



Linguagem de Programação Web de Servidor

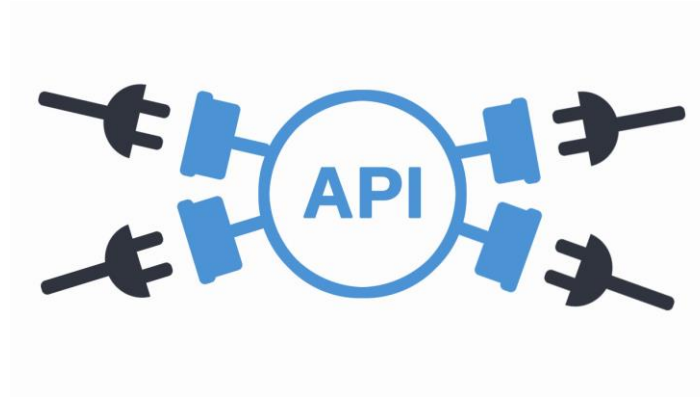
PHP / Laravel

Sara Monteiro

API

- Para finalizar o estudo do Laravel, iremos usar o recurso de criar uma API. Faremos a mesma usando a arquitectura RESTful.
- Uma API RESTful (Representational State Transfer) é um estilo que usa solicitações HTTP para acesso e uso de dados. Esses dados podem ser usados para os tipos de dados GET, PUT, POST e DELETE e referem-se à leitura, atualização, criação e exclusão de operações relativas a recursos.
- Temos ainda o protocolo de acesso a objetos simples (SOAP), mantido pelo World Wide Web Consortium (W3C).
- A principal diferença é que SOAP é um protocolo enquanto REST é um conjunto de princípios de arquitectura.

API



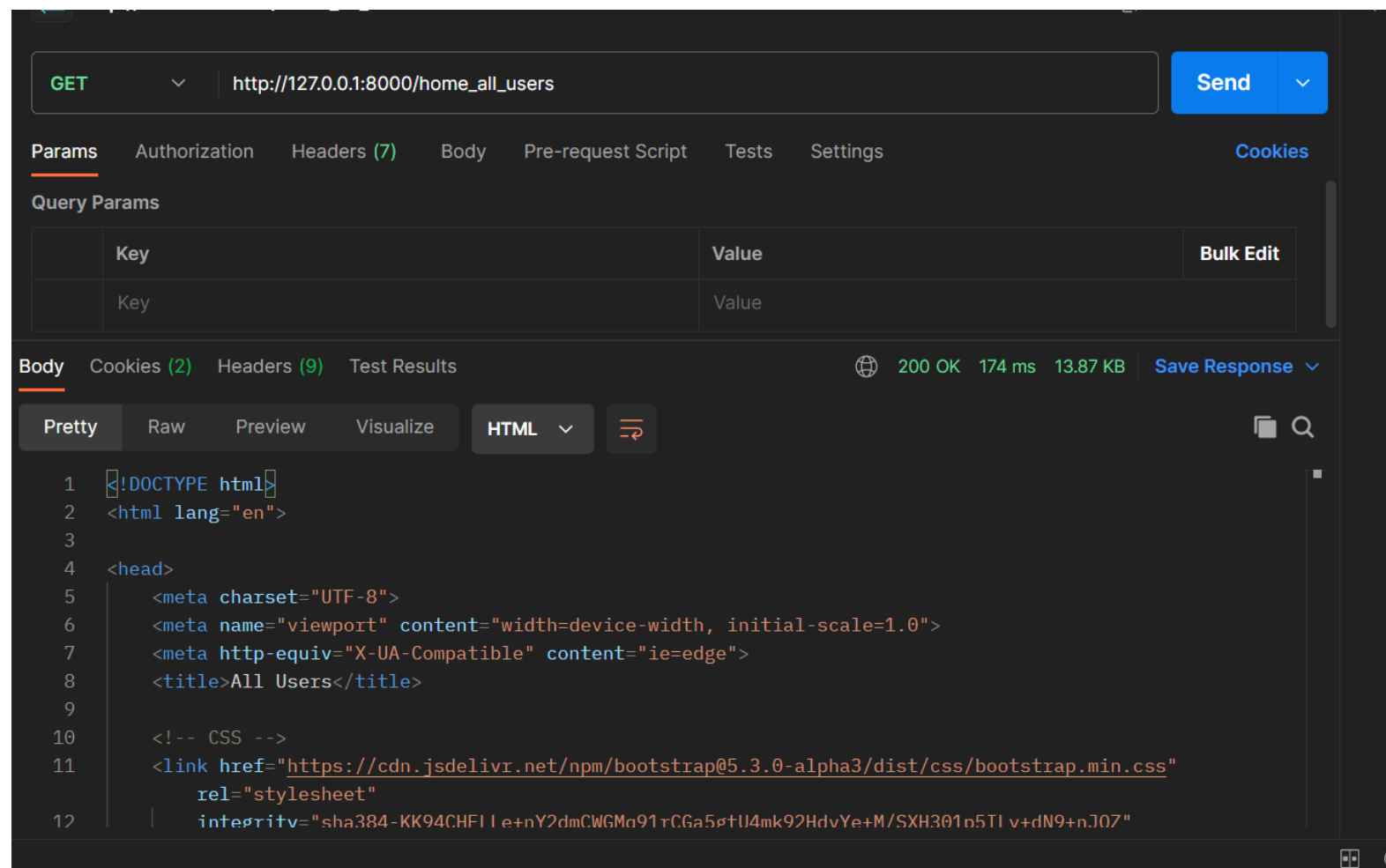
Application Programming Interface

- **Application:** Software que corre tarefas, como o Google Maps que nos dá direcções.
- **Programming:** onde damos instruções à aplicação para correr as tarefas para nós.
- **Interface:** o lugar onde as entidades comunicam um com a outra.

Pedaço de Software que permite que uma aplicação comunique com outra aplicação.

