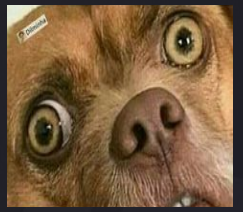
Boas-vindas!

Esteja confortável, pegue uma água
e se acomode em um local tranquilo
que já começamos.

Como você chega?

1



2

3

CODERHOUSE

Esta aula será

- gravada

Resumo
da aula anterior:

- Entendemos o protocolo HTTP
- Aprendemos a criar um servidor usando o módulo http nativo
- Criamos um servidor express
- Aprendemos mais sobre as solicitações GET de um servidor express.

Perguntas?

Aula 07. BACKEND

Introdução ao express

CODERHOUSE

Objetivos da aula

- Conhecer o status do protocolo HTTP
- Entender o conceito de API REST
- Conhecer os métodos POST, PUT, DELETE e use-os com POSTMAN ou ThunderClient
- Aprender mais sobre os métodos POST, PUT, DELETE de forma prática

Vamos revisar os principais pontos da aula anterior?

Glossário – Aula 6

HTTP: Acrônimo de Hyper Text Transfer Protocol. É o protocolo que nos permite comunicar através da Internet.

Protocolo: Conjunto de padrões e normas que devem ser seguidos para realizar uma comunicação correta.

Servidor: Sistema que permite receber solicitações de outros computadores e retornar uma resposta a eles.

xpress: Framework de desenvolvimento para criação de servidores de forma escalável.

Endpoint: Ponto específico (rota) sob o qual o servidor se conectará ao serviço a ser prestado

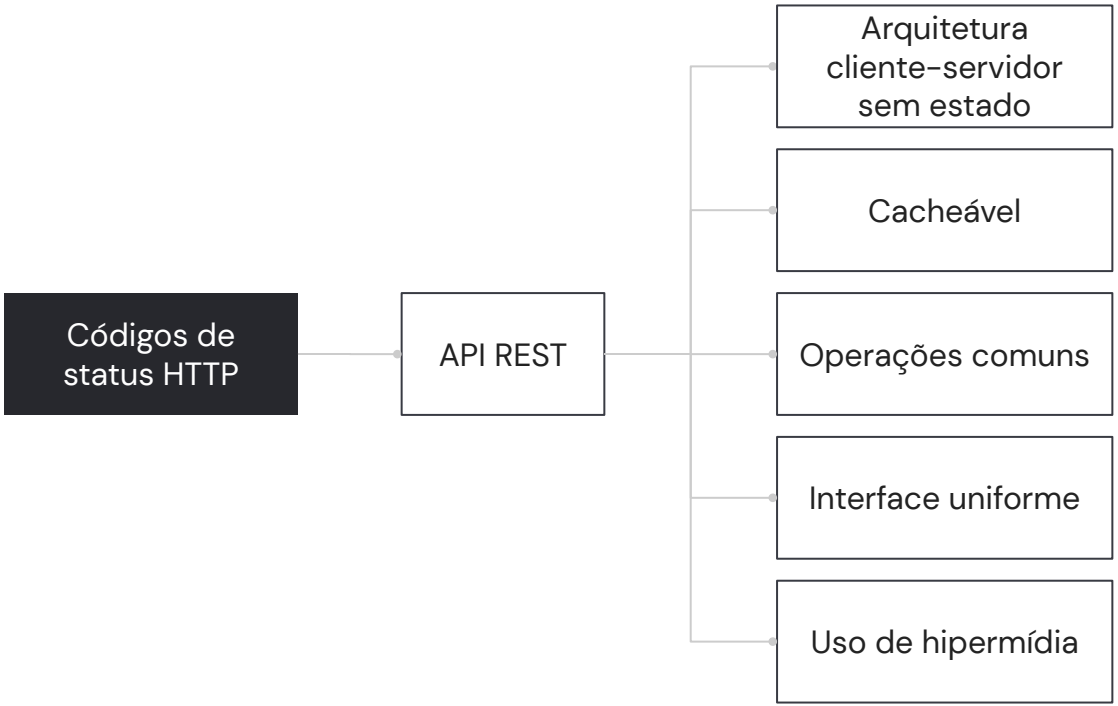
Método GET: método do protocolo HTTP que nos permite indicar ao servidor que queremos fazer uma consulta para obter dados.

Glossário – Aula 6

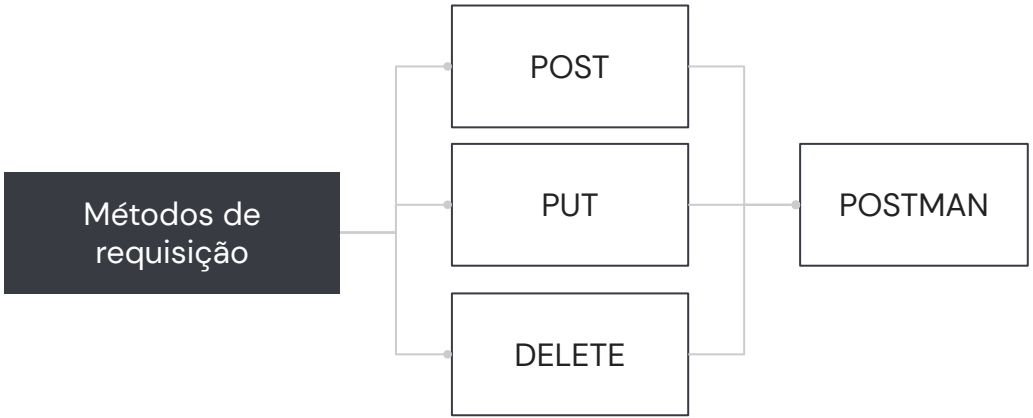
Módulo http: módulo nativo Nodejs que permite programar um servidor http.

Objeto req: Objeto utilizado dentro dos serviços expressos para realizar consultas mais complexas. seus principais componentes são: req.method, req.url req.query, req.params, req.body

MAPA DE CONCEITOS



MAPA DE CONCEITOS

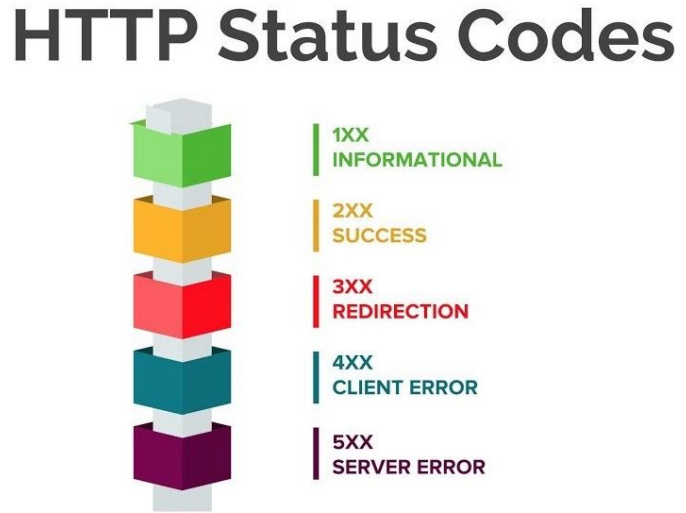


Agora sim... Vamos lá?

