



Software Developer

PHP Laravel

Sara Monteiro

Formulários: Métodos HTTP

- O protocolo HTTP define um conjunto de Métodos que indicam a acção a ser executada para um dado recurso. Mais informação [aqui](#).
- Aprofundaremos o uso do GET e do POST para uso em formulários.

HTTP Method	CRUD operation	Entire Collection (e.g. /users)	Specific Item (e.g. /users/{id})
GET	Read	200 (OK), list of entities. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single entity. 404 (Not Found), if ID not found or invalid.
POST	Create	201 (Created), Response contains response similar to GET /user/{id} containing new ID.	not applicable
PATCH	Update	Batch API	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	204 (No Content). 400(Bad Request) if no filter is specified.	204 (No Content). 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	not implemented	not implemented

Formulários: POST

O método POST é usado essencialmente para criar recursos, nomeadamente: criar um novo utilizador através do Formulário, uma nova tarefa, etc..

No método POST devemos acrescentar um `action="rotaparaondenviam ososdados"` e o `@csrf` que é um helper de validação do Laravel.

```
<h1>Adicionar Utilizador</h1>

<form method="POST" action="{{ route('create_contact') }}">
    @csrf

    <div class="mb-3"> ...
    </div>
    <div class="mb-3"> ...
    </div>
    <div class="mb-3"> ...
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Formulários: Request Helper

O Laravel tem uma class de [Request](#) que nos permite interagir com os pedidos HTTP. Serve também para receber inputs, cookies e ficheiros submetidos: `use Illuminate\Http\Request;`

```
use App\Models\User;  
use Illuminate\Http\Request;
```

```
public function createUser(Request $request)  
{  
    //traz toda a informação do pedido  
    $request->all();  
  
    //traz o input com o name email  
    $request->email;  
}
```

Formulários: Validação no Backend

Na class `$request` temos um método de validação chamado [validate](#), que se usa da seguinte forma: `$request->validate()` com as regras de validação que queremos aplicar.

```
$request->validate([  
    'name' => 'string|max:50',  
    'email' => 'required|email|unique:users'  
]);
```

Formulários: Mensagens de Erro

Usando a validação, temos depois a possibilidade de customizar os nossos próprios erros na blade.

```
<label for= "exampleInputEmail" class= "form-label" >Email address</label>
<input type="email" name="email" class="form-control @error('email')
is-invalid @enderror"
        id="exampleInputEmail" aria-describedby="emailHelp">
        @error('email')
        <div class="invalid-feedback">
            Pf coloque um email!
        </div>
        @enderror
```

Se tentar submeter sem email, aparece a seguinte informação:

We'll never share your email with anyone else.

Email address

Pf coloque um email.

We'll never share your email with anyone else.

Password

Formulários: Manipular os Pedidos

Agora já temos todas as ferramentas para por exemplo adicionar um utilizador na base de dados.

Assim, seguimos os seguintes passos:

1. Adicionar o value="" e o name="nossoNome" no input do formulário.
2. No controller, na rota de chegada, validar se os campos estão de acordo com o que precisamos.
3. Inserir os campos recebidos na base de dados.

```
<input type="email" name="email" value=""  
      class="form-control @error('email') is-invalid @enderror"  
      id="exampleInputEmail1"  
      aria-describedby="emailHelp">  
@error('email')  
<div class="invalid-feedback">
```

```
public function createUser(Request $request)  
{  
    $request->validate([  
        'name' => 'string|max:50',  
        'email' => 'required|email|unique:users',  
        'password' => 'required'  
    ]);  
  
    User::insert([  
        'name' => $request->name,  
        'email' => $request->email,  
        'password' => Hash::make($request->password),  
    ]);  
  
    return redirect()->route('contacts.all')->with('message', 'Contacto adicionado com sucesso');  
}
```

Formulários: Manipular os Pedidos para inserir um registo

No final podemos redireccionar para a página de todos os contactos com uma mensagem nas variáveis de sessão. Na view específica abrimos um campo para mostrar essa mensagem, caso ela exista.

