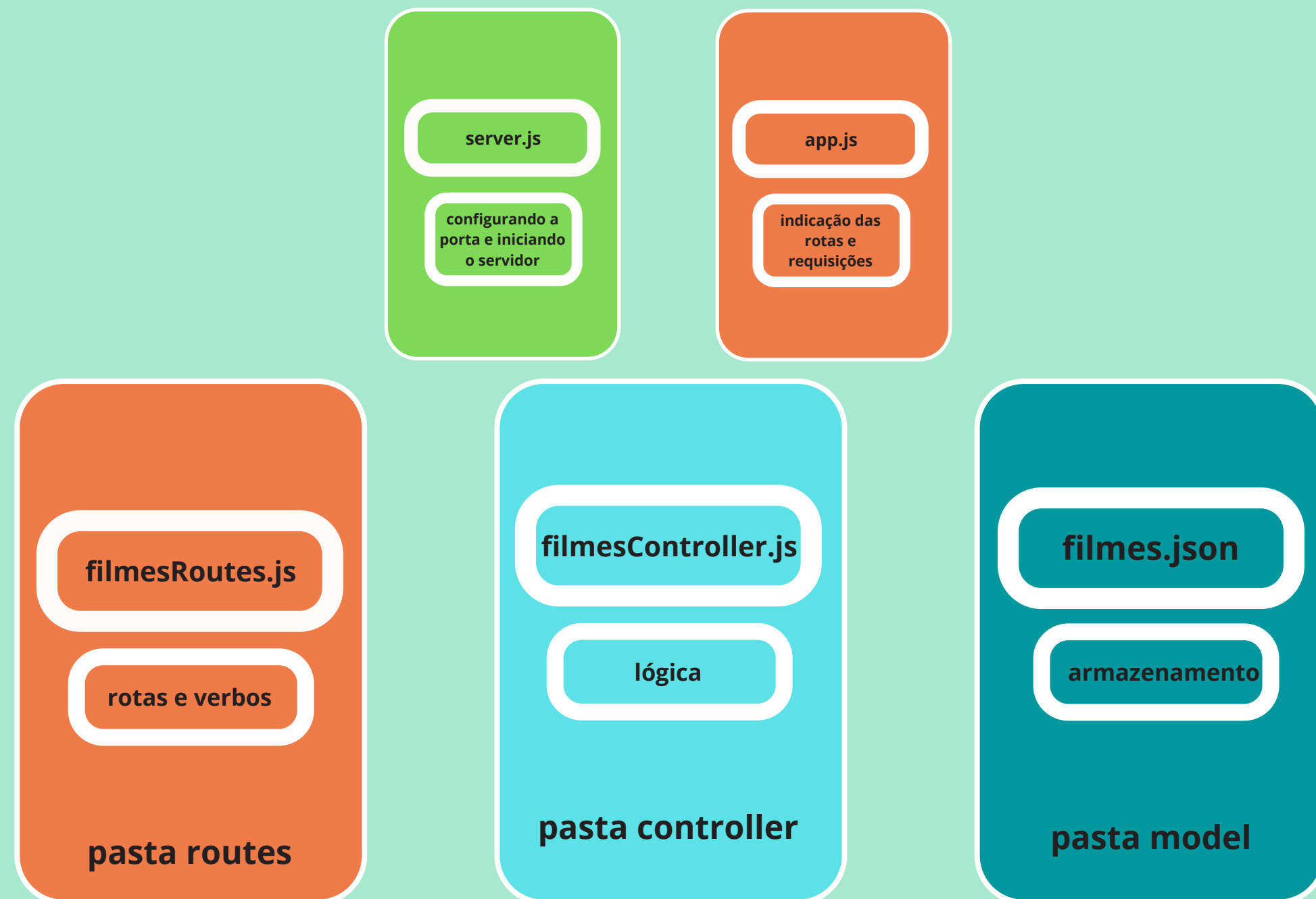
DELETE /podcast/:idDoPodcast

Arquitetura MVC

MVC é um padrão de arquitetura de software, separando sua aplicação em 3 camadas. A camada de interação do usuário(**view**), a camada de manipulação dos dados(**model**) e a camada de controle(**controller**)

Já que estamos lidando com um projeto que tem somente back-end, não lidaremos com as views, porém lidamos com as rotas(**routes**).

