Upload de arquivos com Laravel e VueJS

Em algum momento do desenvolvimento de um sistema, todo desenvolvedor irá se deparar com uploads de arquivos. Naturalmente, para aqueles menos experientes, isto pode ser um pouco complicado. E quando se quer então deixar o sistema mais simples e prático para o usuário final, esta tarefa pode dar algumas dores de cabeça. Este tutorial visa ser um passo a passo inicial para facilitar o desenvolvimento deste processo.

Pré-requisitos

Para este tutorial estarei assumindo que você já possui um projeto Laravel e tenha conhecimentos básicos em PHP, VueJS, e no próprio Laravel.

Estarei assumindo também que estará utilizando um navegador moderno com suporte à objetos do tipo FormData.

Preparando o Backend

Primeiramente, nosso sistema deverá guardar nossos arquivos em algum diretório. O laravel possui um poderoso sistema de arquivos que é muito simples de configurar. Com ele podemos utilizar nosso diretório local para armazenar nossos arquivos, ou utilizar servidores na núvem, como o Amazon S3, por exemplo. Para este tutorial, iremos utilizar o driver de diretório local.

Em seu projeto Laravel, localize e abra o arquivo ***config/filesystems.php***. Localize o índice "*disks*". Nele você irá encontrar vários discos pré-configurados para serem utilizados. Vamos então criar o nosso novo disco, que irá armazenar nossos novos arquivos. Abaixo do disco local, digite o seguinte código:

```
'uploads' => [
    'driver' => 'local',
    'root' => storage_path().'/files/uploads',
],
```

Aqui estaremos chamando nosso novo disco de "*uploads*". Ele irá utilizar o driver "*local*", ou seja, os arquivos serão armazenados no seu computador, ou no próprio servidor. E por último, estará localizado em ***storage/files/uploads***, em seu projeto Laravel.

Agora que temos nosso espaço para armazenar os arquivos, queremos também salvar sua referência no banco de dados, para que possamos acessá-los mais facilmente no sistema.

No terminal, digite o seguinte comando:

> *php artisan make:migration create_file_entries_table*

Este comando irá gerar seu arquivo de migração no diretório ***database/migrations*** para criar a respectiva tabela no banco de dados. Adicionando os campos desejados, o arquivo de migração deverá ficar da seguinte forma:

```
class CreateFileEntriesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('file_entries', function (Blueprint
$table) {
            $table->increments('id');
            $table->string('filename');
            $table->string('mime');
            $table->string('path');
            $table->integer('size');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('file_entries');
    }
}
```

Aqui chamei nossa tabela de file_entries, mas você poderá atribuir o

nome que quiser. Nela, além do *id* e do *timestamps* padrão (*created_at* e *updated_at*), temos os seguintes campos:

filename: Nome original do arquivo;

mime: Mimetype do arquivo;

path: Diretório interno que será guardado o arquivo (explicarei o porquê a seguir)

size: Tamanho do arquivo.

Agora, precisamos do nosso Model e também do nosso Controller para gerenciar nossos arquivos enviados. No terminal digite os seguintes comandos:

> *php artisan make:model FileEntry*

> *php artisan make:controller FileEntriesController*

No model, deveremos adicionar os campos que serão preenchidos na nossa tabela do banco de dados sempre que fizermos um upload. Ele deverá ficar assim:

```
class FileEntry extends Model
{
    protected $fillable = ['filename', 'mime', 'path',
'size'];
}
```

No nosso controller, deveremos criar o método quer irá receber o post do nosso formulário de upload de arquivos, e tratá-lo, salvando no local correto e também no banco de dados. Abra o controller ***FileEntriesController*** e crie o método ***uploadFile*** da seguinte forma:

```
public function uploadFile(Request $request) {
        $file = Input::file('file');
        $filename = $file->getClientOriginalName();

        $path = hash( 'sha256', time());
```

```
if(Storage::disk('uploads')->put($path.'/'.$filename,
File::get($file))) {
            $input['filename'] = $filename;
            $input['mime'] = $file->getClientMimeType();
            $input['path'] = $path;
            $input['size'] = $file->getClientSize();
            $file = FileEntry::create($input);

            return response()->json([
                'success' => true,
                'id' => $file->id
            ], 200);
        }
        return response()->json([
            'success' => false
        ], 500);
    }
```

