

# Como você chega?

Manda no chat qual o seu **#checkin**  
Sentimentos, expectativas pra aula...

{ reprograma }

# PUT & PATCH

{ reprograma }

# Modelo Server-Client

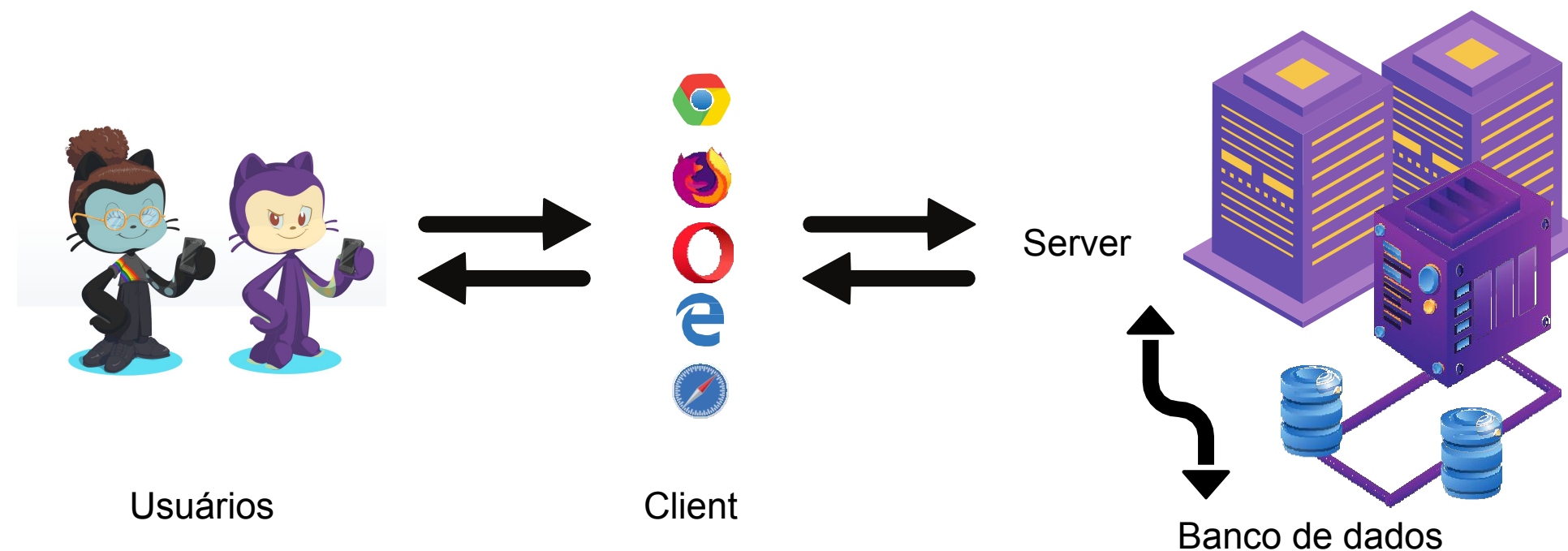
## CLIENTE

Entenda cliente como a interface que o usuário interage. É o Cliente que **solicita** serviços e informações de um ou mais servidores.

## SERVIDOR

E o Servidor é o responsável pelo processo, organização e gerenciamento das informações. É ele que **responde** às solicitações feitas pelo usuário.

Ele é um processo reativo, disparado pela chegada de pedidos de seus clientes.



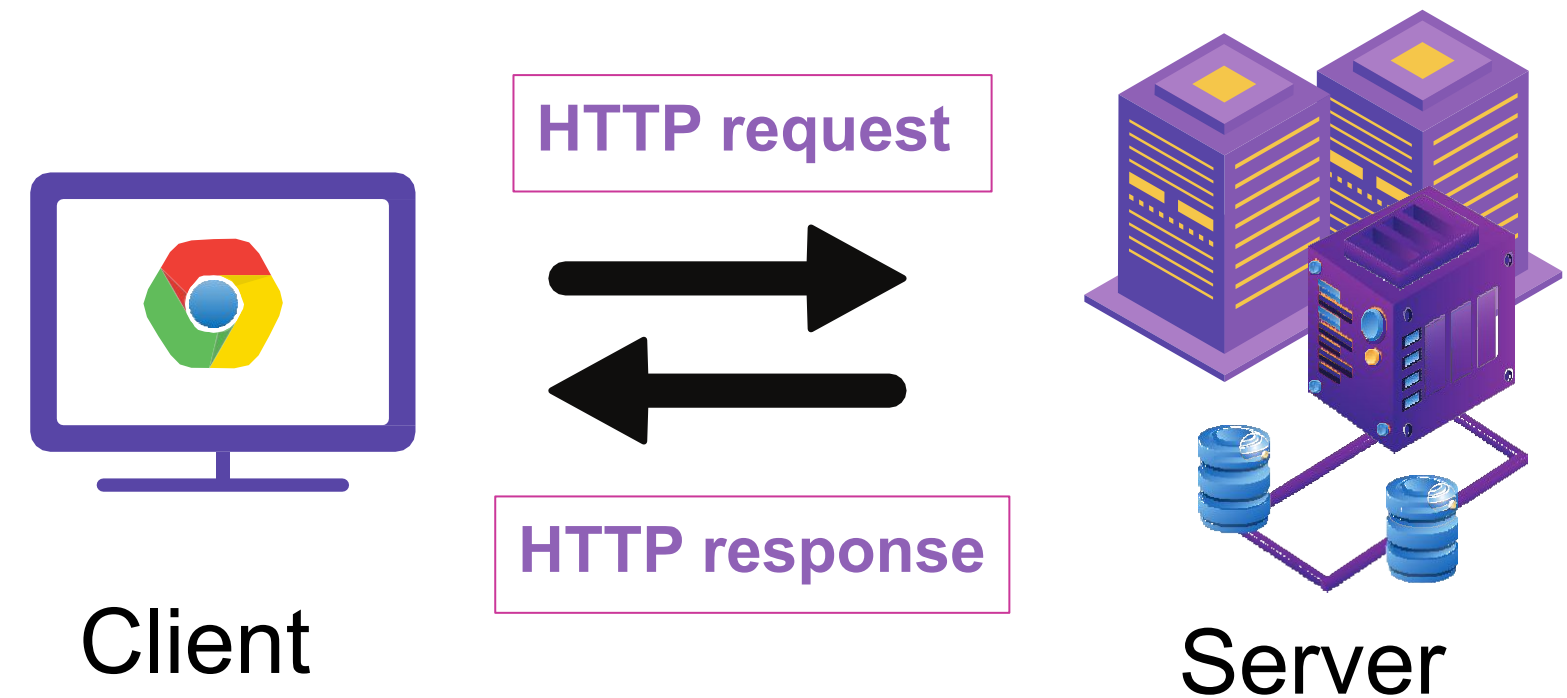
{ reprograma }

# 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.



# HTTP - Status Code

Quando o Client faz uma requisição o Server responde com um código de status numérico também padronizado.

