

[S'inscrire](#)[Se connecter](#)[Accueil](#) ▶ [Cours](#) ▶ Découvrez le framework PHP Laravel

Découvrez le framework PHP Laravel

15 heures Moyenne



[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Ce cours est une mise à jour du [cours sur Laravel 5.1.](#)

Vous trouverez aussi des modifications dans les exercices et un nouveau chapitre !

Vous pratiquez PHP et vous savez créer des sites ? Vous avez l'impression de réécrire souvent les mêmes choses ? Vous posez des questions sur la meilleure façon de traiter une tâche particulière comme créer des formulaires ou envoyer des e-mails ? Vous aimerez disposer d'une boîte à outils toute prête pour tout le code laborieux ?

Alors vous avez besoin d'un *framework* PHP, et Laravel constitue actuellement ce qui se fait de mieux en la matière !

Laravel colle aux plus récentes avancées de PHP et surtout à son approche objet. Découvrir ce framework et plonger dans son code, c'est prendre un cours de programmation et d'esthétique. C'est aussi disposer d'une boîte à outils simple et performante pour construire des applications web sans se soucier de l'intendance habituelle.

Ce cours nécessite d'avoir des connaissances correctes en PHP, HTML, CSS et Javascript ! Si vous avez des lacunes dans ce domaine, vous pouvez facilement les combler avec les nombreux cours à disposition sur ce site !

Ce cours progressif a été conçu en 3 parties :

- La première partie est particulièrement détaillée, elle est destinée à vous habituer au framework, à sa philosophie et à vous présenter les notions essentielles.
- La seconde partie est axée sur les bases de données qui constituent la clé des applications dynamiques. Les autres notions seront évidemment développées et complétées au cours de cette partie qui vous demandera plus d'attention et d'expérimentation.
- La troisième partie confortera vos connaissances, vous montrera quelques spécificités comme la manière

d'obtenir des vues épurées, l'utilisation d'ajax, la localisation, et présentera la façon de déployer une application Laravel selon le serveur de destination. Le code sera plus fourni et moins détaillé, il vous faudra l'analyser et le mettre en pratique pour profiter pleinement du cours.

Laravel évolue rapidement et il est possible que vous constatiez dans certains chapitres des différences avec le code que vous constatez dans votre installation. Merci de me communiquer vos constatations pour que je tienne ce cours le plus à jour possible.

Commencer le cours

[Comment ça marche ?](#)



Partie 1 - Les bases de Laravel



- 1. Présentation générale
- 2. Installation et organisation
- 3. Le routage et les façades
- 4. Les réponses
- 5. Les contrôleurs
- 6. Les entrées
- 7. La validation
- 8. Configuration et session
- 9. L'injection de dépendance

Quiz : Quiz 1

Activité : Créer un site de sondages



Partie 2 - Les bases de données



- 1. Migrations et modèles
- 2. Les ressources (1/2)
- 3. Les ressources (2/2) et les erreurs
- 4. L'authentification
- 5. La relation 1:n
- 6. La relation n:n
- 7. Les commandes et les assistants
- 8. Query Builder

Quiz : Quiz 2

Activité : Construisez un site de sondages avec une base de données



Partie 3 - Plus loin avec Laravel



1. Déploiement
 2. Des vues propres (1/2)
 3. Des vues propres (2/2)
 4. La localisation
 5. Ajax
 6. Les tests unitaires
 7. Événements et autorisations
- Quiz : Quiz 3
- Activité : Perfectionnez votre site de sondages



Certificat de réussite (voir un exemple)

L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

Découvrez le framework PHP Laravel



15 heures

Moyenne

Licence 

Présentation générale

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Dans ce premier chapitre je vais évoquer PHP, son historique rapide et sa situation actuelle. Je vais aussi expliquer l'intérêt d'utiliser un framework pour ce langage et surtout pourquoi j'ai choisi Laravel. J'évoquerai enfin le patron MVC et la Programmation Orientée Objet.

Un framework ?

Approche personnelle

PHP est un langage populaire et accessible. Il est facile à installer et présent chez tous les hébergeurs. C'est un langage riche et plutôt facile à aborder, surtout pour quelqu'un qui a déjà des bases en programmation. On peut réaliser rapidement une application web fonctionnelle grâce à lui. Mais le revers de cette simplicité est que bien souvent le code créé est confus, complexe, sans aucune cohérence. Il faut reconnaître que PHP n'encourage pas à organiser son code et rien n'oblige à le faire.

Lorsqu'on crée des applications PHP on finit par avoir des routines personnelles toutes prêtes pour les fonctionnalités récurrentes, par exemple pour gérer des pages de façon dynamique. Une fois qu'on a créé une fonction ou une classe pour réaliser une tâche il est naturel d'aller la chercher lorsque la même situation se présente. Puisque c'est une bibliothèque personnelle et qu'on est seul maître à bord il faut évidemment la mettre à jour lorsque c'est nécessaire, et c'est parfois fastidieux.

En général on a aussi une hiérarchie de dossiers à laquelle on est habitué et on la reproduit quand on commence le développement d'une nouvelle application. On se rend compte des fois que cette habitude a des effets pervers parce que la hiérarchie qu'on met ainsi en place de façon systématique n'est pas forcément la plus adaptée.

En résumé l'approche personnelle est plutôt du bricolage à la hauteur de ses compétences et de sa disponibilité.

(Re)découvrir PHP

Lorsque j'ai découvert PHP à la fin du dernier millénaire (ça fait plus impressionnant dit comme ça :D) il en était à la version 3. C'était essentiellement un langage de script en général mélangé au HTML qui permettait de faire du templating, des accès aux données et du traitement. La version 4 en 2000 a apporté plus de stabilité et une ébauche de l'approche objet. Mais il a fallu attendre la version 5 en 2004 pour disposer d'un langage de programmation à la hauteur du standard existant pour les autres langages.

Cette évolution incite à perdre les mauvaises habitudes si on en avait. Un site comme <http://www.phptherightway.com> offre de bonnes pistes pour mettre en place de bonnes pratiques.

Les bases de Laravel

- ▶ 1. Présentation générale
- 2. Installation et organisation
- 3. Le routage et les façades
- 4. Les réponses
- 5. Les contrôleurs
- 6. Les entrées
- 7. La validation
- 8. Configuration et session
- 9. L'injection de dépendance
- Quiz : Quiz 1
- Activité : Créer un site de sondages

[Accéder au forum](#)



Donc si vous êtes un bidouilleur de code PHP je vous conseille cette saine lecture qui devrait vous offrir un nouvel éclairage sur ce langage et surtout vous permettre de vous lancer de façon correcte dans le code de Laravel.

Un framework

D'après Wikipedia un framework informatique est un "ensemble cohérent de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel". Autrement dit une base cohérente avec des briques toutes prêtes à disposition. Il existe des frameworks pour tous les langages de programmation et en particulier pour PHP. En faire la liste serait laborieux tant il en existe !

L'utilité d'un framework est d'éviter de passer du temps à développer ce qui a déjà été fait par d'autres souvent plus compétents et qui a en plus été utilisé et validé par de nombreux utilisateurs. On peut imaginer un framework comme un ensemble d'outils à disposition. Par exemple je dois faire du routage pour mon site, je prend un composant déjà tout prêt et qui a fait ses preuves et je l'utilise : gain de temps, fiabilité, mise à jour si nécessaire...

Il serait vraiment dommage de se passer d'un framework alors que le fait d'en utiliser un présente pratiquement uniquement des avantages.

Pourquoi Laravel ?

Constitution de Laravel

Laravel, créé par Taylor Otwell, initie une nouvelle façon de concevoir un framework en utilisant ce qui existe de mieux pour chaque fonctionnalité. Par exemple toute application web a besoin d'un système qui gère les requêtes HTTP. Plutôt que de réinventer quelque chose, le concepteur de Laravel a tout simplement utilisé celui de **Symfony** en l'étendant pour créer un système de routage efficace. De la même manière, l'envoi des emails se fait avec la bibliothèque **SwiftMailer**. En quelque sorte Otwell a fait son marché parmi toutes les bibliothèques disponibles. Nous verrons dans ce cours comment cela est réalisé. Mais Laravel ce n'est pas seulement le regroupement de bibliothèques existantes, c'est aussi de nombreux composants originaux et surtout une orchestration de tout ça.

Vous allez trouver dans Laravel :

- un système de routage perfectionné (RESTful et ressources),
- un créateur de requêtes SQL et un ORM performants,
- un moteur de template efficace,
- un système d'authentification pour les connexions,
- un système de validation,
- un système de pagination,
- un système de migration pour les bases de données,
- un système d'envoi d'emails,
- un système de cache,
- un système d'événements,
- un système d'autorisations,
- une gestion des sessions...

Et bien d'autres choses encore que nous allons découvrir ensemble. Il est probable que certains éléments de cette liste ne vous évoquent pas grand-chose, mais ce n'est pas important pour le moment, tout cela deviendra plus clair au fil des chapitres.

Le meilleur de PHP

Plonger dans le code de Laravel, c'est recevoir un cours de programmation tant le style est clair et élégant et le code merveilleusement organisé. La version actuelle de Laravel est la 5.2, elle nécessite au minimum la version 5.5.9 de PHP. Pour aborder de façon efficace ce framework, il serait souhaitable que vous soyez familiarisé avec ces notions :

les espaces de noms : c'est une façon de bien ranger le code pour éviter des conflits de nommage. Laravel utilise cette possibilité de façon intensive. Tous les composants sont rangés dans des espaces de noms distincts, de même que l'application créée.

- **les fonctions anonymes** : ce sont des fonctions sans nom (souvent appelées closures) qui permettent d'améliorer le code. Les utilisateurs de Javascript y sont habitués. Les utilisateurs de PHP un peu moins parce qu'elle y sont plus récentes. Laravel les utilise aussi de façon systématique.
- **les méthodes magiques** : ce sont des méthodes qui n'ont pas été explicitement décrites dans une classe mais qui peuvent être appelées et résolues.
- **les interfaces** : une interface est un contrat de constitution des classes. En programmation objet c'est le sommet de la hiérarchie. Tous les composants de Laravel sont fondés sur des interfaces. La version 5 a même vu apparaître un lot de contrats pour étendre de façon sereine le framework.
- **les traits** : c'est une façon d'ajouter des propriétés et méthodes à une classe sans passer par l'héritage, ce qui permet de passer outre certaines limitation de l'héritage simple proposé par défaut par PHP.

Un framework n'est pas fait pour remplacer la connaissance d'un langage mais pour assister celui (ou celle) qui connaît déjà bien ce langage. Si vous avez des lacunes il vaut mieux les combler pour profiter pleinement de Laravel.

La documentation

Quand on s'intéresse à un framework il ne suffit pas qu'il soit riche et performant, il faut aussi que la documentation soit à la hauteur. C'est le cas pour Laravel. Vous trouverez la documentation [sur le site officiel](#). Mais il existe de plus en plus de sources d'informations dont voici les principales :

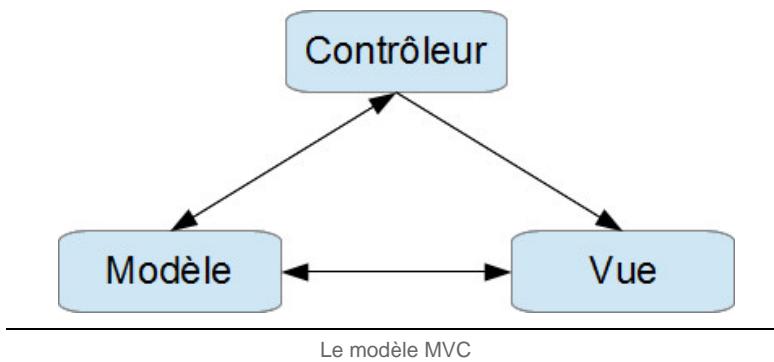
- <http://laravel.fr> : site d'entraide francophone avec un forum actif.
- <http://laravel.io> : le forum officiel.
- <http://laravel.sillo.org> : mon blog créé début 2013 et toujours actif qui constitue une initiation progressive complémentaire du présent cours.
- <http://cheats.jesse-obrien.ca> : une page bien pratique qui résume toutes les commandes.
- <http://www.laravel-tricks.com> : un autre site d'astuces.
- <http://packalyst.com> : le rassemblement de tous les packages pour ajouter des fonctionnalités à Laravel.
- <https://laracasts.com> : de nombreux tutoriels vidéo en Anglais dont un certain nombre en accès gratuit dont [une série complète pour Laravel 5](#).

Il existe aussi de bon livres mais tous en anglais.

MVC ? POO ?

MVC

On peut difficilement parler d'un framework sans évoquer le patron **Modèle-Vue-Contrôleur**. Pour certains il s'agit de la clé de voûte de toute application rigoureuse, pour d'autres c'est une contrainte qui empêche d'organiser judicieusement son code. De quoi s'agit-il ? Voici un petit schéma pour y voir clair :



C'est un modèle d'organisation du code :

- le modèle est chargé de gérer les données
- la vue est chargée de la mise en forme pour l'utilisateur
- le contrôleur est chargé de gérer l'ensemble

En général on résume en disant que le modèle gère la base de données, la vue produit les pages HTML et le contrôleur fait tout le reste. Dans Laravel :

- le modèle correspond à une table d'une base de données. C'est une classe qui étend la classe `Model` qui permet une gestion simple et efficace des manipulations de données et l'établissement automatisé de relations entre tables.
- le contrôleur se décline en deux catégories : contrôleur classique et contrôleur de ressource (je détaillerai évidemment tout ça dans le cours).
- la vue est soit un simple fichier avec du code HTML, soit un fichier utilisant le système de template `Blade` de Laravel.

Laravel propose ce modèle mais ne l'impose pas. Nous verrons d'ailleurs qu'il est parfois judicieux de s'en éloigner parce qu'il y a des tas de chose qu'on arrive pas à caser dans ce modèle. Par exemple si je dois envoyer des emails où vais-je placer mon code ? En général ce qui se produit est l'inflation des contrôleurs auxquels on demande des choses pour lesquelles ils ne sont pas faits.

POO

Laravel est fondamentalement orienté objet. La POO est un design pattern qui s'éloigne radicalement de la programmation procédurale. Avec la POO tout le code est placé dans des classes qui découlent d'interfaces qui établissent des contrats de fonctionnement. Avec la POO on manipule des objets.

Avec la POO, la responsabilité du fonctionnement est répartie dans des classes, alors que dans l'approche procédurale tout est mélangé. Le fait de répartir la responsabilité évite la duplication du code qui est le lot presque forcé de la programmation procédurale. Laravel pousse au maximum cette répartition en utilisant l'injection de dépendance.

L'utilisation de classes bien identifiées, dont chacune a un rôle précis, pilotées par des interfaces claires, dopées par l'injection de dépendances : tout cela crée un code élégant, efficace, lisible, facile à maintenir et à tester. C'est ce que Laravel propose. Alors vous pouvez évidemment greffer là dessus votre code approximatif, mais vous pouvez aussi vous inspirer des sources du framework pour améliorer votre style de programmation.



L'injection de dépendance est destinée à éviter de rendre les classes dépendantes et de privilégier une liaison dynamique plutôt que statique. Le résultat est un code plus lisible, plus facile à maintenir et à tester. Nous verrons ce mécanisme à l'œuvre dans Laravel.

En résumé

- Un framework fait gagner du temps et donne l'assurance de disposer de composants bien codés et fiables.
- Laravel est un framework novateur, complet, qui utilise les possibilités les plus récentes de PHP

et qui est impeccamment codé et organisé.

- La documentation de Laravel est complète, précise et de plus en plus de tutoriels et exemples apparaissent sur la toile.
- Laravel adopte le patron MVC mais ne l'impose pas, il est totalement orienté objet.



Découvrez le framework PHP Laravel

Installation et organisation



L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

Découvrez le framework PHP Laravel



15 heures

Moyenne

Licence



Installation et organisation

Connectez-vous ou [inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Dans ce chapitre nous allons faire connaissance avec le gestionnaire de dépendances Composer. Nous allons voir également comment créer une application Laravel. Nous verrons enfin comment le code est organisé dans cette application.

Pour utiliser Laravel et suivre ce chapitre et l'ensemble du cours vous aurez besoin d'un serveur équipé de PHP avec au minimum la version 5.5.9 et aussi de MySQL. Il existe plusieurs applications "tout-en-un" faciles à installer : [wampserver](#), [xampp](#), [easyphp](#)... Personnellement j'utilise wamp qui répond sans problème à toutes mes attentes et qui permet de basculer entre les versions de PHP et de MySQL en un simple clic.

Il existe aussi une solution toute prête, [Homestead](#), facile à mettre en oeuvre sous Linux, mais beaucoup moins conviviale avec Windows. Pour ce dernier il existe une autre possibilité bien pensée : [Laragon](#).

Quelle que soit l'application que vous utilisez, vérifiez que vous avez la bonne version de PHP (minimum 5.5.9). D'autre part les extensions PDO, Tokenizer, OpenSSL et Mbstring de PHP doivent être activées.

Composer

Présentation

Je vous ai dit que Laravel utilise des composants d'autres sources. Plutôt que de les incorporer directement, il utilise un gestionnaire de dépendance : **Composer**. D'ailleurs pour le coup les composants de Laravel sont aussi traités comme des dépendances. Mais c'est quoi un gestionnaire de dépendance ?

Imaginez que vous créez une application PHP et que vous utilisez des composants issus de différentes sources : Carbon pour les dates, Redis pour les données... Vous pouvez utiliser la méthode laborieuse en allant chercher tout ça de façon manuelle, et vous allez être confronté à des difficultés :

- télécharger tous les composants dont vous avez besoin et les placer dans votre structure de dossier,
- traquer les éventuels conflits de nommage entre les librairies,
- mettre à jour manuellement les librairies quand c'est nécessaire,
- prévoir le code pour charger les classes à utiliser...

Tout ça est évidemment faisable mais avouez que s'il était possible d'automatiser les procédures ce serait vraiment génial. C'est justement ce que fait un gestionnaire de dépendances !



1. Présentation générale
 - ▶ **2. Installation et organisation**
 3. Le routage et les façades
 4. Les réponses
 5. Les contrôleurs
 6. Les entrées
 7. La validation
 8. Configuration et session
 9. L'injection de dépendance
- Quiz : Quiz 1
 Activité : Créer un site de sondages

[Accéder au forum](#)



Installation

Laravel utilise Composer comme gestionnaire de dépendances. Il vous faut donc commencer par l'installer sur votre ordinateur. Selon votre système la procédure est différente, je vous renvoie donc [au site](#) pour obtenir tous les renseignements sur le sujet.

Pour l'installation dans Windows il suffit de [télécharger un setup](#) qui fait tout très proprement et renseigne aussi la variable d'environnement PATH, ce qui permet ensuite d'utiliser Composer à partir de n'importe quel emplacement. Par contre l'installateur vous demandera où se trouve `php.exe` et vous devrez répondre. En effet Composer est un fichier PHP et a besoin d'être exécuté.

Pour les autres systèmes, en particulier Linux, le plus simple est d'utiliser `curl`. Il suffit de suivre [les instructions détaillées sur le site](#).

 Pour aller plus loin avec Composer vous pouvez lire [cet article](#).

Fonctionnement

Pour comprendre le fonctionnement de Composer, il faut connaître le format `JSON` qui est l'acronyme de JavaScript Object Notation. Un fichier `JSON` a pour but de contenir des informations de type étiquette-valeur. Regardez cet exemple élémentaire :

```
{
  "nom": "Durand",
  "prénom": "Jean"
}
```

Les étiquettes sont "nom" et "prénom" et les valeurs correspondantes "Durand" et "Jean". Les valeurs peuvent être aussi des tableaux ou des objets. Regardez ce second exemple :

```
{
  "identité1" : {
    "nom": "Durand",
    "prénom": "Jean"
  },
  "identité2" : {
    "nom": "Dupont",
    "prénom": "Albert"
  }
}
```

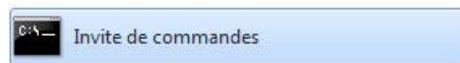
Composer a besoin d'un fichier `composer.json` associé. Ce fichier contient les instructions pour Composer : les dépendances, les classes à charger automatiquement... Voici un extrait de ce fichier pour Laravel :

```
{
  "name": "laravel/laravel",
  "description": "The Laravel Framework.",
  "keywords": ["framework", "laravel"],
  "license": "MIT",
  "type": "project",
  "require": {
    "php": ">=5.5.9",
    "laravel/framework": "5.2.*"
  },
  ...
}
```

Créer une application Laravel

Prérequis

Composer fonctionne en ligne de commande. Vous avez donc besoin de la console (nommée Terminal ou Konsole sur OS X et Linux). Les utilisateurs de Linux sont très certainement habitués à l'utilisation de la console mais il en est généralement pas de même pour les adeptes de Windows. Pour trouver la console sur ce système il faut chercher l'invite de commande :



Trouver la console dans Windows

Ce cours a été créé avec la version 5.2.* de Laravel. Lorsque vous créez une nouvelle application, que ce soit avec **composer create-project** ou avec l'installateur, vous obtenez la dernière version stable. Je m'efforcerai de garder ce cours en phase avec la dernière version de Laravel mais il y aura toujours un délai entre la sortie d'une nouvelle version et ce cours. Si vous rencontrez des différences de fonctionnement avec les exemples utilisés vous pouvez toujours, en attendant la mise à niveau du cours, installer la version précédente de Laravel. Il suffit d'utiliser la commande **create-project** en spécifiant la version comme troisième argument ([documentation complète ici](#)).

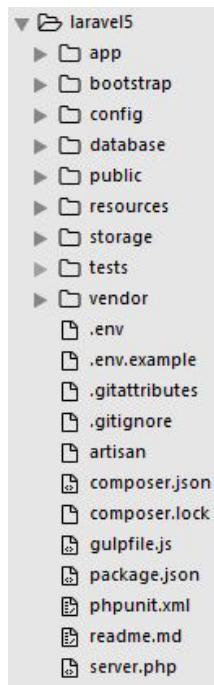
Installation avec Composer

Il y a plusieurs façons de créer une application Laravel. Celle qui me semble la plus simple consiste à utiliser la commande `create-project` de Composer. Par exemple je veux créer une application dans un dossier `laravel5` à la racine de mon serveur, voici la syntaxe à utiliser :

text

```
composer create-project --prefer-dist laravel/laravel laravel5
```

L'installation démarre et je n'ai plus qu'à attendre quelques minutes pour que Composer fasse son travail jusqu'au bout. Vous verrez s'afficher une liste de téléchargements. Au final on se retrouve avec cette architecture :



Architecture des dossiers de Laravel

On peut vérifier que tout fonctionne bien avec l'URL <http://localhost/laravel5/public>. Normalement on doit obtenir cette page très épurée :

Laravel|5

Page d'accueil de Laravel

Sous Windows avec Wamp il est possible d'avoir un souci pour afficher la page d'accueil. Si c'est votre cas il y a 3 solutions :

- n'utilisez pas Wamp mais par exemple [Laragon](#),
- créez un hôte virtuel,
- suivez ce qui est préconisé sur [cette page](#).

Pour les mises à jour ultérieures il suffit encore d'utiliser Composer avec la commande update :

```
composer update
```

text

Installation avec Laravel Installer

Une autre solution pour installer Laravel consiste à utiliser l'installateur. Il faut commencer par installer globalement l'installateur avec Composer :

```
composer global require "laravel/installer"
```

text

Il faut ensuite informer la variable d'environnement `path` de l'emplacement du dossier `.../composer/vendor/bin`.

Pour créer une application il suffit de taper :

```
laravel new monAppli
```

text

Laravel sera alors installé dans le dossier `monAppli`.

Si vous installez Laravel en téléchargeant directement les fichiers sur Github et en utilisant la commande `composer install` il vous faut effectuer deux actions complémentaires. En effet dans ce cas il ne sera pas automatiquement créé de clé de sécurité et vous allez tomber sur une erreur au lancement. Il faut donc la créer avec la commande `php artisan key:generate`. D'autre part vous aurez à la racine le fichier `.env.example` que vous devrez renommer en `.env` pour que la configuration fonctionne.

Autorisations

Au niveau des dossiers de Laravel, le seul qui a besoin de droits d'écriture par le serveur est `storage`.

Serveur

Pour fonctionner correctement, Laravel a besoin de PHP :

- Version >= 5.5.9
- Extension PDO
- Extension Mbstring
- Extension OpenSSL
- Extension Tokenizer

Des URL propres

Pour un serveur Apache il est prévu dans le dossier `public` un fichier `.htaccess` avec ce code :

```
<IfModule mod_rewrite.c>
  <IfModule mod_negotiation.c>
    Options -MultiViews
  </IfModule>

  RewriteEngine On

  # Redirect Trailing Slashes If Not A Folder...
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule ^(.*)/$ /$1 [L,R=301]

  # Handle Front Controller...

```

text

```
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]

# Handle Authorization Header
RewriteCond %{HTTP:Authorization} .
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
</IfModule>
```

Le but est essentiellement d'éviter d'avoir `index.php` dans l'url. Mais pour que ça fonctionne il faut activer le module `mod_rewrite`

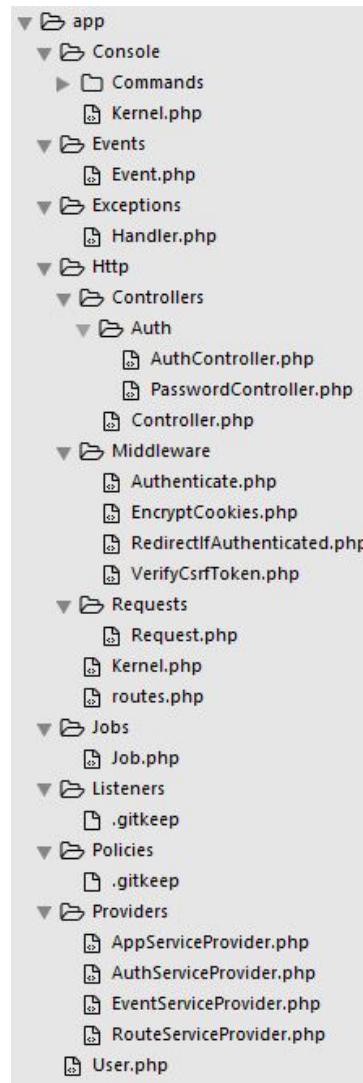
 Une autre façon d'obtenir un Laravel sur mesure est d'utiliser [mon outil en ligne](#). Je le décris un peu [ici](#).

Organisation de Laravel

Maintenant qu'on a un Laravel tout neuf et qui fonctionne voyons un peu ce qu'il contient.

Dossier app

Ce dossier contient les éléments essentiels de l'application :



Dossier App

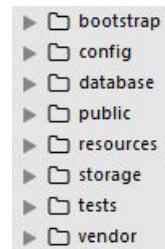
- **Console/Commands** : toutes les commandes en mode console, il y a au départ une commande `Inspire` qui sert d'exemple,
- **Jobs** : commandes concernant les tâches que doit effectuer l'application. C'est une nouveauté de la version 5 que je n'aborderai pas dans ce cours,
- **Events et Listeners** : événements et écouteurs nécessaires pour l'application,

- **Http** : tout ce qui concerne la communication : contrôleurs, routes, middlewares (il y a quatre middlewares de base) et requêtes,
- **Providers** : tous les fournisseurs de services (providers), il y en a déjà 4 au départ. Les providers servent à initialiser les composants.
- **Policies** : une évolution récente qui permet de gérer facilement les droits d'accès.

On trouve également le fichier User.php qui est un modèle qui concerne les utilisateurs pour la base de données.

Évidemment tout cela doit vous paraître assez nébuleux pour le moment mais nous verrons en détail la plupart de ces sections au fil du cours.

Autres dossiers



Voici une description du contenu des autres dossiers :

- **bootstrap** : scripts d'initialisation de Laravel pour le chargement automatique des classes, la fixation de l'environnement et des chemins, et pour le démarrage de l'application,
- **public** : tout ce qui doit apparaître dans le dossier public du site : images, CSS, scripts...
- **vendor** : tous les composants de Laravel et de ses dépendances,
- **config** : toutes les configurations : application, authentification, cache, base de données, espaces de noms, emails, systèmes de fichier, session...
- **database** : migrations et les populations,
- **resources** : vues, fichiers de langage et assets (par exemple les fichiers LESS ou Sass),
- **storage** : données temporaires de l'application : vues compilées, caches, clés de session...
- **tests** : fichiers de tests unitaires.

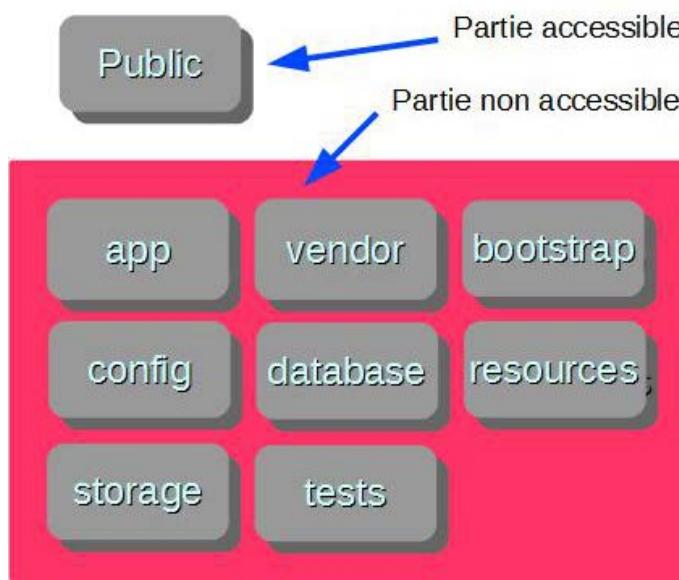
Fichiers de la racine

Il y a un certain nombre de fichiers dans la racine dont voici les principaux :

- **artisan** : outil en ligne de Laravel pour des tâches de gestion,
- **composer.json** : fichier de référence de Composer,
- **phpunit.xml** : fichier de configuration de phpunit (pour les tests unitaires),
- **.env** : fichier pour spécifier l'environnement d'exécution.

Accessibilité

Pour des raisons de sécurité sur le serveur seul le dossier **public** doit être accessible :



Le dossier public est le seul accessible

Cette configuration n'est pas toujours possible sur un serveur mutualisé, il faut alors modifier un peu Laravel pour que ça fonctionne; j'en parlerai dans le chapitre sur le déploiement.

Environnement et messages d'erreur

Par défaut lorsque vous installez Laravel, celui-ci est en mode "debug". Au niveau de l'affichage des erreurs si vous entrez une URL qui n'est pas prévue vous allez obtenir quelque chose comme ceci :

Sorry, the page you are looking for could not be found.

1/1 [NotFoundHttpException](#) in [RouteCollection.php](#) line 161:

```

1. in RouteCollection.php line 161
2. at RouteCollection->match(object(Request)) in Router.php line 802
3. at Router->findRoute(object(Request)) in Router.php line 670
4. at Router->dispatchToRoute(object(Request)) in Router.php line 654
5. at Router->dispatch(object(Request)) in Kernel.php line 246
6. at Kernel->Illuminate\Foundation\Http\{closure}(object(Request))
7. at call_user_func(object(Closure), object(Request)) in Pipeline.php line 139
8. at Pipeline->Illuminate\Pipeline\{closure}(object(Request)) in CheckForMaintenanceMode.php line 44
9. at CheckForMaintenanceMode->handle(object(Request), object(Closure))
10. at call_user_func_array(array(object(CheckForMaintenanceMode), 'handle'), array(object(Request), object(Closure))) in Pipeline.php line 124
11. at Pipeline->Illuminate\Pipeline\{closure}(object(Request))
12. at call_user_func(object(Closure), object(Request)) in Pipeline.php line 103
13. at Pipeline->then(object(Closure)) in Kernel.php line 132
14. at Kernel->sendRequestThroughRouter(object(Request)) in Kernel.php line 99
15. at Kernel->handle(object(Request)) in index.php line 54

```

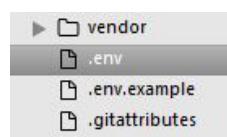
Un message d'erreur en mode "debug"

Pendant la phase de développement on a besoin d'obtenir des messages explicites pour traquer les erreurs inévitables que nous allons faire. En mode "production" il faudra changer ce mode, pour cela ouvrez le fichier `config/app.php` et trouvez cette ligne :

```
'debug' => env('APP_DEBUG', false),
```

abap

Autrement dit on va chercher la valeur dans l'environnement, mais où peut-on le trouver ? Regardez à la racine des dossiers, vous y trouvez un fichier `.env` :



Le fichier de l'environnement

Avec ce contenu :

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=base64:/qnv1yuwcCDFJuki91gc7LBtIzRkJgFxusIX2x1wwUM=
APP_URL=http://localhost

DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

CACHE_DRIVER=file
SESSION_DRIVER=file
...
```

html

Vous remarquez que dans ce fichier la variable `APP_DEBUG` a la valeur `true`. On va la conserver ainsi puisqu'on veut être en mode "debug". Vous êtes ainsi en mode débogage avec affichage de messages d'erreur détaillés. Si vous la mettez à `false` (ou si vous la supprimez), avec une URL non prévue vous obtenez maintenant juste :

Sorry, the page you are looking for could not be found.

Un message d'erreur en mode "production"

Il ne faudra évidemment pas laisser la valeur `true` lors d'une mise en production ! On reparlera de ça lorsqu'on verra la gestion de l'environnement. Vous ne risquerez ainsi plus d'oublier de changer cette valeur parce que Laravel saura si vous êtes sur votre serveur de développement ou sur celui de production.

 La valeur de `APP_KEY` qui sécurise les informations est automatiquement générée lors de l'installation avec `create-project`.

Le composant Html

Dans la version 4 de Laravel il y avait directement le composant `Html` qui permet de créer facilement des formulaires et qui offre un lot d'helpers pour l'écriture du HTML. Dans la version 5 ce composant n'est pas chargé par défaut. Comme nous en aurons besoin dans ce cours, une fois que vous avez réussi à installer une application toute neuve de Laravel vous allez modifier ainsi le fichier `composer.json` :

```
"require": {
    "php": ">=5.5.9",
    "laravel/framework": "5.2.*",
    "laravelcollective/html": "5.2.*"
},
```

javascript

On demande ainsi à Composer de charger le composant `laravelcollective/html`. Lancez alors une mise à jour (attention de bien vous positionner dans le dossier racine de l'application) :

```
composer update
```

text

Attendez la fin du chargement. Il faut ensuite modifier ainsi le fichier `config/app.php` :

```
<?php

/*
 * Application Service Providers...
 */
App\Providers\AppServiceProvider::class,
App\Providers\AuthServiceProvider::class,
App\Providers\EventServiceProvider::class,
App\Providers\RouteServiceProvider::class,
Collective\Html\HtmlServiceProvider::class,

...
```

php

```
'View'  => Illuminate\Support\Facades\View::class,  
'Form'  => Collective\Html\FormFacade::class,  
'Html'   => Collective\Html\HtmlFacade::class,  
...
```

Donc il faut ajouter cette ligne dans le tableau des providers :

```
<?php  
Collective\Html\HtmlServiceProvider::class,
```

php

Il faut aussi ajouter ces deux lignes dans le tableau des alias :

```
<?php  
'Form'  => Collective\Html\FormFacade::class,  
'Html'   => Collective\Html\HtmlFacade::class,
```

php

Ainsi vous allez disposer de ce composant bien utile !

 Le composant utilisé est dérivé de `Illuminate/html` qui ne sera plus suivi.

En résumé

- Pour son installation et sa mise à jour Laravel utilise le gestionnaire de dépendances Composer.
- La création d'une application Laravel se fait à partir de la console avec une simple ligne de commande.
- Laravel est organisé en plusieurs dossiers.
- Le dossier `public` est le seul qui doit être accessible pour le client.
- L'environnement est fixé à l'aide du fichier `.env`
- Le composant `Html` n'est pas prévu par défaut, il faut le charger indépendamment.



Présentation générale

Le routage et les façades



L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

OpenClassrooms

Qui sommes-nous ?

Fonctionnement de nos cours

Recrutement

Nous contacter

En plus

Créer un cours

Professionnels



Affiliation

Entreprises

Universités et écoles

Suivez-nous

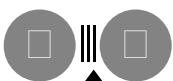


Le blog OpenClassrooms

[S'inscrire](#)[Se connecter](#)[Accueil](#) ▶ [Cours](#) ▶ [Découvrez le framework PHP Laravel](#) ▶ [Le routage et les façades](#)

Découvrez le framework PHP Laravel

15 heures Moyenne



Le routage et les façades

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Dans ce chapitre nous allons nous intéresser au devenir d'une requête HTTP qui arrive dans notre application Laravel. Nous allons voir l'intérêt d'utiliser un fichier htaccess pour simplifier les url. Nous verrons aussi le système de routage pour trier les requêtes.

Les requêtes HTTP

Petit rappel

On va commencer par un petit rappel sur ce qu'est une requête HTTP. Voici un schéma illustratif :



Les requêtes HTTP

Le HTTP (Hypertext Transfer Protocol) est un protocole de communication entre un client et un serveur. Le client demande une page au serveur en envoyant une requête et le serveur répond en envoyant une réponse, en général une page HTML.

La requête du client comporte un certain nombre d'informations mais nous allons nous intéresser pour le moment seulement à deux d'entre elles :

- la **méthode** : get, post, put, delete...
- l'**url** : c'est l'adresse de la page demandée sur le serveur

Notre application Laravel doit savoir interpréter ces informations et les utiliser de façon pertinente pour renvoyer ce que demande le client. Nous allons voir comment cela est réalisé.

.htaccess et index.php

On veut que toutes les requêtes aboutissent obligatoirement sur le fichier `index.php` situé dans le dossier `public`. Pour y arriver on peut utiliser une URL de ce genre :

`http://monsite.fr/index.php/mapage`

Mais ce n'est pas très esthétique avec ce `index.php` au milieu. Si vous avez un serveur Apache lorsque la requête du client arrive sur le serveur où se trouve notre application Laravel elle passe en premier par le fichier `.htaccess`, s'il existe, qui fixe des règles pour le serveur. Il y a justement un fichier `.htaccess` dans le dossier public de Laravel avec une règle de réécriture de telle sorte qu'on peut avoir une url simplifiée :

`http://monsite.fr/mapage`

Un petit schéma pour visualiser cette action :



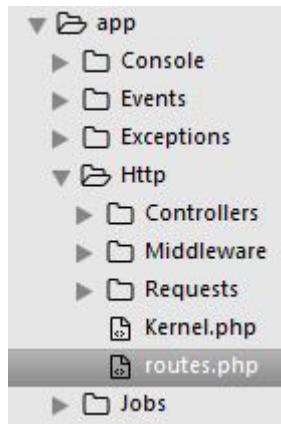
Pour que ça fonctionne il faut que le serveur Apache ait le module `mod_rewrite` activé.

Si vous n'utilisez pas Apache mais Nginx il faut utiliser cette directive :

```
location / {  
    try_files $uri $uri/ /index.php?$query_string;  
}
```

Le cycle de la requête

Lorsque la requête atteint le fichier `public/index.php` l'application Laravel est créée et configurée et l'environnement est détecté. Nous reviendrons plus tard plus en détail sur ces étapes. Ensuite le fichier `routes.php` est chargé. Voici l'emplacement du fichier des routes :



Le fichier des routes

C'est avec ce fichier que la requête va être analysée et dirigée. Regardons ce qu'on y trouve au départ :

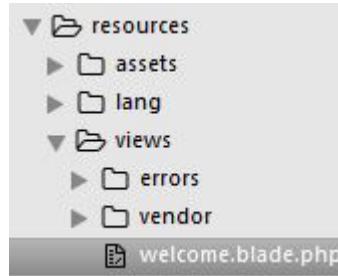
```
<?php
Route::get('/', function () {
    return view('welcome');
});
```

abap

Comme Laravel est explicite vous pouvez déjà deviner à quoi sert ce code :

- **Route** : on utilise le routeur,
- **get** : on regarde si la requête a la méthode "get",
- **'/'** : on regarde si l'url comporte uniquement le nom de domaine,
- dans la fonction anonyme on retourne (`return`) une vue (`view`) à partir du fichier "welcome".

Ce fichier "welcome" se trouve bien rangé dans le dossier des vues :

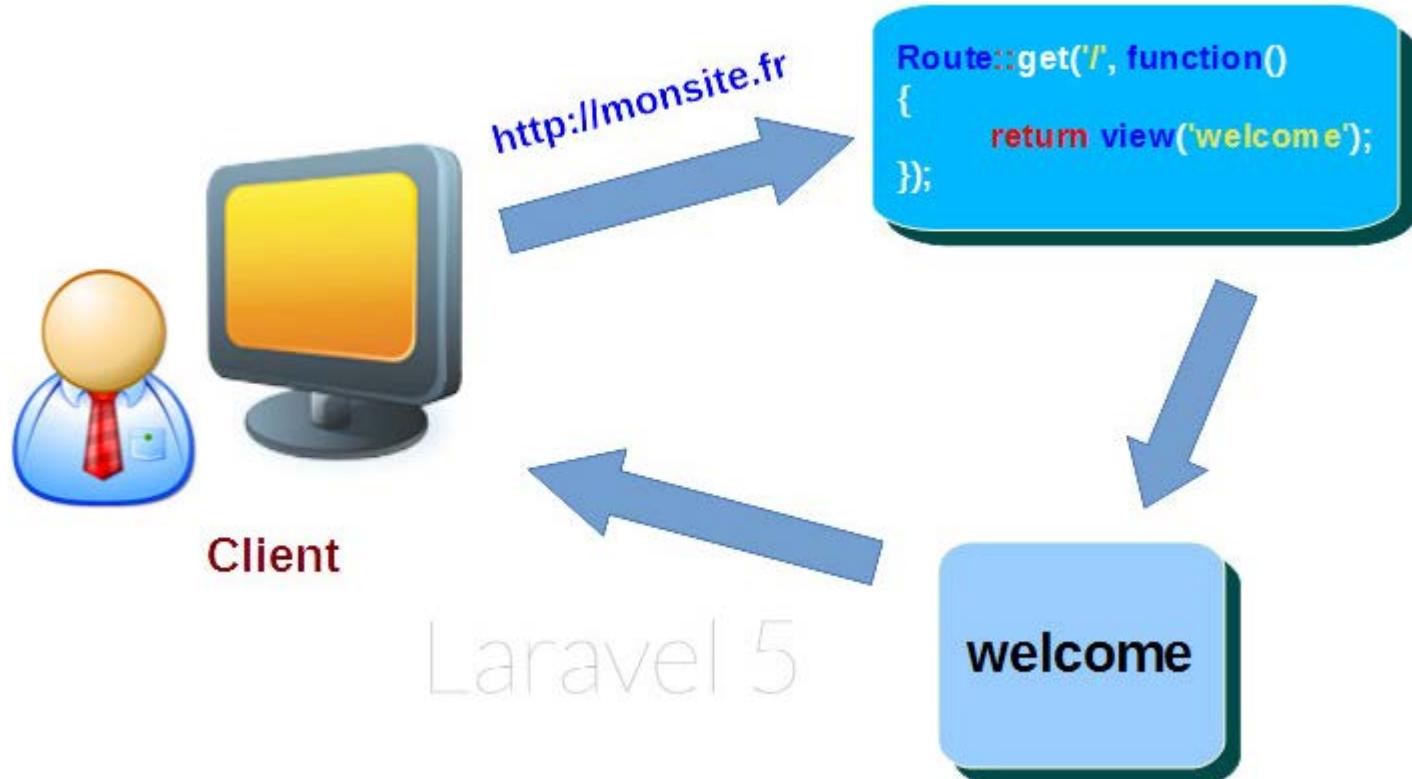


La vue "welcome" dans le dossier des vues

C'est ce fichier comportant du code html qui génère le texte d'accueil que vous obtenez au démarrage initial de Laravel.

Laravel propose plusieurs helpers qui simplifient la syntaxe. Il y a par exemple `view` pour la classe View comme on l'a vu dans le code ci-dessus. Laravel est équipé de nombreux helpers comme celui-ci qui facilitent et accélèrent le codage.

Visualisons le cycle de la requête :



Le cycle de la requête

Sur votre serveur local vous n'avez pas de nom de domaine et vous allez utiliser une url de la forme `http://localhost/tuto/public` en admettant que vous ayez créé Laravel dans un dossier `www/tuto`. Mais vous pouvez aussi créer un hôte virtuel pour avoir une situation plus réaliste.

Plusieurs routes et paramètre de route

A l'installation Laravel a une seule route qui correspond à l'url de base composée uniquement du nom de domaine. Voyons maintenant comment créer d'autres routes. Imaginons que nous avons 3 pages qui doivent être affichées avec ces urls :

1. `http://monsite.fr/1`
2. `http://monsite.fr/2`
3. `http://monsite.fr/3`

J'ai fait apparaître en gras la partie spécifique de l'url pour chaque page. Il est facile de réaliser cela avec ce code :

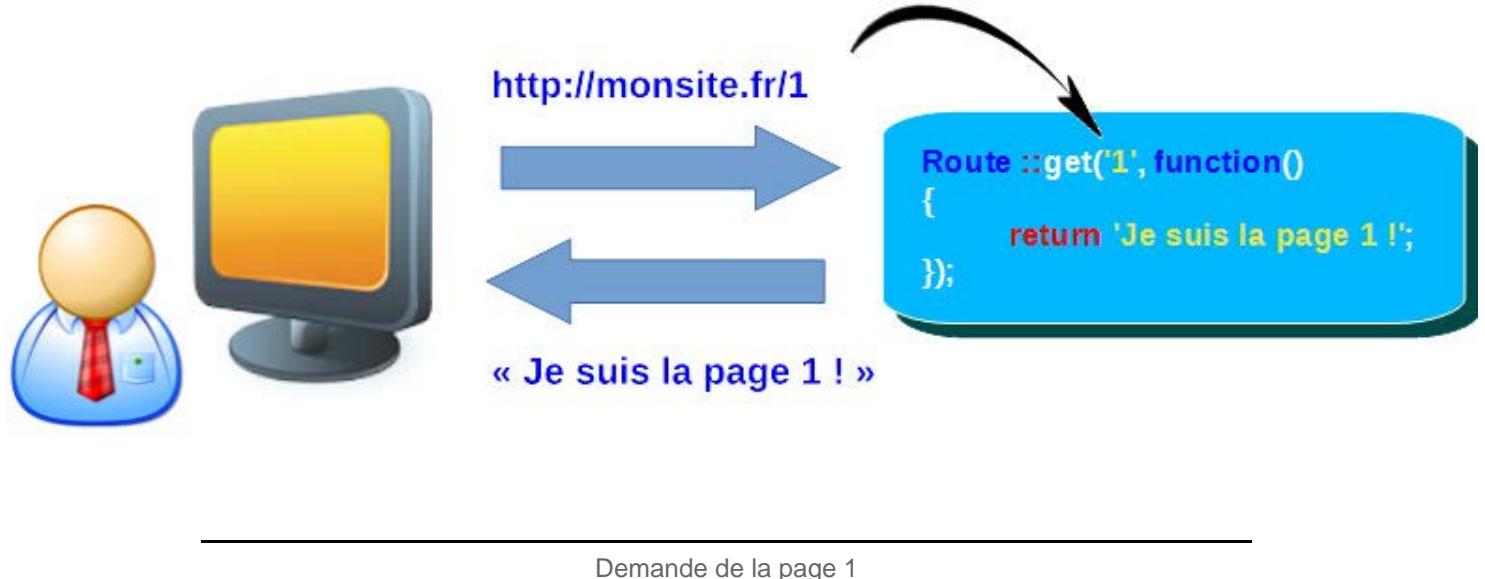
```