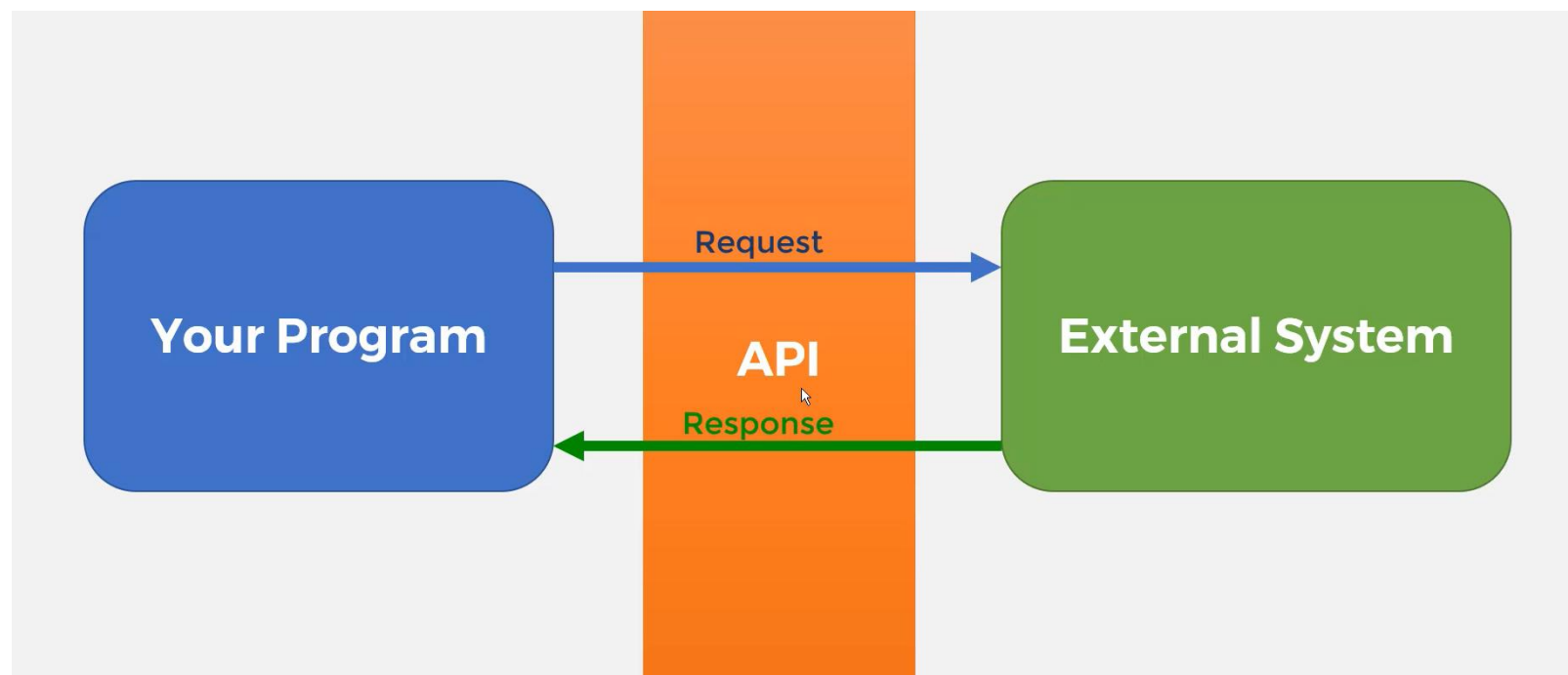
API – Postman

- O melhor recurso para testar APIs é o Postman e pode ser instalado através do [site oficial](#).
- Se testarmos uma das nossas rotas podemos verificar que nos retorna o Body tal como o browser o recebe.



APIs: Benefícios e Usos

- Proporciona aos programadores comandos base para que não tenham que escrever código de raiz
- Fazem com que a nossa aplicação possa comunicar com outra aplicação sem que tenhamos que saber como é que a outra está implementada
- Simplifica o desenvolvimento de aplicação, poupando tempo e dinheiro
- Interagem com outros Websites, obtendo dados em tempo real

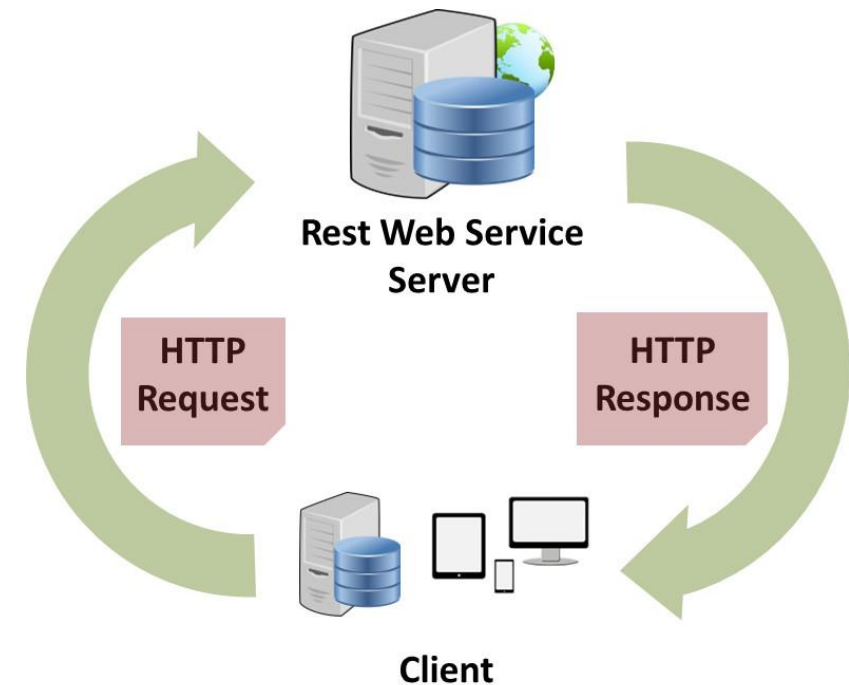


Web Services

Recurso que é tornado disponível através de uma rede, como por exemplo a internet.

Todos os Web Services são APIs mas nem todas as APIs são Web Services. Isto porque algumas APIs operam offline.

Um Web Service necessita de seguir um protocolo standard para transferir dados entre computadores. Normalmente, ele usa o HTTP.



API Endpoints



Um endpoint de API é normalmente o URL de um servidor ou serviço.

É o endereço de uma página ou recurso.

Iremos interagir com uma API fazendo um request e recebendo uma response.

Quando fazemos um request válido, a API responde dando-nos a localização do serviço ou recurso que queremos.

DEMO API

<http://open-notify.org/Open-Notify-API/ISS-Location-Now/>

[API Star Wars](#)

Pedidos HTTP

Existem 4 partes para um pedido HTTP:

1. Request Line
2. Headers
3. Blank Line
4. Body

Apenas a Request Line é obrigatória.

O propósito da Blank Line é separar os headers do body.

Pedidos HTTP

Request Line: linha inicial, onde incluimos o método HTTP a ser usado, o URI (Uniform Resource Indicator) do pedido e a versão do protocolo.

O URI é uma sequência de caracteres que indentificam o recurso.

Ex: GET /api/publishers HTTP/1.1.

Headers: lista de strings a ser enviada e recebida pelo cliente e Web Server.

Ex. Accept-Language: en-Us, fr, de

Body: dados associados ao o pedido ou resposta.

Content-Type e Content-Length headers especificam a natureza do Body.

Os Get normalmente não têm um body porque não precisamos de enviar informação, apenas os POST ou PUT, que marcam o texto a enviar.