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

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

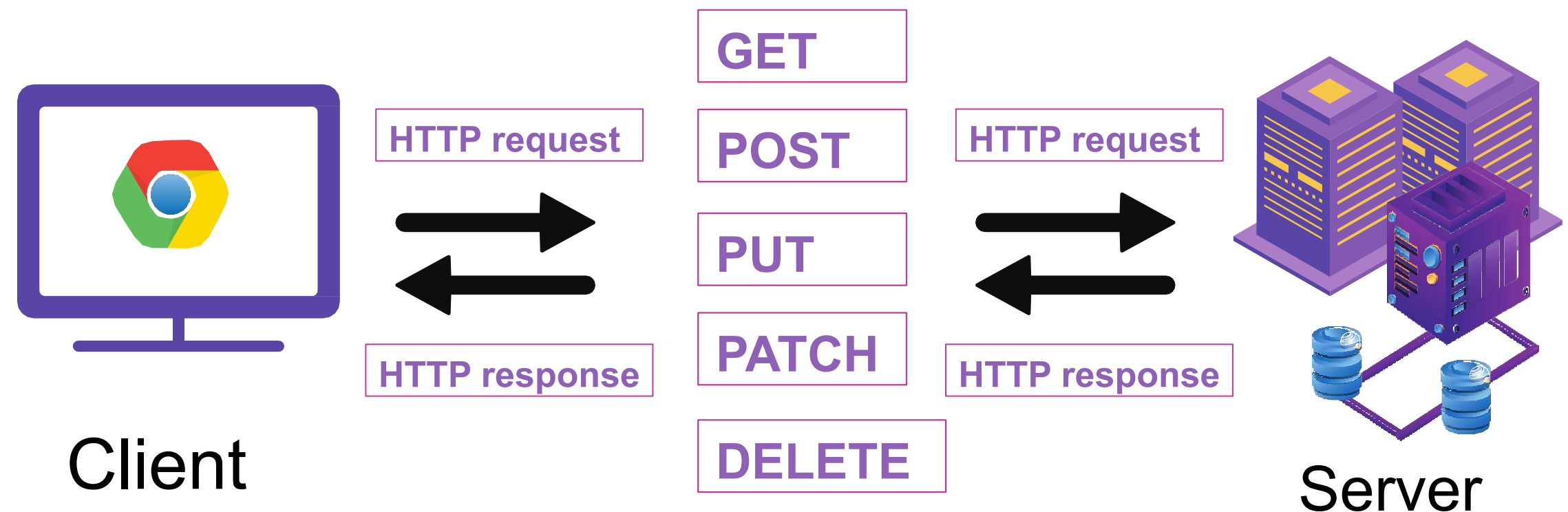
código	tipo de resposta
100-199	informação
200-299	sucesso
300-399	redirecionamento
400-499	erro do cliente
500-599	erro de servidor

{ reprograma }

# 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 respondê-lo com um **response**.



# HTTP - CRUD

CRUD é a composição da primeira letra de 4 operações básicas de um banco de dados, e são o que a maioria das aplicação faz

- ✅ 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

Cada um deles corresponde a uma ação real no banco de dados.

GET	ler
POST	criar
PUT	substituir
PATCH	modificar
DELETE	excluir

# 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, etc.

# Web API e API REST

Web API é uma interface que é disponibilizada de forma remota, pela web, que possibilita a programação aplicativos e softwares.

E as APIs RESTfull são aquelas que são capazes de fazer o REST. Que nada mais é uma API que usa os protocolos HTTP para comunicação entre o usuário e o servidor.



{ reprograma }

# Node.js

O JavaScript do lado do servidor

Interpretador JavaScript que não precisa de navegador.

Ele pode:

Ler e escrever arquivos no seu computador

Conectar com um banco de dados

Se comportar como um servidor



{ reprograma }

## **iniciando o projeto**

npm init

## **instalando as dependências**

npm install NOME-DA-DEPENDÊNCIA

## **iniciando o servidor**

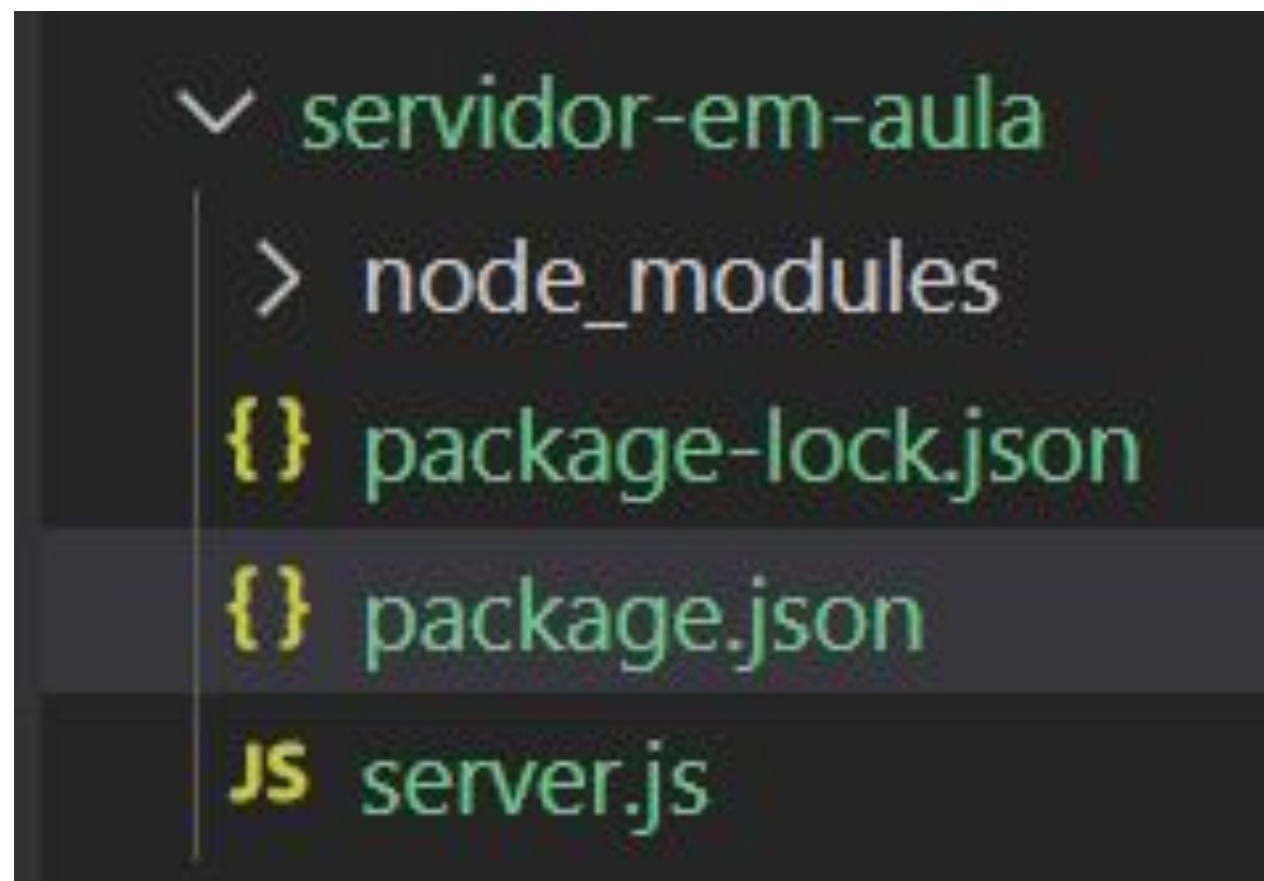
npm start

## **instalando dependencias de um servidor já iniciado**

npm install

{ reprograma }

# Pacotes



Sempre que iniciamos um servidor com Node.js o resumo do seu projeto ficara no **package.json**

Quando instalamos uma dependência pelo npm, ele será referenciado no **package-lock.json** e será instalado na pasta **node\_modules**.



{ reprograma }

# Package.json

O arquivo package.json é o ponto de partida de qualquer projeto NodeJS. Ele é responsável pela descrição do projeto, indicação das dependências de desenvolvimento, etc

# Package-lock.json

O package-lock especifica a versão e suas dependências próprias, assim, a instalação criada será sempre a mesma, toda vez.