Códigos de status HTTP

O que é um código de status?

Quando fazemos uma solicitação ao servidor através do protocolo HTTP, o servidor deve responder não apenas com informações , mas com um status do processo. Este é um código que nos permitirá saber como é o processo, ou como terminou.

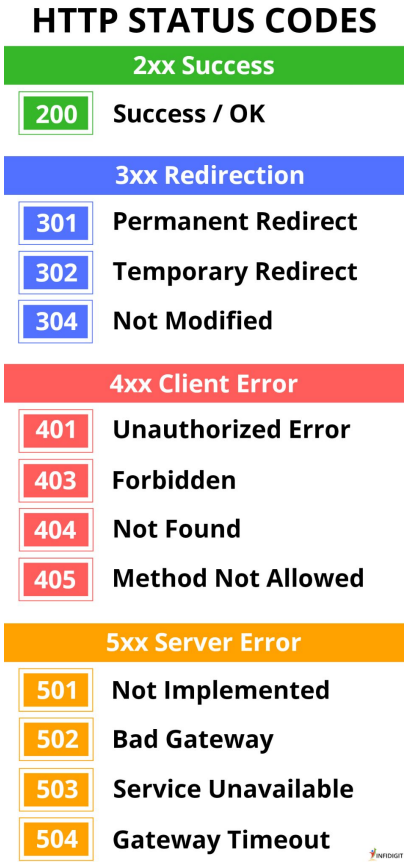


Códigos de status HTTP

Quando o servidor responde com um código de status, isso permite que você saiba o que aconteceu com a consulta que estava fazendo e fornece ao cliente informações sobre o que aconteceu.

- ✓ 1xx: Status “informativo”
- ✓ 2xx: Status “ok”
- ✓ 3xx: Status do redirecionamento.
- ✓ 4xx: Status de erro do cliente.
- ✓ 5xx: Status de erro do servidor.

Documentação sobre Http Status



Para relembrar

Alguns dos mais importantes:

- ✓ **200:** Indica que a solicitação foi processada corretamente. Não houve nenhum tipo de problema desde a consulta até a resposta.
- ✓ **3xx:** Refere-se a redirecionamentos, quando um recurso foi movido ou precisamos apontar para outro serviço.
- ✓ **400:** É utilizado quando o cliente faz uma solicitação que não está de acordo com as regras de comunicação (uma consulta incorreta, talvez tenha falhado no envio de um dado ou veio em um formato incorreto).
- ✓ **401:** É utilizado quando o cliente não se identificou com o servidor sob alguma credencial, não consegue acessar o recurso

Para relembrar

- ✓ **403:** É utilizado quando o cliente já se identificou, mas suas credenciais não possuem nível de privilégios suficiente para acessar o recurso, equivale a dizer "Quem é você, mas nem está autorizado"
- ✓ **404:** É utilizado quando o recurso não foi encontrado, seja algum dado solicitado ou mesmo o próprio endpoint.
- ✓ **500:** É usado quando algo aconteceu no servidor, não necessariamente um erro do cliente, mas um erro ou "detalhe" que o servidor não considerou ao lidar com um caso.

Exemplo

airbnb

Shoot!

Well, this is unexpected...

Error code: 500

An error has occurred and we're working to fix the problem! We'll be up and running shortly.

If you need immediate help from our customer service team about an ongoing reservation, please call us. If it isn't an urgent matter, please visit our Help Center for additional information. Thanks for your patience!

For urgent situations please call us

DADO CURIOSO

Você sabia que...?

Em 1998, um código de status 418 significando “eu sou um bule” foi adicionado como uma piada do Dia da Mentira ao Protocolo de Controle de Cafeteira de Hipertexto. Este estado significa que o servidor se recusa a executar a tarefa solicitada, porque é um bule.

Ver documentação oficial

⚠️ Importante!

Lembre-se que você é o desenvolvedor do servidor, então **é sua responsabilidade saber quando colocar cada código de status. Se não configurarmos nosso servidor para retornar vários status, será muito mais difícil rastrear problemas.**

Entendendo uma API REST

API (Application Programming Interface)

É um conjunto de definições e regras que permitem que duas equipes se integrem para trabalhar juntas. A melhor analogia para entender isso é que uma API atua como um "contrato" entre a frontend e o backend.

Aprofundando conhecimentos em API

A API então permite perguntas como:

- ✓ Qual endpoint devo direcionar para a tarefa de que preciso?
- ✓ Que método devo usar para esse recurso?
- ✓ Que informação devo enviar para realizar corretamente o meu pedido?

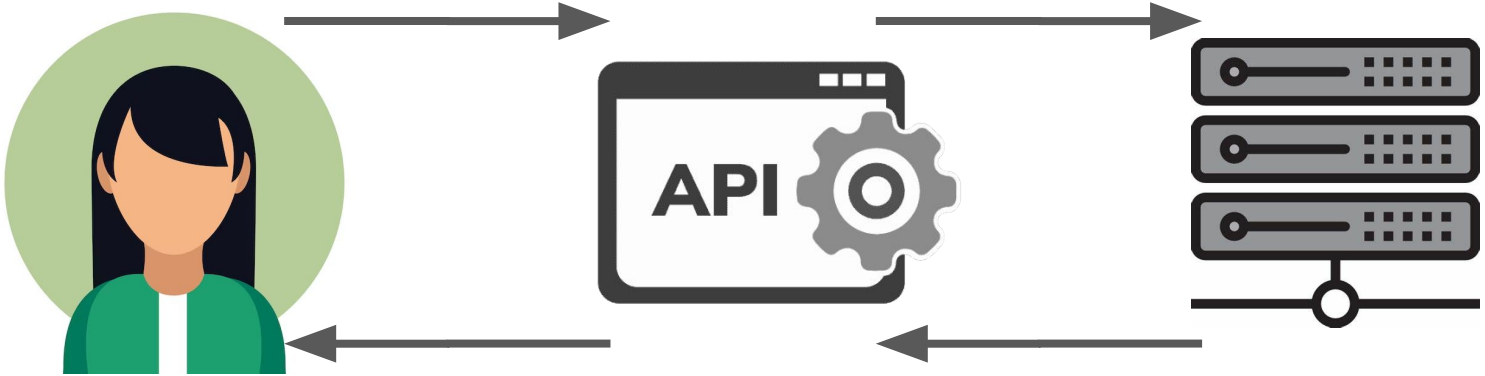
O cliente precisa de algo do servidor, então ele precisa fazer uma solicitação(request).