Primeiramente, iremos receber os dados do arquivo enviado pelo formulário em ***$file = Input::file('file'),*** onde '***file***' deverá ser o nome do campo no formulário. A seguir, pegamos o nome original do arquivo em ***$filename = $file->getClientOriginalName()***. Depois iremos criar o path que irá armazenar o arquivo dentro do nosso disco: ***$path = hash('sha256', time())***. Eu opto por criar um path dessa forma, com um hash aleatório baseado na hora e data atuais, para que os arquivos fiquem mais seguros do acesso direto, uma vez que o endereço dos arquivos ficará, por exemplo, da seguinte forma: *storage/files/uploads /3e8f4a2e6d26c205e52ebcf6518e84bba96ccc9499f01c24448e939cbcb9 f8d4/filename.jpg.*

A seguir, iremos selecionar nosso disco recém criado (*uploads*), pegar o arquivo enviado (*File::get($file)*), e salvá-lo no local correto (*$path.'/'.$filename*) com o método put: ***Storage::disk('uploads')->put($path.'/'.$filename, File::get($file))***.

Caso haja sucesso, salvaremos os dados na nossa tabela através do método *create*: ***$file = FileEntry::create($input)***, e retornar uma resposta JSON confirmando o sucesso juntamente do id do arquivo recém criado. Caso o upload falhe, Será retornado um JSON informando que houve uma falha.

Agora o Frontend

Para agilizar o desenvolvimento do projeto, e focar no upload de arquivos, irei utilizar o CSS padrão que vem com o Laravel. Primeiro,

para tornar nosso sistema seguro e importar o layout padrão, execute o seguinte comando no terminal:

> *php artisan make:auth*

Ele irá criar as páginas de login, registro e recuperação de senha.

Agora, crie o diretório ***resources/views/files*** e dentro dele o arquivo ***index.blade.php*** com o seguinte código:

```
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">Files <a href="{{
url('files/create') }}" class="btn btn-info">Add files</a>
</div>

                <div class="card-body">
                    @if($files->count())
                        <table class="table">
                            <th>Name</th>
                            <th>Size</th>
                            @foreach($files as $file)
                                <tr>
                                    <td>{{ $file->filename
}}</td>
                                    <td>{{ $file->size }}
Bytes</td>
                                </tr>
                            @endforeach
                        </table>
                    @else
                        You have no files yet!
                    @endif
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

Basicamente nosso index irá receber todos os arquivos salvos no banco de dados e listá-los em uma tabela HTML.

No controller ***app/Http/Controllers/FileEntriesController.php*** crie o método index da seguinte forma:

```
public function index() {
    $files = FileEntry::all();

    return view('files.index', compact('files'));
}
```

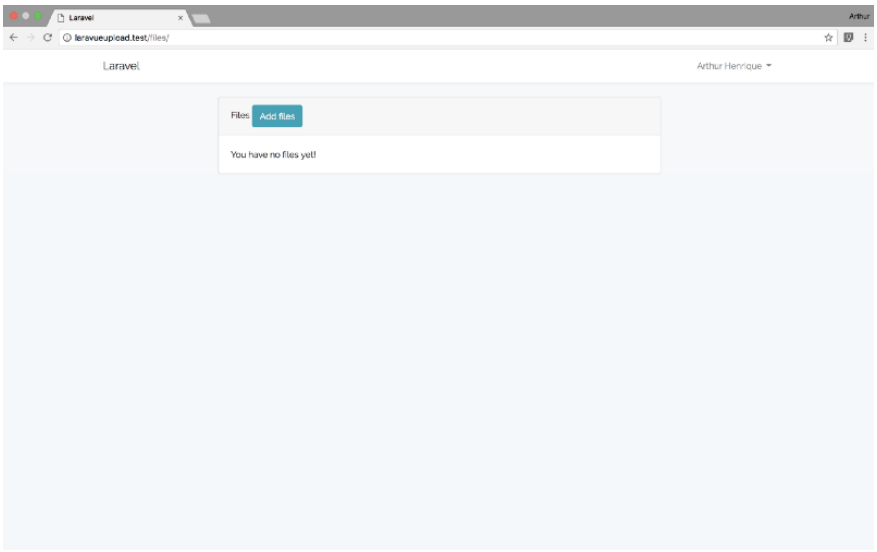
Aqui estamos pegando todos os arquivos salvos no banco de dados e gravando na variável files (***$files = FileEntry::all()***), e depois retornando-a para a nossa view (***return view('files.index', compact('files'))***).

Por fim, deveremos registrar a nossa rota para que seja acessível através da URL. No arquivo ***routes/web.php*** adicione o seguinte trecho de código:

