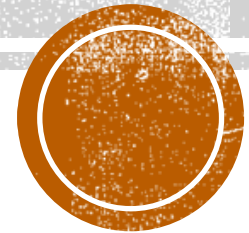




Module: Développer en back-end

10. L'authentification



Filière: Développement digital – option web full stack

Cours réalisé par:

- Asmae YOUALA (ISTA ADARISSA- Fès)
- Ouafae EL MAUDNI (ISTA BAB TIZIMI - Meknès)

PLAN DU COURS:

10. L'authentification

- Introduction
- Les middlewares auth et guest
- Authentification manuelle
- Les gardes
- Récapitulatif des méthodes de la façade Auth
- Authentification avec un kit de démarrage : Laravel Breeze

Introduction

- Laravel fournit un système d'authentification complet et facile à utiliser, qui permet aux développeurs de gérer les fonctionnalités d'authentification telles que :
 - **l'inscription**,
 - la **connexion**,
 - la **réinitialisation de mot de passe**,
 - la **vérification d'email**
 - la **déconnexion des utilisateurs**.
- Laravel inclut des services d'authentification et de session intégrés qui sont généralement accessibles via les façades **Auth** et **Session**;
- Ces fonctionnalités fournissent une authentification basée sur les **cookies** pour les requêtes en provenances du navigateurs Web.
- Ils fournissent des méthodes qui permettent de vérifier les informations d'identification d'un utilisateur et de l'authentifier.
- De plus, ces services stockeront automatiquement les données d'authentification appropriées dans la **session** de l'utilisateur et émettront le **cookie** de session de l'utilisateur.
- Pour mettre en place un système d'authentification, Laravel propose deux méthodes :
 - **L'authentification manuelle** (en utilisant les classes d'authentification offertes par Laravel)
 - L'installation d'un **kit de démarrage** (starter kit) : Outils qui créent et configurent automatiquement les fichiers nécessaires au système d'authentification à savoir: les contrôleurs, les vues, les routes ...
- Le système d'authentification de Laravel utilise **des middleware pour protéger** les routes de l'application

Les middlewares auth et guest

Le middleware auth:

- Le middleware **auth** permet de n'autoriser l'accès qu'aux utilisateurs authentifiés, donc de protéger des routes.
- Ce middleware est déjà présent et déclaré dans **app\Http\Kernel.php** :

```
'auth' => \App\Http\Middleware\Authenticate::class,
```

- On peut utiliser ce middleware directement sur une route

```
Route::get('comptes', function() {  
    // Réserve aux utilisateurs authentifiés  
})->middleware('auth');
```

- Quand un utilisateur n'arrive pas à s'authentifier, il est renvoyé vers la route nommée **login**:

```
protected function redirectTo(Request $request): ?string  
{  
    return $request->expectsJson() ? null : route('login');  
}
```

/*cette ligne de code vérifie si la requête est une requête JSON ou non. Si la requête est une requête JSON, elle ne renvoie rien (null). Sinon, elle redirige l'utilisateur vers la page de connexion en utilisant la route **login***/

Le middleware guest:

- Le middleware **guest** est exactement l'inverse du auth: il permet de n'autoriser l'accès qu'aux utilisateurs non authentifiés. Ce middleware est aussi déjà présent et déclaré dans **app\Http\Kernel.php** :

```
'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
```

Authentification manuelle

Voici les étapes de la mise en place d'un exemple d'authentification manuelle:

1. Modifier et exécuter le fichier de migration de la table "users" (fichier inclus par défaut dans les projets Laravel), en voici un extrait:

```
Schema::create('users', function (Blueprint $table) {  
    $table->id();  
    $table->string('nom'); //On personnalise le nom de la colonne "name"  
    $table->string('email')->unique();  
    $table->string('password');  
    $table->boolean('actif') ->default(true); //On ajoute une nouvelle colonne « actif »  
    $table->timestamp('email_verified_at')->nullable();  
    $table->rememberToken();  
    $table->timestamps();  
});
```

2. Adapter le modèle User correspondant :

```
class User extends Model  
{  
    protected $fillable = [  
        'nom',  
        'email',  
        'password',  
    ];  
}
```

NOTEZ : Il est recommandé de ne pas personnaliser le nom de la colonne "password", car Laravel l'utilise en dur dans son code. Sinon on devra ajouter d'autres configurations.

