Cours Laravel 6 – les bases – formulaires et middlewares

laravel.sillo.org/cours-laravel-6-les-bases-formulaires-et-middlewares/

bestmomo

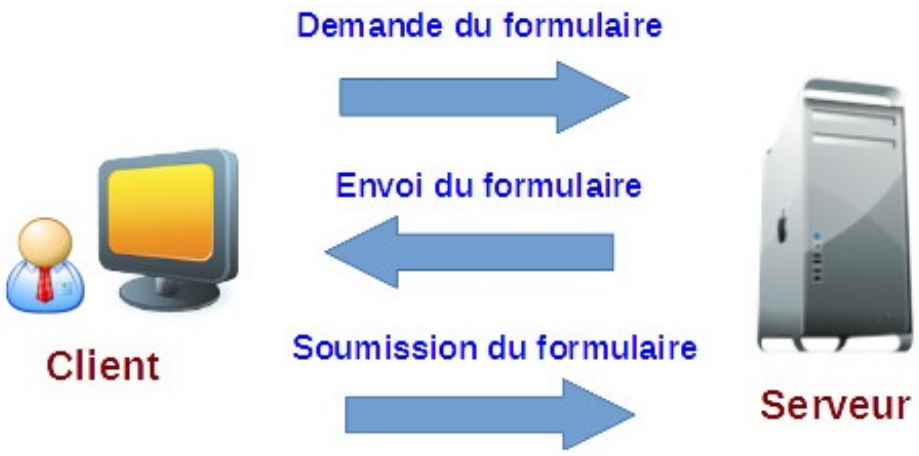
August 28, 2019

Dans bien des circonstances, le client envoie des informations au serveur. La situation la plus générale est celle d'un formulaire. Nous allons voir dans ce chapitre comment créer facilement un formulaire avec Laravel, comment réceptionner les entrées et nous améliorerons notre compréhension du routage.

Nous verrons aussi l'importante notion de middleware.

Scénario et routes

Nous allons envisager un petit scénario avec une demande de formulaire de la part du client, sa soumission et son traitement :



On va donc avoir besoin de deux routes :

- une pour la demande du formulaire avec une méthode **get**,
- une pour la soumission du formulaire avec une méthode **post**.

On va donc créer ces deux routes dans le fichier **routes/web.php** :

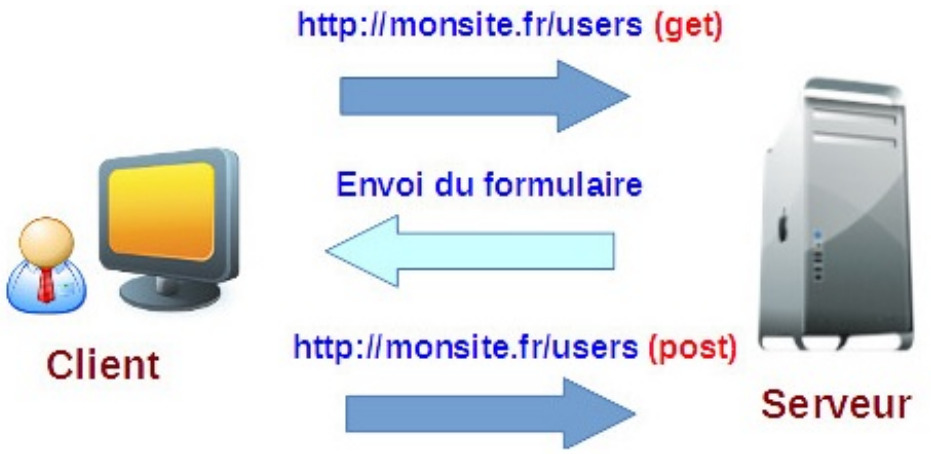
```
Route::get('users', 'UsersController@create');
Route::post('users', 'UsersController@store');
```

Jusque-là on avait vu seulement des routes avec le verbe **get**, on a maintenant aussi une route avec le verbe **post**.

