

HTTP GET e HTTP POST

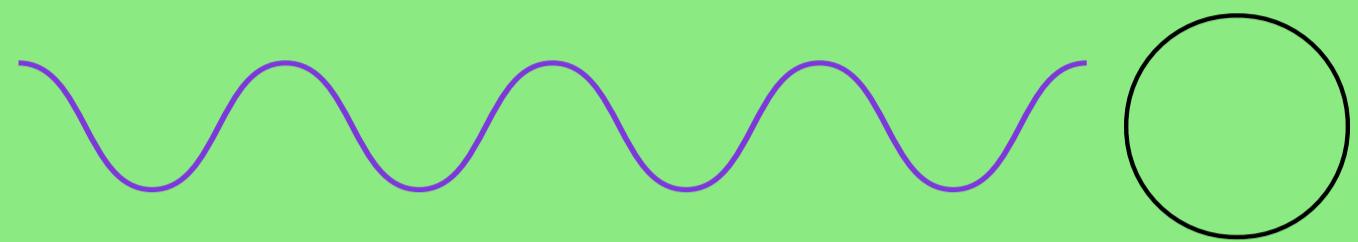
Node.js

Objetivos de Aprendizagem

**Ao final deste módulo,
esperamos que você seja capaz de:**

- Entender os fundamentos de estilos arquitetônicos como REST e SOAP;
- Conhecer a ferramenta Docker;
- Entender sobre autenticação com JWT.





Conceitos de **REST**

O que é REST?

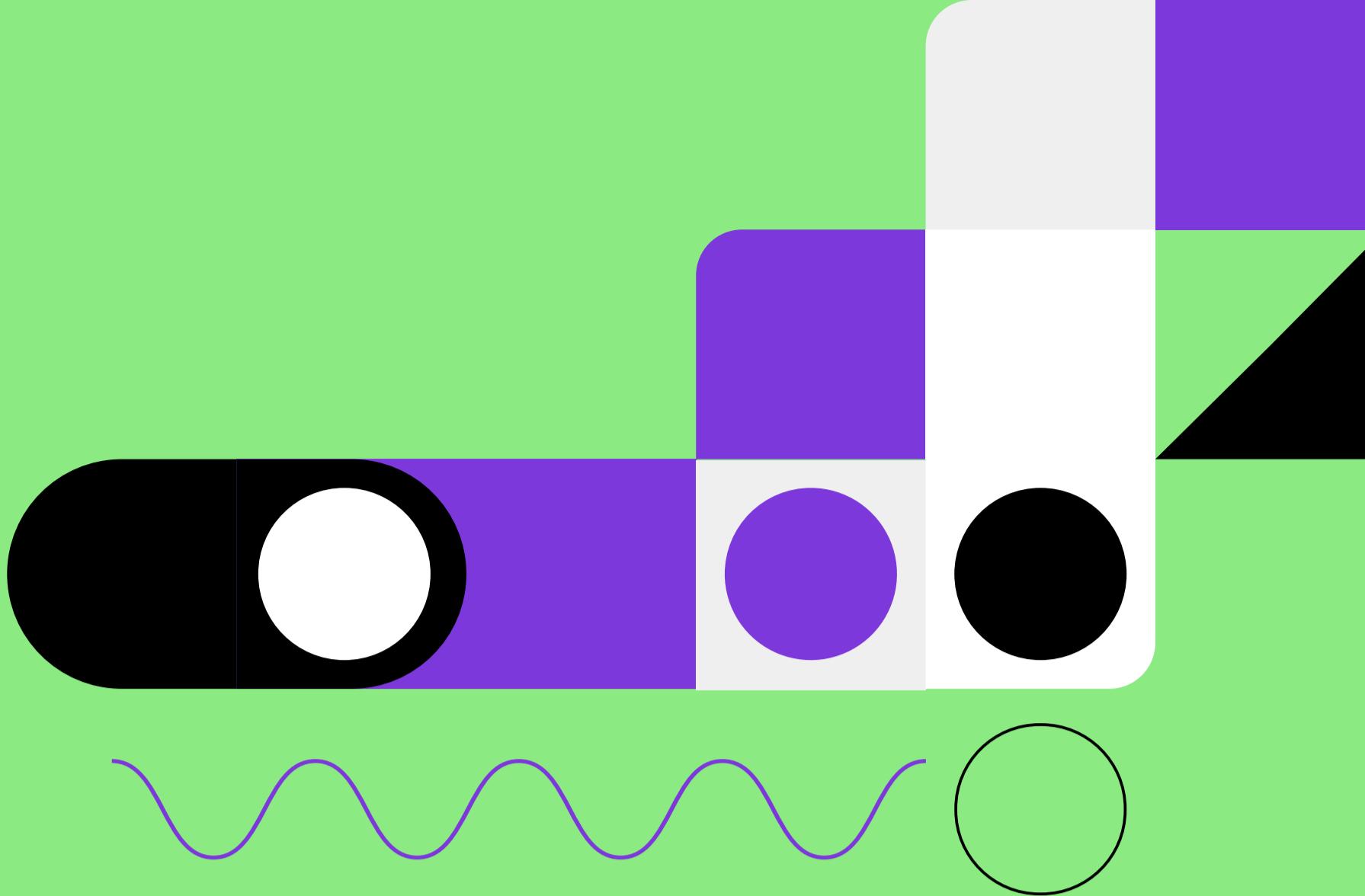
É um estilo arquitetônico que define um conjunto de recomendações para o design de aplicações que usam o protocolo HTTP para transmissão de dados.

Previamente...

Antes de entendermos porque existe o padrão arquitetônico REST, vamos entender sobre o **protocolo HTTP**.

Antes de tudo, é importante ressaltar que tudo na internet é baseado em "pergunta" (**requisição, request**) e "resposta" (**response**).





Protocolo **HTTP**



A principal função do protocolo HTTP é a transmissão de páginas web na Internet.

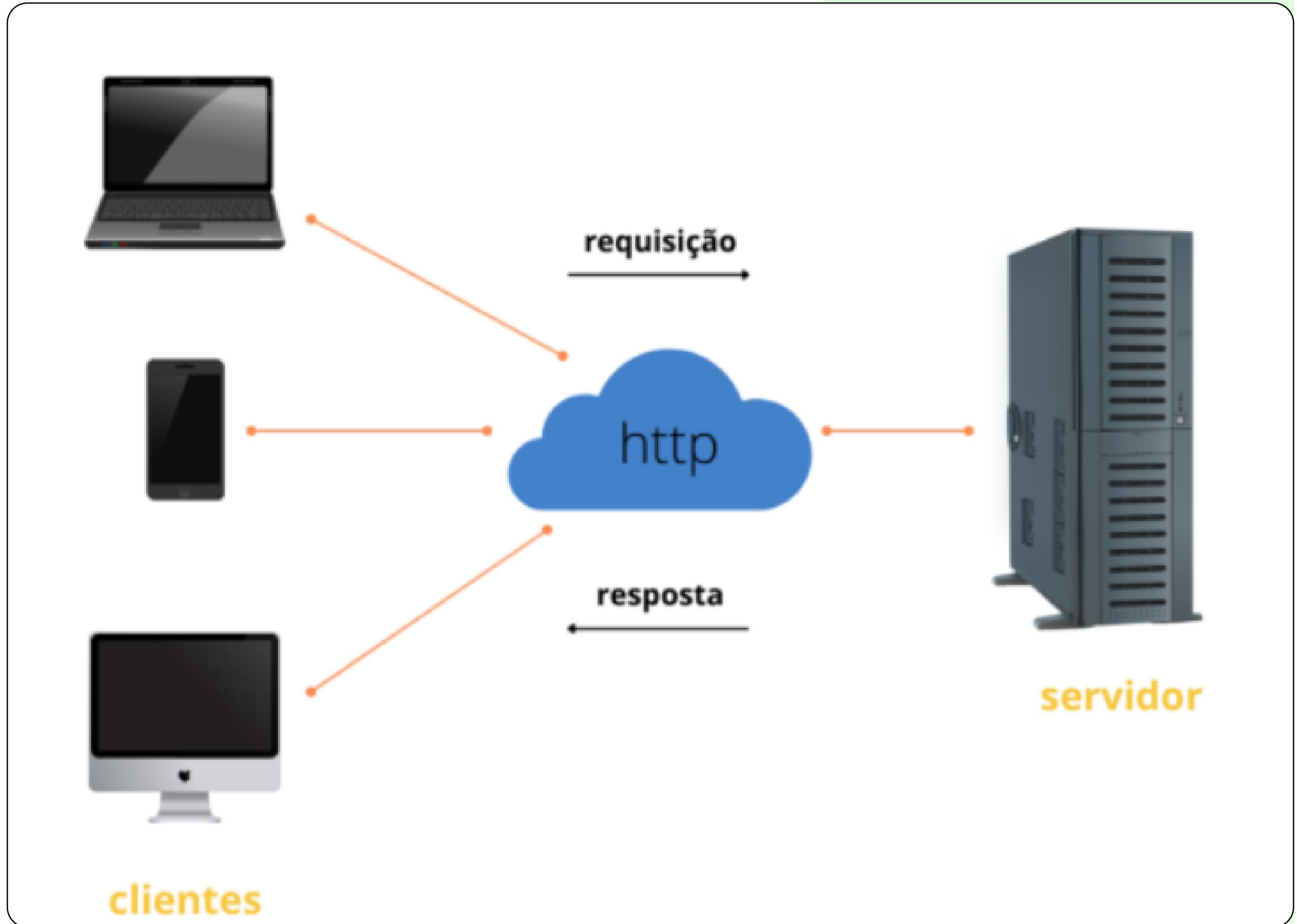
O Protocolo de Transferência de Hipertexto (protocolo HTTP) é usado principalmente em redes baseadas em IP para a transmissão de páginas web de um servidor para o navegador. Ele funciona sem criptografia e não se limita a aplicativos.

O que é **HTTP**?

A abreviatura de “Hypertext Transfer Protocol” **descreve um protocolo sem estado com o qual os dados podem ser transmitidos** em uma rede IP. A aplicação mais importante é a **transferência** de páginas da Internet e dados entre um servidor web e um navegador.

Como funciona o **HTTP**?

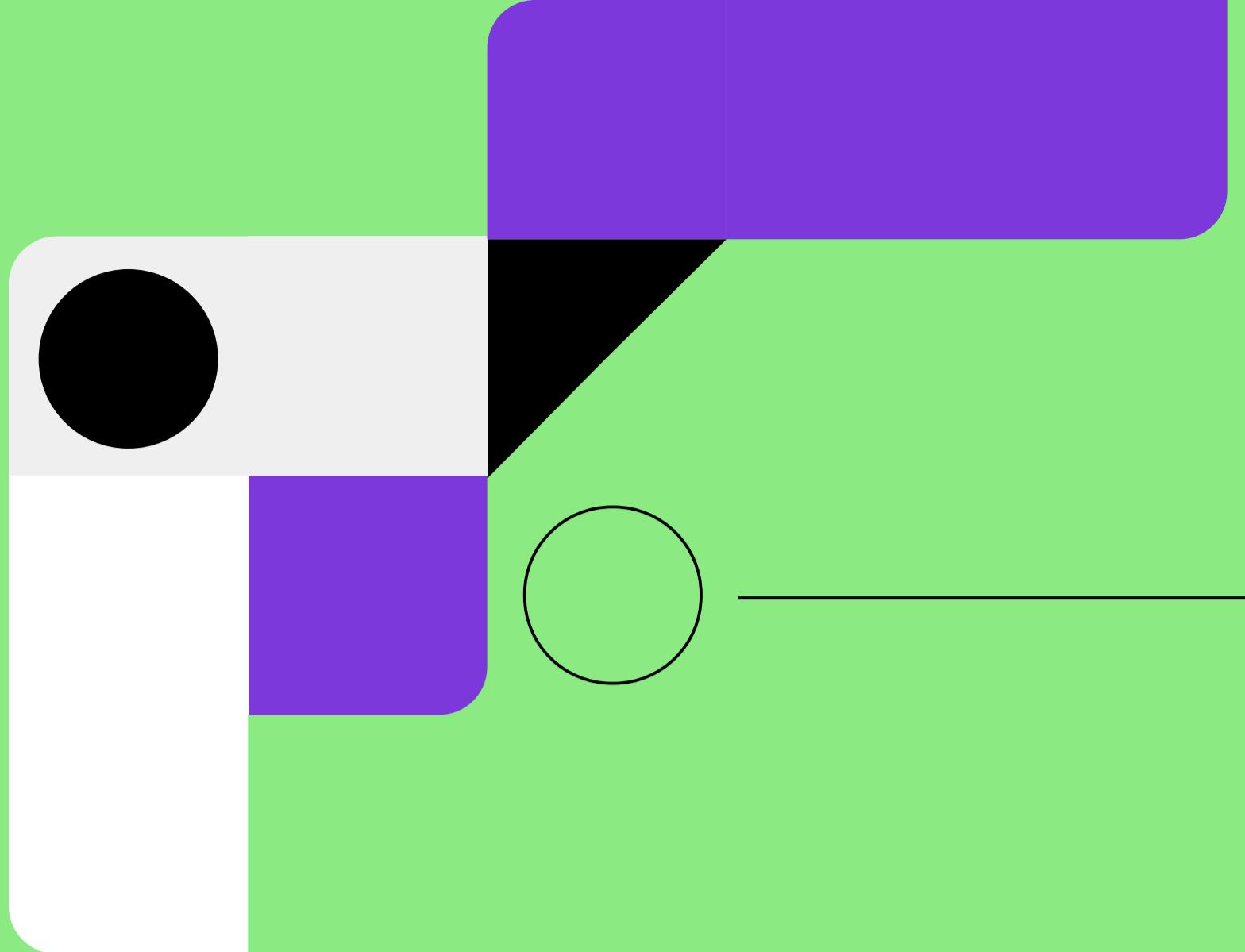
O HTTP existe desde o início dos anos 1990 e é um dos padrões básicos em que se baseia o funcionamento da Internet. Uma unidade de comunicação em **HTTP** é **chamada de mensagem**, sendo feita uma distinção entre solicitação ou consulta e resposta. O significado é óbvio: a solicitação é feita do cliente ao servidor web, enquanto a resposta se refere à ação subsequente do servidor ao cliente.



- Como vemos, o protocolo
HTTP fica no intermédio da
requisição e resposta.

A maneira **como** o **HTTP** **funciona** é a seguinte:

- O cliente inicia uma solicitação ao servidor web digitando-a na barra de endereço do navegador;
- Este, por sua vez, responde inicialmente com um código de status, que consiste em três dígitos;
- Ele contém a informação se a solicitação pode ser respondida com sucesso ou não;
- Todo usuário já viu esse código de status antes: se houver um erro na solicitação, a mensagem 404 aparece, por exemplo;
- Não apenas o conteúdo HTML pode ser transmitido via HTTP, mas também outros formatos;
- Quando fazemos um consumo (requisição) de uma API, para fazer isso utilizamos métodos HTTP (GET, POST, PUT, etc);
- Quando estamos navegando na web, a todo momento o nosso navegador está enviando requisições para um servidor e o servidor, por sua vez, nos devolve uma resposta em um formato específico ou realiza uma ação de acordo com o que pedirmos para ele fazer;
- Nas requisições, especificamos o que chamamos de método HTTP ou verbo. Na versão 1.1 do protocolo HTTP (que é a que todos usamos atualmente) temos 9 verbos diferentes.



Métodos (Verbos) **HTTP**

GET

Essa é a requisição mais comum de todas. Através dessa requisição nós pedimos a representação de um recurso: que pode ser um arquivo html, xml, json, etc.

POST

O método POST é utilizado quando queremos criar um recurso. Quando usamos POST, os dados vão no corpo da requisição e não na URI.

PUT

Requisita que um recurso seja "guardado" na URI fornecida. Se o recurso já existir, ele deve ser atualizado. Se não existir, pode ser criado.

DELETE

Exclui o recurso especificado.

TRACE

Devolve a mesma requisição que for enviada veja se houve mudança e/ou adições feitas por servidores intermediários.

OPTIONS

Retorna os métodos HTTP suportados pelo servidor para a URL especificada.

PATCH

Serve para atualizar partes de um recurso, e não o recurso todo.

CONNECT

Converte a requisição de conexão para um túnel TCP/IP transparente, geralmente para facilitar a comunicação criptografada com SSL (HTTPS) através de um proxy HTTP não criptografado.

HEAD

Retorna somente os cabeçalhos de uma resposta.



Idempotência

Alguns métodos, como o GET, podem ser **chamados diversas vezes** seguidas sem problema nenhum: a resposta será sempre a mesma. Isso porque quando fazermos uma requisição do tipo GET não estamos alterando nada no servidor, somente consultando informações.

Já quando estamos fazendo um POST estamos **criando um novo recurso**.

Sabe quando você manda atualizar a página do seu navegador e ele pergunta se você quer realmente atualizar essa página? Que ele vai reenviar as informações e tal? Então, essa página está fazendo uma **requisição do tipo POST**.

Se executarmos a mesma requisição POST duas vezes, corremos o risco de criar dois recursos iguais. Na maioria das vezes não queremos isso né?

Os métodos que não alteram nada no servidor e que podemos chamar várias vezes são o que chamamos de métodos **idempotentes**.

GET, OPTIONS, HEAD, PUT, TRACE, CONNECT e DELETE são idempotentes.

Percebe algo contra intuitivo na frase acima?! Métodos como **PUT** ou **DELETE** são idempotentes?

Por que?

O método DELETE irá gerar uma alteração no servidor.

Mas a questão da idempotência está mais relacionada a **quantas vezes o verbo pode ser chamado e quantas vezes ele fará alterações nos recursos**.

Se você chamou o DELETE uma vez, ele irá apagar o recurso selecionado, caso exista, se você chamar novamente, **ele não irá mais fazer nada, não irá alterar mais os recursos**.

Devido a isso ele é idempotente. Diferentemente do POST, que ao ser chamado cadastraria novos recursos a cada chamada.

Para entendimento do que é Rest e porque ele é importante, vamos fazer um paralelo com o cotidiano:



Se alguém disser "**Mim dá um copo d'água pra mim**" vai ser estranho, certo? A frase está errada.

O mesmo acontece quando é feito um POST em uma API para buscar um recurso. Não soa bem, é estranho.

Em ambos casos o receptor pode entender a mensagem, mas não estão de acordo com o padrão da linguagem.

Resumindo e citando a importância de **ter** **um padrão:**

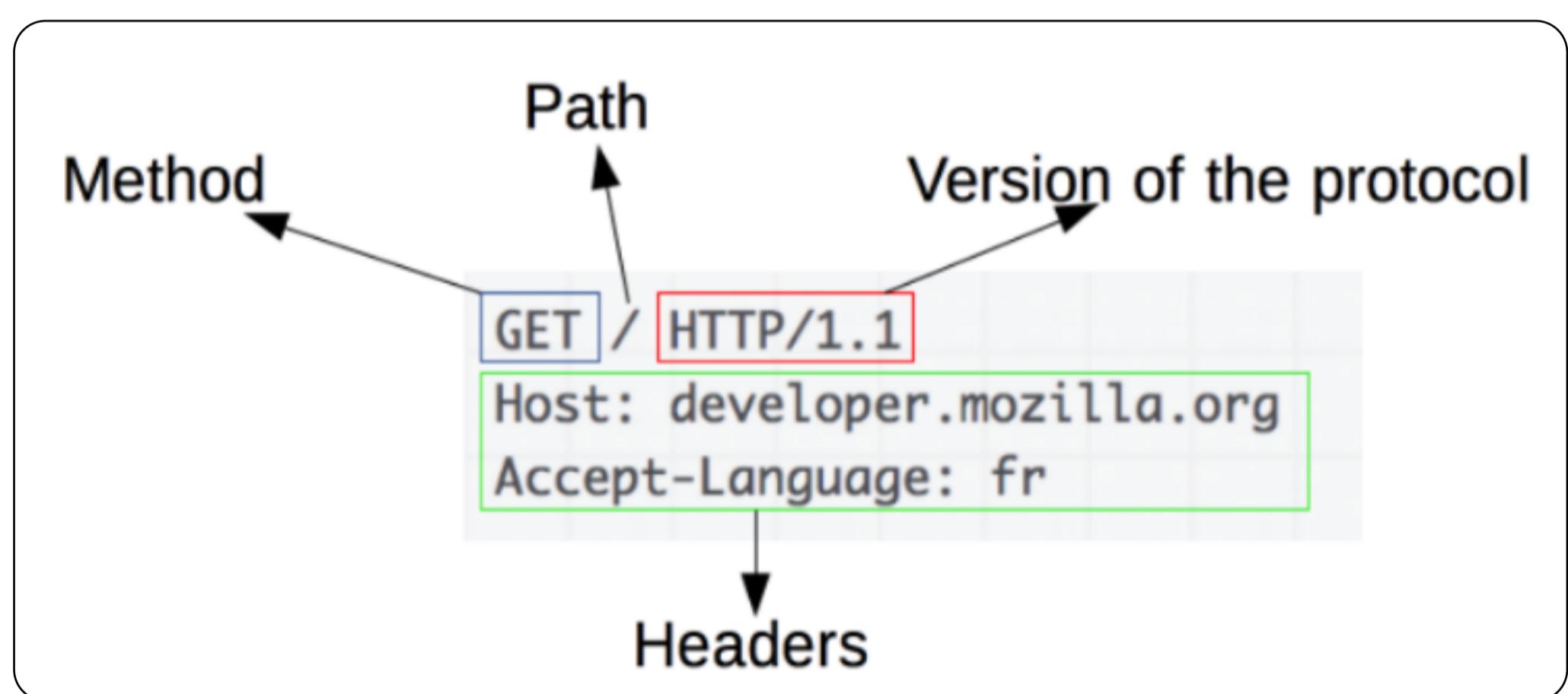
Em linhas gerais e ignorando alguns conceitos. O HTTP assim como o português é um **protocolo de comunicação**. Seguir padrões seus padrões de escrita significa que você se esforça menos para entender.