<?php
Route::get('1', function() { return 'Je suis la page 1 !'; });
Route::get('2', function() { return 'Je suis la page 2 !'; });
Route::get('3', function() { return 'Je suis la page 3 !'; });

```

php

Cette fois je n'ai pas créé de vue parce que ce qui nous intéresse est uniquement une mise en évidence du routage, je retourne donc directement la réponse au client. Visualisons cela pour la page 1 :



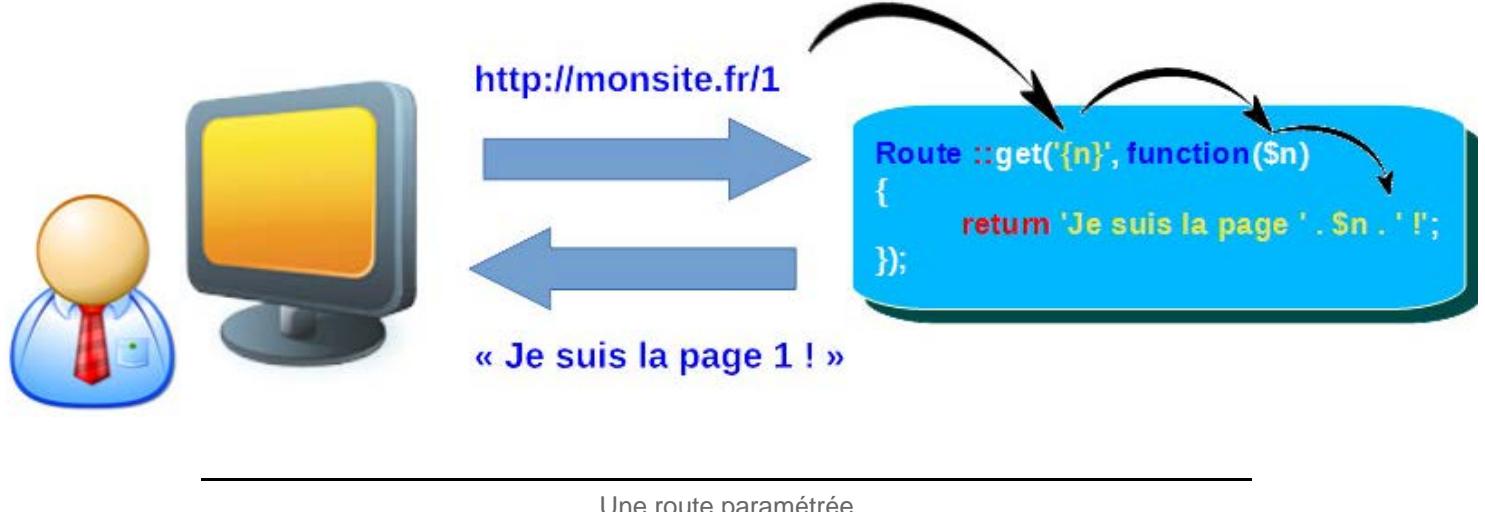
Demande de la page 1

On a besoin du caractère "/" uniquement dans la route de base.

On peut maintenant se poser une question : est-il vraiment indispensable de créer 3 routes alors que la seule différence tient à peu de chose : une valeur qui change. On peut utiliser un paramètre pour une route qui accepte des éléments variables en utilisant des accolades. Regardez ce code :

```
<?php
Route::get('{n}', function($n) {
    return 'Je suis la page ' . $n . ' !';
});
```

php



Une route paramétrée

On peut rendre un paramètre optionnel en lui ajoutant un point d'interrogation mais il ne doit pas être

suivi par un paramètre obligatoire.

Erreur d'exécution et contrainte de route

Dans mon double exemple précédent lorsque je dis que le résultat est exactement le même je mens un peu.
Que se passe-t-il dans les deux cas pour cette url :

`http://monsite.fr/4`

Dans le cas des trois routes vous tombez sur une erreur :

Sorry, the page you are looking for could not be found.

1/1 **NotFoundHttpException** in `RouteCollection.php` line 161:

```
1. in RouteCollection.php line 161
2. at RouteCollection->match(object(Request)) in Router.php line 802
3. at Router->findRoute(object(Request)) in Router.php line 670
4. at Router->dispatchToRoute(object(Request)) in Router.php line 654
5. at Router->dispatch(object(Request)) in Kernel.php line 246
6. at Kernel->Illuminate\Foundation\Http\{closure}(object(Request))
7. at call_user_func(object(Closure), object(Request)) in Pipeline.php line 139
8. at Pipeline->Illuminate\Pipeline\{closure}(object(Request)) in CheckForMaintenanceMode.php line 44
9. at CheckForMaintenanceMode->handle(object(Request), object(Closure))
10. at call_user_func_array(array(object(CheckForMaintenanceMode), 'handle'), array(object(Request), object(Closure))) in Pipeline.php line 124
11. at Pipeline->Illuminate\Pipeline\{closure}(object(Request))
12. at call_user_func(object(Closure), object(Request)) in Pipeline.php line 103
13. at Pipeline->then(object(Closure)) in Kernel.php line 132
14. at Kernel->sendRequestThroughRouter(object(Request)) in Kernel.php line 99
15. at Kernel->handle(object(Request)) in index.php line 54
```

Erreur d'exécution : la route n'existe pas

Par contre dans la version avec le paramètre vous obtenez une réponse valide :

`Je suis la page 4 !`

Ce qui est logique parce qu'une route est trouvée. Le paramètre accepte n'importe quelle valeur et pas seulement des nombres. Par exemple avec cette url :

`http://monsite.fr/nimportequoi`

Vous obtenez :

`Je suis la page nimportequoi !`

Ce qui vous l'avouerez n'est pas très heureux !

Pour éviter ce genre de désagrément il faut contraindre le paramètre à n'accepter que certaines valeurs. On réalise cela à l'aide d'une expression régulière :

```
<?php
Route::get('{n}', function($n) {
    return 'Je suis la page ' . $n . ' !';
})->where('n', '[1-3]');
```

php

Maintenant je peux affirmer que les comportements sont identiques ! Mais il nous faudra régler le problème des routes non prévues. Nous verrons cela dans un prochain chapitre.

Route nommée

Il est parfois utile de nommer une route, par exemple pour générer une URL ou pour effectuer une redirection.

La syntaxe pour nommer une route est celle-ci :

```
<?php
Route::get('/', ['as' => 'home', function()
{
    return 'Je suis la page d\'accueil !';
}]);
```

php

Nous verrons des cas d'utilisation de routes nommées dans les prochains chapitres.

Les façades

Laravel propose de nombreuses façades pour simplifier la syntaxe. Vous pouvez les trouver toutes déclarées dans le fichier `config/app.php` :

```
<?php
'aliases' => [
    'App'      => Illuminate\Support\Facades\App::class,
    'Artisan'   => Illuminate\Support\Facades\Artisan::class,
    'Auth'      => Illuminate\Support\Facades\Auth::class,
    'Blade'     => Illuminate\Support\Facades\Blade::class,
    'Cache'     => Illuminate\Support\Facades\Cache::class,
    'Config'    => Illuminate\Support\Facades\Config::class,
    'Cookie'    => Illuminate\Support\Facades\Cookie::class,
    'Crypt'     => Illuminate\Support\Facades\Crypt::class,
    'DB'        => Illuminate\Support\Facades\DB::class,
    'Eloquent'   => Illuminate\Database\Eloquent\Model::class,
    'Event'     => Illuminate\Support\Facades\Event::class,
    'File'       => Illuminate\Support\Facades\File::class,
    'Gate'      => Illuminate\Support\Facades\Gate::class,
    'Hash'       => Illuminate\Support\Facades\Hash::class,
    'Lang'      => Illuminate\Support\Facades\Lang::class,
    'Log'        => Illuminate\Support\Facades\Log::class,
    'Mail'      => Illuminate\Support\Facades\Mail::class,
    'Password'  => Illuminate\Support\Facades>Password::class,
    'Queue'     => Illuminate\Support\Facades\Queue::class,
    'Redirect'   => Illuminate\Support\Facades\Redirect::class,
    'Redis'     => Illuminate\Support\Facades\Redis::class,
    'Request'   => Illuminate\Support\Facades\Request::class,
    'Response'  => Illuminate\Support\Facades\Response::class,
    'Route'     => Illuminate\Support\Facades\Route::class,
    'Schema'    => Illuminate\Support\Facades\Schema::class,
    'Session'   => Illuminate\Support\Facades\Session::class,
    'Storage'   => Illuminate\Support\Facades\Storage::class,
    'URL'       => Illuminate\Support\Facades\Url::class,
    'Validator' => Illuminate\Support\Facades\Validator::class,
    'View'       => Illuminate\Support\Facades\View::class,
    'Form'      => Collective\Html\FormFacade::class,
    'Html'      => Collective\Html\HtmlFacade::class,
],
```

abap

Vous trouvez dans ce tableau le nom de la façade et la classe qui met en place cette façade. Par exemple pour les routes on a la façade `Route` qui correspond à la classe `Illuminate\Support\Facades\Route`. Regardons cette classe :

```
<?php

namespace Illuminate\Support\Facades;

/**
 * @see \Illuminate\Routing\Router
 */
class Route extends Facade
{
    /**
     * Get the registered name of the component.
     *
     * @return string
     */
    protected static function getFacadeAccessor()
    {
        return 'router';
    }
}
```

On se contente de retourner 'router'. Il faut aller voir dans le fichier `Illuminate\Routing\RoutingServiceProvider` pour trouver l'enregistrement du router :

```
<?php
/**
 * Register the router instance.
 *
 * @return void
 */
protected function registerRouter()
{
    $this->app['router'] = $this->app->share(function ($app) {
        return new Router($app['events'], $app);
    });
}
```

Les providers permettent d'enregistrer des composants dans le conteneur de Laravel. Ici on déclare "router" et on voit qu'on crée une instance de la classe Router (new Router...). Le nom complet est `Illuminate\Routing\Router`. Si vous allez voir cette classe vous trouverez les méthodes qu'on a utilisées dans ce chapitre, par exemple `get` :

```
<?php
/**
 * Register a new GET route with the router.
 *
 * @param string $uri
 * @param \Closure|array|string $action
 * @return \Illuminate\Routing\Route
 */
public function get($uri, $action)
{
    return $this->addRoute(['GET', 'HEAD'], $uri, $action);
}
```

Autrement dit si j'écris en utilisant la façade :

```
<?php
Route::get('/', function() { return 'Coucou'; });
```

J'obtiens le même résultat que si j'écris en allant chercher le routeur dans le conteneur :

```
<?php  
$this->app['router']->get('/', function() { return 'Coucou'; });
```

Ou encore en utilisant un helper :

```
<?php  
app('router')->get('/', function() { return 'Coucou'; });
```

php

La différence est que la première syntaxe est plus simple et intuitive mais certains n'aiment pas trop ce genre d'appel statique.

En résumé

- Laravel possède un fichier `.htaccess` pour simplifier l'écriture des url.
- Le système de routage est simple et explicite.
- On peut prévoir des paramètres dans les routes.
- On peut contraindre un paramètre à correspondre à une expression régulière.
- On peut nommer une route pour faciliter la génération des URL et les redirections.
- Il faut prévoir de gérer toutes les url, même celles qui n'ont aucune route prévue.
- Laravel est équipé de nombreuses façades qui simplifient la syntaxe.
- Il existe aussi des helpers pour simplifier la syntaxe.



Installation et organisation

Les réponses



L'auteur

Maurice Chavelli

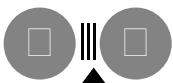
Développeur retraité qui se consacre à l'enseignement des technologies du web.

[S'inscrire](#)[Se connecter](#)

Accueil ▶ Cours ▶ Découvrez le framework PHP Laravel ▶ Les réponses

Découvrez le framework PHP Laravel

15 heures Moyenne



Les réponses

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Nous avons vu précédemment comment la requête qui arrive est traitée par les routes. Voyons maintenant les réponses que nous pouvons renvoyer au client. Nous allons voir le système des vues de Laravel avec la possibilité de transmettre des paramètres. Nous verrons aussi comment créer des templates avec l'outil Blade.

Construire une réponse

Les codes des réponses

Dans le protocole HTTP il existe des codes pour spécifier les réponses. Ces codes sont classés par grandes catégories. Voici les 3 principaux :

- 200 : requête exécutée avec succès
- 404 : l'adresse demandée n'a pas été trouvée
- 500 : erreur sur le serveur

Dans les exemples que j'ai utilisés dans le chapitre précédent sur les routes, je n'ai pas précisé de code. Je me suis contenté de retourner un texte au client. Celui-ci n'a aucune utilité du code et veut quelque chose d'explicite, par contre les moteurs de recherche savent interpréter ces codes.

En plus du simple `return` pour renvoyer la réponse je peux utiliser l'helper `response` et préciser le code :

```
<?php  
Route::get('{n}', function($n) {  
    return response('Je suis la page ' . $n . ' !', 200);  
})->where('n', '[1-3]');
```

php

Maintenant je retourne une véritable réponse HTTP avec le code correspondant.

Il existe la façade Response pour les réponses. On peut donc écrire :

```
<?php  
return Response::make('Je suis la page ' . $n . ' !', 200);
```

php

ce qui est équivalent à ce qu'on a écrit ci-dessus.

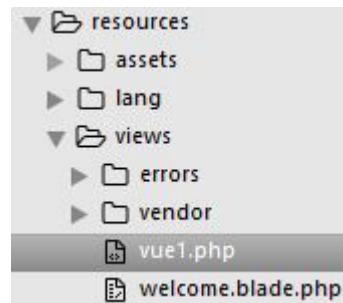
Les vues

Dans une application réelle vous retournez rarement la réponse directement à partir d'une route, vous passerez au moins par une vue. Dans sa version la plus simple une vue est un simple fichier avec du code html :

```
<!doctype html>  
<html lang="fr">  
<head>  
    <meta charset="UTF-8">  
    <title>Ma première vue</title>  
</head>  
<body>  
    Je suis une vue !  
</body>  
</html>
```

html

Il faut enregistrer cette vue (j'ai choisi le nom "vue1") dans le dossier `resources/views` avec l'extension `.php` :



La vue dans le dossier des vues

Même si vous ne mettez que du code HTML dans une vue vous devez l'enregistrer avec l'extension **php**.

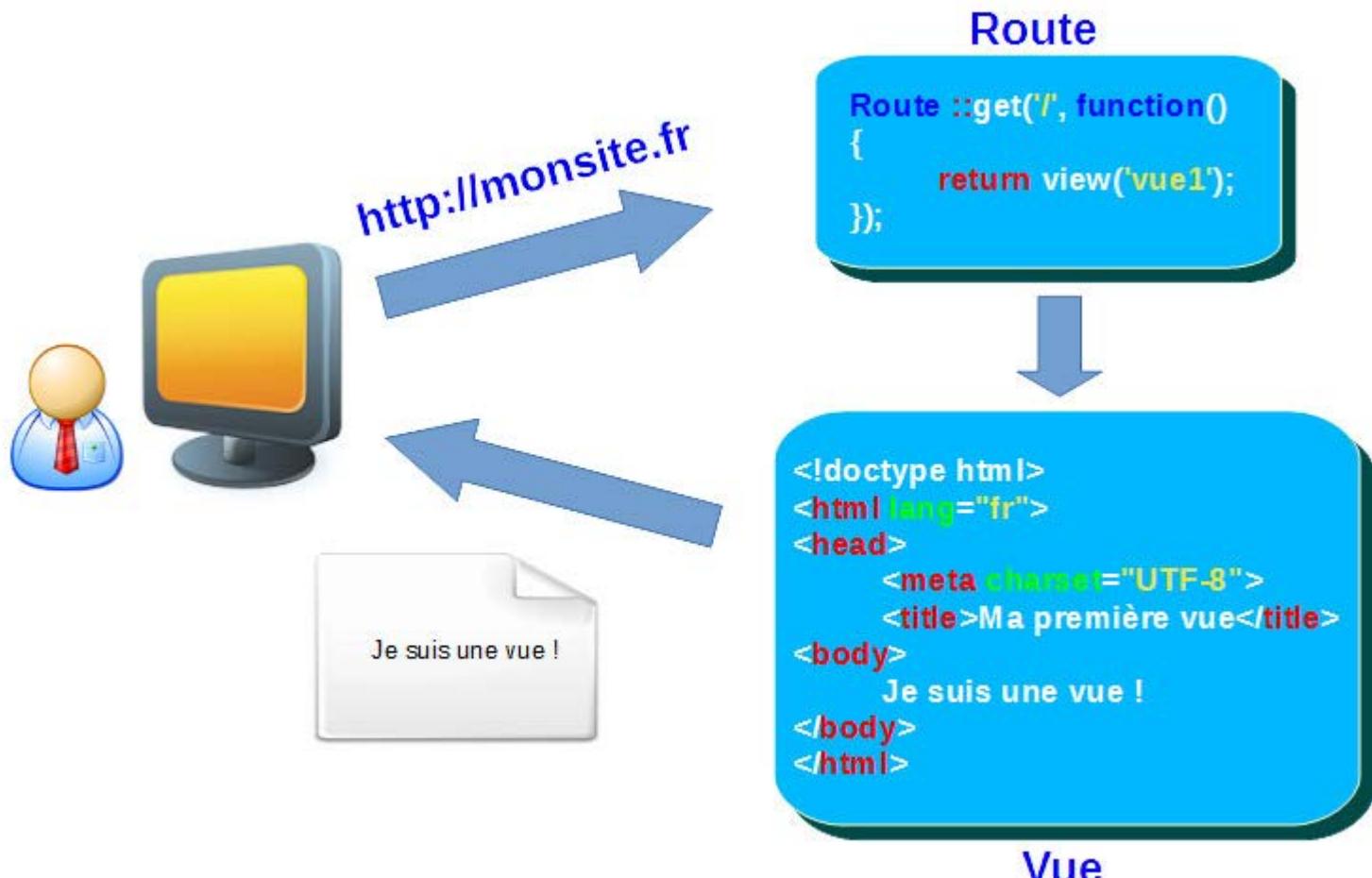
On peut appeler cette vue à partir d'une route avec ce code :

```
<?php  
Route::get('/', function()  
{  
    return view('vue1');  
});
```

php

Je vous rappelle la belle sémantique de Laravel qui se lit comme de la prose : je retourne (`return`) une vue (`view`) à partir du fichier de vue "vue1".

Voici une illustration du processus :



Le cycle de requête avec une vue

Vue paramétrée

En général on a des informations à transmettre à une vue, voyons à présent comment mettre cela en place. Supposons que nous voulons répondre à ce type de requête :

`http://monsite.fr/article/n`

Le paramètre `n` pouvant prendre une valeur numérique. Voyons comment cette url est constituée :



Constitution de l'url

- la base de l'url est constante pour le site, quelle que soit la requête,

- la partie fixe ici correspond aux articles,
- la partie variable correspond au numéro de l'article désiré.

Route

Il nous faut une route pour intercepter ces urls :

```
<?php
Route::get('article/{n}', function($n) {
    return view('article')->with('numero', $n);
})->where('n', '[0-9]+');
```

abap

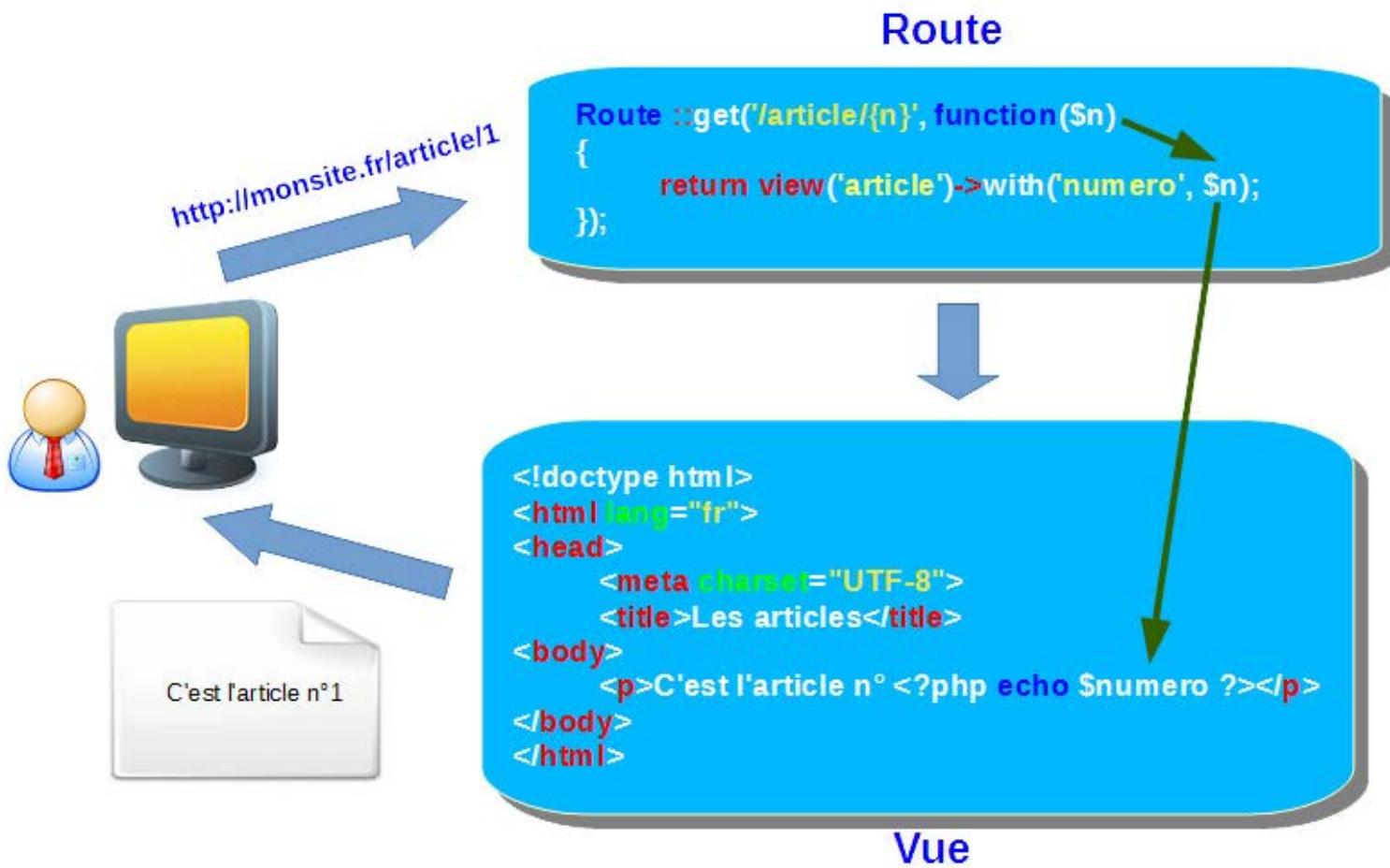
Vue

Il ne nous reste plus qu'à créer la vue `article.php` dans le dossier `resources/views` :

```
<!doctype html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
        <title>Les articles</title>
</head>
<body>
    <p>C'est l'article n° <?php echo $numero ?></p>
</body>
</html>
```

html

Pour récupérer le numéro de l'article on utilise la variable `$numero`. Voici une schématisation du fonctionnement :



Il existe une méthode "magique" pour transmettre un paramètre, par exemple pour transmettre la variable `numero` comme je l'ai fait ci-dessus on peut écrire le code ainsi :

```
<?php  
return view('article')->withNumero($n);
```

Il suffit de concaténer le nom de la variable au mot clé `with`

On peut aussi transmettre un tableau comme deuxième paramètre :

```
<?php  
return view('article', ['numero' => $n]);
```

Blade

Simplifier la syntaxe

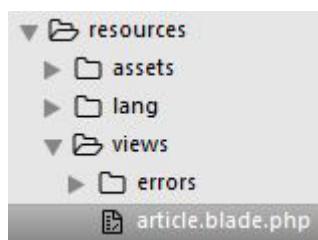
Laravel possède un moteur de template élégant nommé Blade qui nous permet de faire pas mal de choses. La première est de nous simplifier la syntaxe. Par exemple au lieu de la ligne suivante que nous avons prévue dans la vue précédente :

```
<p>C'est l'article n° <?php echo $numero ?></p>
```

On peut utiliser cette syntaxe avec Blade :

```
<p>C'est l'article n° {{ $numero }}</p>
```

Tout ce qui se trouve entre les doubles accolades est interprété comme du code PHP. Mais pour que ça fonctionne il faut indiquer à Laravel qu'on veut utiliser Blade pour cette vue. Ça se fait simplement en modifiant le nom du fichier :



Une vue activée pour Blade

Il suffit d'ajouter "blade" avant l'extension "php". Vous pouvez tester l'exemple précédent avec ces modifications et vous verrez que tout fonctionne parfaitement avec une syntaxe épurée.

Il y a aussi la version avec la syntaxe `{!! ... !!}`. La différence entre les deux versions est que le texte entre les doubles accolades est échappé ou purifié. C'est une mesure de sécurité parce qu'un utilisateur pourrait très bien mettre du code malicieux dans l'url.

Un template

Une fonction fondamentale de Blade est de permettre de faire du templating, c'est à dire de factoriser du code de présentation. Poursuivons notre exemple en complétant notre application avec une autre route chargée d'intercepter des urls pour des factures ainsi que sa vue :

```
<?php
Route::get('facture/{n}', function($n) {
    return view('facture')->withNumero($n);
})->where('n', '[0-9]+');
```

php

```
<!doctype html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Les factures</title>
</head>
<body>
    <p>C'est la facture n° {{ $numero }}</p>
</body>
</html>
```

html

On se rend compte que cette vue est pratiquement la même que celle pour les articles. Il serait intéressant de placer le code commun dans un fichier. C'est justement le but d'un template d'effectuer cette opération.

Voici le template :

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

html

J'ai repris le code commun et prévu deux emplacements repérés par le mot clé @yield et nommés "titre" et "contenu". Il suffit maintenant de modifier les deux vues. Voilà pour les articles :

```
@extends('template')

@section('titre')
    Les articles
@endsection

@section('contenu')
    <p>C'est l'article n° {{ $numero }}</p>
@endsection
```

html

Et voilà pour les factures :

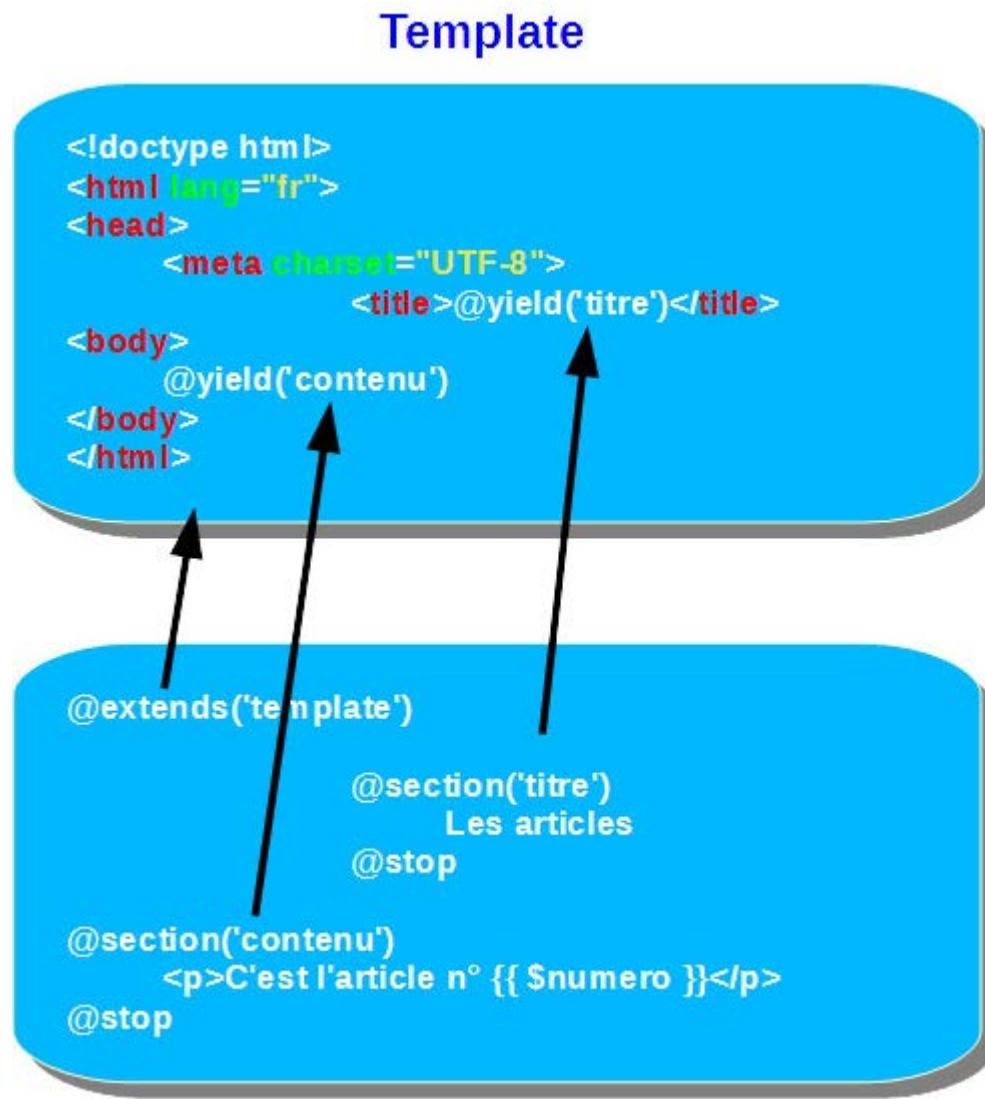
```
@extends('template')

@section('titre')
    Les factures
@endsection

@section('contenu')
    <p>C'est la facture n° {{ $numero }}</p>
@endsection
```

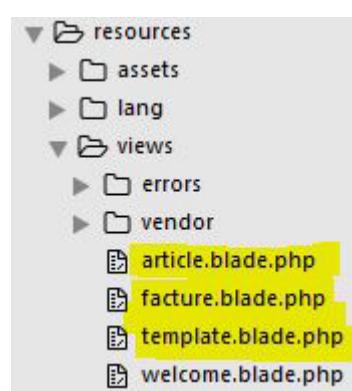
html

Dans un premier temps on dit qu'on veut utiliser le template avec `@extends` et le nom du template "template". Ensuite on remplit les zones prévues dans le template grâce à la syntaxe `@section` en précisant le nom de l'emplacement et en fermant avec `@endsection` (l'ancienne syntaxe `@stop` est encore fonctionnelle). Voici un schéma pour bien visualiser tout ça avec les articles :



Fonctionnement du template

Au niveau du dossier des vues on a donc les trois fichiers :



Dossier des vues

Lorsqu'elles deviendront nombreuses on organisera nos vues dans des dossiers. Vous pouvez d'ailleurs remarquer qu'il en existe déjà plusieurs dans l'installation de base.

Les redirections

Souvent il ne faut pas envoyer directement la réponse mais rediriger sur une autre url. Pour réaliser cela on a l'`helper redirect` :

```
<?php  
return redirect('facture');
```

php

Ici on redirige sur l'url `facture`. On peut aussi rediriger sur une route nommée :

```
<?php  
return redirect()->route('facture');
```

php

Ici on redirige sur la route nommée `facture`.

On verra de nombreux exemples de redirections dans les prochains chapitres.

En résumé

- Laravel offre la possibilité de créer des vues.
- Il est possible de transmettre simplement des paramètres aux vues.
- L'outil Blade permet de créer des templates et d'optimiser ainsi le code des vues.
- On peut facilement effectuer des redirections.



Le routage et les façades

Les contrôleurs



L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

Découvrez le framework PHP Laravel

15 heures Moyenne



Les contrôleurs

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

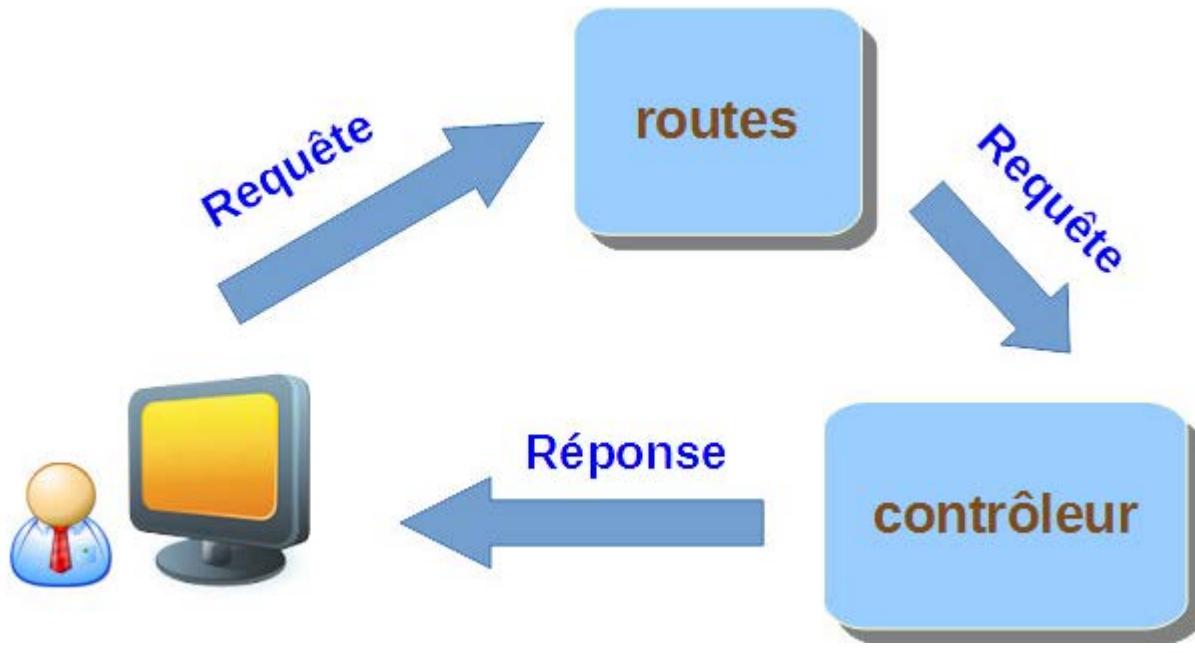
Nous avons vu le cycle d'une requête depuis son arrivée, son traitement par les routes et sa réponse avec des vues qui peuvent être boostées par Blade. Avec tous ces éléments vous pourriez très bien réaliser un site web complet mais Laravel offre encore bien des outils performants que je vais vous présenter.

Pour bien organiser son code dans une application Laravel il faut bien répartir les tâches. Dans les exemples vus jusqu'à présent j'ai renvoyé une vue à partir d'une route, vous ne ferez jamais cela dans une application réelle (même si personne ne vous empêchera de le faire !). Les routes sont juste un système d'aiguillage pour trier les requêtes qui arrivent. Mais alors qui s'occupe de la suite ? Et bien ce sont les contrôleurs, le sujet de ce chapitre.

Les contrôleurs

Rôle

La tâche d'un contrôleur est de réceptionner une requête (qui a déjà été triée par une route) et de définir la réponse appropriée, rien de moins et rien de plus. Voici une illustration du processus :



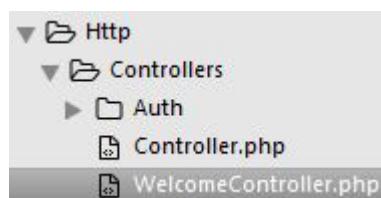
Traitement de la requête par un contrôleur

Constitution

Pour créer un contrôleur nous allons utiliser Artisan, la boîte à outils de Laravel. Dans la console entrez cette commande :

```
php artisan make:controller WelcomeController
```

Si tout se passe bien vous allez trouver le contrôleur ici :



Le contrôleur créé

Avec ce code :

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class WelcomeController extends Controller
{
    //
}
```

Vous allez ajouter la méthode `index`

```
<?php
...
class WelcomeController extends Controller
{
```

php

php

```
public function index()
{
    return view('welcome');
}
```

Analysons un peu ce code :

- On trouve en premier l'espace de nom,
- le contrôleur hérite de la classe `Controller` qui se trouve dans le même dossier et qui permet de factoriser des actions communes à tous les contrôleurs,
- on trouve enfin une méthode `index` qui renvoie quelque chose que maintenant vous connaissez : une vue, en l'occurrence "welcome" dont nous avons déjà parlé. Donc si j'appelle cette méthode je retourne la vue "welcome" au client.

Liaison avec les routes

Maintenant la question qu'on peut se poser est : comment s'effectue la liaison entre les routes et les contrôleurs ? Ouvrez le fichier des routes et entrez ce code :

```
<?php
Route::get('/', 'WelcomeController@index');
```

abap

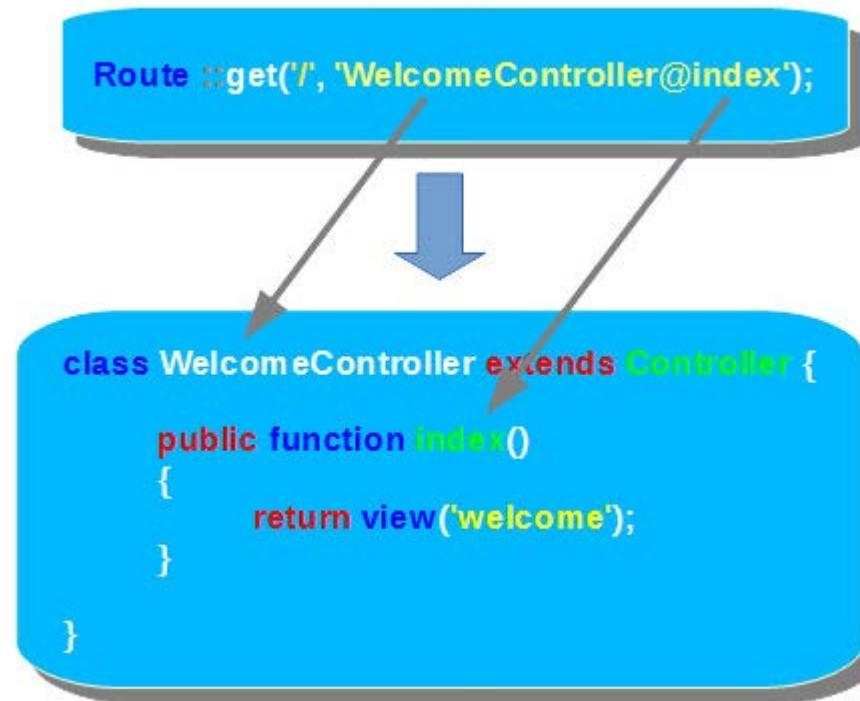
Maintenant avec l'url de base vous devez retrouver la page d'accueil de Laravel :



Page d'accueil

Voici une visualisation de la liaison entre la route et le contrôleur :

Route



Contrôleur

Liaison entre la route et le contrôleur

On voit qu'au niveau de la route il suffit de désigner le nom du contrôleur et le nom de la méthode.



Si vous êtes attentif au code vous avez sans doute remarqué qu'au niveau de la route on ne spécifie pas l'espace de noms du contrôleur, on peut légitimement se demander comment on le retrouve. Laravel nous simplifie la syntaxe en ajoutant automatiquement l'espace de nom .



Si vous devez placer dans un autre espace de noms vos contrôleurs il faut intervenir sur la variable `$namespace` dans le fichier `App\Providers\RouteServiceProvider` :

```
<?php
protected $namespace = 'App\Http\Controllers';
```

php

Cette valeur constitue la base de référence des espaces de noms.

Route nommée

De la même manière que nous pouvons nommer une route classique on peut aussi donner un nom à une route qui pointe une méthode de contrôleur :

```
<?php
Route::get('/', ['uses' => 'WelcomeController@index', 'as' => 'home']);
```

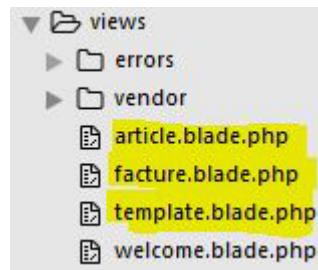
php

Ici on nomme `home` la route vers la méthode `index` du contrôleur `WelcomeController` pour l'URL de base.

Utilisation d'un contrôleur

Voyons maintenant un exemple pratique de mise en œuvre d'un contrôleur. On va conserver notre exemple avec les articles

mais maintenant traité avec un contrôleur. On conserve le même template et les mêmes vues :

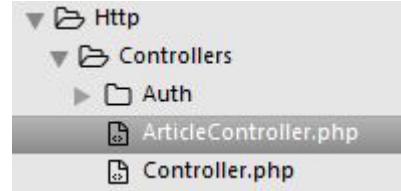


Le template et les vues

On va créer un contrôleur (entraînez-vous à utiliser Artisan) pour les articles :

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
  
use App\Http\Requests;  
use App\Http\Controllers\Controller;  
  
class ArticleController extends Controller  
{  
  
    public function show($n)  
    {  
        return view('article')->with('numero', $n);  
    }  
  
}
```

php



Le dossier des contrôleurs

Dans ce contrôleur on a une méthode **show** chargée de générer la vue. Il ne nous reste plus qu'à créer la route :

```
<?php  
Route::get('article/{n}', 'ArticleController@show')->where('n', '[0-9]+');
```

php

Voici une illustration du fonctionnement avec un contrôleur :

Route

```
Route::get('article/{n}', 'ArticleController@show')->where('n', '[0-9]+');
```

```
class ArticleController extends BaseController {
    public function show($n)
    {
        return View::make('article')->with('numero', $n);
    }
}
```

Contrôleur

Les articles avec un contrôleur

Notez qu'on pourrait utiliser la méthode "magique" pour la transmission du paramètre à la vue :

```
<?php  
return view('article')->withNumero($n);
```

php

En résumé

- Les contrôleurs servent à réceptionner les requêtes triées par les routes et à fournir une réponse au client.
- Artisan permet de créer facilement un contrôleur.
- Il est facile d'appeler une méthode de contrôleur à partir d'une route.
- On peut nommer une route qui pointe vers une méthode de contrôleur.



Les réponses



Les entrées

■ ■ L'auteur ■ ■

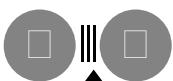
Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

[S'inscrire](#)[Se connecter](#)[Accueil](#) ▶ [Cours](#) ▶ [Découvrez le framework PHP Laravel](#) ▶ [Les entrées](#)

Découvrez le framework PHP Laravel

15 heures Moyenne



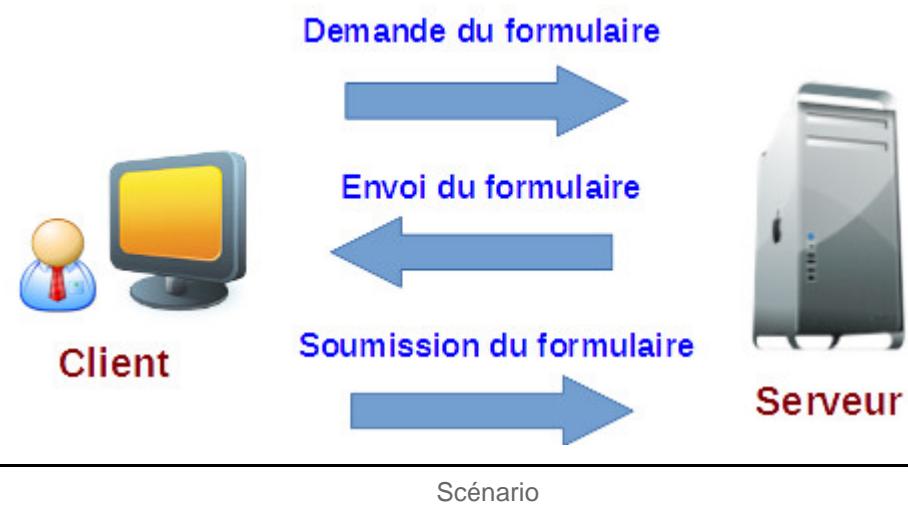
Les entrées

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

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.

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 :

1. une pour la demande du formulaire avec une méthode "get"
2. une pour la soumission du formulaire avec une méthode "post"

On va donc créer ces deux routes dans le fichier `app/Http/routes.php` :

```
<?php
Route::get('users', 'UsersController@getInfos');
Route::post('users', 'UsersController@postInfos');
```

php

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

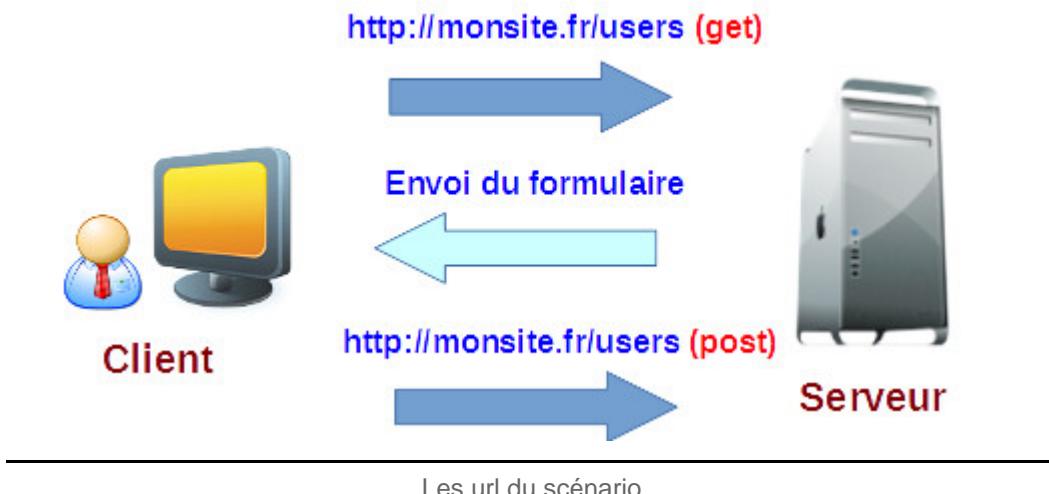


Laravel autorise d'autres verbes comme "put" et "delete" ou plusieurs verbes pour une même route avec "match" et même tous les verbes avec "any".

Les urls correspondantes sont donc :

1. `http://monsite.fr/users` avec la méthode "get"
2. `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 url du scénario

Le middleware

Je parlerai plus en détail des middlewares dans un prochain chapitre. Pour le moment on va se contenter de savoir que c'est du code qui est activé à l'arrivée de la requête (ou à son départ) pour effectuer un traitement. C'est pratique pour arrêter par exemple directement la requête s'il y a un problème de sécurité.

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` :

```
<?php
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
```

php

```
\Illuminate\View\Middleware\ShareErrorsFromSession::class,
\App\Http\Middleware\VerifyCsrfToken::class,
],
['api' => [
    'throttle:60,1',
],
];
```

On trouve les deux middlewares de groupes "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 `app/Http/routes.php` sont incluses dans le groupe "web". Si vous regardez dans le provider `app/Providers/RouteServiceProvider.php` vous trouvez cette inclusion :

```
<?php
protected function mapWebRoutes(Router $router)
{
    $router->group([
        'namespace' => $this->namespace, 'middleware' => 'web',
    ], function ($router) {
        require app_path('Http/routes.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::open(['url' => 'users']) !!}
        {!! Form::label('nom', 'Entrez votre nom : ') !!}
        {!! Form::text('nom') !!}
        {!! Form::submit('Envoyer !') !!}
    {!! Form::close() !!}
@endsection
```

Nous avons déjà vu comment s'organise une vue avec un template, par contre la création du formulaire mérite quelques commentaires. Pour créer un formulaire il faut commencer par l'ouvrir :

```
Form::open(['url' => 'users'])
```

La sémantique est simple : on veut pour un formulaire (`Form` , ouvrir (`open`) celui-ci, et qu'il pointe vers l'url "users".

Ensuite on veut une étiquette (`label`) :

```
Form::label('nom', 'Entrez votre nom :')
```

html

On veut un contrôle de type "text" qui se nomme "nom" :

```
Form::text('nom')
```

html

On veut enfin un bouton de soumission (`submit`) avec le texte "Envoyer !" :

```
Form::submit('Envoyer !')
```

html

Et finalement on veut clore (`close`) le formulaire :

```
Form::close()
```

html

Le code généré pour le formulaire sera alors le suivant :

```
<form method="POST" action="http://monsite.fr/users" accept-charset="UTF-8">
    <input name="_token" type="hidden" value="pV1vWWdUqFDfYsBjKag43C3NvbIC0lHtMnv9BpI">
        <label for="nom">Entrez votre nom : </label>
        <input name="nom" type="text" id="nom">
        <input type="submit" value="Envoyer !">
</form>
```

html

Quelques remarques sur cette génération :

- la méthode par défaut est "post", on n'a pas eu besoin de le préciser,
- l'action est bien générée,
- il y a un contrôle caché (`_token`) destiné à la protection CSRF dont je parlerai plus loin,
- l'étiquette est bien créée avec son attribut "for",
- le contrôle de texte est du bon type avec le bon nom, il est en plus généré un id pour qu'il fonctionne avec son étiquette,
- le bouton de soumission a été généré avec son texte.

Le résultat sera un formulaire sans fioriture :

Le formulaire généré



Si votre formulaire ne se génère pas c'est que vous n'avez peut-être pas chargé le composant `Laravelcollective\Html` comme nous l'avons vu dans le chapitre sur l'installation.

Vous n'êtes pas obligé d'utiliser ce composant pour créer des formulaires mais je vous y encourage parce qu'il simplifie le codage et je l'utilise tout au long de ce cours. Par exemple pour créer le formulaire sans l'utiliser il nous faudrait écrire ceci :

```
@extends('template')

@section('contenu')
    <form method="POST" action="{!! url('users') !!}" accept-charset="UTF-8">
```

html