Les urls correspondantes sont donc :

- **http://monsite.fr/users** avec la méthode **get**,
- **http://monsite.fr/users** avec la méthode **post**.

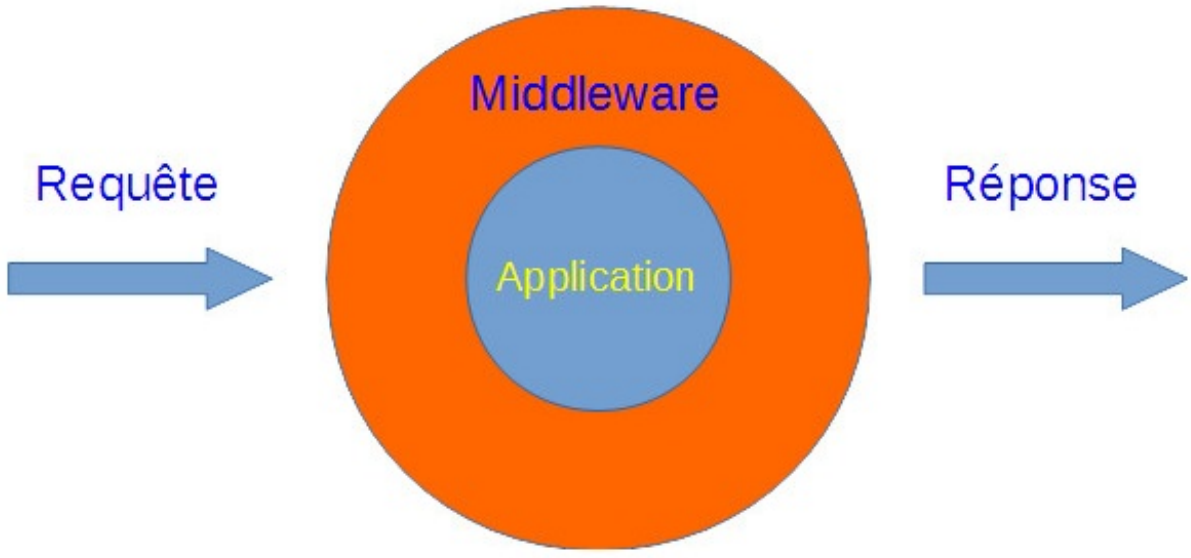
Donc on a la même url, seul le verbe diffère. Voici le scénario schématisé avec les urls :



Les middlewares

Les middlewares sont chargés de filtrer les requêtes HTTP qui arrivent dans l'application, ainsi que celles qui en partent (beaucoup moins utilisé). Le cas le plus classique est celui qui concerne la vérification de l'authentification d'un utilisateur pour qu'il puisse accéder à certaines ressources. On peut aussi utiliser un middleware par exemple pour démarrer la gestion des sessions.

Voici un schéma pour illustrer cela :



On peut avoir en fait plusieurs middlewares en pelures d'oignon, chacun effectue son traitement et transmet la requête ou la réponse au suivant.

Donc dès qu'il y a un traitement à faire à l'arrivée des requêtes (ou à leur départ) un middleware est tout indiqué.

Laravel peut servir comme application « web » ou comme « api ». Dans le premier cas on a besoin :

- de gérer les cookies,
- de gérer une session,
- de gérer la protection CSRF (dont je parle plus loin dans ce chapitre)…

Si vous regardez dans le fichier **app/Http/Kernel.php** :

```
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        // \Illuminate\Session\Middleware\AuthenticateSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],

    'api' => [
        'throttle:60,1',
        'bindings',
    ],
];
```

On trouve les deux middlewares de groupes (ils rassemblent plusieurs middlewares) « web » et « api ». On voit que dans le premier cas on active bien les cookies, les sessions et la vérification CSRF.