Em um time, significa menos gasto de energia para resolver um problema. Torna o desenvolvimento ágil e mais eficiente.

HTTP

Uma API e o Browser se comunicam através de mensagens HTTP. Como já foi citado, existem dois tipos de mensagens HTTP, Requests e Responses (**perguntas e respostas**), cada uma com seu próprio formato.

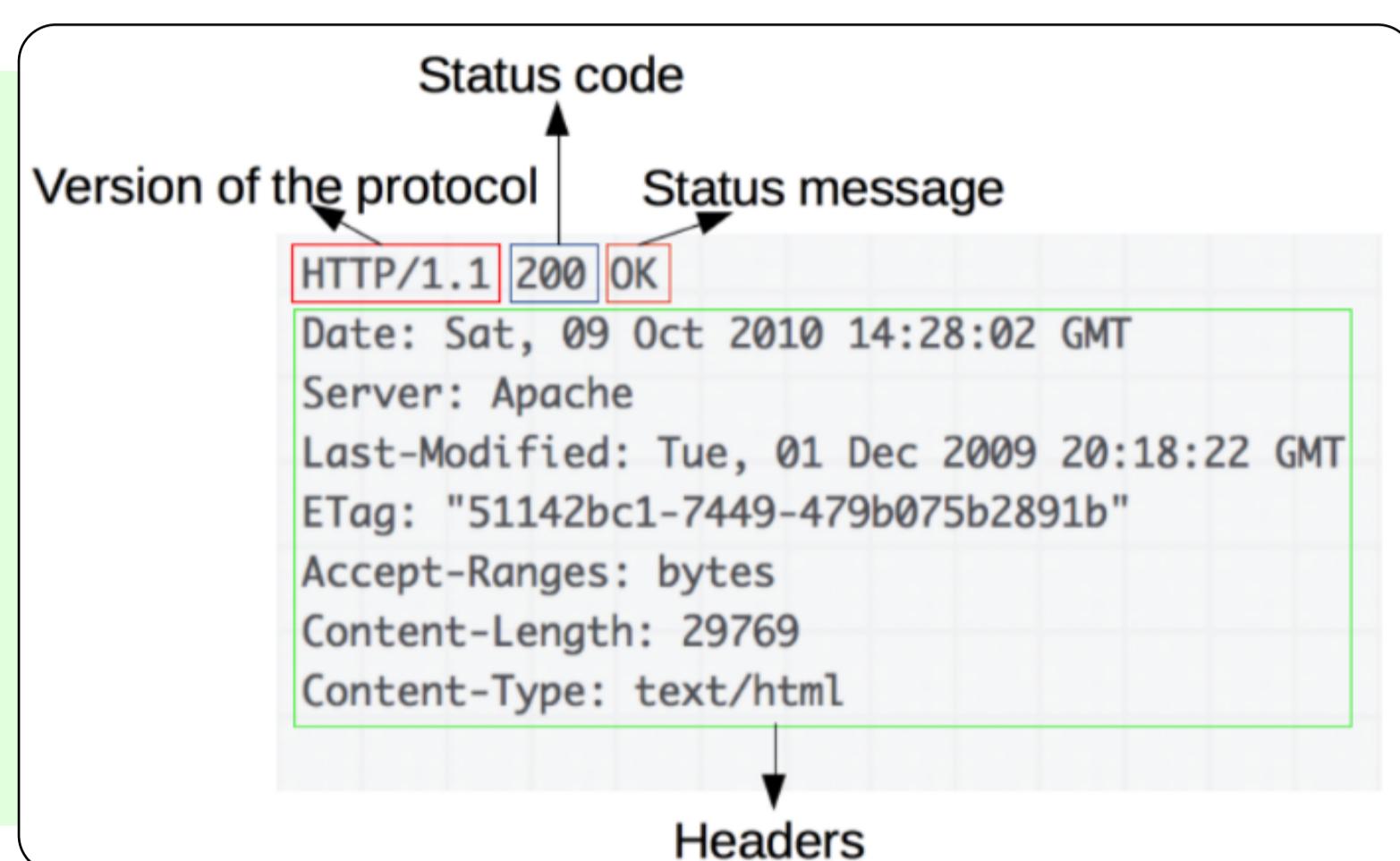
Uma request possui o seguinte formato:



Na imagem é possível observar os seguintes elementos:

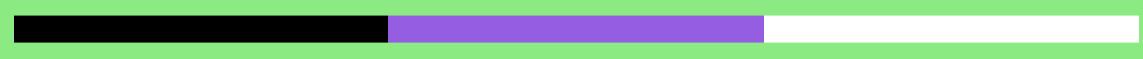
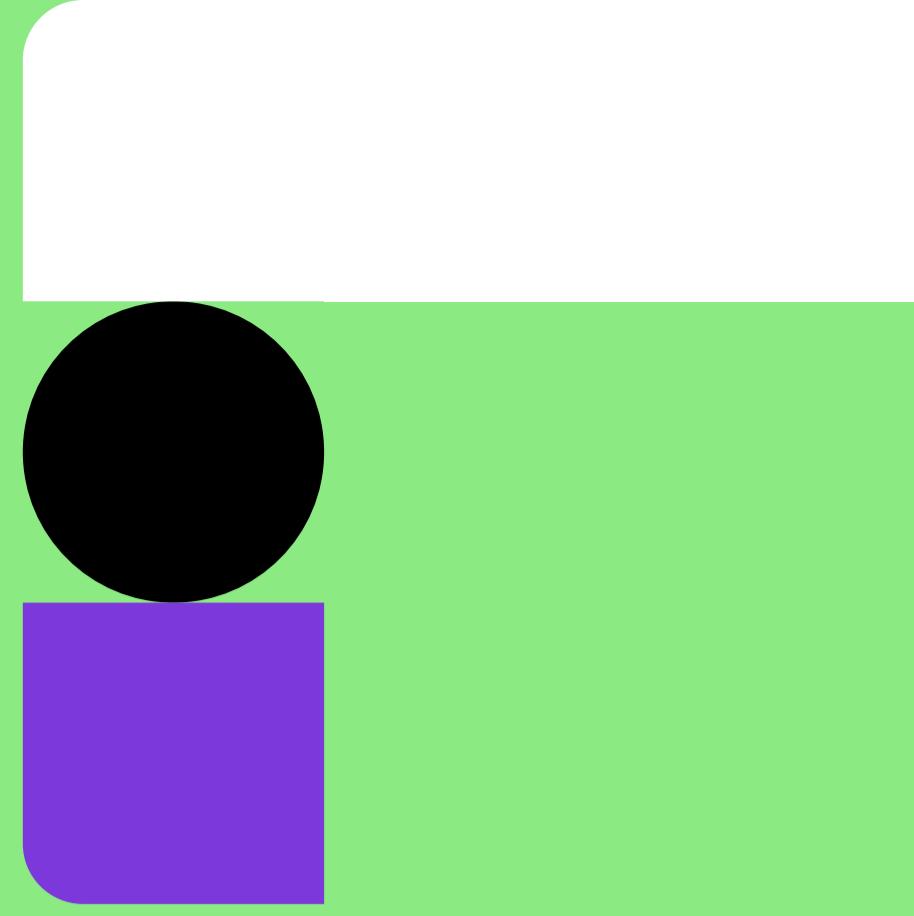
- **Method:** É o método do HTTP. Abaixo há uma tabela com os métodos mais utilizados.
- **Path:** É o endereço do resource que deseja obter.
- **Version of the protocol:** Versão do protocolo HTTP.
- **Headers:** Os headers entregam informações adicionais para o server.
- **Body:** É o conteúdo, geralmente é utilizado em conjunto com os verbos POST, PUT e PATCH.

O HTTP é **connectionless**, isso significa que para toda chamada haverá um RESPONSE. Mesmo que o client que chamou não esteja mais esperando a resposta. Por exemplo, quando o usuário fecha o browser. A response possui o seguinte aspecto:



No response é possível observar:

- A **versão** do protocolo HTTP.
- **Status Code:** Código de status. O código de status é separado por categorias.
 - **2xx:** Status de sucesso
 - **3xx:** Categoria de redirecionamento
 - **4xx:** Erro no Cliente
 - **5xx:** Erro no server
- **Status Message:** A frente do Status Code virá uma breve descrição do que significa o Status Code.
- **Headers:** Haverá informações para o Client, por exemplo o formato da resposta, se o conteúdo pode ser cacheado.
- **Body:** Opcionalmente, dependendo do tipo de request, haverá no body o conteúdo da resposta.



REST

REST é um estilo de arquitetura. **Ele fornece padrões para a comunicação entre sistemas.** REST **não** é um padrão exclusivo para HTTP. Embora as bases do REST e do HTTP sejam as mesmas.

Na arquitetura REST, os clientes enviam solicitações para recuperar ou modificar recursos e os servidores enviam respostas para essas solicitações.

Esse termo foi cunhado por Roy Fielding em sua tese de doutorado. No capítulo cinco em Architectural Styles and the Design of Network-based Software Architectures.



Esse termo foi cunhado por **Roy Fielding** em sua tese de doutorado. No capítulo cinco em Architectural Styles and the Design of Network-based Software Architectures.

REST é o acrônimo de REpresentational State Transfer. Esse padrão é descrito em **seis restrições:**

- Uniform Interface
- Stateless
- Cacheable
- Client-server
- Code on Demand (Opcional)

Não há uma definição formal para o que é RESTful. O que atualmente é mais aceito como explicação é: RESTful é o termo utilizado para descrever API's HTTP que adotam o padrão REST.

Por que utilizar **RESTful**?

APIs HTTP são os meios padrão de comunicação entre os sistemas. A internet é HTTP. Arquiteturas atuais como Microsserviços utilizam esse padrão em larga escala.

Portanto respeitar o padrão do protocolo, permite que ele cumpra seu objetivo: **Ser simples.**

Já pensou se cada carro de cada montadora tivesse padrões diferente para dirigir? No carro da FIAT o acelerador ficasse no pé esquerdo? E da Ford fosse no meio. O volante do Peugeot fosse invertido, girar o volante pra direita fizesse com que o carro fosse pra a esquerda. A cada troca de carro seria um aprendizado completamente novo. Exigindo extremo esforço para aprender o novo padrão.

Aprender é um processo que gasta muita energia. Há estudos que explicam por que repetição é essencial para a vida humana. A humanidade não sobreviveria se tivesse que reaprender tudo todos os dias. Como escovar os dentes, andar e falar.

Ou seja, **seguir um padrão significa diminuir o gasto de energia no processo de aprendizado. Padronizar significa menos energia.**

Princípios do REST

Client-Server

Separar as responsabilidade do frontend do backend. Esse é um conceito bem comum. Separar o front do back há ganhos significativos em testes, escalabilidade com reflexos até na organização dos times dentro da empresa.

Stateless

O servidor não mantém estado. Cada solicitação do client deve conter informações necessárias para o server entender a solicitação. O estado da sessão é mantido inteiramente no client.

Cacheable

A resposta de uma solicitação deve implicitamente ou explicitamente informar se o dado pode ser mantido em cache ou não. O cache deve ser mantido e gerenciado pelo Client.



Uniform interface

Este princípio é definido por quatro restrições:

- Identificar os recursos (URI)
- Manipular recursos através de representações (Verbos HTTP).
- Mensagens auto-descritivas, cada requisição deve conter informações suficientes para o server processar a informação.
- **HATEOAS:** Hypermedia As The Engine Of Application State.
Esta restrição diz que a response deve conter links de conteúdos relacionados ao resource. Por exemplo:

GET http://api.jpproject.net/users/1

Response:

```
{  
    "userName": "marianne",  
    "email": "marianne@gmail.com",  
    "phoneNumber": "11989500000",  
    "name": "Marianne",  
    "links": [  
        {  
            "href": "10/claims",  
            "rel": "claims",  
            "type": "GET"  
        }  
    ]  
}
```



Neste exemplo repare que o response contém links para o resource claims. Dessa forma o controle de fluxo da aplicação é controlado através do server e não escrito na pedra pelo Frontend.

Veja esse próximo exemplo de HATEOAS, onde é possível saber as opções de Status de um resource.

```
{  
  "userName": "marianne",  
  "status": "Active",  
  ...  
  "links": [  
    {"rel": "block", "href": "/users/10/block"},  
    {"rel": "confirm-email", "href": "/users/10/confirm-email"}  
  ]  
}
```

Layered system

O sistema deve ter uma arquitetura em camadas. A camada acima não pode ver além da camada imediatamente abaixo dela.

Code on demand (opcional)

O REST permite que a funcionalidades do client seja estendida baixando e executando applets ou scripts.

Resources

O REST é Resource Based.

Um Resource é a chave da abstração no REST. Um resource é qualquer coisa importante o suficiente para ser referenciado com um nome. O REST usa um URI (Uniform Resource Identifier) para identificar o resource. Resource é qualquer coisa que possa ter uma URI.

— Conclusões até aqui

Espero que você tenha entendido sobre REST e RESTful. A consideração mais importante até aqui é sobre Resources. Entender Resources é a chave para a criação de API's melhores.

Importante

REST não é CRUD.



Os verbos HTTP parecem estar diretamente ligados com o CRUD:

- **C**reate → POST
- **R**ead → GET
- **U**pdate → PUT / PATCH
- **D**elete → DELETE

Por causa de uma confusão na classificação de Resource, algumas pessoas acabam relacionando REST diretamente a um CRUD.

Resources

Há um senso comum que a classificação de um Resource deve seguir certas regras:

- Coisa ao invés de ação;
- Substantivos ao invés de verbos.

```
POST /users # Cria um usuario  
PUT /users/{id} # Atualiza o usuario  
PATCH /users/{id} # Parcialmente atualiza o usuario  
DELETE /users/{id} # Remove o usuario
```

Acima utiliza os verbos HTTP de maneira correta. **Classificou o recurso como um substantivo.** É possível atualizar, remover, buscar e criar um recurso de usuário.

O problema é que no mundo real, no seu dia a dia, as API's não são tão simples. Elas possuem a complexidade do negócio. Muitas vezes **um recurso é um objeto do sistema.**

Apenas uma URL para atualizar e mudar seu estado se torna insuficiente. E se ele for um objeto do seu sistema, haverá comportamentos.

E é necessário expôr certos comportamentos através da API.

Exemplo:

```
POST /candidatos/{id}/submit  
POST /candidatos/{id}/approve  
POST /candidatos/{id}/decline  
POST /candidatos/{id}/transfer
```

O **resource** candidatos possui comportamentos distintos. O approve pode, por exemplo, disparar um Evento que é capturado por outro sistema. Que por sua vez, vai disparar um e-mail para o RH.

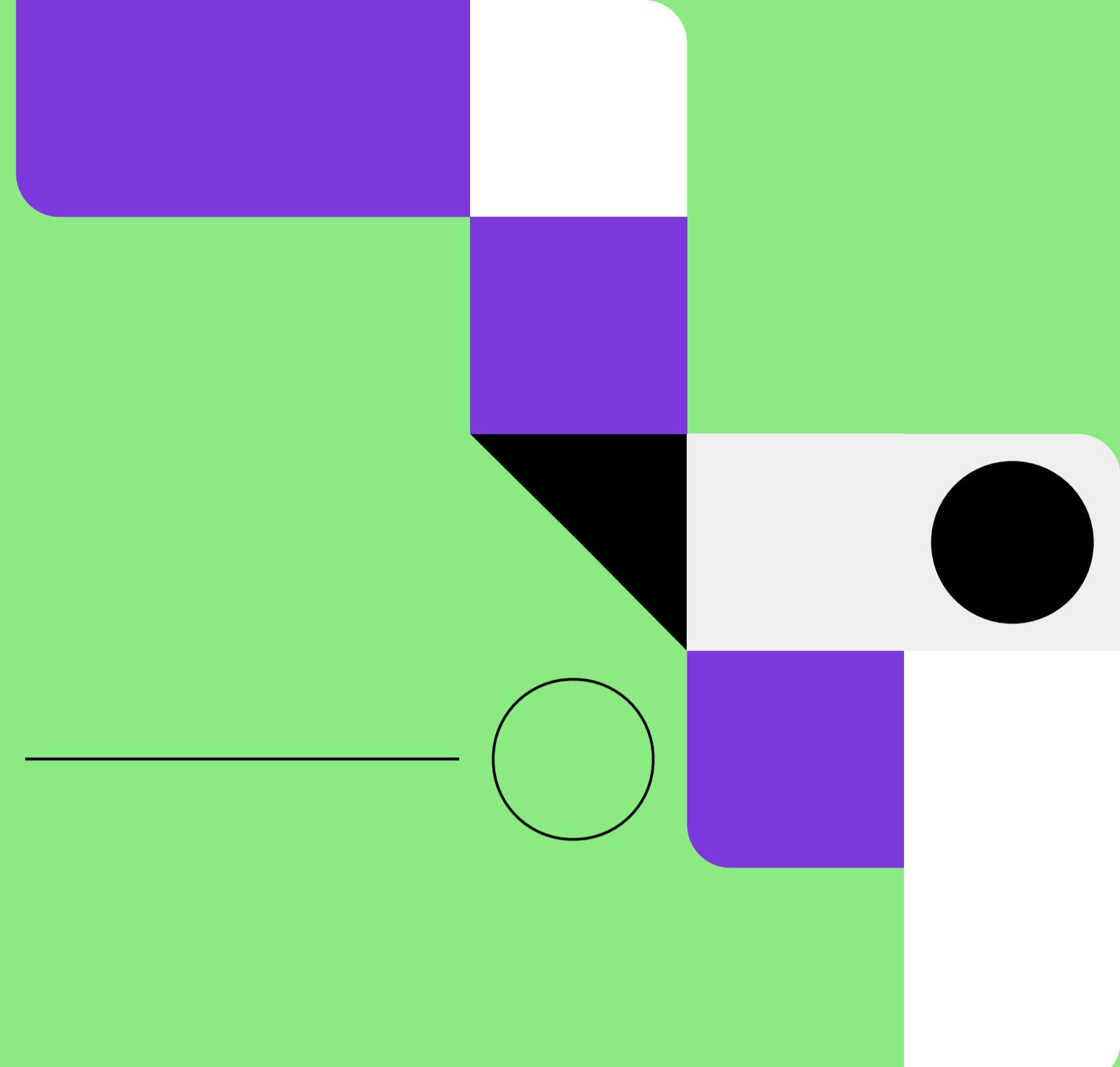
O transfer pode gerar um fluxo de aprovação com outro gestor para aceitar a transferência do candidato.

O REST nunca limitou o Resource a substantivos sem ações.

REST é sobre Resource

Logo uma ação é perfeitamente identificável. candidato/{id}/approve é um resource único, identificável e com uma URI. Além disso ele deixa claro a ação que será feita em conjunto com os verbos HTTP.





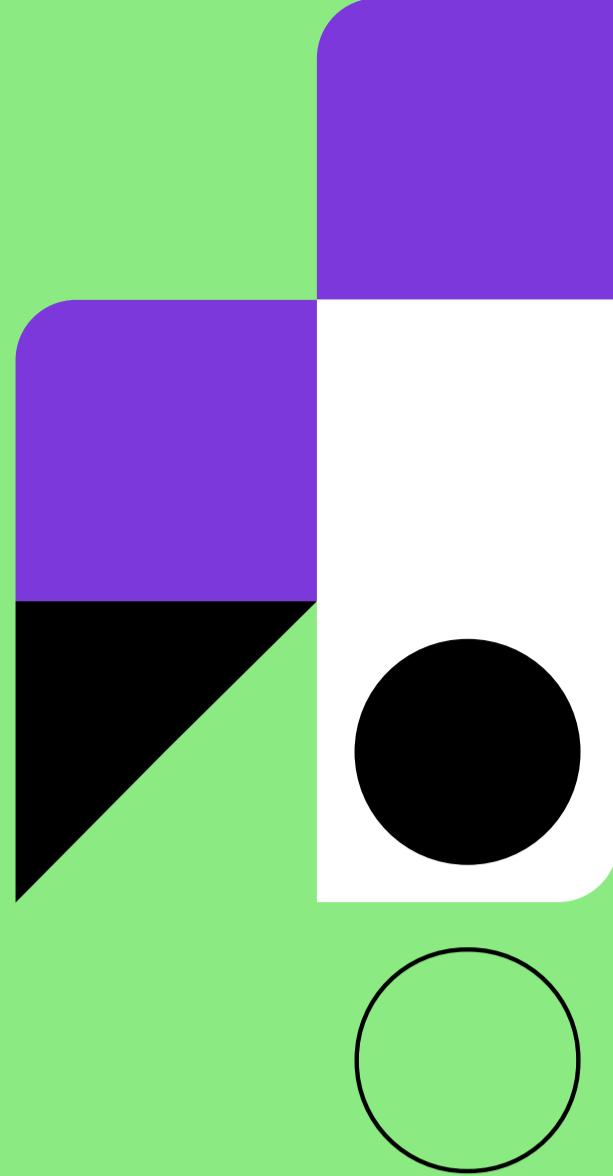
Melhores Práticas REST

HTTP Method	Seguro	Semântica
OPTIONS	Sim	Busca os métodos http válidos e outras opções
GET	Sim	Busca um resource
HEAD	Sim	Busca apenas o header de um resource
PUT	Não	Atualiza um resource
POST	Não	Cria um resource
DELETE	Não	Remove um resource
PATCH	Não	Atualiza parcialmente um resource

Utilizar ações apenas após definir uma URI para um resource, **desde que a ação não sobrescreva o sentido do verbo HTTP**, por exemplo, POST /usuarios/criar . Neste caso basta o verbo HTTP, POST /usuarios .

A ação deve estar ligado a um requisito de negócio. Deve ser clara, POST /usuarios/2/solicitar-ferias . Na Controller o método pode esperar um objeto de tipo **SolicitarFeriasCommand**.