Para que a requisição chegue corretamente ao servidor, ela deve apontar para o endpoint correto, com o método correto, com as informações corretas

O servidor recebe a solicitação. Se todas as especificações da API forem atendidas, o processamento pode ser realizado com sucesso

Requisição

Processamento



Resposta

Resultado

O cliente, tendo cumprido o que a API especificou, poderá obter seu resultado satisfatoriamente e utilizá-lo.

Cumprir o contrato API garante (na maioria dos casos) que haverá um resultado satisfatório

REST

Já temos as regras para comunicar, mas e a estrutura da mensagem?

Quando fazemos uma requisição ou quando recebemos uma resposta, deve possuir um formato . **REST** (REpresentational State Transfer) permite definir a estrutura que os dados devem ter para serem transferidos .

Aprofundamento sobre REST

A API respondia as dúvidas sobre como se comunicar corretamente, porém, o REST define como deve ser o corpo da mensagem a ser transmitida. (Você pode falar com o presidente se cumprir o protocolo (HTTP) e as regras (API), mas de que adianta se a forma como estruturamos nossa mensagem (REST) não é correta?)

Os dois formatos mais importantes são JSON e XML . A utilização da estrutura dependerá das necessidades do projeto. Usaremos JSON.

Como você notará, um JSON se parece com um objeto! então a sintaxe é muito mais amigável.

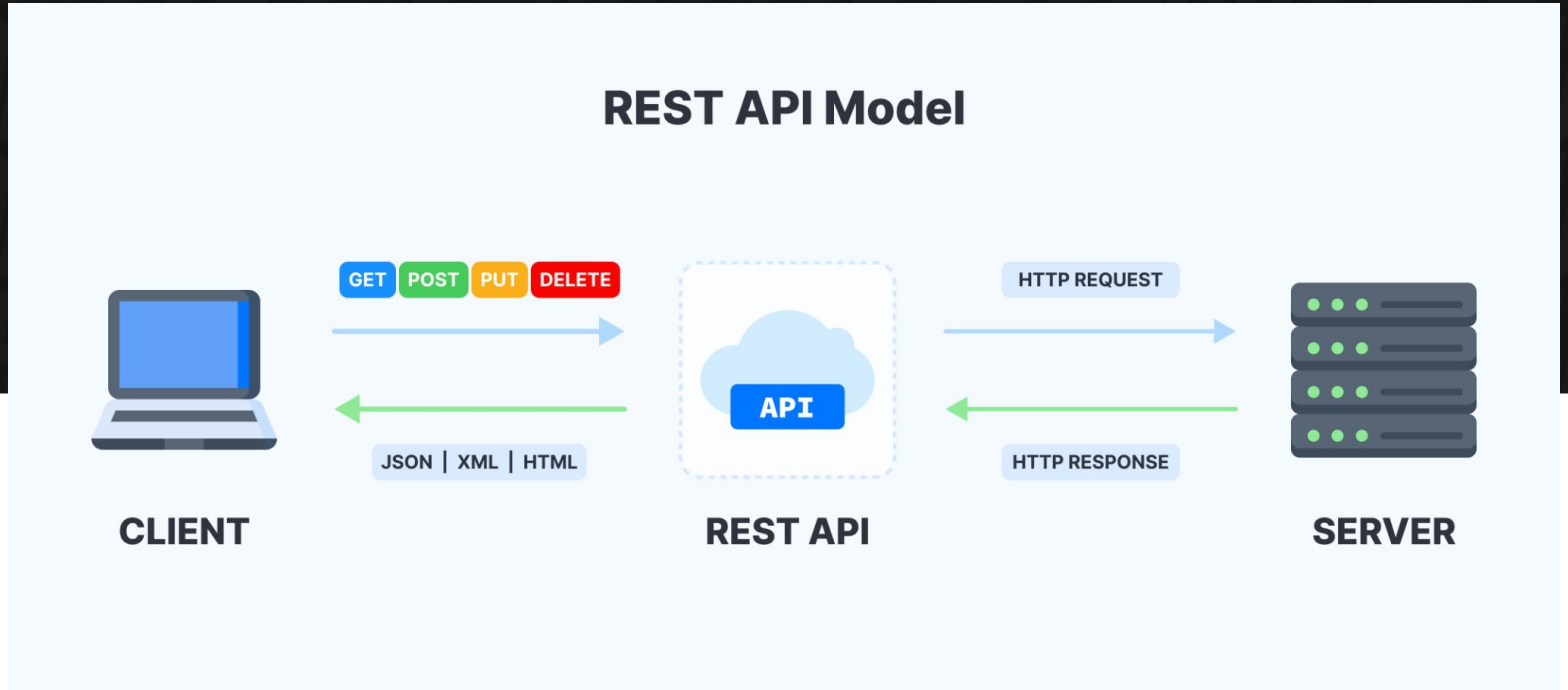
```xml
<fatura>
    <cliente>Gomez </cliente>
    <emissor>Perez S.A.</emissor>
    <tipo>A</tipo>
    <itens>
        <item>Produto 1</item>
        <item>Produto 2</item>
        <item>Produto 3</item>
    </itens>
</fatura>
```

```json
{
    "cliente": "Gomez",
    "emissor": "Peres S.A.",
    "tipo": "A",
    "itens": [
        "Produto 1",
        "Produto 2",
        "Produto 3"
    ]
}
```

Portanto, uma API REST é...

Um modelo completo para ter os protocolos, regras e até a estrutura da informação perfeitamente estipulados, para poder fazer um sistema completo de comunicação entre computadores.

Modelo de uma API REST



Quais características que uma API REST deve ter?

Arquitetura cliente-servidor sem estado

- ✓ **Cada mensagem HTTP** contém todas as informações necessárias para entender a solicitação.
- ✓ Como resultado, **nem o cliente nem o servidor precisam se lembrar de qualquer estado das comunicações entre mensagens.**
- ✓ **Essa restrição mantém o cliente e o servidor fracamente acoplados**: o cliente não precisa conhecer os detalhes de implementação do servidor e o servidor “não se importa” sobre como os dados que ele envia ao cliente são usados.

Cacheável

- ✓ Deve suportar um sistema de cache.
- ✓ A infraestrutura de rede deve suportar um cache multinível.
- ✓ Esse armazenamento evita a repetição de várias conexões entre o servidor e o cliente, nos casos em que solicitações idênticas gerariam a mesma resposta.

Operações comuns:

Todos os recursos por trás da nossa API devem poder ser consumidos através de requisições HTTP, preferencialmente suas principais (POST, GET, PUT e DELETE).

Essas operações são frequentemente equiparadas a operações CRUD em bancos de dados (em inglês: Create, Read, Update, Delete, em português: Criar, Ler, Modificar e Apagar).

Por serem requisições HTTP, devem retornar os códigos de status correspondentes com suas respostas, informando o resultado das mesmas.