Par défaut toutes les routes que vous entrez dans le fichier **routes/web.php** sont incluses dans le groupe « web ». Si vous regardez dans le provider **app/Providers/RouteServiceProvider.php** vous trouvez cette inclusion :

```
protected function mapWebRoutes()
{
    Route::middleware('web')
         ->namespace($this->namespace)
         ->group(base_path('routes/web.php'));
}
```

Le formulaire

Pour faire les choses correctement nous allons prévoir un template **resources/views/template.blade.php** :

```
<!doctype html>
<html lang="fr">
<head>
   <meta charset="UTF-8">
</head>
<body>
   @yield('contenu')
</body>
</html>
```

Et une vue **resources/views/infos.blade.php** qui utilise ce template :

```
@extends('template')

@section('contenu')
   <form action="{{ url('users') }}" method="POST">
      @csrf
      <label for="nom">Entrez votre nom : </label>
      <input type="text" name="nom" id="nom">
      <input type="submit" value="Envoyer !">
   </form>
@endsection
```

Je vous parle plus loin de la signification de la syntaxe **@csrf**.

*L'helper **url** est utilisé pour générer l'url complète pour l'action du formulaire. Pour une route nommée on pourrait utiliser l'helper **route**.*

Le résultat sera un formulaire sans fioriture :

Laravel Collective ?

Pour simplifier l'écriture du code des vues certains utilisent la librairie Html de Laravel Collective.

Cette librairie faisait à l'origine partie de Laravel dans sa version 4. A partir de la version 5 elle a été retirée du projet et récupérée en tant que librairie indépendante. Comme j'étais habitué à ce composant je l'ai naturellement intégré à mes projets et conseillé dans mes cours.

Le temps passant je me suis rendu compte personnellement que je préfère coder mes vues avec la syntaxe classique plutôt que de la masquer avec les méthodes de ce composant malgré les avantages en terme de concision du code et de rapidité de développement. C'est un choix personnel.

Si vous voulez utiliser cette librairie il vous suffit de suivre la procédure d'installation détaillée ici et ensuite de consulter la documentation qui suit et qui est très bien faite. Par exemple pour notre formulaire on se retrouve avec cette syntaxe :

```
@extends('template')

@section('contenu')
   {!! Form::open(['url' => 'users']) !!}
      {!! Form::label('nom', 'Entrez votre nom : ') !!}
      {!! Form::text('nom') !!}
      {!! Form::submit('Envoyer !') !!}
   {!! Form::close() !!}
@endsection
```

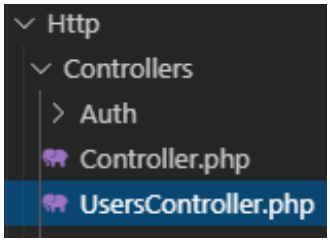
Libre à vous d'adapter les formulaires de ce cours avec cette librairie si elle vous convient mais je n'y ferai plus référence dans les chapitres suivants au profit d'une syntaxe classique.

Le contrôleur

Il ne nous manque plus que le contrôleur pour faire fonctionner tout ça. Utilisez Artisan pour générer un contrôleur :

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

Vous devez le retrouver ici :



Modifiez ensuite son code ainsi :

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UsersController extends Controller
{
   public function create()
   {
      return view('infos');
   }