```
{!! csrf_field() !!}
<label for="nom">Entrez votre nom : </label>
<input name="nom" type="text" id="nom">
<input type="submit" value="Envoyer !">
</form>
@endsection
```

Le contrôleur

Il ne nous manque plus que le contrôleur pour faire fonctionner tout ça :

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

class UsersController extends Controller
{

    public function getInfos()
    {
        return view('infos');
    }

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

php

Mon contrôleur possède deux méthodes :

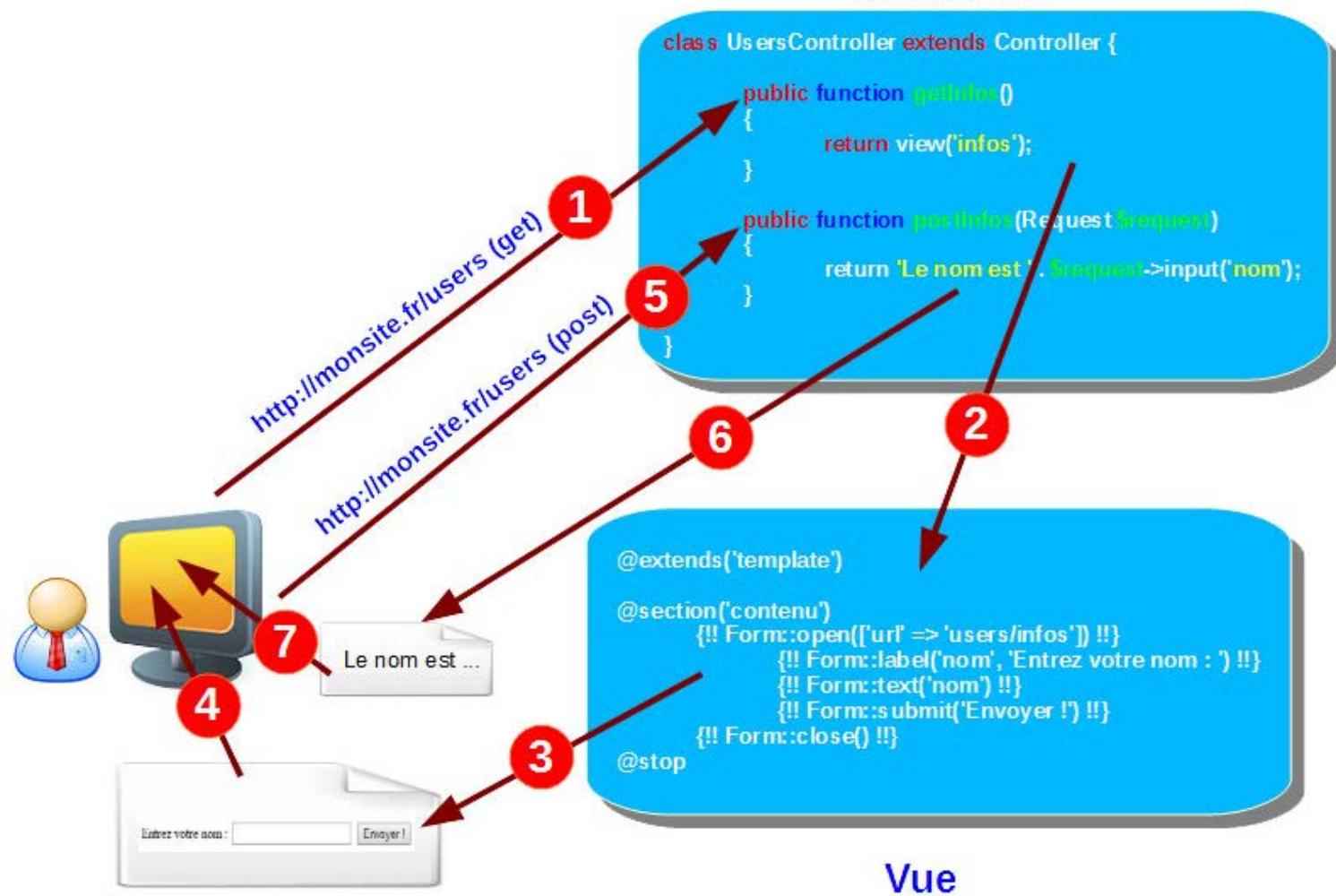
1. la méthode `getInfos` qui reçoit l'url `http://monsite.fr/users` avec le verbe "get" et qui retourne le formulaire,
2. la méthode `postInfos` qui reçoit l'url `http://monsite.fr/users` avec le verbe "post" et qui traite les entrées.

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 :

Contrôleur



Vue

Le scénario en action

- (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.



Il existe la façade `Request` qui permet aussi de récupérer les entrées :

```
<?php
Request::input('nom')
```

php

La protection CSRF

On a vu que le formulaire généré par Laravel comporte un contrôle caché avec une valeur particulière :

```
<input name="_token" type="hidden" value="pV1vWldUqFDfYsBjKag43C3NvzbIC01HtMnv9BpI">
```

html

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 un token aléatoire associé au formulaire de telle sorte qu'à la soumission ce token est vérifié 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` :



Le middleware CSRF

Pour tester l'efficacité de cette vérification essayez un envoi de formulaire sans le token en modifiant ainsi la vue (adaptez la valeur de l'action selon votre contexte) :

```
@extends('template')

@section('contenu')
    <form method="POST" action="http://monsite.fr/users" accept-charset="UTF-8">
        <label for="nom">Entrez votre nom : </label>
        <input name="nom" type="text" id="nom">
        <input type="submit" value="Envoyer !">
    </form>
@endsection
```

Vous tomberez sur cette erreur à la soumission :



En résumé

Erreur dans la vérification du token

- Laravel permet de créer des routes avec différents verbes : get, post...
- Un formulaire peut facilement être créé avec la classe Form.
- Les entrées du client sont récupérées dans la requête.
- On peut se prémunir contre les attaques CSRF, cette défense est mise en place automatiquement par Laravel.

Découvrez le framework PHP Laravel

□ □ □ □ □ □

□ 15 heures □ Moyenne

Licence □ □ □ □



La validation

□ Connectez-vous ou [inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Nous avons vu dans le chapitre précédent un scénario mettant en œuvre un formulaire. Nous n'avons imposé aucune contrainte sur les valeurs transmises. Dans une application réelle, il est toujours nécessaire de vérifier que ces valeurs correspondent à ce qu'on attend. Par exemple un nom doit comporter uniquement des caractères alphabétiques et avoir une longueur maximale, une adresse email doit correspondre à un certain format.

Il faut donc mettre en place des règles de validation. En général on procède à une première validation côté client pour éviter de faire des allers-retours avec le serveur. Mais quelle que soit la pertinence de cette validation côté client elle n'exonère pas d'une validation côté serveur.



On ne doit jamais faire confiance à des données qui arrivent sur le serveur.

Dans l'exemple de ce chapitre je ne prévois pas de validation côté client, d'une part ce n'est pas mon propos, d'autre part elle masquerait la validation côté serveur pour les tests.

Scénario et routes

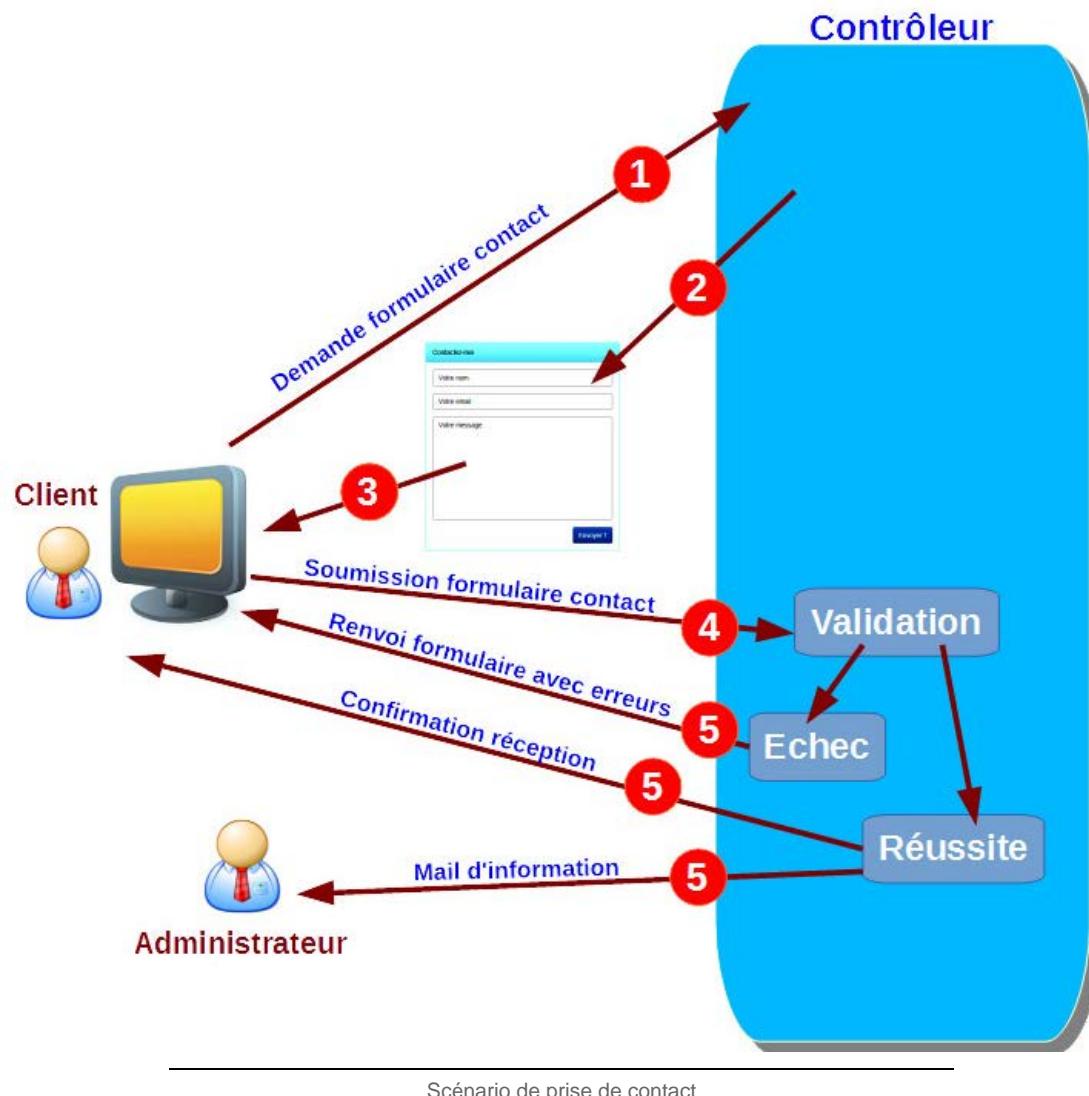
Voici le scénario que je vous propose pour ce chapitre :

□ Les bases de Laravel

- 1. Présentation générale
- 2. Installation et organisation
- 3. Le routage et les façades
- 4. Les réponses
- 5. Les contrôleurs
- 6. Les entrées
- ▶ 7. La validation
- 8. Configuration et session
- 9. L'injection de dépendance
- Quiz : Quiz 1
- Activité : Créer un site de sondages

Accéder au forum





Scénario de prise de contact

1. le client demande le formulaire de contact,
2. le contrôleur génère le formulaire,
3. le contrôleur envoie le formulaire,
4. le client remplit le formulaire et le soumet,
5. le contrôleur teste la validité des informations et là on a deux possibilités :
 - en cas d'échec on renvoie le formulaire au client en l'informant des erreurs et en conservant ses entrées correctes,
 - en cas de réussite on envoie un message de confirmation au client et un email à l'administrateur.

Routes

On va donc avoir besoin de 2 routes :

```
<?php
Route::get('contact', 'ContactController@getForm');
Route::post('contact', 'ContactController@postForm');
```

On aura une seule url (avec verbe "get" pour demander le formulaire et verbe "post" pour le soumettre)

:

<http://monsite.fr/contact>

Les vues

Le template

Pour ce chapitre je vais créer un template réaliste avec l'utilisation de Bootstrap pour alléger le code.

Voici le code de ce template (`resources/views/template.blade.php`) :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Mon joli site</title>
        {!!
Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css') !!}
{!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css') !!}
{!--[if lt IE 9]
    {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.js') }}
    {{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js') }}
}
        <![endif]-->
        <style> textarea { resize: none; } </style>
    </head>
    <body>
        @yield('contenu')
    </body>
</html>
```

abap

Je rappelle que dans Blade il y a deux syntaxes : la double accolade permet de sécuriser le code en échappant les caractères spéciaux alors que l'utilisation de `{!! !!}` n'effectue aucun traitement et doit donc être utilisé avec prudence.

Pour la génération des liens vers les librairies CSS j'ai utilisé la classe `Html` avec sa méthode `style`. Il y a un certain nombre de méthodes pratiques dans cette classe que nous découvrirons petit à petit.

J'ai prévu l'emplacement `@yield` nommé "contenu" pour recevoir les pages du site, pour notre exemple on aura seulement la page de contact et celle de la confirmation.

La vue de contact

La vue de contact va contenir essentiellement un formulaire (`resources/views/contact.blade.php`) :

```
@extends('template')

@section('contenu')
    <br>
    <div class="col-sm-offset-3 col-sm-6">
        <div class="panel panel-info">
            <div class="panel-heading">Contactez-moi</div>
            <div class="panel-body">
                {!! Form::open(['url' => 'contact']) !!}
                <div class="form-group {!! $errors->has('nom') ? 'has-error' :
'' !!}">
                    {!! Form::text('nom', null, ['class' => 'form-control',
'placeholder' => 'Votre nom']) !!}
                    {!! $errors->first('nom', '<small class="help-
block">:message</small>') !!}
                </div>
                <div class="form-group {!! $errors->has('email') ? 'has-error'
: '' !!}">
                    {!! Form::email('email', null, ['class' => 'form-
control', 'placeholder' => 'Votre email']) !!}
                    {!! $errors->first('email', '<small class="help-
block">:message</small>') !!}
                </div>
                <div class="form-group {!! $errors->has('texte') ? 'has-error'
: '' !!}">
                    {!! Form::textarea ('texte', null, ['class' => 'form-
control', 'placeholder' => 'Votre message']) !!}
                    {!! $errors->first('texte', '<small class="help-
block">:message</small>') !!}
                </div>
                {!! Form::submit('Envoyer !', ['class' => 'btn btn-info pull-
right']) !!}
                {!! Form::close() !!}
            </div>
        </div>
    </div>
```

php

```
</div>
@endsection
```

Cette vue étend le template vu ci-dessus et renseigne la section "contenu". Je ne commente pas la mise en forme spécifique à Bootstrap. Le formulaire est généré avec la classe `Form` que nous avons déjà vue au chapitre précédent.

La structure des méthodes pour générer les contrôles du formulaire est toujours la même. Prenons par exemple l'email :

```
<?php
{!! Form::email('email', null, ['class' => 'form-control', 'placeholder' => 'Votre email']) !!}
```

On veut un élément de formulaire (`Form`) de type "email" avec le nom "email" avec une valeur nulle et avec les attributs "class" et "placeholder" en précisant leur valeur.

Le bouton de soumission ne comporte évidemment pas de valeur et on a donc un paramètre de moins pour lui.

En cas de réception du formulaire suite à des erreurs on reçoit une variable `$errors` qui contient un tableau avec comme clés les noms des contrôles et comme valeurs les textes identifiant les erreurs.



La variable `$errors` est générée systématiquement pour toutes les vues.

C'est pour cela que je teste la présence d'une erreur pour chaque contrôle en ajustant le style et en affichant le texte de l'erreur si nécessaire avec la méthode `first` :

```
{!! $errors->first('nom', '<small class="help-block">:message</small>') !!}
```

S'il n'y a aucune erreur rien n'est renvoyé et donc rien n'est affiché, sinon on récupère la première (`first`) et on respecte le format imposé.

Au départ le formulaire se présente ainsi :

Le formulaire vierge

Après une soumission et renvoi avec des erreurs il peut se présenter ainsi :

Contactez-moi

Durand

durand@fr

The email must be a valid email address.

Votre message

The texte field is required.

Envoyer !

Le formulaire avec des erreurs

□ Par défaut les messages sont en anglais. Pour avoir ces textes en français vous devez récupérer les fichiers sur [ce site](#). Placez le dossier "fr" et son contenu dans le dossier `resources/lang`. Ensuite changez cette ligne dans le fichier `config/app.php` :
'locale' => 'fr',
Vous devriez avoir votre Laravel en français :

Contactez-moi

Durand

durand@fr

Le champ E-mail doit être une adresse email valide.

Votre message

Le champ texte est obligatoire.

Envoyer !

Les messages en français

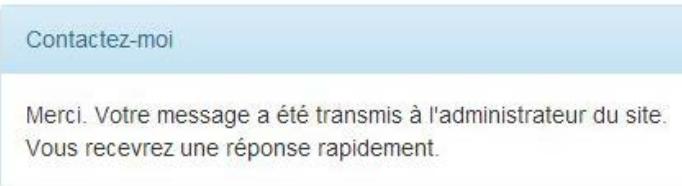
La vue de confirmation

Pour la vue de confirmation (`resources/views/confirm.blade.php`) le code est plus simple et on utilise évidemment le même template :

```
@extends('template')

@section('contenu')
<br>
    <div class="col-sm-offset-3 col-sm-6">
        <div class="panel panel-info">
            <div class="panel-heading">Contactez-moi</div>
            <div class="panel-body">
                Merci. Votre message a été transmis à l'administrateur du site. Vous
recevez une réponse rapidement.
            </div>
        </div>
    </div>
@endsection
```

Ce qui donne cette apparence :



La confirmation

La vue de l'email pour l'administrateur

Nous avons également besoin de créer une vue pour construire l'email pour l'administrateur (`resources/views/email_contact.blade.php`) :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <meta charset="utf-8">
    </head>
    <body>
        <h2>Prise de contact sur mon beau site</h2>
        <p>Réception d'une prise de contact avec les éléments suivants :</p>
        <ul>
            <li><strong>Nom</strong> : {{ $nom }}</li>
            <li><strong>Email</strong> : {{ $email }}</li>
            <li><strong>Message</strong> : {{ $texte }}</li>
        </ul>
    </body>
</html>
```

On doit transmettre à cette vue les entrées de l'utilisateur.

La requête de formulaire

Laravel possède un outil de commande nommé Artisan que nous avons déjà utilisé et qui permet d'effectuer de nombreuses opérations.

 Pour les personnes allergiques à la console j'ai créé [un package](#) qui permet d'avoir les commandes d'Artisan avec une interface conviviale.

On accède à Artisan au niveau de la console :

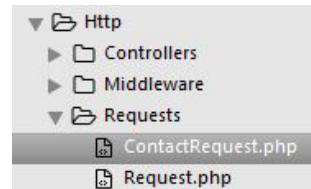
abap

```
php artisan
```

Parmi les nombreuses possibilités nous allons utiliser `make:request` pour créer notre requête de formulaire :

```
php artisan make:request ContactRequest
Request created successfully.
```

La requête est créée ici :



La requête de formulaire

Voyons le code généré :

```
<?php
namespace App\Http\Requests;

use App\Http\Requests\Request;

class ContactRequest extends Request {

    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //
        ];
    }
}
```

La classe générée comporte 2 méthodes :

- **authorize** : pour effectuer un contrôle de sécurité éventuel sur l'identité ou les droits de l'émetteur,
- **rules** : pour les règles de validation.

On va arranger le code pour notre cas :

```
<?php
namespace App\Http\Requests;

class ContactRequest extends Request {

    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
```

```

        return true;
    }

/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
{
    return [
        'nom' => 'required|min:5|max:20|alpha',
        'email' => 'required|email',
        'texte' => 'required|max:250'
    ];
}

}

```

Au niveau de la méthode `rules`, on retourne un tableau qui contient des clés qui correspondent aux champs du formulaire. Vous retrouvez le nom, l'email et le texte. Les valeurs contiennent les règles de validation. Comme il y en a chaque fois plusieurs elles sont séparées par le signe "|". Voyons les différentes règles prévues :

- **required** : une valeur est requise, donc le champ ne doit pas être vide,
- **min** : nombre minimum de caractères, par exemple `min:5` signifie "au minimum 5 caractères",
- **max** : c'est l'inverse de "min" avec un nombre maximum de caractères,
- **alpha** : on n'accepte que les caractères alphabétiques,
- **email** : la valeur doit être une adresse email valide.

Au niveau de la méthode `authorize` je me suis contenté de renvoyer `true` parce que nous ne ferons pas de contrôle supplémentaire.

 Il existe une règle `between:min,max` qui résume l'utilisation des deux règles `max:value` et `min:value`. Vous pouvez trouver toutes les règles disponibles [dans la documentation](#). Vous verrez que la liste est longue !

Le contrôleur

Voici maintenant le code du contrôleur `ContactController` :

```

<?php
namespace App\Http\Controllers;

use Mail;
use App\Http\Requests\ContactRequest;

class ContactController extends Controller {

    public function getForm()
    {
        return view('contact');
    }

    public function postForm(ContactRequest $request)
    {
        Mail::send('email_contact', $request->all(), function($message)
        {
            $message->to('monadresse@free.fr')->subject('Contact');
        });

        return view('confirm');
    }
}

```

La méthode `getForm` ne présente aucune nouveauté par rapport à ce qu'on a vu au chapitre précédent. On se contente de renvoyer la vue `contact` qui comporte le formulaire.

La méthode `postForm` nécessite quelques commentaires. Vous remarquez le paramètres de type `ContactRequest`. On injecte dans la méthode une instance de la classe `ContactRequest` que l'on a précédemment créée. Laravel permet ce genre d'injection de dépendance au niveau d'une méthode. Je reviendrai en détail dans un prochain chapitre sur cette possibilité.

Si la validation échoue parce qu'une règle n'est pas respectée c'est la classe `ContactRequest` qui s'occupe de tout, elle renvoie le formulaire en complétant les contrôles qui étaient corrects et crée une variable `$errors` pour transmettre les messages d'erreurs qu'on utilise dans la vue. Vous n'avez rien d'autre à faire !

J'ai utilisé la façade `Mail`, j'aurais pu aussi injecter dans la méthode un objet comme je l'ai fait pour la requête de formulaire. Nous verrons dans un chapitre ultérieur l'injection de dépendance avec d'autres cas d'utilisation.

Envoyer un email

En cas de réussite de la validation on envoie un email à l'administrateur :

```
<?php
Mail::send('email_contact', $request->all(), function($message)
{
    $message->to('monadresse@free.fr')->subject('Contact');
});
```

php

Laravel utilise SwiftMailer pour accomplir cette tâche. La syntaxe est simple : on utilise la classe des mails (`Mail`) pour envoyer (`send`) un email avec le contenu de la vue "email_contact" en transmettant à cette vue les éléments de la requête (`$request`) en précisant qu'on les veut tous (`all`). On envoie cet email à (`to`) "monadresse@free.fr" avec comme sujet (`subject`) "Contact".

Pour que l'envoi des emails fonctionne il faut renseigner les éléments suivants dans le fichier `.env` :

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.free.fr
MAIL_PORT=25
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

text

Ainsi que cette ligne dans `config/mail.php`

```
'from' => ['address' => 'moi@free.fr', 'name' => 'Administrateur'],
```

php

Ici la configuration correspond à mon hébergeur (free) pour mes tests en local. Si vous avez un hébergeur différent vous devrez évidemment adapter ces valeurs. De même si vous mettez un site en production sur un serveur.

Si tout se passe bien vous devez recevoir un email de ce style :

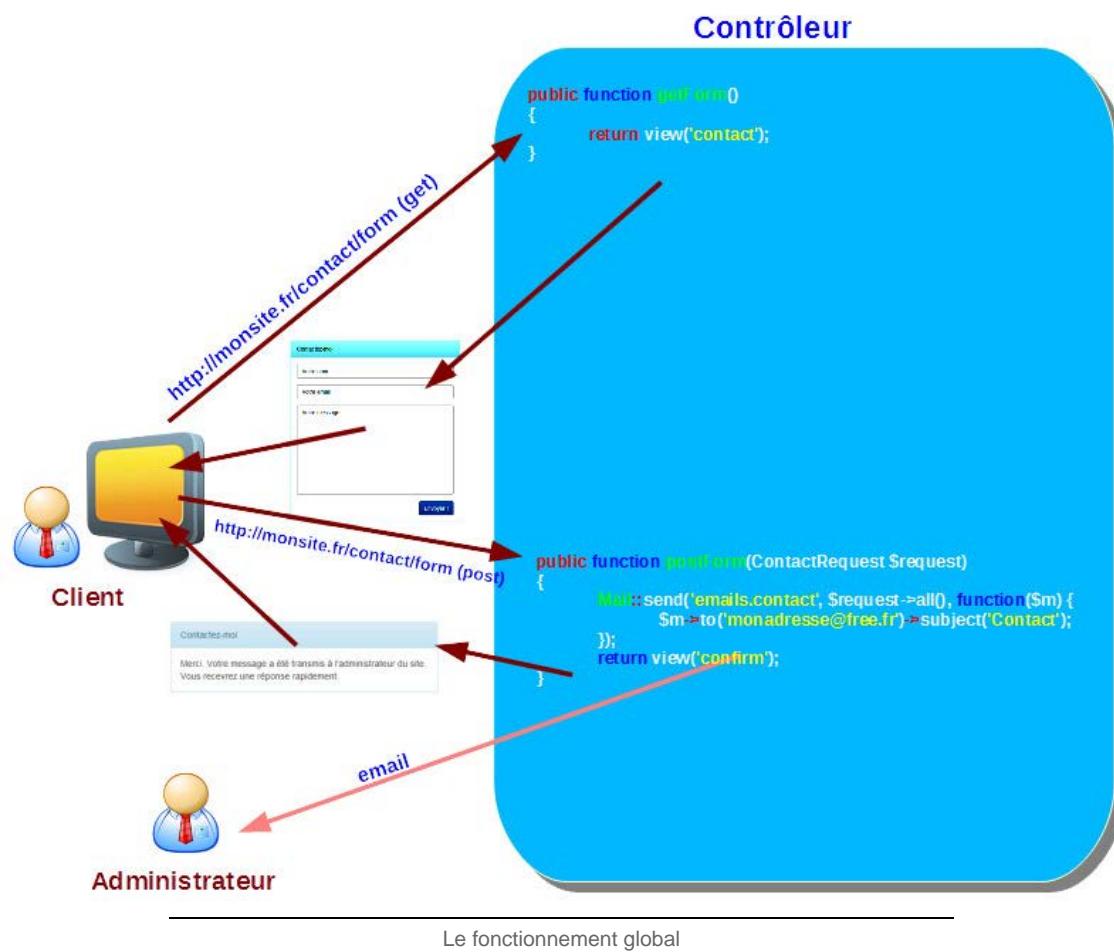
Prise de contact sur mon beau site

Réception d'une prise de contact avec les éléments suivants :

- Nom : Durant
- Email : durand@chezmoi.fr
- Message : Je trouve votre site très laid...

L'email pour l'administrateur

Voici une illustration globale du fonctionnement :



En résumé

- La validation est une étape essentielle de vérification des entrées du client.
- On dispose de nombreuses règles de validation.
- Le validateur génère des erreurs explicites à afficher au client.
- Pour avoir les textes des erreurs en Français il faut aller chercher les traductions et les placer dans le bon dossier.
- Laravel permet l'envoi simple d'email.



Les entrées



Configuration et session

L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

[S'inscrire](#)[Se connecter](#)

Accueil ▶ Cours ▶ Découvrez le framework PHP Laravel ▶ Configuration et session

Découvrez le framework PHP Laravel

15 heures Moyenne



Configuration et session

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

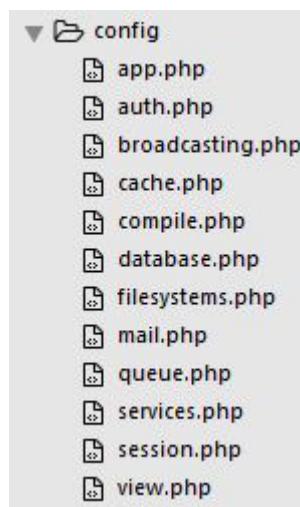
Dans ce chapitre nous verrons la configuration et la gestion des sessions avec un exemple simple d'envoi et d'enregistrement de fichiers images dans un dossier à partir d'un formulaire.

La configuration et les sessions

La configuration

Tout ce qui concerne la configuration de Laravel se trouve dans le dossier

config :



Le dossier de configuration

On a déjà eu l'occasion d'intervenir sur le fichier de configuration des emails. Les fichiers de configuration contiennent en fait juste un tableau avec des clés et des valeurs. Par exemple pour les vues :

<?php

php

```
'paths' => [
    realpath(base_path('resources/views'))
],
```

On a la clé "paths" et la valeur "realpath(base_path('resources/views'))". Pour récupérer une valeur il suffit d'utiliser sa clé avec la façade `Config` et la méthode `get` :

```
<?php
Config::get('view.paths');
```

php

On utilise le nom du fichier (`view`) et le nom de la clé (`paths`) séparés par un point.

Il existe aussi un helper pour simplifier la syntaxe :

```
<?php
config('view.paths');
```

text

On peut de la même façon fixer une valeur :

```
<?php
Config::set('view.paths', [base_path() . '/mes_vues']);
```

php

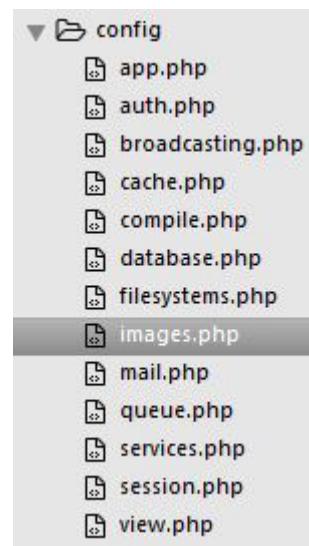
Si je fais effectivement cela mes vues, au lieu d'être cherchées dans le dossier `resources/views` seront cherchées dans le dossier `mes_vues`.

On peut aussi fixer une valeur avec l'helper en passant un tableau comme paramètre :

```
<?php
config('view.paths' => [base_path() . '/mes_vues']);
```

php

Vous pouvez évidemment créer vos propres fichiers de configuration. Pour l'exemple de ce chapitre on va avoir besoin justement d'utiliser une configuration. Comme notre application doit enregistrer des fichiers d'images dans un dossier il faut définir l'emplacement et le nom de ce dossier de destination. On va donc créer un fichier `images.php` :



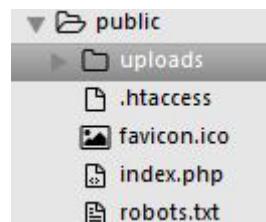
Le fichier de configuration des images

Dans ce fichier on va définir le chemin du dossier :

```
<?php
return ['path' => 'uploads'];
```

php

Tant qu'à faire on crée aussi le dossier correspondant :



Le dossier pour les images

Les sessions

La façade `Session` de Laravel permet une gestion simplifiée des sessions. Vous pouvez ainsi créer une variable de session :

```
<?php
Session::put('clef', 'valeur');
```

Il existe aussi un helper, ainsi on obtient le même résultat avec :

```
<?php
session(['clef' => 'valeur']);
```

Vous pouvez aussi récupérer une valeur à partir de sa clé :

```
<?php
$valeur = Session::get('clef');
```

Ce qui donne avec l'helper :

```
<?php
$valeur = session('clef');
```

Il est souvent utile (ça sera le cas pour notre exemple) de savoir si une certaine clé est présente en session :

```
<?php
if (Session::has('error'))
```

Ou avec l'helper :

```
<?php
if (session()->has('error'))
```

Ces informations demeurent pour le même client à travers ses requêtes. Laravel s'occupe de ces informations, on se contente de lui indiquer un couple clé-valeur et il s'occupe de tout. C'est ce que nous allons voir dans ce chapitre.

 Ce ne sont là que les méthodes de base pour les sessions utiles pour notre exemple, vous trouverez tous les renseignements complémentaires [dans la documentation](#).

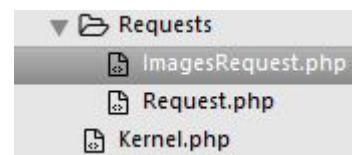
La requête de formulaire

Nous allons encore avoir besoin d'une requête de formulaire pour la validation. Comme nous l'avons déjà vu nous utilisons la commande d'artisan pour la créer :

```
php artisan make:request ImagesRequest
Request created successfully.
```

php

On trouve le fichier bien rangé :



La requête de formulaire pour les images

On complète ainsi le code :

```
<?php

namespace App\Http\Requests;

use App\Http\Requests\Request;

class ImagesRequest extends Request {

    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return ['image' => 'required|image'];
    }
}
```

Seulement deux règles pour le champ `image` :

- le champ est obligatoire (`required`),
- ce doit être une image (`image`)

. Maintenant notre validation est prête.

Les routes et le contrôleur

On va avoir besoin de deux routes :

```
<?php
Route::get('photo', 'PhotoController@getForm');
Route::post('photo', 'PhotoController@postForm');
```

Notre contrôleur s'appelle `PhotoController`, voici son code :

```
<?php
```

php

php

php

```

namespace App\Http\Controllers;

use App\Http\Requests\ImagesRequest;

class PhotoController extends Controller
{

    public function getForm()
    {
        return view('photo');
    }

    public function postForm(ImagesRequest $request)
    {
        $image = $request->file('image');

        if($image->isValid())
        {
            $chemin = config('images.path');

            $extension = $image->getClientOriginalExtension();

            do {
                $nom = str_random(10) . '.' . $extension;
            } while(file_exists($chemin . '/' . $nom));

            if($image->move($chemin, $nom)) {
                return view('photo_ok');
            }
        }

        return redirect('photo')
            ->with('error', 'Désolé mais votre image ne peut pas être envoyée !');
    }
}

```

Donc au niveau des urls :

- `http://monsite.fr/photo` avec le verbe `get` pour la demande du formulaire,
- `http://monsite.fr/photo` avec le verbe `post` pour la soumission du formulaire et l'envoi du fichier image associé.

En ce qui concerne le traitement de la soumission, vous remarquez qu'on récupère le chemin du dossier d'enregistrement qu'on a prévu dans la configuration :

```
<?php
$chemin = config('images.path');
```

php

Pour récupérer le fichier envoyé j'ai utilisé la requête et la méthode `file` :

```
<?php
$image = $request->file('image');
```

php

Pour récupérer l'extension originelle on utilise la méthode `getClientOriginalExtension` qui est l'une des méthodes de `Symfony\Component\HttpFoundation\File`, comme la méthode `isValid` qui nous permet de vérifier la validité du fichier.

On génère un nom aléatoire avec l'helper `str_random` en définissant 10 caractères et on vérifie que le nom n'est

pas déjà pris (ce qui ne serait vraiment pas de chance !).

Enfin on enregistre l'image avec la méthode `move`. Si tout se passe bien on retourne la vue `photo_ok`. Sinon on redirige (`redirect`) vers l'url `photo` en prévoyant dans la session (`with`) une variable `error` avec la valeur "Désolé mais votre image ne peut pas être envoyée !".

Les vues

On va utiliser le template des chapitres précédents (`resources/views/template.blade.php`) :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Mon joli site</title>
        {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css') !!}
        {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css') !!}
    <!--[if lt IE 9]>
        {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.js') }}
        {{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js') }}
    <![endif]-->
    <style> textarea { resize: none; } </style>
</head>
<body>
    @yield('contenu')
</body>
</html>
```

Voici la vue pour le formulaire (`resources/views/photo.blade.php`) :

```
@extends('template')

@section('contenu')
    <br>
    <div class="col-sm-offset-4 col-sm-4">
        <div class="panel panel-info">
            <div class="panel-heading">Envoi d'une photo</div>
            <div class="panel-body">
                @if(session()->has('error'))
                    <div class="alert alert-danger">{!! session('error') !!}</div>
                @endif
                {!! Form::open(['url' => 'photo', 'files' => true]) !!}
                <div class="form-group {!! $errors->has('image') ? 'has-error' : '' !!}>
                    {!! Form::file('image', ['class' => 'form-control']) !!}
                    {!! $errors->first('image', '<small class="help-block">:message</small>') !!}
                </div>
                {!! Form::submit('Envoyer !', ['class' => 'btn btn-info pull-right']) !!}
                {!! Form::close() !!}
            </div>
        </div>
    @endsection
```

Avec cet aspect :

Envoi d'une photo

Choisissez un fichier Aucun fichier choisi

Envoyer !

Le formulaire

Cette vue fait apparaître une possibilité de Blade qu'on n'avait pas encore rencontrée, celle d'utiliser des conditions :

```
@if(session()->has('error'))
    <div class="alert alert-danger">{{ session('error') }}</div>
@endif
```

html

Ici on teste avec **@if** la présence de la clé "error" dans la session. Si cette clé est présente alors on fait apparaître une barre d'alerte avec le texte contenu en session, récupéré avec l'helper **session**.

Remarquez aussi comment est créé le formulaire :

```
Form::open(['url' => 'photo', 'files' => true])
```

html

Le fait d'ajouter l'attribut **files** avec la valeur **true** va avoir pour effet de faire apparaître le type mime nécessaire pour associer un fichier lors de la soumission :

```
enctype="multipart/form-data"
```

html

En cas d'erreur de validation le message est affiché et la bordure du champ devient rouge :

Envoi d'une photo

Choisissez un fichier Aucun fichier choisi

Le champ image est obligatoire.

Envoyer !

Erreur de validation

En cas de problème dans la mémorisation du fichier on retourne un message par l'intermédiaire de la session et on l'affiche dans une barre d'alerte :

Envoi d'une photo

Désolé mais votre image ne peut pas être envoyée !

Choisissez un fichier Aucun fichier choisi

Envoyer !

Erreurs de mémorisation du fichier

Et voici la vue pour la confirmation en retour ([REDACTED]) :

```
html  
@extends('template')  
  
@section('contenu')  
    <br>  
    <div class="col-sm-offset-3 col-sm-6">  
        <div class="panel panel-info">  
            <div class="panel-heading">Envoi d'une photo</div>  
            <div class="panel-body">  
                Merci. Votre photo à bien été reçue et enregistrée.  
            </div>  
        </div>  
    </div>  
@endsection
```

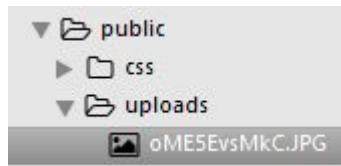
Avec cet aspect :

Envoi d'une photo

Merci. Votre photo à bien été reçue et enregistrée.

La vue de confirmation

Et on retrouve normalement le fichier bien rangé dans le dossier prévu :



Le fichier dans le dossier prévu

Voici le schéma de fonctionnement :

Contrôleur

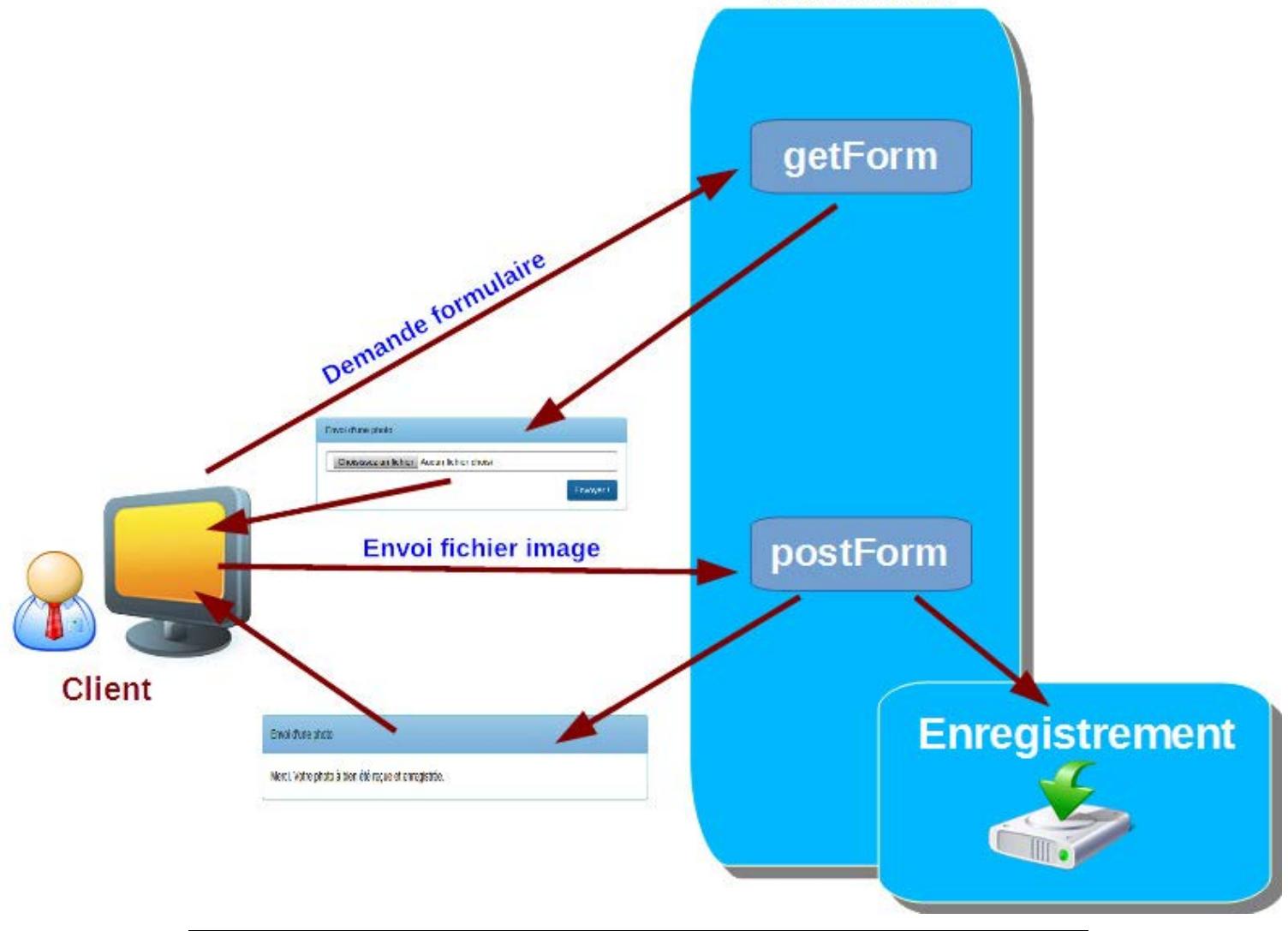


Schéma de fonctionnement

En résumé

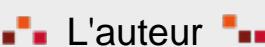
- Les fichiers de configuration permettent de mémoriser facilement des ensembles clé-valeur et sont gérés par la façade `Config` ou l'helper `config`.
- Les sessions permettent de mémoriser des informations concernant un client et sont facilement manipulables avec la façade `Session` ou avec l'helper `session`.



La validation



L'injection de dépendance

 L'auteur

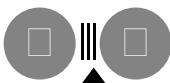
Maurice Chavelli

[S'inscrire](#)[Se connecter](#)

Accueil ▶ Cours ▶ Découvrez le framework PHP Laravel ▶ L'injection de dépendance

Découvrez le framework PHP Laravel

15 heures Moyenne



L'injection de dépendance

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Dans ce chapitre nous allons reprendre l'exemple précédent de l'envoi de photos en nous posant des questions d'organisation du code. Laravel ce n'est pas seulement un framework pratique, c'est aussi un style de programmation. Il vaut mieux évoquer ce style le plus tôt possible dans l'apprentissage pour prendre rapidement les bonnes habitudes.

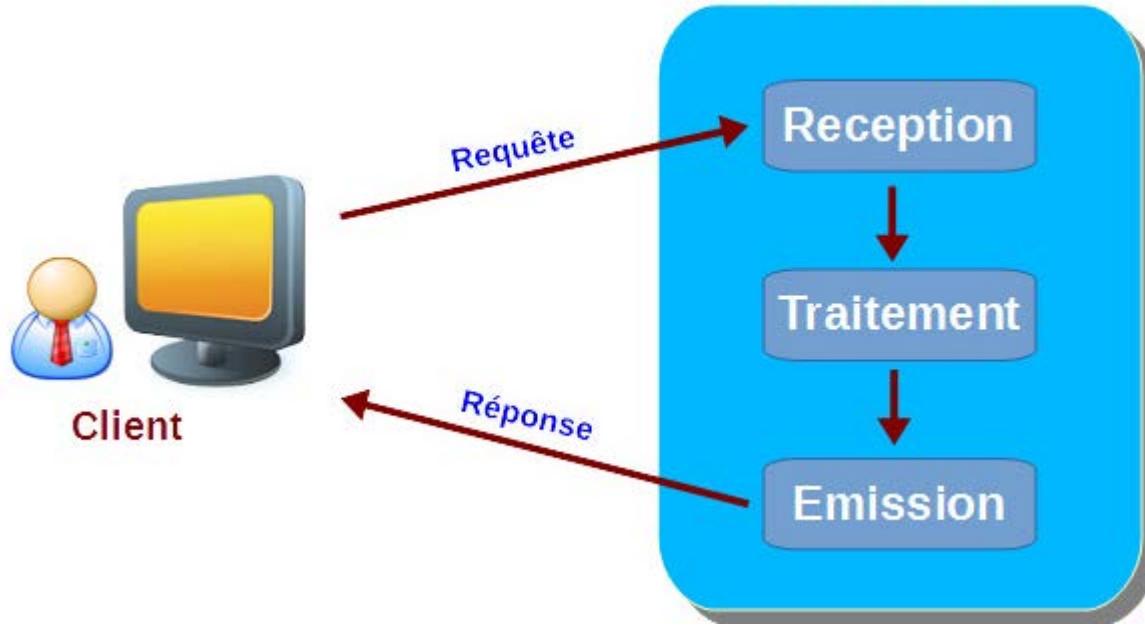
Vous pouvez très bien créer un site complet dans le fichier des routes, vous pouvez aussi vous contenter de contrôleurs pour effectuer tous les traitements nécessaires. Je vous propose une autre approche, plus en accord avec ce que nous offre Laravel.

Le problème et sa solution

Le problème

Je vous ai déjà dit qu'un contrôleur a pour mission de réceptionner les requêtes et d'envoyer les réponses. Entre les deux il y a évidemment du traitement à effectuer, la réponse doit se construire, parfois c'est très simple, parfois plus long et délicat. Mais globalement nous avons pour un contrôleur ce fonctionnement :

Contrôleur



Fonctionnement d'un contrôleur

Reprenons la méthode `postForm` de notre contrôleur `PhotoController` du précédent chapitre :

```

<?php
public function postForm(ImagesRequest $request)
{
    $image = $request->file('image');

    if($image->isValid())
    {
        $chemin = config('images.path');

        $extension = $image->getClientOriginalExtension();

        do {
            $nom = str_random(10) . '.' . $extension;
        } while(file_exists($chemin . '/' . $nom));

        if($image->move($chemin, $nom)) {
            return view('photo_ok');
        }
    }

    return redirect('photo')
        ->with('error','Désolé mais votre image ne peut pas être envoyée !');
}
  
```

Qu'avons-nous comme traitement ? On récupère les références de l'image transmise, on récupère le dossier de destination dans la configuration, on trouve l'extension du fichier image, on génère un nom et enfin on enregistre l'image.

La question est : est-ce qu'un contrôleur doit savoir comment s'effectue ce traitement ? Si vous avez plusieurs contrôleurs dans votre application qui doivent effectuer le même traitement vous allez multiplier cette mise en place. Imaginez que vous avez ensuite envie de modifier l'enregistrement des images, par exemple en les mettant dans une base de données, vous allez devoir retoucher le code de tous vos contrôleurs ! La répétition de code n'est jamais une bonne chose, une saine règle de programmation veut qu'on commence à se poser des questions sur l'organisation du code dès qu'on fait des copies.