Pedidos HTTP - Exemplo

POST /api/publishers HTTP/1.1

Host: abc.com

Content-Type: application/json

{

"Name": "Daniel Tait",

"Age": 32

}

← Request Line

} Headers

← Blank Line

} Message Body

HTTP Status Codes

[Documentação](#)

Status Code - Class Description	Translation
1xx: Informational response These codes let the client know that their request was received and understood by the server. But you need to wait for a final response as the request is still being processed.	<i>Hold on, we're working on it.</i>
2xx: Success This class of codes means that the request was received, understood, and accepted.	<i>Congratulations! Everything worked.</i>
3xx: Redirection These codes indicate that further action must be taken by the client in order to complete the request. Many of these codes are used in URL redirection.	<i>More work required.</i>
4xx: Client Error This class of codes means that the request contains incorrect syntax or cannot be fulfilled.	<i>You (the client) made a mistake.</i>
5xx: Server Error These codes means the server failed to fulfil an apparently valid request	<i>We (the server) made a mistake.</i>

APIs REST

- Representational State Transfer
- Uma API REST é uma Api que cumpre o estilo de arquitectura REST.
- Um estilo de arquitectura é uma série de princípios guia de design e restrições.
- Uma Api que obedece a estas regras é informalmente descrita como sendo RESTful.
- Nestas APIs os recursos são representados num formato de data como JSON.

Princípios REST

1. **Client-server Architecture:** o cliente e o servidor devem estar separados para que possam evoluir de forma independente. Por exemplo, deve-se poder fazer mudanças na aplicação mobile sem implicar mudanças na base de dados.
2. **Stateless:** o web server não deve reter nenhum estado da sessão do cliente no server-side. O servidor também não deve usar informações de pedidos anteriores do cliente. Desta forma é possível escalar as APIs para vários servidores: qualquer um pode gerir o pedido porque toda a informação está contida nele.
3. **Cache-ability:** Caching é a capacidade do cliente guardar cópias das respostas do servidor para dados acedidos frequentemente. Assim, o cliente não terá que fazer vários pedidos para o mesmo recurso.
A resposta de uma REST API deve ser indentificada como cacheable ou non-cacheable.

Princípios REST

4. Layered System: os pedidos e respostas vão por diferentes camadas. Devem ser desenhadas para que o cliente não perceba se estão a comunicar com o servidor final ou uma camada intermédia.

Os sistemas por camadas potenciam a segurança e a escalabilidade.

5. Uniform Interface: uma interface standard que não mude, de forma a simplificar a arquitectura.

6. Code on Demand (opcional): permite que pequenas aplicações como JAVA applets possam ser enviadas via a API para usar dentro da aplicação do cliente.

Permite a criação de aplicações que optimizem e sejam independentes da estrutura do cliente.

{JSON}

- JavaScript Object Notation
- É o formato de ficheiros para armazenar e transmitir dados mais usado actualmente. É conhecido por ser leve e fácil de ler / escrever.
- Apesar de derivar do JS o JSON não é só usado para JS, mas também em outras APIs que forneçam dados para Web como Python, PHP, etc..
- O Objecto de JSON não pode ser usado directamente em JS, temos primeiro que o converter para um objecto JS através do método `JSON.parse(objectoJSON)`.

[documentação](#)

```
"browsers": {  
  "firefox": {  
    "name": "Firefox",  
    "pref_url": "about:config",  
    "releases": {  
      "1": {  
        "release_date": "2004-11-09",  
        "status": "retired",  
        "engine": "Gecko",  
        "engine_version": "1.7"  
      }  
    }  
  }  
}
```

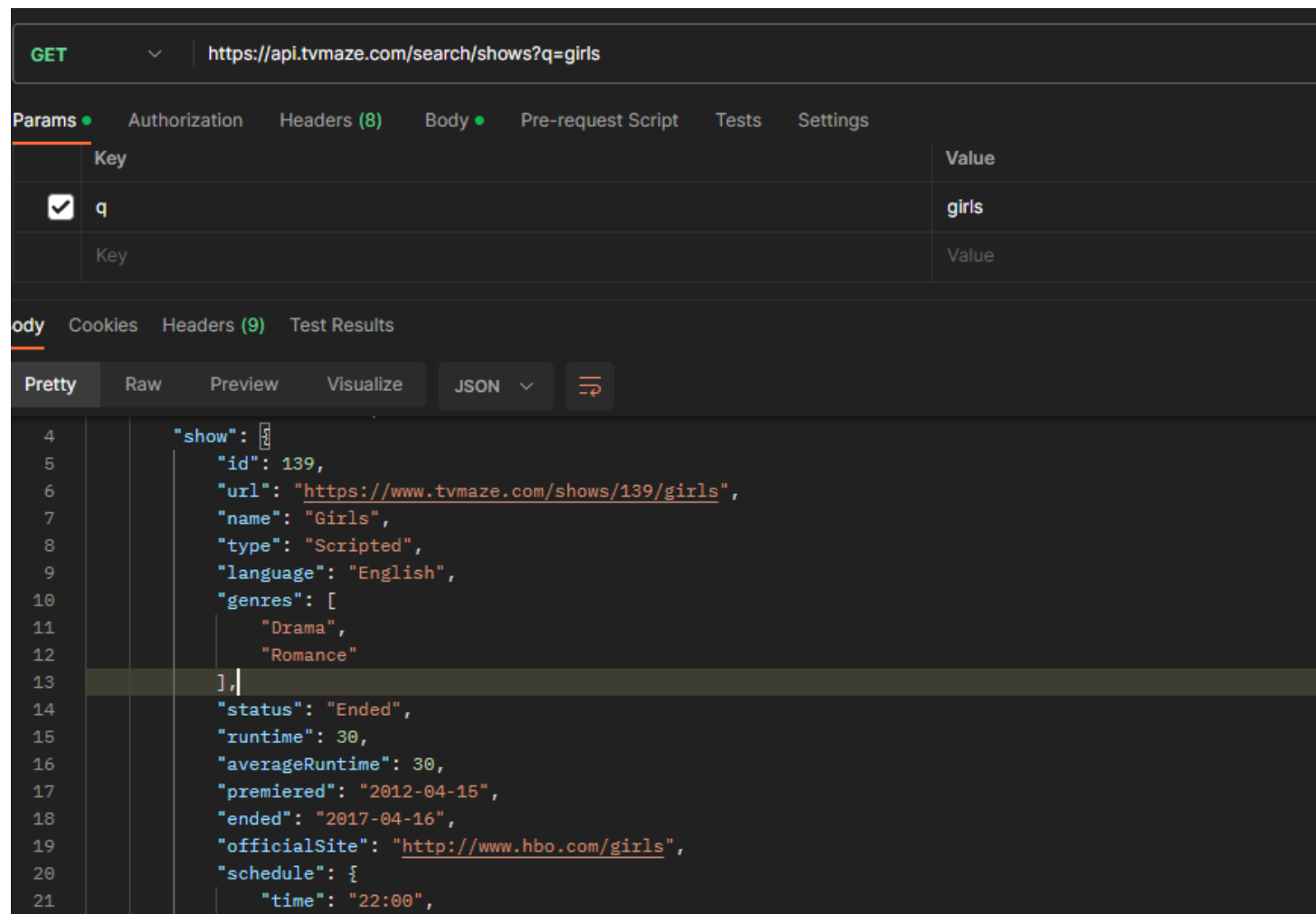

JSON Data Types

- Number 28 9.57 -30 4.5e6
- String "Bob" "Hello World!" "phone 911 now"
- Boolean true false
- Null null
- Array [1, 2, 5] ["Cat", "Dog"]
- Object {"name": "Ann", "age": 21, "cat": "Luna"}

