

Création d'un gestionnaire de membres de club sportif : Le panneau d'administration

Dans ce TD, nous allons travailler sur la mise en place du panneau d'administration de l'application

Table des matières

Les routes.....	2
Contrôle des accès.....	3
Routes pour la gestion des utilisateurs.....	5
Route de suppression des utilisateurs.....	5
Rédigeons l'ensemble des routes.....	6
Le contrôleur des Users.....	7
Afficher les utilisateurs.....	7
Afficher les données dans une vue.....	9
Les méthodes show, edit, update et destroy.....	9
Les méthodes « show » et « edit ».....	10
Notre contrôleur final UsersController.php.....	13

Nous avons vu précédemment qu'avec l'utilisation du package de Laravel Breeze, nous avons un système d'authentification complet et fonctionnel que nous avons testé en saisissant directement les url /login, /register.

Le package Breeze installe

- Le dossier `Auth` dans les `contrôleurs` : c'est notamment dans ce dossier que l'on fera des modifications
- Le fichier `auth.php` dans les `routes`
- Le dossier `auth` dans les `views` (`resources`) : pour modifier le design des pages
- Toutes les dépendances de `vendor` et `node_modules`

Si l'on se connecte, nous arrivons directement au **tableau de bord** de l'application (dashboard).

Ce n'est pas le comportement souhaité dans le cadre de notre application.

Effectivement, une fois connecté, on veut juste rediriger nos membres du club sur l'accueil (ultérieurement sur leur profil) :

seuls les administrateurs pourront accéder au panneau d'administration.

Pour modifier ce comportement, allez dans `app -> providers -> RouteServiceProvider.php` pour changer la valeur de la constante `HOME` comme suit :

```
public const HOME = '/';
```

Les routes

Ouvrez votre fichier `routes/web.php` et observez les lignes ajoutées par Breeze :

```
Route::get('/dashboard', function () {
    return view('dashboard');
})->middleware(['auth'])->name('dashboard');

require __DIR__.'/auth.php';
```

On remarque l'utilisation d'une méthode statique de la classe `Route` fournie par Laravel. Cette méthode s'appelle `get`.

On fournit à cette méthode 2 paramètres :

- le premier est la route en elle-même, c'est à dire ce qui va s'afficher dans l'URL.
- Ensuite, c'est le *callback*, c'est à dire ce qui va être exécuté quand on arrivera sur cette URL (ou route).

Dans le cas de notre panneau d'administration (cette route vient de *Breeze*) Laravel a donc déjà défini la route `/dashboard`.

Quand on y accède, on retourne une vue grâce à `return view('nom de la vue')`.

La fonction `view` est selon le vocabulaire Laravel un « *help*eur » : une fonction définie globalement. De nombreux « *help*eurs » retournent des instances de classes ou des méthodes de classes.

Un fichier `routes/auth.php`, a été généré par Laravel Breeze et est inclus dans `web.php`. La méthode statique `get` et toutes les autres de notre classe `Route` sont très pratiques car elles retournent l'instance de `Route`, on peut donc mettre une suite de méthodes comme suit : `Route::get()->methode1()->methode2()->methode3...`

```
->middleware(['auth'])->name('dashboard');
```

La méthode `middleware()` permet d'assigner un `Middleware` à la route, c'est à dire, pour faire simple, un contrôle d'accès défini dans une classe dont on passe le nom en paramètre (dans notre cas, c'est "auth").

```
->middleware(['auth'])->name('dashboard');
```

La méthode `name()` permet de nommer une route. C'est très pratique pour faire des liens vers cette route dans notre application sans se soucier d'où l'on est.

Ici, on retourne une vue (`dashboard`) sans passer par un contrôleur (= performances plus intéressantes).

Pour gérer nos « users », on aura cependant besoin d'un **contrôleur**...

```
php artisan make:controller UsersController
```

On peut retrouver notre contrôleur dans `app -> Http -> Controllers`.