```
Route::group(['middleware' => 'auth'], function () {
    Route::get('files', 'FileEntriesController@index');
});
```

O trecho ***Route::group(['middleware' => 'auth'], function () {})*** irá proteger nossa nova view, permitindo acesso apenas de usuários autenticados. Dentro dele, adicionamos o nosso *"get",* ***Route::get('files', 'FileEntriesController@index')***, onde o primeiro parâmetro é o nome da rota, e o segundo é o método no controller que irá referenciar esta rota.

Acessando a url no navegador, deverá aparecer a seguinte tela:



Tela index dos arquivos

Trabalhando com VueJS

Agora chegou a hora de fazermos nosso componente vue que irá nos auxiliar no nosso upload de arquivos, além de deixá-lo prático, bonito e simples para nosso usuário final.

Antes de iniciar com o nosso desenvolvimento em VueJS, deveremos instalar as dependências necessárias. Para isso, basta entrarmos na raiz do nosso projeto e digitar o seguinte comando no terminal:

> *npm install*

Aguarde um tempo enquanto ele baixa e instala todas as dependências no projeto. Obs: ao realizar o deploy para a web, deverá executar o comando novamente no servidor.

Agora, crie o arquivo ***resources/assets/js/components /UploadFiles.vue***.

Componentes Vue possuem a seguinte estrutura básica:

```
<template>

</template>

<script>
    export default {
```

```
    }
</script>

<style scoped>

</style>
```

As tags ***<template></template>*** delimitam o código HTML que irá ser apresentada na nossa view. As tags ***<script></script>*** delimitam onde irá a programação. As tags ***<style></style>*** delimitam onde irá o CSS personalizado para o nosso componente. Esta última é opcional.

Template

No nosso arquivo, dentro das tags ***<template></template>***, adicione o seguinte código:

```
<template>
    <div class="container">
        <div class="large-12 medium-12 small-12 filezone">
            <input type="file" id="files" ref="files"
multiple v-on:change="handleFiles()"/>
            <p>
                Drop your files here <br>or click to search
            </p>
        </div>

        <div v-for="(file, key) in files" class="file-
listing">
            <img class="preview"
v-bind:ref="'preview'+parseInt(key)"/>
            {{ file.name }}
            <div class="success-container" v-if="file.id >
0">
                Success
            </div>
            <div class="remove-container" v-else>
                <a class="remove"
v-on:click="removeFile(key)">Remove</a>
            </div>
        </div>

        <a class="submit-button" v-on:click="submitFiles()"
v-show="files.length > 0">Submit</a>
    </div>
</template>
```

Agora temos alguns pontos a destacar:

Input

Primeiro de tudo, no ***<input>*** temos o atributo "***ref***", que nomeia o mesmo, para que consigamos acessá-lo e trabalhar no VueJS. A seguir, temos o método "***handleFiles***" que é executado no ***v-on:change***, ou seja, sempre que há mudanças no campo.

Preview e Remoção

A seguir temos um ***laço for*** em uma * ,* que será mostrada para cada arquivo mapeado pela variável *files.* Dentro da *div* temos a imagem preview, que é mostrada de acordo com o índice do arquivo no array, seguido do nome do arquivo, e duas divs: Success e Remove. Estas duas divs são ligadas por v-if e v-else, onde caso o arquivo ainda não tenha sido submetido, a opção de remover o arquivo estará visível. Caso o arquivo seja submetido, o arquivo não poderá mais ser removido, e apenas a informação "Success" será mostrada, indicando que o arquivo foi enviado corretamente.

Submit

Por fim, criamos o botão de submit com o método "***submitFiles***" que é executado sempre que for clicado (***v-on:click***). Esse botão é mostrado apenas enquanto há arquivos para enviar, conforme o ***v-show*** do Vue.

Script

Primeiro de tudo, vamos instanciar as variáveis que iremos trabalhar:

```
data() {
    return {
        files: []
    }
}
```

files é um array que irá armazenar os arquivos enviados. Ele começa vazio.

A seguir temos os nossos métodos, que deverão estar dentro de:

```
methods: {
```

```
    // your methods here
}
```

O primeiro deles é o ***handleFiles()***:

```
handleFiles() {
    let uploadedFiles = this.$refs.files.files;

    for(var i = 0; i < uploadedFiles.length; i++) {
        this.files.push(uploadedFiles[i]);
    }
    this.getImagePreviews();
},
```

Ele é o responsável por receber todos os arquivos adicionados ao input e organizá-los no array de arquivos. Ao final, ele irá chamar o método seguinte, ***getImagePreviews***, que irá mostrar o preview dos arquivos.

```
getImagePreviews(){
    for( let i = 0; i < this.files.length; i++ ){
        if ( /\.(jpe?g|png|gif)$/i.test( this.files[i].name
) ) {
            let reader = new FileReader();
            reader.addEventListener("load", function(){
                this.$refs['preview'+parseInt( i )][0].src =
reader.result;
            }.bind(this), false);
            reader.readAsDataURL( this.files[i] );
        }else{
            this.$nextTick(function(){
                this.$refs['preview'+parseInt( i )][0].src =
'/img/generic.png';
            });
        }
    }
},
```

Basicamente, ele percorre todo o array de arquivos e primeiro verifica se o mesmo é uma imagem. Caso positivo, ele gera um preview daquele arquivo e adiciona para ser apresentada na div do respectivo arquivo. Caso negativo, ele apresenta uma imagem padrão como preview. A imagem que estarei utilizando está disponibilizada abaixo. Salve-a em ***public/img/generic.png***.

Imagem para arquivos genéricos

A seguir temos o método de remover arquivos:

```
removeFile( key ){
    this.files.splice( key, 1 );
    this.getImagePreviews();
},
```

Ele apenas retira do array de arquivos o respectivo arquivo escolhido e atualiza os previews das imagens.

Por fim, temos o método de submit:

```
submitFiles() {
    for( let i = 0; i < this.files.length; i++ ){
        if(this.files[i].id) {
            continue;
        }
        let formData = new FormData();
        formData.append('file', this.files[i]);

        axios.post('/' + this.post_url,
            formData,
            {
                headers: {
                    'Content-Type': 'multipart/form-data'
                }
            }
        ).then(function(data) {
            this.files[i].id = data['data']['id'];
            this.files.splice(i, 1, this.files[i]);
            console.log('success');
        }.bind(this)).catch(function(data) {
            console.log('error');
        });
    }
}
```

Ele percorre o array de arquivos, verifica se o arquivo já não foi enviado (se possuir um id, já foi enviado) e os envia um por um. Caso haja sucesso, ele atualiza array ***files*** para que seja recarregado no template. Vamos detalhar mais um pouco o que acontece aqui:

Primeiramente nós criamos o formulário com ***let formData = new FormData()***. A seguir adicionamos o arquivo ao formulário para ser enviado: ***formData.append('file', this.files[i])***.

Agora nós enviamos o formulário através de uma requisição utilizando ***axios***. Para isso, utilizaremos o método "***post***". Nele temos três parâmetros. O primeiro é a url de destino, que no nosso caso é "***/files/upload-file***". O segundo parâmetro é o que nós desejamos enviar, que nesse caso é o formulário ***formData*** que criamos anteriormente. No terceiro parâmetro adicionamos o cabeçalho, que contém ***'Content-Type': 'multipart/form-data'*** para que seja possível o envio de arquivos através do formulário.

Por fim, temos os métodos de callback para sucesso e erro. Perceba que no sucesso (***then***) estaremos pegando o id do arquivo recém enviado e atualizando o array ***files*** com a nova informação.

Style

Não irei me aprofundar no CSS para o artigo não ficar muito grande, mas basicamente ele esconde o input dos arquivos e no lugar dele, renderiza a caixa tracejada onde os arquivos poderão ser arrastados.