O **MVC** nada mais é que uma forma de organizar o nosso código



acessando JSON ●
criando rotas ●
criando lógica ●
configurando a porta e iniciando o server ●

E AS DEPENDENCIAS?

express

4.17.1 • Public • Published a year ago

Readme

Explore BETA

30 Dependencies

46.033 Dependents

264 Versions

express

Fast, unopinionated, minimalist web framework for **node**.

npm v4.17.1 downloads 58M/month linux passing windows passing coverage 100%

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

app.get('/', function (req, res) {
  res.send('Hello World')
})

app.listen(3000)
```

Install

> npm i express

± Weekly Downloads

13.961.907

Version

4.17.1

License

MIT

Unpacked Size

208 kB

Total Files

16

Issues

97

Pull Requests

52

Homepage

nodemon

2.0.4 • Public • Published 4 months ago

Readme

Explore BETA

10 Dependencies

2.477 Dependents

215 Versions



nodemon

nodemon is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected.

Install

> npm i nodemon

♥ Fund this package

± Weekly Downloads

2.893.116

Version

2.0.4

License

MIT

Unpacked Size

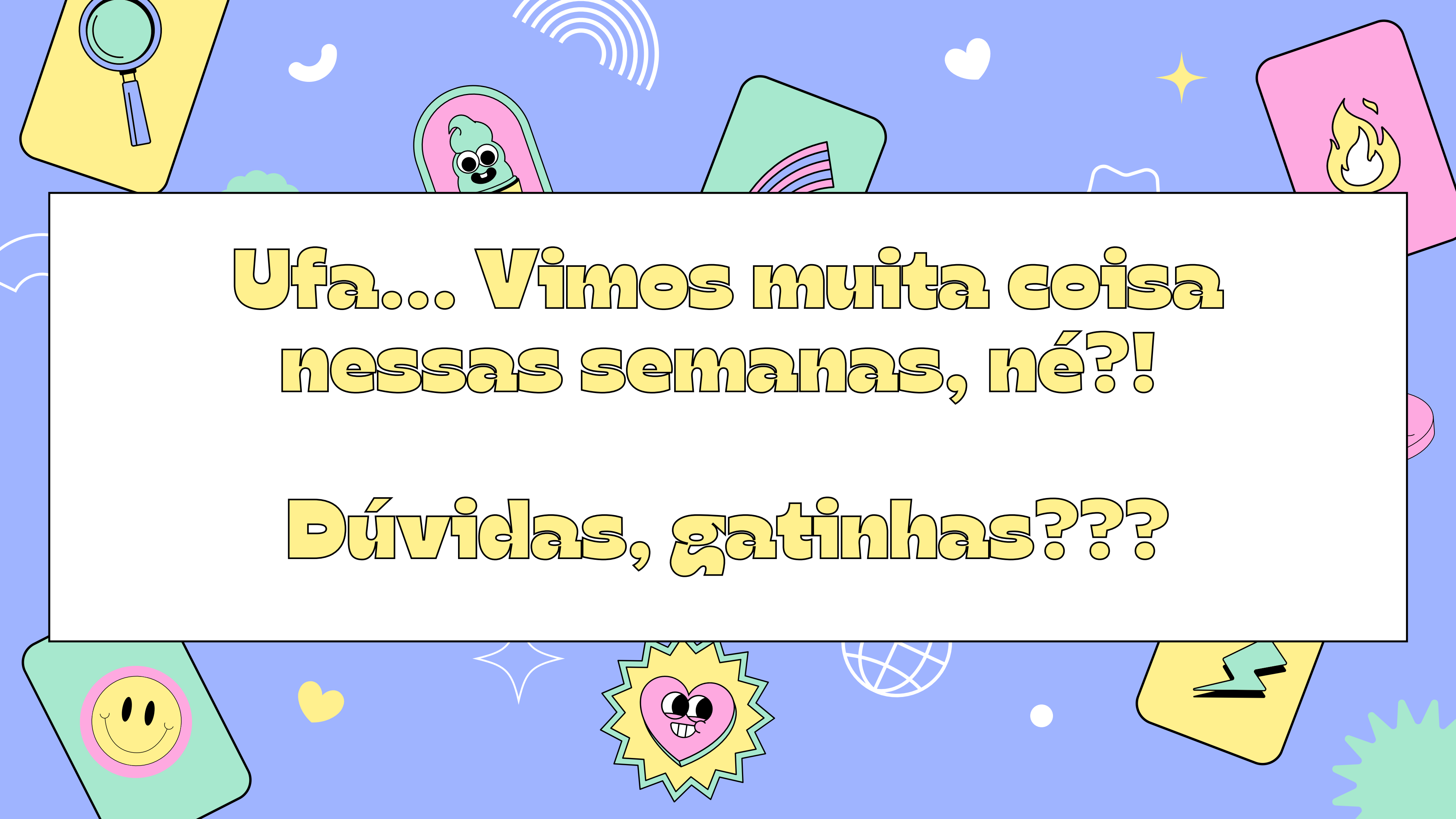
107 kB

Total Files

43

O **Express.js** é um **Framework** rápido e um dos mais utilizados em conjunto com o **Node.js**, facilitando no desenvolvimento de aplicações back-end e até, em conjunto com sistemas de templates, aplicações full-stack.

O **nodemon** é uma biblioteca que ajuda no desenvolvimento de sistemas com o **Node.js** **reiniciando automaticamente o servidor**. Ele fica monitorando a aplicação em Node, e assim que houver qualquer mudança no código, o servidor é reiniciado automaticamente.



**Ufa... Vimos muita coisa
nessas semanas, né?!**

Dúvidas, gatinhas???

Vamos de tarefinha?

Que tal controlarmos
nossos jogos e as fases
que já conseguimos