```
return redirect()->route('contacts.all')->with('message', 'Contacto adicionado  
com sucesso');
```

```
@if (session('message'))  
| <div class="alert alert-success">{{ session('message') }}</div>  
@endif
```

0000

Contacto adicionado com sucesso

Todos os Contactos ▾

#	Nome	Email	
1	Sara	sara@gmail.com	<button>Ver</button> <button>Apagar</button>

Formulários – Exercício



Centro para o Desenvolvimento
de Competências Digitais

À semelhança do efectuado com os Utilizadores, crie um botão, link, etc para adicionar uma tarefa através de um formulário.

O formulário deve conter:

- > nome da tarefa
- > descrição
- > selecção do utilizador a que vai ser atribuída escolhido de entre uma lista dos utilizadores que já temos na nossa base de dados.

No backend deve ser feita a validação com os seguintes items:

- > nome, descrição e users_id required.
- > nome deverá ser uma string com um máximo de 50caracteres.

Formulários: Update

A forma para fazer o update é semelhante à de Adicionar, só que iremos carregar o formulário com os dados do utilizador que já existem. Podemos, por exemplo, transformar a Blade de Ver Contacto num formulário.

No entanto, para fazer update precisamos de um where. Como o campo mais adequado para fazer pesquisas é o id podemos criar na blade um input hidden com o id para que este seja enviado para o backend sob a forma de request->id.

```
<input type="hidden" name="id" value="{{ $ourUser->id }}">
```

```
<form method="POST" action="{{ route('update_contact') }}">
    @csrf
    <div class="mb-3">
```

```
<input type="text" class="form-control" name="name" value="{{
$ourUser->name }}" id="name"
    aria-describedby="emailHelp"
    @error('name')
        <div class="invalid-feedback">
            {{ $message }}
        </div>
    @enderror
    <div id="emailHelp" class="form-text">We'll never share your email with
    anyone else.</div>
</div>
</div class="mb-3">
<label for="exampleInputEmail1" class="form-label">Email address</label>
<input type="email" name="email" value="{{ $ourUser->email }}"
    class="form-control @error('email') is-invalid @enderror"
    id="exampleInputEmail1"
```

Formulários: Update

No Backend fazemos o método de Update através do id e trocamos a mensagem da sessão.

```
}  
public function updateUser(Request $request)  
{  
    User::where('id', $request->id)  
        ->update([  
        'name' => $request->name,  
        'email' => $request->email,  
        'password' => Hash::make($request->password),  
    ]);  
    return redirect()->route('contacts.all')->with('message', 'Contacto actualizado com sucesso');  
}
```

Formulários: GET

O método GET serve por exemplo para filtrar dados num Select. Neste exemplo estamos a carregar todos os contactos e à medida que mudamos a selecção ele coloca no value o id que corresponde ao que escolhemos.

Users

	Email
Todos os Contactos	
Todos os Contactos	
Sara	sara@gmail.com
Bruno	
Hélder	Bruno@gmail.com
Ana	
Marcia	
3	Hélder Hélder@gmail.com

Acção/ Método

Nome pelo qual chamamos o valor / key

```
<form method="GET">
  <select class="custom-select" name="user_id" onchange="this.form.
    submit()">
    <option value="" selected>Todos os Contactos</option>
    @foreach ($contacts as $item)
      <option @if ($item->id == request()->query('user_id'))
        selected @endif value="{{ $item->id }}">
        {{ $item->name }}</option>
    @endforeach
  </select>
</form>
```

Valor a enviar

Formulários: GET

No backend, na mesma função onde chamamos todos os contactos fazemos um if/ else para filtro e caso tenhamos valores na query GET (request()->query('aNossaKey')) filtramos só os contactos correspondentes.

```
public function allContacts()
{
    if (request()->query('user_id')) {
        $contacts = User::where('id', request()->query('user_id'))
            ->get();
    } else {
        $contacts = User::all();
    }
}
```

Procurar Dados

Agora que já sabemos como interagir com a Base de Dados, falta aprender como fazer procura de dados. O método que usaremos é o GET e a forma é bastante semelhante à de para selecionar um User no multiselect.

Users

Todos os Contactos ▾

Procurar

#	Nome	Email	
1	Sara	novosass@gmail.com	<div>VerApagar</div>
2	Bruno	Brunof@gmail.com	<div>VerApagar</div>

Na View:

```
</form>
<form action="">
  <input class="ms-5" type="text" value="{{ request()->query('search') }}" name="search"
  | id="" placeholder="Procurar">
  <button class="btn btn-secondary">Procurar</button>
</form>
```

Procurar Dados

No Controller onde temos a query geral, filtramos caso tenha aquele valor e redefinimos a query:

```
$search = request()->query('search') ? request()->query('search') : null;

$query = DB::table('users');

if (!empty(request()->query('user_id'))) {
    $query->where('id', request()->query('user_id'));
}

if ($search) {
    $query->where("name", "LIKE", "%{$search}%");
    $query->orWhere("email", "LIKE", "%{$search}%");
}

$allUsers = $query->get();
```

Na Tabela serão apresentados apenas os que correspondem à nossa pesquisa.

Users

Todos os Contactos ▾

Brunof@gmail.com

Procurar

#	Nome	Email	
2	Bruno	Brunof@gmail.com	Ver Apagar

Autenticação

- Mecanismo para verificar as Credenciais do Utilizador
- Na maioria das Aplicações Web a Autenticação é gerida pela sessão
- Assim que Autenticado, o User pode aceder a certas funcionalidades da Aplicação
- As credenciais podem ser armazenadas na sessão durante um certo período de tempo para que o User não esteja sempre a ter que as colocar.
- Em Laravel existem uma série de pacotes criados para facilitar o processo de Autenticação.

[Documentação](#)

Autenticação & Segurança

Para termos um sistema de Autenticação funcional e seguro, precisamos:

1. **User Model**
2. **Login Controller**, por exemplo chamado AuthController com as funções:
 - create() -> um formulário para submeter algo, seja uma criação de User ou o Login
 - store() -> cria a sessão caso o Login seja válido
 - destroy()-> para fazer logout.
3. **Rotas** para as funções acima descritas.
4. **Pasta para as Views** relacionadas com Autenticação, por exemplo Auth.

Autenticação & Segurança

Quando o processo de autenticação fica ok o User fica com os seus dados guardados na sessão, podendo ser usados por exemplo para mensagens customizadas:

```
public function store(Request $request)
{
    $credentials = $request->only('email', 'password');

    if (Auth::attempt($credentials)) {
        return redirect()->intended('home');
    }

    return back();
}
```

```
<div class="sm:fixed sm:top-0 sm:right-0 p-6 text-right">
    @auth
        <a href="{{ url('/home') }}"
            class="font-semibold text-gray-600 hover:text-gray-900 dark:text-gray-400
            dark:hover:text-white focus:outline focus:outline-2 focus:rounded-sm
            focus:outline-red-500">Home</a>
        <h1>Olá Utilizador</h1>
    @else
        <a href="{{ route('login') }}"
            class="font-semibold text-gray-600 hover:text-gray-900 dark:text-gray-400
            dark:hover:text-white focus:outline focus:outline-2 focus:rounded-sm
            focus:outline-red-500">Log
        in</a>
    @endauth
</div>
```

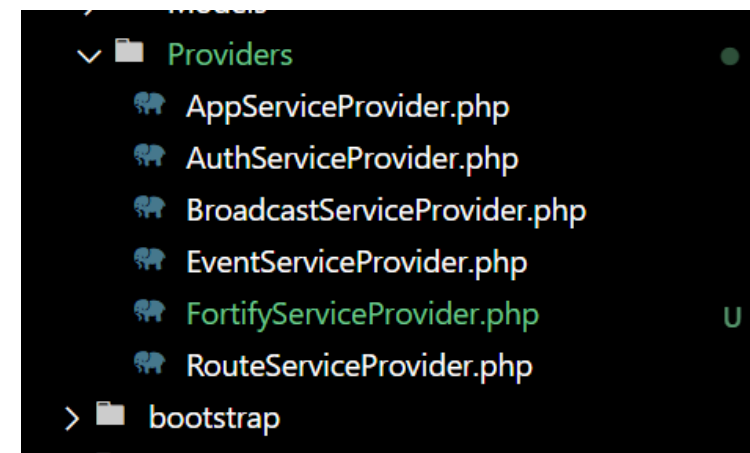
Autenticação - Fortify

Para criar o nosso sistema de Autenticação iremos usar o Fortify como apoio por ser um package agnóstico, deixando espaço para podermos ser flexíveis.

- [Instalação e Setup](#)

Após instalação, vários recursos são adicionados à nossa app.