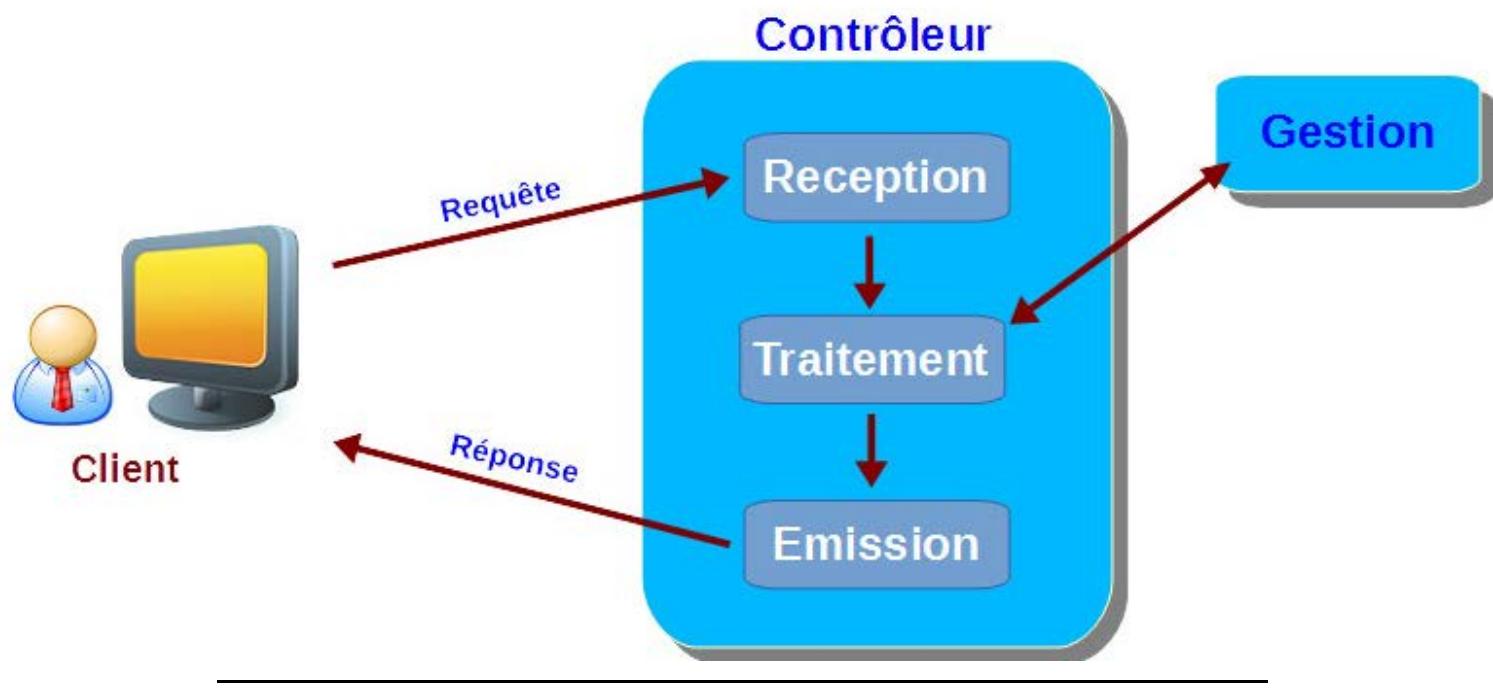


Cette préconisation est connue sous l'acronyme DRY ("Don't Repeat Yourself").

Un autre élément à prendre en compte aussi est la testabilité des classes. Nous verrons cet aspect important du développement trop souvent négligé. Pour qu'une classe soit testable il faut que sa mission soit simple et parfaitement identifiée et il ne faut pas qu'elle soit étroitement liée avec une autre classe. En effet cette dépendance rend les tests plus difficiles.

La solution

Alors quelle est la solution ? L'injection de dépendance ! Voyons de quoi il s'agit. Regardez ce schéma :



L'injection de la gestion

Une nouvelle classe entre en jeu pour la gestion, c'est elle qui est effectivement chargée du traitement, le contrôleur fait juste appel à ses méthodes. Mais comment cette classe est-elle injectée dans le contrôleur ? Voici le code du contrôleur modifié :

```
<?php
namespace App\Http\Controllers;

use App\Http\Requests\ImagesRequest;
use App\Gestion\PhotoGestion;

class PhotoController extends Controller {

    public function getForm()
    {
        return view('photo');
    }

    public function postForm(
        ImagesRequest $request,
        PhotoGestion $photogestion)
    {

        if($photogestion->save($request->file('image'))) {
```

php

```

        return view('photo_ok');

    }

    return redirect('photo')
        ->with('error', 'Désolé mais votre image ne peut pas être envoyée !');

}

}

```

Vous remarquez qu'au niveau de la méthode `postForm` il y a un nouveau paramètre de type `App\Gestion\PhotoGestion`. On utilise la méthode `save` de la classe ainsi injectée pour faire le traitement.

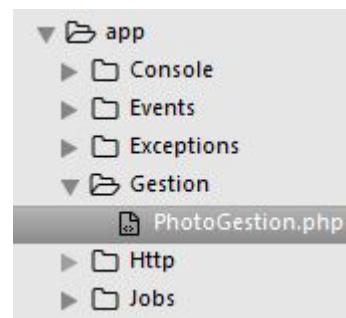
De cette façon le contrôleur ignore totalement comment se fait la gestion, il sait juste que la classe `PhotoGestion` sait la faire. Il se contente d'utiliser la méthode de cette classe qui est "injectée". Maintenant vous vous demandez sans doute comment cette classe est injectée là, en d'autres termes comment et où est créée cette instance. Eh bien Laravel est assez malin pour le faire lui-même.

PHP est très tolérant sur les types des variables. Lorsque vous en déclarez une vous n'êtes pas obligé de préciser que c'est un string ou un array. PHP devine le type selon la valeur affectée. Il en est de même pour les paramètres des fonctions. Mais personne ne vous empêche de déclarer un type comme je l'ai fait ici pour le paramètre de la méthode (malheureusement pour le moment PHP ne reconnaît que les tableaux et les classes). C'est même indispensable pour que Laravel sache quelle classe est concernée. Étant donné que je déclare le type, Laravel est capable de créer une instance de ce type et de l'injecter dans le contrôleur.

 Pour trouver la classe Laravel utilise l'introspection (reflexion en anglais) de PHP qui permet d'inspecter le code en cours d'exécution. Elle permet aussi de manipuler du code et donc de créer par exemple un objet d'une certaine classe. Vous pouvez trouver tous les renseignements dans [le manuel PHP](#).

La gestion

Maintenant qu'on a dit au contrôleur qu'une classe s'occupe de la gestion il nous faut la créer. Pour bien organiser notre application on crée un nouveau dossier et on place notre classe dedans :



Le dossier de gestion

Le codage ne pose aucun problème parce qu'il est identique à ce qu'on avait dans le contrôleur :

```

<?php
namespace App\Gestion;

class PhotoGestion
{
    public function save($image)

```

php

```

    {
        if($image->isValid())
        {
            $chemin = config('images.path');
            $extension = $image->getClientOriginalExtension();

            do {
                $nom = str_random(10) . '.' . $extension;
            } while(file_exists($chemin . '/' . $nom));

            return $image->move($chemin, $nom);
        }

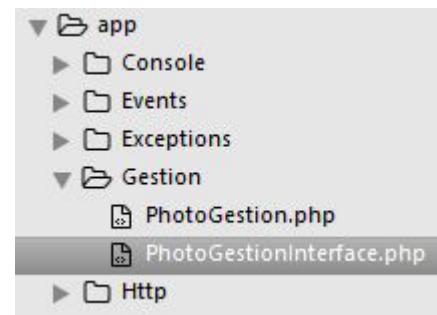
        return false;
    }
}

```

 Attention à ne pas oublier les espaces de noms !

Maintenant notre code est parfaitement organisé et facile à maintenir et à tester.

Mais allons un peu plus loin, créons une interface pour notre classe :



L'interface pour la classe PhotoGestion

Avec ce code :

```

<?php

namespace App\Gestion;

interface PhotoGestionInterface
{

    public function save($image);
}

```

Il suffit ensuite d'en informer la classe PhotoGestion :

```

<?php
class PhotoGestion implements PhotoGestionInterface

```

Ce qui serait bien maintenant serait dans notre contrôleur de référencier l'interface :

```

<?php

namespace App\Http\Controllers;

```

```
use App\Http\Requests\ImagesRequest;
use App\Gestion\PhotoGestionInterface;

class PhotoController extends Controller
{
    public function getForm()
    {
        return view('photo');
    }

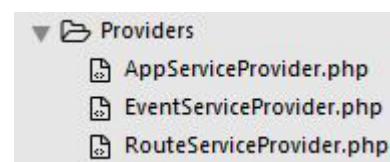
    public function postForm(
        ImagesRequest $request,
        PhotoGestionInterface $photogestion)
    {
        if($photogestion->save($request->file('image'))) {
            return view('photo_ok');
        }
        return redirect('photo')
            ->with('error', 'Désolé mais votre image ne peut pas être envoyée !');
    }
}
```

Le souci c'est que Laravel n'arrive pas à deviner la classe à instancier à partir de cette interface :



Comment s'en sortir ?

Lorsque j'ai présenté la structure de Laravel j'ai mentionné la présence de providers :



Les providers

A quoi sert un provider ? Tout simplement à procéder à des initialisations : événements, middlewares, et surtout des liaisons de dépendance. Laravel possède un conteneur de dépendances qui constitue le cœur de son fonctionnement. C'est grâce à ce conteneur qu'on va pouvoir établir une liaison entre une interface et une classe.

Ouvrez le fichier `app\Providers\AppServiceProvider.php` et ajoutez cette ligne de code :

```
<?php
public function register()
{
    ...

    $this->app->bind(
        'App\Gestion\PhotoGestionInterface',
        'App\Gestion\PhotoGestion'
    );
}
```

La méthode `register` est activée au démarrage de l'application, c'est l'endroit idéal pour notre liaison. Ici on dit à l'application (`app`) d'établir une liaison (`bind`) entre l'interface `App\Gestion\PhotoGestionInterface` et la classe `App\Gestion\PhotoGestion`. Ainsi chaque fois qu'on se référera à cette interface dans une injection Laravel saura quelle classe instancier. Si on veut changer la classe de gestion il suffit de modifier le code du provider.

Maintenant notre application fonctionne :).



Si vous obtenez un message d'erreur vous disant que l'interface ne peut pas être instanciée lancez la commande :

```
php artisan clear-compiled
```

text

En résumé

- Un contrôleur doit déléguer toute tâche qui ne relève pas de sa compétence.
- L'injection de dépendance permet de bien séparer les tâches, de simplifier la maintenance du code et les tests unitaires.
- Les providers permettent de faire des initialisations, en particulier des liaisons de dépendance entre interfaces et classes.


[Configuration et session](#)

[Quiz : Quiz 1](#)


L'auteur

[Maurice Chavelli](#)

Développeur retraité qui se consacre à l'enseignement des technologies du web.

Découvrez le framework PHP Laravel



15 heures

Moyenne

Licence 

Migrations et modèles

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

À partir de ce chapitre, il serait souhaitable que vous installiez une barre de débogage. La plus utile est celle proposée par [barryvdh](#). Suivez les indications fournies pour l'installation, ça vous fera un bon exercice :).

Dans ce chapitre nous allons commencer à aborder les bases de données. C'est un vaste sujet auquel Laravel apporte des réponses efficaces. Nous allons commencer par voir les migrations et les modèles.

Pour ce chapitre je vais encore prendre un exemple simple en imaginant un formulaire destiné à l'inscription à une lettre d'information. On va se contenter d'envoyer un email et de mémoriser cet email dans une table d'une base de données.

Les migrations

Une migration permet de créer et de mettre à jour un schéma de base de données. Autrement dit, vous pouvez créer des tables, des colonnes dans ces tables, en supprimer, créer des index... Tout ce qui concerne la maintenance de vos tables peut être pris en charge par cet outil.

La configuration de la base

Vous devez dans un premier temps avoir une base de données. Laravel permet de gérer les bases de type MySQL, Postgres, SQLite et SQL Server. Je ferai tous les exemples avec MySQL mais tout le code sera aussi valable pour les autres types de bases.

Il faut indiquer où se trouve votre base, son nom, le nom de l'utilisateur, le mot de passe dans le fichier de configuration `.env` :

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

text

Ici nous avons les valeurs par défaut à l'installation de Laravel.

Voici par exemple mes réglages pour ma base de test MySQL nommée "tuto" avec MySQL en local non sécurisé :

```
DB_DATABASE=tuto
DB_USERNAME=root
DB_PASSWORD=
```

html

Les bases de données

- ▶ 1. **Migrations et modèles**
- 2. Les ressources (1/2)
- 3. Les ressources (2/2) et les erreurs
- 4. L'authentification
- 5. La relation 1:n
- 6. La relation n:n
- 7. Les commandes et les assistants
- 8. Query Builder
- Quiz : Quiz 2
- Activité : Construisez un site de sondages avec une base de données

[Accéder au forum](#)



Artisan

Laravel a un outil en ligne de commande : **artisan**. Nous avons déjà utilisé cet outil qui permet de faire beaucoup de choses, vous avez un aperçu des commandes en entrant :

```
php artisan
```

Vous avez une longue liste. Pour ce chapitre nous allons nous intéresser uniquement à celles qui concernent les migrations :

```
make
...
make:migration      Create a new migration file
...
migrate
migrate:install    Create the migration repository
migrate:refresh    Reset and re-run all migrations
migrate:reset      Rollback all database migrations
migrate:rollback   Rollback the last database migration
migrate:status     Show the status of each migration
```

Installer la migration

On va commencer par installer la migration :

```
php artisan migrate:install
Migration table created successfully.
```

Si vous regardez l'effet dans votre base vous allez voir qu'une table a été créée :

| Table | Action |
|------------|---------------------|
| migrations | Afficher Structure |
| 1 table | Somme |

La table des migrations

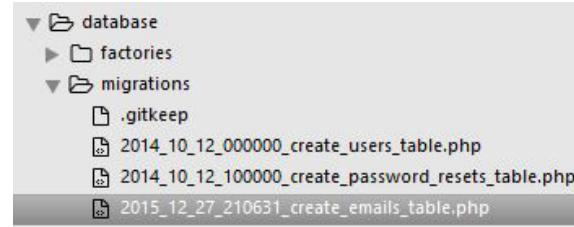
C'est dans cette table que seront mémorisées toutes vos actions au niveau du schéma de la base.

Créer la migration

La deuxième étape consiste à créer la migration pour notre table :

```
php artisan make:migration create_emails_table
Created Migration: 2015_01_15_121123_create_emails_table
```

Si vous regardez maintenant dans le dossier `database/migrations` vous trouvez un fichier du genre `2015_12_27_210631_create_emails_table.php` (la partie numérique qui inclut la date sera évidemment différente pour vous) :



La migration créée



Mais il y a déjà des migrations présentes, à quoi servent-elles ?

Il y a déjà effectivement 2 migrations présentes :

- **table users** : c'est une migration de base pour créer une table des utilisateurs,
- **table password_resets** : c'est une migration liée à la précédente qui permet de gérer le renouvellement des mots de passe en toute sécurité.

Nous nous intéresserons à ces migrations lorsque nous verrons l'authentification dans un chapitre ultérieur. Comme nous n'allons pas avoir besoin immédiatement de ces migrations le

mieux est de les supprimer pour le moment pour éviter de créer des tables inutiles.

Voici le contenu de la migration que nous venons de créer :

```
<?php  
  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Database\Migrations\Migration;  
  
class CreateEmailsTable extends Migration {  
  
    /**  
     * Run the migrations.  
     *  
     * @return void  
     */  
    public function up()  
    {  
        //  
    }  
  
    /**  
     * Reverse the migrations.  
     *  
     * @return void  
     */  
    public function down()  
    {  
        //  
    }  
}
```

php

On dispose dans cette classe de deux fonctions :

- **up** : ici on mettra le code de création
- **down** : ici on mettra le code de suppression

On veut créer une table "emails" avec un id auto-incrémenté et un champ "email" de type texte, et de longueur 100. Voilà le code correspondant :

```
<?php  
public function up()  
{  
    Schema::create('emails', function(Blueprint $table) {  
        $table->increments('id');  
        $table->string('email', 100);  
    });  
}
```

php

On demande au constructeur de schéma (`Schema`) de créer (`create`) la table "emails". Dans la fonction anonyme on définit ce qu'on veut pour la table :

- une colonne "id" auto-incrémentée qui sera ainsi la clé primaire de la table,
- une colonne "email" de type string et de longueur 100.

Pour la méthode `down` on va juste supprimer la table avec un `drop` :

```
<?php  
public function down()  
{  
    Schema::drop('emails');  
}
```

php

Notre migration est maintenant créée.

Utiliser la migration

On va maintenant lancer la migration (utilisation de la méthode `up` de la migration) :

```
php artisan migrate  
Migrated: 2015_12_27_210631_create_emails_table
```

Si on regarde maintenant dans la base on trouve la table "emails" avec ces deux colonnes :

| # | Nom | Type | Interclassement | Attributs | Null | Défaut | Extra |
|---|--------------|--------------|-----------------|-----------|------|--------|----------------|
| 1 | id | int(10) | | UNSIGNED | Non | Aucune | AUTO_INCREMENT |
| 2 | email | varchar(100) | utf8_unicode_ci | | Non | Aucune | |

La table "emails"

Si vous avez fait une erreur vous pouvez revenir en arrière avec un `rollback` qui annule la dernière migration effectuée (utilisation de la méthode `down` de la migration) :

```
php artisan migrate:rollback
Rolled back: 2015_12_27_210631_create_emails_table
```

La table a maintenant été supprimée de la base. Comme on va avoir besoin de cette table on relance la migration. On peut aussi effectuer un rafraîchissement de toutes les migrations avec la commande `refresh` (rollback de toutes les migrations et nouveau lancement de toutes les migrations).

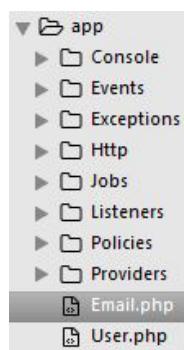
Eloquent

Laravel propose un ORM (acronyme de object-relational mapping ou en bon Français un mappage objet-relationnel) très performant. De quoi s'agit-il ? Tout simplement que tous les éléments de la base de données ont une représentation sous forme d'objets manipulables. Quel intérêt ? Tout simplement de simplifier grandement les opérations sur la base comme nous allons le voir dans toute cette partie du cours.

Avec Eloquent une table est représentée par une classe qui étend la classe Model. Pour notre table `emails` on va à nouveau utiliser Artisan pour la création du modèle :

```
php artisan make:model Email
```

On trouve le fichier ici :



Le nouveau modèle pour les emails

Avec cette trame de base :

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Email extends Model
{
    //
}
```

On va compléter ainsi le code :

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
```

```
class Email extends Model
{
    protected $table = 'emails';

    public $timestamps = false;

}
```

Vous voyez c'est tout simple ! On renseigne le nom de la table associée au modèle. D'autre part par défaut Eloquent tient à jour des colonnes **created_at** et **updated_at** dans la table. Comme nous ne les avons pas prévues, pour désactiver cette action on est obligé de mettre à `false` la propriété `timestamps`. On va mettre ce modèle directement dans le dossier `app`, il est évident que pour une application réelle on organisera les dossiers pour bien ranger tous nos fichiers mais pour le moment on va faire simple.

 Si on ne renseigne pas la propriété `$table` Laravel va déduire le nom de la table à partir du nom du modèle. Autrement dit dans notre cas on pourrait éviter cette ligne de code.

Nous allons voir maintenant comment utiliser cette classe en construisant notre petite application.

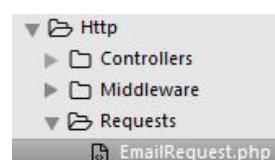
La validation

Pour la validation on va encore créer une requête de formulaire :

```
php artisan make:request EmailRequest
Request created successfully.
```

text

On trouve la requête dans son dossier :



La requête de formulaire

La voici avec le code complété :

```
<?php

namespace App\Http\Requests;

use App\Http\Requests\Request;

class EmailRequest extends Request
{

    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return ['email' => 'required|email|unique:emails'];
    }
}
```

php

On a 3 règles :

- **required** : le champ est requis,
- **email** : on doit avoir une adresse email valide,
- **unique** : l'email ne doit pas déjà exister (`unique`) dans la table `emails` (on sous-entend qu'il s'agit de la colonne `email`).

Remarquez la puissance de la troisième règle : Eloquent va vérifier que notre email n'existe pas déjà dans la table !

Les routes

On va avoir deux routes :

```
<?php  
Route::get('email', 'EmailController@getForm');  
Route::post('email', ['uses' => 'EmailController@postForm', 'as' => 'storeEmail']);
```

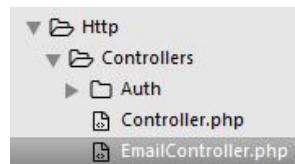
Remarquez que la seconde route est nommée (**storeEmail**).

L'url de base sera donc :

```
http://monsite.fr/email
```

Le contrôleur

On va créer un contrôleur `EmailController` :



Le contrôleur EmailController

Le code du contrôleur reprend l'essentiel de ce que nous avons vu dans les chapitres précédents en utilisant à nouveau la validation injectée :

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Email;  
use App\Http\Requests\EmailRequest;  
  
class EmailController extends Controller  
{  
  
    public function getForm()  
    {  
        return view('email');  
    }  
  
    public function postForm(EmailRequest $request)  
    {  
        $email = new Email;  
        $email->email = $request->input('email');  
        $email->save();  
  
        return view('email_ok');  
    }  
}
```

La nouveauté réside uniquement dans l'utilisation du modèle :

```
<?php  
$email = new Email;  
$email->email = $request->input('email');  
$email->save();
```

Ici on crée une nouvelle instance de `Email`. On affecte l'attribut `email` avec la valeur de l'entrée. Enfin on demande au modèle d'enregistrer cette ligne effectivement dans la table (`save`).

Les vues

On va utiliser le même template que dans les précédents chapitres. Voici la vue pour le formulaire (`resources/views/email.blade.php`) :

```
@extends('template')

@section('contenu')
    <br>
    <div class="col-sm-offset-4 col-sm-4">
        <div class="panel panel-info">
            <div class="panel-heading">Inscription à la lettre d'information</div>
            <div class="panel-body">
                {!! Form::open(['route' => 'storeEmail']) !!}
                <div class="form-group {!! $errors->has('email') ? 'has-error' : '' !!}">
                    {!! Form::email('email', null, array('class' => 'form-control', 'placeholder' => 'Entrez votre email')) !!}
                    {!! $errors->first('email', '<small class="help-block">:message</small>') !!}
                </div>
                {!! Form::submit('Envoyer !', ['class' => 'btn btn-info pull-right']) !!}
                {!! Form::close() !!}
            </div>
        </div>
    </div>
@endsection
```

Cette vue ne présente aucune nouveauté pour vous si ce n'est l'utilisation du nom de la route, elle répond à l'url (avec le verbe get) :

<http://monsite.fr/email>

L'aspect est le suivant :

Le formulaire

Voici maintenant la vue de confirmation (`resources/views/email_ok.blade.php`) :

```
@extends('template')

@section('contenu')
    <br>
    <div class="col-sm-offset-3 col-sm-6">
        <div class="panel panel-info">
            <div class="panel-heading">Inscription à la lettre d'information</div>
            <div class="panel-body">
                Merci. Votre adresse a bien été prise en compte.
            </div>
        </div>
    </div>
@endsection
```

Avec cet aspect :

Inscription à la lettre d'information

Merci. Votre adresse a bien été prise en compte.

La confirmation

Le fonctionnement

Voyons maintenant si tout se passe bien. Je soumets une adresse :

Inscription à la lettre d'information

monadresse@chezmoi.com

Envoyer !

Soumission d'une adresse

Je reçois la confirmation :

Inscription à la lettre d'information

Merci. Votre adresse a bien été prise en compte.

La confirmation

Je regarde dans la base :

| id | email |
|----|------------------------|
| 1 | monadresse@chezmoi.com |

L'adresse dans la base

Je soumets la même adresse :

monadresse@chezmoi.com

La valeur du champ E-mail est déjà utilisée.

Soumission d'une adresse existante

Voyons un peu les requêtes générées par Eloquent avec par exemple la soumission de l'adresse `toto@gui.com` (vous les trouvez à la rubrique Queries de la barre de débogage) :

sql

```
select count(*) as aggregate from `emails` where `email` = 'toto@gui.com'  
insert into `emails` (`email`) values ('toto@gui.com')
```

La première requête est destinée à tester la présence éventuelle de l'adresse dans la table pour répondre à la règle "unique". La seconde insère l'enregistrement dans la table. Vous voyez qu'Eloquent vous simplifie la tâche, vous n'avez pas besoin d'écrire les requêtes SQL, il le fait pour vous. Vous vous contentez de manipuler un objet.

N'hésitez pas à regarder les informations de la barre de débogage, vous y trouverez de précieux renseignements sur les requêtes (HTTP et SQL), les vues utilisées, les routes, les délais, les

exceptions générées... Vous avez aussi un historique en cliquant sur la petite image de dossier :



Ouvrir l'historique de la barre

Organisation du code

Maintenant posons-nous à nouveau la question de l'organisation du code. Dans le contrôleur nous avons mis la gestion du modèle :

```
<?php
$email = new Email;
$email->email = $request->input('email');
$email->save();
```

php

Autrement dit nous avons lié de façon étroite le contrôleur et le modèle. Supposons que nous faisons des modifications dans notre base de données et que nous plaçons l'email dans une autre table. Nous devrons évidemment intervenir dans le code du contrôleur pour tenir compte de cette modification.

Vous pouvez évidemment considérer que c'est peu probable, que la modification du code n'est pas très importante... Mais ici on a une application très simple, dans une situation réelle l'utilisation des modèles sont nombreux et alors la question devient plus pertinente.

Première version

Dans ce cours je m'efforce de vous entraîner à prendre de bonnes habitudes. Plutôt que d'instancier directement une classe dans une autre il vaut mieux une injection et laisser faire le conteneur.

Regardez cette nouvelle version du contrôleur :

```
<?php
namespace App\Http\Controllers;

use App\Email;
use App\Http\Requests\EmailRequest;

class EmailController extends Controller
{

    public function getForm()
    {
        return view('email');
    }

    public function postForm(
        EmailRequest $request,
        Email $email)
    {
        $email->email = $request->input('email');
        $email->save();

        return view('email_ok');
    }
}
```

php

Maintenant le modèle est injecté dans la méthode, c'est plus élégant et efficace. Si jamais vous changez de modèle vous n'avez plus qu'un changement de code limité sur le contrôleur. Mais ce n'est pas encore parfait.

Seconde version

Dans l'idéal on veut que notre contrôleur ne soit pas du tout concerné par un changement dans la gestion des modèles. Voici une façon de procéder :

```
<?php
namespace App\Http\Controllers;

use App\Http\Requests\EmailRequest;
```

php

```
use App\Repositories\EmailRepository;

class EmailController extends Controller
{

    public function getForm()
    {
        return view('email');
    }

    public function postForm(
        EmailRequest $request,
        EmailRepository $emailRepository)
    {
        $emailRepository->save($request->input('email'));

        return view('email_ok');
    }

}
```

Maintenant j'injecte une classe de gestion qui possède la méthode `save`. Voici le contrat avec une interface (`app/Repositories/EmailRepositoryInterface`) :

```
<?php

namespace App\Repositories;

interface EmailRepositoryInterface
{
    public function save($mail);
}
```

Et voici la classe qui implémente cette interface (`app/Repositories/EmailRepository`) :

```
<?php

namespace App\Repositories;

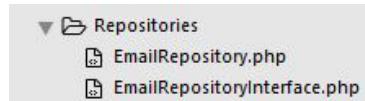
use App\Email;

class EmailRepository implements EmailRepositoryInterface
{
    protected $email;

    public function __construct(Email $email)
    {
        $this->email = $email;
    }

    public function save($mail)
    {
        $this->email->email = $mail;
        $this->email->save();
    }
}
```

Le modèle est injecté dans cette classe. Je l'ai injecté dans le constructeur pour généraliser la démarche en imaginant qu'on créera d'autres méthodes que l'on peut regrouper ici pour gérer les enregistrements. Le code est maintenant parfaitement organisé, facile à modifier et à tester.



La gestion

 Le Design Pattern **Repository** est un des plus répandus. Il permet de gérer la persistance des informations.

Troisième version

On peut enfin, comme on l'a déjà vu, référencer l'interface plutôt que la classe mais dans ce cas il faut informer le conteneur de la dépendance. Modifiez ainsi le fichier `app/Http/Providers/AppServiceProvider.php` :

```
<?php  
public function register()  
{  
    ...  
  
    $this->app->bind(  
        'App\Repositories\EmailRepositoryInterface',  
        'App\Repositories\EmailRepository'  
    );  
}
```

Et finalement le contrôleur :

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Requests\EmailRequest;  
use App\Repositories\EmailRepositoryInterface;  
  
class EmailController extends Controller  
{  
  
    public function getForm()  
    {  
        return view('email');  
    }  
  
    public function postForm(  
        EmailRequest $request,  
        EmailRepositoryInterface $emailRepository)  
    {  
        $emailRepository->save($request->input('email'));  
  
        return view('email_ok');  
    }  
}
```

Maintenant, étant donné que le conteneur sait quelle classe instancier à partir de l'interface passée en paramètre, vous avez un code propre et facile à maintenir et à tester. Si vous changez d'avis sur la manière de stocker les emails il vous suffit de décider d'instancier une autre classe à partir de l'interface, tant que le contrat passé avec le contrôleur ne change pas !

En résumé

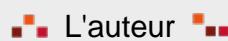
- La base de données doit être configurée pour fonctionner avec Laravel.
- Les migrations permettent d'intervenir sur le schéma des tables de la base.
- Eloquent permet une représentation des tables sous forme d'objets pour simplifier les manipulations des enregistrements.
- Il est judicieux de prévoir la gestion du modèle dans une classe injectée dans le contrôleur.
- La barre de débogage donne de précieux renseignements sur les requêtes.



Activité : Créer un site de sondages

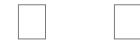


Les ressources (1/2)



L'auteur

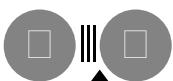
Maurice Chavelli

[S'inscrire](#)[Se connecter](#)

Accueil ▶ Cours ▶ Découvrez le framework PHP Laravel ▶ Les ressources (1/2)

Découvrez le framework PHP Laravel

15 heures Moyenne



Les ressources (1/2)

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Dans ce chapitre nous allons commencer à étudier les ressources qui permettent de créer un contrôleur RESTful adapté à une ressource. Comme exemple pratique nous allons prendre le cas d'une table d'utilisateurs, une situation qui se retrouve dans la plupart des applications.

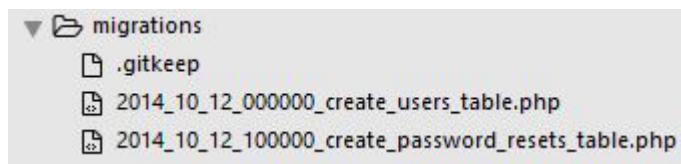
Les données

La migration

A partir d'un Laravel vierge on va commencer par configurer la base comme on l'a vu au chapitre précédent et par installer la migration pour créer la table des migrations dans la base :

```
php artisan migrate:install
Migration table created successfully.
```

Lorsqu'on installe Laravel on se retrouve avec deux migrations déjà présentes :



Les migrations présentes à l'installation.

Pour le moment on va garder seulement la première migration qui concerne la création de la table des utilisateurs. On va juste un peu la modifier ainsi :

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
```

php

```

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function(Blueprint $table)
        {
            $table->increments('id');
            $table->string('name')->unique();
            $table->string('email')->unique();
            $table->string('password', 60);
            $table->boolean('admin')->default(false);
            $table->rememberToken();
            $table->timestamps();
        });
    }

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

```

On aura les champs :

- **id** : entier auto-incrémenté qui sera la clé primaire de la table,
- **name** : texte (varchar 255) pour le nom (**unique**),
- **email** : texte (varchar 255) pour l'email (**unique**),
- **password** : texte de 60 caractères pour le mot de passe (ce qui ne signifie pas que le mot de passe pourra avoir 60 caractères parce qu'on va stocker ici une version cryptée de ce mot de passe),
- **admin** : valeur booléenne pour indiquer s'il s'agit d'un administrateur avec **false** comme valeur par défaut,
- **remember_token** : texte de 100 caractères qui sert pour l'authentification que nous verrons dans un chapitre ultérieur,
- **created_at** et **updated_at** créés par la méthode **timestamps**,

Ensuite on lance la migration :

```

php artisan migrate
Migrated: 2014_10_12_000000_create_users_table

```

Et on doit se retrouver avec la table créée :

| # | Nom | Type | Interclassement | Attributs | Null | Défaut | Extra |
|---|---|--------------|-----------------|-----------|------|--------|----------------|
| 1 | id  | int(10) | | UNSIGNED | Non | Aucune | AUTO_INCREMENT |
| 2 | name | varchar(255) | utf8_unicode_ci | | Non | Aucune | |
| 3 | email | varchar(255) | utf8_unicode_ci | | Non | Aucune | |
| 4 | password | varchar(60) | utf8_unicode_ci | | Non | Aucune | |
| 5 | admin | tinyint(1) | | | Non | 0 | |
| 6 | remember_token | varchar(100) | utf8_unicode_ci | | Oui | NULL | |
| 7 | created_at | timestamp | | | Non | Aucune | |
| 8 | updated_at | timestamp | | | Non | Aucune | |

La table des utilisateurs

Remarquez que le modèle `User` est déjà présent dans le dossier `app` quand vous installez Laravel parce qu'il est un peu particulier comme nous le verrons lorsque nous parlerons de l'authentification. Pour le moment nous allons nous satisfaire du fait qu'il existe déjà sans nous soucier de son contenu.

Une ressource

Création

On va maintenant créer une ressource avec Artisan :

```
php artisan make:controller UserController --resource
Controller created successfully. Vous trouvez comme résultat le contrôleur
app/Http/Controllers/UserController
:
```

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
}
```

abap

php

```
/*
public function create()
{
    //
}

/**
 * Store a newly created resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    //
}

/**
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    //
}

/**
 * Show the form for editing the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function edit($id)
{
    //
}

/**
 * Update the specified resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id)
{
    //
}

/**
 * Remove the specified resource from storage.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function destroy($id)
{
    //
}
}
```

Les 7 méthodes créées couvrent la gestion complète des utilisateurs :

1. **index** : pour afficher la liste des utilisateurs,
2. **create** : pour envoyer le formulaire pour la création d'un nouvel utilisateur,
3. **store** : pour créer un nouvel utilisateur,
4. **show** : pour afficher les données d'un utilisateur,
5. **edit** : pour envoyer le formulaire pour la modification d'un utilisateur,
6. **update** : pour modifier les données d'un utilisateur,
7. **destroy** : pour supprimer un utilisateur.

Les routes

Pour créer toutes les routes il suffit de cette unique ligne de code :

```
<?php  
Route::resource('user', 'UserController');
```

php

Pour connaître les routes ainsi créées on va encore utiliser Artisan :

```
php artisan route:list
```

text

| | | | |
|-----------|--------------------|--------------|---|
| GET HEAD | / user | user.index | App\Http\Controllers\UserController@index |
| POST | / user | user.store | App\Http\Controllers\UserController@store |
| GET HEAD | / user/create | user.create | App\Http\Controllers\UserController@create |
| PUT PATCH | / user/{user} | user.update | App\Http\Controllers\UserController@update |
| GET HEAD | / user/{user} | user.show | App\Http\Controllers\UserController@show |
| DELETE | / user/{user} | user.destroy | App\Http\Controllers\UserController@destroy |
| GET HEAD | / user/{user}/edit | user.edit | App\Http\Controllers\UserController@edit |

Les routes de la ressource

Vous trouvez 7 routes, avec chacune une url, qui pointent sur les 7 méthodes de notre contrôleur. Notez également que chaque route a aussi un nom qui peut être utilisé par exemple pour une redirection. Nous allons à présent considérer chacune de ces routes et créer la gestion des données, les vues, et le code nécessaire au niveau du contrôleur.

Le contrôleur

Il nous faut à présent coder le contrôleur. Pour rester dans la démarche de bonne organisation des classes je vais continuer à limiter le contrôleur à la réception des requêtes et l'envoi des réponses. Il va donc falloir injecter :

- la validation
- la gestion des données.

Pour la validation on va avoir deux cas :

- la création d'un utilisateur avec vérification de l'unicité du nom et de l'email et la conformité du mot de passe,
- la modification d'un utilisateur, avec la même vérification d'unicité mais en excluant l'enregistrement en cours de modification. D'autre part nous n'allons pas inclure le mot de passe dans cette modification.

On va donc avoir besoin de deux requêtes de formulaire : une pour la création et l'autre pour la modification.

Pour la gestion une seule classe `UserRepository` suffira.

Si on considère ces injections voici le code du contrôleur :

```
<?php

namespace App\Http\Controllers;

use App\Http\Requests\UserCreateRequest;
use App\Http\Requests\UserUpdateRequest;

use App\Repositories\UserRepository;

use Illuminate\Http\Request;

class UserController extends Controller
{

    protected $userRepository;

    protected $nbrPerPage = 4;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function index()
    {
        $users = $this->userRepository->getPaginate($this->nbrPerPage);
        $links = $users->render();

        return view('index', compact('users', 'links'));
    }

    public function create()
    {
        return view('create');
    }

    public function store(UserCreateRequest $request)
    {
        $user = $this->userRepository->store($request->all());

        return redirect('user')->withOk("L'utilisateur " . $user->name . " a été créé.");
    }

    public function show($id)
    {
        $user = $this->userRepository->getById($id);

        return view('show', compact('user'));
    }

    public function edit($id)
    {
        $user = $this->userRepository->getById($id);

        return view('edit', compact('user'));
    }

    public function update(UserUpdateRequest $request, $id)
    {
        $this->userRepository->update($id, $request->all());

        return redirect('user')->withOk("L'utilisateur " . $request->input('name') . " a été modifié.");
    }
}
```

```

    public function destroy($id)
    {
        $this->userRepository->destroy($id);

        return back();
    }

}

```

Nous allons évidemment analyser tout ça dans le détail mais globalement vous voyez que le code est très épuré :

- réception de la requête,
- délégation du traitement si nécessaire (validation et gestion),
- envoi de la réponse.

La validation

Création d'un utilisateur

On va créer une requête de formulaire pour la création d'un utilisateur. Je ne vous réitère pas la démarche de création avec artisan parce que nous l'avons déjà vue plusieurs fois. Voici le code de la classe :

```

<?php
namespace App\Http\Requests;

use App\Http\Requests\Request;

class UserCreateRequest extends Request
{

    public function authorize()
    {
        return true;
    }

    public function rules()
    {
        return [
            'name' => 'required|max:255|unique:users',
            'email' => 'required|email|max:255|unique:users',
            'password' => 'required|confirmed|min:6'
        ];
    }
}

```

Avec ces règles :

- **name** : requis, longueur maximale de 255 caractères, et unique dans la table users,
- **email** : requis, adresse valide, longueur maximale de 255 caractères, et unique dans la table users,
- **password** : requis, longueur minimale de 6 caractères et doit correspondre à ce qui est entré dans le champ de confirmation du mot de passe.

Modification d'un utilisateur

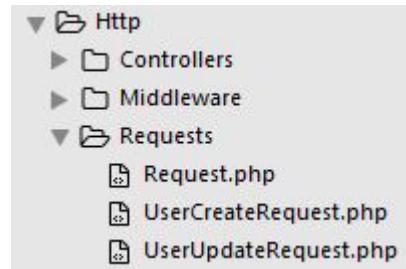
Pour la modification d'un utilisateur nous allons avoir un petit souci. En effet on veut conserver l'unicité du nom et de l'email dans la base, il est donc judicieux de prévoir une règle "unique". Mais comme les valeurs du nom et de l'email existants sont

déjà dans la base on va avoir un échec de la validation en cas de non modification de l'une de ces deux valeurs, ce qui est fortement probable. Comment nous en sortir ? Étant donné que nous disposons d'une fonction on peut effectuer tous les traitements que l'on veut. Voici alors la requête de formulaire pour la modification d'un utilisateur :

```
<?php  
  
namespace App\Http\Requests;  
  
use App\Http\Requests\Request;  
  
class UserUpdateRequest extends Request  
{  
  
    public function authorize()  
    {  
        return true;  
    }  
  
    public function rules()  
    {  
        $id = $this->user;  
        return [  
            'name' => 'required|max:255|unique:users,name,' . $id,  
            'email' => 'required|email|max:255|unique:users,email,' . $id  
        ];  
    }  
}
```

On récupère l'id de l'utilisateur dans l'url. Ensuite on utilise une possibilité d'exclusion de la règle "unique".

Vous devez donc avoir ces 2 fichiers pour la validation :



Les fichiers de la validation

Dans le prochain chapitre nous poursuivrons l'étude de cette ressource avec la gestion des données et les vues.

En résumé

- Lors de la migration le constructeur de schéma permet de fixer toutes les propriétés des champs.
- Une ressource dans Laravel est constituée d'un contrôleur comportant les 7 méthodes permettant une gestion complète.
- Les routes vers une ressource sont créées avec une simple ligne de code.
- Pour une ressource la validation est toujours différente entre la création et la modification et il faut adapter le code pour en tenir compte.



Migrations et modèles



Les ressources (2/2) et les erreurs



Découvrez le framework PHP Laravel



15 heures

Moyenne

Licence 

Les ressources (2/2) et les erreurs

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Dans ce chapitre nous allons poursuivre notre étude de la ressource pour les utilisateurs. Nous avons au chapitre précédent passé en revue la migration, les routes, le contrôleur et la validation. Il nous reste maintenant à voir la gestion des données, les vues, et aussi comment tout cela s'articule pour fonctionner.

Le gestionnaire de données (repository)

Nous avons vu que nous injectons un gestionnaire de données dans le contrôleur en plus des deux classes de validation :

```
<?php
public function __construct(UserRepository $userRepository)
{
    $this->userRepository = $userRepository;
}
```

Ce gestionnaire est chargé de toutes les actions au niveau de la table des utilisateurs. Voici son code ([app/Repositories/UserRepository.php](#)) :

```
<?php
namespace App\Repositories;

use App\User;

class UserRepository
{

    protected $user;

    public function __construct(User $user)
    {
        $this->user = $user;
    }

    private function save(User $user, Array $inputs)
    {
        $user->name = $inputs['name'];
        $user->email = $inputs['email'];
        $user->admin = isset($inputs['admin']);

        $user->save();
    }

    public function getPaginate($n)
    {
        return $this->user->paginate($n);
    }
}
```

Les bases de données

1. Migrations et modèles
 2. Les ressources (1/2)
 - ▶ **3. Les ressources (2/2) et les erreurs**
 4. L'authentification
 5. La relation 1:n
 6. La relation n:n
 7. Les commandes et les assistants
 8. Query Builder
- Quiz : Quiz 2
 Activité : Construisez un site de sondages avec une base de données

[Accéder au forum](#)



```

}

public function store(Array $inputs)
{
    $user = new $this->user;
    $user->password = bcrypt($inputs['password']);

    $this->save($user, $inputs);

    return $user;
}

public function getById($id)
{
    return $this->user->findOrFail($id);
}

public function update($id, Array $inputs)
{
    $this->save($this->getById($id), $inputs);
}

public function destroy($id)
{
    $this->getById($id)->delete();
}

}

```

Vous devez vous retrouver donc avec ce fichier dans le dossier `Repositories` :



Le fichier pour la gestion

Nous allons à présent analyser ses différentes actions.

getPaginate

Cette méthode contient juste une ligne :

```

<?php
public function getPaginate($n)
{
    return $this->user->paginate($n);
}

```

Elle est appelée depuis la méthode `index` du contrôleur :

```

<?php
public function index()
{
    $users = $this->userRepository->getPaginate($this->nbrPerPage);
    $links = $users->render();

    return view('index', compact('users', 'links'));
}

```

Cette méthode répond à l'url du type (avec le verbe `get`) :

```

http://monsite.fr/user

```

On utilise la méthode `paginate` du modèle qui permet de prendre seulement une partie des enregistrements pour faire une pagination. Ici j'ai prévu la valeur 4 (stockée dans la propriété `$nbrPerPage`), donc 4 enregistrements par page. Si vous regardez les requêtes SQL générées vous trouverez :

```

select count(*) as aggregate from `users`
select * from `users` limit 4 offset 0

```

La première requête sert à connaître le nombre total d'enregistrements dans la table users (donnée

nécessaire pour calculer la pagination) et la seconde à sélectionner les 4 premiers pour les afficher sur la première page. Évidemment pour la page 2 on aura la requête :

```
select * from `users` limit 4 offset 4
```

sql

Toujours 4 enregistrements mais avec un offset de 4 pour prendre les 4 enregistrements suivants. L'url sera alors :

```
.../user?page=2
```

Une fois que les enregistrements sont récupérés dans la table on les retourne au contrôleur. Celui-ci les transforme sous la forme d'un tableau avec comme clé "users". Ce tableau sera utilisé par la vue comme nous le verrons bientôt.

La pagination est incluse dans la variable `$links` pour être aussi envoyée dans la vue.

`getById`

Cette méthode contient juste une ligne :

```
<?php
public function getById($id)
{
    return $this->user->findOrFail($id);
}
```

php

La méthode `findOrFail` essaie de récupérer dans la table l'enregistrement dont on transmet l'`id`. Si elle n'y parvient pas elle génère une erreur d'exécution.

L'erreur générée est `ModelNotFoundException`. Nous allons bientôt voir comment on gère les erreurs.

Voici le genre de requête générée :

```
select * from `users` where `id` = '2' limit 1
```

sql

Dans le contrôleur nous avons deux méthodes qui utilisent `getById`.

`show`

Voici la méthode `show` du contrôleur :

```
<?php
public function show($id)
{
    $user = $this->userRepository->getById($id);

    return view('show', compact('user'));
}
```

php

Cette méthode répond à l'url (avec le verbe get) :

```
http://monsite.fr/user/n
```

html

Où `n` est l'id de l'utilisateur qu'on veut afficher.

Une fois que l'enregistrement est ainsi récupéré dans la table on le retourne sous la forme d'un tableau avec comme clé "user". Ce tableau sera utilisé par la vue, comme nous le verrons bientôt.

`edit`

La méthode `getById` est aussi utilisée par la méthode `edit` du contrôleur. Cette méthode répond à l'url (avec le verbe get) :

```
http://monsite.fr/user/n/edit
```

Où `n` est l'id de l'utilisateur qu'on veut modifier.

Elle contient juste deux lignes :

```
<?php
public function edit($id)
{
    $user = $this->userRepository->getById($id);

    return view('edit', compact('user'));
}
```

php

```
<?php
public function edit($id)
{
    $user = $this->userRepository->getById($id);

    return view('edit', compact('user'));
}
```

Le principe est exactement le même que pour la méthode `show` vue ci-dessus.

update

Cette méthode est destinée à mettre à jour l'enregistrement dans la table à partir des données transmises comme paramètres :

```
<?php
public function update($id, Array $inputs)
{
    $this->save($this->getById($id), $inputs);
}
```

On récupère l'enregistrement avec la méthode `getById` avec l'id transmis. Ensuite on met à jour dans la table avec la méthode privée `save` :

```
<?php
private function save(User $user, Array $inputs)
{
    $user->name = $inputs['name'];
    $user->email = $inputs['email'];
    $user->admin = isset($inputs['admin']);

    $user->save();
}
```

La méthode `update` du repository est appelée depuis la méthode `update` du contrôleur :

```
<?php
public function update(UserUpdateRequest $request, $id)
{
    $this->userRepository->update($id, $request->all());

    return redirect('user')->withOk("L'utilisateur " . $request->input('name') . " a été
modifié.");
}
```

Cette méthode répond à l'url (avec le verbe `put`) :

<http://monsite.fr/user/n>

Où n est l'id de l'utilisateur qu'on veut modifier.

store

Voici le code de cette méthode :

```
<?php
public function store(Array $inputs)
{
    $user = new $this->user;
    $user->password = bcrypt($inputs['password']);

    $this->save($user, $inputs);

    return $user;
}
```

Par sécurité le mot de passe entré est ici codé avec l'helper `bcrypt`. Ainsi il ne sera pas inscrit en clair dans la table mais sous forme codée.

On crée un nouvel objet User. On renseigne l'attribut `password` et on enregistre dans la table avec la méthode privée `save` que nous avons déjà vue ci-dessus :

```
<?php
private function save(User $user, Array $inputs)
{
```

```
$user->name = $inputs['name'];
$user->email = $inputs['email'];
$user->admin = isset($inputs['admin']);

$user->save();
}
```

La méthode du repository est appelée depuis la méthode `store` du contrôleur :

```
<?php
public function store(UserCreateRequest $request)
{
    $user = $this->userRepository->store($request->all());

    return redirect('user')->withOk("L'utilisateur " . $user->name . " a été créé.");
}
```

Cette méthode répond à l'url (avec le verbe `post`) :

```
http://monsite.fr/user
```

`destroy`

Elle contient juste une ligne :

```
<?php
public function destroy($id)
{
    $this->getRepository($id)->delete();
}
```

Elle est appelée depuis la méthode `destroy` du contrôleur :

```
<?php
public function destroy($id)
{
    $this->userRepository->destroy($id);

    return redirect()->back();
}
```

On supprime un enregistrement avec la méthode `delete` du modèle. Remarquez la redirection dans le contrôleur avec la méthode `back`. On renvoie ainsi la dernière requête.