Query Strings

- É um parâmetro de pesquisa que enviamos no url e que envia ao Backend a informação do que queremos procurar.
- Podemos ter vários parâmetros de pesquisa

[exemplo](#)



The screenshot shows a REST client interface with a GET request to `https://api.tvmaze.com/search/shows?q=girls`. The 'Params' tab is active, showing a single parameter `q` with the value `girls`. The 'Body' tab is also active, displaying the JSON response in 'Pretty' format. The response is a JSON array containing one object for the show 'Girls'.

```
4  {
5    "show": {
6      "id": 139,
7      "url": "https://www.tvmaze.com/shows/139/girls",
8      "name": "Girls",
9      "type": "Scripted",
10     "language": "English",
11     "genres": [
12       "Drama",
13       "Romance"
14     ],
15     "status": "Ended",
16     "runtime": 30,
17     "averageRuntime": 30,
18     "premiered": "2012-04-15",
19     "ended": "2017-04-16",
20     "officialSite": "http://www.hbo.com/girls",
21     "schedule": {
22       "time": "22:00",
```

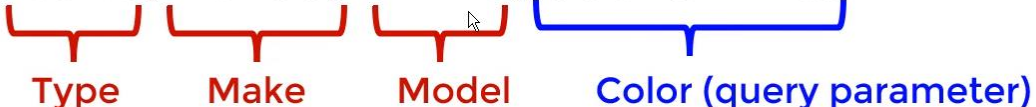
Path Parameters

- São partes variáveis de um caminho URL

`https://abc.com/publishers/george/articles?limit=50`

- Os path parameters são usados para identificar enquanto os query parameters são usados para filtrar os recursos.

GET `/cars/honda/civic/?color=red`



Type Make Model Color (query parameter)

XML



- eXtensible Markup Language
- Formato de ficheiro para guardar e transmitir dados
- Uma Markup Language é uma linguagem que usa tags para definir elementos dentro de um documento
- É também usada para enviar ou receber dados em APIs

XML

- Bastante semelhante a HTML
- As tags são usadas para descrever pedaços de dados

<dog_name>Lassie</dog_name>

↑ ↑ ↑

Start Tag Content End Tag

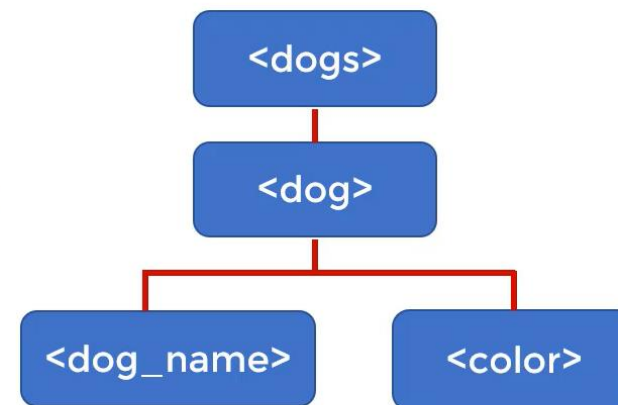
XML vs. HTML

- Ambos criados pelo W3C
- HTML diz ao browser como o documento deve parecer, enquanto o XML descreve o que contém
- Em HTML as tags estão definidas, enquanto em XML são extensíveis e podem ser nomeadas conforme queremos.
- As tags de XML são case-sensitive, ao contrário das de HTML

XML

- Os documentos de XML seguem uma estrutura árvore que começa nas raiz e se expande para as folhas.
- Cada documento deve conter apenas uma root, que é o pai de todos os outros elementos.

```
<dogs>  
  <dog>  
    <dog_name>Lassie</dog_name>  
    <color>Brown</color>  
  </dog>  
  <dog>  
    <dog_name>Oscar</dog_name>  
    <color>White</color>  
  </dog>  
</dogs>
```



Consultar uma API Externa

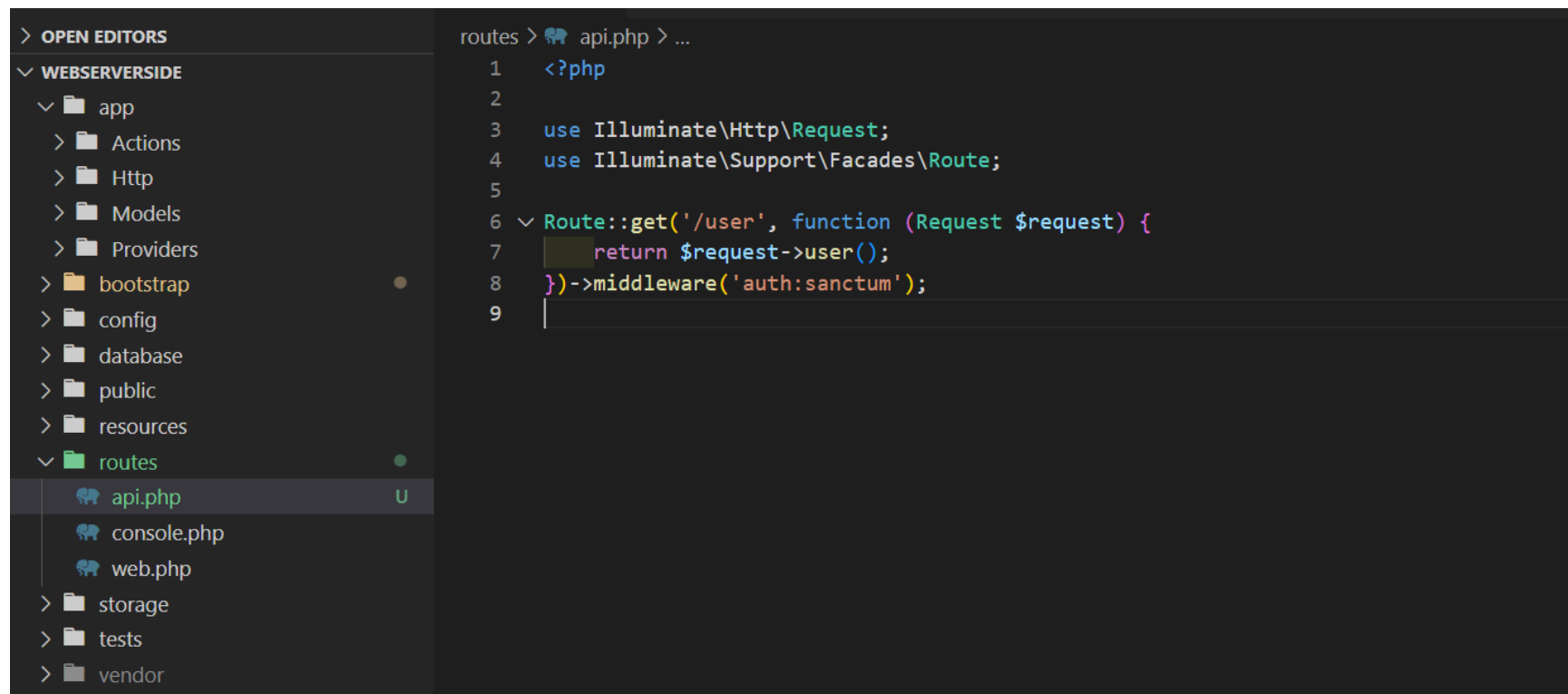
Com o Laravel podemos [consultar uma API](#) externa e usar os dados para mostrar na nossa aplicação. Por exemplo, usar a API do Star Wars para mostrar os filmes com o endpoint `/films/`

```
class StarWarsController extends Controller
{
    public function index(){
        $movies = Http::get('https://swapi.dev/api/films');
        $movies = json_decode($movies);

        return view('starwars.movies', compact('movies'));
    }
}
```