```
<style scoped>
    input[type="file"]{
        opacity: 0;
        width: 100%;
        height: 200px;
        position: absolute;
        cursor: pointer;
    }
    .filezone {
        outline: 2px dashed grey;
        outline-offset: -10px;
        background: #ccc;
        color: dimgray;
        padding: 10px 10px;
        min-height: 200px;
        position: relative;
        cursor: pointer;
    }
    .filezone:hover {
```

```
            background: #c0c0c0;
        }

        .filezone p {
            font-size: 1.2em;
            text-align: center;
            padding: 50px 50px 50px 50px;
        }
        div.file-listing img{
            max-width: 90%;
        }

        div.file-listing{
            margin: auto;
            padding: 10px;
            border-bottom: 1px solid #ddd;
        }

        div.file-listing img{
            height: 100px;
        }
        div.success-container{
            text-align: center;
            color: green;
        }

        div.remove-container{
            text-align: center;
        }

        div.remove-container a{
            color: red;
            cursor: pointer;
        }

        a.submit-button{
            display: block;
            margin: auto;
            text-align: center;
            width: 200px;
            padding: 10px;
            text-transform: uppercase;
            background-color: #CCC;
            color: white;
            font-weight: bold;
            margin-top: 20px;
        }
    </style>
```

Registrando o componente e compilando

Agora que concluímos o nosso componente Vue, devemos registrar o nome que iremos utilizar na nossa view para chamá-lo. Para isso abra o arquivo ***resources/assets/js/app.js*** e adicione o seguinte trecho após o componente de exemplo:

```
Vue.component('upload-files', require('./components
/UploadFiles.vue'));
```

Você poderá atribuir o nome que desejar. Nesse tutorial, optei por escolher "*upload-files*".

Terminada a codificação, devemos agora compilar nosso projeto Vue. Na raiz de seu projeto Laravel, execute o seguinte comando:

> *npm run watch*

Poderíamos utilizar o *run dev* ou o *run build*, mas enquanto ainda estamos desenvolvendo, prefiro utilizar o *watch*, pois a cada mudança feita no código, o mesmo será recompilado sem a necessidade de executar novamente.

Submetendo arquivos

Agora que temos nosso componente desenvolvido, podemos agora criar a view de upload de arquivos no Laravel. Para isso, crie o arquivo ***resources/views/files/create.blade.php*** e adicione o seguinte código:

```
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">Add files</div>

                <div class="card-body">
                    <upload-files></upload-files>
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

Perceba que adicionamos o nosso novo componente ***<upload-files></upload-files>***.

Precisamos agora apenas criar o respectivo método no controller e sua

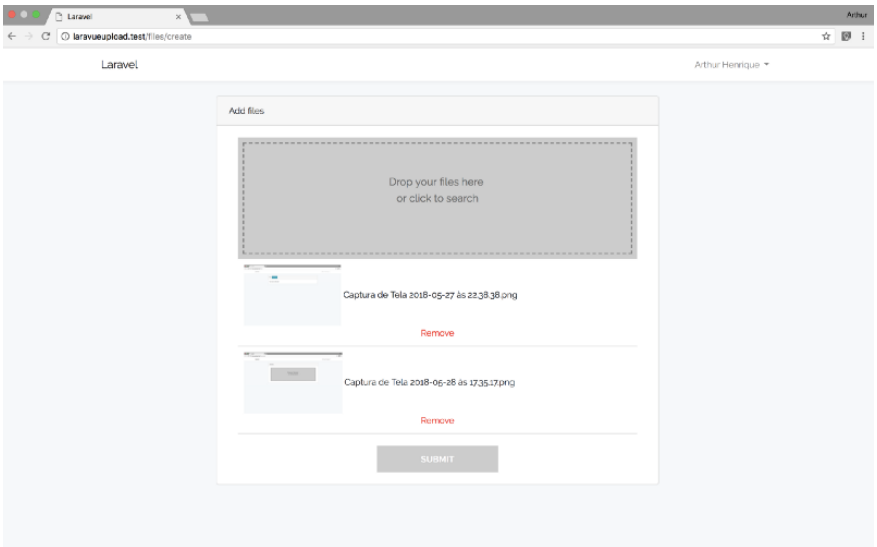
rota. No ***FileEntriesController*** adicione o seguinte método:

```
public function create() {
    return view('files.create');
}
```