```
POST login ..... Laravel\Fortify > AuthenticatedSessionController@store
POST logout ..... logout > Laravel\Fortify > AuthenticatedSessionController@destroy
GET|HEAD portfolio
```



Autenticação Fortify

Ao usar o Fortify, verificamos que na lista de rotas já temos uma rota para login. Assim, basta-nos configurar a nossa View com o formulário que receba os parâmetros descritos [aqui](#) e apontar para a View no `FortifyServiceProvider`.

```
/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Fortify::loginView(function () {
        return view('auth.login');
    });

    Fortify::createUsersUsing(CreateNewUser::class);
    Fortify::updateUserProfileTransformationUsing(UpdateUserProfileTransformation::class);
}
```

```
laravel Blade .....
GET|HEAD login ..... login > Laravel\Fortify > AuthenticatedSessionController@create
POST login ..... Laravel\Fortify > AuthenticatedSessionController@store
POST logout ..... logout > Laravel\Fortify > AuthenticatedSessionController@destroy
```

Receber Dados de um User Autenticado

A partir do momento em que o Utilizador é autenticado com sucesso, temos acesso aos seus dados em toda a aplicação.

Desta forma podemos customizar mensagens com o nome dele, validar o perfil para acesso a determinadas funcionalidades, etc.

[Documentação](#)

Exemplos:

- `Auth::user()` retorna todo o objecto do Utilizador Autenticado
- `Auth::id()` retorna o id do Utilizador Autenticado
- `Auth::user()->name` retorna o nome do Utilizador Autenticado

```
@auth
    <a href="{{ url('/home') }}"
        class="font-semibold text-gray-600 hover:text-gray-900 dark:text-gray-400
        dark:hover:text-white focus:outline focus:outline-2 focus:rounded-sm
        focus:outline-red-500">Home</a>
    <h1>Olá {{ Auth::user()->name }}</h1>
    <form method="POST" action="{{ route('logout') }}">
        @csrf
        <button type="submit">Logout</button>
    </form>
@else
    <a href="{{ route('login') }}"
```

Logout

Ao consultar a lista de rotas podemos verificar que existe numa rota de Logout através do Foritfy. Assim, para instalar o processo de logout bastará encapsular um botão para a funcionalidade num form e apontar para a rota de Logout.

```
<form method="POST" action="{{ route('logout') }}">  
    @csrf  
    <button type="submit">Logout</button>  
</form>
```

```
@else You 5 seconds ago • Uncommitted changes
```

Rotas protegidas

Já aprendemos como podemos bloquear na blade certos componentes para utilizadores não autenticados. No entanto, em aplicações grandes e para maior segurança, podemos escolher bloquear uma rota para determinados perfis.

Proteger rotas é indicado nos casos em que temos vários perfis de utilizadores e queremos “isolar” o acesso que eles têm aos dados.

Serve também para bloquear por exemplo um acesso ao BackOffice de gestão a utilizadores autenticados.. Desta forma sempre que um utilizador não autenticado colocar a url de uma rota protegida, é reencaminhado para o form de login.

A forma de proteger rotas é usando uma Middleware.

Rotas protegidas: Middleware

Podemos encontrar rotas default definidas pelo Laravel.

Para bloquear por exemplo uma rota a users autenticados usamos o alias 'auth' e adicionamos à nossa rota.

[Documentação](#)

Alias	Middleware
auth	Illuminate\Auth\Middleware\Authenticate
auth.basic	Illuminate\Auth\Middleware\AuthenticateWithBasicAuth
auth.session	Illuminate\Session\Middleware\AuthenticateSession
cache.headers	Illuminate\Http\Middleware/SetCacheHeaders
can	Illuminate\Auth\Middleware\Authorize
guest	Illuminate\Auth\Middleware\RedirectIfAuthenticated
password.confirm	Illuminate\Auth\Middleware\RequirePassword
precognitive	Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests
signed	Illuminate\Routing\Middleware\ValidateSignature
subscribed	\Spark\Http\Middleware\VerifyBillableIsSubscribed
throttle	Illuminate\Routing\Middleware\ThrottleRequests or Illuminate\Routing\Middleware\ThrottleRequestsWithRedis
verified	Illuminate\Auth\Middleware\EnsureEmailIsVerified