API em Laravel

Podemos também ser nós a criar a nossa própria API com os recursos do Laravel. Para isso corremos `php artisan install:api`. Irá ser criado um ficheiro de rotas específico para as rotas de API.



```
> OPEN EDITORS
WEBSERVERSIDE
  app
    Actions
    Http
    Models
    Providers
  bootstrap
  config
  database
  public
  resources
  routes
    api.php
    console.php
    web.php
  storage
  tests
  vendor

routes > api.php > ...
1  <?php
2
3  use Illuminate\Http\Request;
4  use Illuminate\Support\Facades\Route;
5
6  Route::get('/user', function (Request $request) {
7      return $request->user();
8  }->middleware('auth:sanctum');
9
```

Construir uma API – Show

```
api.php M X
routes > api.php
12 | Here is where you can register API routes for your application. T
13 | routes are loaded by the RouteServiceProvider and all of them wil
14 | be assigned to the "api" middleware group. Make something great!
15 |
16 | */
17 |
18 | Route::get('/task/{task}', [TaskAPIController::class, 'show']);
19 |
```

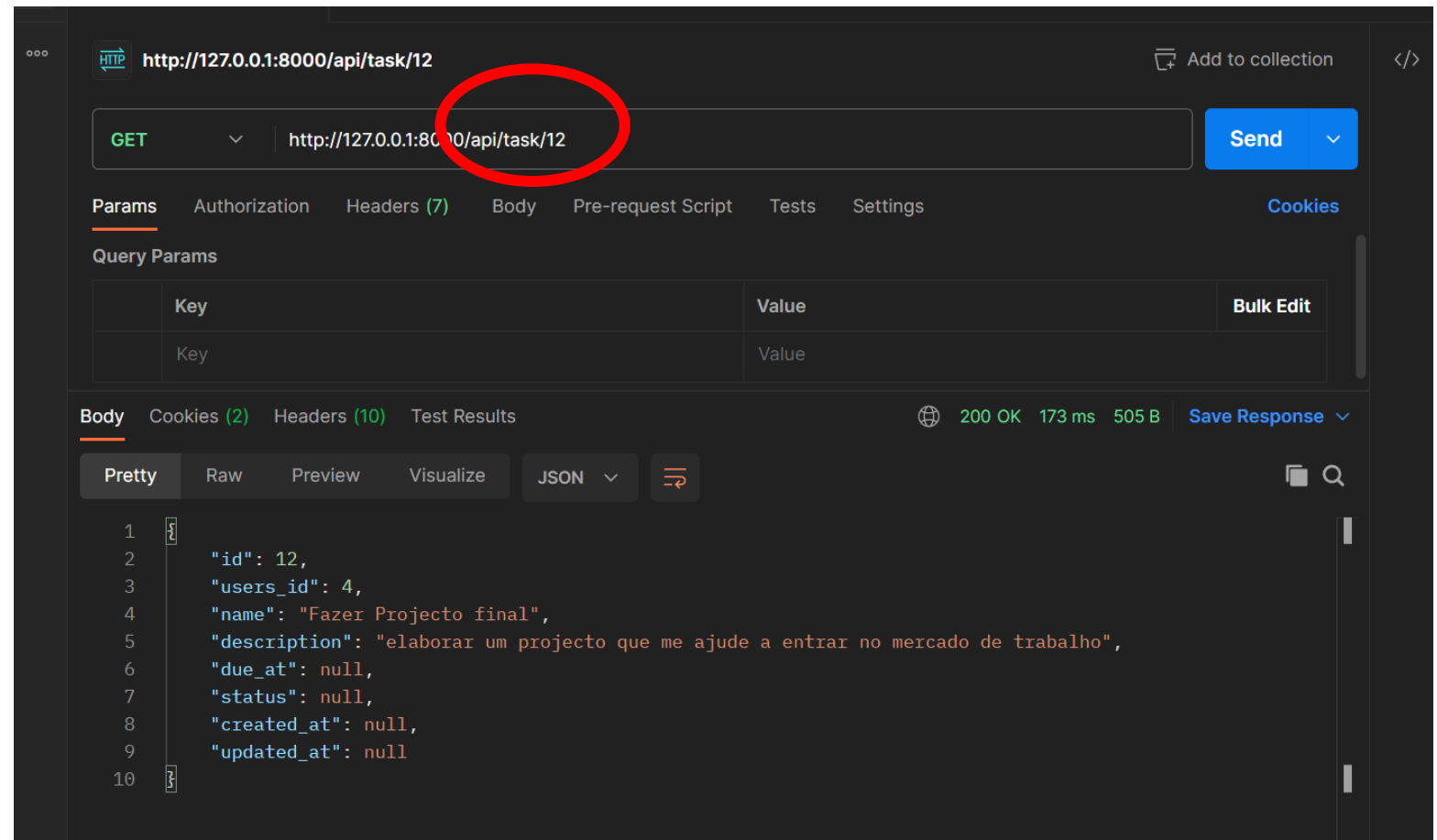
Iremos então construir uma API com a tabela de Tasks que já temos. Para tal, criaremos um novo controller com recursos chamado TaskAPIController e o modelo da tabela, que se chamará Task.

Em seguida, é seguir o processo de retornar uma task registrando a rota no ficheiro das API's.

```
/**
 * Display the specified resource.
 */
public function show(Task $task)
{
    return $task;
}
```

Construir uma API – Show

Uma vez que a nossa função show está associada ao Model, basta trocar o Id e ele irá retornar a tarefa em questão.