Contrôle des accès

Il faut tout d'abord créer un **middleware** qui contrôlera que l'utilisateur qui essaie d'accéder à la route est bien un administrateur.

```
php artisan make:middleware Admin
```

`app -> Http -> Middleware -> Admin.php`

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;

class Admin
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure(\Illuminate\Http\Request): (\Illuminate\Http\Response|\Illuminate\Http\RedirectResponse)  $next
     * @return \Illuminate\Http\Response|\Illuminate\Http\RedirectResponse
     */
    public function handle(Request $request, Closure $next)
    {
        return $next($request);
    }
}
```

En détail :

```
public function handle(Request $request, Closure $next)
{
    return $next($request);
}
```

La méthode `handle()` est celle appelée quand on assigne un middleware à une route.

C'est donc à l'intérieur de celle-ci que l'on agit.

Le **premier paramètre** passé automatiquement par Laravel est la requête que l'on cherche à exécuter (on a donc plein d'infos sur la route, le contexte, l'utilisateur connecté...) et **le second**, passé automatiquement lui aussi, est ce qu'on doit faire si l'on passe le contrôle d'accès défini dans la méthode.

Nous, on veut simplement vérifier que notre utilisateur courant est administrateur.

Heureusement, le paramètre `$request` a une méthode `user()` qui nous retourne toutes les informations sur l'utilisateur connecté, notamment la valeur du booléen "admin" que l'on a mis dans base dans la table users !

Réécrivons la fonction :

```
public function handle(Request $request, Closure $next)
{
    if($request->user()->admin) {
        return $next($request);
    } else {
        abort(403); // voir les codes http1
    }
}
```

Pour pouvoir utiliser ce middleware, il faut simple dire à Laravel que l'on peut l'assigner à des routes.

Pour ça rendez-vous dans  app -> Http -> Kernel.php et modifiez le tableau `$routeMiddleware` comme suit (cf. dernière ligne) 'admin' :

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'password.confirm' => \Illuminate\Auth\Middleware\RequirePassword::class,
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
    'admin' => \App\Http\Middleware\Admin::class, // Ici notre middleware !
];
```

Maintenant nous pouvons créer la route :

```
Route::get('/users', [UsersController::class, 'index'])->middleware(['auth', 'admin'])->name('users');
```

Tu peux mettre un `echo` quelconque dans une méthode `index()` dans `UserController` pour voir le résultat lorsque tu vas sur <http://127.0.0.1:8000/users>

```
class UserController extends Controller
{
    public function index ()
    {
        echo 'yes we can!';
    }
}
```

Si tu n'es pas connecté, cela te ramènera vers le formulaire de connexion (grâce au middleware `auth`).

Si tu l'es **et** que tu es admin (si tu utilises l'utilisateur créé grâce au factory), dans ce cas la page s'affichera avec ce que tu as stipulé dans l'écho...
Sinon, tu auras une erreur 403...

Nous devons également appliquer le middleware `Admin` sur le Tableau de bord...

Modifiez la route dans le fichier `web.php` 😊

¹ https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP

Routes pour la gestion des utilisateurs

Continuons d'écrire nos routes pour la gestion des utilisateurs...

Tout d'abord, listons les actions du CRUD :

- Lister les utilisateurs
- Voir un utilisateur en particulier (voir le profil)
- Modifier un utilisateur (modifier le profil)
- Ajouter un utilisateur (s'inscrire)
- Supprimer un utilisateur

On part du principe que la modification d'un utilisateur reviendra à la modification du profil par ce dernier → cela sera donc la même route aussi bien pour l'utilisateur que pour les administrateurs

Nous avons donc besoin des routes suivantes (je précise les types de requête également²) :

GET : lister les utilisateurs

DELETE : supprimer un utilisateur

Mémo requêtes http et Méthodes :

GET : récupérer des informations

POST : envoyer des informations

PUT, PATCH : modifier des informations

DELETE : supprimer des informations

Laravel nous simplifie la vie pour gérer tous les types de requêtes.

Pour dire qu'une route est de type TYPE, il nous suffit d'écrire :

`Route::TYPE('route', callback)...;`

Reprenons un extrait de notre fichier de routes web.php :

```
Route::get('/dashboard', function () { return view('dashboard'); } )
->middleware(['auth'])->name('dashboard');

Route::get('/users', [UserController::class, 'index'])->middleware(['auth',
'admin'])->name('users');
```

Route de suppression des utilisateurs

```
Route::delete('/users/{id}', [UserController::class,
'destroy'])->middleware(['auth', 'admin'])->name('users.destroy');
```

Les "mots" que l'on met entre accolades dans une route correspondent aux données qui peuvent changer dans la route. Ce sont les paramètres de la route.

Sans réécriture d'URL (<https://httpd.apache.org/docs/2.4/fr/rewrite/intro.html>), notre URL pour supprimer un utilisateur aurait ressemblé à `index.php?action=users.destroy&id=1`.

Les informations sont directement écrites dans l'URL et l'utilisateur curieux (ou mal intentionné) peut facilement les modifier !

Maintenant, avec la réécriture d'URL, nous faisons une requête de type delete en utilisant l'URL suivante : `users/1`.