O time ganha liberdade para fazer design melhores, oferecendo recursos que não são baseados em CRUD. E sem ferir o padrão REST.



O que é API?

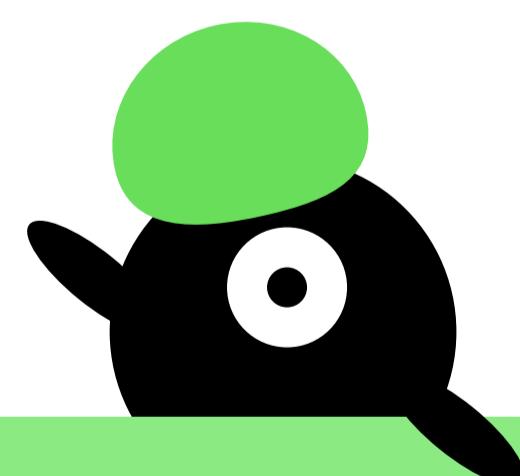
Application Programming
Interface (API)

O conceito de API é bem amplo. API é um **conjunto de padrões que fornecem uma Interface** para que possamos Programar nossas Aplicações. Na prática uma API é um conjunto de código de software que pode ser acessado de diversos locais.

Partindo desta explicação simplista, precisamos entender o conceito de Endpoint, que é literalmente uma ponta de um canal de comunicação. No nosso caso, um Endpoint é uma das pontas da API, que pode ser acessada por uma URL, e que executa um trecho de código quando isso acontece.

Como consumir uma **API**

Para colocar a mão na massa, instale o [POSTMAN](#)



1) Entendendo a API

A primeira informação que precisamos saber é **qual ação/funcionalidade vamos testar, e em qual URL está o endpoint a ser testado.**

Logo após, precisamos saber qual é o método da requisição, que pode ser basicamente:

- **GET:** obter dados;
- **POST:** inserir dados;
- **PUT:** alterar dados;
- **DELETE:** remover dados.

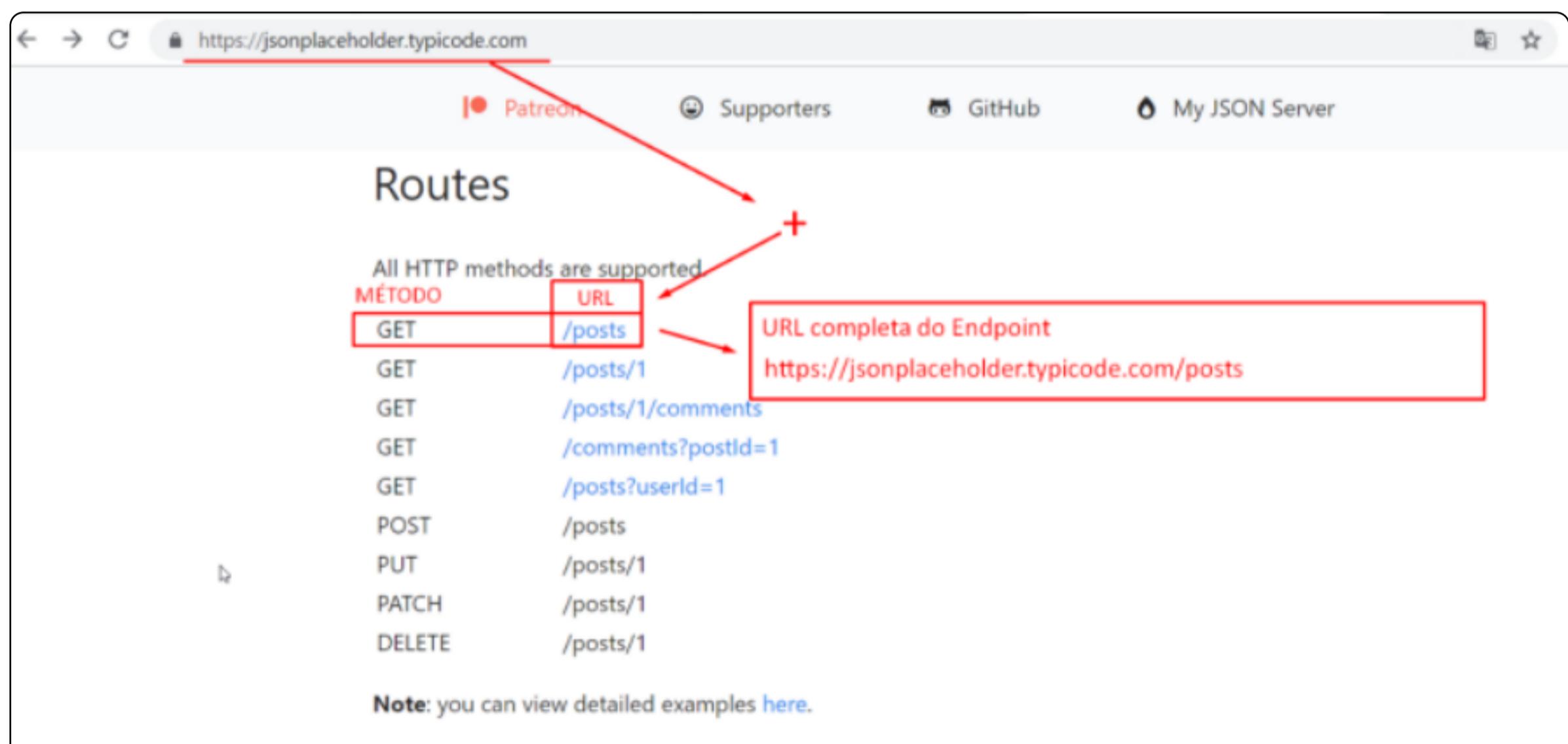
Saber quais são os **headers** da API. Headers são alguns dados-chave que uma API recebe, como por exemplo o tipo de conteúdo do retorno, o método de autenticação, etc.

Esses são os atributos básicos de uma API.



2) Consumindo uma API

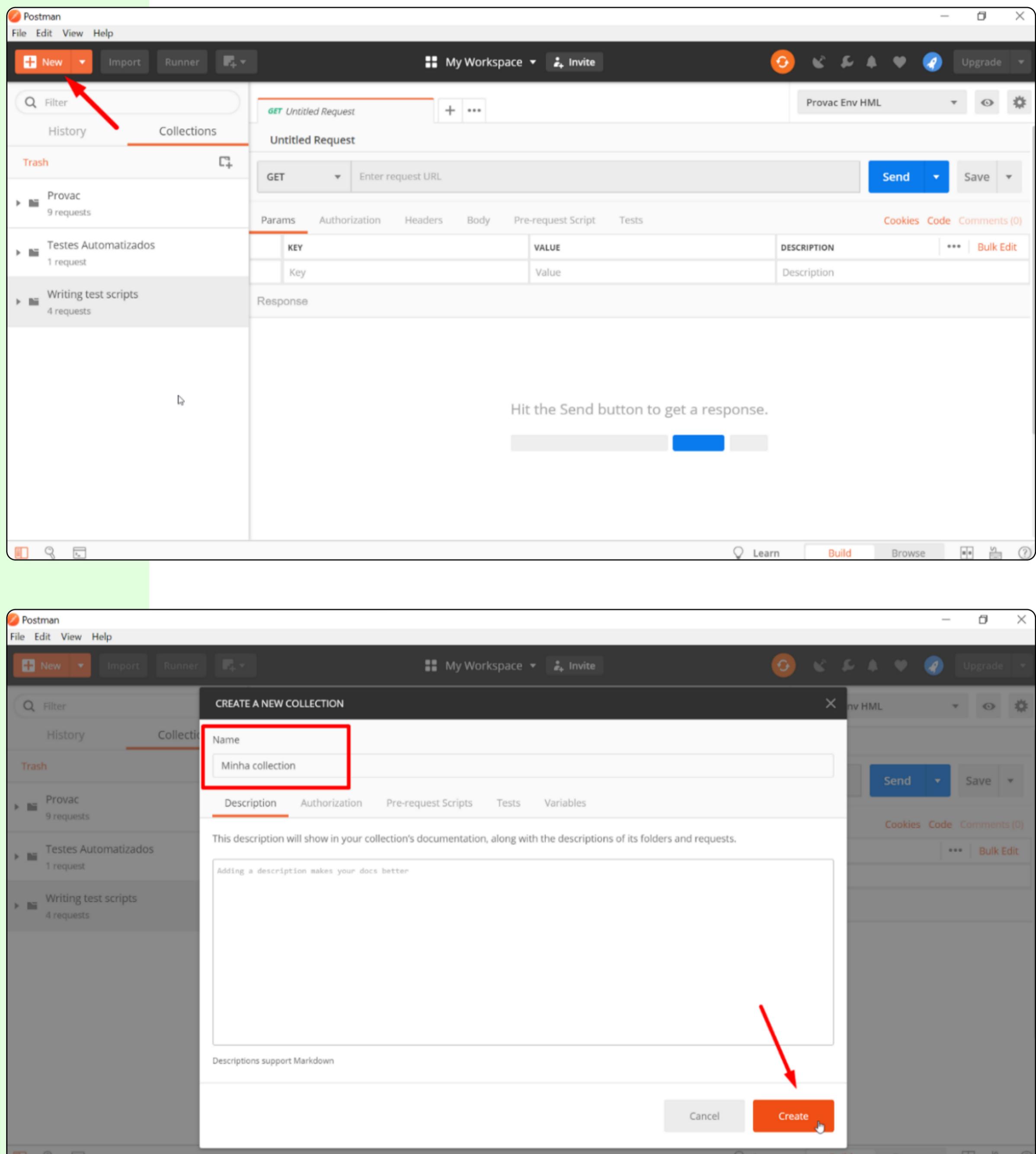
Para este exemplo, vamos usar o Endpoint de ‘posts’ , disponibilizado pelo site [JSONPlaceholder](https://jsonplaceholder.typicode.com), um serviço que disponibiliza APIs fake para testes. Esta API retorna os posts cadastrados no banco de dados.



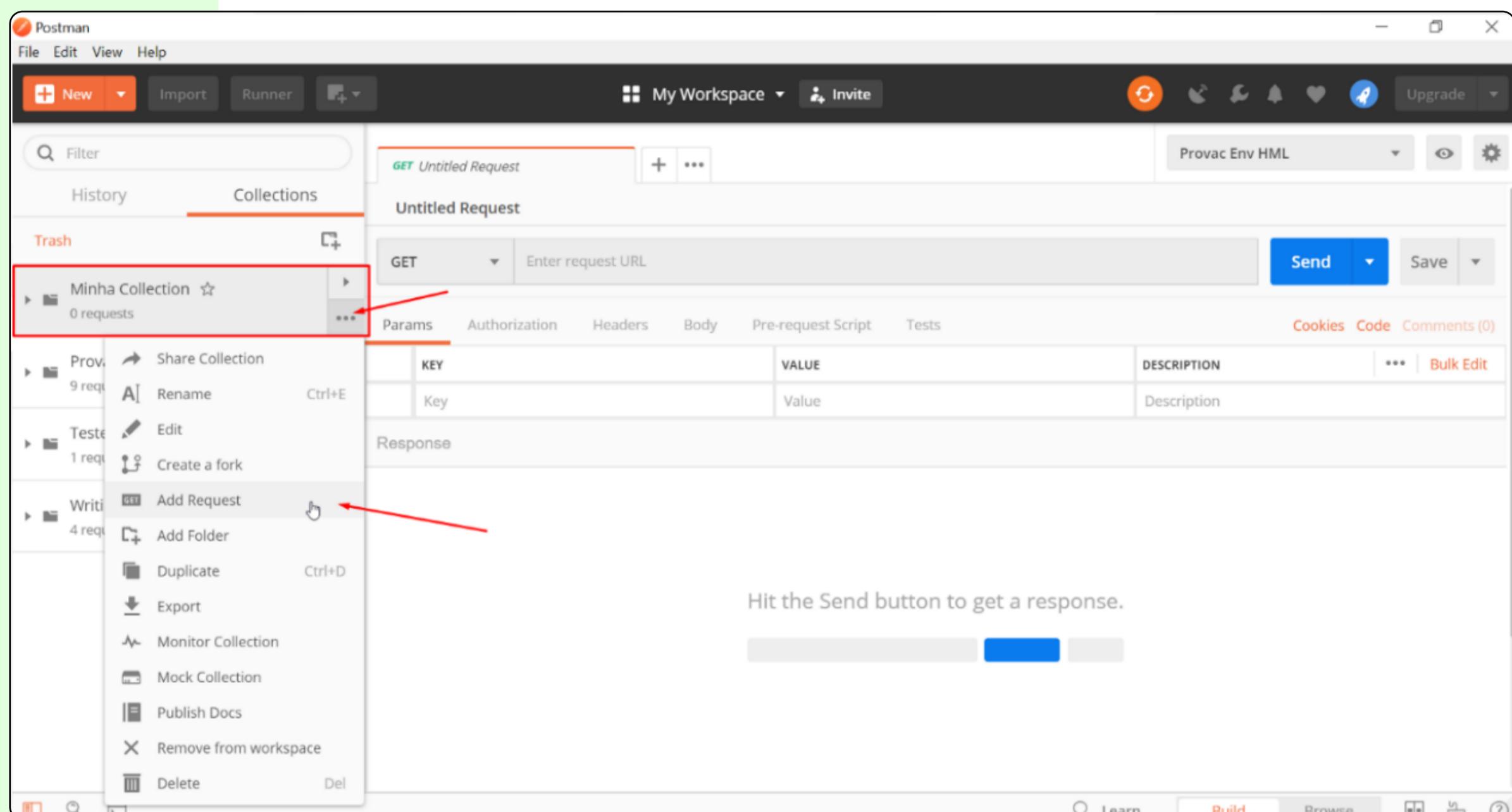
Conforme a imagem mostra, temos os seguintes dados:

- **URL:** <https://jsonplaceholder.typicode.com/posts>
- **Método:** GET
- **Headers:** nenhum obrigatório

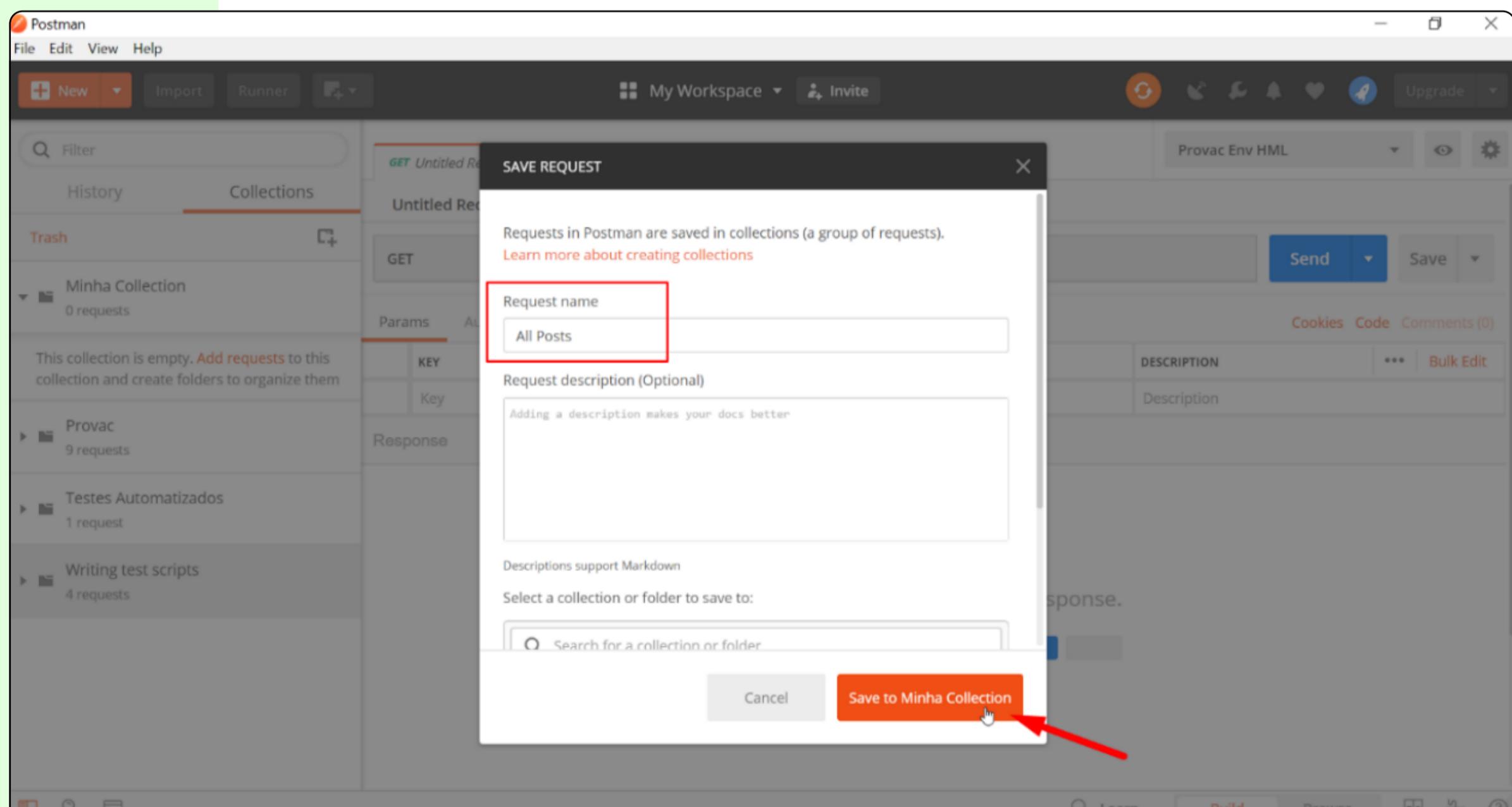
Agora com o Postman aberto, vamos criar uma Collection. Clique no botão “New” e depois em “Collection”. Dê um nome à sua Collection e clique no botão “Create”.



Agora criamos uma Request:



E damos um nome para ela. Geralmente uma request recebe o nome do que ela faz de fato, e neste caso ela retorna todos os posts.



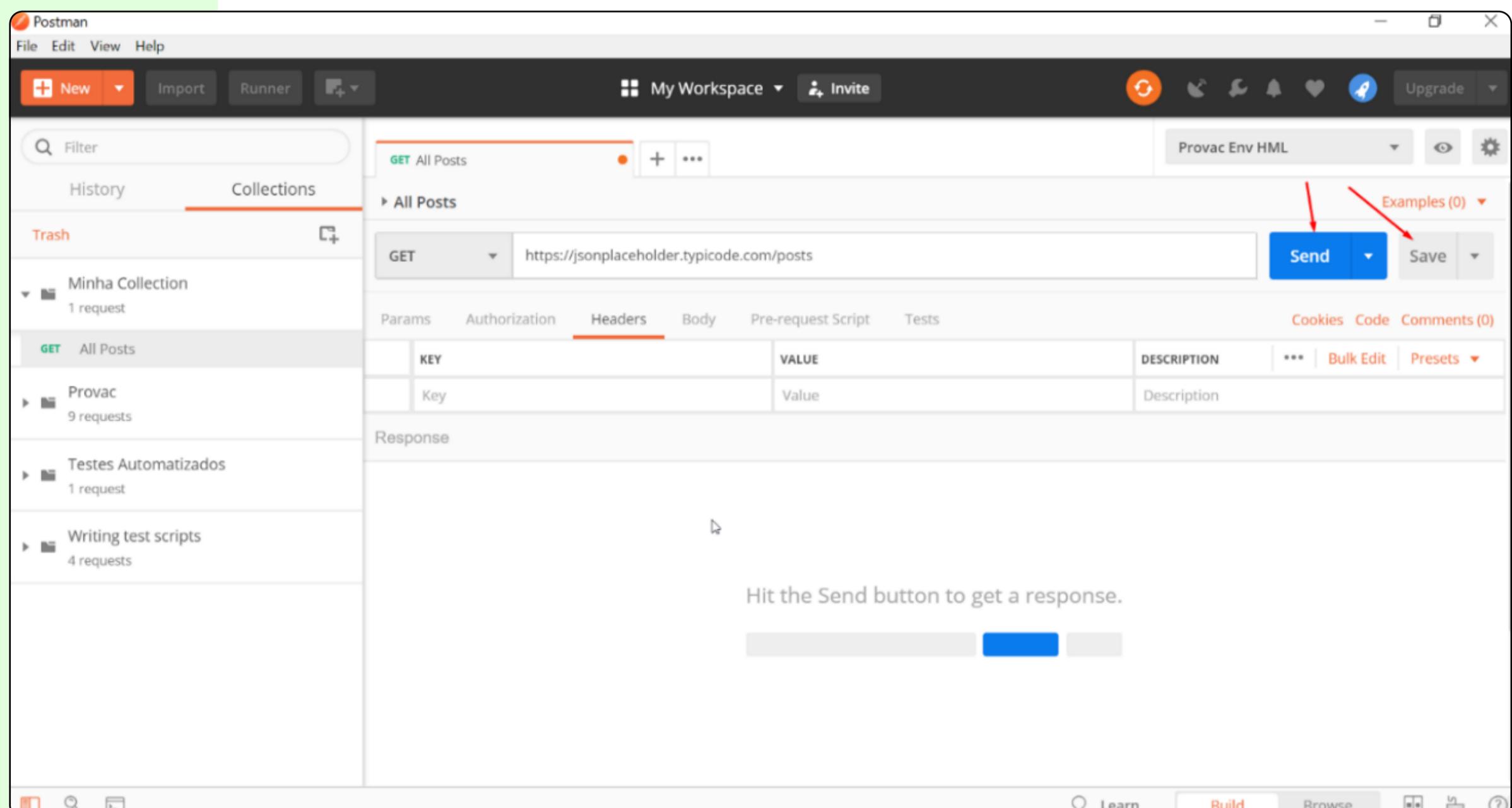
Com a request “All Posts” criada, vamos agora pegar aqueles dados que coletamos na análise e replicar no Postman. O método da request é GET.