Cette méthode répond à l'url (avec le verbe `delete`) :

```
http://monsite.fr/user/n
```

Où n est l'id de l'enregistrement qu'on veut supprimer.

`Les vues`

Voyons à présent les vues pour l'interaction avec le client. J'ai adopté pour les vues le même nom que les méthodes appelantes du contrôleur pour faciliter la compréhension.

`Le template`

Nous aurons le même template pour toutes les vues, c'est celui que nous avons déjà utilisé dans les précédents chapitres :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Mon joli site</title>
        {!!
Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css') !!}
{!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css') !!}
{!--[if lt IE 9]>
    {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.js') }}
{{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js') }}
}}
```

```

<![endif]-->
<style> textarea { resize: none; } </style>
</head>
<body>
    @yield('contenu')
</body>
</html>

```

Vue index

Cette vue est destinée à afficher la liste paginée des utilisateurs avec des boutons pour pouvoir accomplir toutes les actions. Voici le code de cette vue :

```

@extends('template')

@section('contenu')
    <br>
    <div class="col-sm-offset-4 col-sm-4">
        @if(session()->has('ok'))
            <div class="alert alert-success alert-dismissible">{!! session('ok') !!}</div>
        @endif
        <div class="panel panel-primary">
            <div class="panel-heading">
                <h3 class="panel-title">Liste des utilisateurs</h3>
            </div>
            <table class="table">
                <thead>
                    <tr>
                        <th>#</th>
                        <th>Nom</th>
                        <th></th>
                        <th></th>
                        <th></th>
                    </tr>
                </thead>
                <tbody>
                    @foreach ($users as $user)
                        <tr>
                            <td>{!! $user->id !!}</td>
                            <td class="text-primary"><strong>{!! $user->name !!}</strong></td>
                            <td>{!! link_to_route('user.show', 'Voir', [$user->id], ['class' => 'btn btn-success btn-block']) !!}</td>
                            <td>{!! link_to_route('user.edit', 'Modifier', [$user->id], ['class' => 'btn btn-warning btn-block']) !!}</td>
                            <td>
                                {!! Form::open(['method' => 'DELETE', 'route' => ['user.destroy', $user->id]]) !!}
                                    {!! Form::submit('Supprimer', ['class' => 'btn btn-danger btn-block', 'onclick' => 'return confirm(\"Vraiment supprimer cet utilisateur ?\")']) !!}
                                {!! Form::close() !!}
                            </td>
                        </tr>
                    @endforeach
                </tbody>
            </table>
        </div>
        {!! link_to_route('user.create', 'Ajouter un utilisateur', [], ['class' => 'btn btn-info pull-right']) !!}
        {!! $links !!}
    </div>
@endsection

```

Avec cet aspect :

Liste des utilisateurs

| # | Nom | Voir | Modifier | Supprimer |
|---|---------|------|----------|-----------|
| 1 | Dupont | Voir | Modifier | Supprimer |
| 2 | Durand | Voir | Modifier | Supprimer |
| 3 | Martin | Voir | Modifier | Supprimer |
| 5 | Zebulon | Voir | Modifier | Supprimer |

[Ajouter un utilisateur](#)

« 1 2 »

La vue index

Remarquez que pour générer la méthode `delete` on est obligé de créer un formulaire.

La vue teste aussi la présence d'une variable "ok" dans la session et affiche le message correspondant dans une barre si c'est le cas.

Vue show

La vue `show` sert à afficher la fiche d'un utilisateur avec son nom, son adresse email et son appartenance éventuelle au groupe des administrateurs :

```
@extends('template')

@section('contenu')
    <div class="col-sm-offset-4 col-sm-4">
        <br>
        <div class="panel panel-primary">
            <div class="panel-heading">Fiche d'utilisateur</div>
            <div class="panel-body">
                <p>Nom : {{ $user->name }}</p>
                <p>Email : {{ $user->email }}</p>
                @if($user->admin == 1)
                    Administrateur
                @endif
            </div>
        </div>
        <a href="javascript:history.back()" class="btn btn-primary">
            <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
        </a>
    </div>
@endsection
```

Avec cet aspect :

Fiche d'utilisateur

Nom : Durand

Email : durande@chezmoi.fr

Administrateur

[Retour](#)

La vue show

Vue edit

La vue `edit` sert à la modification d'un utilisateur, elle affiche un formulaire :

```
@extends('template')

@section('contenu')
    <div class="col-sm-offset-4 col-sm-4">
        <br>
        <div class="panel panel-primary">
            <div class="panel-heading">Modification d'un utilisateur</div>
            <div class="panel-body">
                <div class="col-sm-12">
                    {!! Form::model($user, ['route' => ['user.update', $user->id], 'method' => 'put', 'class' => 'form-horizontal panel']) !!}
                    <div class="form-group {!! $errors->has('name') ? 'has-error' : '' !!}">
                        {!! Form::text('name', null, ['class' => 'form-control', 'placeholder' => 'Nom']) !!}
                        {!! $errors->first('name', '<small class="help-block">:message</small>') !!}
                    </div>
                    <div class="form-group {!! $errors->has('email') ? 'has-error' : '' !!}">
                        {!! Form::email('email', null, ['class' => 'form-control', 'placeholder' => 'Email']) !!}
                        {!! $errors->first('email', '<small class="help-block">:message</small>') !!}
                    </div>
                    <div class="form-group">
                        <div class="checkbox">
                            <label>
                                {!! Form::checkbox('admin', 1, null)}
                            </label>
                        </div>
                    </div>
                    {!! Form::submit('Envoyer', ['class' => 'btn btn-primary pull-right']) !!}
                    {!! Form::close() !!}
                </div>
            </div>
            <a href="javascript:history.back()" class="btn btn-primary">
                <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
            </a>
        </div>
    @endsection
```

Avec cet aspect :

The screenshot shows a modal dialog titled "Modification d'un utilisateur". Inside, there are two input fields: one for the name containing "Durand" and another for the email containing "durand@chezmoi.fr". Below these fields is a checkbox labeled "Administrateur" which is checked. At the bottom right of the modal is a blue "Envoyer" button. At the bottom left of the page, outside the modal, is a blue "Retour" button with a circular arrow icon.

La vue edit



Remarquez l'utilisation de :

```
Form::model
```

html

qui permet de lier le formulaire au modèle et ainsi de renseigner automatiquement les contrôles qui possèdent le même nom qu'un champ de l'enregistrement.

Vue create

Cette vue sert à afficher le formulaire pour créer un utilisateur, c'est quasiment la même que pour la modification avec le mot de passe en plus :

```
@extends('template')

@section('contenu')
    <div class="col-sm-offset-4 col-sm-4">
        <br>
        <div class="panel panel-primary">
            <div class="panel-heading">Création d'un utilisateur</div>
            <div class="panel-body">
                <div class="col-sm-12">
                    {!! Form::open(['route' => 'user.store', 'class' => 'form-horizontal panel']) !!}
                    <div class="form-group {!! $errors->has('name') ? 'has-error' : '' !!}">
                        {!! Form::text('name', null, ['class' => 'form-control', 'placeholder' => 'Nom']) !!}
                        {!! $errors->first('name', '<small class="help-block">:message</small>') !!}
                    </div>
                    <div class="form-group {!! $errors->has('email') ? 'has-error' : '' !!}">
                        {!! Form::email('email', null, ['class' => 'form-control', 'placeholder' => 'Email']) !!}
                        {!! $errors->first('email', '<small class="help-block">:message</small>') !!}
                    </div>
                    <div class="form-group {!! $errors->has('password') ? 'has-error' : '' !!}">
                        {!! Form::password('password', ['class' => 'form-control', 'placeholder' => 'Mot de passe']) !!}
                        {!! $errors->first('password', '<small class="help-block">:message</small>') !!}
                    </div>
                    <div class="form-group">
                        {!! Form::password('password_confirmation', ['class' => 'form-control', 'placeholder' => 'Confirmation mot de passe']) !!}
                    </div>
                    <div class="form-group">
                        <div class="checkbox">
                            <label>
                                {!! Form::checkbox('admin', 1, null) !!}
                            </label>
                        </div>
                    </div>
                    {!! Form::submit('Envoyer', ['class' => 'btn btn-primary pull-right']) !!}
                    {!! Form::close() !!}
                </div>
            </div>
            <a href="javascript:history.back()" class="btn btn-primary">
                <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
            </a>
        </div>
    @endsection
```

Avec cet aspect :

Création d'un utilisateur

Nom

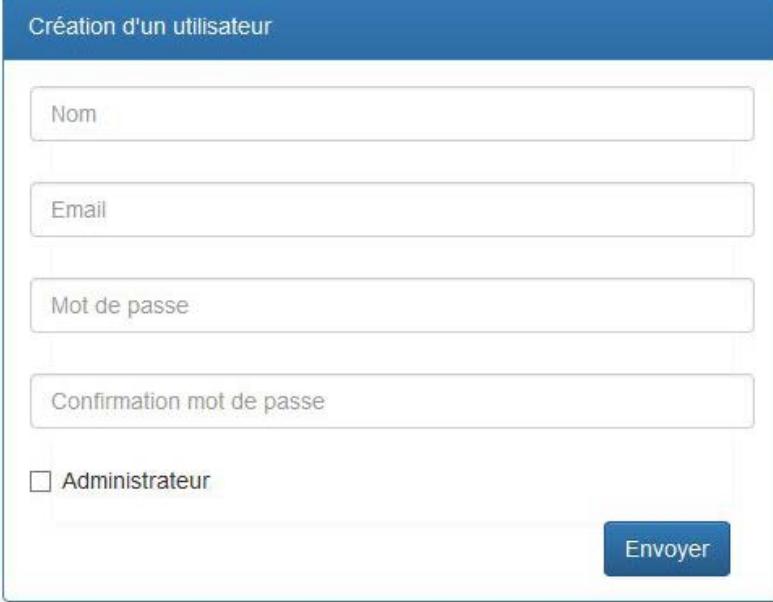
Email

Mot de passe

Confirmation mot de passe

Administrateur

Envoyer



 Retour

La vue create

Avec évidemment la validation active :

Création d'un utilisateur

Nom
Le champ Nom est obligatoire.

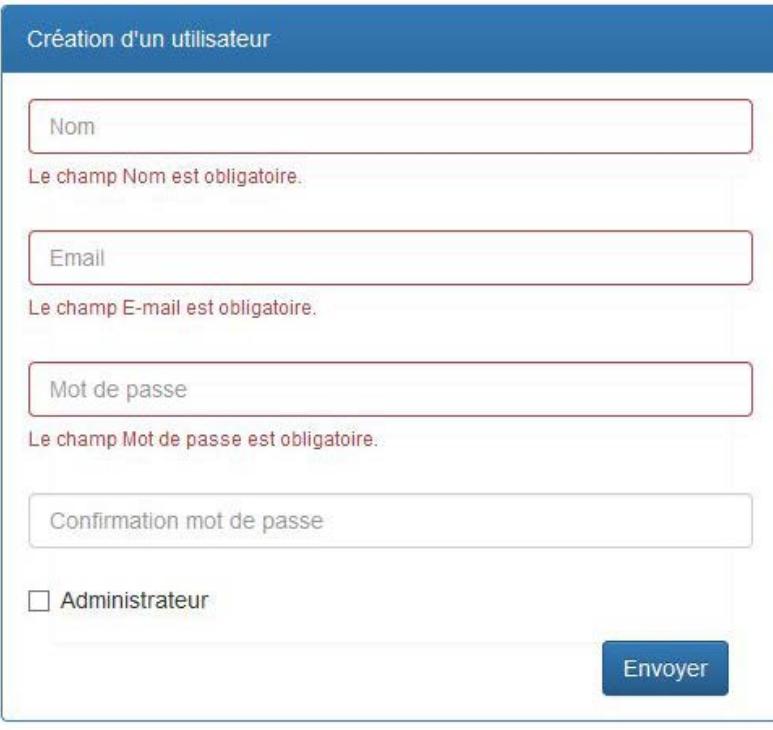
Email
Le champ E-mail est obligatoire.

Mot de passe
Le champ Mot de passe est obligatoire.

Confirmation mot de passe

Administrateur

Envoyer

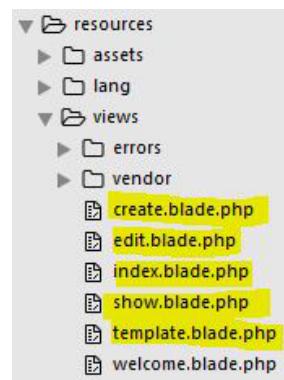


 Retour

La validation en action

Nous avons vu dans un chapitre précédent comment avoir ces messages en Français.

Vous devez donc avoir ces 5 vues :



Les cinq vues pour la ressource

Réflexion sur le code

Un repository de base

Notre code fonctionne correctement, mais est-il vraiment performant ? Lorsqu'on crée une classe, une bonne question est de se demander si le code est réutilisable. Si on observe le repository créé pour les utilisateurs on se rend compte qu'il est très ciblé sur le modèle concerné et si on a une autre ressource dans l'application il est fort probable qu'on va bidouiller avec du copier-coller, ce qui est toujours source de répétition de code et d'erreurs. Est-il possible de créer un repository de base pour les ressources ? Voici une solution :

```
<?php
namespace App\Repositories;

abstract class ResourceRepository
{
    protected $model;

    public function getPaginate($n)
    {
        return $this->model->paginate($n);
    }

    public function store(Array $inputs)
    {
        return $this->model->create($inputs);
    }

    public function getById($id)
    {
        return $this->model->findOrFail($id);
    }

    public function update($id, Array $inputs)
    {
        $this->getById($id)->update($inputs);
    }

    public function destroy($id)
    {
        $this->getById($id)->delete();
    }
}
```

Il est difficile de faire plus concis. Le seul élément qui va nous indiquer de quel modèle il s'agit est la propriété `$model`. Par contre le reste est tout à fait anonyme. Du coup le repository pour les utilisateurs va se trouver très simple à écrire :

```
<?php
namespace App\Repositories;

use App\User;

class UserRepository extends ResourceRepository
```

```
{
    public function __construct(User $user)
    {
        $this->model = $user;
    }
}
```

On a toutefois quelques petits soucis. Par exemple on a vu qu'il faut crypter le mot de passe et on réalisait cela dans le repository. On pourrait surcharger la méthode `store` pour le prévoir mais ça casserait la jolie harmonie du code.

Le modèle

Une autre solution plus élégante est de prévoir un "mutator" au niveau du modèle. Mais voyons déjà ce que nous avons dans ce modèle `User` :

```
php
<?php

namespace App;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * The attributes excluded from the model's JSON form.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];
}
```

Je reviendrai plus en détail sur les traits utilisés quand on verra l'authentification. Pour le moment on va s'intéresser à la propriété `$fillable`. Lorsqu'on crée un enregistrement avec la méthode `create` comme on l'a prévu ci-dessus il y a un risque au niveau de la sécurité. Dans le tableau transmis en paramètre on a normalement seulement les champs que l'on désire renseigner. Mais si un petit malin envoie d'autres informations elles risquent fort de se propager jusqu'à la table. Pour éviter cela on définit dans le modèle les champs qui peuvent être mis à jour avec cette méthode (on parle de mise à jour de masse) dans la propriété `$fillable`. Comme on a aussi le champ `admin` à renseigner il faut compléter le tableau :

```
php
<?php
protected $fillable = ['name', 'email', 'password', 'admin'];
```

On veut également crypter le mot de passe. Il suffit de mettre en place ce "mutator" :

```
php
<?php
public function setPasswordAttribute($password)
{
    $this->attributes['password'] = bcrypt($password);
}
```

Ainsi chaque fois qu'on va assigner l'attribut `password` il passera par cette méthode et on aura un cryptage de la valeur.

Le contrôleur

Il nous ne reste plus que la gestion de la case à cocher pour l'administration. On ne peut pas le résoudre comme on l'a fait pour le mot de passe puisque l'on n'a pas toujours l'information (la case à

cocher n'est transmise que si elle est cochée). On va donc prévoir ce traitement dans le contrôleur :

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Requests\UserCreateRequest;  
use App\Http\Requests\UserUpdateRequest;  
  
use App\Repositories\UserRepository;  
  
use Illuminate\Http\Request;  
  
class UserController extends Controller  
{  
  
    protected $userRepository;  
  
    protected $nbrPerPage = 4;  
  
    public function __construct(UserRepository $userRepository)  
    {  
        $this->userRepository = $userRepository;  
    }  
  
    public function index()  
    {  
        $users = $this->userRepository->getPaginate($this->nbrPerPage);  
        $links = $users->render();  
  
        return view('index', compact('users', 'links'));  
    }  
  
    public function create()  
    {  
        return view('create');  
    }  
  
    public function store(UserCreateRequest $request)  
    {  
        $this->setAdmin($request);  
  
        $user = $this->userRepository->store($request->all());  
  
        return redirect('user')->withOk("L'utilisateur " . $user->name . " a été créé.");  
    }  
  
    public function show($id)  
    {  
        $user = $this->userRepository->getById($id);  
  
        return view('show', compact('user'));  
    }  
  
    public function edit($id)  
    {  
        $user = $this->userRepository->getById($id);  
  
        return view('edit', compact('user'));  
    }  
  
    public function update(UserUpdateRequest $request, $id)  
    {  
        $this->setAdmin($request);  
  
        $this->userRepository->update($id, $request->all());  
  
        return redirect('user')->withOk("L'utilisateur " . $request->input('name') . " a été  
modifié.");  
    }  
  
    public function destroy($id)  
    {  
        $this->userRepository->destroy($id);  
  
        return redirect()->back();  
    }  
  
    private function setAdmin($request)  
    {
```

```

if(!$request->has('admin'))
{
    $request->merge(['admin' => 0]);
}
}

```

C'est la fonction privée `setAdmin` qui est chargée de gérer la case à cocher. On teste qu'on n'a pas la clé `admin` dans le tableau de données et, si c'est le cas, on l'ajoute en utilisant la méthode `merge`.

On en arrive à un code plus clair et plus facile à réutiliser et à maintenir. On pourrait pousser la réflexion au niveau du contrôleur et créer un contrôleur de base pour les ressources. Il suffirait de prévoir un préfixe pour les vues et de refléchir aux injections, et on pourrait obtenir quelque chose de très élégant, mais je ne vais pas poursuivre cette réflexion pour ne pas alourdir ce chapitre, l'important est de comprendre le principe. D'autre part on arrive rapidement à des impasses dans une application réelle qui oblige souvent à multiplier les codes spécifiques.

Les erreurs

La gestion des erreurs constitue une part importante dans le développement d'une application. On dit parfois que c'est dans ce domaine qu'on fait la différence entre le professionnel et l'amateur. Que nous propose Laravel dans ce domaine ?

Puisque nous sommes dans l'accès aux données il peut arriver un souci avec la connexion à la base. Voyons un peu ce que nous obtenons si MySQL ne répond plus... Avec notre application si j'arrête le service de MySQL puis que je lance l'url :

<http://monsite.fr/user>

J'obtiens :

Whoops, looks like something went wrong.

1/1 PDOException in Connector.php line 47:
SQLSTATE[HY000] [2002] Aucune connexion n'a pu être établie car l'ordinateur cible l'a expressément refusée.

Erreur suite à arrêt de MySQL

Je n'ai affiché ici que la partie supérieure. Ne soyez pas impressionné par la quantité de messages de la page d'erreurs, avec un peu d'habitude vous serez heureux de disposer de tous ces renseignements lorsqu'une erreur se produit dans votre application.

La première question qu'on peut se poser est : cet affichage des erreurs est parfait pour la phase de développement mais sur une application en ligne il vaut mieux cacher tout ça pour deux raisons évidentes : l'utilisateur n'en a rien à faire, par contre ça pourrait servir à quelqu'un de mal intentionné et lui fournir de précieux renseignements sur le fonctionnement de vos scripts.

Si vous regardez dans le fichier `.env` dont je vous ai déjà parlé vous trouvez cette ligne :

`APP_DEBUG=true`

Si on met `false` ici pour voir la différence on obtient plus que le laconique :

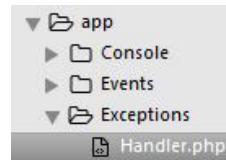
Whoops, looks like something went wrong.

Le message d'erreur simplifié de Laravel

Maintenant c'est plus sommaire ! Mais pour le coup ça devient trop laconique pour l'utilisateur et surtout ça ne lui donne aucun lien pour accéder à une autre page. Vous pouvez aussi considérer qu'un message en anglais n'est pas adapté pour votre site francophone. Malheureusement la page

correspondante se trouve dans le framework qu'on ne va évidemment pas aller bricoler !

Regardez ce fichier :



Le fichier des erreurs

Avec ce code :

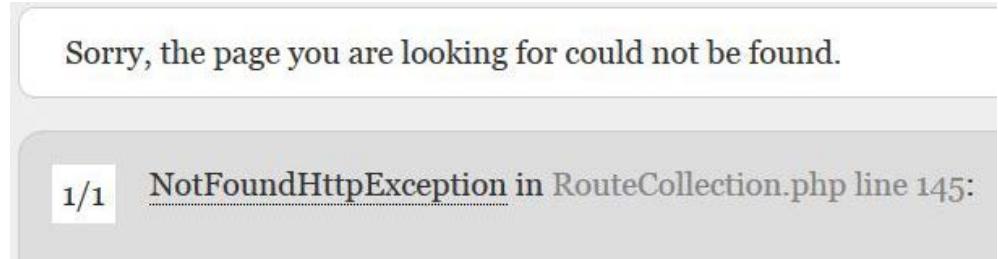
```
<?php  
  
namespace App\Exceptions;  
  
use Exception;  
use Illuminate\Auth\Access\AuthorizationException;  
use Illuminate\Database\Eloquent\ModelNotFoundException;  
use Symfony\Component\HttpKernel\Exception\HttpException;  
use Illuminate\Foundation\Validation\ValidationException;  
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;  
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;  
  
class Handler extends ExceptionHandler  
{  
    /**  
     * A list of the exception types that should not be reported.  
     *  
     * @var array  
     */  
    protected $dontReport = [  
        AuthorizationException::class,  
        HttpException::class,  
        ModelNotFoundException::class,  
        ValidationException::class,  
    ];  
  
    /**  
     * Report or log an exception.  
     *  
     * This is a great spot to send exceptions to Sentry, Bugsnag, etc.  
     *  
     * @param \Exception $e  
     * @return void  
     */  
    public function report(Exception $e)  
    {  
        return parent::report($e);  
    }  
  
    /**  
     * Render an exception into an HTTP response.  
     *  
     * @param \Illuminate\Http\Request $request  
     * @param \Exception $e  
     * @return \Illuminate\Http\Response  
     */  
    public function render($request, Exception $e)  
    {  
        return parent::render($request, $e);  
    }  
}
```

Toutes les erreurs d'exécution sont traitées ici. On découvre que tout est archivé avec la méthode `report`. Mais où se trouve cet archivage ? Regardez dans `storage/logs`, vous trouvez un fichier `laravel.log`. En l'ouvrant vous trouvez les erreurs archivées avec leur trace. Par exemple dans notre cas on a bien :

[2015-12-28 17:09:42] local.ERROR: exception 'PDOException' with message 'SQLSTATE[HY000]
[2002] Aucune connexion n'a pu être établie car l'ordinateur cible l'a expressément refusée..

C'est important de disposer de ce genre d'archivage des erreurs sur une application en production.

Un cas fréquent est celui de la page non trouvée (erreur 404) :



L'erreur 404

Si on veut changer le message de Laravel pour le rendre à notre goût et surtout adapté à notre langue il faut prévoir une vue. Par exemple (`resources/views/errors/404.blade.php`) :

```
@extends('template')

@section('contenu')
    <br>
    <div class="col-sm-offset-4 col-sm-4">
        <div class="panel panel-danger">
            <div class="panel-heading">
                <h3 class="panel-title">Il y a un problème !</h3>
            </div>
            <div class="panel-body">
                <p>Nous sommes désolés mais la page que vous désirez n'existe pas...
            </p>
        </div>
    </div>
@endsection
```

Maintenant pour une url qui n'aboutit pas on obtient :



Notre vue d'erreur 404 personnalisée

Avouez que c'est quand même mieux et Laravel est assez intelligent pour aller chercher cette vue automatiquement sans qu'on ait quelque chose de particulier à faire !

On a vu dans ce chapitre que lorsqu'on arrête le service MySQL on génère une erreur **PDOException**. Comment intercepter cette erreur ? Voilà une solution :

```
<?php

namespace App\Exceptions;

use Exception;
use Illuminate\Auth\Access\AuthorizationException;
use Illuminate\Database\Eloquent\ModelNotFoundException;
use Symfony\Component\HttpKernel\Exception\HttpException;
use Illuminate\Foundation\Validation\ValidationException;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use Illuminate\Foundation\Exceptions\Handler asExceptionHandler;

class Handler extends ExceptionHandler
{
    ...

    /**
     * Render an exception into an HTTP response.
     *
     * @param \Illuminate\Http\Request $request
     * @param Exception $exception
     */
    public function render($request, Exception $exception)
    {
        if ($exception instanceof ModelNotFoundException) {
            $model = $exception->getModel();

            return $this->errorPage($request, "The requested :model was not found.", [
                'model' => $model
            ]);
        }

        if ($exception instanceof ValidationException) {
            return $this->errorPage($request, "The validation failed.", [
                'errors' => $exception->getErrors()
            ]);
        }

        if ($exception instanceof AuthorizationException) {
            return $this->errorPage($request, "You do not have permission to access this resource.");
        }

        if ($exception instanceof NotFoundHttpException) {
            return $this->errorPage($request, "The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.");
        }

        if ($exception instanceof PDOException) {
            return $this->errorPage($request, "There was a problem connecting to the database. Please try again later.");
        }

        if ($exception instanceof HttpException) {
            return $this->errorPage($request, "The requested page does not exist.", [
                'code' => $exception->getStatusCode()
            ]);
        }

        if ($exception instanceof \Exception) {
            return $this->errorPage($request, "An unexpected error occurred while processing your request. Please try again later.");
        }
    }
}
```

```
* @param \Exception $e
* @return \Illuminate\Http\Response
*/
public function render($request, Exception $e)
{
    if($e instanceof \PDOException)
    {
        return response()->view('errors.pdo', [], 500);
    }
    return parent::render($request, $e);
}
```

Ainsi on affichera une page spécifique pour cette erreur.

En résumé

- Créer un gestionnaire (repository) indépendant du contrôleur pour les accès aux données permet de disposer d'un code clair et facile à maintenir et tester.
- Il est important de penser à la réutilisation du code que l'on crée.
- Les vues doivent utiliser au maximum les possibilités de Blade, des helpers et de la classe Form pour être concises et lisibles.
- La gestion des erreurs ne doit pas être négligée, il faut enlever le mode de débogage sur un site en production et prévoir des messages explicites pour les utilisateurs en fonction de l'erreur rencontrée.



Les ressources (1/2)

L'authentification



L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

OpenClassrooms

Qui sommes-nous ?

Fonctionnement de nos cours

Recrutement

Nous contacter

En plus

Créer un cours

CourseLab

Conditions Générales d'Utilisation

□ Professionals

Affiliation

Entreprises

Universités et écoles

□ Suivez-nous

Le blog OpenClassrooms



English

Español

Accueil ▶ Cours ▶ Découvrez le framework PHP Laravel ▶ L'authentification

Découvrez le framework PHP Laravel



□ 15 heures

□ Moyenne

Licence □ □ □ □



L'authentification

□ Connectez-vous ou inscrivez-vous pour bénéficier de toutes les fonctionnalités de ce cours !

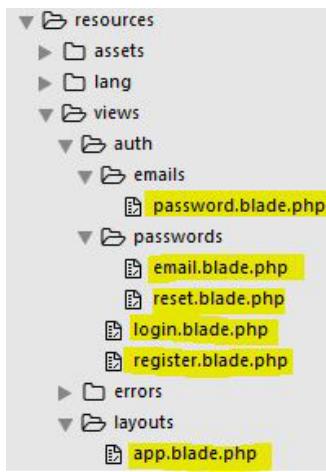
L'authentification constitue une tâche fréquente. En effet il y a souvent des parties d'un site qui ne doivent être accessibles qu'à certains utilisateurs, ne serait-ce que l'administration. La solution proposée par Laravel est d'une grande simplicité parce que tout est déjà préparé comme nous allons le voir dans ce chapitre.

Artisan

La version 5.0 de Laravel était équipée des vues, assets et routes pour l'authentification. L'arrivée de la version 5.1 a vu la disparition de ces éléments suite à de nombreuses et animées discussions. Finalement avec la version 5.2 on n'a toujours pas tout ça de base mais il y a une commande Artisan pour le générer. Vous allez donc utiliser cette commande :

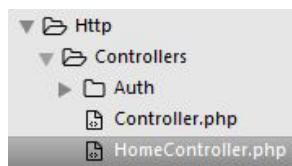
```
php artisan make:auth
```

Si tout se passe bien vous devriez avoir ces vues :



Les vues de l'authentification

Vous devez aussi trouver un nouveau contrôleur :



Les bases de données

1. Migrations et modèles
 2. Les ressources (1/2)
 3. Les ressources (2/2) et les erreurs
- 4. L'authentification
5. La relation 1:n
 6. La relation n:n
 7. Les commandes et les assistants
 8. Query Builder
- Quiz : Quiz 2
- Activité : Construisez un site de sondages avec une base de données

[Accéder au forum](#)



Un nouveau contrôleur

Et dans le fichier des routes ce nouveau code :

```
<?php  
Route::auth();  
Route::get('/home', 'HomeController@index');
```

php

Vous êtes maintenant prêt pour ce chapitre !



`Route::auth()` génère automatiquement toutes les routes de l'authentification

Les tables

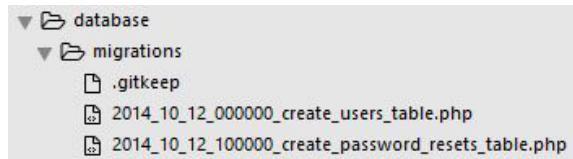
La table users

Nous allons utiliser la table `users` que nous avons créée au chapitre précédent. Par défaut Laravel considère que cette table existe et il s'en sert comme référence pour l'authentification.

Par défaut également c'est Eloquent qui est utilisé comme driver, il est aussi possible de changer ce fonctionnement.

La table password_reset

Lors de l'installation il existe deux migrations présentes :



Les 2 migrations présentes à l'installation

Lors des précédents chapitres je vous ai fait supprimer la seconde parce qu'on avait juste besoin de la table des utilisateurs. Pour ce chapitre on va avoir besoin de la seconde table qui va nous servir pour la réinitialisation des mots de passe.

On va faire un peu le ménage en lançant une commande d'Artisan pour supprimer toutes les tables que vous avez créées :

```
php artisan migrate:reset
```

php

Il ne devrait alors plus vous rester que la table `migrations` vide. Si ce n'est pas le cas faites le nécessaire dans votre base.

Avec les deux migrations présentes lancez alors la commande d'Artisan pour créer les tables :

```
php artisan migrate
```

php

Vous devriez normalement obtenir ceci :



Les 3 tables

Maintenant que les tables sont prêtes, passons à la suite.



Conservez la migration que nous avons faite lors des précédents chapitres pour les utilisateurs avec la colonne `admin`. N'utilisez pas celle fournie de base avec Laravel qui ne comporte pas

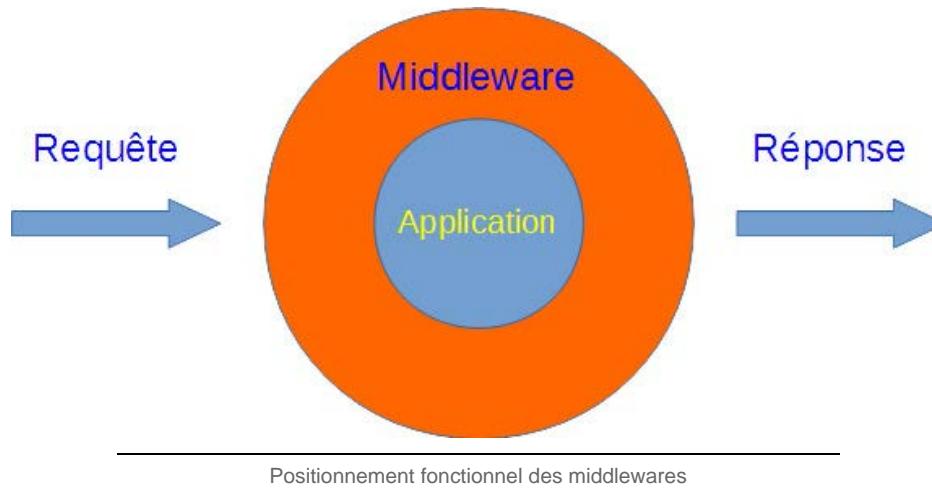
cette colonne.

Les middlewares



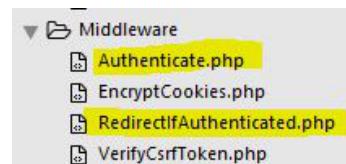
C'est quoi un middleware ?

Voici un petit schéma pour illustrer cela :



Un middleware effectue un traitement à l'arrivée de la requête ou à son départ. Par exemple la gestion des sessions ou des cookies dans Laravel se fait dans un middleware, ainsi que l'authentification. On a en fait plusieurs middlewares en pelures d'oignon, chacun effectue son traitement et transmet la requête ou la réponse au suivant.

Laravel s'installe avec deux fichiers middlewares qui concernent l'authentification :



Les middlewares pour l'authentification

Authenticate

Ce middleware permet de savoir si un utilisateur n'est pas authentifié. Voici son code :

```
<?php  
namespace App\Http\Middleware;  
  
use Closure;  
use Illuminate\Support\Facades\Auth;  
  
class Authenticate  
{  
    /**  
     * Handle an incoming request.  
     *  
     * @param \Illuminate\Http\Request $request  
     * @param \Closure $next  
     * @param string|null $guard  
     * @return mixed  
     */  
    public function handle($request, Closure $next, $guard = null)  
    {  
        if (Auth::guard($guard)->guest()) {  
            if ($request->ajax() || $request->wantsJson()) {  
                return response('Unauthorized.', 401);  
            } else {  
                return redirect()->guest('login');  
            }  
        }  
    }  
}
```

php

```

        }
    }

    return $next($request);
}
}

```

On regarde si l'utilisateur est juste un invité (`guest`). Si c'est le cas on teste si la requête est en Ajax, auquel cas on renvoie `Unauthorized`. Si elle n'est pas en Ajax on fait une redirection vers la route `login` pour que l'invité puisse s'authentifier. Si ce n'est pas un invité on laisse la requête suivre normalement son cours. Tout cela n'est évidemment pas figé dans le marbre et on peut changer tout ce qu'on veut. Pour ce cours je vais me contenter de prendre le code par défaut.

 La méthode `redirect()->guest` équivaut à la méthode `redirect()` avec la différence que l'URL est mémorisée en session pour pouvoir rediriger l'utilisateur à l'issue de son authentification vers l'URL qu'il voulait initialement atteindre.

RedirectIfAuthenticated

Ce middleware permet de savoir si l'utilisateur est authentifié. C'est donc l'exact inverse du précédent. Voici son code :

```

<?php
namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\Auth;

class RedirectIfAuthenticated
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @param string|null $guard
     * @return mixed
     */
    public function handle($request, Closure $next, $guard = null)
    {
        if (Auth::guard($guard)->check()) {
            return redirect('/');
        }

        return $next($request);
    }
}

```

Si l'utilisateur est authentifié, ce qui nous est indiqué par la méthode `check`, alors on renvoie à la page de base du site (`/`). Sinon on laisse la requête suivre normalement son cours.

Routes et contrôleur

Les routes

On a vu qu'on a créé des routes avec Artisan :

```

<?php
Route::auth();
Route::get('/home', 'HomeController@index');

```

Comme ce n'est pas très explicite voyons un peu les routes ainsi créées :

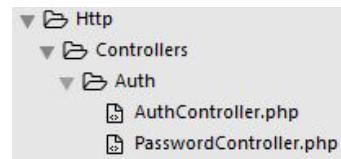
| | |
|---|-----------|
| GET HEAD home | |
| GET HEAD login | |
| POST login | |
| GET HEAD logout | |
| POST password/email | |
| POST password/reset | |
| GET HEAD password/reset/{token?} | |
| GET HEAD register | |
| POST register | |
|
 | |
| App\Http\Controllers\HomeController@index | web,auth |
| App\Http\Controllers\Auth\AuthController@showLoginForm | web,guest |
| App\Http\Controllers\Auth\AuthController@login | web,guest |
| App\Http\Controllers\Auth\AuthController@logout | web |
| App\Http\Controllers\Auth\PasswordController@sendResetLinkEmail | web,guest |
| App\Http\Controllers\Auth\PasswordController@reset | web,guest |
| App\Http\Controllers\Auth\PasswordController@showResetForm | web,guest |
| App\Http\Controllers\Auth\AuthController@showRegistrationForm | web,guest |
| App\Http\Controllers\Auth\AuthController@register | web,guest |

Les routes créées

Nous allons analyser tout ça dans ce chapitre.

Les contrôleurs

Il est fait référence à deux contrôleurs que l'on trouve ici :



Les deux contrôleurs de l'authentification

AuthController

Ce contrôleur est destiné à gérer :

1. l'enregistrement des utilisateurs
2. la connexion
3. la déconnexion

Si on regarde son code :

```

<?php
namespace App\Http\Controllers\Auth;

use App\User;
use Validator;
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\ThrottlesLogins;
use Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers;

class AuthController extends Controller
{
    /*
    |--------------------------------------------------------------------------
    | Registration & Login Controller
    |--------------------------------------------------------------------------
    |
    | This controller handles the registration of new users, as well as the
    | authentication of existing users. By default, this controller uses
    | a simple trait to add these behaviors. Why don't you explore it?
    |
    */

    use AuthenticatesAndRegistersUsers, ThrottlesLogins;

    /**
     * Where to redirect users after login / registration.
     *
  
```

```

 * @var string
 */
protected $redirectTo = '/';

/**
 * Create a new authentication controller instance.
 *
 * @return void
 */
public function __construct()
{
    $this->middleware($this->guestMiddleware(), ['except' => 'logout']);
}

/**
 * Get a validator for an incoming registration request.
 *
 * @param array $data
 * @return \Illuminate\Contracts\Validation\Validator
 */
protected function validator(array $data)
{
    return Validator::make($data, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255|unique:users',
        'password' => 'required|min:6|confirmed',
    ]);
}

/**
 * Create a new user instance after a valid registration.
 *
 * @param array $data
 * @return User
 */
protected function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => bcrypt($data['password']),
    ]);
}
}

```

On ne retrouve aucune des méthodes auxquelles il est fait référence dans les routes. Vous pouvez remarquer l'utilisation du trait `\Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers`. Donc toutes les méthodes se trouvent dans le framework lui-même. Si nous avons besoin de les personnaliser la seule solution est donc de les surcharger.



Pour mémoire un trait est destiné à ajouter des fonctionnalités à une classe sans passer par l'héritage.



Le trait `\ThrottlesLogins` est destiné à mettre en place une protection contre les attaques "brute force".

PasswordController

Ce contrôleur est destiné uniquement à permettre la réinitialisation du mot de passe en cas d'oubli par l'utilisateur. Si on regarde aussi son code :

```

<?php
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\ResetsPasswords;

class PasswordController extends Controller
{
    /*
    |--------------------------------------------------------------------------
    | Password Reset Controller
    |--------------------------------------------------------------------------
    |
    | This controller is responsible for handling password reset requests
    | and uses a simple trait to temporarily disable the CSRF middleware.
    |
    | More information on this trait can be found on the package's README.
    |
    */
}
```

```

|-----|
|
| This controller is responsible for handling password reset requests
| and uses a simple trait to include this behavior. You're free to
| explore this trait and override any methods you wish to tweak.
|
|*/
|
use ResetsPasswords;

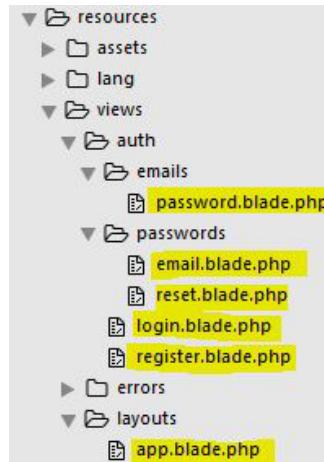
/**
 * Create a new password controller instance.
 *
 * @return void
 */
public function __construct()
{
    $this->middleware('guest');
}
}

```

On se rend compte que lui aussi utilise un trait (`Illuminate\Foundation\Auth\ResetsPasswords`).

Les vues

Vous allez trouver les vues ici :



Les vues de l'authentification

Vous n'êtes évidemment pas obligé d'utiliser ces vues si vous voulez les intégrer visuellement dans un site (c'est d'ailleurs ce qui sera fait au cours de l'activité de ce chapitre) mais pour ce chapitre on va les utiliser directement. Elles sont toutes conçues de la même manière. Voici par exemple la vue pour la connexion (`resources/views/auth/login.blade.php`) :

```

@extends('layouts.app')                                     html

@section('content')


Login



<form class="form-horizontal" role="form" method="POST" action="{{ url('/login') }}>
    {!! csrf_field() !!}
    <div class="form-group{{ $errors->has('email') ? ' has-error' : '' }}>
        <label class="col-md-4 control-label">E-Mail Address</label>
        <div class="col-md-6">
            <input type="email" class="form-control" name="email" value="{{ old('email') }}>
            @if ($errors->has('email'))
                <span class="help-block">
                    <strong>{{ $errors->first('email') }}</strong>
                </span>
            @endif
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-6 col-md-offset-4">
            <button type="submit" class="btn btn-primary">Connexion</button>
        </div>
    </div>
</form>


```

```

        @endif
    </div>
</div>

<div class="form-group{{ $errors->has('password') ? ' has-error' : '' }}>
    <label class="col-md-4 control-label">Password</label>

    <div class="col-md-6">
        <input type="password" class="form-control" name="password">

        @if ($errors->has('password'))
            <span class="help-block">
                <strong>{{ $errors->first('password') }}</strong>
            </span>
        @endif
    </div>
</div>

<div class="form-group">
    <div class="col-md-6 col-md-offset-4">
        <div class="checkbox">
            <label>
                <input type="checkbox" name="remember"> Remember Me
            </label>
        </div>
    </div>
</div>

<div class="form-group">
    <div class="col-md-6 col-md-offset-4">
        <button type="submit" class="btn btn-primary">
            <i class="fa fa-btn fa-sign-in"></i>Login
        </button>
    </div>
</div>
<a class="btn btn-link" href="{{ url('/password/reset') }}>Forgot Your
Password?</a>
</div>
</div>
</form>
</div>
</div>
</div>
</div>
</div>
@endsection

```

Évidemment tout est en anglais ! D'autre part on voit qu'on utilise un template (`app`).

L'enregistrement d'un utilisateur

La validation

Dans le contrôleur `AuthController` vous trouvez ce code :

```

<?php
protected function validator(array $data)
{
    return Validator::make($data, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255|unique:users',
        'password' => 'required|min:6|confirmed',
    ]);
}

```

On a vu jusqu'à présent la validation se faire à partir d'une requête de formulaire et là c'est réalisé différemment. Sans doute parce qu'il était difficile pour le framework de gérer correctement les espaces de noms dans ce cas. On trouve aussi dans cette classe une méthode pour créer l'utilisateur :

```

<?php
protected function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => bcrypt($data['password']),
    ]);
}

```

}

Comme on a ajouté une colonne `admin` il faut un peu modifier le code pour aussi l'enregistrer :

```
php
<?php
public function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => bcrypt($data['password']),
        'admin' => isset($data['admin'])
    ]);
}
```

Attention, dans le chapitre précédent on a mis en place dans le modèle **User** un mutateur pour crypter automatiquement le mot de passe, si vous le cryptez deux fois vous allez avoir des soucis ! Il faut donc choisir la méthode que vous voulez utiliser.

Le contrôleur

Avant d'envisager une authentification un visiteur doit pouvoir s'enregistrer.

Si on regarde le contenu du trait `AuthenticatesAndRegistersUsers` on se rend compte qu'il fait appel à deux autres traits :

```
php
<?php

namespace Illuminate\Foundation\Auth;

trait AuthenticatesAndRegistersUsers
{
    use AuthenticatesUsers, RegistersUsers {
        AuthenticatesUsers::redirectTo insteadof RegistersUsers;
        AuthenticatesUsers::getGuard insteadof RegistersUsers;
    }
}
```

Il y a deux méthodes dans le trait `RegistersUsers` pour l'enregistrement des utilisateurs :

```
php
<?php

namespace Illuminate\Foundation\Auth;

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

trait RegistersUsers
{
    use RedirectsUsers;

    /**
     * Show the application registration form.
     *
     * @return \Illuminate\Http\Response
     */
    public function getRegister()
    {
        return $this->showRegistrationForm();
    }

    /**
     * Show the application registration form.
     *
     * @return \Illuminate\Http\Response
     */
    public function showRegistrationForm()
    {
        if (property_exists($this, 'registerView')) {
            return view($this->registerView);
        }

        return view('auth.register');
    }
}
```

```

/**
 * Handle a registration request for the application.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function postRegister(Request $request)
{
    return $this->register($request);
}

/**
 * Handle a registration request for the application.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function register(Request $request)
{
    $validator = $this->validator($request->all());

    if ($validator->fails()) {
        $this->throwValidationException(
            $request, $validator
        );
    }

    Auth::guard($this->getGuard())->login($this->create($request->all()));

    return redirect($this->redirectToPath());
}

/**
 * Get the guard to be used during registration.
 *
 * @return string|null
 */
protected function getGuard()
{
    return property_exists($this, 'guard') ? $this->guard : null;
}

```

Voyons de plus près ces deux méthodes :

- **getRegister** : ici on renvoie la vue `auth.register` qui doit contenir le formulaire pour l'enregistrement (notez qu'il est testé la présence éventuelle d'une propriété `registerView` qui permet de changer facilement la localisation et le nom de la vue en créant cette propriété).
- **postRegister** : ici on traite la soumission du formulaire, la validation est assurée par la méthode `qu'on a vue` ci-dessus, et si la validation est correcte on crée dans la base cet utilisateur avec la méthode `create`, on connecte le nouvel utilisateur avec la méthode `login`, enfin on renvoie à l'url définie dans la méthode `redirectToPath`

Comme tout ce code se situe dans le framework nous ne devons pas directement le modifier. On peut toutefois modifier l'url de redirection. Regardez la méthode `redirectToPath` placée dans le trait

`RedirectUsers` :

```

<?php
public function redirectToPath()
{
    if (property_exists($this, 'redirectToPath'))
    {
        return $this->redirectToPath();
    }

    return property_exists($this, 'redirectTo') ? $this->redirectTo : '/home';
}

```

On teste la présence éventuelle d'une propriété `redirectToPath`. Si elle est présente on l'utilise pour la redirection. On teste aussi la présence d'une propriété `redirectTo`, sinon on redirige vers `home`. Donc si on veut une redirection spécifique il suffit de créer une propriété `redirectToPath` ou `redirectTo` dans le contrôleur. Pour toute autre modification on devra surcharger les méthodes.

L'url est `..../register`.

La vue auth.register

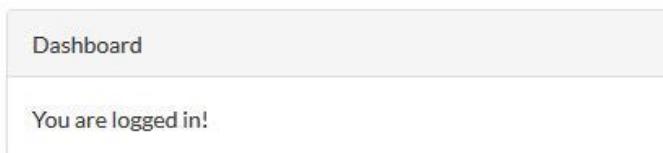
Il nous faut une vue pour l'enregistrement, on a vu que Laravel nous en propose une. Voici l'aspect normalement obtenu :

The screenshot shows a registration form titled "Register". It contains four input fields: "Name", "E-Mail Address", "Password", and "Confirm Password". Below the fields is a blue "Register" button.

Le formulaire pour l'enregistrement

Lorsqu'un utilisateur est créé et connecté il est renvoyé sur la route définie par la fonction

`redirectToPath` qu'on a vue ci-dessus. Comme la route, le contrôleur et la vue sont prévues, on obtient :



Utilisateur enregistré et connecté

La connexion et la déconnexion

Maintenant que les utilisateurs peuvent s'enregistrer passons à la connexion. Par défaut Laravel prévoit de le faire à partir de l'adresse email. On va conserver ce comportement.