Authentification manuelle

4. Ajouter un contrôleur **InscriptionController** pour gérer l'inscription d'un nouvel utilisateur :

Ajouter à ce contrôleur deux méthodes :

- **show** permettant d'afficher une vue « **register** » où on affiche un formulaire d'inscription (à créer sous le répertoire views/auth)
- **register** permettant d'ajouter le nouvel utilisateur à la table “**users**”

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;

class InscriptionController extends Controller
{
    public function show(){
        return view ("auth.register");
    }

    public function register(Request $request){
        $request->validate([
            'nom' => 'required|string|max:255',
            'email' =>
                'required|string|email|max:255|unique:users',
            'password' =>
                'required|string|min:8|confirmed',
        ]);

        $user = User::create([
            'nom' => $request->nom,
            'email' => $request->email,
            'password' => Hash::make($request->password),
        ]);

        auth()->login($user);

        return redirect()->route('home');
    }
}
```

La règle **confirmed** permet de vérifier automatiquement que le champ password et le champ password_confirmation sont identiques. Laravel ajoute simplement _confirmation au nom du champ.

Authentification manuelle

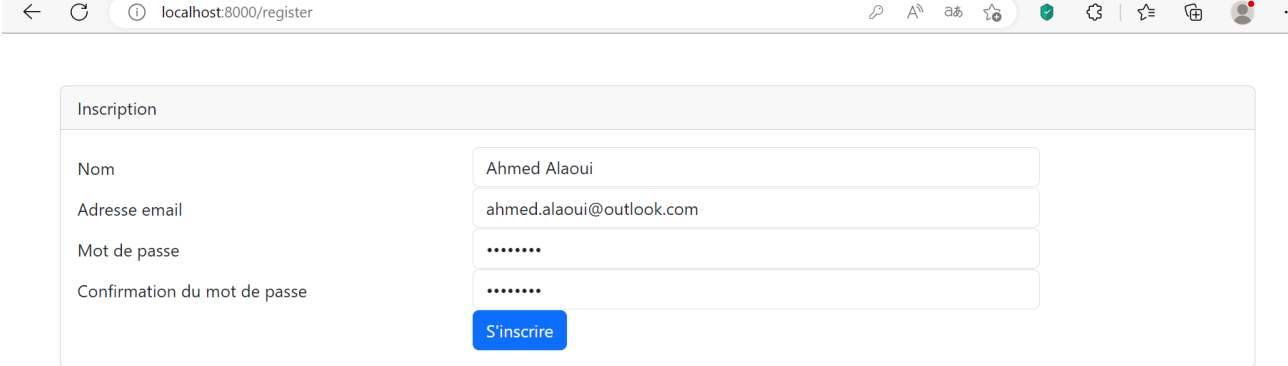
5. Configurer les routes suivantes:

```
Route::get('/', function () {  
    return view('welcome');  
})->name("home");
```

```
Route::middleware("guest")->group(function(){  
    //Les routes d'inscription  
    Route::get("/register", [InscriptionController::class, "show"]);  
    Route::post("/register", [InscriptionController::class, "register"])  
        ->name("register");  
});
```

- Afin de ne pas autoriser aux utilisateurs authentifiés une réinscription, on a protégé les deux routes d'inscription par le middleware 'guest'
- Pour terminer la configuration de ce dernier, il faut modifier le fichier **RouteServiceProvider**, sous le répertoire **app/Providers** => Attribuer à la constante **HOME** la route à ouvrir au cas où on refuse d'accéder à la route protégée.

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



The screenshot shows a web browser window with the address bar displaying 'localhost:8000/register'. The page title is 'Inscription'. The form has four input fields: 'Nom' with the value 'Ahmed Alaoui', 'Adresse email' with the value 'ahmed.alaoui@outlook.com', 'Mot de passe' with masked characters '.....', and 'Confirmation du mot de passe' with masked characters '.....'. A blue button labeled 'S'inscrire' is positioned below the confirmation field.

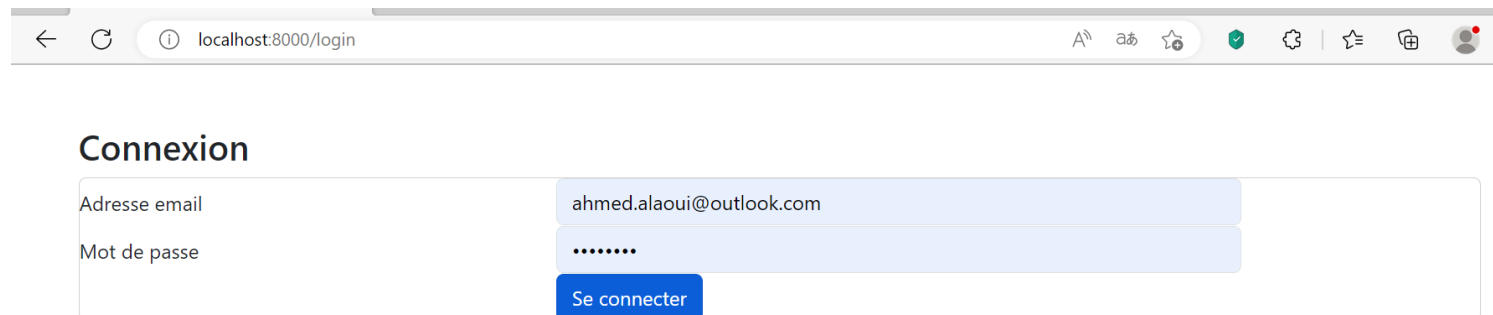
Authentification manuelle

6. Ajouter un contrôleur **ConnexionController** pour gérer la connexion d'un utilisateur :

Ajouter à ce contrôleur deux méthodes :

- **show** permettant d'afficher une vue « **login** » où on affiche un formulaire de connexion (à créer sous le répertoire **views/auth**)