```
Route::get('/home_dashboard', [DashboardController::class, 'index'])->name('home_dashboard')->middleware('auth');
```


Rotas protegidas: Middleware

Podemos também criar as nossas próprias Middlewares correndo o comando
php artisan make:middleware
EnsureTokenIsValid.
No ficheiro definimos o código que da validação que pretendemos fazer e aplicamos a nossa middleware à rota.

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::get('/profile', function () {
    // ...
})->middleware(EnsureTokenIsValid::class);
```

```
namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureTokenIsValid
{
    /**
     * Handle an incoming request.
     *
     * @param  \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)  $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        if ($request->input('token') !== 'my-secret-token') {
            return redirect('home');
        }

        return $next($request);
    }
}
```

Middleware - Exercício



Centro para o Desenvolvimento
de Competências Digitais

1. Criar um controller chamado DashboardController que irá trabalhar as views de Backoffice.
2. Criar uma rota para a Home do Dashboard e encaminhar para uma função do Controller que nos retorne uma View.
3. Na nossa rota, adicionar a middleware de user autenticado.
4. No Navbar, num dos menús, colocar BackOffice e reencaminhar para a rota criada. Na View colocar Olá, (nome do user autenticado).
5. Através de uma migração, adicionar uma coluna chamada user_type à tabela de users, que receba um inteiro. Correr a migração e colocar directamente o valor 1 num dos utilizadores.
6. Na dashboard criar um alerta Bootstrap que diga Conta de Administrador, caso se autentique com o administrador.

Autenticação – Recuperação de Password

```
POST reset-password ..... password.update > Laravel\Fortify > NewPasswordController@store
GET|HEAD reset-password/{token} ..... password.reset > Laravel\Fortify > NewPasswordController@create
GET|HEAD sanctum/csrf-cookie ..... sanctum.csrf.cookie > Laravel\Sanctum > CsrpCookieController@store

<label for="exampleInputPassword1" class="form-label">Password</label>
<input name="password" type="password" class="form-control" id="examplePassword1">
</div>
<div class="mb-3 form-check">
<input type="checkbox" class="form-check-input" id="exampleCheck1">
<label class="form-check-label" for="exampleCheck1">Check me out</label>
</div>
<button type="submit" class="btn btn-primary">Submit</button>
<a href="{{ route('password.request') }}">Esqueceu-se da pass?</a>
</form>
```

Uma vez que temos já as rotas para recuperação da pass disponíveis através do Fortify, vamos criar no nosso form de login um sítio para solicitar essa recuperação.

A seguir, teremos que registar no nosso FortifyServiceProvider a funcionalidade.

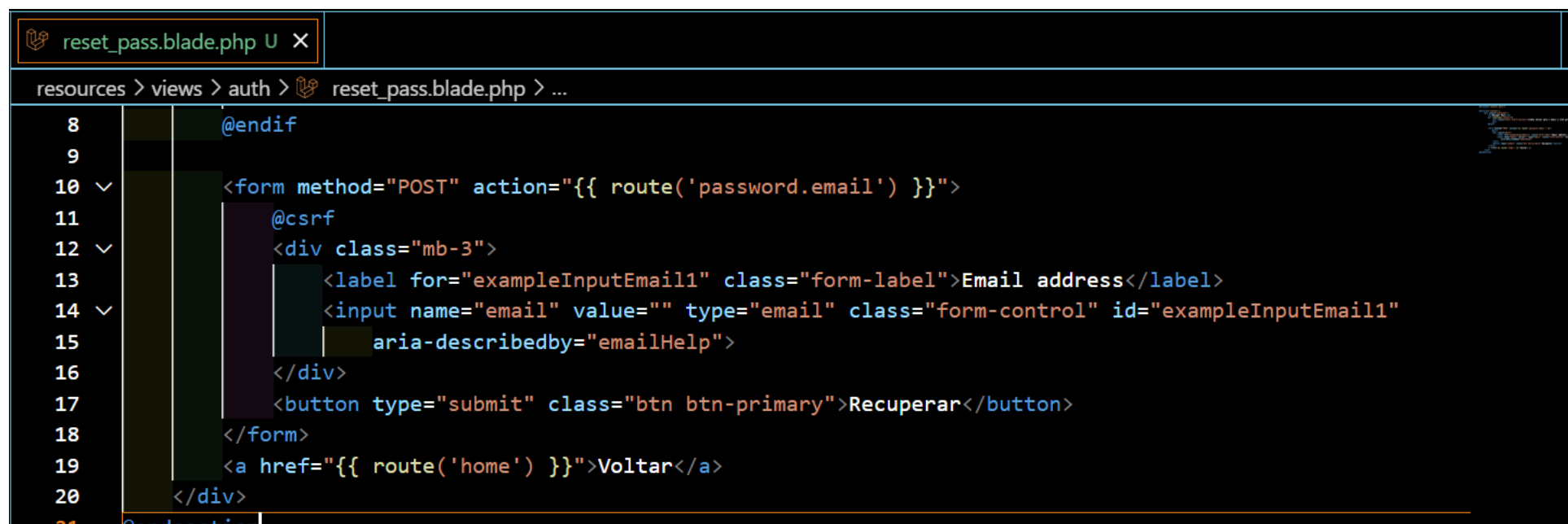
```
FortifyServiceProvider.php U X
app > Providers > FortifyServiceProvider.php > FortifyServiceProvider > boot

25 /**
26  * Bootstrap any application services.
27  */
28 public function boot(): void
29 {
30     Fortify::loginView(function () {
31         return view('auth.login');
32     });
33
34     Fortify::requestPasswordResetLinkView(function () {
35         return view('auth.reset_pass');
36     });
37
38     Fortify::createUsersUsing(CreateNewUser::class);
```