The screenshot shows the Postman application window. In the top left, there's a sidebar with 'Minha Collection' containing requests like 'GET All Posts', 'Provac', 'Testes Automatizados', and 'Writing test scripts'. The main area shows a 'GET All Posts' request. The 'Headers' tab is selected, showing a single header entry: 'Key' under 'KEY' and 'Value' under 'VALUE'. Below the table, it says 'Hit the Send button to get a response.' A red arrow points from the text above to the 'Enter request URL' input field, which is currently empty.

A URL é <https://jsonplaceholder.typicode.com/posts>

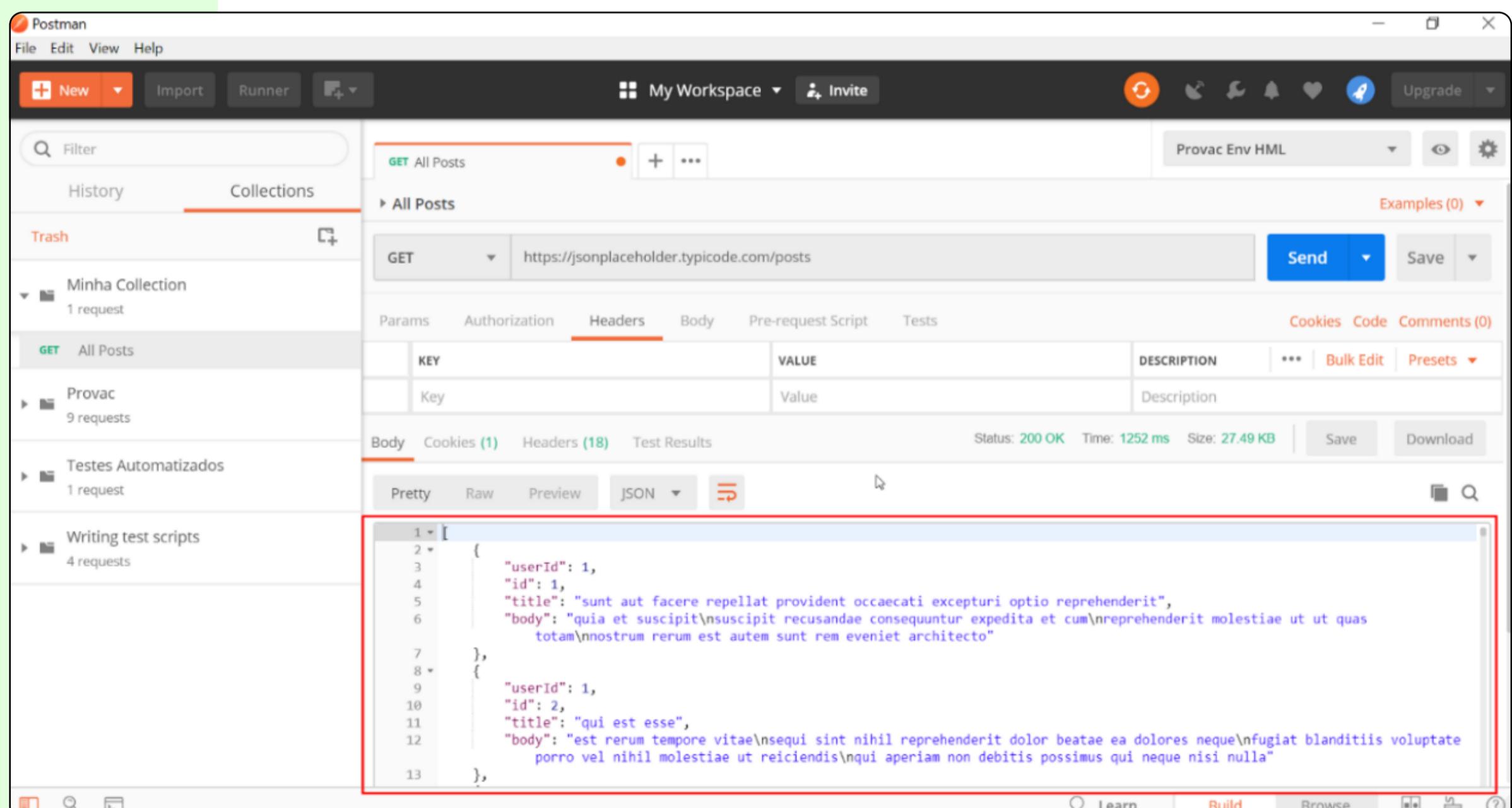
This screenshot shows the same Postman interface as the previous one, but with a red arrow pointing from the text 'A URL é https://jsonplaceholder.typicode.com/posts' to the 'Enter request URL' input field. The URL 'https://jsonplaceholder.typicode.com/posts' is now typed into this field. The rest of the interface remains the same, with the 'Headers' tab selected and a single header entry visible.

Agora é só Salvar essa request, e clicar em **Send**.



The screenshot shows the Postman application interface. On the left, there's a sidebar with 'History' and 'Collections'. Under 'Collections', there's a section for 'Minha Collection' containing 'All Posts', 'Provac', 'Testes Automatizados', and 'Writing test scripts'. In the main area, a request is being configured: method 'GET', URL 'https://jsonplaceholder.typicode.com/posts', and 'Headers' tab selected. The 'Send' button is highlighted with a red arrow. Below the request, it says 'Hit the Send button to get a response.'

E então o webservice retorna todos os posts que estão no banco de dados:



The screenshot shows the Postman interface after the 'Send' button was clicked. The response status is '200 OK'. The 'Body' tab is selected, showing a JSON array of posts. The first two posts are highlighted with a red box. The JSON output is as follows:

```
1 * [
2 *   {
3 *     "userId": 1,
4 *     "id": 1,
5 *     "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
6 *     "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas
7 *     totam\nnostrum rerum est autem sunt rem eveniet architecto"
8 *   },
9 *   {
10 *     "userId": 1,
11 *     "id": 2,
12 *     "title": "qui est esse",
13 *     "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor beatae ea dolores neque\nfugiat blanditiis voluptate
porro vel nihil molestiae ut reiciendis\nqui aperiam non debitis possimus qui neque nisi nulla"
}
]
```

Pronto, esse pequeno tutorial mostra como funciona o consumo de uma API.

Mas qual a diferença entre **endpoint** e **API**?

Um **endpoint** de um web service é a URL onde seu serviço pode ser acessado por uma aplicação cliente.

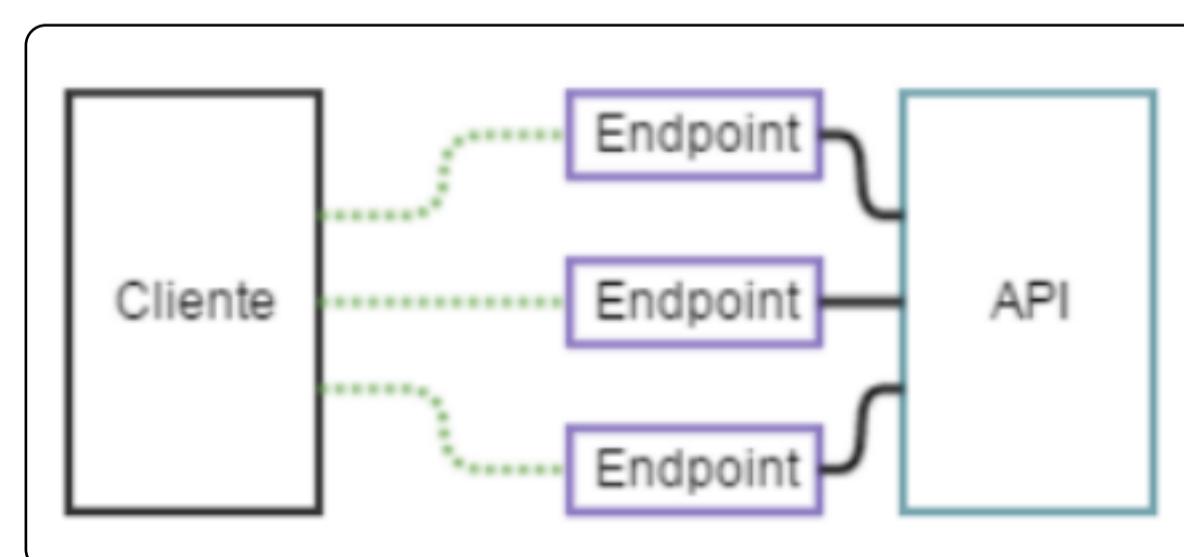
Uma **API** é um conjunto de rotinas, protocolos e ferramentas para construir aplicações.

APIs podem existir sem endpoints. Endpoints também podem existir sem APIs.

Imagine uma implementação simples, que retorna apenas a data e hora do servidor; a simplicidade da operação não exige a implementação de uma API exclusivamente para isso.

Hoje em dia é comum se referir a uma coleção de endpoints pertencentes a um dado serviço como API, por proximidade e acoplamento; em muitos casos o serviço é desenhado e planejado tendo em mente a exposição via endpoints.

Um modelo típico de implementação pode ser interpretado assim:



Onde endpoints são interfaces entre a API e a aplicação consumidora.



SOAP

Simple Object Access Protocol

Definição

SOAP é um protocolo baseado em XML para troca de informações em um ambiente distribuído. É utilizado para troca de mensagens entre aplicativos distribuídos pela rede. Estes aplicativos, ou “Web services”, possuem uma interface de acesso simples e bem definida.

Os Web services são componentes que permitem às aplicações enviar e receber dados em formato XML. Cada aplicação pode ter a sua própria "linguagem", que é traduzida para uma linguagem universal, o formato XML.

Estrutura do protocolo

Envelope

Toda mensagem SOAP deve conter o Envelope. É o elemento raiz do documento XML. O Envelope pode conter declarações de namespaces e também atributos adicionais como o que define o estilo de codificação (encoding style). Um "encoding style" define como os dados são representados no documento XML.

Header

É um cabeçalho opcional. Ele carrega informações adicionais, como por exemplo, se a mensagem deve ser processada por um determinado nó intermediário (É importante lembrar que, ao trafegar pela rede, a mensagem normalmente passa por diversos pontos intermediários, até alcançar o destino final). Quando utilizado, o Header deve ser o primeiro elemento do Envelope.



Body

Este elemento é obrigatório e contém o payload, ou a informação a ser transportada para o seu destino final. O elemento Body pode conter um elemento opcional Fault, usado para carregar mensagens de status e erros retornadas pelos "nós" ao processarem a mensagem.

A estrutura da mensagem SOAP é definida em um documento XML que contém os seguintes elementos:

```
<SOAP-ENV:envelope>
<!– Elemento raiz do SOAP e define que essa é uma mensagem SOAP—>
<SOAP-ENV:header>
<!–Especifica informações específicas como autenticação (opcional)—>
</SOAP-ENV:header>
<SOAP-ENV:body>
<!–O elemento BODY contém o corpo da mensagem—>
<SOAP-ENV:fault>
<!–O elemento FAULT contém os erros que podem ocorrer—>
</SOAP-ENV:fault>
</SOAP-ENV:body>
</SOAP-ENV:envelope>
```

Envelope (obrigatório): é responsável por definir o conteúdo da mensagem.



encodingStyle: atributo que tem como objetivo especificar como as informações devem ser codificadas.

```
<SOAP-ENV:Envelope xmlns:SOAP ENV="http://schemas.xmlsoap.org/soap"  
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">  
<SOAP-ENV:Header>  
...  
</SOAP-ENV:Header>  
<SOAP-ENV:Body>  
...  
</SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

Header (opcional): contém os dados do cabeçalho.

```
<SOAP-ENV:Envelope xmlns:SOAP ENV="http://schemas.xmlsoap.org/soap"  
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">  
<SOAP-ENV:Header>  
<a:authentication xmlns:a="http://www.soap.com/soap/authentication">  
<a:username>Marianne</a:username>  
<a:password>12345</a:password>  
</a:authentication>  
</SOAP-ENV:Header>  
<SOAP-ENV:Body>  
...  
</SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```



actor: especifica o receptor que deve processar o elemento do cabeçalho.

```
<SOAP-ENV:Envelope xmlns:SOAP ENV="http://schemas.xmlsoap.org/soap"  
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">  
<SOAP-ENV:Header>  
<a:authentication xmlns:a="http://www.soap.com/soap/authentication"  
SOAP-ENV:actor="http://www.soap.com/soap/authenticator">  
<a:username>Marianne</a:username>  
<a:password>12345</a:password>  
</a:authentication>
```

mustUnderstand: especifica se uma entrada de cabeçalho é obrigatória ou opcional (booleano)

```
<SOAP-ENV:Envelope xmlns:SOAP ENV="http://schemas.xmlsoap.org/soap/"  
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">  
<SOAP-ENV:Header>  
<a:authentication xmlns:a="http://www.soap.com/soap/authentication"  
SOAP-ENV:mustUnderstand="1">  
<a:username>Marianne</a:username>  
<a:password>12345</a:password>  
</a:authentication>
```



Body (obrigatório)

Contém a codificação atual de uma chamada a um método e todos os argumentos de entrada ou uma resposta codificada que contém o resultado de uma chamada de um método.

Fault

Contém as informações dos erros ocorridos no envio da mensagem. Apenas nas mensagens de resposta do servidor.

Envelope SOAP

É a parte obrigatória de uma mensagem SOAP. Ele funciona como um recipiente de todos os outros elementos da mensagem, possivelmente o cabeçalho e o corpo, assim como os namespaces de cada um. Assim como o nome e o endereço de uma carta entregue pelo correio, o envelope SOAP precisa das informações específicas do protocolo de transporte que está ligado a ele, com o intuito de garantir a chegada ao local certo.

Especificamente no HTTP, temos um cabeçalho que se chama SOAPAction, indicador do endereço de entrega da mensagem. Um dos principais motivos de implementarmos o cabeçalho desta maneira é porque administradores de sistemas podem configurar seus firewalls para filtrar as mensagens baseadas nas informações dos cabeçalhos, sem consultar o XML.

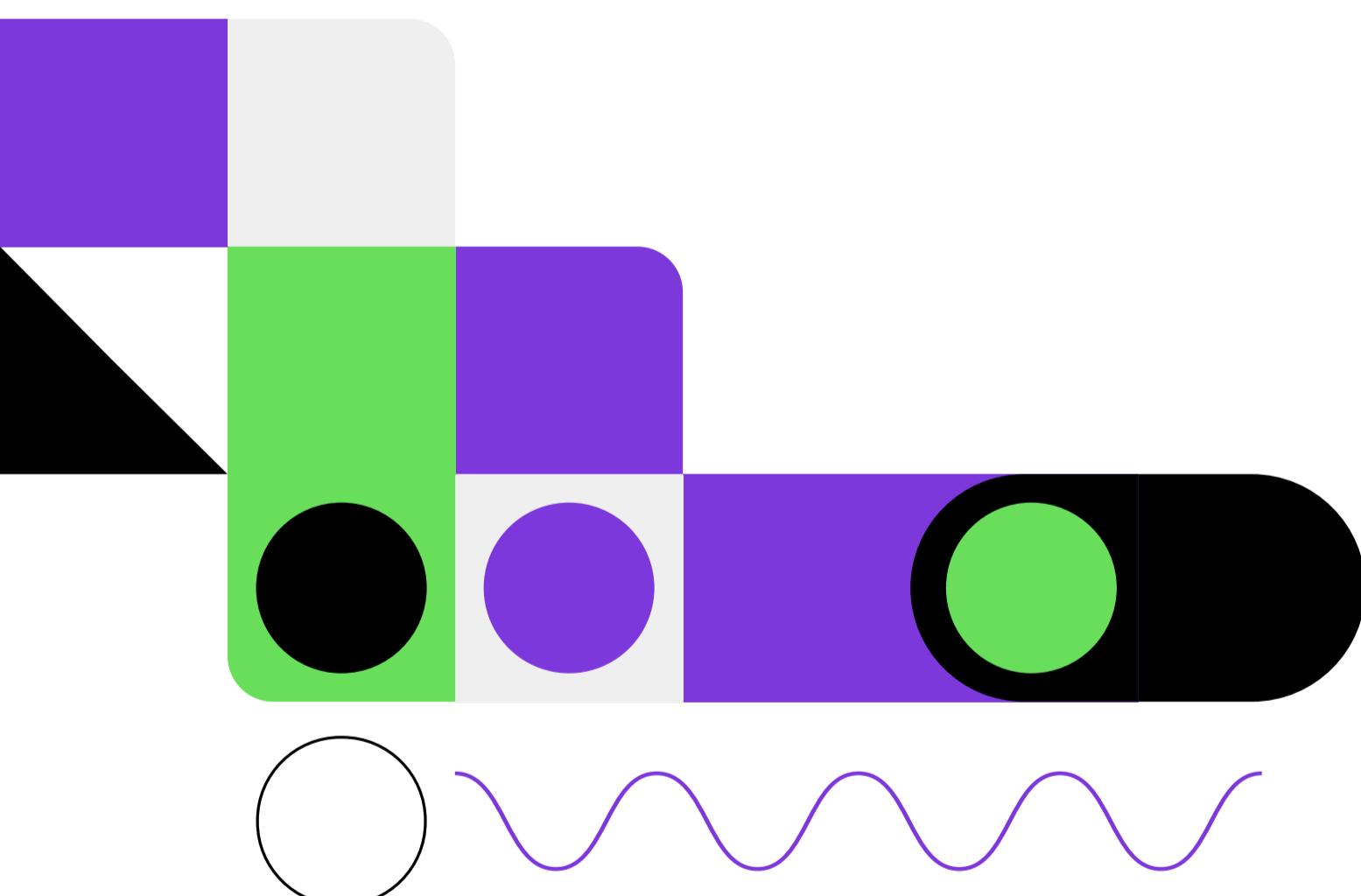
Elemento	Namespace/URL
Envelope	http://schemas.xmlsoap.org/soap/envelope
Serializador	http://schemas.xmlsoap.org/soap/encoding
SOAP-ENV	http://schemas.xmlsoap.org/soap/envelope
SOAP-ENC	http://schemas.xmlsoap.org/soap/envelope
Xsi	http://www.w3.org/1999/XMLSchema-instance
Xsd	http://www.w3.org/1999/XMLSchema

REST x SOAP

O REST opera por meio de uma interface consistente para acessar os recursos nomeados. É mais usado quando se publica uma API pública pela Internet.

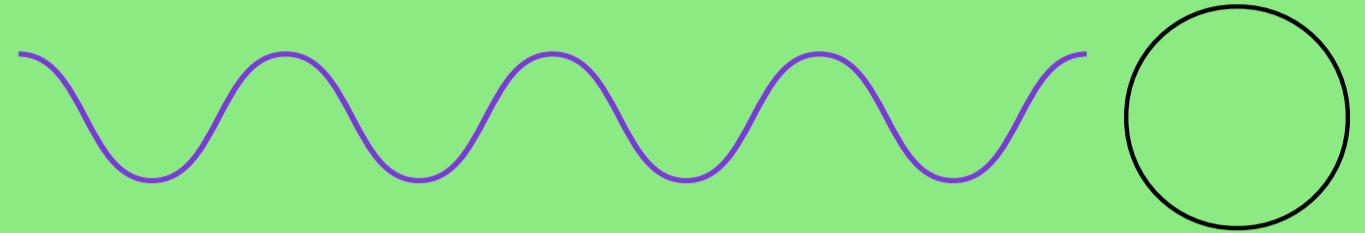
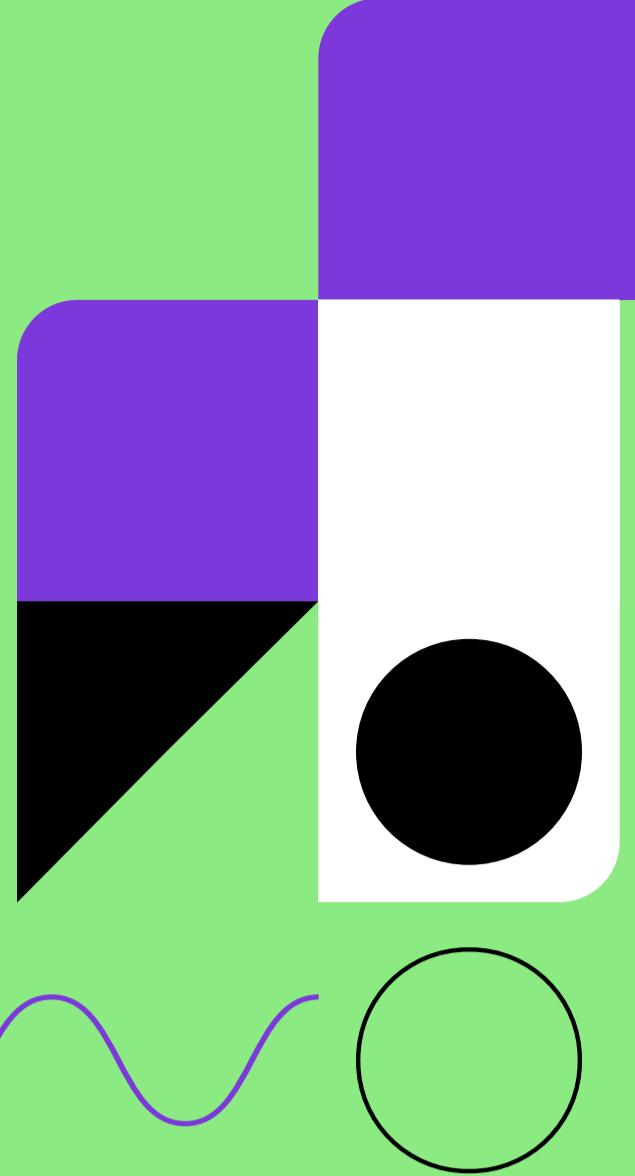
Já o SOAP, por outro lado, expõe componentes da lógica do aplicativo como serviços, e não como dados. Além disso, opera por meio de diferentes interfaces.