```
namespace App\Http\Controllers;  
use Illuminate\Http\Request;  
use Illuminate\Support\Facades\Auth;  
  
class ConnexionController extends Controller  
{  
    public function show(){  
        return view("auth.login");  
    }  
}
```



The screenshot shows a web browser window with the address bar displaying 'localhost:8000/login'. The page content includes a title 'Connexion' and a form with two input fields: 'Adresse email' containing 'ahmed.alaoui@outlook.com' and 'Mot de passe' containing masked characters '.....'. A blue button labeled 'Se connecter' is positioned below the password field.

Authentication manuelle

- **login** permettant de vérifier les coordonnées de l'utilisateur et de l'authentifier.

```
public function login(Request $request){  
  
    if(Auth::attempt(["email"=>$request->email, "password"=>$request->password, $aktif=>1])){  
        $request->session()->regenerate();  
        return redirect()->intended();  
    }  
  
    return back()->withErrors([  
        'email' => 'Adresse email invalide!',  
    ])->onlyInput('email');  
}
```

- La méthode **attempt** accepte un tableau de paires clé/valeur comme premier argument. Les valeurs du tableau seront utilisées pour trouver l'utilisateur dans la table « users » de la base de données;
- La méthode **attempt** retourne **true** si l'authentification a réussi. Sinon, **false** sera retourné;
- Le deuxième argument que peut prendre la méthode **attempt** est un booléen indiquant si la fonctionnalité « remember me » sera activée ou non;
- La méthode **intended** fournie par le redirecteur de Laravel redirigera l'utilisateur vers l'URL à partir duquel il tentait d'accéder avant d'être intercepté par le middleware d'authentification. Un URI de secours peut être donné à cette méthode (en paramètre) au cas où la destination prévue n'est pas disponible.

Authentication manuelle

6. Configurer les routes de connexion:

```
Route::middleware("guest")->group(function(){
    //Les routes d'inscription
    //.....
    //Les routes de connexion
    Route::get("/login", [ConnexionController::class, "show"]);
    Route::post("/login", [ConnexionController::class, "login"])
        ->name("login");
});
```

Authentification manuelle

7. Ajouter le contrôleur **DeconnexionController** et y insérer la méthode **logout**:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class DeconnexionController extends Controller
{
    public function logout(Request $request){
        Auth::logout();
        $request->session()->invalidate();
        $request->session()->regenerateToken();
        return redirect('/');
    }
}
```

La méthode **logout** fournie par la façade **Auth** supprimera les informations d'authentification de la session de l'utilisateur afin que les prochaines requêtes ne soient pas authentifiées.

En plus d'appeler la méthode **logout**, il est **recommandé d'invalidier** la session de l'utilisateur et de **régénérer** son jeton CSRF .

8. Configurer la route de déconnexion:

```
Route::middleware("auth")->group(function(){
    Route::post("/logout", [DeconnexionController::class, "logout"])
        ->name("logout");
});
```

Authentification manuelle

8. Sous la vue principale, on peut ajouter ce bout de code, affichant :

- Pour les utilisateurs connectés : le nom et un lien permettant la déconnexion
- Pour les utilisateurs non authentifiés, deux liens : l'un pour la connexion et l'autre pour l'inscription

@auth

```
{{auth()->user()->nom}}  
    <form action="{{ route('logout') }}" method="post">  
        @csrf  
        <button type="submit">Se déconnecter</button>  
    </form>
```

@endauth

@guest

```
<a href="{{ route('login') }}" >Se connecter</a>  
    <a href="{{ route('register') }}"> S'inscrire</a>
```

@endguest

Les directives blade **@auth** et **@guest** peuvent être utilisées pour déterminer rapidement si l'utilisateur actuel est authentifié ou s'il est un invité.