   public function store(Request $request)
   {
      return 'Le nom est ' . $request->input('nom');
   }
}
```

Le contrôleur possède deux méthodes :

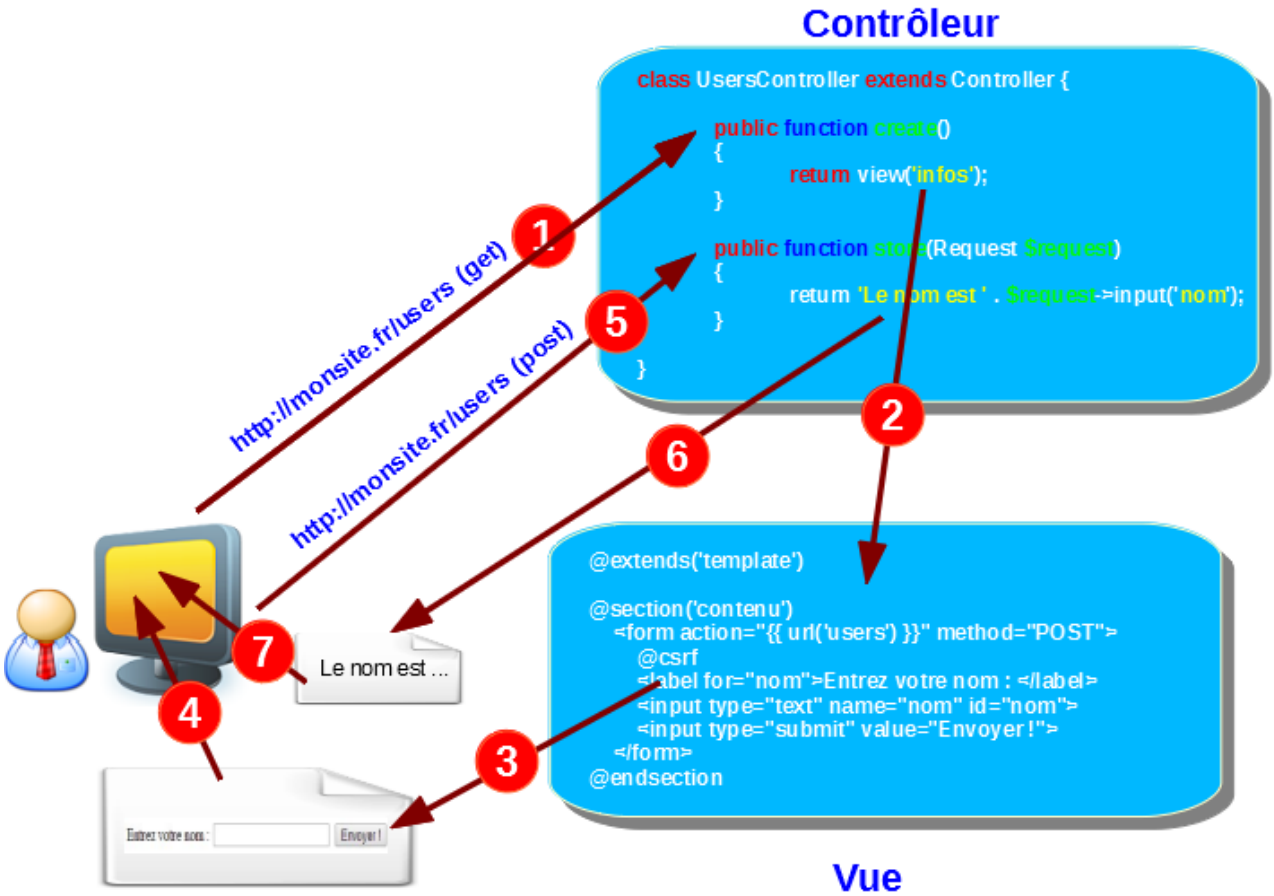
- la méthode **create** qui reçoit l'url **http://monsite.fr/users** avec le verbe **get** et qui retourne le formulaire,

- la méthode **store** qui reçoit l'url **http://monsite.fr/users** avec le verbe **post** et qui traite les entrées.

Pour la première méthode il n'y a rien de nouveau et je vous renvoie aux chapitres précédents si quelque chose ne vous paraît pas clair. Par contre nous allons nous intéresser à la seconde méthode.

Dans cette seconde méthode on veut récupérer l'entrée du client. Encore une fois la syntaxe est limpide : on veut dans la requête (**request**) les entrées (**input**) récupérer celle qui s'appelle **nom**.

Si vous faites fonctionner tout ça vous devez au final obtenir l'affichage du nom saisi. Voici une schématisation du fonctionnement qui exclue les routes pour simplifier :



(1) le client envoie la requête de demande du formulaire qui est transmise au contrôleur par la route (non représentée sur le schéma),
(2) le contrôleur crée la vue « infos »,
(3) la vue « infos » crée le formulaire,
(4) le formulaire est envoyé au client,
(5) le client soumet le formulaire, le contrôleur reçoit la requête de soumission par l'intermédiaire de la route (non représentée sur le schéma),
(6) le contrôleur génère la réponse,
(7) la réponse est envoyée au client.

La protection CSRF

On a vu que le formulaire généré par Laravel comporte une ligne un peu particulière :