Simplificando, o REST acessa os dados enquanto o SOAP executa operações por meio de um conjunto mais padronizado de mensagens. Ainda assim, na maioria dos casos, tanto REST como o SOAP podem ser usados para obter o mesmo resultado (e ambos são infinitamente escaláveis), com algumas diferenças na forma como são configurados.



Os tópicos abaixo vão ajudar a entender o embate **REST x SOAP**

- A API REST não possui um padrão oficial, pois é um **estilo de arquitetura**. A API SOAP, por outro lado, possui um padrão oficial porque é um **protocolo**;
- REST usa vários padrões como HTTP, JSON, URL e XML, enquanto SOAP é baseada em HTTP e XML;
- Como REST implementa vários padrões, são necessários menos recursos e largura de banda em comparação com o SOAP que usa XML para a sua criação, resultando em um arquivo de tamanho grande;
- As maneiras pelas quais as duas APIs expõem as lógicas de negócios também são diferentes. REST aproveita a exposição da URL como @path ("/ WeatherService"), enquanto o uso da API SOAP de interfaces de serviços como @WebService;
- REST usa a linguagem de descrição de aplicações da Web e SOAP usa a linguagem de descrição de serviços da Web para descrever as funcionalidades oferecidas pelos serviços da web;
- REST é compatível com JavaScript e também pode ser implementado facilmente. Já SOAP também é conveniente com JavaScript, mas não suporta uma implementação maior.



NODEMON

Configurando Nodemon

O nodemon é uma biblioteca que ajuda no desenvolvimento de sistemas com o Node.js reiniciando automaticamente o servidor.

Imagine a seguinte situação, você está desenvolvendo uma aplicação com o Node, e criou uma rota, para acessá-la, é preciso reiniciar o servidor. Caso crie um controller, ou algum outro método para se conectar com essa rota, será necessário reiniciar novamente para que seja aplicado as alterações.

Veja que a cada alteração, é necessário reiniciar o servidor, para que a mudança seja realmente aplicada, e fazer isso durante um dia inteiro de desenvolvimento é bem desgastante. É exatamente esse problema que o Nodemon resolve, ele fica monitorando a aplicação em Node, e assim que houver qualquer mudança no código, o servidor é reiniciado automaticamente.

Com isso é possível ganhar muito mais produtividade durante o desenvolvimento.



Instalação

Há duas formas de instalação. A primeira forma, é utilizada para caso você queira acessar o nodemon em qualquer diretório do seu sistema operacional, com isso, qualquer diretório que tenha uma aplicação Node, poderá ser iniciada com ele.

```
npm install -g nodemon
```

ou

```
yarn global add nodemon
```

Com essa instalação, é possível colocar no console nodemon index.js e ele iniciará o servidor.



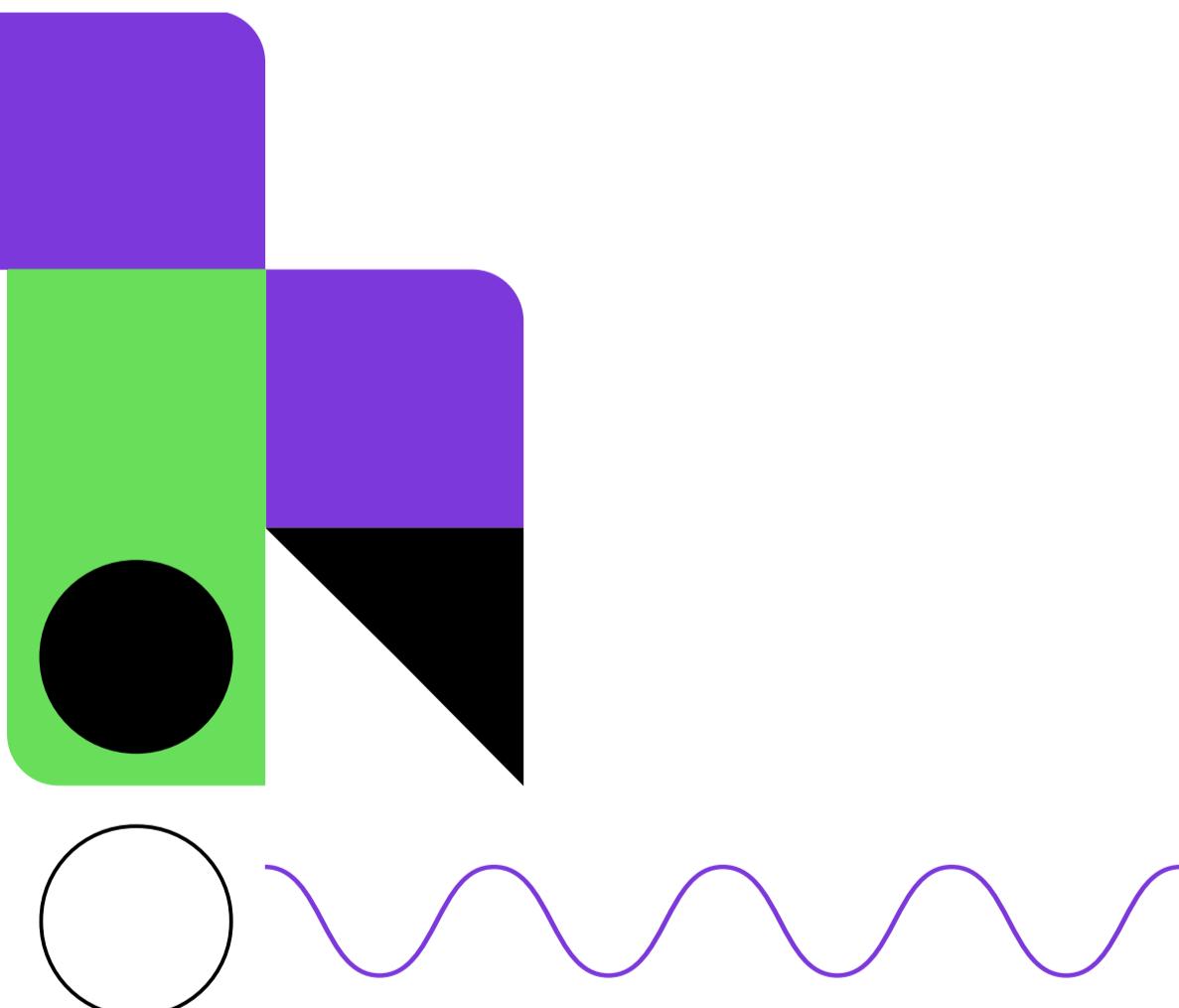
A segunda forma, é instalar como dependência de desenvolvimento no projeto, dessa forma não é possível iniciar em qualquer diretório, e será preciso criar um script no package.json para iniciar a aplicação.

```
npm install -save-dev nodemon
```

ou

```
yarn add nodemon -dev
```

Com essa instalação, ele ficará no package.json na parte de devDependencies.



Como utilizar

Para o exemplo, vou usar uma aplicação Node com o Express, e que tenha uma rota.

```
/server.js
const express = require("express")
const app = express()
const PORT = 3000

app.get("/", (req, res) => {
    res.json({message: 'ok'})
})

app.listen(PORT, () => {
    console.log(`Running in http://localhost:${PORT}`)
})
```

Navegue até o diretório da aplicação pelo terminal e digitar o seguinte comando:

```
nodemon server.js
```

A aplicação será iniciada e começará a ser monitorada, e a cada alteração, será reiniciada automaticamente, a saída no terminal será essa:

```
vitorceron@vitorceron-Aspire-E5-573G ~/Documentos/Projetos/nodejs-nodemon $ nodemon server.js
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Running in http://localhost:3000
```

A cada alteração, ele irá reiniciar e mostrar essa mesma saída.

Iniciar por script

No package.json, é possível criar um script para iniciar a aplicação, esse item é muito utilizado, pois caso a aplicação tenha que subir muitos serviços, ou um comando muito extenso, compensa criar um script para isso.

No arquivo package.json, há uma chave scripts, que normalmente vem assim por padrão:

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

Para criar um novo script de inicio da aplicação, é possível criar uma chave start, abaixo da chave test, e passar para essa chave o comando de inicialização, que nesse caso será com o nodemon.

Atualize o package.json para deixar dessa forma a chave de scripts:

```
"scripts": {  
  "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  "start": "nodemon server.js"  
},
```

Dessa forma o script já está criado e estará funcionando, para testar, navegue até o diretório do projeto pelo terminal, e execute o seguinte comando:

```
npm start  
//ou  
yarn start //se estiver utilizando o yarn
```

A saída do terminal, será similar a essa:

```
vitorceron@vitorceron-Aspire-E5-573G ~/Documentos/Projetos/nodejs-nodemon $ npm start  
> nodejs-nodemon@1.0.0 start  
> nodemon server.js  
  
[nodemon] 2.0.7  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node server.js`  
Running in http://localhost:3000  
]
```

Arquivo de configuração

É possível passar algumas configurações a mais para o nodemon realizar algumas tarefas de forma diferente. Como por exemplo, é possível adicionar um delay, na reinicialização do servidor, monitorar alguns arquivos específicos, ignorar o monitoramento de arquivos e entre outras configurações.

É possível criar um arquivo chamado nodemon.json no projeto e colocar todas as configurações dentro dele.

Veja o exemplo:

```
//nodemon.json
{
  "verbose": true,
  "ignore": [
    ".git",
    "node_modules/**/node_modules"
  ],
  "delay": 10000
}
```

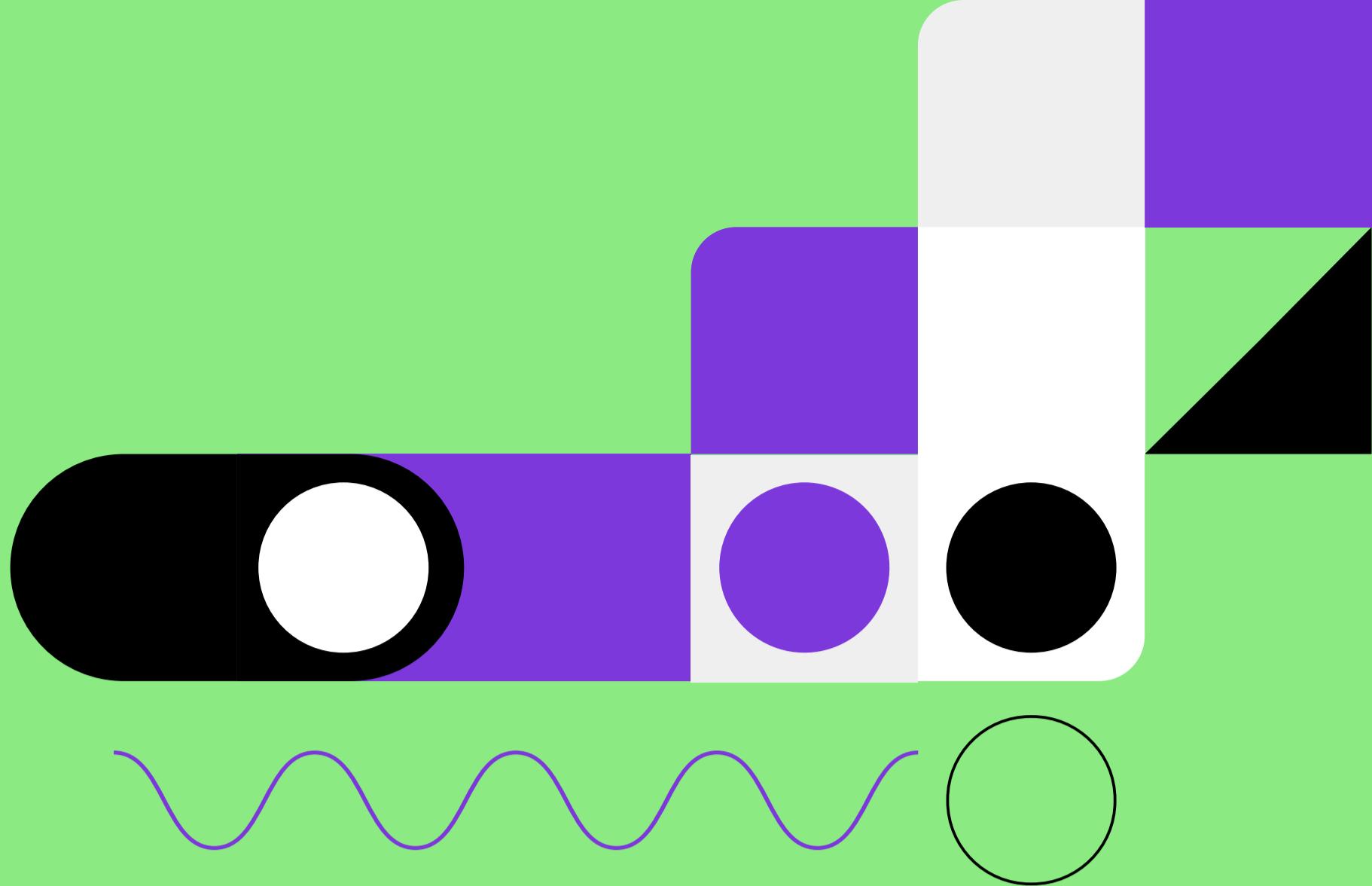
Neste arquivo, basicamente está dizendo que é para ignorar qualquer alteração relacionada ao git, e também no diretório node_modules, que é o diretório que fica as bibliotecas.

Também tem uma configuração de delay, que irá demorar 10 segundos para reiniciar após qualquer mudança.

Com esse arquivo de configuração, quando rodar a aplicação, seja por qualquer uma das duas formas, terá uma saída como essa no terminal:

```
vitorceron@vitorceron-Aspire-E5-573G ~/Documentos/Projetos/nodejs-nodemon $ npm start
> nodejs-nodemon@1.0.0 start
> nodemon server.js

[nodemon] 2.0.7
[nodemon] reading config ./nodemon.json
[nodemon] to restart at any time, enter `rs`
[nodemon] or send SIGHUP to 13222 to restart
[nodemon] ignoring: .git node_modules/**/node_modules
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
[nodemon] forking
[nodemon] child pid: 13235
[nodemon] watching 4 files
Running in http://localhost:3000
```



DOCKER

Configurando Docker

O Docker é uma forma de abstração da infraestrutura de projetos que nos beneficia tanto em tempo de desenvolvimento quanto em produção.

O que é o Docker?

Com certeza você já ouviu uma famosa frase 'Na minha máquina funciona'... então se na sua máquina funciona, por que não enviar ela para produção?

Pois bem, o que o Docker faz é parecido com isto. Em termos gerais ele **abstrai a infraestrutura das aplicações através de imagens prontas**.

O que temos são basicamente '**máquinas virtuais**' de baixo custo rodando somente com a infraestrutura das nossas aplicações.

Por exemplo, uma das imagens mais famosas que temos no Docker é a **Alpine**, que contém apenas 5MB. Esta imagem contém apenas o Core do sistema operacional, no caso Linux, que a maioria das aplicações precisam para serem executadas.



A ideia é que a partir dela, você possa adicionar o que mais precisar, como a SDK do .NET, Node, PHP e depois fazer um pacote disto.

Com este pacote montado, você envia ele completo, do jeito que está, para produção, assim não corre o risco de dar algo errado.

Embora o Docker seja voltado para infraestrutura, um dos seus grandes usos também são **ambientes de desenvolvimento**.

Hoje trabalhamos com diversos bancos de dados, diversas tecnologias e serviços, isto tudo gera um acúmulo de instalações muito grandes.

Imagina instalar o SQL Server, MySQL e MongoDb, todos na sua máquina, sendo executados como serviços do Windows. É complicado.

Neste caso, o que podemos fazer é utilizar imagens prontas de máquinas com SQL Server, MySQL e Mongo e simplesmente executá-las quando quisermos.

Quer criar um projeto com SQL Server? Só fazer o download da imagem oficial dele para Docker, executar e pronto, temos o serviço rodando.

Criando a Conta

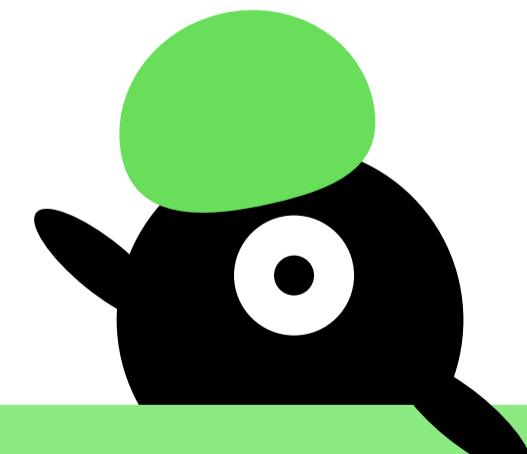
A primeira coisa que precisamos fazer é criar uma conta no **Docker Hub**, ela vai ser utilizada tanto para ativação do Docker em nossa máquina quanto para gerenciar nossas imagens.

Caso já tenha uma conta criada, pode prosseguir para o passo seguinte, do caso contrário, navegue até o site oficial do Docker, clique em **Sign In** e em seguida em **Sign Up**.

Preencha todas as informações necessárias e pronto. É importante lembrar que eles verificarão sua conta via E-mail, então não esqueça este passo também.

Instalando o Docker

O Docker possui uma versão para Desktop que irá facilitar nossa vida. Se você gosta de linha de comando, não se preocupe pois ele também fará a instalação do CLI.



Testando a Instalação

Ao instalar o Docker automaticamente fazemos a instalação do seu CLI, que disponibiliza o comando docker em nosso terminal.

Desta forma, abra um terminal e digite o seguinte comando para ver a versão do Docker instalada:

```
docker -version
```

Vamos então executar nosso famoso **Hello World**, que vai baixar e executar um contêiner para testarmos se tudo está funcionando corretamente.

```
docker run hello-world
```

Ao executar o comando, veremos a mensagem 'Unable to find image hello-world:latest locally', o que significa que ele não conseguiu encontrar nenhuma imagem chamada hello-world localmente.

Faz todo sentido, afinal não fizemos download de nada ainda, mas mesmo assim, ele fará uma busca online e encontrará esta imagem.

Seguindo adiante ele fará o download e execução da imagem, exibindo a mensagem '**This message shows that your installation appears to be working correctly.**' caso tudo tenha dado certo.

Listando as Imagens

Para listar as imagens que temos localmente é só executar o comando abaixo.

```
docker image ls
```

Como você deve imaginar, estas imagens ocupam espaço em disco, e podemos removê-las e baixá-las novamente a qualquer momento.

No caso, para remover uma imagem baixada, podemos utilizar o comando abaixo.

```
docker image rmi ID_DA_IMAGEM
```

É importante frisar que uma imagem não poderá ser apagada caso esteja em uso.

Listando os Contêineres

Para listar os contêineres podemos utilizar o comando abaixo, porém neste caso o hello-world não aparecerá.

```
docker container ls
```

Isto ocorre pelo hello-world ser um contêiner de testes apenas, então para vê-lo listado, precisamos executar o comando com --all no final.

```
docker container ls -all
```

Desta forma você verá todos os contêineres que estão na máquina.

Iniciando e parando um **contêiner**

Para executar ou para um contêiner, podemos usar o comando start/stop.

```
docker container start ID_DO_CONTAINER  
docker container stop ID_DO_CONTAINER
```

Para ver a lista completa do que podemos fazer com um contêiner basta executar o comando abaixo.

```
docker container -help
```

Removendo um **contêiner**

Com o contêiner parado, podemos removê-lo utilizando o comando abaixo. Em seguida podemos também executar o comando para remover a imagem baixada.

```
docker container rm ID_DO_CONTAINER
```



Docker File

O Dockerfile é o arquivo de instruções que o Docker utilizará para saber como publicar nossa aplicação em um contêiner, bem como qual imagem ele deve utilizar. O primeiro passo que precisamos fazer neste arquivo é definir qual imagem vamos utilizar em nossa aplicação.

Lembre-se que a imagem é a **base para construção**, contendo apenas o necessário do SO para execução da nossa aplicação. Neste caso, vamos utilizar a imagem oficial do Node para Docker.

Para definir que estamos utilizando uma imagem como base, utilizamos a palavra `FROM`, seguida da imagem que queremos utilizar.

```
FROM node:current-slim
```

Note que após o nome da imagem temos dois pontos e `current-slim`. Na verdade o que vem depois dos dois pontos é a TAG da imagem.

As tags são utilizadas para definir versões destas imagens e no caso, `current` diz que estamos pegando a versão mais atual do Node (LTS - Long Term Support) e `slim` diz que é a versão enxuta.

Cada imagem tem suas tags, então você deve sempre olhar no Docker Hub pela versão que precisa utilizar.

A segunda definição que iremos fazer em nosso arquivo será o caminho, dentro da imagem, onde ficará o código fonte da nossa aplicação. Isto é feito utilizando a palavra **WORKDIR**, como mostrado abaixo.

```
WORKDIR /usr/src/app
```

Agora precisamos copiar o arquivo **package.json**, que contém os pacotes necessários para execução da nossa API.

Lembre-se que no caso do Node, não precisamos instalar uma SDK ou algo do tipo, porém toda aplicação tem a pasta `node_modules` com as bibliotecas que utilizamos durante a construção do App. Para executar esta cópia vamos utilizar a palavra **COPY**, como mostrado abaixo.

```
COPY package.json .
```

Note que no fim do comando há um `'.'`, definindo que o arquivo será copiado para raiz da nossa aplicação definido no **WORKDIR**.

Com tudo pronto, vamos executar o comando **NPM INSTALL** para instalar todos os pacotes que precisamos, e isto é feito utilizando a palavra **RUN**.

```
RUN npm install
```

Até o momento temos o código da nossa aplicação sendo copiado para a imagem e em seguida sendo instalados os pacotes que a API precisa para rodar.