# node\_modules

Na node\_modules estarão baixadas as dependências que o seus pacotes precisarão pra funcionar

# .gitignore

Devemos ignorar a node\_modules, pois ela é muito pesada e desnecessária no versionamento, como git.

Se apagarmos a node\_modules ou clonarmos um projeto que já possui package.json e package-lock.json, ou seja, já foi inicializado, só precisamos dar o comando **npm install** que as dependências serão baixadas de novo e pasta node\_modules reaparecerá.

{ reprograma }

# Express


npm install express

# nodemon

npm install nodemon

**express**  
4.17.1 • Public • Published a year ago

[Readme](#) [Explore](#) [30 Dependencies](#) [46.033 Dependents](#) [264 Versions](#)



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()
```

Install  
`> npm i express`

± Weekly Downloads  
**13.961.907**


Version	License
4.17.1	MIT

Unpacked Size	Total Files
	16

Pull Requests  
52

**nodemon**  
2.0.4 • Public • Published 4 months ago

[Readme](#) [Explore](#) [10 Dependencies](#) [2.477 Dependents](#) [215 Versions](#)



Install  
`> npm i nodemon`

[Fund this package](#)

± Weekly Downloads  
**2.893.116**

Version	License
2.0.4	MIT

Unpacked Size	Total Files
107 kB	43

**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.



{ reprograma }

# cors

npm install cors

**CORS (Cross-Origin Resource Sharing)** é uma especificação permite que um site acesse recursos de outro site mesmo estando em domínios diferentes.

Os navegadores fazem uso de uma funcionalidade de segurança chamada **Same-Origin Policy**: um recurso de um site só pode ser chamado por outro site se os 2 sites estiverem sob o mesmo domínio.

Isso porque o navegador considera recursos do mesmo domínio somente aqueles que usam o **mesmo protocolo** (http ou https), a mesma **porta** e o **mesmo endereço**

## cors

2.8.5 • Public • Published 2 years ago

Readme

Explore BETA

2 Dependencies

7.113 Dependents

34 Versions

## cors

npm v2.8.5 downloads 17M/month build passing coverage 100%

CORS is a node.js package for providing a **Connect/Express** middleware that can be used to enable **CORS** with various options.

Follow me (@troygoode) on Twitter!

- Installation
- Usage

### Install