Les gardes (guards)

- Les **gardes** Laravel définissent comment les utilisateurs sont authentifiés pour chaque demande (request).
- Laravel est livré avec des gardes pour l'authentification, mais nous pouvons également en créer d'autres.
- Ces configurations peuvent être consultées et éditées dans le fichier **auth.php** sous le répertoire **config**:

Voici un extrait de son contenu par défaut:

```
'guards' => [  
    'web' => [  
        'driver' => 'session',  
        'provider' => 'users',  
    ],  
,  
  
    'providers' => [  
        'users' => [  
            'driver' => 'eloquent',  
            'model' => App\Models\User::class,  
        ],  
    ],  
,
```

- Laravel est livré avec une garde appelée « **web** » qui maintient l'état en utilisant le stockage de **session** et les **cookies**.
- La garde « web » est configuré pour exploiter le fournisseur (provider) **users** ;
- La configuration du fournisseur « users » indique l'utilisation du pilote « Eloquent » et du modèle **App\Models\User**;
=> On peut ajouter un autre fournisseur en utilisant un autre pilote (ou autre méthode de connexion) et/ou une autre table de base de données ou Modèle;
=> Le nouveau fournisseur peut être exploité pour en créer une nouvelle garde

Récapitulatif des méthodes de la façade Auth:

- Voici une liste non exhaustive des méthodes parmi les plus utilisées de la façade `Illuminate\Support\Facades\Auth` :

Notez: On peut également appeler ces méthodes en utilisant le helper `auth()`

Méthode	Fonction	Exemple
<code>Auth::user()</code>	Récupère une instance de l'utilisateur actuellement authentifié	<code>{{Auth::user()->nom}}</code> ou <code>{{auth()->user()->nom}}</code>
<code>Auth::id()</code>	Récupère l'identifiant de l'utilisateur actuellement authentifié	
<code>Auth::check()</code>	Détermine si l'utilisateur effectuant la requête HTTP entrante est authentifié ou non	<pre>if (Auth::check()) { // utilisateur authentifié }</pre>
<code>Auth::attempt(\$credentials)</code>	Vérifie et authentifie l'utilisateur si les données sont correctes	<pre>if(Auth::attempt(["email"=>\$request->email, "password"=>\$request->password]){ // authentification réussie }</pre>

Récapitulatif des méthodes de la façade Auth:

Méthode	Fonction
<code>Auth::guard('admin')->attempt(\$credentials)</code>	Le nom du garde passé à la méthode guard doit correspondre à l'un des gardes configurés dans le fichier de configuration auth.php (on peut y définir une autre méthode de connexion et/ou une autre table auquel on veut se connecter)
<code>Auth::login(\$user);</code>	Sert à authentifier l'utilisateur passé en paramètre. L'instance d'utilisateur donnée doit être une implémentation du contrat <i>Illuminate\Contracts\Auth\Authenticatable</i>
<code>Auth::guard('admin')->login(\$user);</code>	Sert à authentifier l'utilisateur passé en paramètre en utilisant la garde passée en paramètre (ici: « admin »)
<code>Auth::viaRemember()</code>	Détermine si l'utilisateur actuellement authentifié a été authentifié à l'aide du cookie "se souvenir de moi" :
<code>Auth::loginUsingId(1);</code>	Authentifie l'utilisateur en utilisant son id
<code>Auth::logout();</code>	Supprime les informations d'authentification de la session de l'utilisateur afin que les prochaines demandes ne soient pas authentifiées.

Authentification avec un kit de démarrage : Laravel Breeze

Introduction

- Laravel offre un ensemble de kits de démarrage (**starter kit**) d'authentification prêts à l'emploi.
- Ces kits de démarrage sont **facile à mettre en œuvre** grâce aux commandes d'artisan intégrées qui créent **automatiquement** les fichiers nécessaires pour l'authentification, tels que les contrôleurs, les vues, les modèles et les routes.
- Les développeurs peuvent **personnaliser** ces fichiers selon leurs besoins pour adapter le système d'authentification à leur application.
- Laravel offre plusieurs choix pour les kits de démarrage d'authentification. Voici quelques exemples :
 - **Breeze** est un package qui implémente les fonctionnalités d'authentification de Laravel. C'est une solution simple et rapide pour l'authentification
 - **Fortify** est conçu pour être facilement intégré à des applications existantes et offre des fonctionnalités de sécurité avancées. Laravel Fortify est plus minimaliste, il se contente de nous donner la logique backend de l'authentification mais aucune vue, route etc..
 - **Jetstream** est plus avancé, c'est un pack très complet intégrant déjà un **design** travaillé basé sur des technologies puissantes et très récentes (Inertia + VueJS, Livewire + Blade).