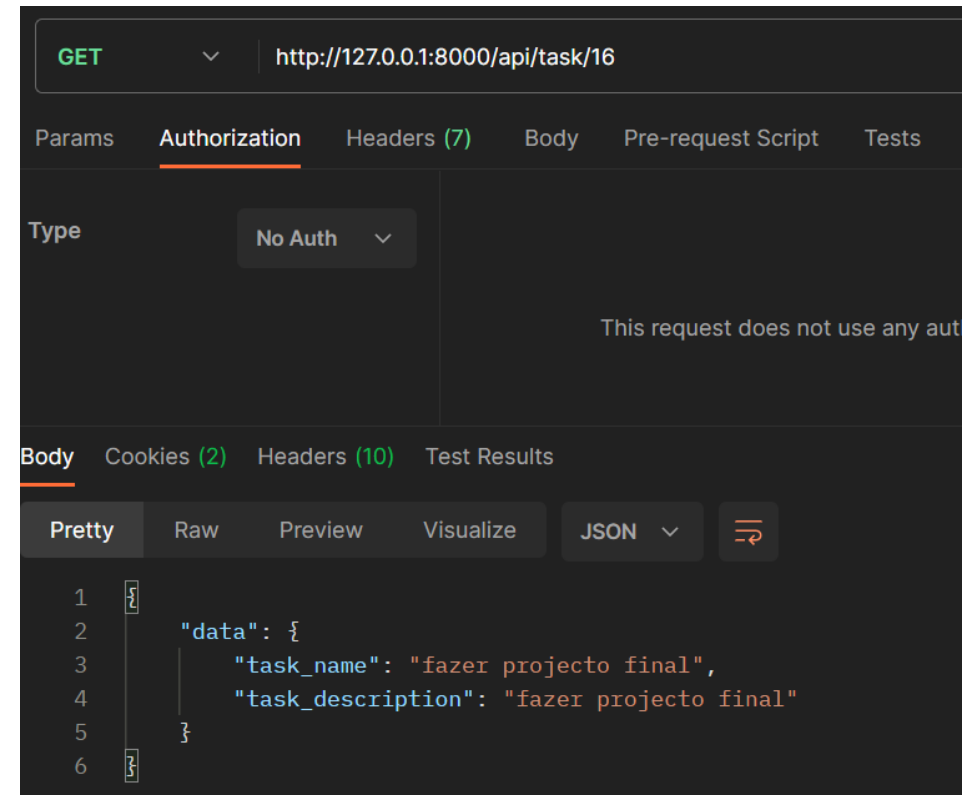
Construir uma API – Recurso Customizado

Podemos customizar a maneira como recebemos os dados da nossa BD e os enviamos na nossa Api. Para tal, iremos construir um recurso para as Tasks através de `php artisan make:resource GiftResource`.

Na função devemos adicionar o novo Recurso e ele passa a retornar o objecto JSON dentro de um array data apenas com os campos que queremos.

```
@return array<string, mixed>  
*/  
public function toArray(Request $request): array  
{  
    // return parent::toArray($request);  
  
    return [  
        'task_name' => $this->name,  
        'task_description' => $this->description,  
    ];  
}
```

```
*/  
public function show(Task $task): TaskResource  
{  
    return new TaskResource($task);  
}
```



Construir uma API – Recurso API em Laravel

O Laravel dispõe de um [recurso de API's chamado apiResource](#) que nos permite com um único registo nas rotas criar todas as rotas para os métodos da API.

```
Route::apiResource('/task', TaskAPIController::class);
```

Correndo a lista das rotas vemos que todas foram adicionadas à função correspondente.

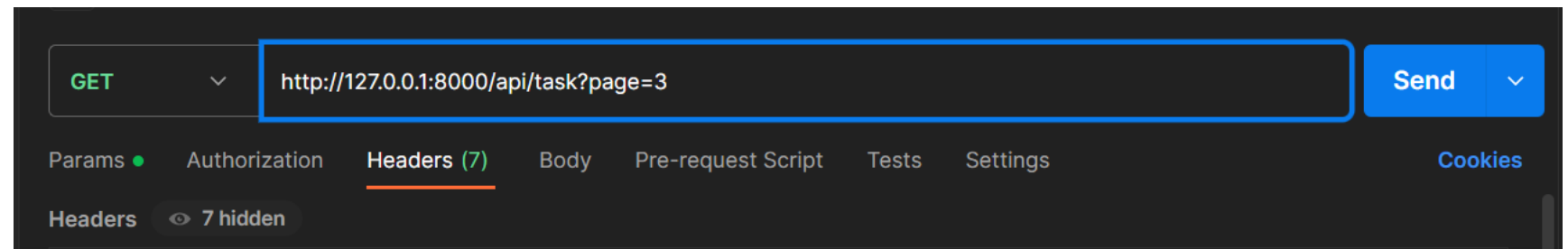
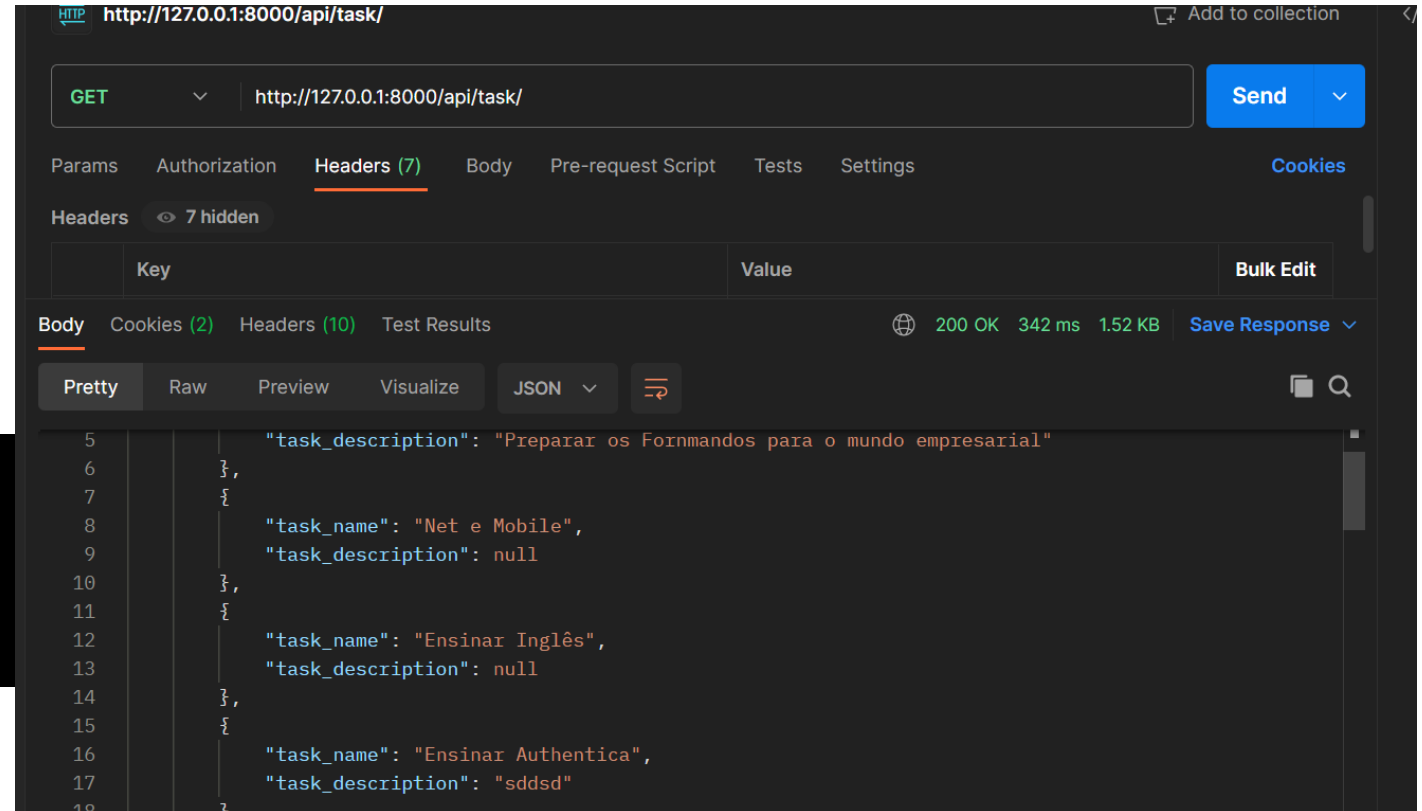
```
POST      _ignition/update-config .. Ignition.updateConfig > Spatie\LaravelIgnition > UpdateConfigController
GET|HEAD  api/task/{task} ..... {task}.index > TaskAPIController@index
POST      api/task/{task} ..... {task}.store > TaskAPIController@store
GET|HEAD  api/task/{task}/{task} ..... {task}.show > TaskAPIController@show
PUT|PATCH api/task/{task}/{task} ..... {task}.update > TaskAPIController@update
DELETE    api/task/{task}/{task} ..... {task}.destroy > TaskAPIController@destroy
POST      create task ..... create task > UserController@storeTask
```