```
> npm i cors
```

± Weekly Downloads

3.985.079

Version

2.8.5

License

MIT

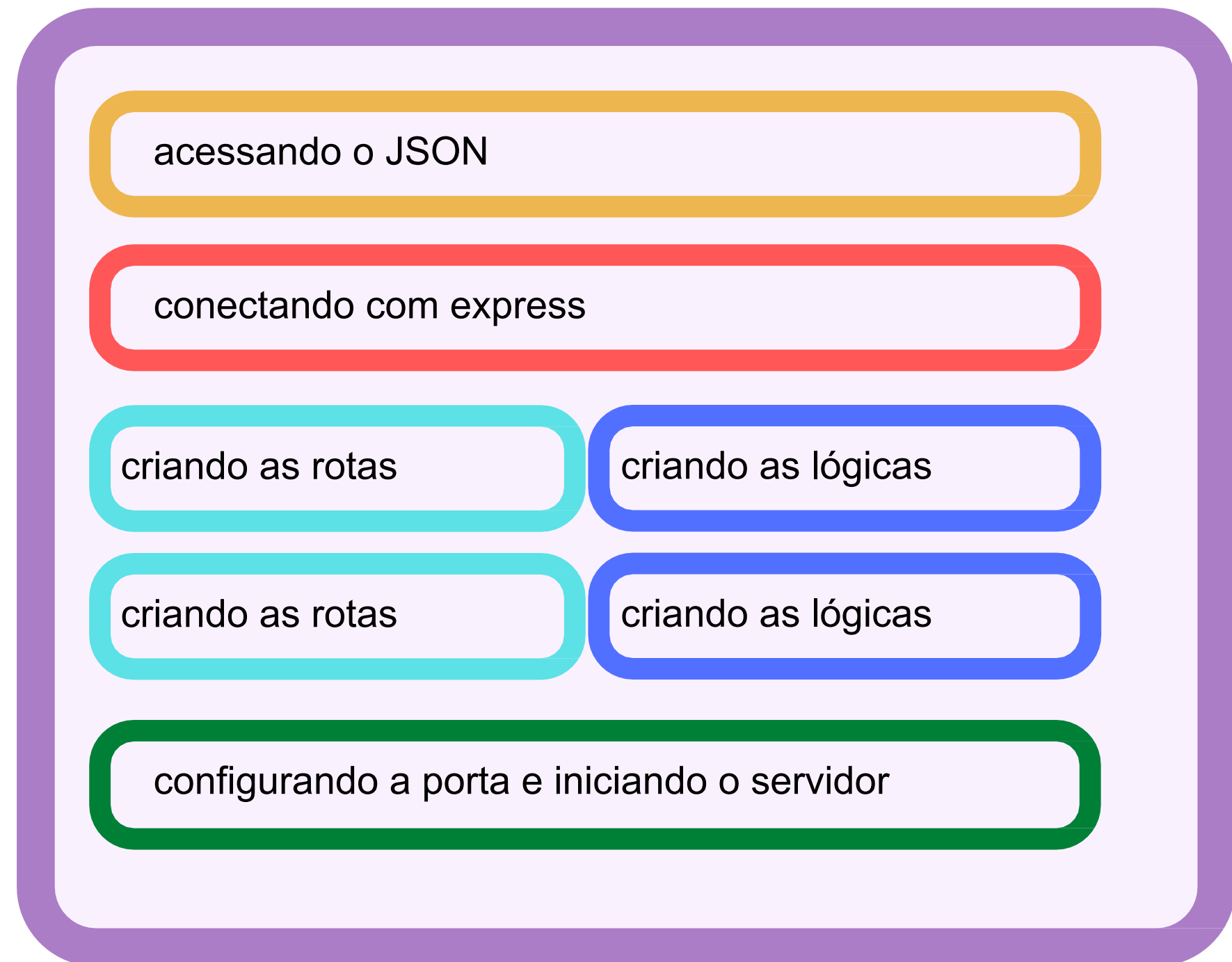
Pensando assim, o front-end e o back-end deveriam estar no mesmo Servidor e na mesma camada, o que não acontece!

Para resolver esse problema usamos o CORS

# Arquitetura

Vamos começar a organizar isso aqui

Até hoje nós estávamos fazendo tudo dentro de um arquivo só, o `server.js`