E no ***routes/web.php*** adicione a rota para esta view, e também a rota que receberá a requisição POST do ***Axios***:

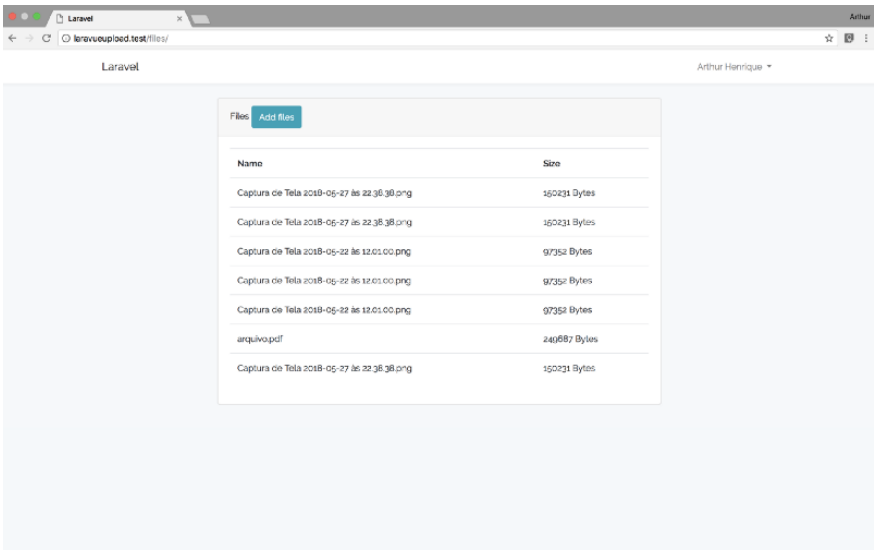
```
Route::get('files/create', 'FileEntriesController@create');
Route::post('files/upload-file',
'FileEntriesController@uploadFile');
```

Testando nosso novo recurso, ele ficará assim:



Tela de upload de arquivos com dois arquivos selecionados para submissão

Nosso script agora está submetendo os arquivos e salvando no banco de dados. Se voltarmos para a view index, a listagem dos arquivos será apresentada:



Listagem dos arquivos enviados

Caso queira que ao clicar no nome do arquivo, o arquivo seja enviado para download, basta adicionar a seguinte rota:

```
Route::get('files/{path_file}/{file}', function($path_file =
null, $file = null) {
    $path = storage_path().'/files/uploads/'.$path_file.'
/'.$file;
    if(file_exists($path)) {
        return Response::download($path);
    }
});
```

E no nome do arquivo, o seguinte link:

```
<a href="{{ url('files/'.$file->path.'/'.$file->filename)
}}">{{ $file->filename }}</a>
```

Bônus: Adicionando arquivos em outros Models

Nosso componente está funcionando perfeitamente e salvando os arquivos na respectiva tabela de arquivos, mas, geralmente, desejamos associar os arquivos enviados à algum outro model como por exemplo documentos pessoais do usuário. Adicionando uma pequena mudança

em nosso código, podemos deixar o componente recém criado versátil e reaproveitado em todo o projeto.

Para isso, no nosso componente Vue, deveremos criar novos parâmetros que serão enviados através da chamada do componente. Eles são necessários pois para cada model, o endereço em que os arquivos serão enviados pelo Axios poderá mudar, e também o "*name*" do input poderá mudar. Abra o ***resources/assets/js/components/UploadFiles.vue*** e realize a seguinte modificação em negrito no template:

```
<template>
    <div class="container">
        <div class="large-12 medium-12 small-12 filezone">
            <input type="file" id="files" ref="files"
multiple v-on:change="handleFiles()"/>
            <p>
                Drop your files here <br>or click to search
            </p>
        </div>

        <div v-for="(file, key) in files" class="file-
listing">
            <img class="preview"
v-bind:ref="'preview'+parseInt(key)"/>
            {{ file.name }}
            <div class="success-container" v-if="file.id >
0">
                Success
                <input type="hidden" :name="input_name"
:value="file.id"/>
            </div>
            <div class="remove-container" v-else>
                <a class="remove"
v-on:click="removeFile(key)">Remove</a>
            </div>
        </div>

        <a class="submit-button" v-on:click="submitFiles()"
v-show="files.length > 0">Submit</a>
    </div>
</template>
```