Interface uniforme

- ✓ Deve suportar um sistema de cache.
- ✓ Em um sistema REST, **cada ação (mais corretamente, cada recurso) deve ter um URI (Uniform Resource Identifier), um identificador único.**
- ✓ Isso nos dá acesso às informações, tanto para consultá-las, como para modificá-las ou excluí-las, mas também para compartilhar sua localização exata com terceiros.

Uso de hipermídia

- ✓ Cada vez que uma solicitação é feita ao servidor e ele retorna uma resposta, parte da informação retornada também pode ser hiperlinks de navegação associados a outros recursos do cliente.
- ✓ Com isso, **é possível navegar de um recurso REST para vários outros simplesmente seguindo links sem a necessidade do uso de registros ou outra infraestrutura adicional.**

Break

5 minutos e voltamos!

Break

10 minutos e voltamos!

CODERHOUSE

Métodos de Requisição

Métodos de Requisição

Um método **é uma definição que faz parte do protocolo HTTP, que é usado para canalizar o tipo de solicitação que estou fazendo em um determinado endpoint.** Desta forma, o cliente pode chamar o mesmo terminal, mas com métodos diferentes , indicando qual operação deseja realizar com o referido recurso. Os principais métodos são:

- ✓ **GET:** obter um recurso
- ✓ **POST:** Criar ou adicionar um recurso
- ✓ **PUT:** Modificar um recurso
- ✓ **DELETE:** Excluir um recurso

Princípios

Um aplicativo RESTful requer uma abordagem de design diferente da maneira típica de pensar sobre um sistema: **o oposto de RPC.**

RPC (Remote Procedure Calls, chamadas a procedimentos remotos) baseia seu funcionamento nas operações que o sistema pode realizar (ações, geralmente verbos). Ex: getUser()

Em REST, ao contrário, **a ênfase é colocada em recursos** (geralmente substantivos), especialmente os nomes atribuídos a cada tipo de recurso. Ex: Usuários.

Cada funcionalidade relacionada a este recurso teria seus próprios identificadores e requisições HTTP.

Exemplos de como declarar endpoints corretamente

Sim	Não	Por quê?
GET *api/frutas* GET *api/frutas/:pid*	GET *api/frutas/obter* GET *api/frutas/obter/:pid*	GET já significa "obter", então não há motivo para redundância com o método e o endpoint.
POST *api/frutas*	POST *api/frutas/adicionar*	POST já se refere à criação de uma nova fruta, portanto, o método e o endpoint são redundantes

“api/frutas” é perfeitamente funcional, sem a necessidade de declarar coisas adicionais, então podemos reutilizar o endpoint, desde que seus métodos sejam diferentes.

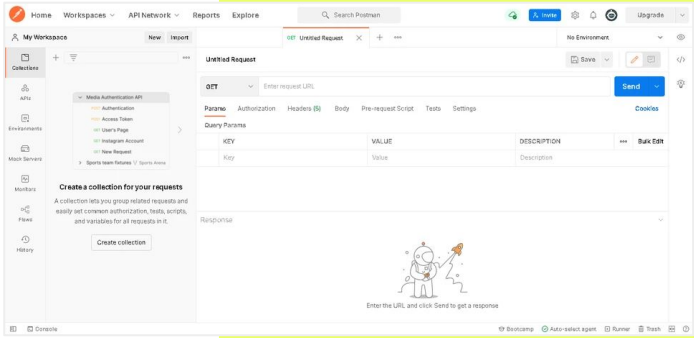
Postman

Postman

O problema: O navegador só pode enviar solicitações com o método GET da url (por isso poderíamos usá-lo do navegador sem problemas na aula passada), no entanto, para poder usar o restante dos métodos, não será possível com o navegador

A solução: O POSTMAN é um software profissional que nos permitirá gerir os pedidos fingindo ser um cliente. Desta forma, quebramos a limitação do navegador e podemos testar todos os nossos endpoints.

Baixar Postman



CODERHOUSE

Método POST

Método POST

Ele é usado para "criar" recursos, POST é **usado para operações onde não precisamos obter um recurso, mas adicionar um**. Alguns dos casos em que são usados são:

- ✓ Registrar um usuário
- ✓ Entrar um usuário
- ✓ Criar um produto
- ✓ Criar um animal de estimação
- ✓ Criar um carrinho de compras
- ✓ Enviar informações para um e-mail.

Ele conta com o recurso req.body, onde o corpo representa as informações que o cliente envia para criar.

⚠️ Importante!

Para que nosso servidor expresso possa interpretar automaticamente mensagens do tipo JSON em formato urlencoded ao recebê-las, devemos indicá-lo explicitamente, adicionando as seguintes linhas após criá-lo.

```
app.use(express.json())
app.use(express.urlencoded({ extended: true }))
```

⚠️ Importante!

{extended:true} especifica que o objeto req.body conterá valores de qualquer tipo em vez de apenas strings.