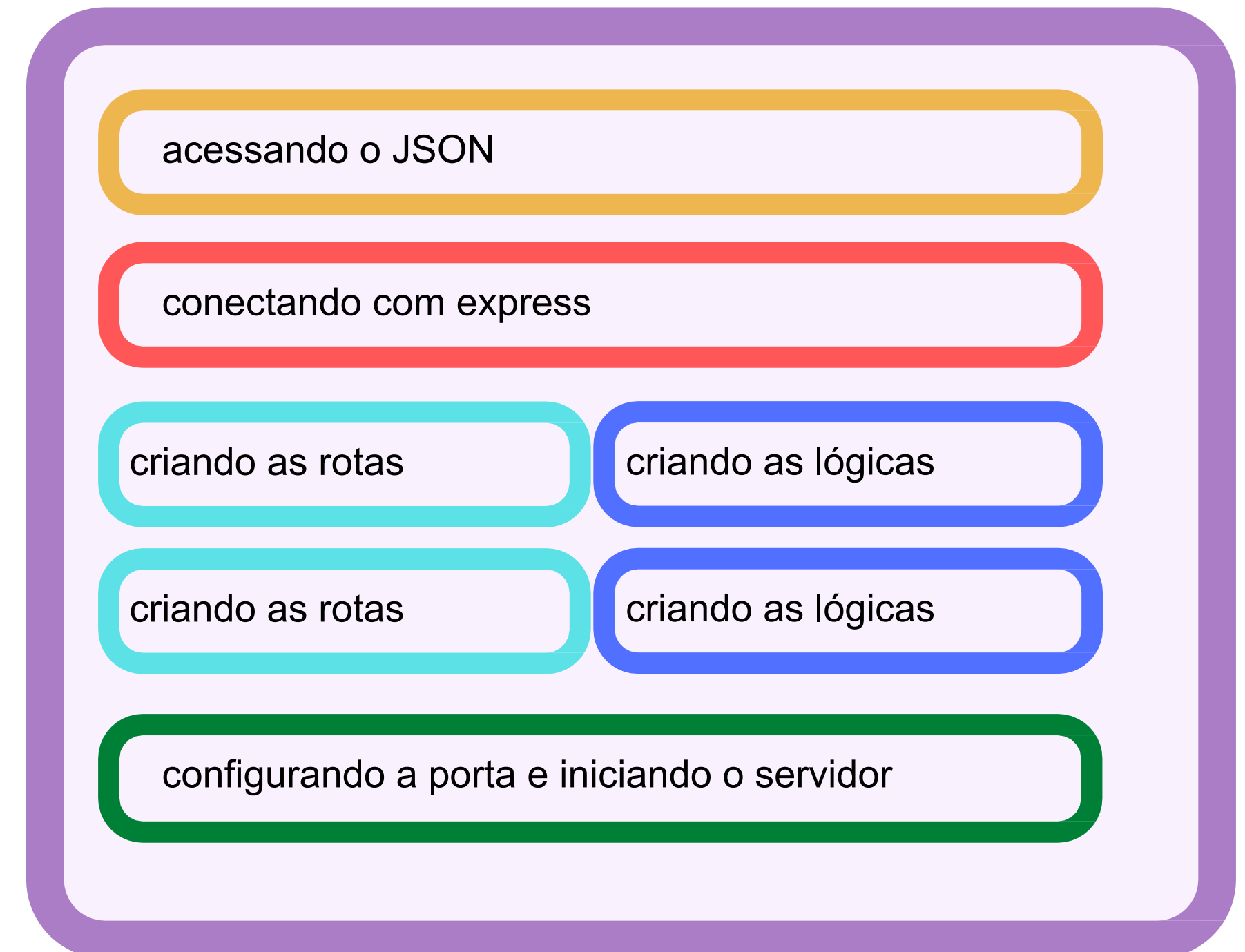


# Arquitetura - MVC

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 lidaremos com as **rotas (routes)**.

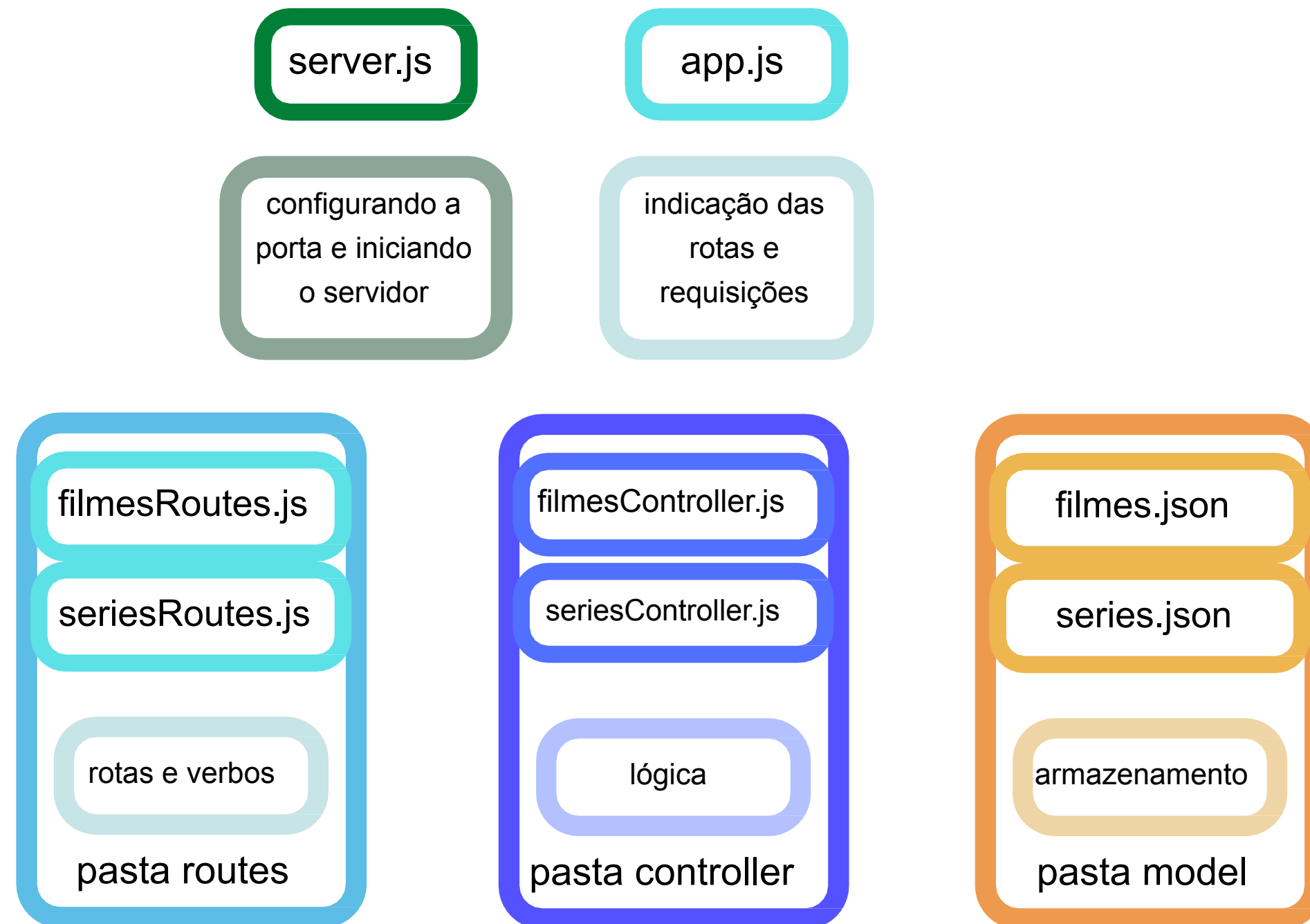
O MVC nada mais é que uma forma de **organizar** o nosso código.





{ reprograma }

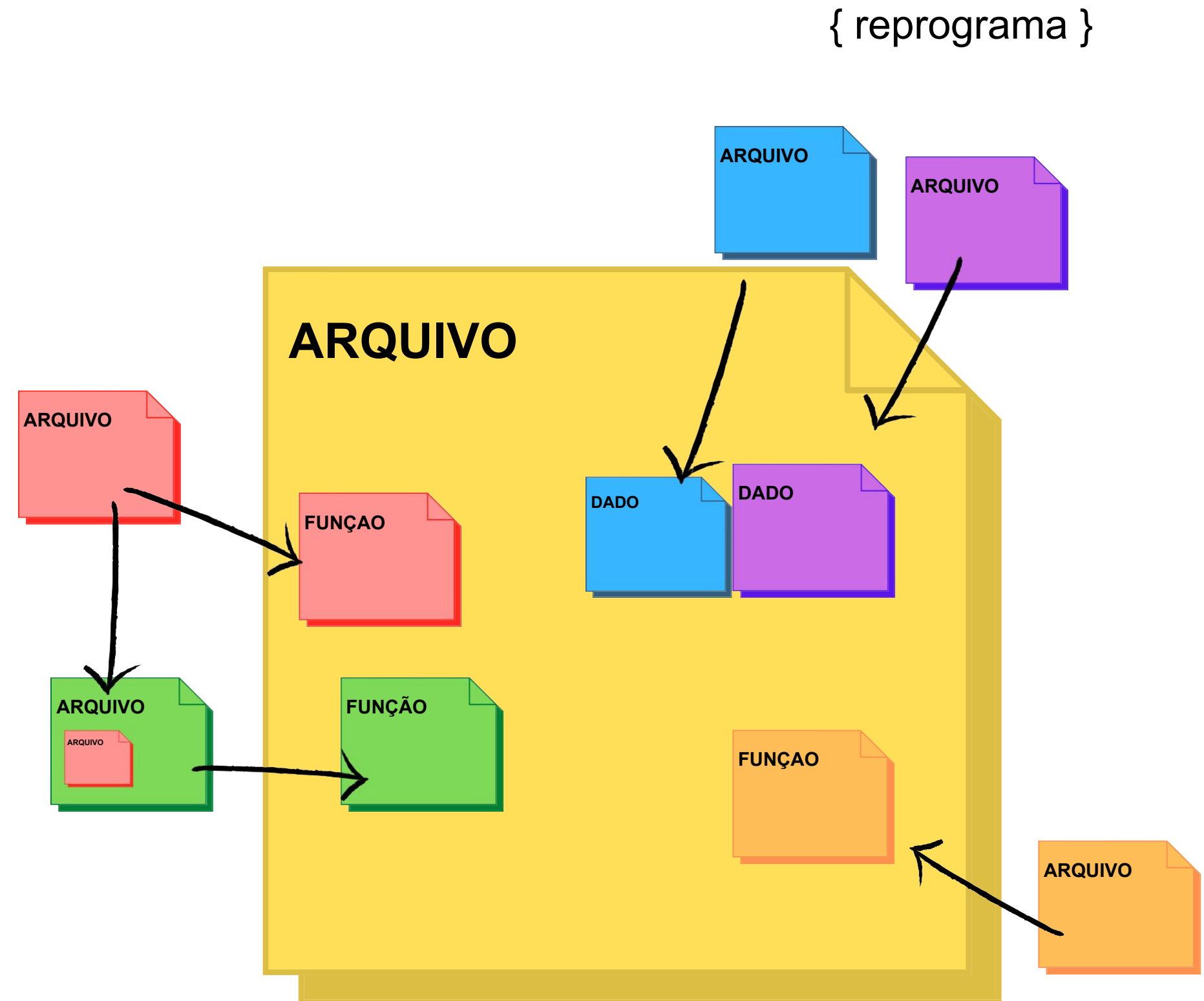
# Arquitetura - MVC