Laravel Breeze : Installation

- Laravel **Breeze** fournit les mécanismes nécessaires pour gérer automatiquement :

- **L'inscription**
- **La connexion**
- **La déconnexion**
- **L'oubli de mot de passe**

Voici les étapes à suivre pour mettre en place l'authentification avec Breeze

1. Installer Breeze en utilisant Composer:

```
composer require laravel/breeze --dev
```

NB: (au cas d'erreur exécuter : `composer update -w && composer dumpautoload`)

2. Exécuter les commandes suivantes pour installer les dépendances de Breeze:

```
php artisan breeze:install
```

```
php artisan migrate
```

```
npm install
```

```
npm run dev
```

- Avec Breeze, on peut choisir entre blade ou react ou vue ou api:

Lors de l'installation, on pourra avoir des questions

où on exprimera nos choix :

Pour notre cours, on utilisera **blade** :

```
$ php artisan breeze:install

Which stack would you like to install?
blade ..... 0
react ..... 1
vue ..... 2
api ..... 3
> 0

Would you like to install dark mode support? (yes/no) [no]
> no

Would you prefer Pest tests instead of PHPUnit? (yes/no) [no]
> no

INFO Installing and building Node dependencies.
```

Laravel Breeze : Installation

L'exécution de la commande : `php artisan breeze:install` permettra d'ajouter les fichiers nécessaires à son fonctionnement à savoir: les routes, les vues, les contrôleurs...

```
GET|HEAD confirm-password ..... password.confirm > Auth\ConfirmablePasswordController@show
POST confirm-password ..... Auth\ConfirmablePasswordController@store
GET|HEAD dashboard ..... dashboard
POST email/verification-notification ..... verification.send > Auth>EmailVerificationNotificationController@store
GET|HEAD forgot-password ..... password.request > Auth>PasswordResetLinkController@create
POST forgot-password ..... password.email > Auth>PasswordResetLinkController@store
GET|HEAD login ..... login > Auth\AuthenticatedSessionController@create
POST login ..... Auth\AuthenticatedSessionController@store
POST logout ..... logout > Auth\AuthenticatedSessionController@destroy
PUT password ..... password.update > Auth>PasswordController@update
GET|HEAD profile ..... profile.edit > ProfileController@edit
PATCH profile ..... profile.update > ProfileController@update
DELETE profile ..... profile.destroy > ProfileController@destroy
GET|HEAD register ..... register > Auth\RegisteredUserController@create
POST register ..... Auth\RegisteredUserController@store
POST reset-password ..... password.store > Auth\NewPasswordController@store
GET|HEAD reset-password/{token} ..... password.reset > Auth\NewPasswordController@create
GET|HEAD sanctum/csrf-cookie ..... sanctum.csrf-cookie > Laravel\Sanctum > CsrfCookieController@show
GET|HEAD verify-email ..... verification.notice > Auth>EmailVerificationPromptController
GET|HEAD verify-email/{id}/{hash} ..... verification.verify > Auth\VerifyEmailController
```

```
Auth
AuthenticatedSessionController.php
ConfirmablePasswordController.php
EmailVerificationNotificationController.php
EmailVerificationPromptController.php
NewPasswordController.php
PasswordController.php
PasswordResetLinkController.php
RegisteredUserController.php
VerifyEmailController.php
Controller.php
ProfileController.php
```

```
views
auth
confirm-password.blade.php
forgot-password.blade.php
login.blade.php
register.blade.php
reset-password.blade.php
verify-email.blade.php
components
layouts
app.blade.php
guest.blade.php
navigation.blade.php
profile
partials
delete-user-form.blade.php
update-password-form.blade.php
update-profile-information-form.blade.php
edit.blade.php
dashboard.blade.php
welcome.blade.php
```

3. Exécuter les migrations pour créer la table des utilisateurs (users)

4. Configurer l'application pour utiliser l'authentification par défaut de Laravel en modifiant le fichier **config/auth.php** pour utiliser le modèle **App\Models\User**:

```
'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => App\Models\User::class,
    ],
],
```

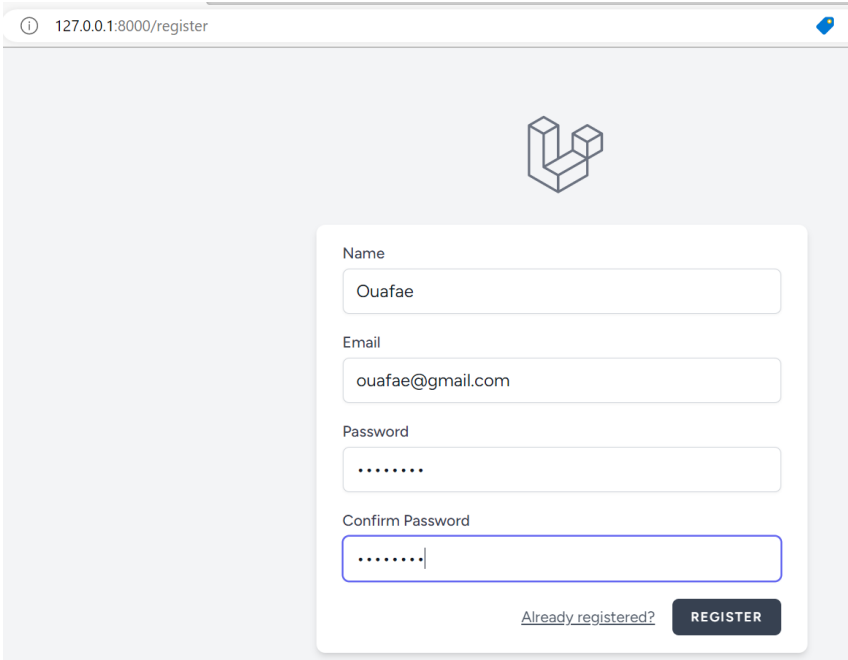
Laravel Breeze: Inscription d'un utilisateur:

- L'enregistrement d'un utilisateur passe par l'url **.../register**

```
GET|HEAD  register ..... register > Auth\RegisteredUserController@create  
POST      register ..... Auth\RegisteredUserController@store
```

- le contrôleur **RegisteredUserController** contient une méthode **create** permettant de renvoyer une vue :

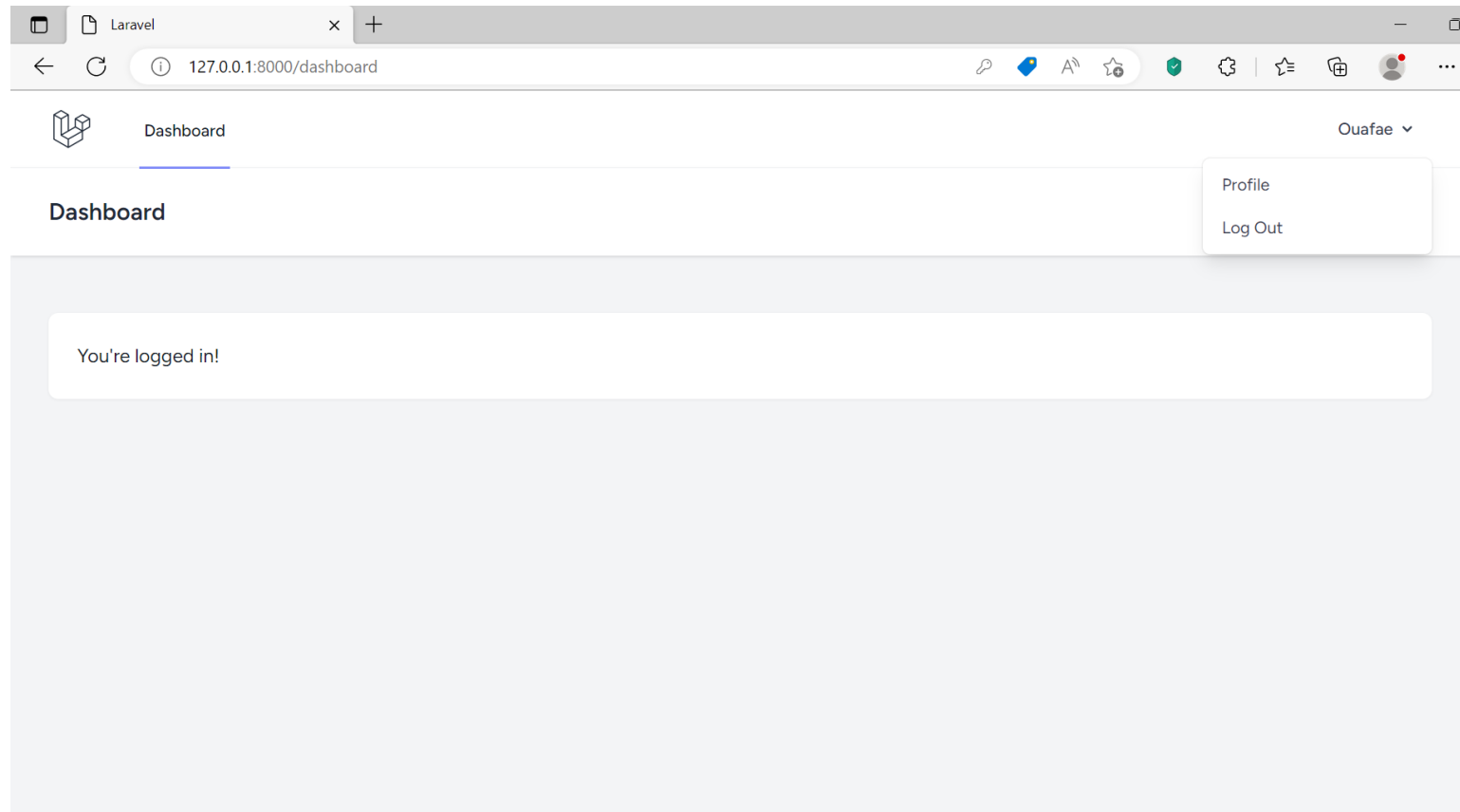
```
public function create(): View  
{  
    return view('auth.register');  
}
```



- **La validation:** La validation de l'enregistrement est présente dans la méthode **store** de ce contrôleur. Ainsi, on peut ajouter ou modifier des règles de validation ou même des colonnes à la table User.