Autenticação – Recuperação de Password

Em seguida criamos a View de recuperação de Pass semelhante à do login, mas apenas com espaço para colocar o email e cujo POST envie para password.email.

Para que o email seja enviado, precisamos de configurar o nosso email no .env file.

A screenshot of a code editor window titled 'reset_pass.blade.php U'. The breadcrumb navigation shows 'resources > views > auth > reset_pass.blade.php > ...'. The code is in a dark theme and shows a Blade template for a password reset form. It includes a form with a POST method, CSRF protection, an email input field, and a 'Recuperar' button. Line numbers 8 through 20 are visible on the left.

```
8      @endif
9
10     <form method="POST" action="{{ route('password.email') }}">
11         @csrf
12         <div class="mb-3">
13             <label for="exampleInputEmail1" class="form-label">Email address</label>
14             <input name="email" value="" type="email" class="form-control" id="exampleInputEmail1"
15                 aria-describedby="emailHelp">
16         </div>
17         <button type="submit" class="btn btn-primary">Recuperar</button>
18     </form>
19     <a href="{{ route('home') }}">Voltar</a>
20 </div>
```

Servidor de Email

As configurações de email encontram-se no ficheiro .env. Para usar o gmail temos que criar uma password de aplicação. Podemos seguir este [tutorial](#).

Aqui iremos criar uma conta gratuita no [mailtrap](#) e usar desta forma. Para colocar uma nova pass, registaremos a nossa view para colocar a nova pass.

```
30
31 MAIL_MAILER=smtp
32 MAIL_HOST=sandbox.smtp.mailtrap.io
33 MAIL_PORT=2525
34 MAIL_USERNAME=37a2ec90109421
35 MAIL_PASSWORD=85b6b3c9a07463
36 MAIL_ENCRYPTION=tls
37 MAIL_FROM_ADDRESS="hello@example.com"
38 MAIL_FROM_NAME="${APP_NAME}"
39
```

```
public function boot(): void
{
    Fortify::loginView(function () {
        return view('auth.login');
    });

    Fortify::requestPasswordResetLinkView(function () {
        return view('auth.reset_pass');
    });

    Fortify::resetPasswordView(function(){
        return view('auth.new_pass');
    })
}
```

Autenticação – Actualizar Password

Por fim, falta apenas a Blade de actualizar a password. Para que funcione precisamos de colocar no value do email o `{{ $request->email }}` e enviar o token de validação. Assim, a actualização ficará concluída e é-nos enviado para o form de login.

```
@endif
<form method="POST" action="{{ route('password.update') }}">
    @csrf
    <div class="mb-3">
        <label for="exampleInputEmail1" class="form-label">Email address</label>
        <input name="email" value="{{ request()->email }}" type="email" class="form-control"
            id="exampleInputEmail1" aria-describedby="emailHelp">
    </div>
    <div class="mb-3">
        <label for="password" class="form-label">Password</label>
        <input type="password" class="form-control @error('password') is-invalid @enderror"
            name="password" />
        @error('password')
            <div class="invalid-feedback">{{ $message }}</div>
        @enderror
    </div>
    <div class="mb-3">
        <label for="password" class="form-label">Password Confirmation</label>
        <input type="password" class="form-control" name="password_confirmation" />
        <input type="hidden" name="token" value="{{ request()->route('token') }}">
    </div>
    <input type="hidden" name="token" value="{{ request()->route('token') }}">
    <button type="submit" class="btn btn-primary">Submeter nova pass!</button>
</form>
```