Construir uma API – Index

Para retornar todas as tasks iremos criar uma coleção através de php artisan `make:resource TaskResourceCollection` –collection.

```
*/  
public function index(): TaskResourceCollection  
{  
    return new TaskResourceCollection(resource: Task::paginate());  
}  
  
/**  
 * Show the form for creating a new resource  
 */
```

O método das coleções `paginate()` recebe uma linha da BD por página.



Construir uma API – Store

```
5 use Illuminate\Database\Eloquent\Model;
6
7
8 class Task extends Model
9 {
10     use HasFactory;
11
12     protected $fillable = [
13         'name',
14         'description',
15         'users_id',
16     ];
17 }
18
```

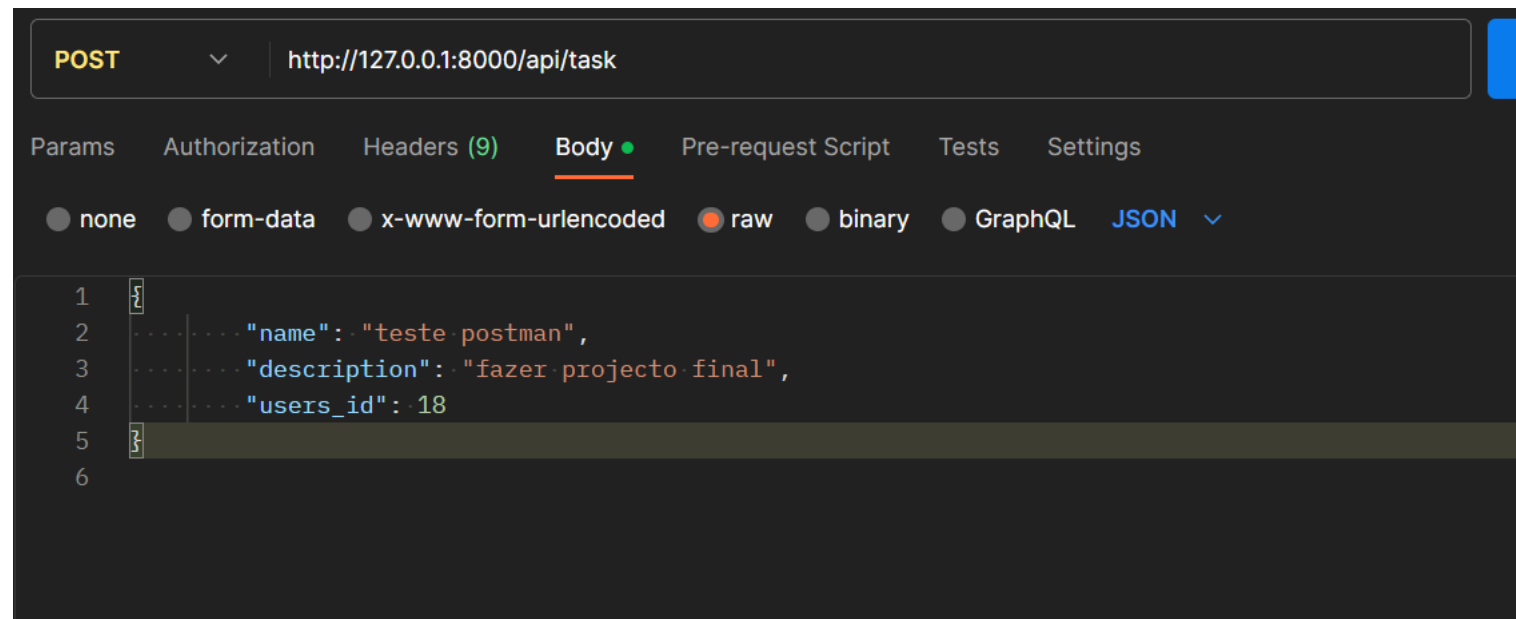
Em seguida editamos o store para criar uma nova task com a validação e criação de um novo recurso Task.

A primeira coisa a fazer é no Model Task adicionar os campos que têm que ser preenchidos.

```
1 /*
2 public function store(Request $request)
3 {
4
5     $request->validate([
6         'name' => 'required',
7         'description' => 'required',
8         'users_id' => 'required',
9     ]);
10
11     $task = Task::create($request->all());
12
13     return new TaskResource($task);
14 }
15 */
```

Construir uma API – Store

Por fim, no postman enviamos em JSON o recurso que queremos criar.
Não esquecer que o método é o POST.