Laravel Breeze: Inscription d'un utilisateur:

- Une fois que l'utilisateur est créé dans la base il est automatiquement connecté et il est redirigé vers une page « **dashboard** ».
- On peut y repérer le nom de l'utilisateur connecté, avec la possibilité d'accéder au profil ou se déconnecter.

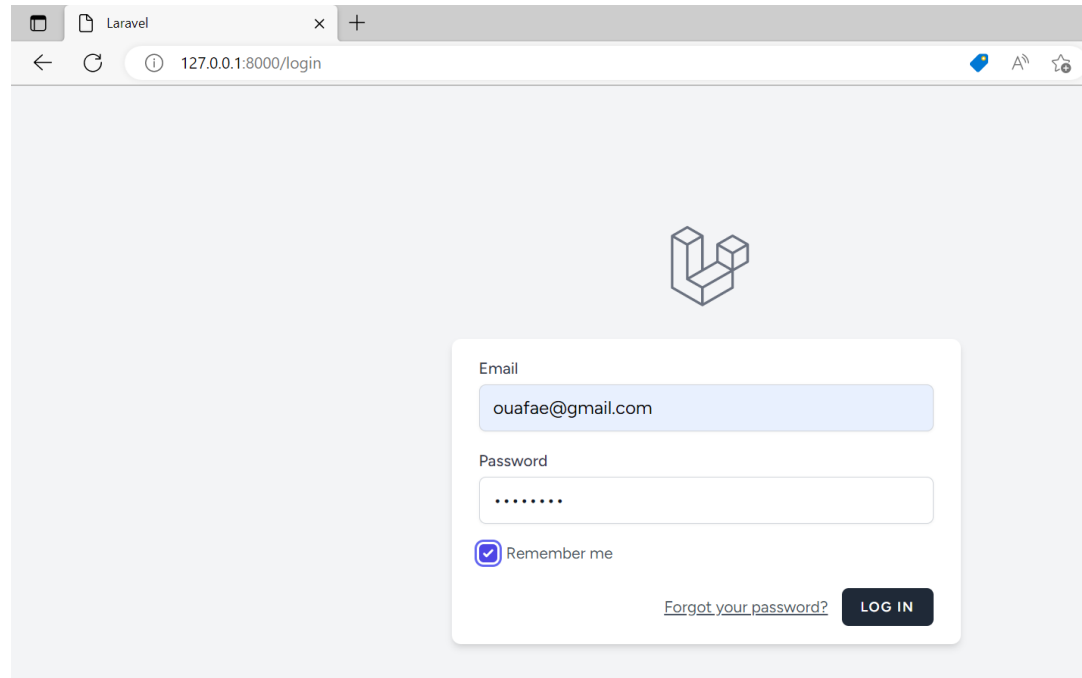


Laravel Breeze: La connexion d'un utilisateur

- La route `/login`

GET|HEAD `login` `login` > `Auth\AuthenticatedSessionController@create`

POST `login` `Auth\AuthenticatedSessionController@store`



- Le contrôleur **AuthenticatedSessionController** dispose d'une méthode appelée "**create**" qui renvoie simplement une vue.
- Dans le formulaire de connexion, il y a une option "se rappeler de moi" ou "remember me" :
 - Si cette option est cochée, l'utilisateur reste connecté de manière permanente jusqu'à ce qu'il se déconnecte manuellement.
 - Pour que cela fonctionne, une colonne "remember_token" doit être présente dans la table "users".

Laravel Breeze: La déconnexion d'un utilisateur

- La route **/logout** est une route qui est utilisée pour déconnecter l'utilisateur actuellement authentifié. Cette route est généralement utilisée avec une méthode HTTP POST

POST logout logout > Auth\AuthenticatedSessionController@destroy

- Pour utiliser la route `/logout` dans votre application Laravel, vous pouvez ajouter une balise form à votre vue qui utilise la méthode HTTP POST et l'URL `/logout`. Voici un exemple de code que vous pouvez utiliser :

```
<form id="logout-form" action="{{ route('logout') }}" method="POST">
    @csrf
    <button type="submit" class="btn btn-primary">Déconnexion</button>
</form>
```