passar?

- [GET]** */games*
- Retorna todos os jogos
- [GET]** */games/:id*
- Retornar apenas um jogo específico
- [POST]** */games*
- Cadastrar novo jogo
- [PUT]** */games/:id*
- Atualizar um jogo específico
- [DELETE]** */games/:id*
- Deletar um jogo específico
- [PATCH]** */games/:id/liked*
- Atualizar se gostou ou não do jogo.

Vamos de tarefinha?

Nessa api queremos poder cadastrar séries, cada uma com inúmeras temporadas e cada temporada com uma lista de episódios.

- [GET]** */series*
- Retorna todas series
- [GET]** */series/genero*
- Retornar series de um genero específico
- [GET]** */series/:id*
- Retornar apenas uma série específico
- [POST]** */series*
- Cadastrar nova série
- [DELETE]** */series/:id*
- Deletar uma série específica
- [PATCH]** */series/:id/liked*
- Atualizar se gostou ou não da série.

Vamos de tarefinha?

DESAFIOOOOOOOO!!!!

Nossa API de séries contém várias temporadas e essas contém vários episódios. Podemos criar mais algumas rotas para trabalhar com essas temporadas e episódios:

[POST] */series/:id/season/:seasonId/episode*

- Cadastrar novo episódio na temporada, onde :id é o id da série e :seasonId é o id da temporada;

[POST] */series/:id/season*

- Cadastrar nova temporada na série, onde o :id é o id da série;

[DELETE] */series/:id/season/:seasonId*

- Deletar uma temporada específica, onde :id é o id da série e :seasonId é o id da temporada;

[DELETE] */series/:id/season/:seasonId/episode/:episodeId*

- Deletar um episódio específico na temporada, onde :id é o id da série, :seasonId é o id da temporada e :episodeId é o id do episódio;

[PATCH] */series/:id/season/:seasonId/episode/:episodeId/watched*

- Atualizar se o episódio foi assistido ou não, onde :id é o id da série, :seasonId é o id da temporada e :episodeId é o id do episódio