Porém, precisamos definir em qual porta esta API irá rodar, e isto é feito pela palavra **EXPOSE**, como mostrado abaixo.

```
EXPOSE 8080
```

Embora tenhamos copiado os arquivos e instalado os pacotes, nossa API ainda não está executando, e para isto precisamos de fato executar o comando **NPM START**.

Desta forma, vamos utilizar a palavra **CMD** que nos permite passar uma lista de comandos a serem executados.

```
CMD [ 'npm' , 'start' ]
```

Por fim, vamos executar mais um COPY para copiar o resto da nossa aplicação para a imagem.

COPY . .

Note que utilizamos o mesmo esquema anterior, copiando os arquivos da raiz (!) da nossa aplicação para a raiz (.) da imagem. O arquivo final Dockerfile fica com o seguinte conteúdo.

```
# Copie o arquivo do seu host para sua localização atual
COPY package.json .

# Execute o comando dentro do seu sistema de arquivos de imagem
RUN npm install

# Informe ao Docker que o contêiner está escutando
# na porta especificada em tempo de execução.
EXPOSE 8080

# Execute o comando especificado no contêiner.
CMD [ "npm", "start" ]

# Copie o restante do código-fonte do seu aplicativo do host
# para o sistema de arquivos de imagem.
COPY . .
```

Criando uma imagem

Nosso desafio agora é criar uma imagem customizada a partir desta API/Dockerfile que criamos. Vamos juntar a imagem padrão definida no Dockerfile com o restante da execução do script.

Dentre os comandos que podemos executar, temos o **docker image build**, para criar uma imagem, e vamos utilizá-lo para gerar o que precisamos aqui.

Execute o seguinte comando:

```
docker image build -t dockerapi:1.0 .
```

Para facilitar futuras execuções, vamos dar um nome para esta imagem, adicionando o parâmetro **-t**, seguido pelo **nome:versão da imagem**.

Por fim, devemos especificar o diretório onde está nosso Dockerfile, que se for na raiz da API/Projeto, é definido no comando pelo **"."**.

Para verificar se tudo deu certo, basta executar o comando **docker image ls** para listar as imagens que temos na máquina.

Rodando o **contêiner**

A última coisa que precisamos fazer é pegar esta imagem gerada e executar ela como um contêiner, e isto é feito pelo comando docker container run.

Novamente, certifique-se que você está na raiz da API, e execute o seguinte comando.

```
docker container run -publish 8080:3000 -detach -name api dockerapi
```

O primeiro parâmetro que temos no comando é o -- publish, que vai fazer o redirecionamento do tráfego para a porta onde nossa API está rodando.

Neste caso, vamos rodar nosso contêiner na porta 8080, ou seja, acessaremos nossa API no Browser pela URL <http://localhost:8080>, porém, internamente nossa API está rodando na porta 3000.

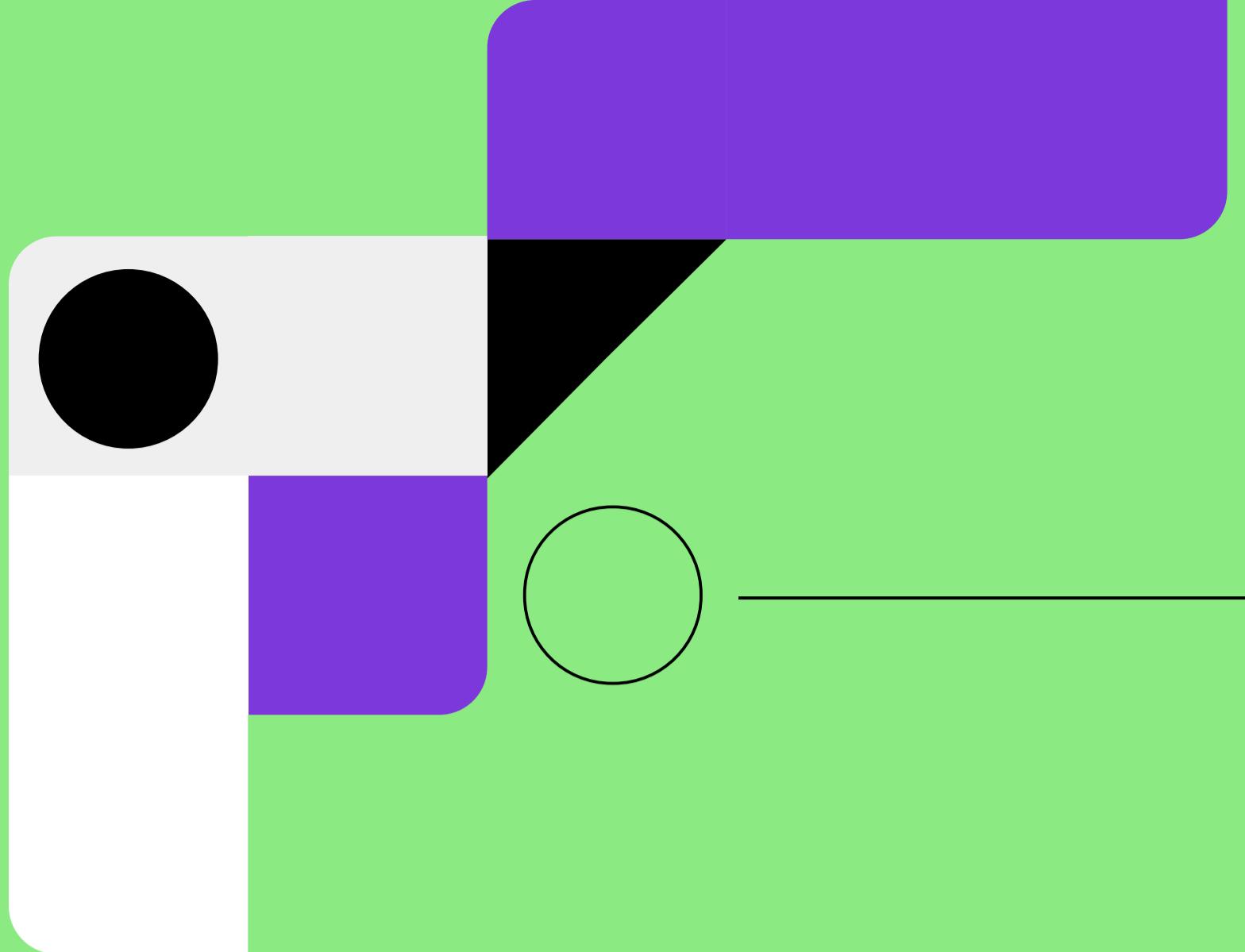


- O **--publish** cuidará exatamente deste redirecionamento de tráfego para nós. Você também pode alterar a porta de execução para outra qualquer, diferente da 8080 se quiser.
- O **--detach** diz para o Docker executar este contêiner em background, e não vai travar o processo do nosso terminal ou Visual Studio Code que acabamos de executar.
- O **--name** especifica o nome do contêiner, e fica a seu critério colocar qualquer nome que quiser, desde que não tenha espaços ou caracteres especiais.

Neste momento você pode abrir o browser no endereço <http://localhost:8080> para ver a mensagem 'Docker Node API Running' na tela, o que significa que nossa API está rodando.

Por fim temos o nome da imagem especificado, que no caso é a que acabamos de criar. Agora você pode usar os comandos **docker container ls** para ver este contêiner em execução, além do **docker container start ID** e **docker container stop ID** para iniciar ou parar ele.

Criar e executar imagens e contêineres no Docker não é uma tarefa difícil. Neste artigo vimos a instalação e primeiros passos para 'containerizar' nossas aplicações, mas ainda há muito mais a se aprender sobre Docker.



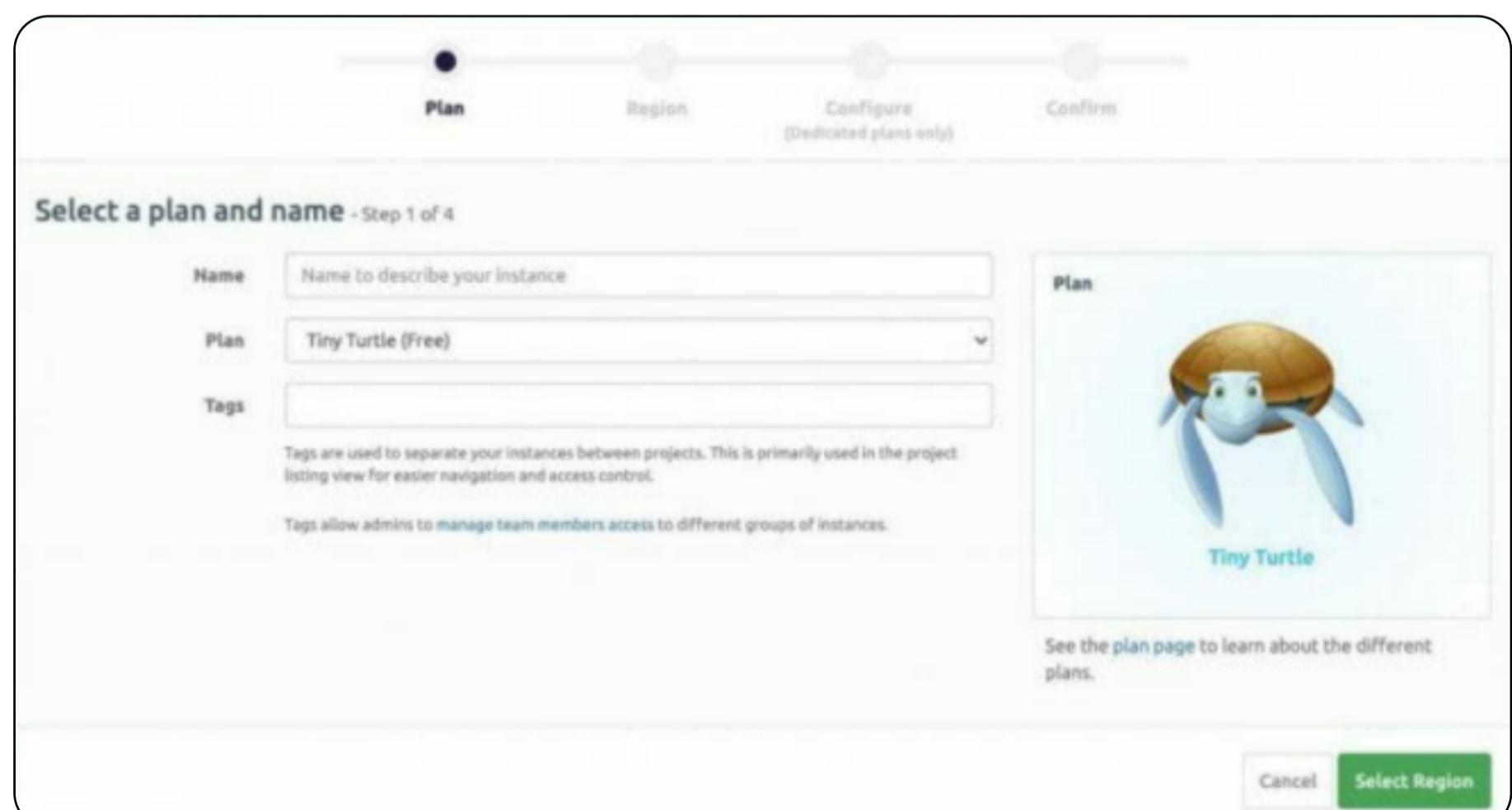
POSTGRESQL

Configurando ElephantSQL

Vou apresentar como que você pode usar Node.js com PostgreSQL.

Setup do Projeto

Iremos configurar o plano free da elephantsql.com para não ter de instalar na máquina, eles chamam de **Tiny Turtle**.



Eles provisionam a instância na Amazon, você pode selecionar a região da sua preferência, caso esteja usando outros serviços da Amazon por lá.

Quando terminar de criar sua instância, clicando no nome dela você terá acesso às credenciais de acesso, guarde-as, precisaremos delas depois para conectar no banco.

Abaixo, o SQL de criação da tabela que vamos usar, você pode executar ele na funcionalidade browse que tem no painel administrativo do ElephantSQL:

```
CREATE TABLE clientes (
    id SERIAL CONSTRAINT pk_id_cliente PRIMARY KEY,
    nome varchar(150) NOT NULL,
    idade integer NOT NULL,
    uf varchar(2) NOT NULL
)

CREATE TABLE clientes (
    id SERIAL CONSTRAINT pk_id_cliente PRIMARY KEY,
    nome varchar(150) NOT NULL,
    idade integer NOT NULL,
    uf varchar(2) NOT NULL
)
```

Agora, crie uma pasta na sua máquina para o projeto, com o nome que quiser. Dentro, crie um **db.js** e um **index.js**, ambos vazios.

Abra o terminal e dentro dessa pasta rode um “**npm init -y**” para inicializar o projeto e criar o **package.json**.

```
npm init -y
```

Ainda no terminal, mande instalar a dependência **pg**, como abaixo.

```
npm install pg
```



Criando a conexão

Agora vamos abrir o nosso **db.js** e criar a conexão com nosso banco de dados, usando o pacote que acabamos de instalar. Antes de tudo, esta será uma função assíncrona que utilizará **async/await**, logo, declaro ela usando `async`.

Segundo, não podemos sair criando conexões infinitas no banco pois isso além de ser lento é inviável do ponto de vista de infraestrutura. Usaremos aqui um conceito chamado **connection pool**, onde um objeto irá gerenciar as nossas conexões, para abrir, fechar e reutilizar conforme possível. Vamos guardar este **único pool em uma variável global**, que testamos logo no início da execução para garantir que se já tivermos um pool, que vamos utilizar o mesmo.

```
async function connect() {
  if (global.connection)
    return global.connection.connect()

  const { Pool } = require ('pg')
  const pool = new Pool({
    connectionString: 'postgres:sua_url'
  })

  //apenas testando a conexão
  const client = await pool.connect()
  console.log("Criou pool de conexões no PostgreSQL!")

  const res = await client.query('SELECT NOW()')
  console.log(res.rows[0])
  client.release()

  //guardando para usar sempre o mesmo
  global.connection = pool
  return pool.connect()
}
```



Passando essa verificação inicial, nós começamos o código da função carregando o nosso pacote pg, mais especificamente uma classe Pool dentro dele que fará o gerenciamento de conexões pra gente. Com esta classe Pool, consigo instanciar um novo pool de conexões passando a connection string fornecida pelo meu provedor. Aqui você deve substituir pelos dados da sua instalação de PostgreSQL, no formato postgres://usuario:senha@servidor:porta/banco

O código a seguir, marcado com um comentário, é apenas para testarmos se nossa conexão foi realizada com sucesso: abrimos uma conexão com o pool e usamos ela para fazer uma consulta pela hora atual do servidor, imprimindo no console o resultado.

Note que usamos o await no **client.query**, pois ele é assíncrono. Note também que depois de usar, liberamos a conexão de novo usando **client.release**. No fim da function de conexão, eu armazeno ela em uma variável global.

Importante você garantir que tanto no **if** inicial quanto no **return**, que estejamos chamando a função connect do pool para que essa conexão resultante seja usado por quem chamar esta função de conexão que criamos.

Para testar este código, crie uma chamada a esta conexão ao fim do módulo e importe-o no **index.js**, para dispará-la.

```
const db = require("./db")
```

Como resultado, irá imprimir no console a mensagem de que se conectou com sucesso no PostgreSQL.

SELECT

Certo, fizemos a conexão, e agora como vamos utilizá-la para fazer um CRUD básico no Postgres?

No mesmo módulo **db.js**, cria outra function, desta vez para select, logo abaixo da anterior.

```
async function selectCustomers() {  
  const client = await connect();  
  const res = await client.query('SELECT * FROM clientes');  
  return res.rows;  
}  
  
module.exports = { selectCustomers }
```

Nós realizamos a conexão (que internamente vai decidir reaproveitar ou não) e depois usamos a função query com este objeto client recém criado.

Note o uso de await para que fique mais fácil de gerenciar tanto o retorno da conexão quanto o retorno da query, deixando-os visualmente síncronos (eles não deixam de ser assíncronos, é apenas efeito visual ou syntax sugar).



No fim do módulo, diferente da função de connect, exportamos esta função pois queremos usar ela no nosso **index.js**.

```
//index.js
(async () => {
  const db = require("./db");
  console.log('Começou!');

  console.log('SELECT * FROM CLIENTES');
  const clientes = await db.selectCustomers();
  console.log(clientes);

})();
```

Primeiro, temos o require do nosso db, uns logs de marcação e na sequência chamo a função de selectCustomers exportada no módulo db que vai retornar o nosso array de clientes, que vou só jogar no console, mas que em uma aplicação real tenho certeza que você será mais criativo.

Como estou usando o await aqui também, e ele existe ser usado em funções async, criei uma função async anônima ao redor de todo código que automaticamente é chamada quando mandarmos executar o index.js. Chamamos isso de **IIFE**.

Faça isso agora e verá que funciona como deveria, printando no console todos os clientes do seu banco de dados, tabela clientes.



INSERT

Agora é hora de inserirmos clientes no Postgre via JavaScript.
Volte ao nosso db.js e insira uma nova função.

```
async function insertCustomer(customer){  
  const client = await connect();  
  const sql = 'INSERT INTO clientes(nome,idade,uf) VALUES ($1,$2,$3);  
  const values = [customer.nome, customer.idade, customer.uf];  
  return await client.query(sql, values);  
}
```

O SQL possui conteúdo dinâmico, que são os valores a serem inseridos na linha da tabela clientes. E embora seja muito tentado concatenar a string SQL na mão, não faça isso sob risco de SQL Injection!

Ao invés disso, coloque **\$1**, **\$2**, etc no lugar dos campos e quando chamar o client.query, o segundo parâmetro aceita um array de valores que serão corretamente substituídos na sua string SQL, já prevendo SQL Injection.



No mais, a construção desta função segue a regra das anteriores e não esqueça de exportá-la no seu db.js para usarmos no index.js, como abaixo.

```
/index.js
(async () => {
    const db = require("./db");
    console.log('Começou!');

    console.log('INSERT INTO CLIENTES');
    const result = await db.insertCustomer({
        nome: "Zé", idade: 18, uf: "SP"
    });
    console.log(result.rowCount);

    console.log('SELECT * FROM CLIENTES');
    const clientes = await db.selectCustomers();
    console.log(clientes);
})();
```

SELECT veio antes, para que fique mais fácil de ver se ele funcionou ou não.

Também optei por printar o rowCount do resultado, que talvez seja a única informação que lhe interesse pois, no fundo, se der algum erro na execução do SQL, vai disparar uma exception que vai derrubar essa nossa aplicação bem simples. Sim, você pode embrulhar tudo com try/catch para tratar os erros.

UPDATE

```
async function updateCustomer(id, customer){  
    const client = await connect();  
    const sql = 'UPDATE clientes SET nome=$1, idade=$2, uf=$3 WHERE id=$4';  
    const values = [customer.nome, customer.idade, customer.uf, id];  
    return await client.query(sql, values);  
}
```

Note o mesmo padrão de construção do INSERT, com a leve diferença que o update requer um ID, importantíssimo nesse tipo de comando SQL.

E no index.js, seguimos a mesma estrutura do INSERT também, apenas tome cuidado para informar um ID existente na sua base de dados ao invés do meu '1', ou até mesmo modifique o código para pegar um dos ids retornados pelo SELECT anterior.

```
console.log('UPDATE CLIENTES');  
const result2 = await db.updateCustomer(1, {nome: "Zé José", idade: 19});  
console.log(result2.rowCount);
```



DELETE

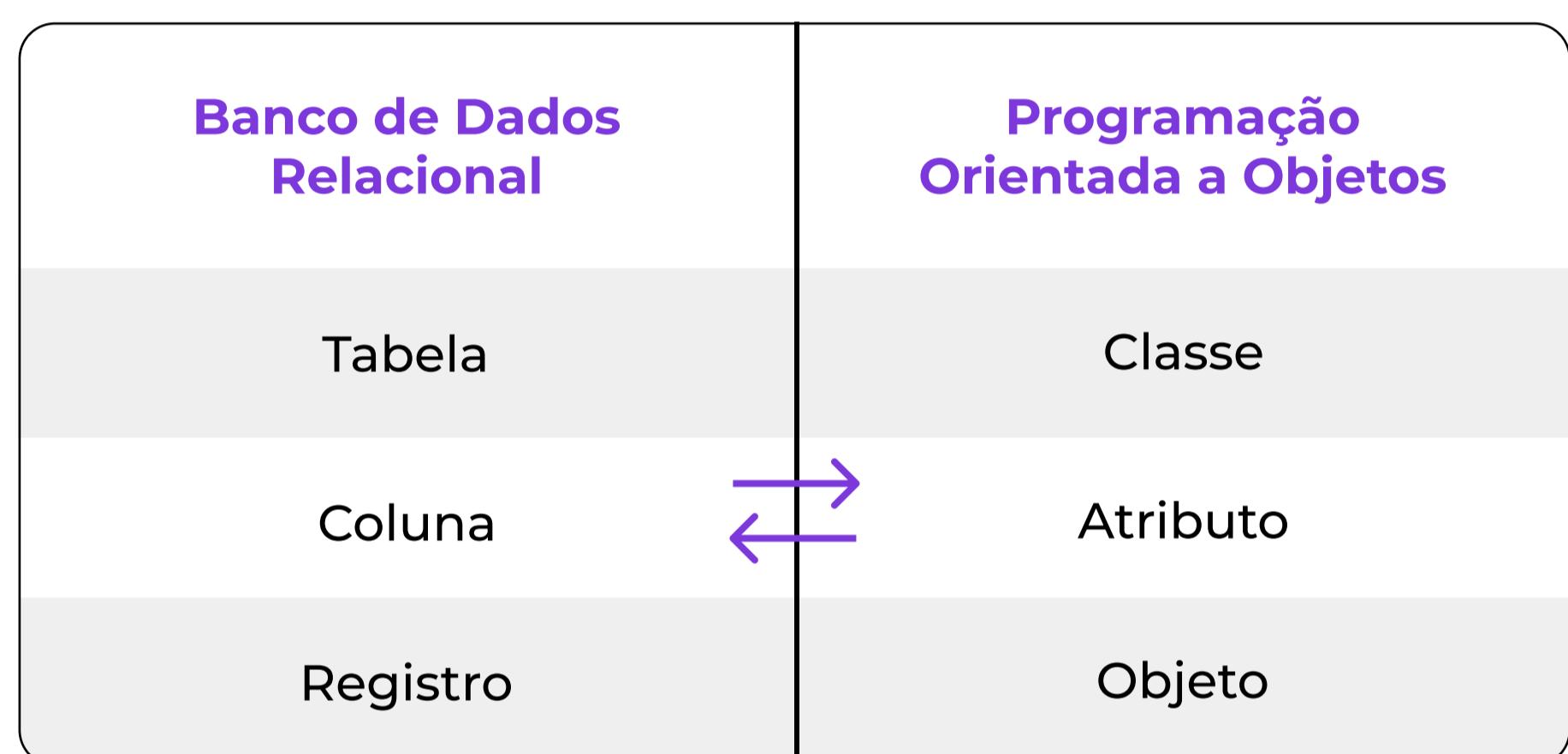
E para finalizar, nosso deleteCustomer no db.js.

```
async function deleteCustomer(id){  
  const client = await connect();  
  const sql = 'DELETE FROM clientes where id=$1';  
  return await client.query(sql, [id]);  
}  
  
module.exports = {  
  selectCustomers, insertCustomer, updateCustomer, deleteCustomer  
}
```

```
console.log('DELETE FROM CLIENTES');  
const result3 = await db.deleteCustomer(1);  
console.log(result3.rowCount);
```

ORM

O **ORM (Object Relational Mapping)** ou Mapeamento Objeto Relacional se trata de uma técnica na programação que consiste em fazer a ponte entre modelo relacional (banco de dados) e objetos (aplicação), através de frameworks capazes de converter dados entre sistemas utilizando linguagens de programação orientada a objetos, por exemplo, JAVA, C#, PHP, entre outras.