```
@csrf
```

Si on regarde le code généré on trouve quelque chose dans ce genre :

```
<input type="hidden" name="_token" value="iIjW9PMNsV6VKT2sIc16ShoTf6SdYVZolVUGsxDI">
```

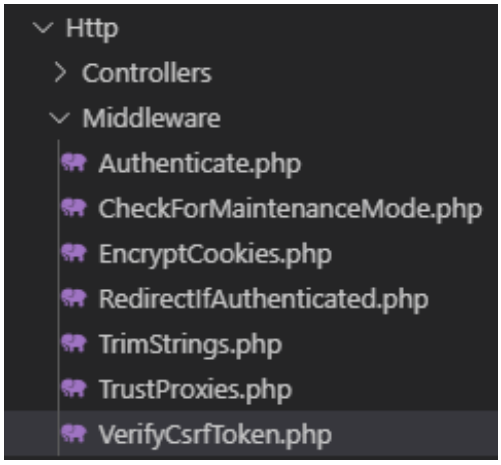
A quoi cela sert-il ?

Tout d'abord **CSRF** signifie **Cross-Site Request Forgery**. C'est une attaque qui consiste à faire envoyer par un client une requête à son insu. Cette attaque est relativement simple à mettre en place et consiste à envoyer à un client authentifié sur un site un script dissimulé (dans une page web ou un email) pour lui faire accomplir une action à son insu.

Pour se prémunir contre ce genre d'attaque Laravel génère une valeur aléatoire (**token**) associée au formulaire de telle sorte qu'à la soumission cette valeur est vérifiée pour être sûr de l'origine.

Vous vous demandez peut-être où se trouve ce middleware CSRF ?

Il est bien rangé dans le dossier **app/Http/Middleware** :

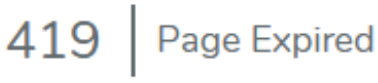


Pour tester l'efficacité de cette vérification essayez un envoi de formulaire sans le token en modifiant ainsi la vue :

```
@extends('template')