```
-- 📁 NOME-DO-SEU-SERVIDOR
| .gitignore
| package-lock.json
| package.json
| server.js
|-- 📁 node_modules
|-- 📁 src
|   | app.js
|   |
|   | 📁 ---controller
|   |   | NOMEController.js
|   |
|   | 📁 ---model
|   |   | NOME.json
|   |
|   | 📁 ---routes
|   |   | NOMERoute.js
```

# Mas vai ser sempre assim?

Não! Quanto maior a aplicação sua arquitetura se torna mais complexa, muitas vezes existem mais camadas de organização e comunicações entre arquivos. E apesar de usarem o MVC acabam ficando bem maiores e mais setorizadas



{ reprograma }

# HTTP - GET

Dentro do CRUD o método GET é representado pela letra R

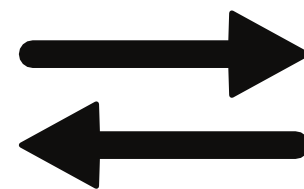
👁️ R: **Read** (ler) - exibir as informações de um registro

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



Client

HTTP request



HTTP response

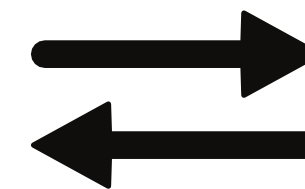
GET por id

GET todos

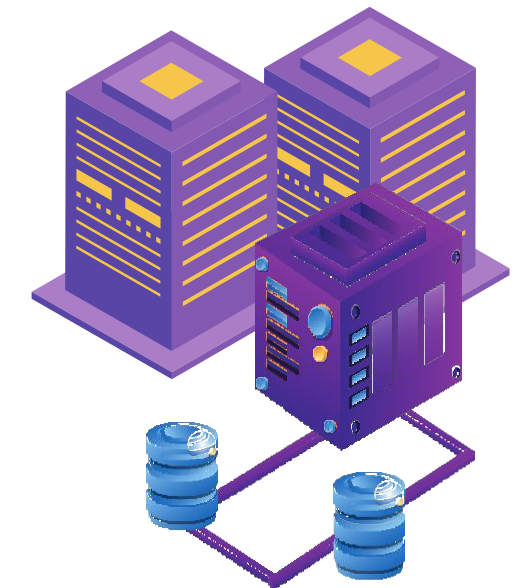
GET por nome

GET por gênero

HTTP request



HTTP response



Server

# Parâmetros

## Path params

- são aqueles que são adicionados diretamente na URL
- `/rota/:id`
- `request.params.id` bons,
- porém limitantes, por exemplo, se quisermos filtrar por uma string

## Query params

- são aqueles que são adicionados a chave e o valor desejados
- `/rota?id=1234`
- `request.query.id`
- a forma mais efetiva de fazer requests com strings ou diversos valores

{ reprograma }

# HTTP - POST & DELETE

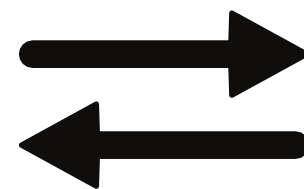
✓ C: **Create** (criar) - criar um novo registro

✗ D: **Delete** (apagar) - apagar um registro



Client

HTTP request



HTTP response

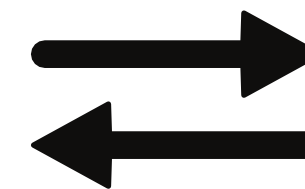
POST

criar

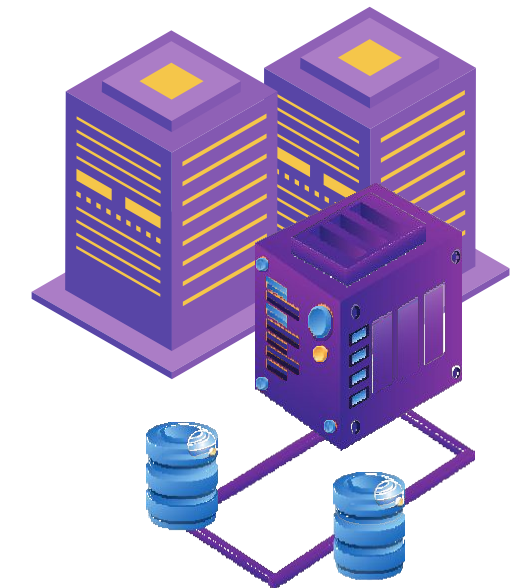
DELETE

excluir

HTTP request



HTTP response



Server

{ reprograma }

# Body

- são usados nos métodos POST, PATCH e PUT
- enviam dados a serem cadastrados no banco de dados
- request.body