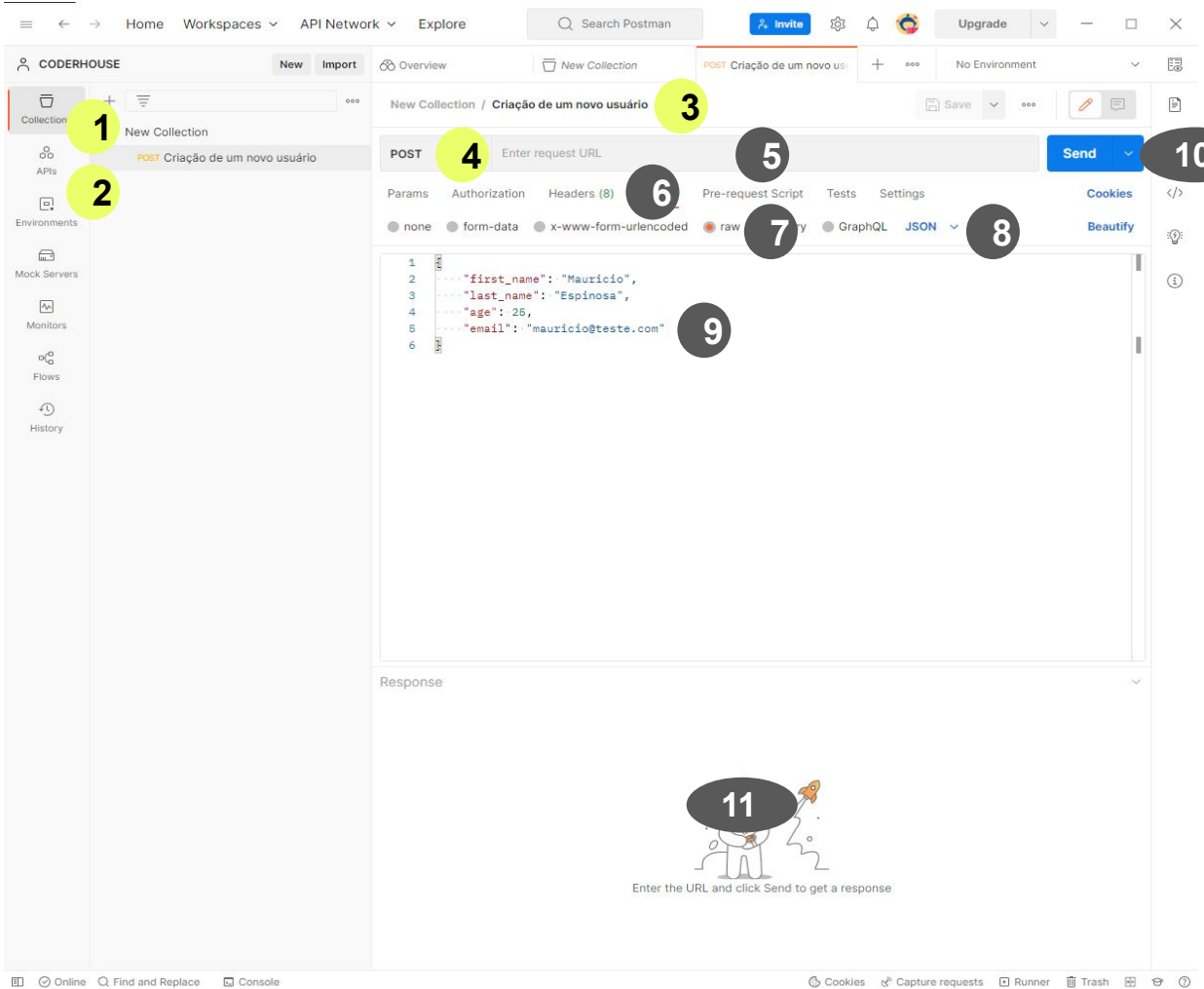
Sem esta linha, o servidor não saberá interpretar os objetos recebidos!

Como usar um método POST

app.js > ...

```
import express from 'express';

const app = express();

const server = app.listen(8080, () =>console.log("Escuta na PORTA 8080"));
//Primeiro temos que configurar nosso servidor para que ele receba informações do cliente.

app.use(express.json()); //Conforme o método indica, agora o servidor poderá receber jsons no momento da requisição
app.use(express.urlencoded({extended:true})); //permite para que sejam enviadas informações também da URL

let users = []; //Aqui iremos armazenar os usuários que estamos criando. Vamos criá-los a partir do método POST

//Agora, ao invés de chamar app.get, chamaremos app.post, indicando que queremos CRIAR um recurso (usuário)
app.post('/api/user', (req, res)=> {
    let user = req.body; //Lembre-se que req.body é o JSON que o usuário enviará ao fazer a requisição
    //Podemos validar que determinados campos sejam atendidos antes de adicioná-lo.
    if(!user.first_name||!user.last_name){ //verificamos se o primeiro ou último nome estava faltando.
        /**
        * Por se tratar de um erro onde o cliente se enganou ao enviar informações incompletas, o status que
        * Retornaremos um status 400. Colocaremos antes do .send conforme indicado abaixo.
        */
        return res.status(400).send({status: "error",error: "Valores incompletos"})
    }
    //Caso não tenha inserido o if, significa que o cliente enviou os campos completos
    //Passamos a adicioná-lo ao array users
    users.push(usuário);
    res.send({status: "success", message: "User created"}) //O status 200 é implícito quando não o indicamos.
});
```

CODERHOUSE


Home
Workspaces
API Network
Explore
Search Postman
Invite
Upgrade
CODERHOUSE
New
Import
Collection
APIs
Environments
Mock Servers
Monitors
Flows
History
New Collection
POST Criação de um novo usuário
Overview
New Collection
POST Criação de um novo us
No Environment
New Collection / Criação de um novo usuário
Save
POST
Enter request URL
Send
Params
Authorization
Headers (8)
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
GraphQL
JSON
Beautify
{
"first_name": "Mauricio",
"last_name": "Espinosa",
"age": 25,
"email": "mauricio@teste.com"
}
Response
Enter the URL and click Send to get a response
Online
Find and Replace
Console
Cookies
Capture requests
Runner
Trash
1
2
3
4
5
6
7
8
9
10
11

Passo a passo

1

Criar uma collection

Endpoint a ser utilizado, de forma a coincidir com o declarado no servidor.

2

Criar uma request

Incluir dentro da coleção, para manter a ordem do projeto.