@section('contenu')
   <form action="{{ url('users') }}" method="POST">
      <label for="nom">Entrez votre nom : </label>
      <input type="text" name="nom" id="nom">
      <input type="submit" value="Envoyer !">
   </form>
@endsection
```

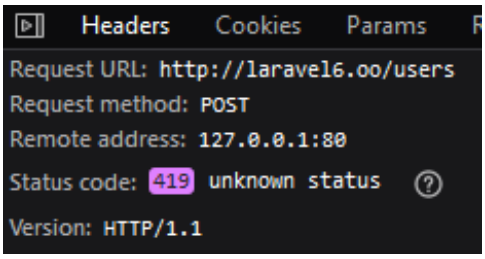
Vous tombez sur cette page à la soumission :

Comme le token n'est pas bon Laravel en conclut qu'il a expiré parce qu'évidemment il a une durée de validité limitée liée à la session.



Analysons un peu mieux cette réponse :

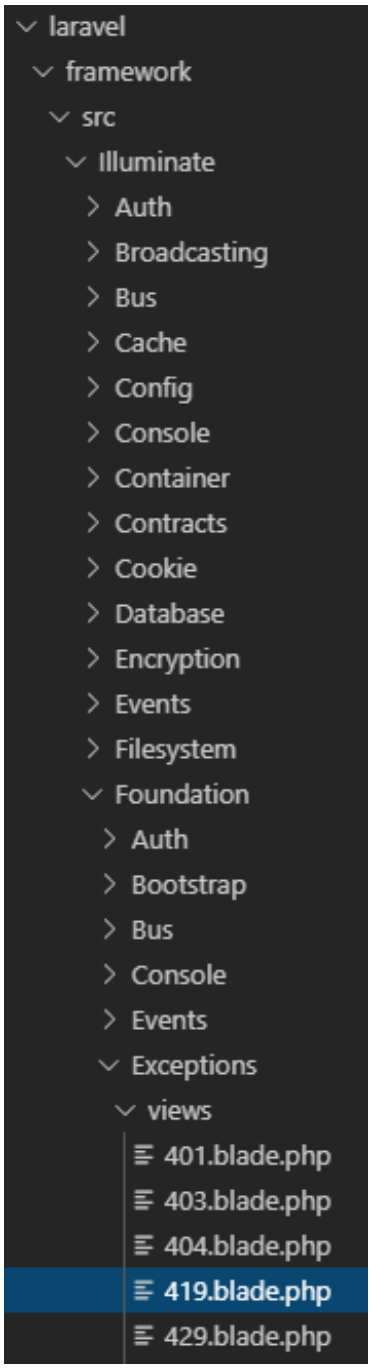
On voit une réponse 419 qui ne fait pas encore partie du standard HTTP mais qui est de plus en plus utilisée comme alternative à 401.



Page d'erreur personnalisée

La page d'erreur sur laquelle on est tombée n'est pas très explicite et en anglais. On pourrait la vouloir en français, ou tout simplement en changer le style. Mais où se trouve le code de cette page ? Regardez dans le dossier **vendor** :

Vous trouver un dossier des vues par défaut de Laravel pour les erreurs les plus classiques. Vous n'allez évidemment pas modifier directement ces vues dans le dossier **vendor** ! Mais vous pouvez surcharger ces vues en créant un dossier **resources/views/errors** et en copiant le fichier concerné :

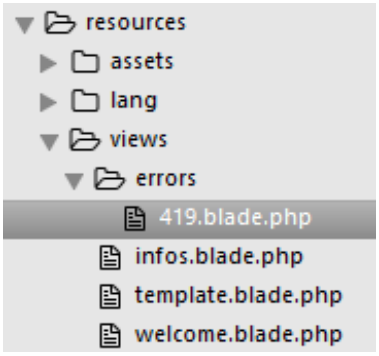


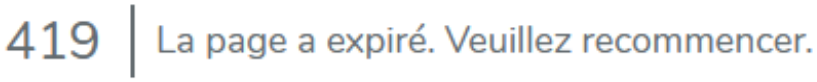
On a alors de base ce code :

```
@extends('errors::minimal')

@section('title', __('Page Expired'))
@section('code', '419')
@section('message', __('Page Expired'))
```

Adaptez le code selon vos goûts. Je n'ai pas encore parlé des possibilités de Laravel au niveau des langues. La syntaxe **__(...)** permet de faire en sorte que le même code servent pour plusieurs langues. Mais si vous n'avez que le français alors vous pouvez directement changer :

```
@extends('errors::minimal')

@section('title', 'Page expirée')
@section('code', '419')
@section('message', 'La page a expiré. Veuillez recommencer.')
```





Vous pouvez intervenir ainsi pour tous les code d'erreur.

En résumé

- Laravel permet de créer des routes avec différents verbes : get, post…
- Un middleware permet de filtrer les requêtes.
- Les entrées du client sont récupérées dans la requête.
- On peut se prémunir contre les attaques CSRF.
- On peut personnaliser les pages d'erreur par défaut de Laravel et même en créer de nouvelles.