Percebam que adicionamos um input que só será apresentado se o arquivo possuir um id, ou seja, ele foi enviado corretamente. Esse input tem o nome de "***input_name***" e este nome será enviado através das ***props*** do VueJS. Como a URL de envio de arquivo poderá mudar também, dependendo do model, iremos criar uma props chamada "*post_url*" para recebê-la. Para isso, adicione o seguinte código em negrito no início do script:

```
<script>
    export default {
        props: ['input_name', 'post_url'],
        data() {
            return {
                files: []
            }
        },
```

Agora, iremos adicionar o prop da URL na nossa chamada Axios. O id do arquivo recém enviado já era salvo anteriormente, então isso não sofre mudanças. O método ***submitFiles*** deverá ficar da seguinte forma:

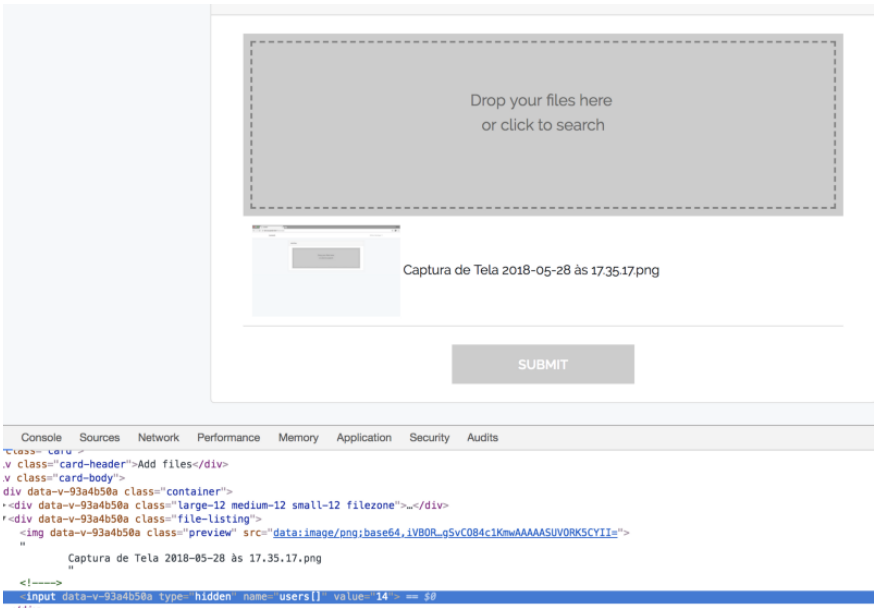
```
submitFiles() {
    for( let i = 0; i < this.files.length; i++ ){
        if(this.files[i].id) {
            continue;
        }
        let formData = new FormData();
        formData.append('file', this.files[i]);

        axios.post('/' + this.post_url,
            formData,
            {
                headers: {
                    'Content-Type': 'multipart/form-data'
                }
            }
        ).then(function(data) {
            this.files[i].id = data['data']['id'];
            this.files.splice(i, 1, this.files[i]);
            console.log('success');
        }.bind(this)).catch(function(data) {
            console.log('error');
        });
    }
}
```

Agora que adicionamos estas duas props no nosso componente Vue, sempre que utilizarmos o componente criado, deveremos passá-las como parâmetro da seguinte forma:

```
<upload-files :input_name="'users[]'"
:post_url="'files/upload-file'"></upload-files>
```

Onde, nesse exemplo, nosso input personalizado será um ***users[]*** e nossa URL de submissão será ***files/upload-file***. Ele agora está pronto para ser usado em qualquer formulário, bastando informar qual será o nome do input (como o upload pode ser de vários arquivos, devemos utilizar os colchetes [] para indicar array), e qual a URL que será submetido os arquivos.

Para cada arquivo enviado com sucesso, suas IDs serão armazenadas em um input para serem enviadas no formulário e assim serem associadas ao seu model, conforme destaque na inspeção de elementos da imagem abaixo:



Para tratar o resultado do formulário na submissão, deverá ser feito algo do tipo:

```
foreach($input['users'] as $file) {
    //some action to save the file_id
}
```

Isso é tudo....