La connexion

On a deux méthodes concernées dans le trait `AuthenticatesUsers` :

```
<?php  
  
namespace Illuminate\Foundation\Auth;  
  
use Illuminate\Http\Request;  
use Illuminate\Support\Facades\Auth;  
use Illuminate\Support\Facades\Lang;  
  
trait AuthenticatesUsers  
{  
    use RedirectsUsers;  
  
    /**  
     * Show the application login form.  
     *  
     * @return \Illuminate\Http\Response  
     */  
    public function getLogin()  
    {  
        return $this->showLoginForm();  
    }  
  
    /**  
     * Show the application login form.  
     *
```

php

```

/*
 * @return \Illuminate\Http\Response
 */
public function showLoginForm()
{
    $view = property_exists($this, 'loginView')
        ? $this->loginView : 'auth.authenticate';

    if ($view->exists($view)) {
        return view($view);
    }

    return view('auth.login');
}

/**
 * Handle a login request to the application.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function postLogin(Request $request)
{
    return $this->login($request);
}

/**
 * Handle a login request to the application.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function login(Request $request)
{
    $this->validateLogin($request);

    // If the class is using the ThrottlesLogins trait, we can automatically throttle
    // the login attempts for this application. We'll key this by the username and
    // the IP address of the client making these requests into this application.
    $throttles = $this->isUsingThrottlesLoginsTrait();

    if ($throttles && $lockedOut = $this->hasTooManyLoginAttempts($request)) {
        $this->fireLockoutEvent($request);

        return $this->sendLockoutResponse($request);
    }

    $credentials = $this->getCredentials($request);

    if (Auth::guard($this->getGuard())->attempt($credentials, $request->has('remember'))) {
        return $this->handleUserWasAuthenticated($request, $throttles);
    }

    // If the login attempt was unsuccessful we will increment the number of attempts
    // to login and redirect the user back to the login form. Of course, when this
    // user surpasses their maximum number of attempts they will get locked out.
    if ($throttles && ! $lockedOut) {
        $this->incrementLoginAttempts($request);
    }

    return $this->sendFailedLoginResponse($request);
}

/**
 * Validate the user login request.
 *
 * @param \Illuminate\Http\Request $request
 * @return void
 */
protected function validateLogin(Request $request)
{
    $this->validate($request, [
        $this->loginUsername() => 'required', 'password' => 'required',
    ]);
}

/**
 * Send the response after the user was authenticated.
 *
 * @param \Illuminate\Http\Request $request
 */

```

```
* @param  bool  $throttles
* @return \Illuminate\Http\Response
*/
protected function handleUserWasAuthenticated(Request $request, $throttles)
{
    if ($throttles) {
        $this->clearLoginAttempts($request);
    }

    if (method_exists($this, 'authenticated')) {
        return $this->authenticated($request, Auth::guard($this->getGuard())->user());
    }

    return redirect()->intended($this->redirectPath());
}

/**
 * Get the failed login response instance.
 *
 * @param \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
protected function sendFailedLoginResponse(Request $request)
{
    return redirect()->back()
        ->withInput($request->only($this->loginUsername(), 'remember'))
        ->withErrors([
            $this->loginUsername() => $this->getFailedLoginMessage(),
        ]);
}

/**
 * Get the failed login message.
 *
 * @return string
 */
protected function getFailedLoginMessage()
{
    return Lang::has('auth.failed')
        ? Lang::get('auth.failed')
        : 'These credentials do not match our records.';
}

/**
 * Get the needed authorization credentials from the request.
 *
 * @param \Illuminate\Http\Request  $request
 * @return array
 */
protected function getCredentials(Request $request)
{
    return $request->only($this->loginUsername(), 'password');
}

/**
 * Log the user out of the application.
 *
 * @return \Illuminate\Http\Response
 */
public function getLogout()
{
    return $this->logout();
}

/**
 * Log the user out of the application.
 *
 * @return \Illuminate\Http\Response
 */
public function logout()
{
    Auth::guard($this->getGuard())->logout();

    return redirect(property_exists($this, 'redirectTo') ? $this->redirectTo : '/');
}

/**
 * Get the guest middleware for the application.
 */

```

```

public function guestMiddleware()
{
    $guard = $this->getGuard();

    return $guard ? 'guest:'.$guard : 'guest';
}

/**
 * Get the login username to be used by the controller.
 *
 * @return string
 */
public function loginUsername()
{
    return property_exists($this, 'username') ? $this->username : 'email';
}

/**
 * Determine if the class is using the ThrottlesLogins trait.
 *
 * @return bool
 */
protected function isUsingThrottlesLoginsTrait()
{
    return in_array(
        ThrottlesLogins::class, class_uses_recursive(static::class)
    );
}

/**
 * Get the guard to be used during authentication.
 *
 * @return string|null
 */
protected function getGuard()
{
    return property_exists($this, 'guard') ? $this->guard : null;
}
}

```

Voyons de plus près ces deux méthodes :

- **getLogin** : ici on renvoie la vue `auth.login` (sauf s'il existe éventuellement une vue `auth.authenticate` ou si on a créé une propriété `loginView`) qui doit contenir le formulaire pour la connexion.
- **postLogin** : ici on traite la soumission du formulaire, la validation est assurée directement dans la méthode avec la puissante méthode `validate` qui est une alternative intéressante aux requêtes de formulaires. Si la validation est correcte on vérifie qu'on a pas d'attaque (throttle) puis les données sont vérifiées dans la table avec la méthode `attempt`. Si tout va bien on renvoie comme défini par la méthode `handleUserWasAuthenticated`, sinon on redirige sur le formulaire avec un message d'erreur défini par la fonction `getFailedLoginMessage`.

Au fil des évolutions de nombreuses fonctions ont été mises en place pour éviter de surcharger dans le contrôleur, pour chaque cas il convient de bien regarder le code pour déterminer la meilleure stratégie à adopter.

L'url est `..../login`.

La vue auth.login

Cette vue est aussi prévue. Voici l'aspect du formulaire de connexion :

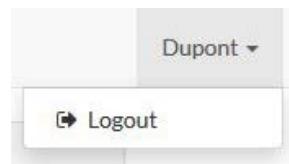
The screenshot shows a clean, modern login interface. At the top left is the word "Login". Below it are two input fields: one for "E-Mail Address" and another for "Password". To the right of the password field is a "Remember Me" checkbox. At the bottom are two buttons: a blue "Login" button with a key icon and a link "Forgot Your Password?".

Le formulaire de connexion

S'il y a déjà un utilisateur connecté vous allez être piégé par le middleware pour accéder à ce formulaire. Déconnectez l'utilisateur par le menu ou supprimez les cookies mémorisés par le navigateur.

La déconnexion

Dans la barre de menu est prévue la déconnexion :



La commande de déconnexion

Voici la méthode concernée dans le trait `AuthenticatesUsers` :

```
<?php
/**
 * Log the user out of the application.
 *
 * @return \Illuminate\Http\Response
 */
public function getLogout()
{
    return $this->logout();
}

/**
 * Log the user out of the application.
 *
 * @return \Illuminate\Http\Response
 */
public function logout()
{
    Auth::guard($this->getGuard())->logout();

    return redirect(property_exists($this, 'redirectAfterLogout') ? $this->redirectAfterLogout : '/');
}
```

php

L'utilisateur est déconnecté avec la méthode `logout` et il est ensuite redirigé à la racine du site ou à la route définie par la propriété `redirectAfterLogout` si elle existe, sinon vers la racine "/". L'oubli du mot de passe

Il y a la table `password_resets` dans notre base :

| # | Nom | Type |
|---|------------|--------------|
| 1 | email | varchar(255) |
| 2 | token | varchar(255) |
| 3 | created_at | timestamp |

La table password_resets

On voit qu'on va mémoriser ici l'adresse email, le jeton (token) et le timestamp (par défaut les jetons sont valables pendant une heure).

On a aussi un contrôleur :



Le contrôleur PasswordController

Les URL auront la forme `....password/...`

Dans le formulaire de connexion il est prévu un lien en cas d'oubli du mot de passe :

Remember Me
Login [Forgot Your Password?](#)

Le lien pour l'oubli du mot de passe

L'url correspondant est `....password/email`

Voici l'aspect du formulaire :

Reset Password

E-Mail Address

Send Password Reset Link

Le formulaire pour l'oubli du mot de passe

On demande à l'utilisateur de saisir son adresse email pour pouvoir le retrouver dans la table des utilisateurs.

Voici le code dans le trait `ResetsPasswords` :

```

<?php
/**
 * Display the form to request a password reset link.
 *
 * @return \Illuminate\Http\Response
 */
public function getEmail()
{
    return $this->showLinkRequestForm();
}

/**
 * Display the form to request a password reset link.
 */
  
```

php

```

/*
 * @return \Illuminate\Http\Response
 */
public function showLinkRequestForm()
{
    if (view()->exists('auth.passwords.email')) {
        return view('auth.passwords.email');
    }

    return view('auth.password');
}

/**
 * Send a reset link to the given user.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function postEmail(Request $request)
{
    return $this->sendResetLinkEmail($request);
}

/**
 * Send a reset link to the given user.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function sendResetLinkEmail(Request $request)
{
    $this->validate($request, ['email' => 'required|email']);

    $broker = $this->getBroker();

    $response = Password::broker($broker)->sendResetLink($request->only('email'), function (Message
$message) {
        $message->subject($this->getEmailSubject());
    });

    switch ($response) {
        case Password::RESET_LINK_SENT:
            return $this->getSendResetLinkEmailSuccessResponse($response);

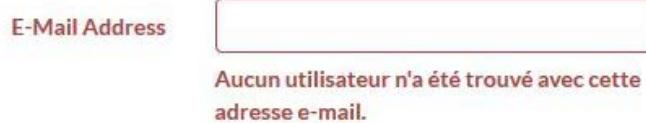
        case Password::INVALID_USER:
        default:
            return $this->getSendResetLinkEmailFailureResponse($response);
    }
}

```

La soumission du formulaire est prévue dans la méthode

`postEmail`. On a deux cas :

- L'utilisateur n'est pas valide (l'adresse email n'existe pas) : on redirige sur le formulaire avec le message d'erreur dans la variable `error`, avec cet aspect :



Le message d'erreur pour une adresse inconnue

- l'utilisateur est valide (on lui a envoyé un email) : on redirige sur le formulaire avec le message dans la variable `status`.

Nous vous avons envoyé par courriel le lien de réinitialisation du mot de passe !

L'email a bien été envoyé.



C'est quoi la méthode `trans` ?

C'est un helper qui permet d'adapter le texte linguistiquement à partir de la valeur de la clé `locale` dans `config/app.php`. Nous verrons cela en détail dans le chapitre sur la localisation.

Pour que l'email soit effectivement envoyé il faut que tout soit bien configuré dans le fichier `.env` :

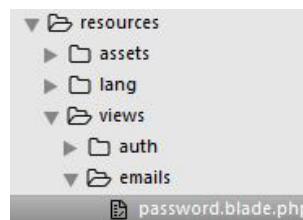
```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.free.fr
MAIL_PORT=25
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=""
```

Et dans `config/mail.php` :

```
'from' => array('address' => 'moi@free.fr', 'name' => 'Administrateur'),
```

La configuration ci-dessus correspond à ma situation en local avec le prestataire `free`. Votre cas doit être sans doute différent.

A quoi ressemble l'email envoyé ? Vous trouvez la vue correspondante ici :



La vue pour l'email

Avec ce code :

```
Click here to reset your password: {{ url('password/reset/'.$token) }}
```

html

C'est clair et concis. Voyons ce qu'on obtient en réception :

Click here to reset your password:

<http://.../password/reset/6e8ad7609e1b7cc5a483d81524ca8443f0138b73>

Voyons ce qu'il s'est passé dans la table `password_resets` :

| email | token | created_at |
|---------------|--|---------------------|
| durand@gem.fr | 7ad478ecdcd696c4ebfa649f5f9454f92606220a | 2015-01-28 17:34:14 |

La table `password_reminders`

On a la mémorisation de l'adresse email, du jeton (token) et le moment de la création.

Si on utilise l'url de l'email on est dirigé sur la route `password/reset` et donc sur la méthode

`showResetForm` du trait `ResetsPasswords` :

```
<?php
/**
 * Display the password reset view for the given token.
 *
 * If no token is present, display the link request form.
 *
 * @param \Illuminate\Http\Request $request
 * @param string|null $token
 * @return \Illuminate\Http\Response
 */
public function showResetForm(Request $request, $token = null)
{
    if (is_null($token)) {
```

php

```

        return $this->getEmail();
    }

    $email = $request->input('email');

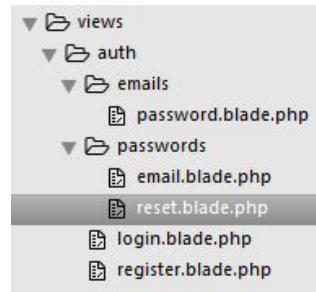
    if (property_exists($this, 'resetView')) {
        return view($this->resetView)->with(compact('token', 'email'));
    }

    if (view()->exists('auth.passwords.reset')) {
        return view('auth.passwords.reset')->with(compact('token', 'email'));
    }

    return view('auth.reset')->with(compact('token', 'email'));
}

```

Le jeton est transmis dans la variable \$token. On vérifie sa présence sinon on renvoie le formulaire. S'il est présent on retourne le formulaire de saisie du nouveau mot de passe avec la vue `reset` dans le dossier `auth/passwords` :



La vue de reset

Avec cet aspect :

Le formulaire pour le nouveau mot de passe

À la soumission on tombe sur la méthode `postReset` du trait :

```

<?php
/**
 * Reset the given user's password.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function postReset(Request $request)
{
    return $this->reset($request);
}

/**
 * Reset the given user's password.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */

```

```

public function reset(Request $request)
{
    $this->validate(
        $request,
        $this->getResetValidationRules(),
        $this->getResetValidationMessages(),
        $this->getResetValidationCustomAttributes()
    );

    $credentials = $request->only(
        'email', 'password', 'password_confirmation', 'token'
    );

    $broker = $this->getBroker();

    $response = Password::broker($broker)->reset($credentials, function ($user, $password) {
        $this->resetPassword($user, $password);
    });

    switch ($response) {
        case Password::PASSWORD_RESET:
            return $this->getResetSuccessResponse($response);

        default:
            return $this->getResetFailureResponse($request, $response);
    }
}

```

On trouve une validation pour les entrées. Si tout se passe bien le nouveau mot de passe est mémorisé et l'utilisateur connecté. Pour la redirection elle va évidemment dépendre du succès ou de l'échec.

En résumé

- L'authentification est totalement et simplement prise en charge par Laravel.
- Un middleware permet d'effectuer un traitement à l'arrivée ou au départ de la requête.
- On peut utiliser les middlewares `Authenticate` et `RedirectIfAuthenticated` pour autoriser ou interdire un accès.
- Un système complet de renouvellement du mot de passe est prévu.



Les ressources (2/2) et les erreurs

La relation 1:n



L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

OpenClassrooms

Qui sommes-nous ?

Fonctionnement de nos cours

Recrutement

Nous contacter

□ Professionals

Affiliation

Entreprises

□

Découvrez le framework PHP Laravel



□ 15 heures □ Moyenne

Licence □ □ □ □



La relation 1:n

□ Connectez-vous ou inscrivez-vous pour bénéficier de toutes les fonctionnalités de ce cours !

Pour le moment nous n'avons manipulé qu'une table avec Eloquent. Dans le présent chapitre nous allons utiliser deux tables et les mettre en relation. La relation la plus répandue et la plus simple entre deux tables est celle qui fait correspondre un enregistrement d'une table à plusieurs enregistrements de l'autre table, on parle de relation de un à plusieurs ou encore de relation de type 1:n. Nous verrons également dans ce chapitre comment créer un middleware.

Comme exemple pour ce chapitre, je vais prendre le cas d'un petit blog personnel avec :

- un affichage des articles,
- des visiteurs qui pourront consulter les articles,
- des utilisateurs enregistrés qui pourront aussi rédiger des articles (donc possibilité de se connecter et se déconnecter),
- des administrateurs qui pourront aussi supprimer des articles.

Pour ne pas trop alourdir le code, je ne vais pas prévoir la modification des articles.

Les données

Les migrations

Nous allons continuer à utiliser la table `users` que nous avons vue aux chapitres précédents. Nous allons créer une nouvelle table `posts` destinée à mémoriser les articles. Si vous avez déjà créé la table `users` avec des enregistrements supprimez cette table, nous allons la recréer.

Nous avons déjà défini la migration de la table `users` et vous devez avoir le fichier dans le dossier `database/migrations`. Je vous en rappelle le code :

```
<?php  
  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Database\Migrations\Migration;  
  
class CreateUsersTable extends Migration {  
  
    public function up()  
    {  
        Schema::create('users', function(Blueprint $table)  
        {  
            $table->increments('id');  
            $table->string('name')->unique();  
            $table->string('email')->unique();  
            $table->string('password', 60);  
            $table->boolean('admin')->default(false);  
        });  
    }  
  
    public function down()  
    {  
        Schema::dropIfExists('users');  
    }  
}
```

| | | |
|---|----------------------|---|
| □ | Les bases de données | □ |
| <ol style="list-style-type: none"> 1. Migrations et modèles 2. Les ressources (1/2) 3. Les ressources (2/2) et les erreurs 4. L'authentification 5. La relation 1:n 6. La relation n:n 7. Les commandes et les assistants 8. Query Builder <p>□ Quiz : Quiz 2</p> <p>□ Activité : Construisez un site de sondages avec une base de données</p> | | |
| Accéder au forum | | |
| □ | □ | □ |

```
        $table->rememberToken();
        $table->timestamps();
    });

    public function down()
    {
        Schema::drop('users');
    }

}
```

On va créer une migration aussi pour la table `posts` :

```
php artisan make:migration create_posts_table
```

Et on va compléter ainsi le code :

```
php
<?php

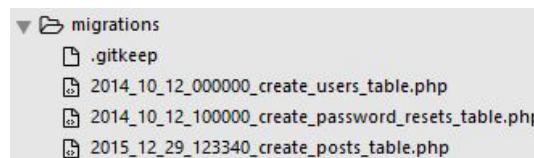
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreatePostsTable extends Migration {

    public function up()
    {
        Schema::create('posts', function(Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
            $table->string('titre', 80);
            $table->text('contenu');
            $table->integer('user_id')->unsigned();
            $table->foreign('user_id')
                ->references('id')
                ->on('users')
                ->onDelete('restrict')
                ->onUpdate('restrict');
        });
    }

    public function down()
    {
        Schema::table('posts', function(Blueprint $table) {
            $table->dropForeign('posts_user_id_foreign');
        });
        Schema::drop('posts');
    }
}
```

Normalement vous devez avoir ces 3 migrations :



Les 3 migrations

Lancez la migration :

```
php artisan migrate
```

Vous devez ainsi vous retrouver avec les trois tables dans votre base ainsi que la table `migrations` :



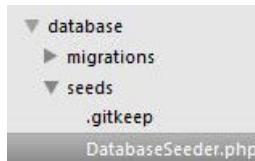
Les 4 tables



Pour que la population que nous allons faire ensuite fonctionne il faut repartir sur une nouvelle migration pour la table `users`. En effet on va avoir besoin que la clé de la table commence à 1.

La population

Nous allons voir maintenant comment remplir nos tables avec des enregistrements pour faire nos essais. Nous allons pour cela créer deux fichiers dans le dossier `database/seeds`. Normalement vous devez déjà avoir dans ce dossier le fichier `DatabaseSeeder.php`:



Le fichier DatabaseSeeder

Avec ce code :

```

<?php
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        // $this->call(UserTableSeeder::class);
    }
}
  
```

La méthode `run` est destinée à exécuter les fichiers pour la population. Vous avez déjà la ligne commentée de lancement pour la table `users`. Nous allons donc la dé-commenter et ajouter le code pour la table posts :

```

<?php
public function run()
{
    $this->call(UserTableSeeder::class);
    $this->call(PostTableSeeder::class);
}
  
```

Mettez bien les lignes dans cet ordre, vous comprendrez bientôt pourquoi c'est nécessaire.

Ensuite on va créer le fichier `UserTableSeeder.php` pour la population de la table `users` :

```

<?php
use Illuminate\Database\Seeder;

class UserTableSeeder extends Seeder {

    public function run()
    {
        DB::table('users')->delete();

        for($i = 0; $i < 10; ++$i)
        {
            DB::table('users')->insert([
                'name' => 'Nom' . $i,
                'email' => 'email' . $i . '@blob.fr',
                'password' => bcrypt('password' . $i),
                'admin' => rand(0, 1)
            ]);
        }
    }
}
  
```

}

On va créer ainsi 10 utilisateurs.

On prévoit aussi le fichier `PostTableSeeder.php` avec ce code :

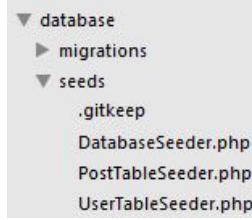
```
<?php  
  
use Illuminate\Database\Seeder;  
use Carbon\Carbon;  
  
class PostTableSeeder extends Seeder {  
  
    private function randDate()  
    {  
        return Carbon::createFromDate(null, rand(1, 12), rand(1, 28));  
    }  
  
    public function run()  
    {  
        DB::table('posts')->delete();  
  
        for($i = 0; $i < 100; ++$i)  
        {  
            $date = $this->randDate();  
            DB::table('posts')->insert([  
                'titre' => 'Titre' . $i,  
                'contenu' => 'Contenu' . $i . ' Lorem ipsum dolor sit amet, consectetur  
adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim  
veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute  
irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur  
sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.',  
                'user_id' => rand(1, 10),  
                'created_at' => $date,  
                'updated_at' => $date  
            ]);  
        }  
    }  
}
```

On va ainsi créer 100 articles affectés de façon aléatoire aux 10 utilisateurs. Nous allons voir bientôt comment s'effectue la liaison entre les deux.



La classe [Carbon](#), issue d'un package chargé par Laravel, permet la manipulation facile des dates. N'hésitez pas à l'utiliser dès que vous avez des dates à gérer.

Vous devez avoir maintenant ces fichiers dans le dossier des populations :



Les fichiers de population

Il suffit maintenant d'utiliser la commande d'Artisan pour lancer la population :

```
php artisan db:seed
```

Normalement vous devez avoir les deux tables remplies à l'issue de cette commande.



Si vous avez un message vous disant que la classe `userTableSeeder` , ou l'autre classe, n'est pas trouvée effectuez la commande "composer dumpautoload", puis relancez la population.

Les tables doivent maintenant être garnies, par exemple pour les utilisateurs (users) :

| id | name | email | password | admin |
|-----------|-------------|----------------|---|--------------|
| 1 | Nom0 | email0@blop.fr | \$2y\$10\$ApuwKOZdqeY1ZCi0YPHpeuTDaecarM93Vkk1NNAeREMP... | 1 |
| 2 | Nom1 | email1@blop.fr | \$2y\$10\$yZN9Z1uw2iSh6/6Cj3.CJO8lFfpe6EQUtk5uo9IIQIR... | 1 |
| 3 | Nom2 | email2@blop.fr | \$2y\$10\$k7xHrtOVBFUCv5Ei.Ju7h.OSLyqaht/iRCef.LJOWGT... | 0 |
| 4 | Nom3 | email3@blop.fr | \$2y\$10\$WL7UwNftEDLNFX2b.i9Qn.Ow01ZT3sgtRdwYy7gVI... | 1 |
| 5 | Nom4 | email4@blop.fr | \$2y\$10\$5r2Bp8HFtKCxB9fpR/ogCeWunTtempqdoZm6ONwktS... | 0 |
| 6 | Nom5 | email5@blop.fr | \$2y\$10\$PXYiimun0D1Noygfa/kONsgpUrfiO.SydByji3qvz... | 0 |
| 7 | Nom6 | email6@blop.fr | \$2y\$10\$ubXPT/Wh9f65rCHU5i./Lehsq51OZTJju8rFn1OW4Pe... | 0 |
| 8 | Nom7 | email7@blop.fr | \$2y\$10\$jsurpKAc3qfLBZ7ycbztO.0nbS9xl8/miuQI6R8wDmx... | 0 |
| 9 | Nom8 | email8@blop.fr | \$2y\$10\$9EF.gS8x53nzwHLbGp5Rf.mX7ckTsKiVLnJ11LrqGFP... | 1 |
| 10 | Nom9 | email9@blop.fr | \$2y\$10\$a84ghb9xhXiVEHjfUf6XZe5OiNbtRJfjBRz5N0Lk.xj... | 1 |

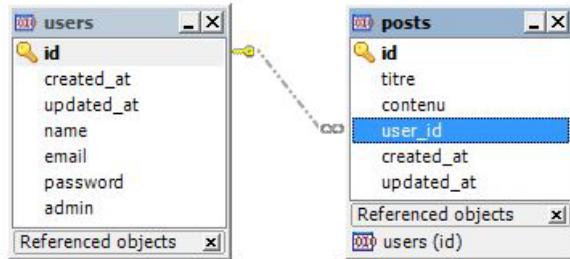
La table users

La relation

On a la situation suivante :

- un utilisateur peut écrire plusieurs articles,
- un article est écrit par un seul utilisateur.

Il faut trouver un moyen pour référencer cette relation dans les tables. Le principe est simple : on prévoit dans la table `posts` une ligne destinée à recevoir l'identifiant de l'utilisateur rédacteur de l'article. On appelle cette ligne une clé étrangère parce qu'on enregistre ici la clé d'une autre table. Voici une représentation visuelle de cette relation :



La clé étrangère

Vous voyez la relation dessinée entre la clé `id` dans la table `users` et la clé étrangère `user_id` dans la table `posts`. La migration que l'on a créée ci-dessus est destinée aussi à informer la base de cette relation. Regardez ce code :

```

<?php
$table->foreign('user_id')
    ->references('id')
        ->on('users')
        ->onDelete('restrict')
        ->onUpdate('restrict');
  
```

php

Dans la table on déclare une clé étrangère (`foreign`) nommée `user_id` qui référence (`references`) la ligne `id` dans la table (`on`) `users`. En cas de suppression (`onDelete`) ou de modification (`onUpdate`) on a une restriction (`restrict`). Que signifient ces deux dernières conditions ?

Imaginez que vous avez un utilisateur avec l'id 5 qui a deux articles, donc dans la table `posts` on a deux enregistrements avec `user_id` qui a la valeur 5. Si on supprime l'utilisateur que va-t-il se passer ? On risque de se retrouver avec nos deux enregistrements dans la table `posts` avec une clé étrangère qui ne correspond à aucun enregistrement dans la table `users`. En mettant "restrict" on empêche la suppression d'un utilisateur qui a des articles. On doit donc commencer par supprimer ses

articles avant de le supprimer lui-même. On dit que la base assure l'intégrité référentielle. Elle n'acceptera pas non plus qu'on utilise pour `user_id` une valeur qui n'existe pas dans la table `users`.

Une autre possibilité est "cascade" à la place de "restrict". Dans ce cas si vous supprimez un utilisateur ça supprimera en cascade les articles de cet utilisateur. C'est une option qui est rarement utilisée parce qu'elle peut s'avérer dangereuse, surtout dans une base comportant de multiples tables en relation. Mais c'est aussi une stratégie très efficace parce que c'est le moteur de la base de données qui se charge de gérer les enregistrements en relation, vous n'avez ainsi pas à vous en soucier au niveau du code.

On pourrait aussi ne pas signaler à la base qu'il existe une relation et la gérer seulement dans notre code. Mais c'est encore plus dangereux parce que la moindre erreur de gestion des enregistrements dans votre code risque d'avoir des conséquences importantes dans votre base avec de multiples incohérences.

Les modèles

Nous avons déjà un modèle `User` (`app/User.php`). Il va juste falloir ajouter une méthode pour pouvoir facilement aller trouver les articles d'un utilisateur. Ajoutez ce code dans le modèle `User`:

```
<?php  
public function posts()  
{  
    return $this->hasMany('App\Post');  
}
```

php

On déclare ici qu'un utilisateur a plusieurs (`hasMany`) articles (`posts`). On aura ainsi une méthode pratique pour récupérer les articles d'un utilisateur.



Soyez vigilant avec les espaces de noms !

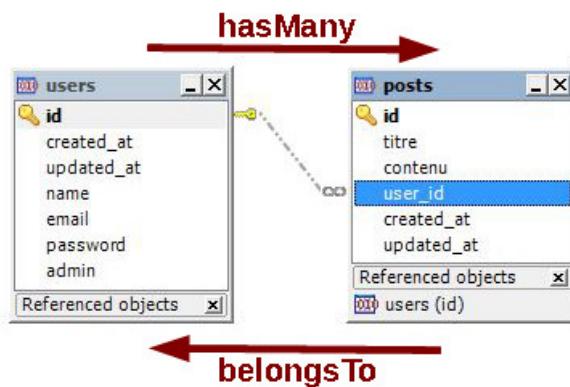
Il nous faut aussi le modèle Post :

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class Post extends Model  
{  
  
    protected $fillable = ['titre','contenu','user_id'];  
  
    public function user()  
    {  
        return $this->belongsTo('App\User');  
    }  
}
```

php

Ici on a la méthode `user` (au singulier) qui permet de trouver l'utilisateur auquel appartient (`belongsTo`) l'article. C'est donc la réciproque de la méthode précédente.

Voici une schématisation de cette relation avec les deux méthodes :



Les deux méthodes de la relation

Je vous rappelle que si vous ne spécifiez pas de manière explicite le nom de la table dans un modèle, Laravel le déduit à partir du nom du modèle en le mettant au pluriel (à la mode anglaise) et en mettant la première lettre en minuscule.

Les deux méthodes mises en place permettent de récupérer facilement un enregistrement lié. Par exemple pour avoir tous les articles de l'utilisateur qui a l'id 1 :

```
<?php
$articles = App\User::find(1)->posts;
```

De la même manière on peut trouver l'utilisateur qui a écrit l'article d'id 1 :

```
<?php
$user = App\Post::find(1)->user;
```

Vous voyez que le codage devient limpide avec ces méthodes :).

Laravel dispose de l'outil **tinker** qui permet d'entrer des commandes dans la console et ainsi interagir directement avec l'application. Il faut le démarrer avec la commande **php artisan tinker**. On peut ensuite l'utiliser directement :

```
php artisan tinker
Psy Shell v0.6.1 (PHP 5.5.12 ÔÇö cli) by Justin Hileman
>>> App\User::find(1)->posts
=> Illuminate\Database\Eloquent\Collection {#661
  all: [
    App\Post {#662
      id: 47,
      created_at: "2015-03-19 12:39:12",
      updated_at: "2015-03-19 12:39:12",
      titre: "Titre46",
      contenu: "Contenu46 Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.",
      user_id: 1,
    },
    ...
  ]
}
```

Le contrôleur et les routes

Le contrôleur

Maintenant que tout est en place au niveau des données voyons un peu la gestion de tout ça. On va créer un contrôleur pour les articles qu'on va appeler `PostController`. Ce contrôleur devra gérer plusieurs chose :

- la réception de la requête pour afficher les articles du blog et la réponse adaptée,
- la réception de la requête pour le formulaire pour créer un nouvel article et son envoi,

- la réception de la soumission du formulaire de création d'un nouvel article (réservé à un utilisateur connecté) et son enregistrement,
- la réception de la demande de suppression d'un article (réservé à un administrateur) et sa suppression.

Pour simplifier je ne vais pas prévoir la possibilité de modifier un article.

Je vais utiliser un contrôleur de ressource. Voici son code :

```
<?php
namespace App\Http\Controllers;

use App\Repositories\PostRepository;
use App\Http\Requests\PostRequest;

class PostController extends Controller
{
    protected $postRepository;

    protected $nbrPerPage = 4;

    public function __construct(PostRepository $postRepository)
    {
        $this->middleware('auth', ['except' => 'index']);
        $this->middleware('admin', ['only' => 'destroy']);

        $this->postRepository = $postRepository;
    }

    public function index()
    {
        $posts = $this->postRepository->getPaginate($this->nbrPerPage);
        $links = $posts->render();

        return view('posts.liste', compact('posts', 'links'));
    }

    public function create()
    {
        return view('posts.add');
    }

    public function store(PostRequest $request)
    {
        $inputs = array_merge($request->all(), ['user_id' => $request->user()->id]);

        $this->postRepository->store($inputs);

        return redirect(route('post.index'));
    }

    public function destroy($id)
    {
        $this->postRepository->destroy($id);

        return redirect()->back();
    }
}
```

php

Comme à l'accoutumée j'injecte la requête de formulaire et le repository.

Notez l'utilisation des middleware pour filtrer les utilisateurs, nous allons voir cela un peu plus loin.

Les routes

On a vu dans le chapitre sur les ressources comment créer les routes de ce genre de contrôleur. Il va juste falloir indiquer qu'on ne veut pas utiliser les 7 méthodes disponibles mais juste certaines :

```
<?php
Route::resource('post', 'PostController', ['except' => ['show', 'edit', 'update']]);
```

php

Avec `except` j'indique que je ne veux pas de route pour les 3 méthodes citées. Voici ce que ça donne en utilisant artisan pour visualiser les routes (`php artisan route:list`) :

| Method | URI |
|------------|-------------|
| GET HEAD | post |
| GET HEAD | post/create |
| POST | post |
| DELETE | post/{post} |

| Name | Action |
|--------------|---|
| post.index | App\Http\Controllers\PostController@index |
| post.create | App\Http\Controllers\PostController@create |
| post.store | App\Http\Controllers\PostController@store |
| post.destroy | App\Http\Controllers\PostController@destroy |

Les routes

Le repository

Pour la gestion on va placer les fichiers dans le dossier `app\Repositories` comme nous l'avons déjà fait pour la gestion des utilisateurs :



Le repository

Avec ce code :

```
<?php
namespace App\Repositories;

use App\Post;

class PostRepository
{
    protected $post;

    public function __construct(Post $post)
    {
        $this->post = $post;
    }

    public function getPaginate($n)
    {
        return $this->post->with('user')
            ->orderBy('posts.created_at', 'desc')
            ->paginate($n);
    }

    public function store($inputs)
    {
        $this->post->create($inputs);
    }

    public function destroy($id)
    {
        $this->post->findOrFail($id)->delete();
    }
}
```

Nous allons voir plus loin l'utilité de toutes ces méthodes.

Les middlewares

On a vu que dans le contrôleur on applique deux middlewares :

- **auth** : accès réservé aux utilisateurs authentifiés à part pour la méthode `index` pour afficher le blog,
- **admin** : accès réservé aux administrateurs pour la méthode `destroy`.

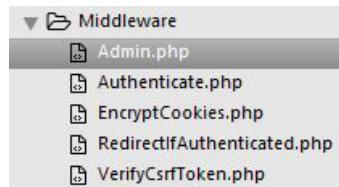
Le premier middleware est déjà prévu dans Laravel, par contre le second n'existe pas. Il faut donc le créer.

Encore une fois nous allons utiliser Artisan :

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

text

On trouve le fichier bien rangé :



Le middleware pour l'administration

On obtient ce code :

```
<?php  
  
namespace App\Http\Middleware;  
  
use Closure;  
  
class Admin  
{  
  
    /**  
     * Handle an incoming request.  
     *  
     * @param \Illuminate\Http\Request $request  
     * @param \Closure $next  
     * @return mixed  
     */  
    public function handle($request, Closure $next)  
    {  
        //  
    }  
}
```

php

On doit écrire notre code, ce qui donne :

```
<?php  
  
namespace App\Http\Middleware;  
  
use Closure;  
use Illuminate\Http\RedirectResponse;  
  
class Admin  
{  
  
    /**  
     * Handle an incoming request.  
     *  
     * @param \Illuminate\Http\Request $request  
     * @param \Closure $next  
     * @return mixed  
     */  
    public function handle($request, Closure $next)  
    {  
        if ($request->user()->admin)  
        {  
            return $next($request);  
        }  
        return new RedirectResponse(url('post'));  
    }  
}
```

php

Si l'utilisateur n'est pas un administrateur on redirige sur l'affichage du blog. Remarquez qu'on ne vérifie pas à ce niveau qu'on a un utilisateur authentifié parce que dans le constructeur du contrôleur le filtre auth est placé avant le filtre admin. Si c'était l'inverse on tomberait évidemment sur une erreur

en cas de tentative d'accès à l'url pour la suppression d'un article.

On a créé le middleware mais ça ne suffit pas, il faut maintenant un lien entre le nom qu'on veut donner au filtre et la classe qu'on vient de créer. Regardez dans le fichier `app/Http/Kernel.php` ces lignes de code :

```
<?php
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
];
```

php

Vous trouvez ici tous les middlewares déclarés, il suffit d'ajouter le nouveau :

```
<?php
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'admin' => \App\Http\Middleware\Admin::class,
];
```

php

La validation

Voyons maintenant la validation. Il faut encore créer une requête de formulaire :

```
php artisan make:request PostRequest
```

text

Elle se place dans le dossier :



La requête de formulaire

Et on complète ainsi le code :

```
<?php
namespace App\Http\Requests;

use App\Http\Requests\Request;

class PostRequest extends Request
{

    public function authorize()
    {
        return true;
    }

    public function rules()
    {
        return [
            'titre' => 'required|max:80',
            'contenu' => 'required'
        ];
    }
}
```

php

Fonctionnement

La liste des articles

La liste des articles est obtenue avec l'url (verbe `get`) :

```
.../post
```

Qui arrive sur la méthode `index` du contrôleur :

```
<?php
```

php

```
public function index()
{
    $posts = $this->postRepository->getPaginate($this->nbrPerPage);
    $links = $posts->render();

    return view('posts.liste', compact('posts', 'links'));
}
```

Ici on envoie le nombre d'articles par page (placé dans la propriété `$nbrPerPage`) à la méthode `getPaginate` du repository :

```
<?php
public function getPaginate($n)
{
    return $this->post->with('user')
        ->orderBy('posts.created_at', 'desc')
        ->paginate($n);
}
```

On veut les articles avec (`with`) l'utilisateur (`user`), dans l'ordre des dates de création (`posts.created_at`) descendant (`desc`) avec une pagination de n articles (`$n`).

 Il existe la méthode `latest` (et `oldest` pour l'inverse) qui permet de simplifier la syntaxe :

```
<?php
return $this->post->with('user')
->latest('posts.created_at')
->paginate($n);
```

L'ajout d'un article

La demande du formulaire de création d'un article se fait avec l'url (verbe `get`) :

```
.../post/create
```

Le contrôleur renvoie directement la vue :

```
<?php
public function create()
{
    return view('posts.add');
}
```

Le retour du formulaire se fait avec l'url (verbe `post`) :

```
.../post
```

On arrive sur la méthode `store` du contrôleur :

```
<?php
public function store(PostRequest $request)
{
    $inputs = array_merge($request->all(), ['user_id' => $request->user()->id]);

    $this->postRepository->store($inputs);

    return redirect(route('post.index'));
}
```

On injecte la requête de formulaire, je n'insiste pas parce qu'il n'y a rien de nouveau à ce niveau. On récupère les entrées du formulaire pour le titre et le contenu. Pour l'identifiant de l'utilisateur on sait qu'il est forcément connecté alors on récupère cet identifiant avec la requête. Si la validation se passe bien on envoie à la méthode `store` du repository :

```
<?php
public function store($inputs)
{
    $this->post->create($inputs);
}
```

Suppression d'un article

Enfin on supprime un article avec l'url (verbe `delete`) :

.../post/{id}

Où {id} représente l'identifiant de l'article à supprimer. On tombe sur la méthode `destroy` du contrôleur :

```
<?php
public function destroy($id)
{
    $this->postRepository->destroy($id);

    return redirect()->back();
}
```

php

Qui envoie à la méthode `destroy` du repository :

```
<?php
public function destroy($id)
{
    $this->post->findOrFail($id)->delete();
}
```

php

Là on supprime l'article avec la méthode `delete` du modèle.

Les vues

Voyons à présent les vues. On va un peu modifier notre template

(`resources/views/template.blade.php`) :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Mon joli site</title>
        {!!
Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css') !!}
{!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css') !!}
        <!--[if lt IE 9]>
            {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.js') }}
            {{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js') }}
        <!--[endif]-->
        <style> textarea { resize: none; } </style>
    </head>
    <body>
        <header class="jumbotron">
            <div class="container">
                <h1 class="page-header">{!! link_to_route('post.index', 'Mon joli blog') !!}</h1>
                @yield('header')
            </div>
        </header>
        <div class="container">
            @yield('contenu')
        </div>
    </body>
</html>
```

html

Nous avons besoin d'une vue pour afficher les articles du blog et quelques boutons pour la gestion (`resources/views/posts/liste.blade.php`) :

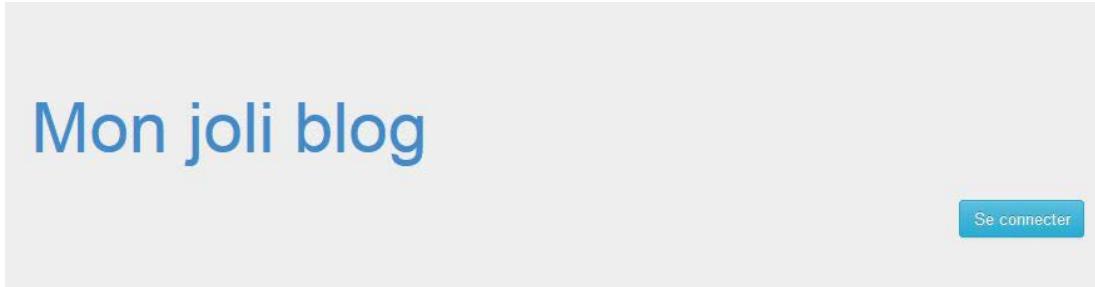
```
@extends('template')

@section('header')
@if(Auth::check())
    <div class="btn-group pull-right">
        {!! link_to_route('post.create', 'Créer un article', [], ['class' => 'btn btn-info']) !!}
        {!! link_to('logout', 'Deconnexion', ['class' => 'btn btn-warning']) !!}
    </div>
@else
    {!! link_to('login', 'Se connecter', ['class' => 'btn btn-info pull-right']) !!}
@endif
@endsection
```

html

```
@section('contenu')
@if(isset($info))
    <div class="row alert alert-info">{{ $info }}</div>
@endif
{!! $links !!}
@foreach($posts as $post)
    <article class="row bg-primary">
        <div class="col-md-12">
            <header>
                <h1>{{ $post->titre }}</h1>
            </header>
            <hr>
            <section>
                <p>{{ $post->contenu }}</p>
                @if(Auth::check() and Auth::user()->admin)
                    {!! Form::open(['method' => 'DELETE', 'route' =>
['post.destroy', $post->id]]) !!}
                    {!! Form::submit('Supprimer cet article',
['class' => 'btn btn-danger btn-xs', 'onclick' => 'return confirm(\'Vraiment supprimer cet article ?\')']) !!}
                    {!! Form::close() !!}
                @endif
                <em class="pull-right">
                    <span class="glyphicon glyphicon-pencil"></span> {{ $post->user->name }} le {!! $post->created_at->format('d-m-Y') !!}
                </em>
            </section>
        </div>
    </article>
    <br>
@endforeach
{!! $links !!}
@endsection
```

Avec cet aspect pour un utilisateur non connecté :



Titre2

Contenu2 Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Nom0 le 23-12-2015

Titre66

Contenu66 Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Nom7 le 17-12-2015

L'aspect du blog

Le bouton « Se connecter » envoie sur l'url :

.../login

Ce qui correspond à ce que nous avons vu au chapitre précédent et doit aboutir sur le formulaire de

connexion si vous avez tout en place comme nous l'avons prévu précédemment :

Le formulaire de connexion



Pour mémoire j'utilise ici les vues par défaut de Laravel. Libre à vous évidemment de franciser ces vues ou, encore mieux, les adapter au langage de l'utilisateur (nous verrons cet aspect dans un chapitre ultérieur).

Pour que votre application fonctionne bien avec le contrôleur `AuthController` il faut modifier la propriété `redirectTo` dans ce contrôleur :

```
php
<?php
protected $redirectTo = 'post';
```

Ainsi vous serez bien redirigé vers le blog en cas de connexion.

De la même manière il faut prévoir une redirection après déconnexion dans le même contrôleur :

```
php
<?php
protected $redirectAfterLogout = 'post';
```

La génération des articles dans la vue se fait avec un `foreach` :

```
html
@foreach($posts as $post)
...
@endforeach
```

Si vous vous connectez avec un utilisateur qui n'est pas administrateur (regardez dans votre table pour en trouver un, comme la population est aléatoire on ne sait pas à l'avance qui l'est et qui ne l'est pas). Vous retournez au blog avec deux boutons supplémentaires :

[Créer un article](#) [Déconnexion](#)

Les boutons pour un utilisateur de base

On utilise une condition pour adapter la page :

```
html
@if(Auth::check())
...
@else
...
@endif
```

La méthode `check` de la classe `Auth` permet de savoir si l'utilisateur est connecté.



Il existe l'helper `auth()` qui vous évite le recours à la façade :

```
php
@if(auth()->check())
```

Le premier bouton « Créer un article » génère l'url :

.../post/create

Ce qui a pour effet d'obtenir le formulaire de création que nous allons bientôt voir.

Le second bouton « Deconnexion » génère l'url :

.../logout

Qui correspond aussi à ce que nous avons vu dans le chapitre précédent.

Si l'utilisateur connecté est un administrateur alors il a en plus pour chaque article un bouton de suppression :



Le bouton de suppression

On utilise encore une condition pour détecter un administrateur :

```
@if(Auth::check() and Auth::user()->admin)
...
@endif
```

Il faut que l'utilisateur soit connecté (`check()`) et que ce soit un administrateur (`user()->admin`).

J'ai prévu une confirmation de la suppression avec un peu de Javascript.

Voici maintenant la vue pour le formulaire de création d'un article

(`resources/views/posts/add.blade.php`) :

```
@extends('template')

@section('contenu')
    <br>
    <div class="col-sm-offset-3 col-sm-6">
        <div class="panel panel-info">
            <div class="panel-heading">Ajout d'un article</div>
            <div class="panel-body">
                {!! Form::open(['route' => 'post.store']) !!}
                    <div class="form-group {!! $errors->has('titre') ? 'has-error' : '' !!}">
                        <input type="text" name="titre" value="Titre" placeholder="Titre" />
                        @error('titre')
                            <small class="help-block">{{ $message }}</small>
                        @enderror
                    </div>
                    <div class="form-group {!! $errors->has('contenu') ? 'has-error' : '' !!}">
                        <input type="text" name="contenu" value="Contenu" placeholder="Contenu" />
                        @error('contenu')
                            <small class="help-block">{{ $message }}</small>
                        @enderror
                    </div>
                    <div class="form-group">
                        <input type="submit" value="Envoyer !!" class="btn btn-primary pull-right" />
                    </div>
                {!! Form::close() !!}
            </div>
        </div>
        <a href="javascript:history.back()" class="btn btn-primary">
            <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
        </a>
    </div>
@endsection
```

Il n'y a rien de bien nouveau dans cette vue. Voici son apparence :

Ajout d'un article

Titre

Contenu

Envoyer !

Retour

Formulaire de création d'un article

On peut entrer le titre et le contenu qui sera ensuite validé dans le contrôleur et enregistré si tout se passe bien.



Il arrive parfois que les vues ne se génèrent pas correctement. Laravel utilise un cache pour les vues en `storage/framework/views`. Vous pouvez sans problème supprimer tous les fichiers (mais pas `.gitignore` !) pour obliger Laravel à générer de nouvelles vues. Il existe une commande Artisan pour le faire :

```
php artisan view:clear
```

En résumé

- Une relation de type **1:n** nécessite la création d'une clé étrangère côté **n**.
- On peut remplir les tables d'enregistrements avec la population.
- Une relation dans la base nécessite la mise en place de méthodes spéciales dans les modèles.
- Avec les middlewares il est facile de gérer l'accès aux méthodes des contrôleurs.



L'authentification



La relation n:n

L'auteur

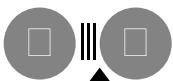
Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

[S'inscrire](#)[Se connecter](#)[Accueil](#) ▶ [Cours](#) ▶ [Découvrez le framework PHP Laravel](#) ▶ [La relation n:n](#)

Découvrez le framework PHP Laravel

15 heures Moyenne



La relation n:n

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Dans le précédent chapitre nous avons vu la relation de type 1:n, la plus simple et la plus répandue. Nous allons maintenant étudier la relation de type n:n, plus délicate à comprendre et à mettre en œuvre. Nous allons voir qu'Eloquent permet de simplifier la gestion de ce type de relation.

Je vais poursuivre l'exemple du blog personnel débuté au chapitre précédent en ajoutant la possibilité d'ajouter des mots-clés (`tags`) aux articles. Ce chapitre est un peu long mais j'ai préféré tout rassembler ici.