Autenticação – UX

Ao longo de todo o processo não esquecer de customizar na blade as mensagens que orientam o utilizador, seja ela a indicar que vai ser enviado um link para actualizar a password ou a indicar erros de email, password, etc..

```
<div class="container">
  <h1>Recuper Pass</h1>
  @if (session('status'))
    <div class="alert alert-success">Iremos enviar para o email o link para recuperação da pass!</div>
  @endif
```

```
<div class="mb-3">
  <label for="password" class="form-label">Password</label>
  <input type="password" class="form-control @error('password') is-invalid @enderror"
    name="password" />
  @error('password')
    <div class="invalid-feedback">{{ $message }}</div>
  @enderror
</div>
```

Armazenamento de ficheiros

- O Laravel tem um [sistema](#) muito completo para armazenar ficheiros
- A integração do Laravel Filesystem fornece drivers simples para trabalhar com sistemas de arquivos locais, SFTP e Amazon S3, sendo muito simples alternar entre o desenvolvimento local e o servidor de produção, pois a API permanece a mesma para cada sistema.
- As configurações de armazenamento encontram-se no ficheiro filesystems.php e o sistema de armazenamento é definido como local por defeito

```
'default' => env('FILESYSTEM_DISK', 'local'),

/*
|-----
| Filesystem Disks
|-----
|
| Here you may configure as many filesystem "disks" as you wish, and you
| may even configure multiple disks of the same driver. Defaults have
| been set up for each driver as an example of the required values.
|
| Supported Drivers: "local", "ftp", "sftp", "s3"
|
*/

'disks' => [
    'local' => [
        'driver' => 'local',
        'root' => storage_path('app'),
        'throw' => false,
    ],

```


Armazenamento de ficheiros – tornar os ficheiros acessíveis

Por norma, vamos ter ficheiros que vão ser acedidos visualizados por outros: por exemplo foto. Para isso temos que os tornar públicos. Para isso temos que mudar a configuração no .env e criar um symbolic link para que os ficheiros que guardamos no storage tenham link na pasta public.

```
CACHE_DRIVER=file  
FILESYSTEM_DISK=public  
QUEUE_CONNECTION=sync
```

```
PS C:\Users\Utilizador\Documents\Work\Cesae\myCode\laravelSDTests> php artisan storage:link
```

```
INFO The [C:\Users\Utilizador\Documents\Work\Cesae\myCode\laravelSDTests\public\storage] link has been connected to [C:\Users\Utilizador\Documents\Work\Cesae\myCode\laravelSDTests\storage\app/public].
```

Armazenamento de ficheiros – Ler ficheiros

Para interagir com a Storage do Laravel iremos usar a Facade Storage: `Illuminate\Support\Facades\Storage`.

Ao longo do nosso desenvolvimento iremos precisar de usar imagens e para isso precisamos de saber os seus caminhos. Podemos usar os seguintes comandos para isso:

- `Storage::url('aminhaimagem.img')` -> retorna o caminho através do servidor.
- `Storage::path('aminhaimagem.img')` -> retorna o caminho na nossa aplicação.
- `Storage::exists('aminhaimagem.img')` -> retorna true ou false conforme exista ou não
- `Storage::size('aminhaimagem.img')` -> retorna o tamanho da imagem em bytes
- `Storage::lastModified('aminhaimagem.img')` -> retorna a data em que foi modificado
- `Storage::download('aminhaimagem.img')` -> retorna o download

Armazenamento de ficheiros – Guardar ficheiros

Podemos criar e alterar ficheiros de texto da seguinte forma:

- `Storage::put('localização relativa ao disco', 'Conteúdo do ficheiro');` -> guardar ficheiros
- `Storage::prepend('localização relativa ao disco', 'Conteúdo do ficheiro Adicionado');` -> Adicionar conteúdo ao ficheiro
- `Storage::append('localização relativa ao disco', 'Conteúdo do ficheiro Adicionado');` -> Adicionar conteúdo ao ficheiro