```
{  
  "descricao": "exemplo de Body",  
  "nomeColaborador": "Ana"  
}
```

{ reprograma }

## Parâmetros

Tanto o body quando o query e o path são parâmetros enviados na requisição e podem ser acessados pelo servidor para definir a requisição e as ações.

## request.params

usado para pesquisa simples, enviado diretamente na rota

## request.query

usado para pesquisa de uma ou múltiplas strings

## request.body

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

# Body Parse

```
app.use(express.json())
```

Quando recebemos um request os dados do body são enviados de uma forma que não conseguimos facilmente acessar e manipular.

Por isso, devemos "parsear" o body: essa função analisa e transforma num json manipulavel



{ reprograma }

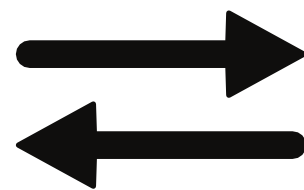
# HTTP - PUT & PATCH

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



Client

HTTP request



HTTP response

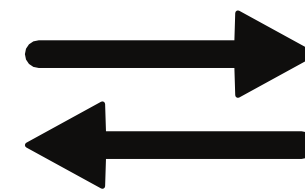
PUT

substituir

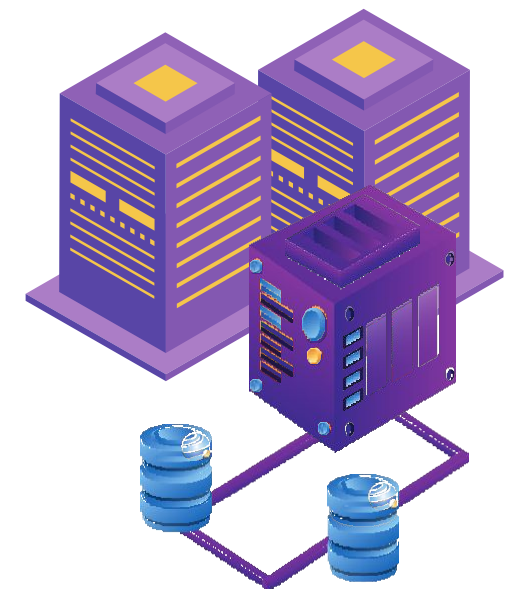
PATCH

modificar

HTTP request



HTTP response



Server

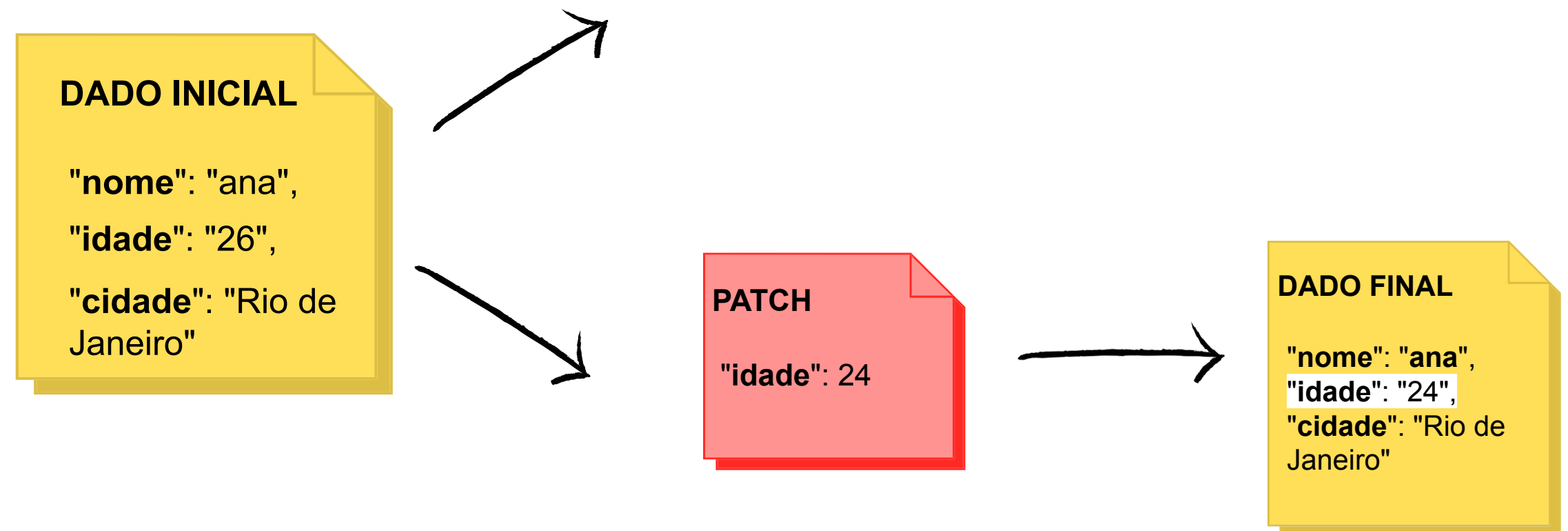
{ reprograma }

# PUT x PATCH

É tudo a mesma coisa?

**NÃO!**

O PUT substitui todo o objeto que você deseja modificar, já o PATCH modifica somente uma propriedade dentro do seu objeto.



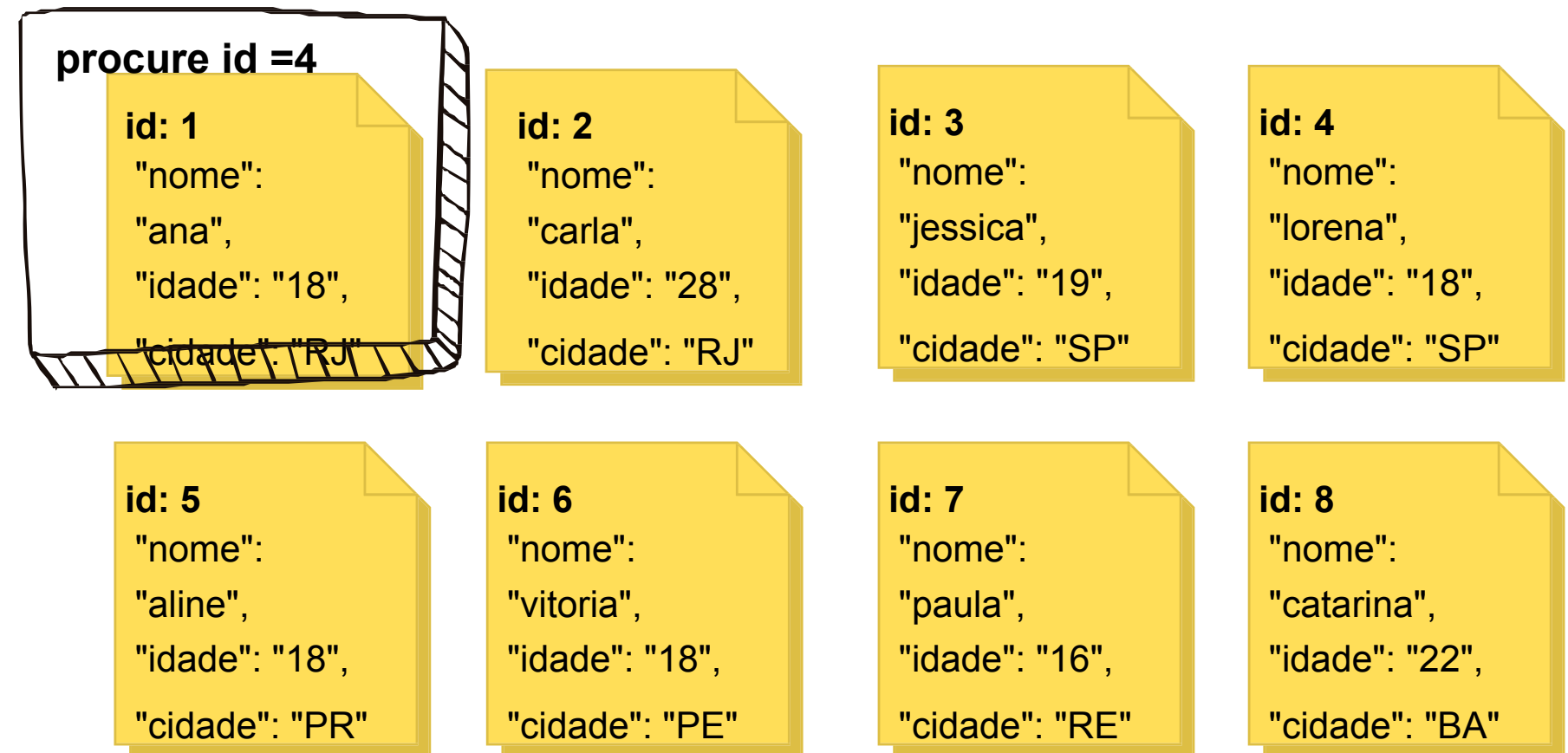
{ reprograma }

# PUT x PATCH

Mas então por que ainda usamos o PUT?

Muitas vezes ainda usamos o PUT pela performance que ele tem quando relacionado o banco de dados. Substituir um dado inteiro é mais rápido do que somente uma propriedade dele.

Por exemplo, vamos simular a uma edição do campo idade no dado de id=4



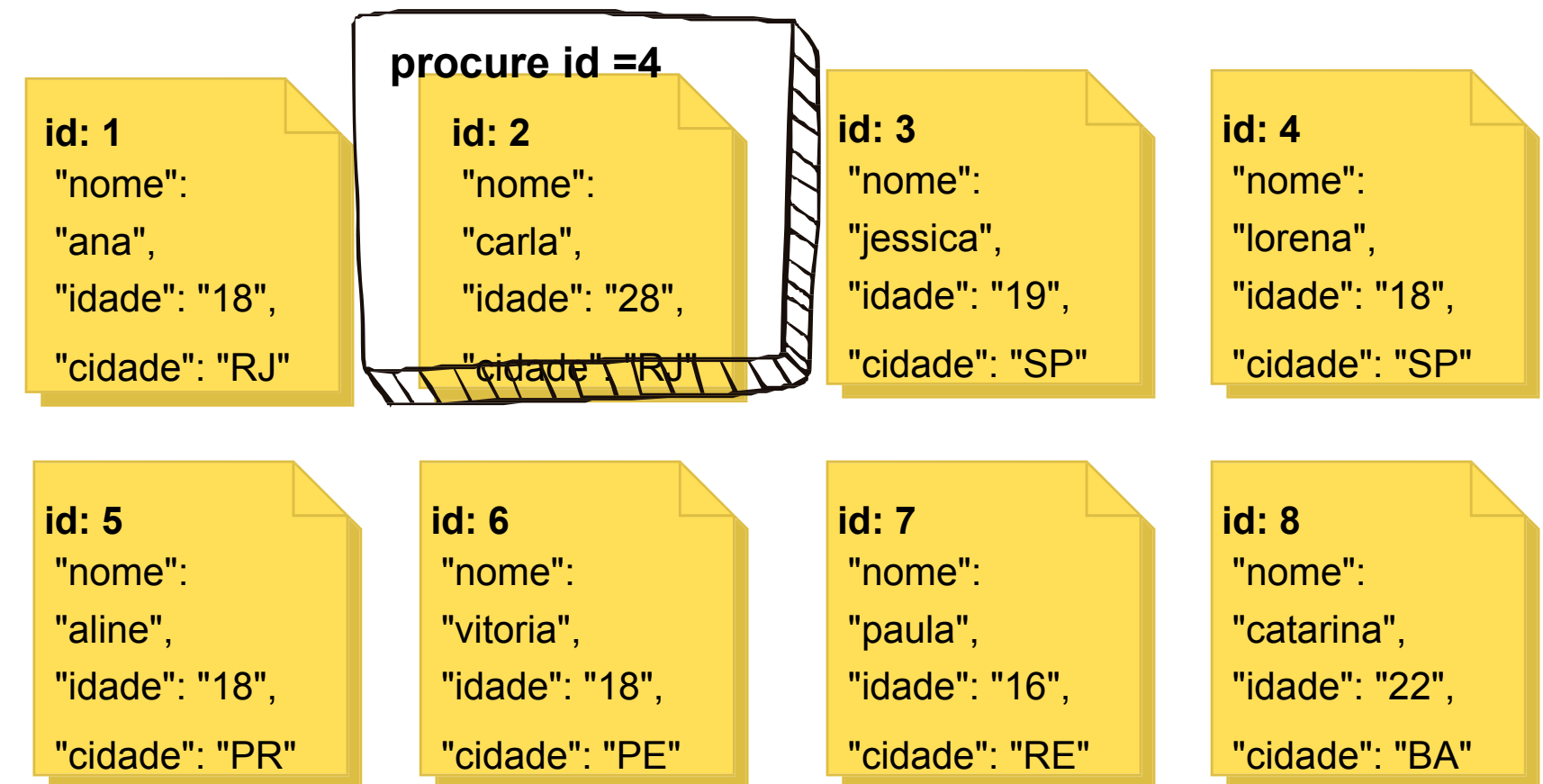
{ reprograma }

# PUT x PATCH

Mas então por que ainda usamos o PUT?

Por exemplo, vamos simular a uma edição do campo idade no dado de id=4.