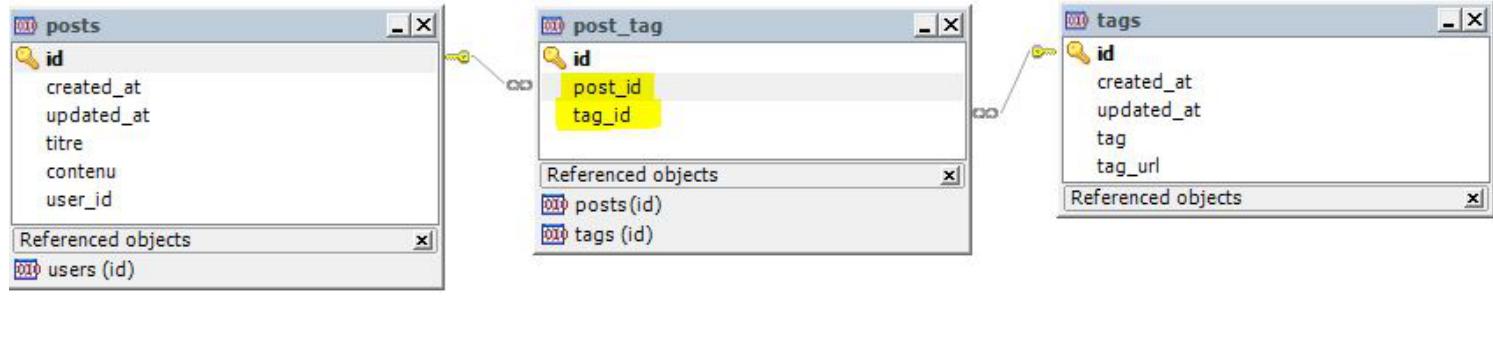
Les données

La relation n:n

Imaginez une relation entre deux tables A et B qui permet de dire :

- je peux avoir une ligne de la table A en relation avec plusieurs lignes de la table B,
- je peux avoir une ligne de la table B en relation avec plusieurs lignes de la table A.

Cette relation ne se résout pas comme nous l'avons vu au chapitre précédent avec une simple clé étrangère dans une des tables. En effet il nous faudrait des clés dans les deux tables et plusieurs clés, ce qui n'est pas possible à réaliser. La solution consiste à créer une table intermédiaire (nommée table pivot) qui sert à mémoriser les clés étrangères. Voici un schéma de ce que nous allons réaliser :



La table pivot

La table pivot `post_tag` contient les clés des deux tables :

- **post_id** pour mémoriser la clé de la table `posts`,
- **tag_id** pour mémoriser la clé de la table `tags`.

De cette façon on peut avoir plusieurs enregistrements liés entre les deux tables, il suffit à chaque fois d'enregistrer les deux clés dans la table pivot. Évidemment au niveau du code ça demande un peu d'intendance parce qu'il y a une table supplémentaire à gérer.

Par convention le nom de la table pivot est composé des deux noms des tables au singulier pris dans l'ordre alphabétique.

Les migrations

Nous allons continuer à utiliser les tables `users` et `posts` que nous avons vues aux chapitres précédents. Nous allons créer une nouvelle table `tags` destinée à mémoriser les mots-clés. Commencez par supprimer toutes les tables de votre base de données, sinon vous risquez de tomber sur des conflits avec les enregistrements que nous allons créer.

Supprimez aussi la table migrations.

Normalement vous devez déjà disposer des migrations pour les tables `users`, `password_resets` et `posts`. Nous allons ajouter les deux tables : `tags` et `post_tag`.

Créez une nouvelle migration pour la table `tags` :

```
php artisan make:migration create_tags_table
```

Et entrez ce code :

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateTagsTable extends Migration {

    public function up()
    {
```

php

```

Schema::create('tags', function(Blueprint $table) {
    $table->increments('id');
    $table->timestamps();
    $table->string('tag', 50)->unique();
    $table->string('tag_url', 60)->unique();
});
}

public function down()
{
    Schema::drop('tags');
}
}

```

On prévoit les champs :

- **id** : clé unique incrémentée,
- **created_at** et **updated_at** créées par timestamps,
- **tag** : le mot clé unique limité à 50 caractères,
- **tag_url** : la version du tag à inclure dans l'url (avec 60 comme limite pour couvrir les cas les plus défavorables).

Il nous faut deux champs pour le tag, en effet il va falloir qu'on le transmette dans l'url pour la recherche par tag, or l'utilisateur risque de rentrer des accents par exemple (ou pire des "/"), nous allons convertir ces caractères spéciaux en caractères adaptés aux urls.

Créez une nouvelle migration pour la table `post_tag` :

```
php artisan make:migration create_post_tag_table
```

Et entrez ce code :

```

<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreatePostTagTable extends Migration {

    public function up()
    {
        Schema::create('post_tag', function(Blueprint $table) {
            $table->increments('id');
            $table->integer('post_id')->unsigned();
            $table->integer('tag_id')->unsigned();
            $table->foreign('post_id')->references('id')->on('posts')
                ->onDelete('restrict')
                ->onUpdate('restrict');

            $table->foreign('tag_id')->references('id')->on('tags')
                ->onDelete('restrict')
                ->onUpdate('restrict');
        });
    }

    public function down()
    {
        Schema::table('post_tag', function(Blueprint $table) {
            $table->dropForeign('post_tag_post_id_foreign');
            $table->dropForeign('post_tag_tag_id_foreign');
        });
    }
}

```

php

```

    });
    Schema::drop('post_tag');
}
}

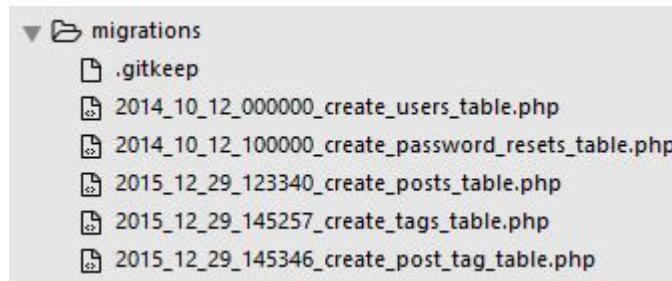
```

On prévoit les champs :

- **post_id** : clé étrangère pour la table posts,
- **tag_id** : clé étrangère pour la table tags.

J'ai encore prévu l'option "restrict" pour les cascades pour sécuriser les opérations sur la base.

Normalement vous devez avoir ces migrations :



Les migrations

Lancez les migrations :

```
php artisan migrate
```

Vous devez ainsi vous retrouver avec ces 6 tables dans votre base :



Les tables

 Pour ne pas recevoir d'erreur il faut que les migrations se fassent dans le bon ordre ! L'ordre des migrations est donné par leur date de création, il suffit donc de changer la date dans le nom des migrations pour en changer l'ordre.

La population

Vous avez déjà les fichiers pour les tables `users` et `posts`. On va créer celui pour les tags (`TagTableSeeder.php`) :

```
<?php
```

php

```
use Illuminate\Database\Seeder;
use Carbon\Carbon;

class TagTableSeeder extends Seeder {

    private function randDate()
    {
        return Carbon::createFromDate(null, rand(1, 12), rand(1, 28));
    }

    public function run()
    {
        DB::table('tags')->delete();

        for($i = 0; $i < 20; ++$i)
        {
            $date = $this->randDate();
            DB::table('tags')->insert(array(
                'tag' => 'tag' . $i,
                'tag_url' => 'tag' . $i,
                'created_at' => $date,
                'updated_at' => $date
            ));
        }
    }
}
```

On aura ainsi 20 tags.

On crée aussi le fichier pour la table pivot (`PostTagTableSeeder.php`) :

```
<?php
use Illuminate\Database\Seeder;

class PostTagTableSeeder extends Seeder {

    public function run()
    {
        for($i = 1; $i <= 100; ++$i)
        {
            $numbers = range(1, 20);
            shuffle($numbers);
            $n = rand(3, 6);
            for($j = 1; $j < $n; ++$j)
            {
                DB::table('post_tag')->insert(array(
                    'post_id' => $i,
                    'tag_id' => $numbers[$j]
                ));
            }
        }
    }
}
```

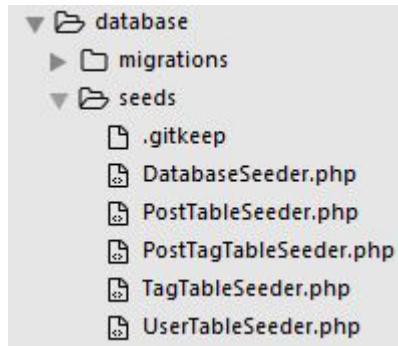
De façon aléatoire on crée plusieurs affectations de tags aux articles.

Il ne reste plus qu'à mettre à jour le fichier DatabaseSeeder.php :

```
<?php
use Illuminate\Database\Seeder;
```

```
class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        $this->call(UserTableSeeder::class);
        $this->call(PostTableSeeder::class);
        $this->call(TagTableSeeder::class);
        $this->call(PostTagTableSeeder::class);
    }
}
```

Vous devez avoir ces fichiers :



Les fichiers de population

Il ne reste plus qu'à lancer la population :

```
php artisan db:seed
```

Info Si vous recevez un message vous disant que l'une des classes n'existe pas lancez un **composer dumpautoload**. Si les tables avaient commencé à se remplir repartez de zéro en les supprimant toutes.

Astuce Il est possible d'effectuer en bloc une migration et une population avec la syntaxe :

```
php artisan migrate --seed
```

text

Les modèles

On va avoir besoin de déclarer la relation n:n dans le modèle **Post** :

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Post extends Model
{
    protected $fillable = ['titre', 'contenu', 'user_id'];
    public function user()
```

php

```

    {
        return $this->belongsTo('App\User');
    }

    public function tags()
    {
        return $this->belongsToMany('App\Tag');
    }

}

```

La méthode `tags()` permettra de récupérer les tags qui sont en relation avec l'article. On utilise la méthode `belongsToMany` d'Eloquent pour le faire.

On va aussi avoir besoin d'un modèle pour les tags :

```

<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

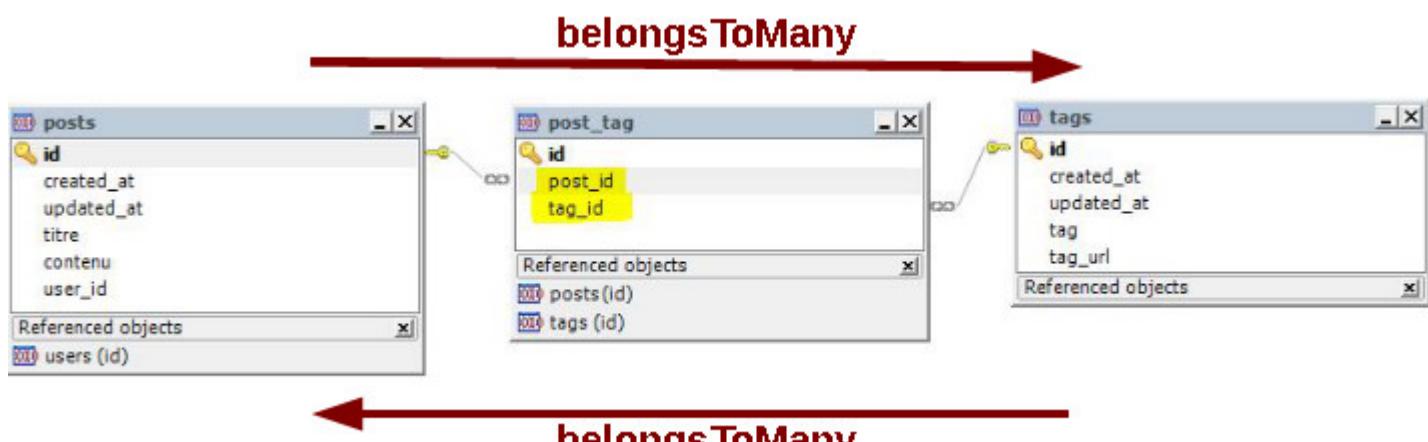
class Tag extends Model
{
    protected $fillable = ['tag', 'tag_url'];

    public function posts()
    {
        return $this->belongsToMany('App\Post');
    }
}

```

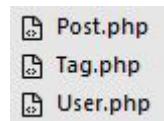
On a la méthode réciproque de la précédente : `posts()` permettra de récupérer les articles en relation avec le tag.

Voici une schématisation de cette relation avec les deux méthodes symétriques :



La relation n:n

On se retrouve avec ces trois modèles :



Les 3 modèles

On pourrait créer un dossier pour les ranger, mais comme il y en a peu on va les garder comme cela. Si on le faisait il faudrait adapter les espaces de noms en conséquence. Il faudrait surtout bien renseigner l'espace de nom dans le fichier `config/auth.php` pour le modèle `User`.

La validation

Nous allons avoir un cas de validation un peu particulier. En effet comme je l'ai dit ci-dessus les tags vont être entrés dans un contrôle de texte séparés par des virgules. On a prévu dans la table `tags` qu'ils ne devraient pas dépasser 50 caractères. On ne dispose pas dans l'arsenal des règles de validation de Laravel d'une telle possibilité, il va donc falloir la créer.

On a déjà créé une classe `PostRequest` dans le chapitre précédent, il faut ajouter la règle pour les tags :

```
<?php
namespace App\Http\Requests;

use App\Http\Requests\Request;

class PostRequest extends Request
{

    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'titre' => 'required|max:80',
            'contenu' => 'required',
            'tags' => ['Regex:/^([A-Za-z0-9-éèàù]{1,50})(,[A-Za-z0-9-éèàù]{1,50})*$/']
        ];
    }
}
```

php

Comme le cas est particulier j'ai utilisé une expression rationnelle. Il ne reste plus qu'à traiter le message.

Si vous regardez dans le fichier `resources/lang/en/validation.php` vous trouvez ce code :

php

```
<?php
'custom' => [
    'attribute-name' => [
        'rule-name' => 'custom-message',
    ],
],
]
```

C'est ici qu'on peut ajouter des messages spécifiques. On va donc écrire :

php

```
<?php
'custom' => [
    'tags' => [
        'regex' => "tags, separated by commas (no spaces), should have a maximum of 50 characters.",
    ],
],
]
```

On va faire la même chose dans le fichier du Français :

php

```
<?php
'custom' => [
    'tags' => [
        'regex' => "Les mots-clés, séparés par des virgules (sans espaces), doivent avoir au maximum 50 caractères alphanumériques.",
    ],
],
]
```

La gestion

Le contrôleur et les routes

Maintenant que tout est en place au niveau des données et de la validation voyons un peu la gestion de tout ça.

On a déjà un contrôleur `PostController` mais on doit le compléter pour le fonctionnement avec les tags :

php

```
<?php

namespace App\Http\Controllers;

use App\Repositories\PostRepository;
use App\Repositories\TagRepository;
use App\Http\Requests\PostRequest;

class PostController extends Controller
{

    protected $postRepository;

    protected $nbrPerPage = 4;

    public function __construct(PostRepository $postRepository)
    {
        $this->middleware('auth', ['except' => ['index', 'indexTag']]);
        $this->middleware('admin', ['only' => 'destroy']);

        $this->postRepository = $postRepository;
    }

    public function index()
    {
        $posts = $this->postRepository->getWithUserAndTagsPaginate($this->nbrPerPage);
        $links = $posts->render();

        return view('posts.liste', compact('posts', 'links'));
    }
}
```

```

public function create()
{
    return view('posts.add');
}

public function store(PostRequest $request, TagRepository $tagRepository)
{
    $inputs = array_merge($request->all(), ['user_id' => $request->user()->id]);

    $post = $this->postRepository->store($inputs);

    if(isset($inputs['tags']))
    {
        $tagRepository->store($post, $inputs['tags']);
    }

    return redirect(route('post.index'));
}

public function destroy($id)
{
    $this->postRepository->destroy($id);

    return redirect()->back();
}

public function indexTag($tag)
{
    $posts = $this->postRepository->getWithUserAndTagsForTagPaginate($tag, $this->nbrPerPage);
    $links = $posts->render();

    return view('posts.liste', compact('posts', 'links'))
        ->with('info', 'Résultats pour la recherche du mot-clé : ' . $tag);
}
}

```

J'ai ajouté la méthode `indexTag` qui doit lancer la recherche des articles qui comportent ce tag et envoyer les informations dans la vue `liste`. J'ai aussi un peu remanié le code.

Il faut ajouter la route pour aboutir sur cette nouvelle méthode :

```

<?php
Route::resource('post', 'PostController', ['except' => ['show', 'edit', 'update']]);
Route::get('post/tag/{tag}', 'PostController@indexTag');

```

Les repositories

Voici le repository pour les articles (`app/Repositories/PostRepository.php`) modifié pour tenir compte des tags :

```

<?php namespace App\Repositories;

use App\Post;

class PostRepository {

    protected $post;

    public function __construct(Post $post)
    {
        $this->post = $post;
    }
}

```

```

private function queryWithUserAndTags()
{
    return $this->post->with('user', 'tags')
        ->orderBy('posts.created_at', 'desc');
}

public function getWithUserAndTagsPaginate($n)
{
    return $this->queryWithUserAndTags()->paginate($n);
}

public function getWithUserAndTagsForTagPaginate($tag, $n)
{
    return $this->queryWithUserAndTags()
        ->whereHas('tags', function($q) use ($tag)
    {
        $q->where('tags.tag_url', $tag);
    })->paginate($n);
}

public function store($inputs)
{
    return $this->post->create($inputs);
}

public function destroy($id)
{
    $post = $this->post->findOrFail($id);
    $post->tags()->detach();
    $post->delete();
}
}

```

 Vous êtes peut-être surpris par la longueur de certains des noms de fonctions. C'est un choix syntaxique.
Je préfère des noms explicites, quitte à les allonger.

Et voici le repository pour les tags (`app\Repositories\TagRepository.php`) :

```

<?php
namespace App\Repositories;

use App\Tag;
use Illuminate\Support\Str;

class TagRepository
{

protected $tag;

public function __construct(Tag $tag)
{
    $this->tag = $tag;
}

public function store($post, $tags)
{
    $tags = explode(',', $tags);

    foreach ($tags as $tag) {

```

php

```

        $tag = trim($tag);

        $tag_url = Str::slug($tag);

        $tag_ref = $this->tag->where('tag_url', $tag_url)->first();

        if(is_null($tag_ref))
        {
            $tag_ref = new $this->tag([
                'tag' => $tag,
                'tag_url' => $tag_url
            ]);

            $post->tags()->save($tag_ref);
        } else {

            $post->tags()->attach($tag_ref->id);
        }
    }

}

```

Fonctionnement

Nous allons à présent analyser ce code.

La liste des articles

La méthode du repository des articles est modifiée et renommée pour ajouter la table `tags` :

```

<?php
public function getWithUserAndTagsPaginate($n)
{
    return $this->queryWithUserAndTags()->paginate($n);
}

```

Pour clarifier le code j'ai créé une fonction privée qui va nous servir plusieurs fois :

```

<?php
private function queryWithUserAndTags()
{
    return $this->post->with('user', 'tags')
        ->orderBy('posts.created_at', 'desc');
}

```

Vous remarquez qu'on a ajouté la table `tags` comme paramètre de la méthode `with` en plus de `users`. On va en effet avoir besoin des informations des tags pour l'affichage dans la vue.

Il est intéressant de voir les requêtes générées par Eloquent, par exemple pour la première page :

```

sql
select count(*) as aggregate from `posts`

select * from `posts` order by `posts`.`created_at` desc limit 4 offset 0

select * from `users` where `users`.`id` in ('7', '2', '6')

select `tags`.* , `post_tag`.`post_id` as `pivot_post_id` , `post_tag`.`tag_id` as `pivot_tag_id` from `tags` inner

```

```
join `post_tag` on `tags`.`id` = `post_tag`.`tag_id` where `post_tag`.`post_id` in ('25', '1', '18', '14')
```

On voit que :

1. on demande le nombre total d'articles pour la pagination,
2. on demande les 4 premières lignes des articles avec l'ordre des dates,
3. on demande les utilisateurs qui correspondent aux articles sélectionnés,
4. on demande les tags concernés par les articles.

On se rend compte là du travail effectué par Eloquent pour nous !

Nouvel article

L'enregistrement d'un nouvel article va évidemment être un peu plus délicat à cause de la présence des tags.

Dans le repository des posts on va se contenter d'enregistrer l'article :

```
<?php
public function store($inputs)
{
    return $this->post->create($inputs);
}
```

C'est dans le repository des tags que le plus gros du travail va se faire :

```
<?php
public function store($post, $tags)
{
    $tags = explode(',', $tags);

    foreach ($tags as $tag) {

        $tag = trim($tag);

        $tag_url = Str::slug($tag);

        $tag_ref = $this->tag->where('tag_url', $tag_url)->first();

        if(is_null($tag_ref))
        {
            $tag_ref = new $this->tag([
                'tag' => $tag,
                'tag_url' => $tag_url
            ]);

            $post->tags()->save($tag_ref);
        } else {

            $post->tags()->attach($tag_ref->id);

        }
    }
}
```

Ce code mérite quelques commentaires. Les tags sont envoyés par le formulaire (que nous verrons plus loin) sous la forme de texte avec comme séparateur une virgule. Par exemple :

tag1,tag2,tag3

Dans le contrôleur la première chose est de vérifier qu'il y a des tags saisis :

```
<?php
if(isset($inputs['tags']))
{
    $tagRepository->store($post, $inputs['tags']);
}
```

php

Si c'est le cas on appelle la méthode `store` du repository en transmettant les tags et une référence du modèle créé. Dans le repository on crée un tableau en utilisant le séparateur (virgule) :

```
<?php
$tags = explode(',', $tags);
```

php

Ensuite on parcourt le tableau :

```
<?php
foreach ($tags as $tag)
```

php

Par précaution on supprime les espaces éventuels :

```
<?php
$tag = trim($tag);
```

php

On crée la version pour url du tag (avec la méthode `slug` de la classe `Str`) :

```
<?php
$tag_url = Str::slug($tag);
```

php

On regarde si ce tag existe déjà :

```
<?php
$tag_ref = $this->tag->where('tag_url', $tag_url)->first();
```

php

Si ce n'est pas le cas on le crée :

```
<?php
$tag_ref = new $this->tag([
    'tag' => $tag,
    'tag_url' => $tag_url
]);
$post->tags()->save($tag_ref);
```

php

Remarquez comment la méthode `save` ici permet à la fois de créer le tag et de référencer la table pivot.

Si le tag existe déjà on se contente d'informer la table pivot avec la méthode `attach` :

```
<?php
$post->tags()->attach($tag_ref->id);
```

php

Suppression d'un article

Quand on va supprimer un article il faudra aussi supprimer les liens avec les tags :

```
<?php
public function destroy($id)
{
    $post = $this->post->findOrFail($id);
    $post->tags()->detach();
    $post->delete();
```

php

}

La méthode `detach` permet de supprimer les lignes dans la table pivot.

La recherche par tag

Il nous reste enfin à voir la recherche par sélection d'un tag :

```
<?php
public function getWithUserAndTagsForTagPaginate($tag, $n)
{
    return $this->queryWithUserAndTags()
        ->whereHas('tags', function($q) use ($tag)
        {
            $q->where('tags.tag_url', $tag);
        })->paginate($n);
}
```

php

Vous remarquez que par rapport au code de la méthode `getWithUserAndTagsPaginate` on a ajouté la méthode `whereHas`. Cette méthode permet d'ajouter une condition sur une table chargée. Il est intéressant là aussi de voir les requêtes générées par Eloquent :

```
sql
select count(*) as aggregate from `posts` where (select count(*) from `tags` inner join `post_tag` on `tags`.`id` = `post_tag`.`tag_id` where `post_tag`.`post_id` = `posts`.`id` and `tags`.`tag_url` = 'tag-14') >= '1'

select * from `posts` where (select count(*) from `tags` inner join `post_tag` on `tags`.`id` = `post_tag`.`tag_id` where `post_tag`.`post_id` = `posts`.`id` and `tags`.`tag_url` = 'tag-14') >= '1' order by `posts`.`created_at` desc limit 4 offset 0

select * from `users` where `users`.`id` in ('3', '1', '6', '4')

select `tags`.* , `post_tag`.`post_id` as `pivot_post_id` , `post_tag`.`tag_id` as `pivot_tag_id` from `tags` inner join `post_tag` on `tags`.`id` = `post_tag`.`tag_id` where `post_tag`.`post_id` in ('13', '76', '87', '37')
```

Il y en a 5 plutôt chargées :

1. on compte les enregistrements pour la pagination (avec une jointure),
2. on récupère les 4 lignes des articles (avec une jointure),
3. on récupère les utilisateurs rédacteurs des articles,
4. on récupère les tags concernés par les articles (avec une jointure).

Il y a une chose que je n'ai pas géré dans tout ce code, c'est le cas des tags orphelins en cas de suppression d'un article. Cette gestion n'est pas obligatoire parce qu'il n'est pas vraiment gênant d'avoir des tags orphelins. On pourrait prévoir une maintenance épisodique de la base ou une action de l'administrateur.

Les vues

Le template

On conserve le même template (`resources/views/template.blade.php`) :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Mon joli site</title>
```

html

```

{!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css') !!}
{!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css') !!}
}
<!--[if lt IE 9]>
    {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.js') }}
    {{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js') }}
<![endif]-->
<style> textarea { resize: none; } </style>
</head>
<body>
<header class="jumbotron">
<div class="container">
    <h1 class="page-header">{!! link_to_route('post.index', 'Mon joli blog') !!}</h1>
    @yield('header')
</div>
</header>
<div class="container">
    @yield('contenu')
</div>
</body>
</html>

```

La liste

Voici la vue pour la liste des articles (`resources/views/posts/liste.blade.php`) :

```

html
@extends('template')

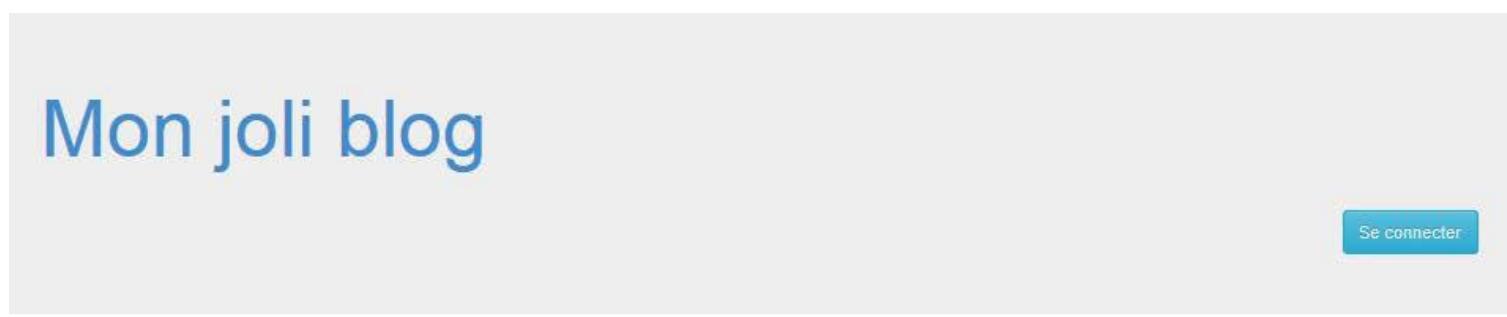
@section('header')
@if(Auth::check())
    <div class="btn-group pull-right">
        {!! link_to_route('post.create', 'Créer un article', [], ['class' => 'btn btn-info']) !!}
        {!! link_to('logout', 'Deconnexion', ['class' => 'btn btn-warning']) !!}
    </div>
@else
    {!! link_to('login', 'Se connecter', ['class' => 'btn btn-info pull-right']) !!}
@endif
@endsection

@section('contenu')
@if(isset($info))
    <div class="row alert alert-info">{!! $info !!}</div>
@endif
{!! $links !!}
@foreach($posts as $post)
    <article class="row bg-primary">
        <div class="col-md-12">
            <header>
                <h1>{!! $post->titre !!}</h1>
                <div class="pull-right">
                    @foreach($post->tags as $tag)
                        {!! link_to('post/tag/' . $tag->tag_url, $tag->tag, ['class' => 'btn btn-xs btn-info']) !!}
                    @endforeach
                </div>
            </header>
            <hr>
            <section>
                <p>{!! $post->contenu !!}</p>
                @if(Auth::check() and Auth::user()->admin)
                    {!! Form::open(['method' => 'DELETE', 'route' => ['post.destroy', $post->id]]) !!}
                    {!! Form::submit('Supprimer cet article', ['class' =>

```

```
'btn btn-danger btn-xs ', 'onclick' => 'return confirm(\'Vraiment supprimer cet article ?\')]) !!}
        {!! Form::close() !!}
    @endif
    <em class="pull-right">
        <span class="glyphicon glyphicon-pencil"></span> {{ $post->user->name }} le {!! $post->created_at->format('d-m-Y') !!}
    </em>
</section>
</div>
</article>
<br>
@endforeach
{!! $links !!}
@endsection
```

Avec cet aspect :



Titre62

Contenu62 Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Nom8 le 28-12-2015

Titre75

Contenu75 Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Nom7 le 26-12-2015

La liste des articles

Les tags apparaissent sous la forme de petits boutons. Le fait de cliquer sur un de ces boutons lance la recherche à partir de ce tag et affiche les articles correspondant ainsi qu'une barre d'information :

Mon joli blog

[Se connecter](#)

Résultats pour la recherche du mot-clé : tag10

« [1](#) [2](#) [3](#) »

Titre79

Contenu79 Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Nom5 le 25-12-2015

Titre30

Contenu30 Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Nom0 le 16-11-2015

La recherche par tag

Un utilisateur connecté dispose en plus du bouton pour créer un article. L'administrateur a en plus le bouton de suppression :

Mon joli blog

[Créer un article](#) [Déconnexion](#)

« [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) ... [24](#) [25](#) »

Titre79

[tag9](#) [tag10](#) [tag12](#) [tag14](#)

Contenu79 Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

[Supprimer cet article](#)

Nom5 le 25-12-2015

Les boutons pour les utilisateurs connectés

La vue de création d'un article

Le formulaire de création d'un article (`resources/views/posts/add.blade.php`) a été enrichi d'un contrôle de texte pour la saisie des tags :

```
@extends('template')

@section('contenu')
    <br>
    <div class="col-sm-offset-3 col-sm-6">
        <div class="panel panel-info">
            <div class="panel-heading">Ajout d'un article</div>
            <div class="panel-body">
                {!! Form::open(['route' => 'post.store']) !!}
                <div class="form-group {!! $errors->has('titre') ? 'has-error' : '' !!}>
                    {!! Form::text('titre', null, ['class' => 'form-control',
'placeholder' => 'Titre']) !!}
                    {!! $errors->first('titre', '<small class="help-block">:message</small>') !!}
                </div>
                <div class="form-group {!! $errors->has('contenu') ? 'has-error' : '' !!}>
                    {!! Form::textarea ('contenu', null, ['class' => 'form-control',
'placeholder' => 'Contenu']) !!}
                    {!! $errors->first('contenu', '<small class="help-block">:message</small>') !!}
                </div>
                <div class="form-group {{ $errors->has('tags') ? 'has-error' : '' }}>
                    {!! Form::text('tags', null, array('class' => 'form-control',
'placeholder' => 'Entrez les tags séparés par des virgules')) !!}
                    {!! $errors->first('tags', '<small class="help-block">:message</small>') !!}
                </div>
                {!! Form::submit('Envoyer !', ['class' => 'btn btn-info pull-right']) !!}
                {!! Form::close() !!}
            </div>
        </div>
        <a href="javascript:history.back()" class="btn btn-primary">
            <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
        </a>
    </div>
@endsection
```

Est aussi géré le message d'erreur pour la validation des tags :

Ajout d'un article

Les belles heures

Les belles heures sont une exploration contemplative...

ddddd

Les mots-clés, séparés par des virgules (sans espaces), doivent avoir au maximum 50 caractères alphanumériques.

Envoyer !

Le formulaire de création d'un article

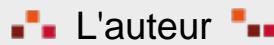
Je ne détaille pas le code de toutes ces vues, il n'est pas bien compliqué et recouvre des situations déjà rencontrées.

En résumé

- Une relation de type n:n nécessite la création d'une table pivot.
 - Eloquent gère élégamment les tables pivots avec des méthodes adaptées.
 - On peut créer des règles et des messages de validation personnalisés.

La relation 1:n

Les commandes et les assistants



Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

[S'inscrire](#)[Se connecter](#)

Accueil ▶ Cours ▶ Découvrez le framework PHP Laravel ▶ Les commandes et les assistants

Découvrez le framework PHP Laravel

15 heures Moyenne



Les commandes et les assistants

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Dans les chapitres précédents on a plusieurs fois dû créer des migrations, des populations, des contrôleurs, des modèles... Artisan possède des commandes pour effectuer certaines de ces opérations mais il ne va pas bien loin et il nous a fallu créer pas mal de code qui, de toute évidence, pourrait plus ou moins facilement être automatisé.

Il est possible d'améliorer les commandes d'artisan ou même de s'en créer des nouvelles. Il existe aussi des assistants pour nous aider dans ces tâches un peu pénibles ou répétitives.

Améliorer une commande

Il existe dans Artisan une commande pour générer un modèle :

```
php artisan make:model MonModele
```

text

Voici ce qu'on obtient :

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class MonModele extends Model
{
    //
}
```

php

Ce n'est pas grand chose mais c'est déjà ça, une coquille un peu vide.



Cette commande permet toutefois de créer une migration avec l'option **--migration**.

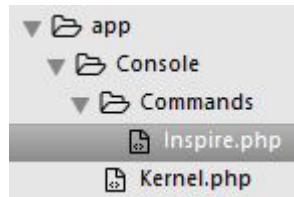
Si on a prévu de placer les modèles dans un dossier spécifique il suffit de renseigner l'espace de nom lors de la commande :

```
php artisan make:model App\Models\MonModele
```

php

On a vu qu'il était pratique d'avoir une propriété `$fillable` pour répertorier les colonnes qu'on peut mettre à jour sans risque avec un assignement de masse. Ce qui serait bien serait d'avoir une option pour ajouter cette propriété.

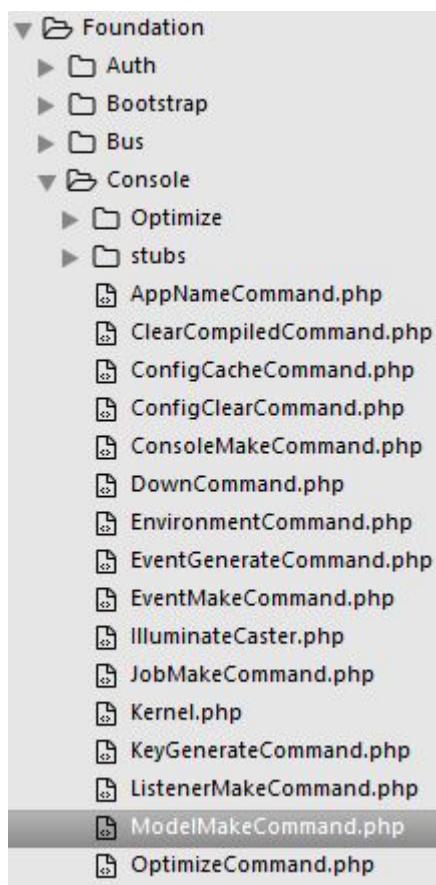
Si vous regardez dans le dossier des commandes "console" on voit qu'il en existe déjà une :



Les commandes "console"

Il s'agit d'un exemple qui peut vous guider pour réaliser une commande.

Puisque la commande existe déjà dans Artisan on ne va pas réinventer la roue et nous contenter d'étendre les possibilités de celle-ci. Il faut un peu fouiller dans le framework pour trouver cette commande :



La commande ModelMakeCommand de Laravel

Avec ce code :

```
<?php
```

php

```
namespace Illuminate\Foundation\Console;

use Illuminate\Console\GeneratorCommand;
use Symfony\Component\Console\Input\InputOption;

class ModelMakeCommand extends GeneratorCommand
{
    /**
     * The console command name.
     *
     * @var string
     */
    protected $name = 'make:model';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Create a new Eloquent model class';

    /**
     * The type of class being generated.
     *
     * @var string
     */
    protected $type = 'Model';

    /**
     * Execute the console command.
     *
     * @return void
     */
    public function fire()
    {
        if (parent::fire() !== false) {
            if ($this->option('migration')) {
                $table = str_plural(snake_case(class_basename($this->argument('name'))));

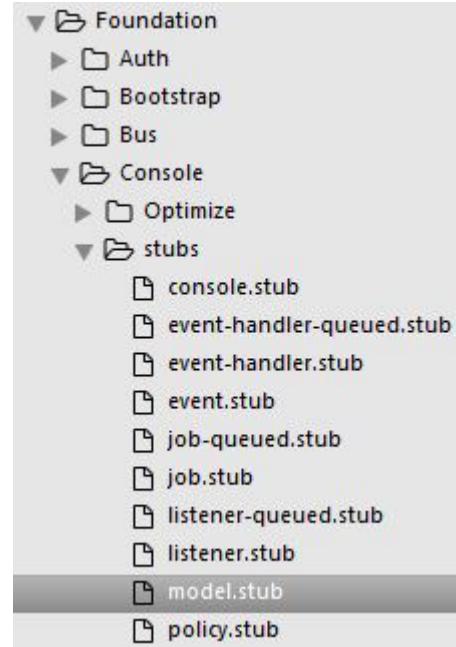
                $this->call('make:migration', ['name' => "create_{$table}_table", '--create' => $table]);
            }
        }
    }

    /**
     * Get the stub file for the generator.
     *
     * @return string
     */
    protected function getStub()
    {
        return __DIR__.'/stubs/model.stub';
    }

    /**
     * Get the default namespace for the class.
     *
     * @param string $rootNamespace
     * @return string
     */
    protected function getDefaultNamespace($rootNamespace)
    {
        return $rootNamespace;
    }
}
```

```
/**
 * Get the console command options.
 *
 * @return array
 */
protected function getOptions()
{
    return [
        ['migration', 'm', InputOption::VALUE_NONE, 'Create a new migration file for the model.'],
    ];
}
```

La commande utilise un gabarit (stub) qu'on trouve ici :



Le stub de la commande

Avec ce code :

```
<?php
namespace DummyNamespace;

use Illuminate\Database\Eloquent\Model;

class DummyClass extends Model
{
    //
```

php

Les "Dummy" permettent d'avoir des emplacements aux données variables, ici le nom de l'espace de nom et celui de la classe.



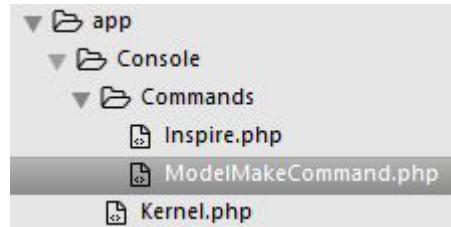
Dans la version 5.0 on avait une autre syntaxe constituée d'accolades comme pour Blade.

On va commencer par créer notre classe en étendant celle de Laravel. Alors créons la nouvelle commande avec artisan :

text

```
php artisan make:console ModelMakeCommand
```

On la retrouve ici :



Notre commande

Avec ce code généré :

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class ModelMakeCommand extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'command:name';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Command description.';

    /**
     * Create a new command instance.
     *
     * @return void
     */
    public function __construct()
    {
        parent::__construct();
    }

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        //
    }
}
```

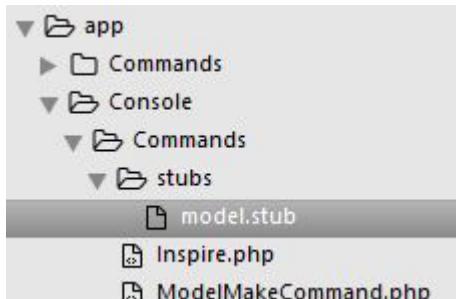
On l'enregistre dans `app\Console\Kernel.php` :

```
<?php
protected $commands = [
```

php

```
Commands\Inspire::class,
Commands\ModelMakeCommand::class,
];
```

On va aussi créer notre stub :



Notre stub

Avec ce code pour ajouter la propriété `$fillable` :

```
<?php
namespace DummyNamespace;

use Illuminate\Database\Eloquent\Model;

class DummyClass extends Model {
    /**
     * Attributes that should be mass-assignable.
     *
     * @var array
     */
    protected $fillable = [DummyFillable];
}
```

php

Il ne nous reste plus qu'à modifier le code généré automatiquement pour d'une part étendre la classe de Laravel, et d'autre part assurer le fonctionnement de l'option `fillable` :

```
<?php
namespace App\Console\Commands;

use Illuminate\Foundation\Console\ModelMakeCommand as BaseModelCommand;
use Symfony\Component\Console\Input\InputOption;
use Illuminate\Support\Facades\Schema;

class ModelMakeCommand extends BaseModelCommand
{
    protected $exclude = ['id', 'password', 'created_at', 'updated_at'];

    protected function getStub()
    {
        return __DIR__.'/stubs/model.stub';
    }

    protected function buildClass($name)
    {
        $stub = $this->files->get($this->getStub());
```

php

```

        return $this->replaceNamespace($stub, $name)
        ->replaceFillable($stub)
        ->replaceClass($stub, $name);
    }

protected function replaceFillable(&$stub)
{
    if($this->input->getOption('fillable'))
    {
        // On construit le nom de la table à partir du nom du modèle
        $table = str_plural(strtolower($this->getNameInput()));
        // On récupère le nom des colonnes
        $columns = Schema::getColumnListing($table);
        // On exclut les colonnes non désirées
        $columns = array_filter($columns, function($value)
        {
            return !in_array($value, $this->exclude);
        });
        // On ajoute des apostrophes
        array_walk($columns, function(&$value) {
            $value = "'" . $value . "'";
        });
        // CSV format
        $columns = implode(',', $columns);
    }

    $stub = str_replace('DummyFillable', isset($columns)? $columns : '', $stub);

    return $this;
}

protected function getOptions()
{
    return [
        ['migration', 'm', InputOption::VALUE_NONE, 'Create a new migration file for the model.'],
        ['fillable', null, InputOption::VALUE_NONE, 'Set the fillable columns.', null]
    ];
}
}

```

Je ne vais pas détailler le code de cette commande, je vous laisse l'analyser si vous désirez en comprendre le fonctionnement. Le but est juste de montrer qu'il est possible de le réaliser avec un exemple concret.

On va tester cette commande avec la table `posts` qui nous a servi pour le petit blog :

```
php artisan make:model Post --fillable
```

php

On obtient ce modèle :

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model  {

    /**
     * Attributes that should be mass-assignable.
     *
     * @var array
     */
    protected $fillable = ['titre', 'contenu', 'user_id'];
}
```

php

{}

On a bien eu la création de la propriété et le remplissage du tableau avec exclusion des colonnes prévues.

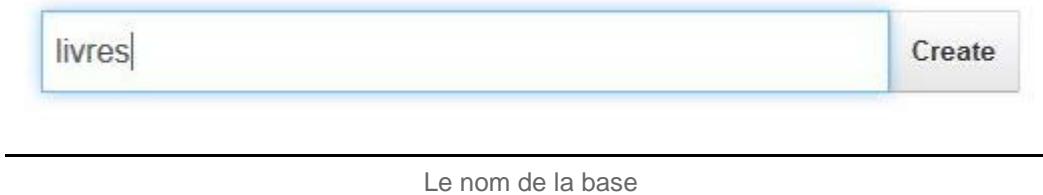
Laravel Schema Designer

Le designer de schéma [que vous pouvez trouver ici](#) est une démarche visuelle et efficace. Il suffit de dessiner les tables et de renseigner les champs et relations et ensuite sont générés pour nous :

- migrations,
- populations,
- modèles.

Prenons un exemple avec des livres et des auteurs et une relation 1:n entre elles.

La première chose consiste à créer un compte, ce qui est gratuit. Ensuite vous donnez un nom à votre base :



Le nom de la base

Création des tables

Ensuite vous cliquez sur le bouton pour ajouter une table :



Créer une table

Ensuite vous renseignez les champs :

Table name: livres

Model class name: Livre

Model namespace: Model namespace (optional)

Color: Blue

Add Laravel ID increments column.

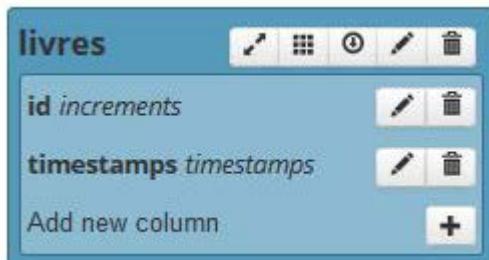
Add Laravel timestamps.

Add Laravel soft delete.

Add Close

Le formulaire de création d'une table

Vous avez alors votre table visuellement à l'écran :



La table des livres

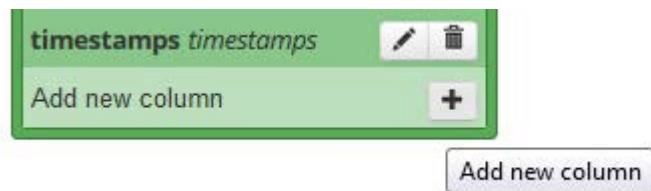
Créez de la même manière la table des auteurs (vous pouvez jouer avec les couleurs pour bien distinguer les tables, ce qui s'avère utile dès qu'il y en a beaucoup) :



La table des auteurs

Création des champs

On va ensuite renseigner les champs pour les tables. Commençons par les auteurs. Il suffit de cliquer sur le bouton comportant le signe "+" :



Le formulaire est très complet, il suffit de le renseigner :

The dialog box has a title 'Add column to table auteurs'. It contains the following fields:

- Name: nom
- Type: STRING (VARCHAR)
- Length: 100
- Default value: Default
- Enum value: enum1, enum2

Below these are several checkboxes with the following options:

- Auto Incremental Unsigned Integer
- Primary Key Index
- Nullable Unique Index
- Fillable Guarded
- Visible Hidden
- Foreign key

At the bottom right are three buttons: 'Add' (blue), 'Add Another' (white), and 'Close' (white).

Le formulaire de création d'un champ

Après validation on voit notre nouveau champ dans la table :

The table has a green header bar with the word 'auteurs' repeated twice. Below it is a toolbar with icons for edit, delete, and a plus sign. A button labeled 'Add new column' is visible. In the main area, there are three columns listed: 'id increments', 'timestamps timestamps', and 'nom string (100)'. The 'nom' column is highlighted with a yellow background. Below the table, there's another 'Add new column' button.