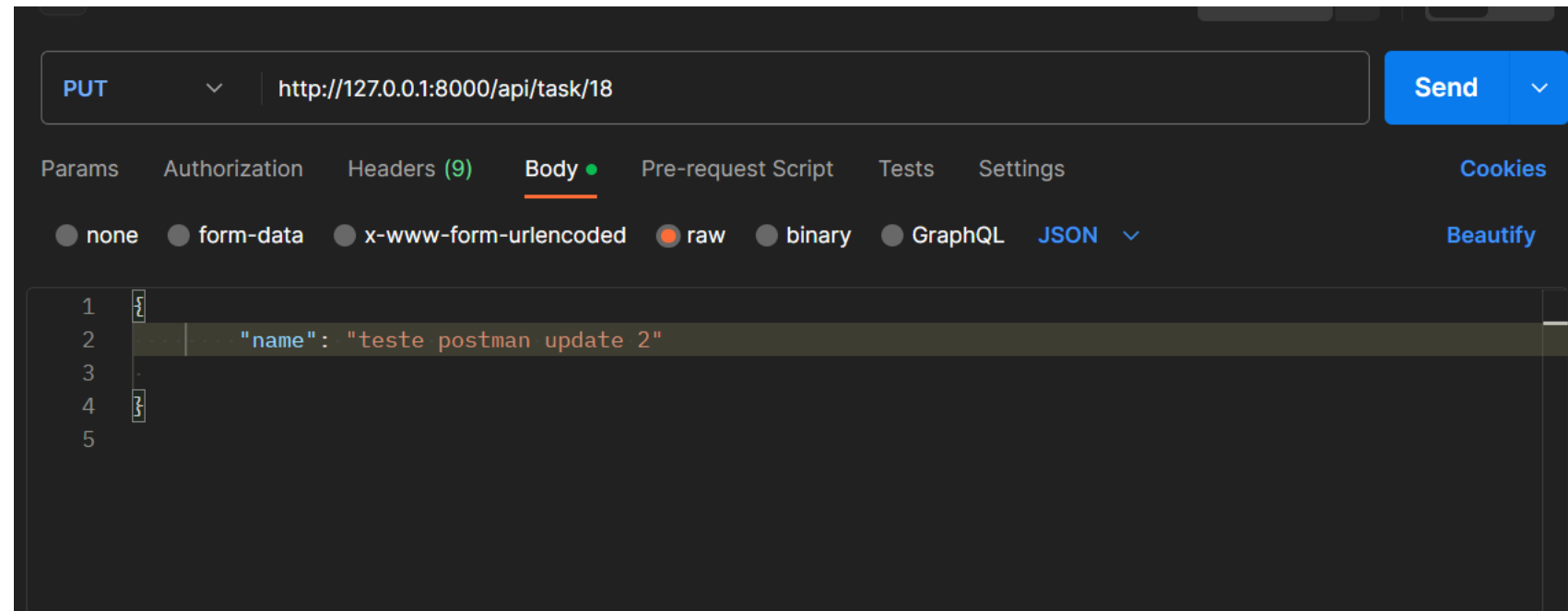


Construir uma API – Update

```
* Update the specified resource in storage.
*/
public function update(Request $request, Task $task): TaskResource
{
    $task = $task->update($request->all());

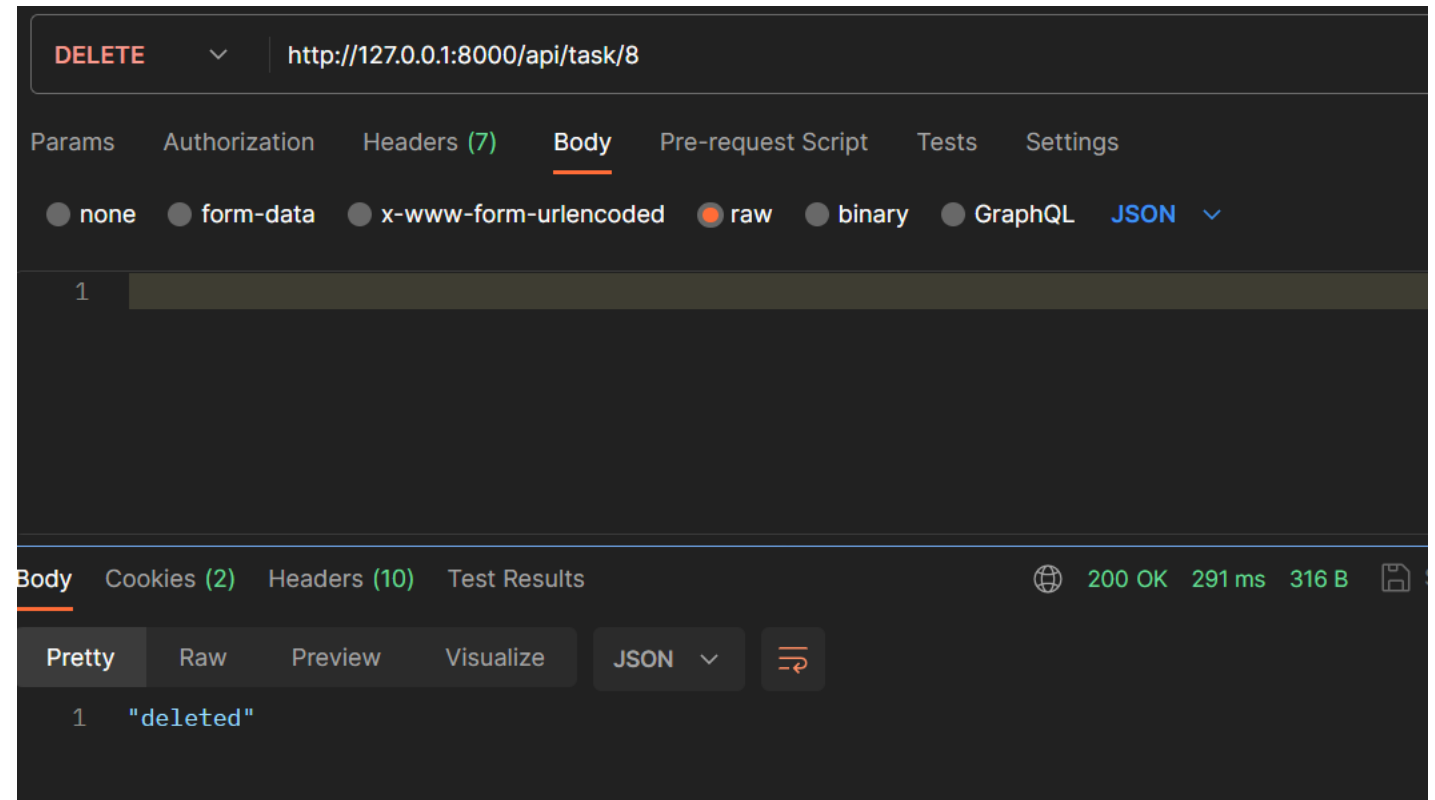
    return new TaskResource($task);
}

/**
 * Remove the specified resource from storage
 */
```



Construir uma API – Delete

```
*/  
public function destroy(Task $task)  
{  
    $task = $task->delete();  
    return response()->json('deleted');  
}
```



Recursos

- [Documentação Laravel](#)
- [Laracasts](#)