No banco de dados nosso programa tem que percorrer pela memória procurando pelo id que queremos



{ reprograma }

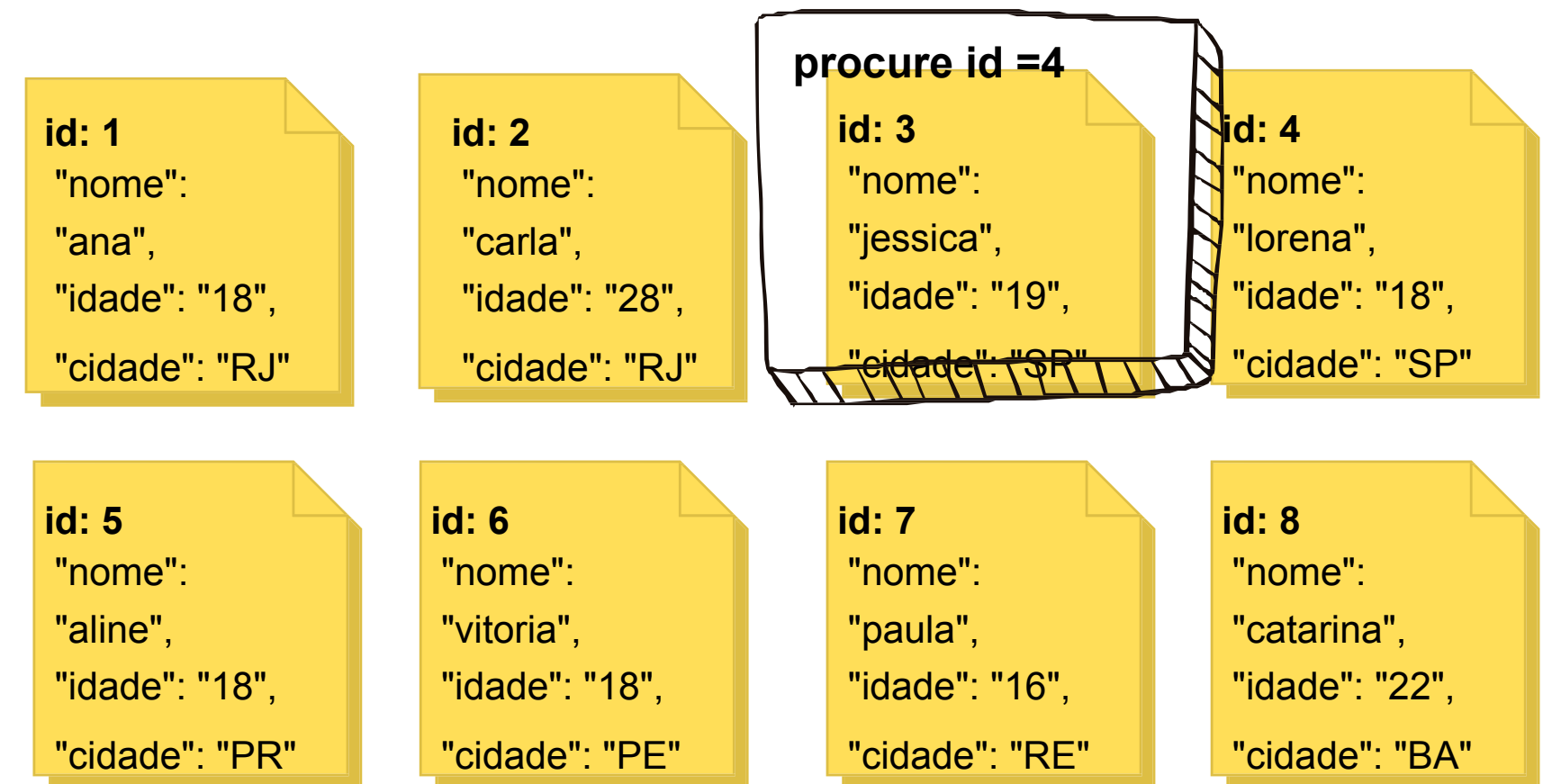
# PUT x PATCH

Mas então por que ainda usamos o PUT?

Por exemplo, vamos simular a uma edição do campo idade no dado de id=4.

No banco de dados nosso programa tem que percorrer pela memória procurando pelo id que queremos

Ele procura somente pelo índice que indicamos.

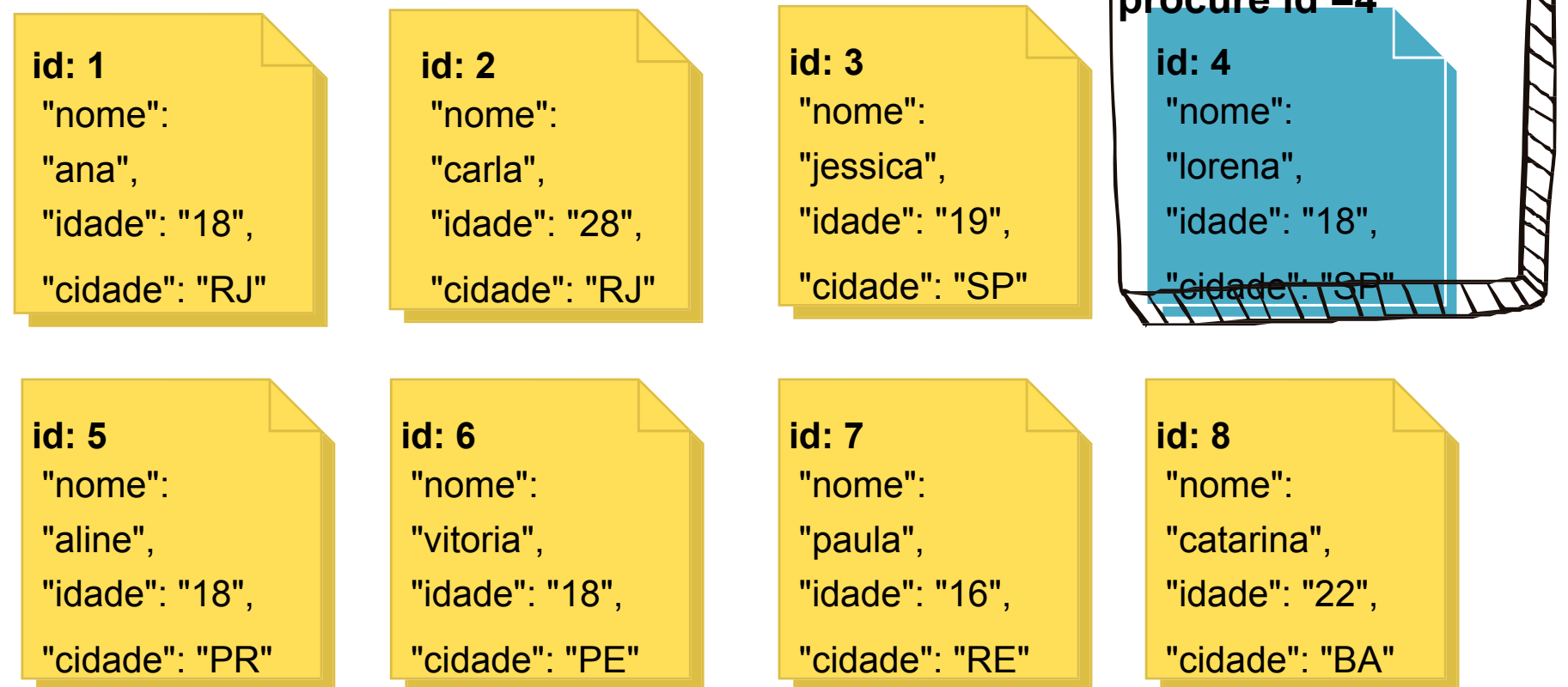


# PUT x PATCH

Mas então por que ainda usamos o PUT?

Se estivéssemos escolhido um método PUT, a procura pararia aqui e o dado seria substituído por inteiro!

{ reprograma }



{ reprograma }

# PUT x PATCH

Mas então por que ainda usamos o PUT?

Porém estivéssemos escolhido um método **PATCH**, a procura continuaria, agora dentro do dado

**procure id =4**

**id:**

**1"nome":**

idade

"lorena",

"idade": 18,

"cidade": "SP"

{ reprograma }

# PUT x PATCH

Mas então por que ainda usamos o PUT?

Porém estivéssemos escolhido um método **PATCH**, a procura continuaria, agora dentro do dado. Quando encontrado a propriedade, o dado seria modificado.

Tudo isso seriam frações de segundos para um computador, mas se tivéssemos dezenas de milhares de dados sendo modificados o tempo todo, como uma rede social, por exemplo, isso poderia causar certa lentidão no banco de dados

**procure id =4**

```
id:  
1'nome':  
"lorena",  
"idade": 18,  
"cidade": "SP"
```



{ reprograma }

# blog

da Reprograma

# Demandas da API

Lá vem a galera de negócio....

- devo conseguir ver todos os post
- devo conseguir um post especifico
- devo conseguir deletar post
- devo conseguir atualizar post
- devo conseguir atualizar titulo do post
- devo conseguir atualizar qualquer parte do post separadamente

{ Reprograma }

# **Vamos continuar!**

{ reprograma }



**Para casa**