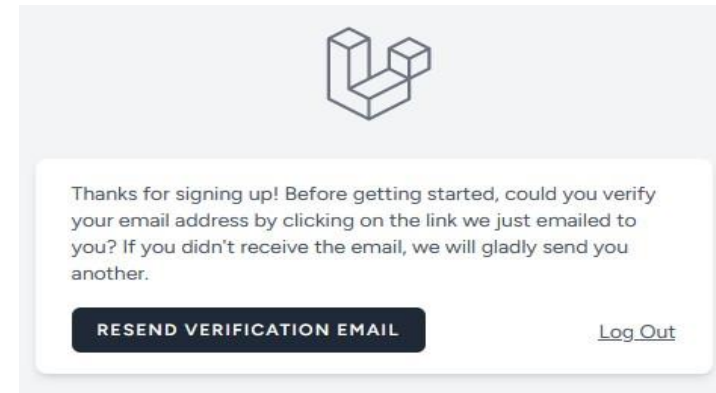
- Lorsque l'utilisateur soumet ce formulaire, Laravel utilise la route `/logout` pour déconnecter l'utilisateur actuellement authentifié. Vous pouvez personnaliser le comportement de cette route en modifiant la méthode **destroy** du contrôleur **AuthenticatedSessionController**

Laravel Breeze: La vérification de l'email

- Pour s'assurer qu'un utilisateur a bien confirmé son email, il est nécessaire que la classe modèle **User** implémente l'interface **MustVerifyEmail**.

```
use Illuminate\Contracts\Auth\MustVerifyEmail;  
  
...  
class User extends Authenticatable implements MustVerifyEmail{  
    // ...  
}
```

- Après l'enregistrement , l'utilisateur reçoit ce message:



- Lorsqu'il utilise le lien de validation, l'email est bien vérifié et il est dirigé vers la page de connexion.
- Pour protéger les routes qui ne peuvent pas être accessibles avant la vérification de l'adresse e-mail de l'utilisateur, on utilise le middleware "**verified**". Par exemple:

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

Laravel Breeze: La vérification de l'email – Configuration d'un service de messagerie

- Avant d'envoyer un email, il faut tout d'abord commencer par configurer un service de messagerie, tel que : SMTP, Amazon SES ...
- Les services de messagerie peuvent être local ou cloud;
- Ces services peuvent être configurés via le fichier ***config/mail.php*** du projet Laravel;
- Pour tester l'envoi des emails dans un environnement de développement (sans réellement les envoyer), on peut les enregistrer dans le journal (log) de notre projet ou utiliser un service de messagerie permettant de simuler localement l'envoi des emails à des fausses adresses;
- Il existe plusieurs outils permettant d'installer et de tester localement un serveur e-mail à savoir: HELO / Mailtrap / Mailpit / **MailHog** ;
- Dans ce cours, on va utiliser MailHog;

Configuration de MailHog

- MailHog permet de consulter les emails sortants et de l'examiner sans réellement envoyer le courrier à ses destinataires.
- Pour commencer, mettez à jour le fichier **.env** de votre application pour utiliser les paramètres de messagerie suivants :

```
MAIL_MAILER=smtp
MAIL_HOST=localhost
MAIL_PORT=1025
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
MAIL_FROM_ADDRESS="admin@example.com"
MAIL_FROM_NAME="${APP_NAME}"
```

- Téléchargez et exécutez le dernier release de MailHog depuis : <https://github.com/mailhog/MailHog>

Laravel Breeze: La vérification de l'email – Configuration d'un service de messagerie

Une fois MailHog configuré, vous pouvez accéder au tableau de bord MailHog à l'adresse **http://localhost:8025** et consulter les emails envoyés depuis votre projet;

The screenshot displays the MailHog web interface. At the top, there's a header with the MailHog logo, a search bar, and a GitHub link. Below the header, a navigation bar contains icons for back, delete, download, and share. The main content area shows an email from 'Laravel <admin@example.com>' with the subject 'Verify Email Address' and recipient 'ouafae@gmail.com'. The email body is displayed in HTML format, featuring a 'Hello!' greeting, a request to verify the email address, a 'Verify Email Address' button, and a note that no further action is required if the user did not create an account. The email ends with 'Regards,'. On the left sidebar, there's a 'Connected' status, an 'Inbox (1)' section, and a 'Delete all messages' button. Below this, a 'Jim' card is visible, which includes a description of Jim as a chaos monkey, a link to find out more at GitHub, and an 'Enable Jim' button. At the bottom left, the URL 'localhost:8025/api/v1/messages/.../download' is shown.

MailHog

Search

GitHub

← 🗑️ ⬇️ ➦

Connected

Inbox (1)

⊗ Delete all messages

From: Laravel <admin@example.com>

Subject: **Verify Email Address**

To: ouafae@gmail.com

Show headers ▼

HTML Plain text Source MIME

Jim

Jim is a chaos monkey.
[Find out more at GitHub.](#)

Enable Jim

Hello!

Please click the button below to verify your email address.

Verify Email Address

If you did not create an account, no further action is required.

Regards,

localhost:8025/api/v1/messages/.../download