² <https://laravel.sillo.org/cours-laravel-8-les-bases-le-routage/> : Les requêtes HTTP

Il faut savoir que dans notre cas, à la place de `{id}`, on pourrait mettre tout et n'importe quoi. Notre route va accepter toutes les requêtes qui ont la même forme, comme par exemple : `users/1`, `users/1233`, `users/bonjour`...

Heureusement, Laravel nous permet de vérifier ces paramètres de route avec des regex (expressions régulières, qui ressemblent à `^[0-9]{2,}[aA-zZ]?$`)³. Les regex vont, par exemple, permettre de vérifier que l'id de l'utilisateur qui est donné ne contient que des chiffres.

Rédigeons l'ensemble des routes

```
Route::get('/dashboard', function () { return view('dashboard'); } )
->middleware(['auth'])->name('dashboard');

Route::get('/users', [UserController::class, 'index'])->middleware(['auth',
'admin'])->name('users');

Route::get('/users/profile/{user}', [UserController::class, 'show'])
->name('users.show');

Route::get('/users/{user}', [UserController::class, 'edit'])
->middleware(['auth'])->name('users.edit');

Route::put('/users/{user}', [UserController::class, 'update'])
->middleware(['auth'])->name('users.update');

Route::delete('/users/{user}', [UserController::class, 'destroy'])
->middleware(['auth', 'admin'])->name('users.destroy');
```


Je peux mettre la même URL pour plusieurs routes car le type de la route (get, put...) lui n'est pas le même !
Laravel redirigera donc vers la bonne méthode du contrôleur même si l'URL est la même.

`users/{id}` ou `users/{user}` ??

Laravel propose ce qu'on appelle de l'**implicit binding** : au lieu de passer dans nos routes l'ID de l'utilisateur, on va passer l'utilisateur entier.

³ <https://www.quennec.fr/trucs-astuces/langages/php/php-regex>

Le contrôleur des Users

Reprenons notre  app -> Http -> Controllers -> UsersController.php

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class UsersController extends Controller
{
    // test
    public function index ()
    {
        echo 'yes !';
    }
}
```

La classe Controller permet d'utiliser dans nos controllers les fonctionnalités offertes par Laravel pour gérer les droits d'accès, les vues...

La classe Request permet comme son nom l'indique de gérer les requêtes ! On l'utilisera quand on doit récupérer des informations envoyées par formulaire.

Afficher les utilisateurs

Rappel :

Notre route est

```
Route::get('/users', [UsersController::class, 'index'])
->middleware(['auth', 'admin'])->name('users');
```

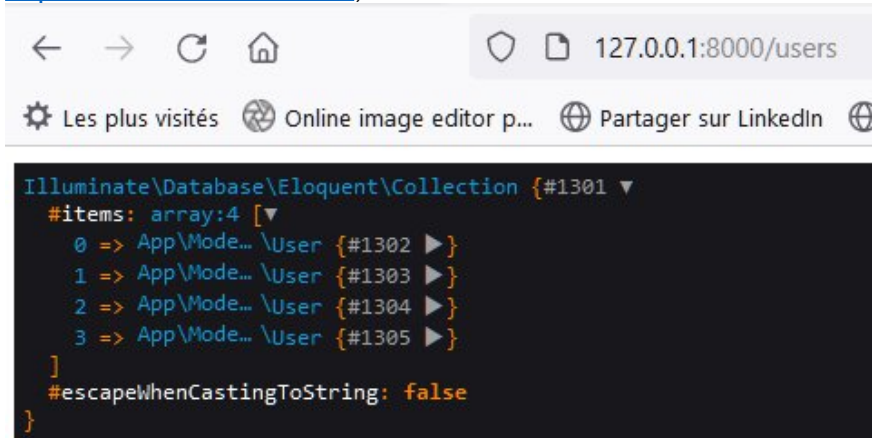
Notre méthode doit donc s'appeler "index". On ne lui passe aucun argument car la route n'a pas d'attributs et on ne va pas traiter de formulaire.

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Models\User;
class UsersController extends Controller
{
    public function index ()
    {
        // On récupère tous les utilisateurs
        // On les affiche
        $users = User::all();
        dd($users);
    }
}
```

Laravel simplifie bien les choses !

dd() est un autre helpueur de Laravel, C'est un var_dump amélioré.

En se connectant sur le site en tant qu'administrateur ⁴ et en allant à l'adresse <http://127.0.0.1:8000/users>, cela affiche :



Ce qui est retourné est une Collection.

Les collections sont un format de donnée offert par Laravel, une sorte de tableau amélioré, sur lequel on peut faire des opérations de traitement rapidement.