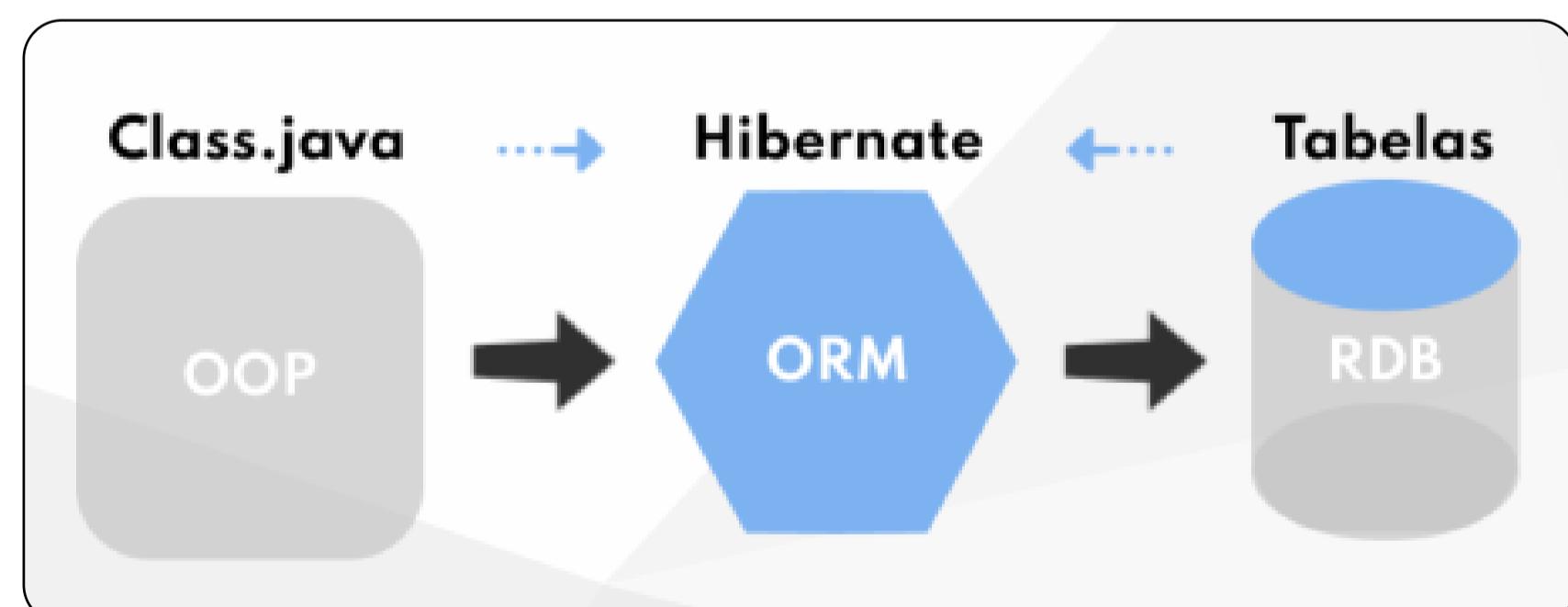
Como demonstrado na imagem acima, ocorre a conversão de classes para tabelas e vice-versa assim como as demais informações quando utilizamos um framework ORM em nossa aplicação.

Isso facilita demais no desenvolvimento e evita a necessidade de termos que ficar escrevendo queries (consultas) no banco de dados ou mesmo o oposto.

Vejamos alguns frameworks ORM disponível no mercado:

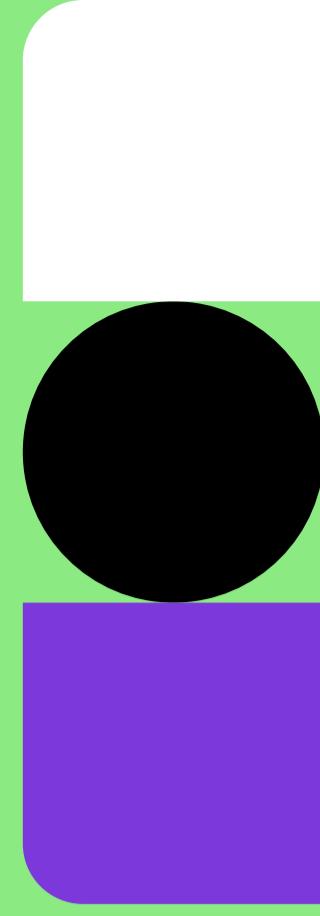
```
[  
  {  
    "linguagem" : "Java",  
    "framework" : [  
      "Hibernate", "ActiveJPA", "Cayenne", "Ebean", "JOOQ"  
    ]  
  },  
  {  
    "linguagem": "Kotlin",  
    "framework": [ "Exposed", "Ktorm" ]  
  },  
  {  
    "linguagem" : "Python",  
    "framework" : [ "DjangoORM", "SqlAlchemy" ]  
  },  
  {  
    "linguagem" : "Node",  
    "framework" : [ "Sequelize", "TypeORM" ]  
  }  
]
```

Existem muitos outros frameworks ORM criados para facilitar o desenvolvimento de aplicações, até o JavaScript que é Orientado a Protótipos não ficou de fora com o Sequelize ORM para Node.js.



OOP/ POO = Programação Orientada a Objetos

RDB = Banco de Dados Relacional



Breve tutorial
TypeORM
com Node

Instale o **typeorm** e o **reflect-metadata** pelo **npm**:

```
npm install typeorm reflect-metadata -save
```

É importante que o **reflect-metadata** seja importado no arquivo **main** da aplicação Node (aqui nesse exemplo será **app.ts**):

```
import "reflect-metadata";
```

Você vai precisar instalar as tipagens do Node:

```
npm install @types/node -save
```

Instale o driver da base de dados que irá utilizar. (aqui usarei o Postgre como exemplo)

```
npm install pg -save
```

Configuração Typescript

- Certifique que o seu Typescript é igual ou superior a versão 3.3
- Habilite o **"emitDecoratorMetadata":true** e o **"experimentalDecorators":true** no **tsconfig.json**
- Habilitar o **es6** na seção de compilador do seu **tsconfig.json**



Primeiros Passos

Para usarmos o CLI do Typeorm, é necessário que o instalamos globalmente em nossas máquinas:

```
npm install typeorm -global
```

E para criar uma Aplicação do zero, com Express e Postgre com o CLI do TypeORM, basta executar o seguinte comando:

```
typeorm init -name 03-typeorm-example -database postgres -express
```

Migrations

Durante a vida de uma aplicação, e principalmente durante o seu momento inicial de desenvolvimento, é muito comum ter o surgimento de demandas de migração da base de dados com a aplicação para que ambas estruturas estejam sincronizadas. Esse processo feito de forma manual, depende de uma comunicação estreita com a equipe de desenvolvimento para que não haja desencontros.

E uma das estratégias que muitos ORM's oferecem, para aperfeiçoar esse processo, é o uso de migrations. No TypeORM isso é possível através do seu CLI.



Script Package

Para executar o comando typeorm vamos criar o trecho do script, adicione o trecho de código no **package.json**:

```
"scripts": {  
  "dev:server": "ts-node-dev -r tsconfig-paths/register --inspect  
    --transpile-only --ignore-watch node_modules src/server.ts",  
  "start": "ts-node src/server.ts",  
  "typeorm": "ts-node-dev -r tsconfig-paths/register  
    ./node_modules/typeorm/cli.js"  
},
```

Para poder criar e acessar a base de dados temos que criar as configurações, para isso crie na raiz do projeto o arquivo **ormconfig.json**.

No arquivo **ormconfig.json** coloque o código abaixo:

```
[{  
  "name": "default",  
  "type": "postgres",  
  "host": "localhost",  
  "port": 5432,  
  "username": "postgres",  
  "password": "docker",  
  "database": "baseOrm",  
  "entities": ["./src/models/**/*.ts"],  
  "migrations": ["./src/migrations/*.ts"],  
  "cli": {  
    "migrationsDir": "./src/migrations/"  
  }  
}]
```



- **type:** Tipo de base de dados que pode ser: mysql, postgres, cockroachdb, mariadb, sqlite, better-sqlite3, cordova, nativescript, oracle, mssql, mongodb, sqljs, react-native;
- **host:** Se for usar acesso a banco de dados remoto ou uma VM, utilize o IP;
- **port:** porta de acesso ao banco de dados;
- **username:** usuário com acesso ao banco de dados;
- **password:** senha de acesso ao banco de dados;
- **database:** nome da base de dados;
- **entities:** local onde vamos criar nossas entidades, essas entidades são as que vamos mapear;
- **migrations:** informa o local onde nossas migrations são carregadas;
- **migrationsDir:** local onde as migrations devem ser criada pelo CLI;

Entidades

Para exemplificar vou criar duas entidades com relacionamento one-to-one são elas: Profile e User

profile.ts

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm";

@Entity("profiles")
export default class Profile {
  @PrimaryGeneratedColumn("uuid")
  id: string;

  @Column()
  gender: string;

  @Column()
  photo: string;
}
```



```
// user.ts
import{
    Entity,
    PrimaryGeneratedColumn,
    Column,
    OneToOne,
    Join Column
} from "typeorm";
import Profile from "./profile";
@Entity("users")
export default class User{
    @PrimaryGeneratedColumn("uuid")
    id: string;
    @Column()
    name: string;
    @Column()
    profile_id: string;
    @OneToOne(type => Profile)
    @JoinColumn({ name: "profile_id" })
    profile: Profile;
}
```

Como pode ser observado temos duas entidades: **user.ts que possui profile.ts**.

TypeORM CLI

Depois de adicionar os pacotes, configuração os dados de acesso ao banco de dados e criar nossas entidades agora é hora de executar o comando para criar as tabelas.

a) Criar migrations

```
yarn typeorm migration:create -n CreateProfile
yarn typeorm migration:create -n CreateUser
```



Migrations

Após executar o passo acima será criados as migrations, temos que adicionar os códigos que vão criar as tabelas, segue abaixo:

a) Migration Profile

```
import { MigrationInterface, QueryRunner, Table } from "typeorm";
export default class CreateProfile implements MigrationInterface {
  public async up(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.createTable(
      new Table({
        name: "profiles",
        columns: [
          {
            name: "id",
            type: "uuid",
            isPrimary: true,
            generationStrategy: "uuid",
            default: "uuid_generate_v4()"
          }
        ]
      })
    );
  }

  public async down(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.dropTable("profiles");
  }
}
```

b) Migration User

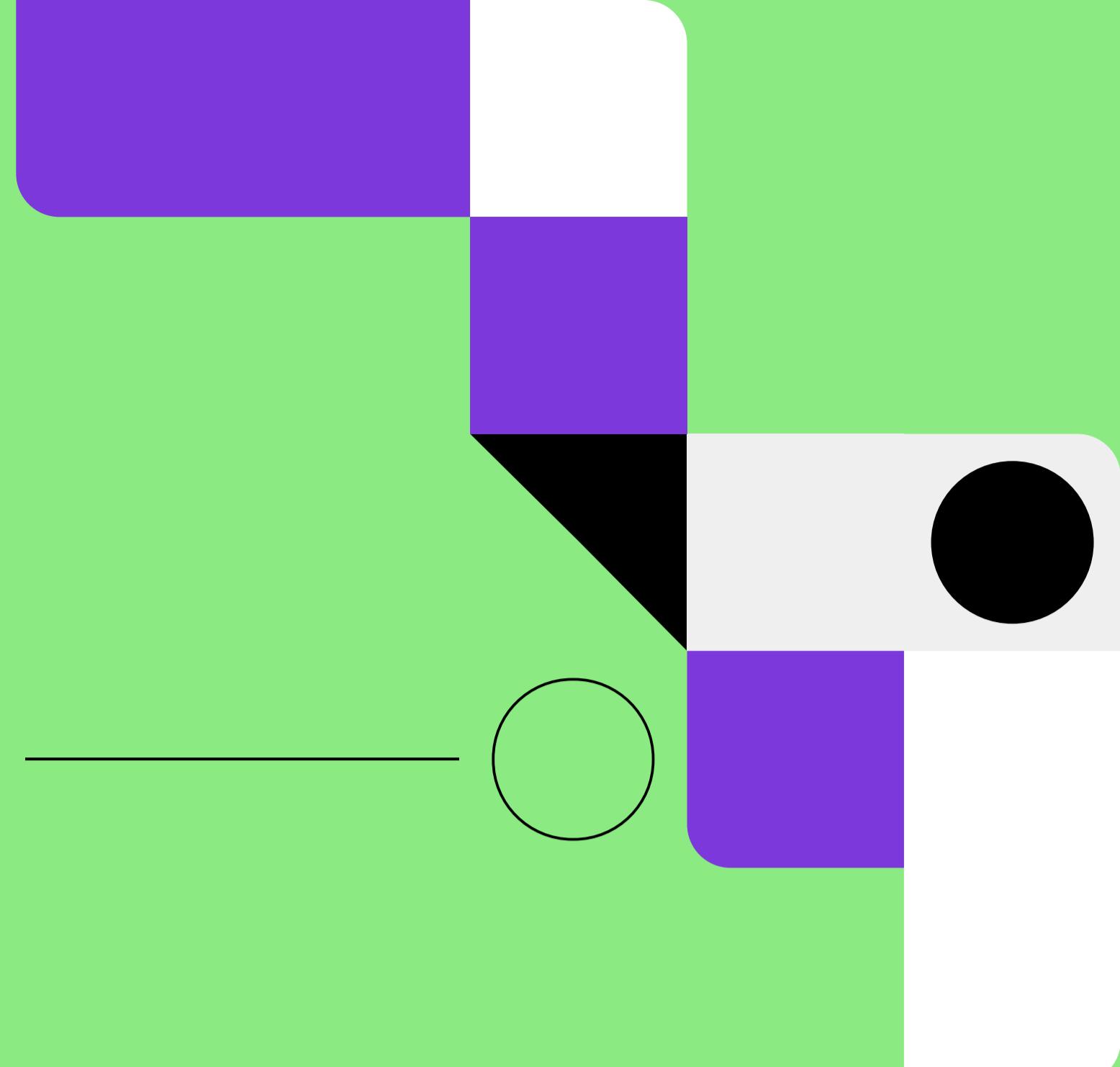
```
import Migration Interface,QueryRunner,Table}from"typeorm";
export default class Create Profile implements Migration Interface{
  public async up(queryRunner:QueryRunner):Promise<void>{
    await queryRunner.createTable(
      new Table({
        name:"profiles",
        columns:[
          {
            name:"id",
            type:"uuid",
            is Primary:true,
            generationStrategy:"uuid",
            default:"uuid_generate_v4()"
          },
        );
      });
    public async down(queryRunner:QueryRunner):Promise<void>{
      await queryRunner.dropTable("profiles");
    }
}
```

Após criar as migrations vamos executar o comando para rodar as migrations e criar as tabelas:

```
yarn typeorm migration:run
```

Caso queira reverter a criação das tabelas pode ser executado o comando:

```
yarn typeorm migration:revert
```



Criando um **password hash** para seu usuário no **banco de dados**

Vamos ver uma forma de manter a segurança da senha dos usuários que você irá cadastrar no banco de dados, o que nada mais é do que o **password hash**.

Usaremos as tecnologias: Postgres, express, nodejs e sequelize, e biblioteca que foi utilizada como **hash bcryptjs**.

O model **User.js** está assim:

```
import Sequelize, {  
  Model  
} from 'sequelize';  
class User extends Model {  
  static init(sequelize) {  
    super.init({  
      name: Sequelize.STRING,  
      email: Sequelize.STRING,  
      password: Sequelize.STRING,  
    }, {  
      sequelize,  
    });  
    return this;  
  }  
}  
export default User;
```

E o **UserController** está assim:

```
import User from '../models/User';  
class UserController {  
  async store(req, res) {  
    const {  
      name,  
      email  
    } = await User.create(  
      req.body,  
    );  
    return res.json({  
      name,  
      email  
    });  
  }  
  export default new UserController();
```

Se a gente inserir um novo usuário (user) assim, você perceberá que seu sistema não está muito seguro, porque a senha está muito simples e visível no banco de dados.

O que iremos fazer agora é mudar a migration do usuário para trocar o campo de password para password_hash como mostrado na tabela users:

The screenshot shows the pgAdmin interface for managing a PostgreSQL database. On the left, there's a sidebar with various navigation options like Columns, Constraints, Foreign Keys, etc. The main area displays the schema of a table named 'users'. The table has 7 columns:

Column Name	#	Data type	Length	Precision	Scale	Identity	Collation	Not Null	Default
id	1	int4		10				<input checked="" type="checkbox"/>	nextval('users_id_seq'::regclass)
name	2	varchar	255	255			default	<input type="checkbox"/>	
email	3	varchar	255	255			default	<input type="checkbox"/>	
password_hash	4	varchar	255	255			default	<input checked="" type="checkbox"/>	
created_at	5	timestamptz		35	6			<input checked="" type="checkbox"/>	
updated_at	6	timestamptz		35	6			<input type="checkbox"/>	
deleted_at	7	timestamptz		35	6			<input type="checkbox"/>	

At the bottom of the pgAdmin window, there are several icons: a magnifying glass for search, a pencil for edit, a star for favorite, a save icon labeled 'Guardar', a revert icon labeled 'Reverter', and a refresh/reload icon labeled 'Renovar'. The status bar at the bottom left shows '7 items'.

No model User podemos passar o **password_hash** como **sequelize.STRING**. Depois criamos um campo virtual para password, pois iremos usar um addHook que antes de salvar no banco de dados, vai passar pela criptografia do hash:

```

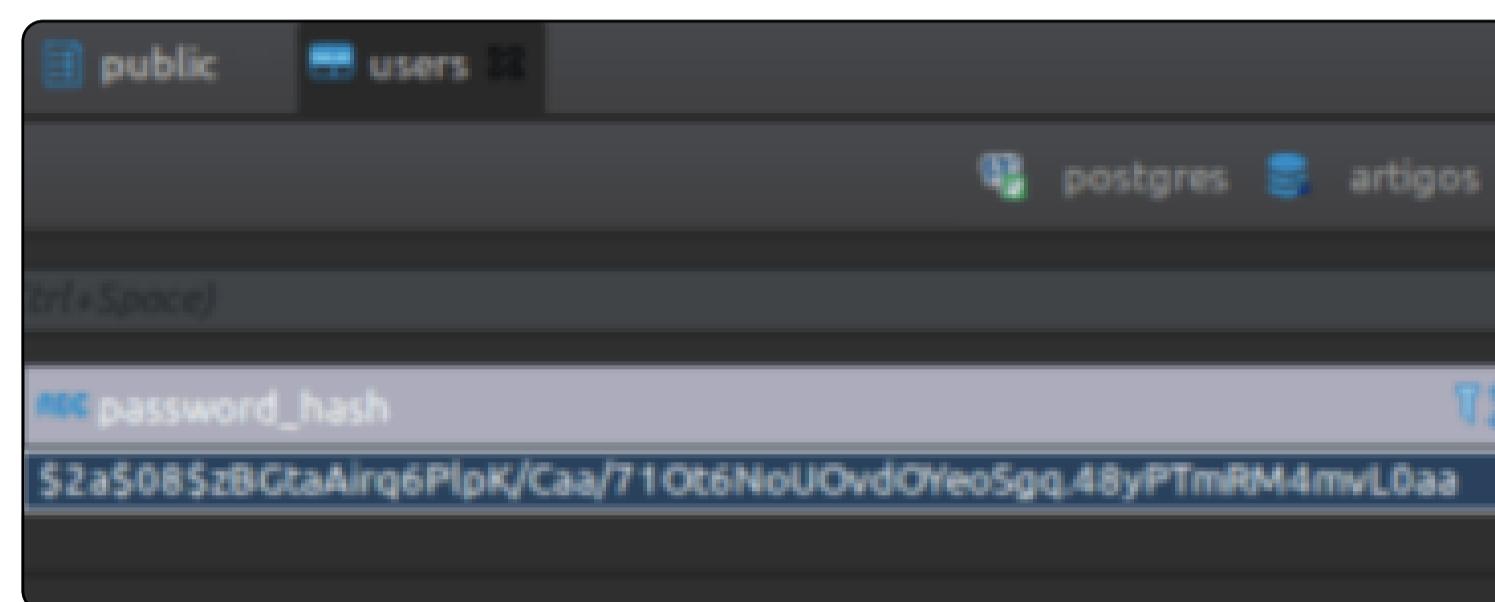
import Sequelize, { Model } from 'sequelize';
import bcrypt from 'bcryptjs';
class User extends Model {
  static init(sequelize) {
    super.init({
      name: Sequelize.STRING,
      email: Sequelize.STRING,
      password: Sequelize.VIRTUAL,
      password_hash: Sequelize.STRING,
    }, { sequelize });
    // password hash
    this.addHook('beforeSave', async client => {
      if (client.password) {
        client.password_hash = await bcrypt.hash(client.password, 8);
      }
    });
    return this;
  }
}
export default User;

```

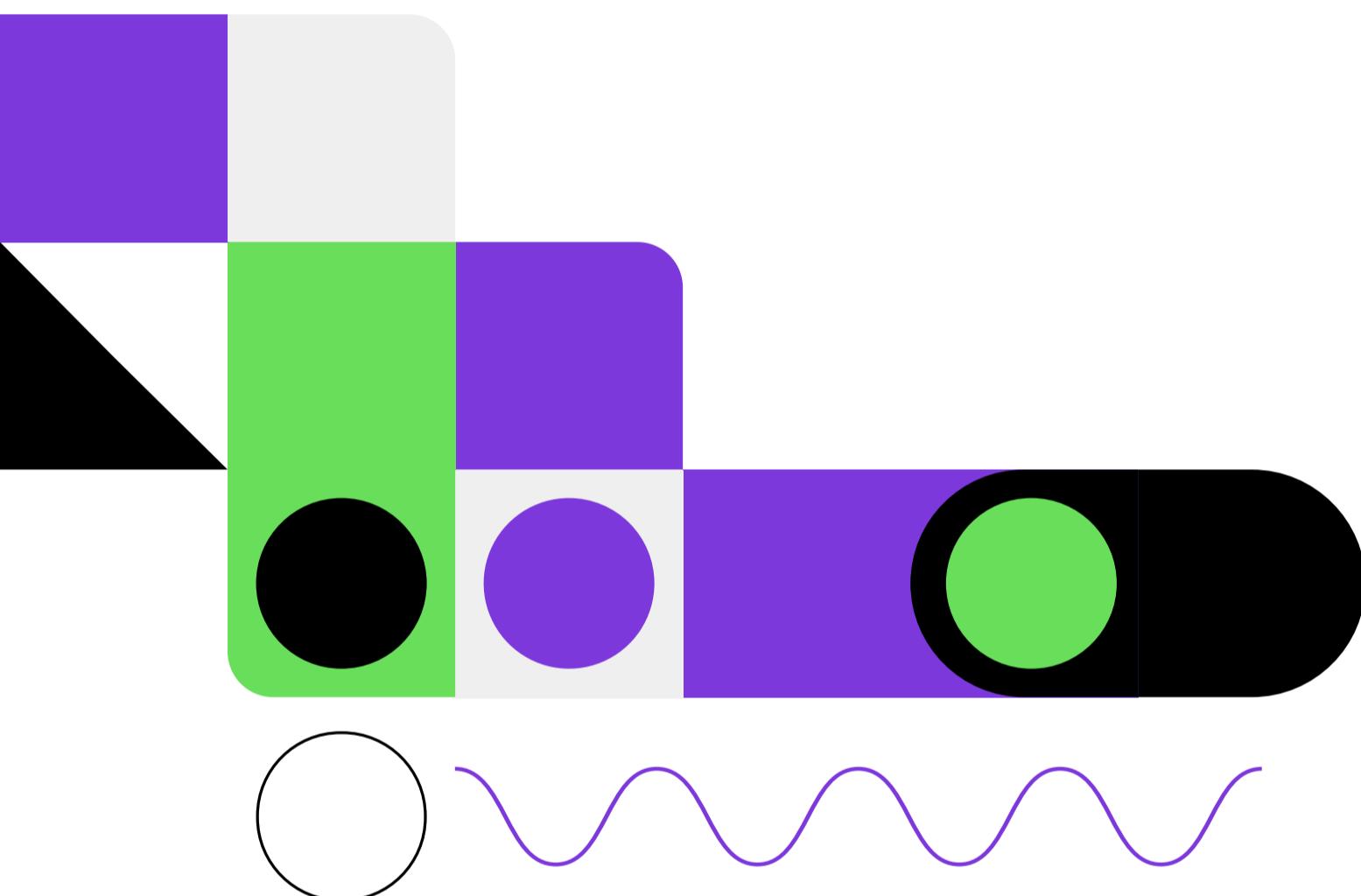
Com isso agora no UserController iremos utilizar a função hash da biblioteca bcryptjs para transformar a password passada no req.body serencriptografada. O segundo parâmetro é o salt, que será o número de vezes que a senha será misturada. Aqui usarei 8, sendo que demora cada vez mais tempo para criptografar de acordo com o número de rodadas, como visto na documentação do **bcryptjs**:

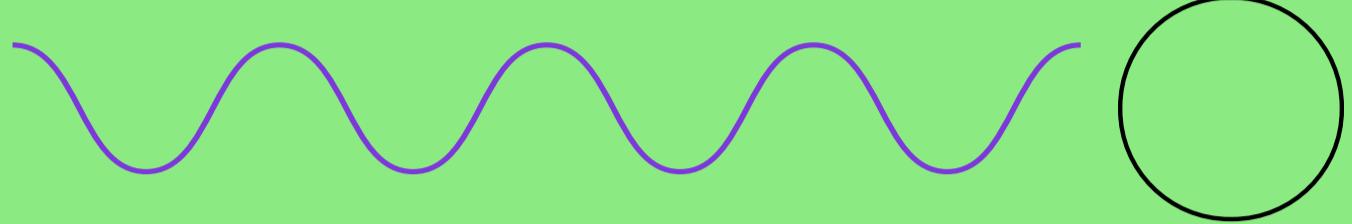
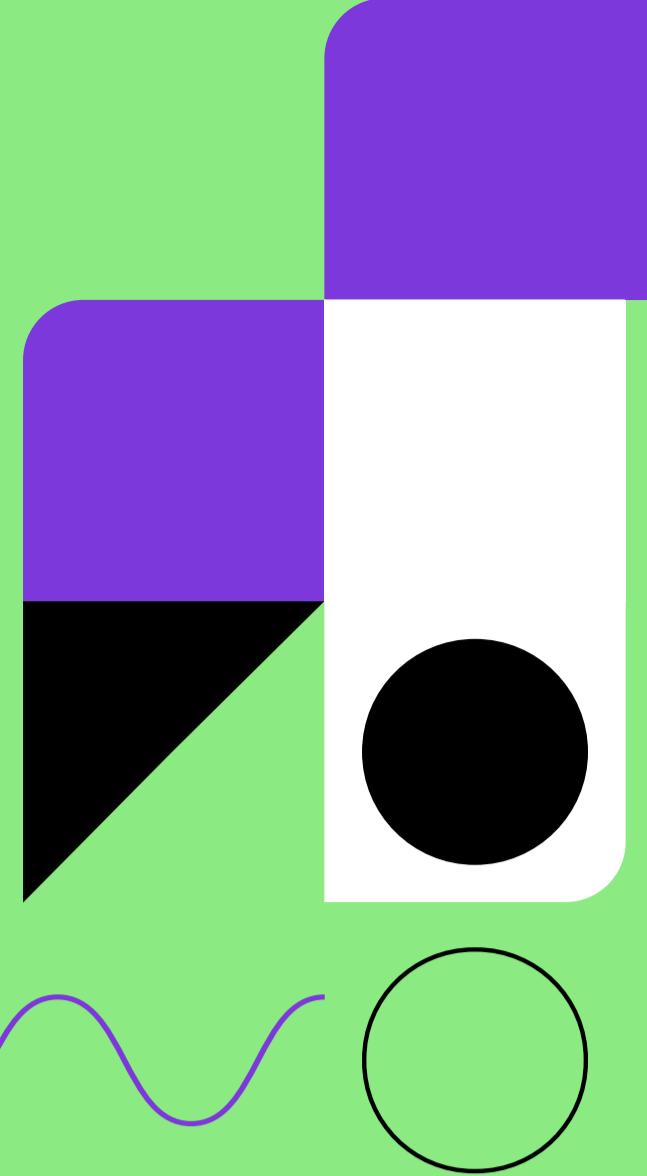
rounds=8 : ~40 hashes/sec
rounds=9 : ~20 hashes/sec
rounds=10: ~10 hashes/sec
rounds=11: ~5 hashes/sec
rounds=12: 2-3 hashes/sec
rounds=13: ~1 sec/hash
rounds=14: ~1.5 sec/hash
rounds=15: ~3 sec/hash
rounds=25: ~1 hour/hash
rounds=31: 2-3 days/hash

E olha como ficou o nosso password_hash no banco de dados com um usuário criado com a mesma senha 123456:



A screenshot of a PostgreSQL database interface. The top bar shows 'public' and 'users 12'. Below the bar, there are icons for 'postgres' and 'artigos'. The main area shows a table with one row. The first column is labeled 'password_hash' and contains the value '52a5085zBGtaAiq6PlpK/Caa/71Ot6NoUOvdOYeo5gq.48yPTmRM4ImwL0aa'.





AUTENTICAÇÃO JWT



Vamos abordar JSON Web Tokens como uma forma de garantir a autenticação e autorização de APIs de maneira bem simples e segura, sendo o JWT um padrão para segurança de APIs RESTful atualmente.

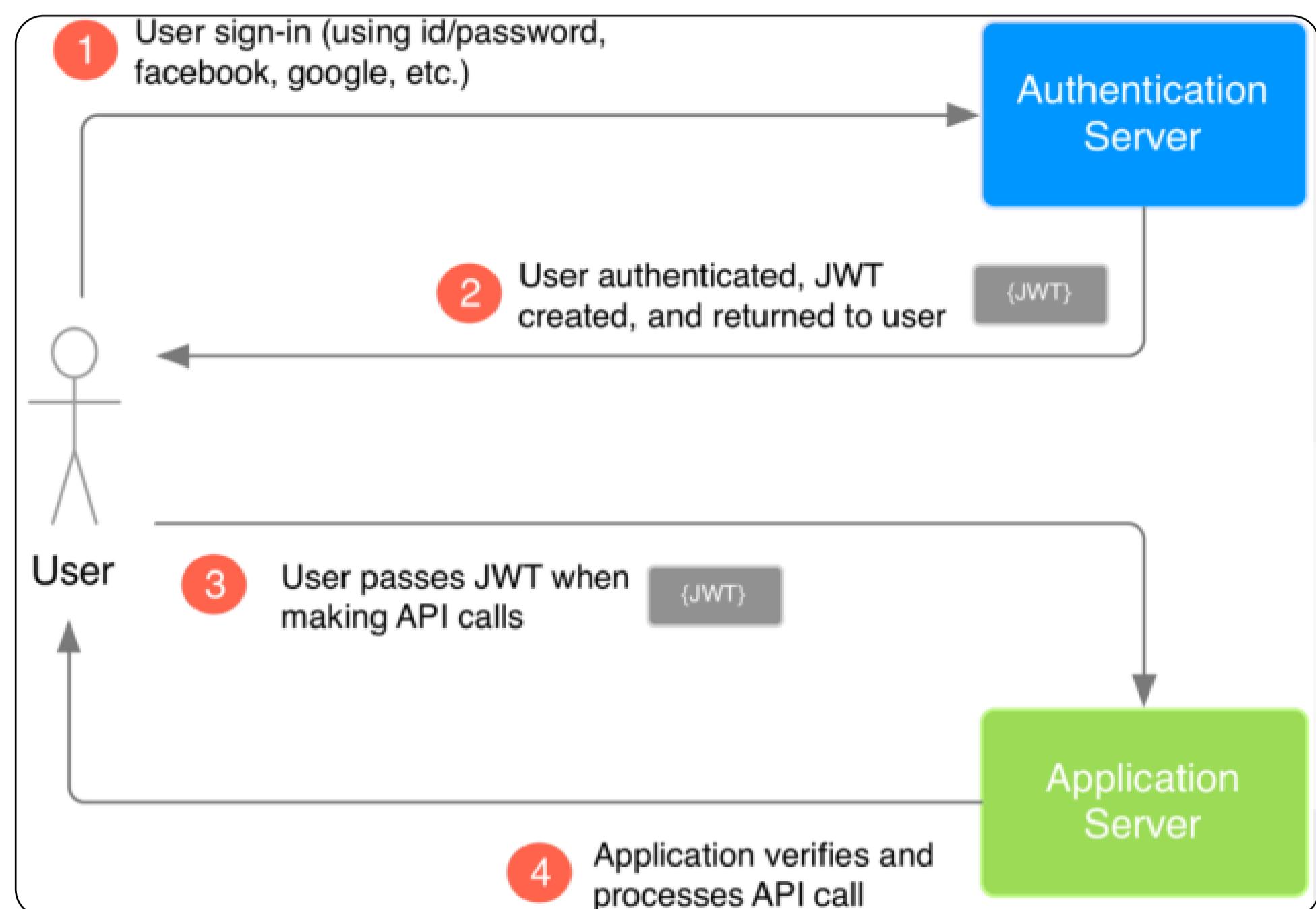
1 – JSON Web Tokens

JWT, resumidamente, é uma string de caracteres que, caso cliente e servidor estejam sob HTTPS, permite que somente o servidor que conhece o ‘segredo’ possa validar o conteúdo do token e assim confirmar a autenticidade do cliente. O token não é “criptografado”, mas “assinado”, de forma que só com o secret essa assinatura possa ser comprovada, o que impede que atacantes “criem” tokens por conta própria.

Em termos práticos, quando um usuário se autentica no sistema ou web API (com usuário e senha), o servidor gera um token com data de expiração pra ele. Durante as requisições seguintes do cliente, o JWT é enviado no cabeçalho da requisição e, caso esteja válido, a API irá permitir acesso aos recursos solicitados, sem a necessidade de se autenticar novamente.



O diagrama a seguir mostra este fluxo, passo-a-passo:



O conteúdo do JWT é um payload JSON que pode conter a informação que você desejar, que lhe permita mais tarde conceder autorização a determinados recursos para determinados usuários. Minimamente ele terá o ID do usuário autenticado ou da sessão (se estiver trabalhando com este conceito), mas pode conter muito mais do que isso, conforme a sua necessidade, embora guardar conteúdos “sensíveis” no seu interior não é uma boa ideia, pois como disse antes, ele não é criptografado.

2 – Estruturando a API

Vamos mockar os dados retornados pela API e as credenciais de autenticação inicial para ir logo para a geração e posterior verificação dos tokens. Exemplo: (salve o código abaixo no arquivo: /jwt/index.js)

```
//index.js
const http = require('http');
const express = require('express');
const app = express();

const bodyParser = require('body-parser');
app.use(bodyParser.json());

app.get('/', (req, res, next) => {
    res.json({message: "OK!"});
})

app.get('/clientes', (req, res, next) => {
    console.log("Retornou todos clientes!");
    res.json([{id: 1, nome: 'Marianne'}]);
})

const server = http.createServer(app);
server.listen(3000);
console.log("Servidor escutando na porta 3000 ...")
```

Com esse JS em mãos, vamos instalar algumas dependências na nossa aplicação para fazê-la funcionar:

```
npm i express body-parser
```

Rode a API com “**node index**” e ao acessar localhost:3000 deve listar apenas uma mensagem de OK e ao acessar o caminho /clientes no navegador, deve listar um array JSON como abaixo.



```
[{id:1, nome:'Marianne'}]
```

Isso mostra que a API está funcionando em ambas as rotas e sem segurança, afinal, não tivemos de nos autenticar para fazer os GETs que fizemos.

3 – Adicionando o JWT

Agora, vamos instalar duas novas dependências para incrementar o nosso projeto, permitindo adicionar autenticação via JWT:

```
npm i jsonwebtoken dotenv-safe
```

A saber:

- **jsonwebtoken**: pacote que implementa o protocolo JSON Web Token;
- **dotenv-safe**: pacote para gerenciar facilmente variáveis de ambiente, não é obrigatório para JWT, mas uma boa prática para configurações em geral;

Vamos começar usando o **dotenv-safe**, criando dois arquivos. Primeiro, o arquivo **.env.example**, com o template de variáveis de ambiente que vamos precisar:

```
# .env.example, commit to repo  
SECRET=
```

E depois, o arquivo **.env**, com o valor do segredo à sua escolha:

```
#.env, don't commit to repo  
SECRET=mysecret
```

Este segredo será utilizado pela biblioteca jsonwebtoken para assinar o token de modo que somente o servidor consiga validá-lo, então é de bom tom que seja um segredo forte.

Para que esse arquivo de variáveis de ambiente seja carregado assim que a aplicação iniciar, adicione a seguinte linha logo no início do arquivo **index.js** da sua API, aproveitando para inserir também as linhas dos novos pacotes que vamos trabalhar:

```
require("dotenv-safe").config();  
const jwt = require('jsonwebtoken');
```

Isso deixa nossa API preparada para de fato lidar com a autenticação e autorização.

4 – Autenticação

Caso você não saiba a diferença, **autenticação é você provar que você é você mesmo**. Já **autorização é você provar que possui permissão** para fazer ou ver o que você está tentando.

Antes de gerar o JWT é necessário que o usuário passe por uma autenticação tradicional, geralmente com usuário e senha. Essa informação fornecida é validada junto a uma base de dados e somente caso ela esteja ok é que geramos o JWT para ele.

Assim, vamos criar uma nova rota /login que vai receber um usuário e senha hipotético e, caso esteja ok, retornará um JWT para o cliente:

```
//authentication
app.post('/login', (req, res, next) => {
  //esse teste abaixo deve ser feito no seu banco de dados
  if(req.body.user = 'marianne' && req.body.password = '123'){
    //auth ok
    const id = 1; //esse id viria do banco de dados
    const token = jwt.sign({ id }, process.env.SECRET, {
      expires In: 300 // expires in 5min
    });
    return res.json({ auth:true, token: token });
  }
  res.status(500).json({message: 'Login inválido!'});
})
```

Aqui temos o seguinte cenário: o cliente posta na URL /login um user e um pwd, que simula uma ida ao banco meramente verificando se user é igual a marianne e se pwd é igual a 123. Estando ok, o banco me retornaria o ID deste usuário.

Esse ID está sendo usado como payload do JWT que está sendo assinado, mas poderia ter mais informações conforme a sua necessidade. Além do payload, é passado o SECRET, que está armazenado em uma variável de ambiente como mandam as boas práticas de segurança. Por fim, adicionei uma expiração de 5 minutos para este token, o que quer dizer que o usuário autenticado poderá fazer suas requisições por 5 minutos antes do sistema ou Web API pedir que ele se autentique novamente. Caso o user e password não coincidam, será devolvido um erro ao usuário.



5 – Autorização

Vamos criar uma função de verificação em nosso index.js, com o intuito de, dada uma requisição que está chegando, a gente verifica se ela possui um JWT válido:

```
function verifyJWT(req, res, next){  
  const token = req.headers['x-access-token'];  
  if (!token) return res.status(401).json({  
    auth: false, message: 'No token provided.'  
  });  
  
  jwt.verify(token, process.env.SECRET, function(err, decoded) {  
    if (err) return res.status(500).json({  
      auth: false, message: 'Failed to authenticate token.'  
    });  
  
    // se tudo estiver ok, salva no request para uso posterior  
    req.userId = decoded.id;  
    next();  
  });  
}
```

O token é obtido a partir do cabeçalho **x-access- token**, que se não existir já gera um erro logo de primeira.

Caso exista, verificamos a autenticidade desse token usando a função verify, usando a variável de ambiente com o SECRET. Caso ele não consiga verificar o token, irá gerar um erro. Em seguida chamamos a função next que passa para o próximo estágio de execução das funções no pipeline do middleware do Express, mas não antes de salvar a informação do id do usuário para a requisição, visando poder ser utilizado pelo próximo estágio.

Esta função atuará como um middleware, mas como a usaremos?



Basta inserirmos sua referência na chamada GET /clientes que já existia em nossa API:

```
app.get('/clientes', verifyJWT, (req, res, next) => {
  console.log("Retornou todos clientes!");
  res.json([{id:1, nome:'marianne'}]);
})
```

Assim, antes de responder os GETs de clientes, a API vai criar essa camada intermediária de autorização baseada em JWT, que obviamente vai bloquear requisições que não estejam autenticadas e autorizadas, conforme as suas regras para tal.

O resultado, tentando chamar a rota /clientes sem estar autenticado é esse:

```
{ "auth": false, "message": "No token provided." }
```

Para que seja possível acessá-la o cliente da API deve primeiro obter um token se autenticando com um usuário válido na rota POST de login. Essa requisição teremos de realizar usando POSTMAN, Insomnia ou similar (cURL,etc):

Rota **POST**: localhost:3000/login

```
{
  "auth": true,
  "token": "passe_o_token"
}
```

A saída permitida será:

```
[{ id: 1, nome: 'Marianne' }]
```

Bom, concluímos esse módulo por aqui.

No próximo daremos seguimento para mais conteúdos extremamente importantes para o mercado de trabalho.

Referências

- <https://blog.tecnospeed.com.br/>
- <https://balta.io/blog/>
- <https://renan04-marques.medium.com/>
- <https://www.luiztools.com.br/>