3

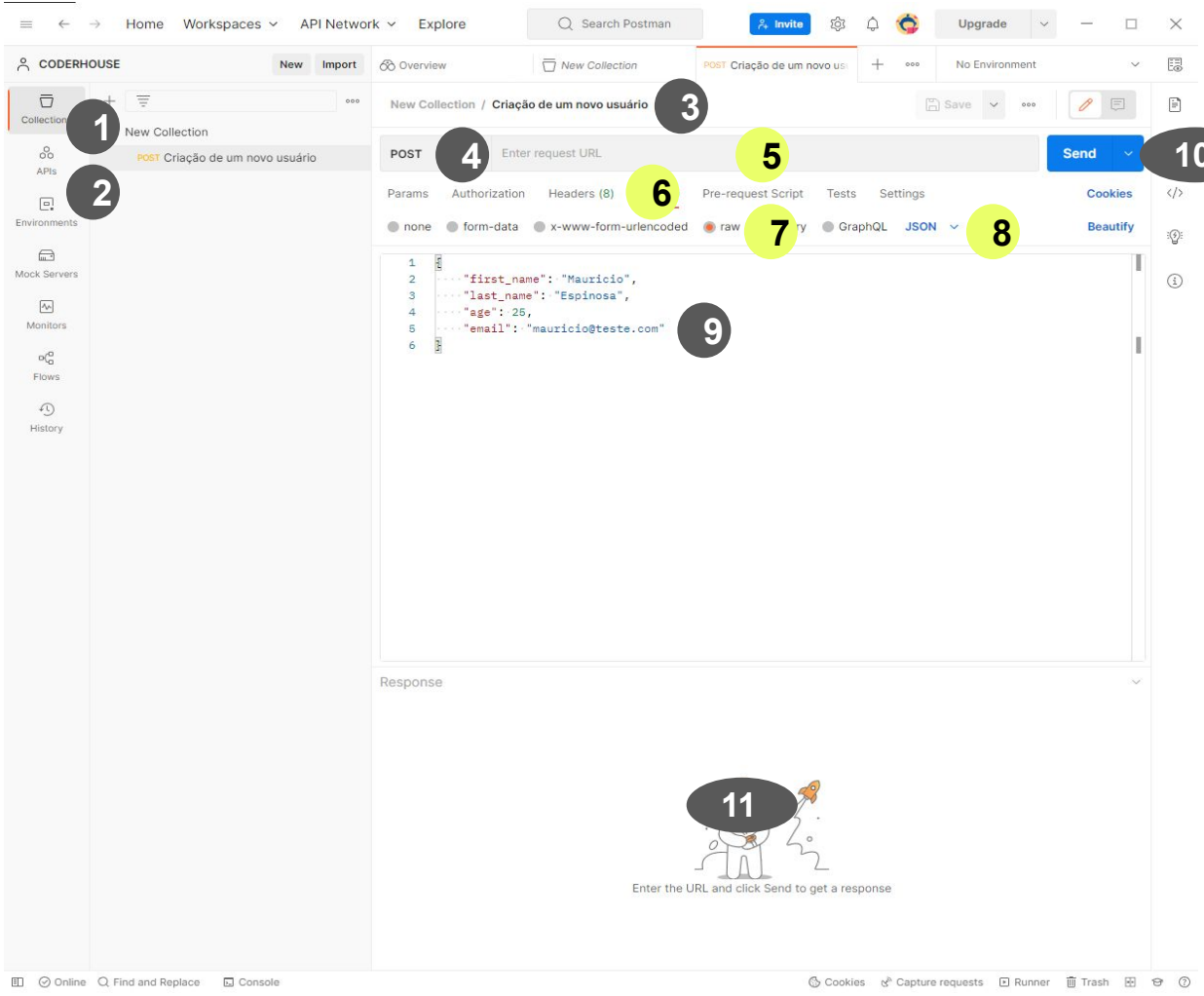
Adicione um nome a nossa request

Indicamos um nome que seja útil para reconhecer a operação realizada pelo referido request.

4

Mudamos o MÉTODO

Crucial entendermos que estamos trabalhando com vários métodos.


Home
Workspaces
API Network
Explore
Search Postman
Invite
Upgrade
CODERHOUSE
New
Import
Collections
APIs
Environments
Mock Servers
Monitors
Flows
History
New Collection
POST Criação de um novo usuário
Overview
New Collection
POST Criação de um novo us
No Environment
New Collection / Criação de um novo usuário
Save
POST
Enter request URL
Send
Params
Authorization
Headers (8)
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
GraphQL
JSON
Beautify
{
"first_name": "Mauricio",
"last_name": "Espinosa",
"age": 25,
"email": "mauricio@teste.com"
}
Response
Enter the URL and click Send to get a response
Online
Find and Replace
Console
Cookies
Capture requests
Runner
Trash
1
2
3
4
5
6
7
8
9
10
11

Passo a passo

5

Inserimos a URL

Endpoint a ser utilizado, de forma a coincidir com o declarado no servidor.

6

Selecionamos a aba body

Para inserir conteúdo para enviar ao servidor.

7

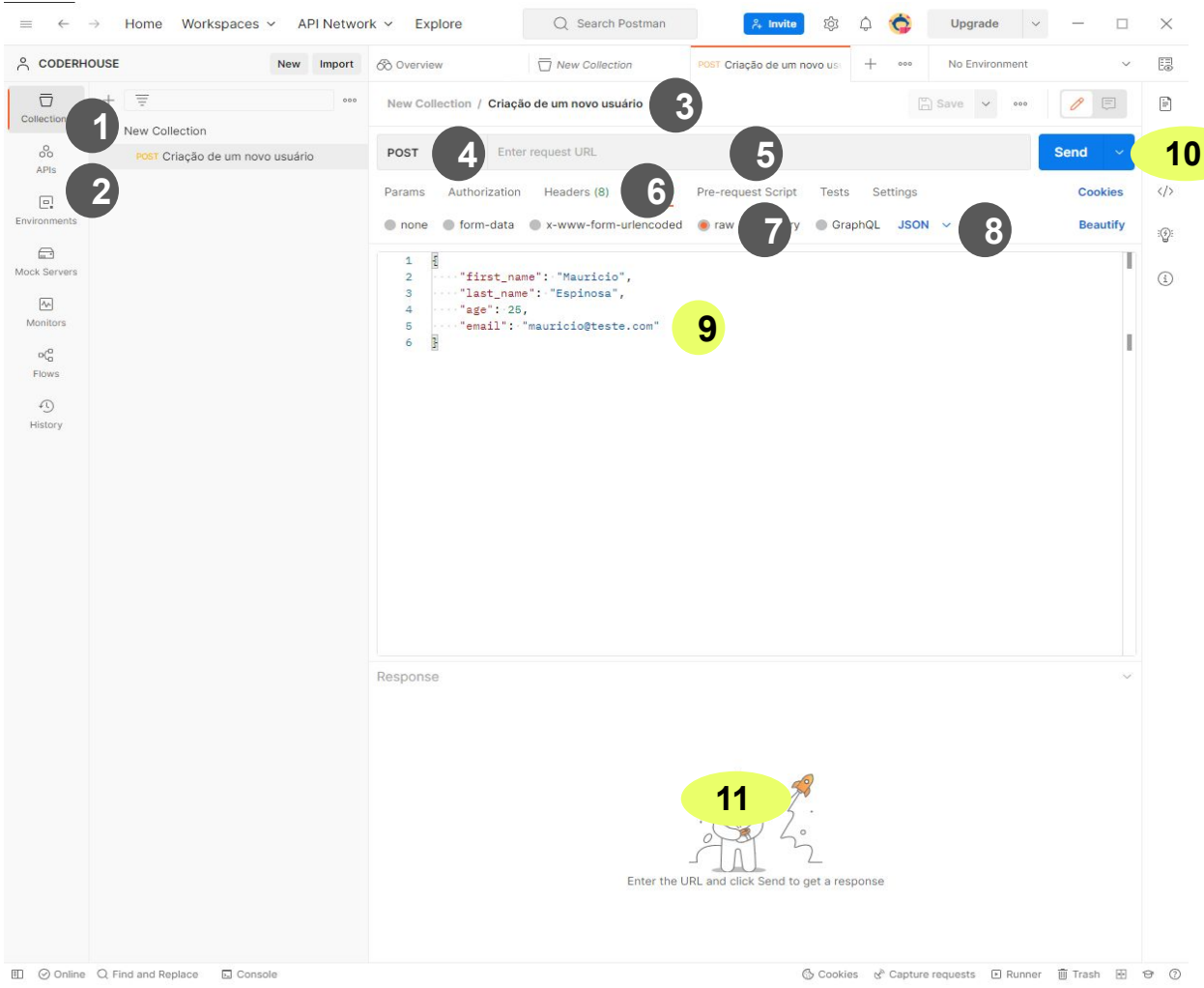
Colocamos o modo raw

Indicando que o corpo será criado do zero.

8

JSON

Selecionamos a opção "JSON", pois é a estrutura que utilizaremos.