Dans cette collection on a la liste de tous les utilisateurs, sous forme de classe "App\Models\User".

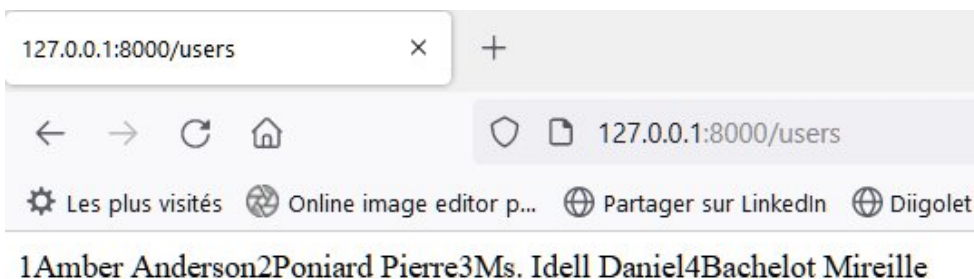
- Si l'on veut afficher tous les utilisateurs, il nous suffit d'itérer sur la collection (`foreach`)

```
<?php

namespace App\Http\Controllers;

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

class UsersController extends Controller
{
    public function index()
    {
        $users = User::all();
        foreach($users as $user) {
            echo $user->id;
            echo $user->name;
        }
    }
}
```



⁴ Je rappelle qu'on a mis le middleware Admin sur la route, si on se connecte en tant qu'utilisateur simple on ne pourra pas accéder à la page !

Afficher les données dans une vue

On retourne "view", comme pour les routes !

```
<?php

namespace App\Http\Controllers;

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

class UsersController extends Controller
{
    public function index()
    {
        $users = User::all();
        return view('users.index', compact('users'));
    }
}
```

`users.index` est le chemin vers la vue.

On va chercher dans le dossier des vues le dossier "users" et dans ce dossier le fichier "index.blade.php".

Il faut donc créer ce dossier "users" dans les vues et ce fichier "index.blade.php" que nous mettrons en forme un peu plus tard.

Les méthodes show, edit, update et destroy

```
public function show(User $user) {
}

public function edit(User $user) {
}

public function update(Request $request, User $user) {
}

public function destroy(User $user) {
}
```

Certaines méthodes prennent un paramètre comme `Request` ou `User`, mais dans notre routeur nous n'avons pas passé ces paramètres à nos contrôleurs...

En réalité, c'est Laravel qui va passer ces paramètres pour nous.

C'est ce qu'on appelle **l'injection de dépendances**.

`Request` sera accessible tout le temps mais on ne s'en sert en général que pour traiter les formulaires.

Ensuite, `User` est passé grâce à [l'implicite binding](#) que l'on a vu quand on faisait nos routes.

C'est grâce au paramètre dans nos routes comme celle-ci :

`Route::get('/users/profile/{user}', ...)` !

Les méthodes « show » et « edit ».

Le but est simplement de retourner la vue...

```
<?php

namespace App\Http\Controllers;

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

class UsersController extends Controller
{
    public function index()
    {
        $users = User::all();
        return view('users.index', compact('users'));
    }

    public function show(User $user) {
        return view('users.show', compact('user'));
    }

    public function edit(User $user) {
        return view('users.edit', compact('user'));
    }

    public function update(Request $request, User $user) {

    }

    public function destroy(User $user) {
    }
}
```

La méthode « destroy »

```
public function destroy(User $user) {
    $user->delete();
    return redirect()->back()->with('success', 'L\'utilisateur a bien été supprimé');
}
```

redirect() est un helper qui nous permet de rediriger vers une autre page. Ici on veut rediriger à la page précédente et on affiche un message flash nommé "success". C'est un message stocké en session qui s'effacera une fois qu'il aura été affiché une fois.

Delete() fait partie des méthodes offertes par Eloquent : [les méthodes Eloquent](#) ↗

La méthode `update()`

⚠ Il faut, avant d'enregistrer en base de données, vérifier les données du formulaire.

Heureusement, Laravel met à notre disposition dans la classe Request une méthode validate qui permet de valider les inputs selon des critères que l'on définit.

Les critères peuvent être : je veux que ce champ ait une valeur comprise entre 2 et 10 caractères, que ce soit un nombre, qu'il ne soit pas vide, que ce soit une adresse email...