No que diz respeito ao armazenamento de imagens, iremos guardar o objecto da Imagem (que vem por exemplo no nosso request do Front End)

- `Storage::putFile('pasta para o ficheiro', 'objecto do ficheiro');` -> Adicionar imagem a uma pasta
- `Storage::putFileAs('pasta para o ficheiro', 'objecto do ficheiro', 'nome do ficheiro');` -> Adicionar imagem com nome específico a uma pasta

Armazenamento de ficheiros – Aplicar ao nosso form

Para aplicar o armazenamento de ficheiros, criaremos um update para a nossa lista de users e adicionaremos uma coluna chamada photo na tabela para guardar o caminho da imagem. O caminho deve ser guardado como string.

Para preparar o nosso form para uma imagem devemos ter em consideração o seguinte:

- no form adicionar `enctype="multipart/form-data"` para que receba ficheiros
- No input, `type=file` para sabermos que estamos a receber um ficheiro
- Também no input, `accept="image/*"` para sabermos que tipo de imagem aceitar, tamanho, etc

```
<form method="POST" action="{{ route('edit_user') }}" enctype="multipart/form-data">
    @csrf
    <input type="hidden" name="id" value="{{ $ourUser->id }}">
    <div class="mb-3">
        <label for="name" class="form-label">Name</label>
        <input type="text" name="name" value="{{ $ourUser->name }}" class="form-control" id="name">
    </div>
    <div class="mb-3">
        <label for="photo" class="form-label">Photo</label>
        <input type="file" accept="image/*" name="photo" class="form-control" id="photo">
        @error('photo')
            photo
        @enderror
    </div>
    <button type="submit" class="btn btn-primary">Atualizar</button>
</form>
```

Armazenamento de ficheiros – Guardar na Base de Dados

No backend devemos primeiro fazer debug para verificar que atributos estamos a receber na imagem, e devemos criar uma validação para que o campo seja do tipo imagem.

```
public function editUser(Request $request)
{
    dd($request->photo);

    $request->validate([
        [
            'name' => 'required',
            'photo' => 'image',
        ]
    ]);
}
```

```
Illuminate\Http\UploadedFile {#380 ▼ // app\Http\Controllers\UserController.php:61
  -test: false
  -originalName: "photo-1623172959921-630212f71058.avif"
  -mimeType: "image/avif"
  -error: 0
  #hashName: null
  path: "C:\Users\Utilizador\AppData\Local\Temp"
  filename: "php8C94.tmp"
  basename: "php8C94.tmp"
  pathname: "C:\Users\Utilizador\AppData\Local\Temp\php8C94.tmp"
  extension: "tmp"
  realPath: "C:\Users\Utilizador\AppData\Local\Temp\php8C94.tmp"
  aTime: 2023-05-30 10:44:14
  mTime: 2023-05-30 10:44:14
  cTime: 2023-05-30 10:44:14
  inode: 13229323905566555
  size: 12445
  perms: 0100666
  owner: 0
}
```

Armazenamento de ficheiros – Guardar na Base de Dados

Para armazenar o ficheiro na base de dados, iremos armazenar o caminho que nos é retornado como o sítio onde a nossa imagem foi guardada no projecto. A coluna na base de dados guardará então a direcção para o nosso ficheiro.

```
    $photo = null;

    if ($request->hasFile('photo')) {
        $photo = Storage::putFile('uploadedImages', $request->photo);
    }

    DB::table('users')
        ->where('id', $request->id)
        ->update([
            'name' => $request->name,
            'photo' => $photo
        ]);

    return redirect('home_all_users')->with('message', 'Utilizador actualizado com sucesso');
}
```

Armazenamento de ficheiros – Mostrar uma imagem

Por último, falta-nos mostrar a photo do nosso utilizador caso ele tenha uma guardada. Para isso, na lista de utilizadores da nossa Blade iremos adicionar uma coluna para o efeito e mostrá-la através do helper asset.

```
<th scope="row">{{ $item->id }}</th>
<td><img width="30px" height="30px"
src='{{ $item->photo ? asset('storage/' . $item->photo) : asset('images/
nophoto.jpg') }}'> You, 1 second ago • Uncommitted changes
</td>
<td>{{ $item->name }}</td>
<td>{{ $item->email }}</td>
```

Recursos

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