Home
Workspaces
API Network
Explore
Search Postman
Invite
Upgrade
CODERHOUSE
New
Import
Collections
APIs
Environments
Mock Servers
Monitors
Flows
History
New Collection
POST Criação de um novo usuário
Overview
New Collection
POST Criação de um novo us
No Environment
New Collection / Criação de um novo usuário
Save
POST
Enter request URL
Send
Params
Authorization
Headers (8)
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
GraphQL
JSON
Beautify
{
"first_name": "Mauricio",
"last_name": "Espinosa",
"age": 25,
"email": "mauricio@teste.com"
}
Response
Enter the URL and click Send to get a response
Online
Find and Replace
Console
Cookies
Capture requests
Runner
Trash
1
2
3
4
5
6
7
8
9
10
11

Passo a passo

9

Especificamos

O JSON a enviar.

10

Send

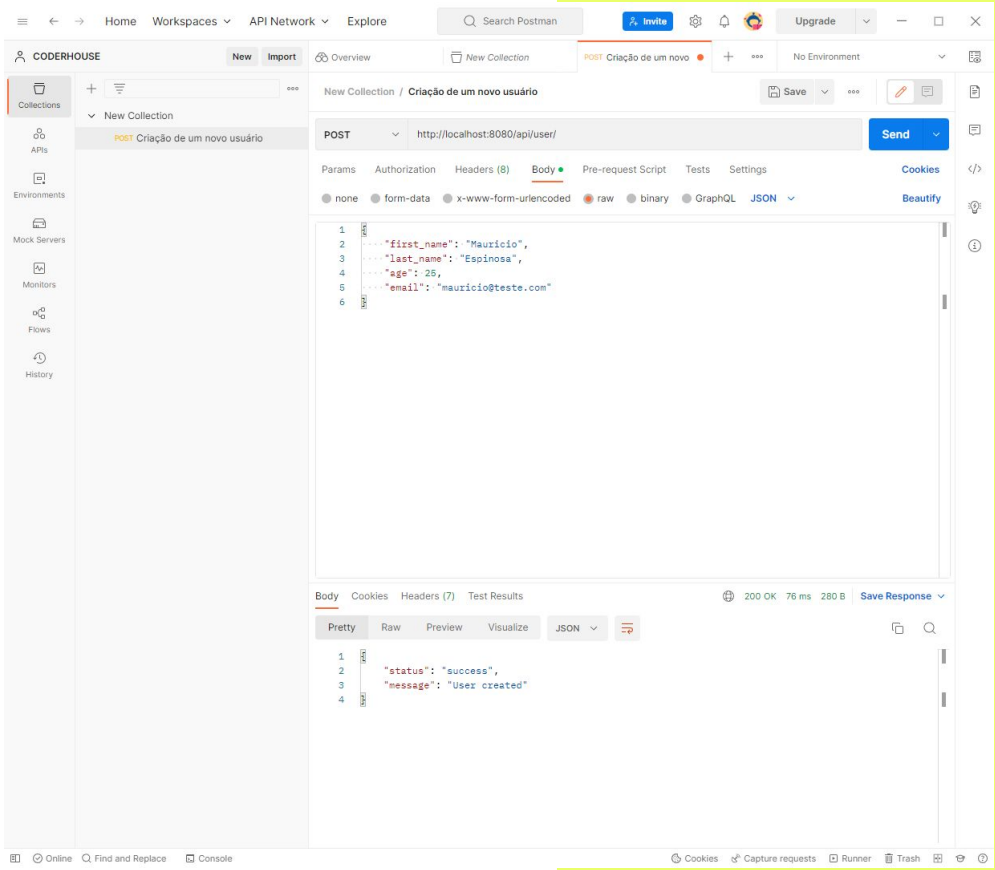
Clique no botão SEND.

11

Quais são os resultados?

No painel visualizamos o resultado.

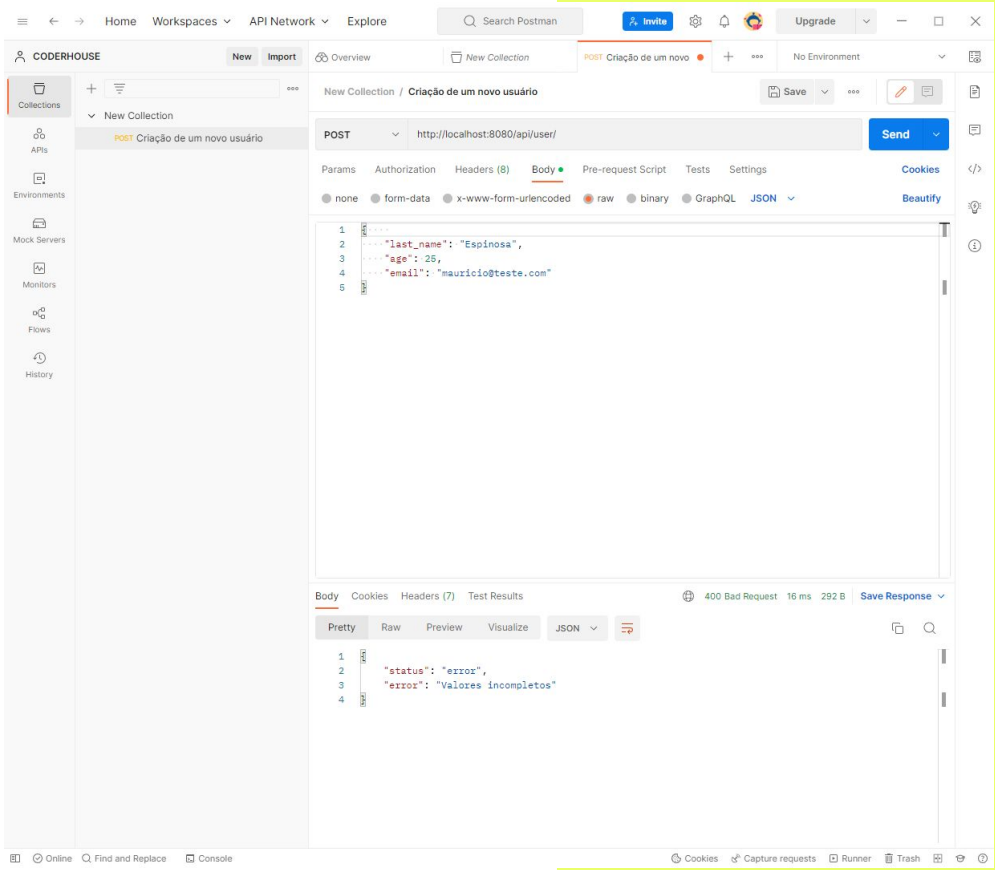
Exemplo de teste do endpoint com dados corretos



CODERHOUSE

Exemplo testando o endpoint sem o campo "first_name" que é inválido.