Sans la méthode validate de la classe Request, il faudrait vérifier les inputs à la main avec une suite de if() et des fonctions comme mb_strlen() pour compter les caractères, empty(), filter_var() pour vérifier des formats etc.(ou s'appuyer sur HTML5 😊)

Laravel a déjà bon nombre de règles prédéfinies pour la méthode validate() que l'on retrouve ici : <https://laravel.com/docs/8.x/validation#available-validation-rules> 📄

Dans le cas du update, on va autoriser une personne à modifier son profil.

Elle peut modifier son "name" et son "email".

On va commencer par définir :

- name : entre 2 et 50 caractères, requis
- email : requis, format d'email

Ce qui donne :

```
public function update(Request $request, User $user) {
    // Afficher une valeur :
    dd($request->input('name')); // On considère que notre requête contient
    un champ "name"
    // On valide les données
    $validatedData = $request->validate([
        'name' => ['required', 'min:2', 'max:50'],
        'email' => 'required|email',
    ]);
}
```

Pour récupérer la valeur d'une donnée passée dans le formulaire : il nous suffit d'utiliser `$request->input()`

Pour valider les données avec Laravel :

1. soit avec un tableau,
2. soit en séparant les règles par un "pipe" (|).

Par exemple, nous avons : 'email' => 'required|email'.

Cela signifie que email est obligatoire (required) et qu'il doit ressembler à un mail (des caractères, puis le symbole "@", puis des caractères etc..

Dès qu'une règle ne sera pas respectée, Laravel générera tout seul une erreur et ne continuera pas l'exécution du code.

Il y aura une redirection vers la page précédente avec une donnée de session qui contiendra toutes les erreurs. On pourra alors les afficher.

```
$user->update($validatedData);
return redirect()->back()->with('success', 'Les informations ont bien été
modifiées');
```

Du coup, dans notre méthode update() on peut continuer d'écrire notre code en toute confiance car il ne sera exécuté que si toutes les validations d'input sont passées ! Les données validées sont ensuite stockées dans le tableau \$validatedData

❗ Un petit focus sur \$validatedData

Ce tableau ressemblera à :

```
$validatedData = ['name'=>$request->input('name'),
'email'=>$request->input('email'), 'submit'=>$request->input('submit'),
'csrf'=>$request->input('csrf')];
```

\$request contient toutes les données envoyées par notre formulaire.

En général, on a un input de type submit qui a le nom "submit", c'est donc normal qu'on le retrouve dans la \$request ! Vu qu'on a mis aucune règle de validation dessus (aucun intérêt de le faire), il est considéré comme une donnée validée ;).

Concernant CSRF, c'est un input caché généré par Laravel pour se protéger de la faille du même nom. Les failles CSRF (Cf. [la faille CSRF](#) ↗, ou encore [CSRF par l'OWASP](#) ↗) sont très courantes dans les sites web.

Laravel nous offre un moyen simple de nous en prémunir. Nous n'avons rien à faire, c'est lui qui fait tout ! Il est dans \$validatedData pour la même raison que "submit".

Enregistrement des modifications dans la base de données

Eloquent met à notre disposition la méthode update.

On va lui passer validatedData.

Il va tout faire pour nous ensuite : modifier les données name et email si une modification a eu lieu, et laisser les 2 autres champs de côté (grâce au \$fillable qu'on a rempli dans nos modèles)

La méthode complète

```
public function update(Request $request, User $user) {
    // Afficher une valeur :
    dd($request->input('name')); // On considère que notre requête contient
un champ "name"
    // On valide les données
    $validatedData = $request->validate([
        'name' => ['required', 'min:2', 'max:50'],
        'email' => 'required|email',
    ]);

    $user->update($validatedData);

    return redirect()->back()->with('success', 'Les informations ont bien
été modifiées');
}
```

Notre contrôleur final UsersController.php

```
<?php

namespace App\Http\Controllers;

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

class UsersController extends Controller
{
    public function index()
    {
        $users = User::all();
        return view('users.index', compact('users'));
    }

    public function show(User $user) {
        return view('users.show', compact('user'));
    }

    public function edit(User $user) {
        return view('users.edit', compact('user'));
    }

    public function update(Request $request, User $user) {
        // Afficher une valeur :
        dd($request->input('name')); // On considère que notre requête
        contient un champ "name"
        // On valide les données
        $validatedData = $request->validate([
            'name' => ['required', 'min:2', 'max:50'],
            'email' => 'required|email',
        ]);
        $user->update($validatedData);
        return redirect()->back()->with('success', 'Les informations ont
        bien été modifiées');
    }

    public function destroy(User $user) {
        $user->delete();
        return redirect()->back()->with('success', 'L\'utilisateur a bien
        été supprimé');
    }
}
```