Observe que agora o status é 400 e o código é erro.



CODERHOUSE

Método PUT

Já vimos como criar um recurso, agora como podemos modificar esse recurso?
O método PUT é usado para isso.

CODERHOUSE

Para relembrar

Método PUT

Para trabalhar com PUT, não apenas enviamos o body da request, mas também enviamos o id, nome ou qualquer identificador como parâmetros para que o servidor saiba qual recurso específico atualizar.

Existem duas formas de atualizar um recurso: **atualizar apenas os campos obrigatórios, ou enviar o objeto inteiro para atualização**, ambas as formas são válidas quando falamos em atualização, e vai depender do contexto.

Como é uma atualização básica usando o método PUT?

app.js > app.put('/api/user') callback

```
import express from 'express';

const app = express();

const server = app.listen(8080, () =>console.log("Escuta na PORTA 8080"));
//Primeiro temos que configurar nosso servidor para que ele receba informações do cliente.

app.use(express.json()); //Conforme o método indica, agora o servidor poderá receber jsons no momento da requisição
app.use(express.urlencoded({extended:true})); //permite para que sejam enviadas informações também da URL

let users = []; //Aqui iremos armazenar os usuários que estamos criando. Vamos criá-los a partir do método POST

app.put('/api/user', (req, res) => {
    let user = req.body;

    //Validamos se os campos chaves foram enviados para podermos alterar.
    if (!user.first_name || !user.last_name) { //verificamos se o primeiro ou último nome estava faltando.
        return res.status(400).send({status: "error",error: "Valores chaves não enviados"})
    }

    //Buscamos pelos valores chave para alterar os outros valores
    users = users.map(obj => {
        if (obj.first_name == user.first_name && obj.last_name == user.last_name) {
            return {...user};
        }

        return {...obj};
    });
    res.send({status: "success", message: "User changed"}) //O status 200 é implícito quando não o indicamos.
});

//Agora, ao invés de chamar app.get, chamaremos app.post, indicando que queremos CRIAR um recurso (usuário)
app.post('/api/user', (req, res)=> {
    let user = req.body; //Lembre-se que req.body é o JSON que o usuário enviará ao fazer a requisição
    //Podemos validar que determinados campos sejam atendidos antes de adicioná-lo.
    if (!user.first_name || !user.last_name) { //verificamos se o primeiro ou último nome estava faltando.
        /**
        * Por se tratar de um erro onde o cliente se enganou ao enviar informações incompletas, o status que
        * Retornaremos um status 400. Colocaremos antes do .send conforme indicado abaixo.
        */
```

CODERHOUSE

Para relembrar

Método DELETE

Como o nome indica, **usamos este método quando queremos deletar um recurso.** Aqui não é necessário enviar nada do body, porém é importante indicar o identificador no req.params para que o servidor reconheça qual recurso deve ser deletado.

Como é uma exclusão básica usando o método DELETE?

```
49
50
51  app.delete('/api/user/:name', (req, res)=>{
52      let name = req.params.name;
53      let currentLength = users.length;
54
55      users = users.filter(user=>user.first_name != name);
56      if (users.length===currentLength) {//Se o comprimento da matriz for igual, então nada foi removido return
57          res.status(404).send({status:"error",error:"Usuário não encontrado" });
58      }
59
60      res.send({status:"sucesso",mensagem: "Usuário deletado"});
61  })
```

Exemplo ao vivo

Integrando todos os métodos

- ✓ Um método GET para o mesmo endpoint será adicionado ao código de explicação, a fim de completar os 4 métodos principais.
- ✓ Será feito um fluxo completo com o POSTMAN onde podemos ver todos os endpoints trabalhando juntos.

Servidor com GET, POST, PUT, DELETE

Servidor com integração de todos os métodos

Duração: **30 minutos**

CODERHOUSE

ATIVIDADE EM SALA

Servidor com GET, POST, PUT, DELETE

Descrição da atividade.

Dada a frase: "Frase inicial", crie uma aplicação que contenha um servidor em express, que possua os seguintes métodos:

- ✓ GET '/api/frase': retorna um objeto que como campo 'frase' contém a frase completa
- ✓ GET '/api/palavras/:pos': retorna um objeto que como campo 'busca' contém a palavra encontrada na frase na posição dada (considere que a primeira palavra é #1).

ATIVIDADE EM SALA

- ✓ POST '/api/palavras': recebe um objeto com uma palavra no campo ‘palavra' e adiciona no final da frase. Retorna um objeto que contém a palavra adicionada como campo 'adicionado', e no campo 'pos' a posição em que a palavra foi adicionada.
- ✓ PUT '/api/palavras/:pos': recebe um objeto com uma palavra no campo 'palavra' e substitui aquela encontrada na posição dada na frase. Retorna um objeto que contém a nova palavra no campo 'atualizado' e a anterior no campo 'anterior'.
- ✓ DELETE '/api/palavras/:pos': apaga uma palavra da frase, de acordo com a posição dada (considere que a primeira palavra tem a posição #1).
- ✓ Usando o POSTMAN para testar a funcionalidade

Para pensar

Notamos que podemos usar o mesmo endpoint para o mesmo tipo de recurso, desde que existam métodos diferentes.

Se eles acontecerem novamente, existe uma maneira de agrupar esses endpoints para ter um código muito mais limpo?

Coloque sua resposta no chat

Perguntas?

Como foi a aula?

1

Que bom

O que foi super legal na aula e podemos sempre trazer para as próximas?

2

Que pena

O que você acha que não funcionou bem e precisamos melhorar?

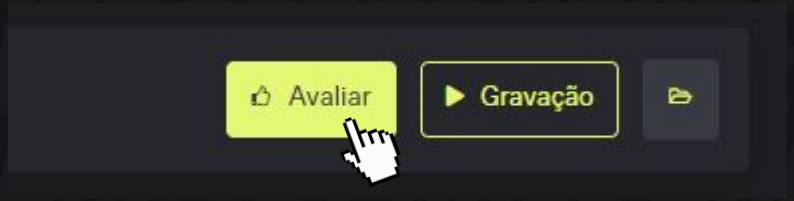
3

Que tal

Qual sugestão deveríamos tentar em próximas aulas?

CODERHOUSE

O que você achou da aula?



Seu feedback vale pontos para o Top 10!! 😎

Deixe sua opinião!

1. Acesse a plataforma
2. Vá na aula do dia
3. Clique em Avaliar

CODERHOUSE

Resumo
da aula de hoje:

- ✓ Protocolo HTTP
- ✓ API, REST, API REST
- ✓ Métodos POST, PUT, DELETE
- ✓ POSTMAN

Você quer saber mais?
Tem material extra da aula no próximo slide.😉

MATERIAL AMPLIADO

Material Extra

- ✓ Aprofundamento sobre métodos de requisição

Obrigado por estudar conosco!

CODERHOUSE