Le champ ajouté

Deux boutons sont présents pour le modifier ou le supprimer. On va de la même manière renseigner la table des livres avec le titre et la clé étrangère :

Edit column: auteur_id

| | | | |
|---|--|--|--|
| Name: | auteur_id | | |
| Type: | INTEGER | | |
| Length: | 10 | | |
| Default value: | Default | | |
| Enum value: | enum1, enum2 | | |
| <input type="checkbox"/> Auto Incremental <input checked="" type="checkbox"/> Unsigned Integer
<input type="checkbox"/> Primary key <input type="checkbox"/> Index
<input type="checkbox"/> Nullable <input type="checkbox"/> Unique Index
<input type="checkbox"/> Fillable <input type="checkbox"/> Guarded
<input type="checkbox"/> Visible <input type="checkbox"/> Hidden
<input checked="" type="checkbox"/> Foreign key | | | |
| References | On (table) | onUpdate | onDelete |
| id
<input type="button" value="▼"/> | livres
<input type="button" value="▼"/> | restrict
<input type="button" value="▼"/> | restrict
<input type="button" value="▼"/> |
| <input type="button" value="Save"/> <input type="button" value="Close"/> | | | |

La clé étrangère

On se retrouve donc avec nos deux tables et leurs champs renseignés :

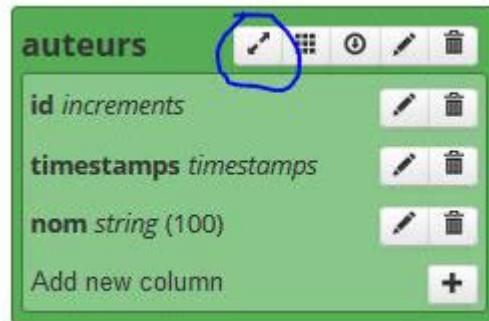
| auteurs | |
|-------------------|----------------------------------|
| <i>id</i> | increments |
| <i>timestamps</i> | <i>timestamps</i> |
| nom | string (100) |
| Add new column | <input type="button" value="+"/> |

| livres | |
|-------------------|----------------------------------|
| <i>id</i> | increments |
| <i>timestamps</i> | <i>timestamps</i> |
| titre | string (100) |
| auteur_id | FK integer |
| Add new column | <input type="button" value="+"/> |

Les deux tables créées avec leurs champs renseignés

Création des relations

Il ne nous reste plus qu'à définir les relations. Commençons par les auteurs. Il faut cliquer sur ce bouton :



Le bouton de création d'une relation

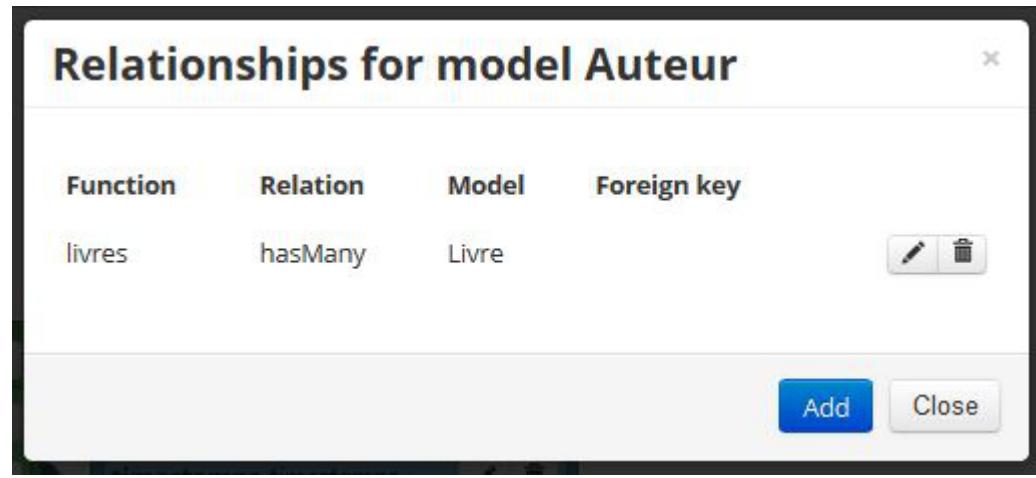
On a alors ce formulaire à compléter :

Add relationship to model Auteur

| | |
|---|-------------------------|
| Function name: | livres |
| Relation: | hasMany |
| <input type="checkbox"/> Use namespaces | |
| Related model: | Livre |
| Foreign keys: | Use a comma if multiple |
| Extra methods: | ex. ->withPivot('foo') |
| <pre>public function livres(){ \$this->hasMany('Livre'); }</pre> | |
| <p style="text-align: right;">Add Back Close</p> | |

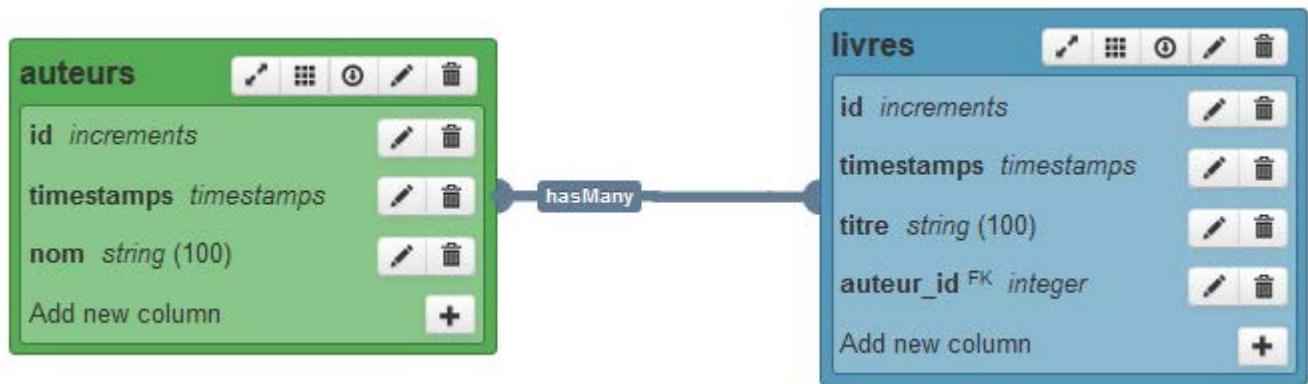
Le formulaire de création de la relation

Avec ensuite la liste des relations existantes :



Les relations existantes

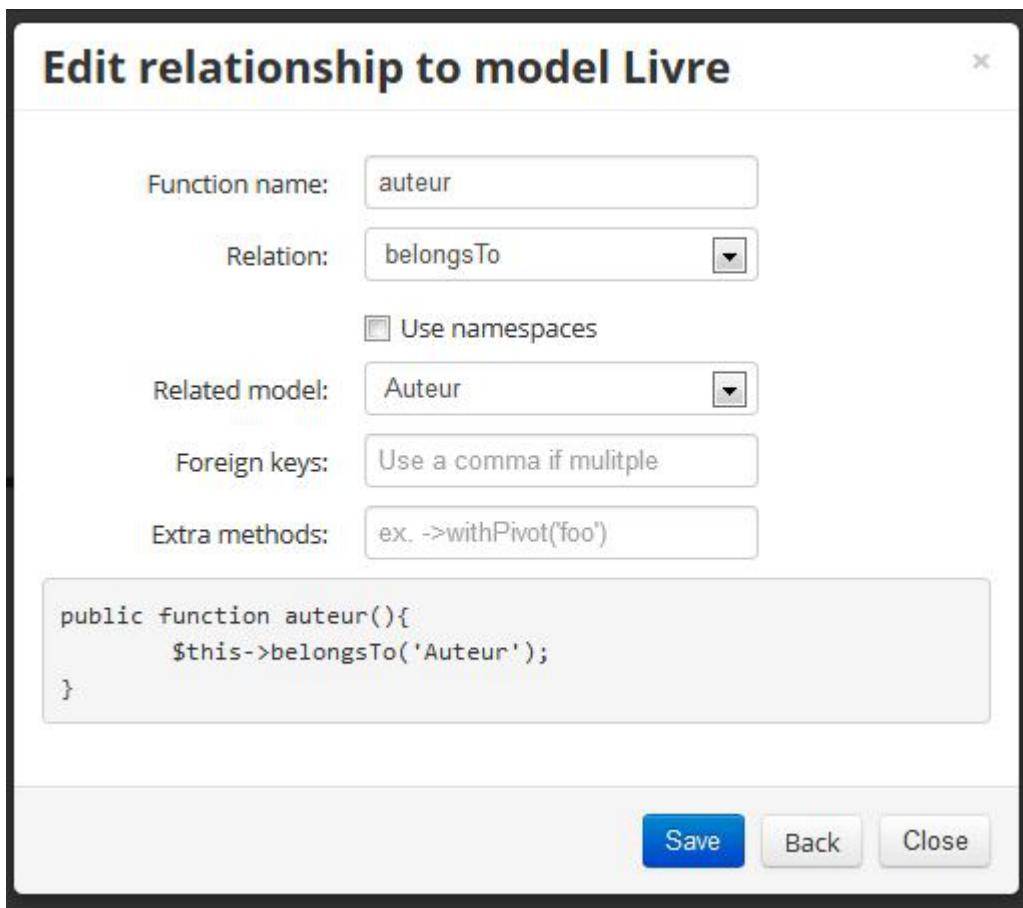
Evidemment nous n'en avons qu'une. La relation apparaît visuellement sous la forme d'une liaison avec apparition du nom de la méthode au survol de la souris :



La relation représentée visuellement

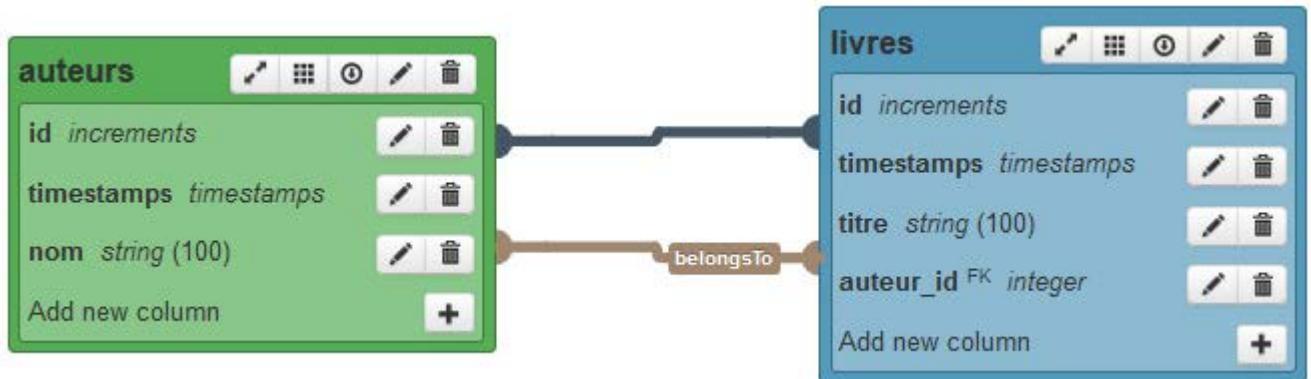
Pratique non ? :)

De la même manière on crée la relation vue du côté des livres :



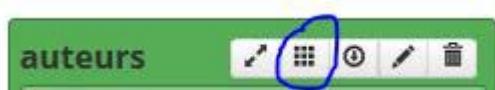
La relation vue du côté des livres

On a également une représentation visuelle avec le nom de la méthode qui apparaît au survol :



La représentation visuelle de la méthode belongsTo

Pour ajouter la population il faut cliquer sur ce bouton :



Le bouton pour la population

On a alors un formulaire à compléter :

Seed name: AuteurTableSeeder
nom: 'Dupont'

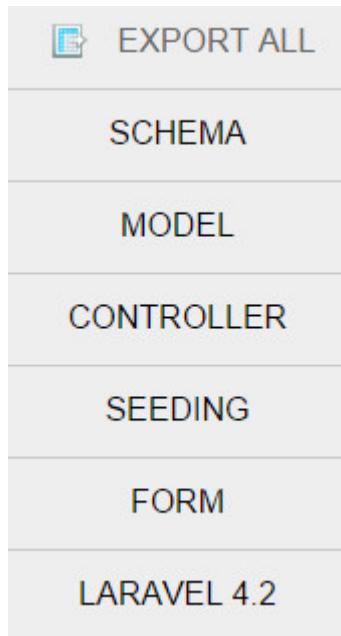
NOTE: Remember to use quotation marks for strings e.g. 'user1' or Hash::make('test').

Save Back

Le formulaire pour la population

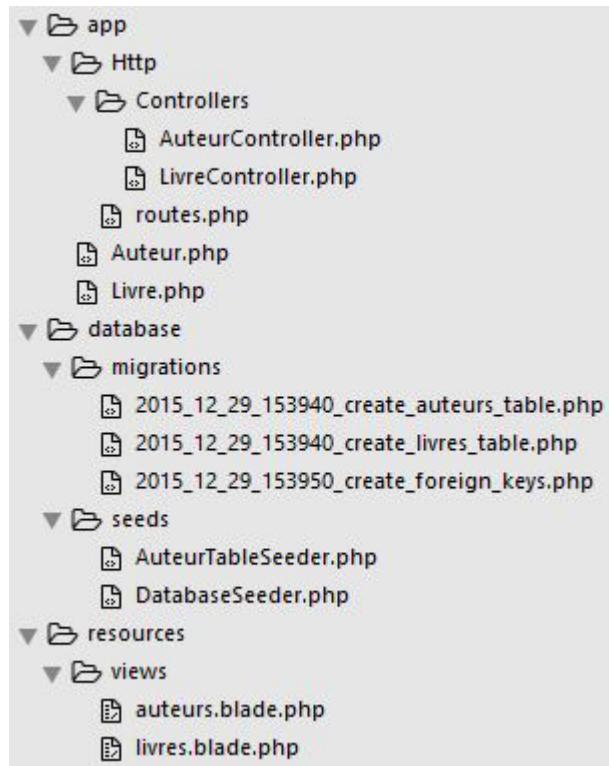
Exportation des fichiers

Maintenant notre schéma est terminé, il ne reste plus qu'à exporter les fichiers :



Le menu de l'exportation

On va tout demander en cliquant directement sur "EXPORT ALL". On reçoit un fichier compressé avec tout ça :



Les fichiers exportés

On a donc les migrations, par exemple celle pour les auteurs :

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateAuteursTable extends Migration {

    public function up()
    {
        Schema::create('auteurs', function(Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
            $table->string('nom', 100)->unique();
        });
    }

    public function down()
    {
        Schema::drop('auteurs');
    }
}
```

Une migration spécifique pour les clés étrangères :

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateForeignKeys extends Migration {

    public function up()
    {
        Schema::table('livres', function(Blueprint $table) {
```

```

        $table->foreign('auteur_id')->references('id')->on('livres')
            ->onDelete('restrict')
            ->onUpdate('restrict');
    });

}

public function down()
{
    Schema::table('livres', function(Blueprint $table) {
        $table->dropForeign('livres_auteur_id_foreign');
    });
}
}

```

La population qu'on a prévue pour un auteur :

```

<?php
php

class AuteurTableSeeder extends Seeder {

    public function run()
    {
        //DB::table('auteurs')->delete();

        // AuteurTableSeeder
        Auteur::create(array(
            'nom' => 'Dupont'
        ));
    }
}

```

Avec le DatabaseSeeder à jour :

```

<?php
php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder {

    public function run()
    {
        Model::unguard();

        $this->call('AuteurTableSeeder');
        $this->command->info('Auteur table seeded!');
    }
}

```

Les modèles, par exemple pour les auteurs :

```

<?php
php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Auteur extends Model {

    protected $table = 'auteurs';
    public $timestamps = true;
    protected $fillable = array('nom');

    public function livres()
}

```

```
{
    return $this->hasMany('Livre');
}

}
```

On a même des vues avec un squelette de formulaire, par exemple pour les auteurs :

```
{!! Form::open(array('route' => 'route.name', 'method' => 'POST')) !!}
<ul>
    <li>
        {!! Form::label('nom', 'Nom:') !!}
        {!! Form::text('nom') !!}
    </li>
    <li>
        {!! Form::submit() !!}
    </li>
</ul>
{!! Form::close() !!}
```

html

Ce générateur crée aussi des contrôleurs et est vraiment très convivial avec son interface graphique.

En résumé

- Artisan propose un grand arsenal de commandes pour générer du code. Il est possible de créer de nouvelles commandes ou d'étendre les possibilités des commandes existantes.
- Le designer de schéma permet, à partir d'une interface visuelle conviviale, de générer des migrations, des modèles, des populations, des contrôleurs, des vues.



La relation n:n



Query Builder

L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

OpenClassrooms



Qui sommes-nous ?

Fonctionnement de nos cours

Recrutement

Nous contacter

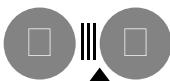
[S'inscrire](#)

Se connecter

[Accueil](#) ▶ [Cours](#) ▶ [Découvrez le framework PHP Laravel](#) ▶ [Query Builder](#)

Découvrez le framework PHP Laravel

15 heures Moyenne



Query Builder

[Connectez-vous](#) ou [inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Nous avons jusqu'à présent utilisé Eloquent pour générer nos requêtes. Malgré la pertinence de celui-ci, il est parfois nécessaire de générer des requêtes qui dépassent ses compétences. C'est alors qu'intervient le Query builder, un outil pratique de génération de requêtes avec une syntaxe explicite qui est le parfait compagnon d'Eloquent.

Les données

Migration

Pour effectuer des tests, nous aurons besoin de données. Créez une nouvelle migration :

```
php artisan make:migration create_livres_table
```

Et entrez ce code :

```
<?php  
  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Database\Migrations\Migration;  
  
class CreateLivresTable extends Migration {  
  
    /**  
     * Run the migrations.  
     *  
     * @return void  
     */  
    public function up()  
    {  
        Schema::create('auteurs', function(Blueprint $table) {  
            $table->increments('id');  
            $table->timestamps();  
        });  
    }  
  
    public function down()  
    {  
        Schema::dropIfExists('auteurs');  
    }  
}
```

php

```

        $table->string('nom', 100)->unique();
    });
Schema::create('editeurs', function(Blueprint $table) {
    $table->increments('id');
    $table->timestamps();
    $table->string('nom', 100)->unique();
});
Schema::create('livres', function(Blueprint $table) {
    $table->increments('id');
    $table->timestamps();
    $table->string('titre', 100);
    $table->integer('editeur_id')->unsigned();
    $table->text('description');
});
Schema::create('auteur_livre', function(Blueprint $table) {
    $table->increments('id');
    $table->integer('auteur_id')->unsigned();
    $table->integer('livre_id')->unsigned();
});
Schema::table('livres', function(Blueprint $table) {
    $table->foreign('editeur_id')->references('id')->on('editeurs')
        ->onDelete('restrict')
        ->onUpdate('restrict');
});
Schema::table('auteur_livre', function(Blueprint $table) {
    $table->foreign('auteur_id')->references('id')->on('auteurs')
        ->onDelete('restrict')
        ->onUpdate('restrict');
});
Schema::table('auteur_livre', function(Blueprint $table) {
    $table->foreign('livre_id')->references('id')->on('livres')
        ->onDelete('restrict')
        ->onUpdate('restrict');
});
}

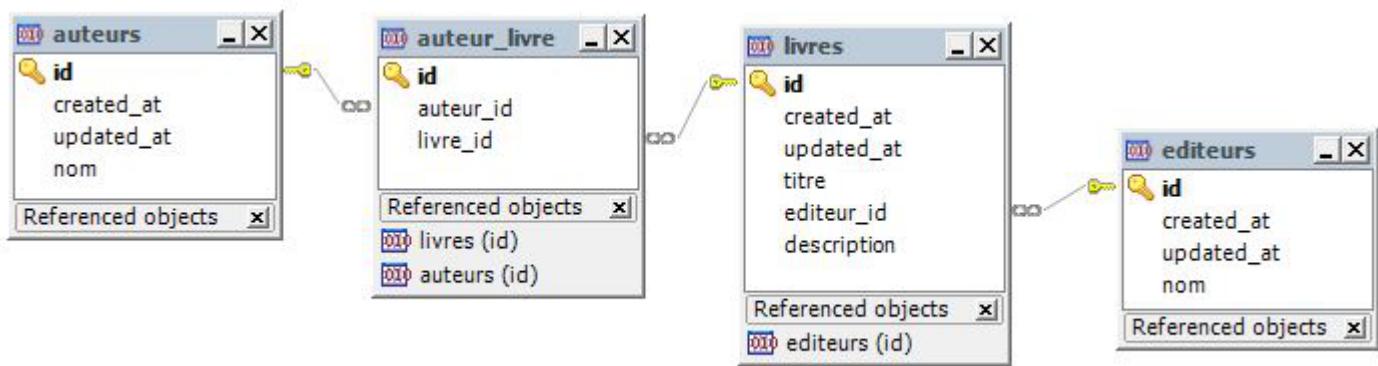
/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
    Schema::table('livres', function(Blueprint $table) {
        $table->dropForeign('livres_editeur_id_foreign');
    });
    Schema::table('auteur_livre', function(Blueprint $table) {
        $table->dropForeign('auteur_livre_auteur_id_foreign');
    });
    Schema::table('auteur_livre', function(Blueprint $table) {
        $table->dropForeign('auteur_livre_livre_id_foreign');
    });
    Schema::drop('auteur_livre');
    Schema::drop('livres');
    Schema::drop('auteurs');
    Schema::drop('editeurs');
}
}

```

Et lancez la migration :

```
php artisan migrate
```

Le résultat est la création de ces 4 tables avec ces relations :



Les 4 tables et leurs relations

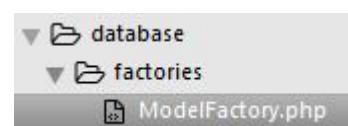
On a comme relations :

- **1:n** entre editeurs et livres,
- **n:n** entre auteurs et livres.

Population

Pour la population nous allons utiliser la librairie [fzaninotto/Faker](#) qui est chargée par défaut par Laravel. Cette librairie permet de générer des noms, adresses, textes, nombres... avec une grande facilité. C'est très pratique par exemple lorsqu'on effectue des tests.

Laravel intègre cette librairie et permet de créer des "Model Factories". Regardez ce fichier :



Le ModelFactory

Avec ce code :

```

<?php
/*
|-----|
| Model Factories
|-----|
|
| Here you may define all of your model factories. Model factories give
| you a convenient way to create models for testing and seeding your
| database. Just tell the factory how a default model should look.
|
*/
$factory->define(App\User::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => bcrypt(str_random(10)),
        'remember_token' => str_random(10),
    ];
});
  
```

php

```
];
});
```

Vous avez un exemple avec un factory pour le modèle User. On va donc créer nos factories pour nos modèles :

```
<?php
$factory->define(App\Editeur::class, function (Faker\Generator $faker) {
    return [
        'nom' => $faker->name,
    ];
});

$factory->define(App\Auteur::class, function (Faker\Generator $faker) {
    return [
        'nom' => $faker->name,
    ];
});

$factory->define(App\Livre::class, function (Faker\Generator $faker) {
    return [
        'titre' => $faker->sentence(3),
        'description' => $faker->text,
        'editeur_id' => $faker->numberBetween(1, 40),
    ];
});
```

php

Ensuite on prévoit ce code dans `DatabaseSeeder` :

```
<?php

use Illuminate\Database\Seeder;
use Faker\Factory;

class DatabaseSeeder extends Seeder {

    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        factory(App\Editeur::class, 40)->create();
        factory(App\Auteur::class, 40)->create();
        factory(App\Livre::class, 80)->create();

        for ($i = 1; $i < 41; $i++) {
            $number = rand(2, 8);
            for ($j = 1; $j <= $number; $j++) {
                DB::table('auteur_livre')->insert([
                    'livre_id' => rand(1, 40),
                    'auteur_id' => $i
                ]);
            }
        }
    }
}
```

php

Lancez ensuite la population :

```
php artisan db:seed
```

On aura ainsi 40 éditeurs et 40 auteurs avec des noms aléatoires. On aura aussi 80 livres affectés à des éditeurs et aussi des relations entre les livres et les auteurs. Donc de quoi effectuer tranquillement des requêtes.

Il nous faut aussi les modèles pour faire marcher tout ça si on veut utiliser Eloquent. Pour les éditeurs :

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Editeur extends Model
{
    public function livres()
    {
        return $this->hasMany('App\Livre');
    }
}
```

php

Pour les auteurs :

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Auteur extends Model
{
    public function livres()
    {
        return $this->belongsToMany('App\Livre');
    }
}
```

php

Pour les livres :

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Livre extends Model
{
    public function auteurs()
    {
        return $this->belongsToMany('App\Auteur');
    }
    public function editeur()
    {
        return $this->belongsTo('App\Editeur');
    }
}
```

php

Les sélections

Liste des éditeurs

Commençons par des choses simples, on veut tous les éditeurs. Avec Eloquent c'est facile :

```
<?php
$editeurs = App\Editeur::all();
foreach ($editeurs as $editeur) {
    echo $editeur->nom, '<br>';
}
```

php

Vous obtenez ainsi la liste complète des éditeurs. Avec le Query Builder la syntaxe est la suivante :

```
<?php
$editeurs = DB::table('editeurs')->get();
foreach ($editeurs as $editeur) {
    echo $editeur->nom, '<br>';
}
```

php

Il faut bien désigner la table et utiliser ensuite la méthode `get()`. Le résultat est le même.

Tableau des valeurs d'une colonne

Pour obtenir un tableau des valeurs d'une colonne on utilise la méthode `lists()` :

```
<?php
$editeurs = DB::table('editeurs')->lists('nom');
foreach ($editeurs as $editeur) {
    echo $editeur, '<br>';
}
```

php

Ce qui fonctionne aussi avec Eloquent :

```
<?php
$editeurs = App\Editeur::lists('nom');
foreach ($editeurs as $editeur) {
    echo $editeur, '<br>';
}
```

php

Ce qui correspond à cette requête SQL :

```
select nom from editeurs
```

sql

Ligne particulière

Si nous voulons trouver une ligne particulière, avec Eloquent :

```
<?php
$editeur = App\Editeur::find(10);
echo $editeur->nom;
```

php

Nous aurons le nom de l'éditeur qui a l'id 10. Avec Query Builder il faut écrire :

```
<?php
$editeur = DB::table('editeurs')->whereId(10)->first();
echo $editeur->nom;
```

php

On précise la table, on précise l'id avec la méthode `where` et on prend le premier enregistrement (`first()`).

En SQL pur on écrirait :

```
sql
SELECT * FROM editeurs WHERE id = 10
```

Colonne isolée

On peut sélectionner juste une colonne avec la méthode `pluck` :

```
php
<?php
$editeur = DB::table('editeurs')->whereId(10)->pluck('nom');
dd($editeur);
```

Ce qui en SQL s'écrit :

```
sql
select nom from editeurs where id = 10
```

On peut aussi faire un `select` pour sélectionner des colonnes :

```
php
<?php
$editeurs = App\Editeur::select('nom')->get();
foreach ($editeurs as $editeur) {
    echo $editeur->nom, '<br>';
}
```

Ou simplement avec le Query Builder :

```
php
<?php
$editeurs = DB::table('editeurs')->select('nom')->get();
foreach ($editeurs as $editeur) {
    echo $editeur->nom, '<br>';
}
```

Le requête est encore celle-ci :

```
sql
select nom from editeurs
```

Lignes distinctes

On peut aussi utiliser la méthode `distinct` pour avoir des lignes distinctes :

```
php
<?php
$livres = App\Livre::select('editeur_id')->distinct()->get();
foreach ($livres as $livre) {
    echo $livre->editeur_id, '<br>';
}
```

Et en version Query Builder :

```
php
<?php
$livres = DB::table('livres')->select('editeur_id')->distinct()->get();
foreach ($livres as $livre) {
    echo $livre->editeur_id, '<br>';
}
```

Avec cette requête SQL :

```
sql
select distinct editeur_id from livres
```

Plusieurs conditions

On a vu qu'on peut utiliser la méthode `where`, on peut lui adjoindre la méthode `orWhere` :

```
<?php
$livres = App\Livre::where('titre', '<', 'c')
->orWhere('titre', '>', 'v')
->get();
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

php

En version Query Buider :

```
<?php
$livres = DB::table('livres')
->where('titre', '<', 'c')
->orWhere('titre', '>', 'v')
->get();
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

php

Avec cette requête SQL :

```
select * from livres where titre < 'c' or titre > 'v'
```

sql

Encadrer des valeurs

On peut encadrer des valeurs avec `whereBetween` :

```
<?php
$livres = App\Livre::whereBetween('titre', array('g', 'k'))->get();
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

php

En version Query Builder :

```
<?php
$livres = DB::table('livres')->whereBetween('titre', array('g', 'k'))->get();
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

php

Avec cette requête SQL :

```
select * from livres where titre between 'g' and 'k'
```

sql

On peut faire l'inverse avec `whereNotBetween` :

```
<?php
$livres = App\Livre::whereNotBetween ('titre', array('g', 'k'))->get();
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

php

En version Query Builder :

```
<?php
$livres = DB::table('livres')->whereNotBetween ('titre', array('g', 'k'))->get();
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

php

Avec cette requête SQL :

```
sql
select * from livres where titre not between 'g' and 'k'
```

Prendre des valeurs dans un tableau

On peut aussi prendre des valeurs dans un tableau avec `whereIn` :

```
php
<?php
$livres = App\Livre::whereIn('editeur_id', [10, 11, 12])->get();
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

En version Query Builder :

```
php
<?php
$livres = DB::table('livres')->whereIn('editeur_id', [10, 11, 12])->get();
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

Avec cette requête SQL :

```
sql
select * from livres where editeur_id in ('10', '11', '12')
```

Ordonner et grouper

On peut aussi ordonner et grouper :

```
php
<?php
$livres = App\Livre::
select('editeur_id', DB::raw('count(id) as livre_count'))
->groupBy('editeur_id')
->get();
foreach ($livres as $livre) {
    echo $livre->editeur_id . ' => ' . $livre->livre_count, '<br>';
}
```

Ici on groupe les lignes par éditeur et on compte les livres. La clause `select` mérite un petit commentaire.

Quand on ne sait pas faire quelque chose avec le Query Builder on a toujours la ressource d'utiliser une expression brute avec `DB::raw` comme je l'ai fait ici pour utiliser la fonction `count` de SQL.



Si vous utilisez une expression brute dans une requête celle-ci n'est pas immunisée contre les injections SQL, vous devez donc prendre vous-même des mesures de sécurité.

En version Query Builder :

```
php
<?php
$livres = DB::table('livres')
->select('editeur_id', DB::raw('count(id) as livre_count'))
->groupBy('editeur_id')
->get();
foreach ($livres as $livre) {
    echo $livre->editeur_id . ' => ' . $livre->livre_count, '<br>';
}
```

Avec cette requête SQL :

```
select editeur_id, count(id) as livre_count from livres group by editeur_id
```

sql

Les jointures

Trouver les titres des livres pour un éditeur dont on a l'id

Pour le moment on a vu des requêtes qui ne concernent qu'une seule table, ce qui n'est pas le plus répandu.

Lorsque deux tables sont concernées on doit faire une jointure. Considérons un premier exemple : je veux les titres des livres pour un éditeur dont j'ai l'id :

```
<?php
$livres = App\Editeur::find(11)->livres;
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

php

Eloquent est à l'aise dans ce genre de recherche. Avec le Query Builder on doit faire :

```
<?php
$livres = DB::table('editeurs')
->where('editeurs.id', 11)
->join('livres', 'livres.editeur_id', '=', 'editeurs.id')
->get();
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

php

La jointure s'effectue avec la méthode `join`. Les requêtes générées dans les deux cas ne sont pas les mêmes, avec Eloquent on en a deux :

```
select * from editeurs where id = '11' limit 1
select * from livres where livres.editeur_id = '11'
```

sql

Et avec Query Builder une seule :

```
select * from editeurs inner join livres on livres.editeur_id = editeurs.id where editeurs.id = '11'
```

sql

En fait Eloquent dans ce cas ne fait pas de jointure.

Trouver les livres d'un auteur dont on connaît le nom

Maintenant cherchons les livres d'un auteur dont on connaît le nom :

```
<?php
$livres = App\Livre::whereHas('auteurs', function($q)
{
    $q->whereNom('Osbaldo White');
})->get();
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

php

Là encore Eloquent s'en tire élégamment. Il génère la requête suivante :

```
select * from livres where (select count(*) from auteurs inner join auteur_livre on auteurs.id =
auteur_livre.auteur_id where auteur_livre.livre_id = livres.id and nom = 'Osbaldo White') >= '1'
```

sql

La requête est un peu acrobatique mais elle fonctionne. Avec le Query Builder on peut écrire quelque chose

comme ça :

```
<?php
$livres = DB::table('livres')
->join('auteur_livre', 'livres.id', '=', 'auteur_livre.livre_id')
->join('auteurs', 'auteurs.id', '=', 'auteur_livre.auteur_id')
->where('auteurs.nom', '=', 'Osbaldo White')
->get();
foreach ($livres as $livre) {
    echo $livre->titre, '<br>';
}
```

php

Une double jointure qui génère la requête :

```
select * from livre inner join auteur_livre on livres.id = auteur_livre.livre_id inner join auteurs on auteurs.id = auteur_livre.auteur_id where auteurs.nom = 'Osbaldo White'
```

sql

Trouver les auteurs pour un éditeur dont on connaît l'id

Allons plus loin et trouvons les auteurs pour un éditeur dont on connaît l'id :

```
<?php
$livres = App\Editeur::find(10)->livres;
foreach ($livres as $livre) {
    foreach($livre->auteurs as $auteur) {
        echo $auteur->nom, '<br>';
    }
}
```

php

On trouve facilement les livres de l'éditeur, mais pour aller jusqu'aux auteurs il faut boucler sur les résultats. Voici les 3 requêtes générées :

```
select * from editeurs where id = '10' limit 1
select * from livres where livres.editeur_id = '10'
select auteurs.*, auteur_livre.livre_id as pivot_livre_id, auteur_livre.auteur_id as pivot_auteur_id from auteurs
inner join auteur_livre on auteurs.id = auteur_livre.auteur_id where auteur_livre.livre_id = '28'
```

sql

C'est bien optimisé. Voici ce qu'on peut réaliser avec le Query Builder :

```
<?php
$auteurs = DB::table('auteurs')
->select('auteurs.nom')
->join('auteur_livre', 'auteurs.id', '=', 'auteur_livre.auteur_id')
->join('livres', 'livres.id', '=', 'auteur_livre.livre_id')
->join('editeurs', function($join)
{
    $join->on('editeurs.id', '=', 'livres.editeur_id')
        ->where('editeurs.id', '=', 10);
})
->get();
foreach($auteurs as $auteur) {
    echo $auteur->nom, '<br>';
}
```

php

C'est évidemment plus compact au point de vue requête avec une seule :

```
select auteurs.nom from auteurs inner join auteur_livre on auteurs.id = auteur_livre.auteur_id inner join livres
on livres.id = auteur_livre.livre_id inner join editeurs on editeurs.id = livres.editeur_id and editeurs.id =
'10'
```

sql

Attention aux requêtes imbriquées

Il faut être prudent dans certaines situations. Par exemple supposez que vous voulez avoir la liste des auteurs avec pour chaque nom d'auteur la liste de ses ouvrages. Vous pourriez écrire ce genre de code :

```
<?php
$auteurs = App\Auteur::all();
foreach ($auteurs as $auteur) {
    echo '<h1>' . $auteur->nom . '</h1>';
    foreach($auteur->livres as $livre) {
        echo $livre->titre, '<br>';
    }
}
```

Ça fonctionne très bien mais... si vous regardez vos requêtes vous allez être effrayé ! Dans mon cas j'en trouve 41 ! Tout simplement parce que pour chaque auteur vous lancez une requête pour trouver ses livres. Dans ce genre de situation il faut absolument utiliser le chargement lié, c'est-à-dire demander à Eloquent de charger la table livres avec la méthode with :

```
<?php
$auteurs = App\Auteur::with('livres')->get();
foreach ($auteurs as $auteur) {
    echo '<h1>' . $auteur->nom . '</h1>';
    foreach($auteur->livres as $livre) {
        echo $livre->titre, '<br>';
    }
}
```

Le changement peut paraître minime mais on n'a plus que 2 requêtes maintenant :

```
sql
select * from auteurs
select livres.*, auteur_livre.auteur_id as pivot_auteur_id, auteur_livre.livre_id as pivot_livre_id from livres
inner join auteur_livre on livres.id = auteur_livre.livre_id where auteur_livre.auteur_id in ('1', '2', '3', '4',
'5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '20', '21', '22', '23',
'24', '25', '26', '27', '28', '29', '30', '31', '32', '33', '34', '35', '36', '37', '38', '39', '40')
```

Comment faire la même chose avec le Query Builder ? Ce n'est pas si simple, voici une solution avec utilisation d'une expression brute :

```
php
<?php
$results = DB::table('auteurs')
->select('nom', DB::raw('group_concat(titre) as titres'))
->groupBy('nom')
->join('auteur_livre', 'auteurs.id', '=', 'auteur_livre.auteur_id')
->join('livres', 'livres.id', '=', 'auteur_livre.livre_id')
->get();
foreach ($results as $result) {
    echo '<h1>' . $result->nom . '</h1>';
    $titres = explode(',', $result->titres);
    foreach($titres as $titre) {
        echo $titre, '<br>';
    }
}
```

Maintenant on n'a plus qu'une seule requête :

```
sql
select nom, group_concat(titre) as titres from auteurs inner join auteur_livre on auteurs.id =
auteur_livre.auteur_id inner join livres on livres.id = auteur_livre.livre_id group by nom
```

Vous n'arrivez pas toujours à réaliser ce que vous désirez avec seulement Eloquent, il vous faudra alors utiliser le Query Builder. Nous avons vu dans ces quelques exemples comment l'utiliser.

Ce chapitre est loin d'épuiser le sujet du Query Builder. Vous pourrez trouver tous les compléments utiles sur [cette page de la documentation](#).

En résumé

- Eloquent permet de faire beaucoup de manipulations sur les tables et est à l'aise avec les relations.
- Le Query Builder est le compagnon parfait pour Eloquent.
- Parfois il est plus efficace d'utiliser le Query Builder qu'Eloquent.
- Parfois on doit utiliser des expressions brutes dans les requêtes mais il faut alors penser à se protéger des injections SQL.



Les commandes et les assistants



Quiz : Quiz 2

L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

OpenClassrooms



Qui sommes-nous ?

Fonctionnement de nos cours

Recrutement

Nous contacter

Professionnels



Affiliation

Entreprises

Universités et écoles

En plus



Créer un cours

Découvrez le framework PHP Laravel

15 heures Moyenne

Licence



Déploiement

Connectez-vous ou [inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Une application se développe et se teste en local mais arrive un moment où il faut la mettre sur un serveur pour qu'elle devienne visible et accessible. Ce déploiement n'est pas forcément une tâche aisée selon le contexte. Dans ce chapitre nous allons faire un petit tour d'horizon de ce qu'il convient de faire sans pouvoir être exhaustif étant donné la multiplicité des configurations existantes.

L'environnement

Créer des environnements

La version 5 de Laravel a un peu révolutionné les habitudes pour l'environnement par rapport à la version 4. Elle fait usage maintenant d'un package tiers : [DotEnv](#). On y a gagné en clarté mais pas forcément en ergonomie. Voyons comment cela se présente. Quand vous installez Laravel avec Composer vous trouvez à la racine le fichier `.env` avec ce contenu :

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=oJiCzJoPu9FvTjmTz1jpsNHVSB7i1n1
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

Ça se présente sous la forme clé/valeur. Par exemple on a déjà vu cela en oeuvre dans

`config/app.php` :

```
<?php
'debug' => env('APP_DEBUG', false),
```

Plus loin avec Laravel

- ▶ 1. **Déploiement**
- 2. Des vues propres (1/2)
- 3. Des vues propres (2/2)
- 4. La localisation
- 5. Ajax
- 6. Les tests unitaires
- 7. Événements et autorisations
- Quiz : Quiz 3
- Activité : Perfectionnez votre site de sondages

[Accéder au forum](#)



L'helper `env` permet d'aller lire la valeur de `APP_DEBUG` dans le fichier `.env` et d'affecter la valeur de la clé `debug` de la configuration de l'application.

On trouve aussi par exemple le réglage de MySql :

```
<?php
'host'      => env('DB_HOST', 'localhost'),
'database'  => env('DB_DATABASE', 'forge'),
'username'   => env('DB_USERNAME', 'forge'),
'password'   => env('DB_PASSWORD', ''),
```

php

Remarquez que l'helper accepte une valeur par défaut comme deuxième paramètre. Que fait cet helper ? Tout simplement il stocke les valeurs dans la super globale d'environnement `$_ENV` du serveur.

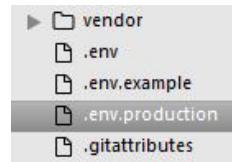
Moralité : pour avoir plusieurs environnements il faut créer plusieurs fichiers `.env`. Le cas le plus classique sera d'avoir un environnement local de développement et un autre de déploiement. Il faut juste ne pas se mélanger les pinceaux, en particulier ne pas enlever le fichier `.gitignore` de la racine qui prévoit de ne pas envoyer le fichier `.env` qui peut contenir des données sensibles (si vous utilisez git).

La variable `APP_ENV` dans le fichier `.env` est justement destinée à connaître l'environnement actuel :

```
APP_ENV=local
```

text

Lorsque je crée une application je fais une copie du fichier d'environnement pour régler mes variables en situation production. Pour ne pas qu'il y ait de confusion je le nomme de façon explicite :



Le fichier d'environnement "production"

Dans ce fichier je fixe toutes les valeurs nécessaires :

```
APP_ENV=production
APP_DEBUG=false
APP_KEY=5avz0M3IGPu4GFxBBLPhQs00MNJREzoL
...
```

text

Il me suffit ensuite d'envoyer ce fichier là sur le serveur et de le renommer.

 Selon où vous hébergez votre application vous pouvez avoir la possibilité de définir les variables d'environnement sans avoir à utiliser le fichier `.env`. C'est le cas si vous utilisez par exemple [Forge](#).

Le déploiement

Nous allons envisager plusieurs situations pour le déploiement selon votre hébergement.

Les besoins de Laravel

Laravel n'a pas besoin de grand chose mais quand même :

- PHP >= 5.5.9
- Extension PHP PDO
- Extension PHP OpenSSL
- Extension PHP Mbstring
- Extension PHP Tokenizer

D'autre part le dossier `storage` doit avoir les droits d'écriture sur le serveur.

Selon la méthode d'installation que vous allez employer vérifiez que vous avez bien une clé de cryptage dans votre environnement.

La solution royale

Commençons par la situation idéale : vous disposez de SSH et vous avez un dossier disponible avec possibilité d'avoir des dossiers non accessibles.

Si vous disposez de `git` sur le serveur vous pouvez évidemment l'utiliser et vous n'aurez aucun souci.

Si vous avez Composer installé globalement sur le serveur c'est aussi parfait. Vous pouvez alors installer l'installateur de Laravel :

```
composer global require "laravel/installer"
```

Il ne vous reste plus alors qu'à utiliser cet installateur :

```
laravel new application
```

Mais il est probable que vous ne disposiez pas de Composer installé. Dans ce cas vous commencez par envoyer [l'archive phar de composer](#). Vous accédez à votre dossier avec SSH :

```
login as: mon_login
login@mon_login.org's password:*****
```

Ensuite vous lancez la création avec composer :

```
php composer create-project --prefer-dist laravel/laravel
```

Vous attendez quelques minutes que toutes les dépendances s'installent.

Ca devrait fonctionner mais avec un Laravel anonyme qui n'est pas encore une application :



Laravel installé

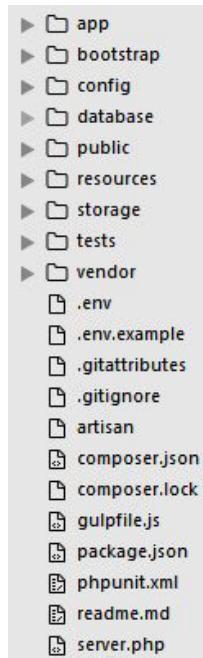
Il va falloir ensuite envoyer le dossier app, le `composer.json`, le fichier `.env`, créer une base de données... tout ce qui spécifie une application.

 Avec SSH il existe des applications de déploiement automatisé comme [envoy](#), [capistrano](#) ou [magallanes](#).

La solution intermédiaire

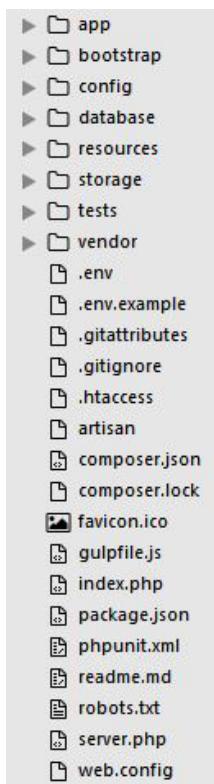
Supposons maintenant que vous disposez de SSH mais que vous êtes obligé de mettre votre application complètement accessible. Dans ce cas vous devez supprimer le dossier `public` et effectuer quelques manipulations pour que ça fonctionne. Voici la procédure :

Commencez par installer normalement Laravel. Vous devez avoir cette architecture :



L'architecture de Laravel

On va commencer par transférer tout ce qui se trouve dans le dossier public à la racine et supprimer ce dossier. Vous devez avoir maintenant cette situation :



Le dossier public a été supprimé et son contenu transféré

Evidemment maintenant plus rien ne fonctionne !

Ouvrez le fichier `index.php` et trouvez cette ligne :

```
<?php
require __DIR__.'/../bootstrap/autoload.php';
```

php

Remplacez-la par celle-ci :

```
<?php
require __DIR__.'/bootstrap/autoload.php';
```

php

Dans le même fichier trouvez cette ligne :

```
<?php  
$app = require_once __DIR__.'/.../bootstrap/app.php';
```

php

Remplacez-la par celle-ci :

```
<?php  
$app = require_once __DIR__.'/bootstrap/app.php';
```

php

Ces modifications sont nécessaires à cause du déplacement de ce fichier et de la modification de son emplacement relatif.

Maintenant la page d'accueil de Laravel va s'afficher correctement.

Le souci c'est que maintenant tout devient accessible et il faut prendre des précautions, par exemple en prévoyant dans les dossiers que vous voulez inaccessibles (app, bootstrap...) un `.htaccess` avec :

```
Deny from all
```

Les seuls fichiers accessibles restent ceux qui sont présents à la racine.

La solution laborieuse

Si vous ne disposez pas de SSH l'affaire devient plus laborieuse parce que la seule solution qui vous reste est le FTP. Comme il y a beaucoup de fichiers l'affaire peut être longue selon la vitesse de votre transmission.

D'autre part les mises à jour doivent être soigneusement gérées parce qu'elles vont aussi forcément passer par le FTP.

Le mode maintenance

Quand vous mettez à jour votre application il peut y avoir des moments où il vaut mieux que personne ne se connecte. Laravel propose un mode maintenance facile à mettre en oeuvre avec Artisan :

```
php artisan down
```

Et bien sûr il y a la commande inverse :

```
php artisan up
```

Évidemment encore faut-il que vous ayez accès à Artisan sur votre serveur...



En mode maintenance la vue affichée est celle située là :

text

```
resources/views/errors/503.blade.php
```

En résumé

- Il faut créer des environnements pour répondre aux différentes situations : local, distant...
- Le déploiement est facile et rapide si on dispose de git ou du SSH.
- On peut supprimer le dossier `public` tout en ayant un Laravel fonctionnel, il faut juste prendre des mesures pour assurer la sécurité.
- Il est possible de déployer une application avec le FTP mais c'est un peu laborieux et les mises à jour doivent être soigneusement gérées.

Activité : Construisez un site de sondages avec une base de données

Des vues propres (1/2)



[S'inscrire](#)

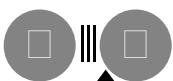
Se connecter



Accueil ▶ Cours ▶ Découvrez le framework PHP Laravel ▶ Des vues propres (1/2)

Découvrez le framework PHP Laravel

15 heures Moyenne



Des vues propres (1/2)

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

On a vu comment créer des classes bien organisées et se contentant d'effectuer leur tâche. Par contre au niveau des vues c'est une autre histoire. En général on utilise un framework CSS, par exemple Bootstrap dont je me suis servi dans les exemples de ce cours. Mais que se passe-t-il le jour où ce framework évolue ou si on décide d'en changer ?

Les macros

Le problème

Regardez la vue que nous avons créée dans l'exemple de blog personnel pour la création d'un article

(`resources/views/posts/add.blade.php`) :

```
html
@extends('template')

@section('contenu')
    <br>
    <div class="col-sm-offset-3 col-sm-6">
        <div class="panel panel-info">
            <div class="panel-heading">Ajout d'un article</div>
            <div class="panel-body">
                {!! Form::open(['route' => 'post.store']) !!}
                    <div class="form-group {!! $errors->has('titre') ? 'has-error' : '' !!}>
                        {!! Form::text('titre', null, ['class' => 'form-control',
'placeholder' => 'Titre']) !!}
                        {!! $errors->first('titre', '<small class="help-block">:message</small>') !!}
                    </div>
                    <div class="form-group {!! $errors->has('contenu') ? 'has-error' : '' !!}>
                        {!! Form::textarea ('contenu', null, ['class' => 'form-control',
'placeholder' => 'Contenu']) !!}
                        {!! $errors->first('contenu', '<small class="help-block">:message</small>') !!}
                    </div>
                
```

```

        </div>
        <div class="form-group {{ $errors->has('tags') ? 'has-error' : '' }}">
            {!! Form::text('tags', null, array('class' => 'form-control',
'placeholder' => 'Entrez les tags séparés par des virgules')) !!}
            {!! $errors->first('tags', '<small class="help-block">:message</small>') !!}
        </div>
        <a href="javascript:history.back()" class="btn btn-primary">
            <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
        </a>
    </div>
@endsection

```

Quel est le degré de dépendance entre Bootstrap et ce code ? Petit point :

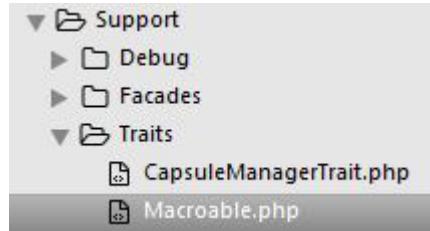
- mise en page avec la grille,
- utilisation du composant `panel`,
- utilisation des classes `btn` , `btn-info` , `form-control` , `form-group` .. pour le formulaire,
- utilisation de la classe `has-error` pour le retour de validation,
- ...

Si je décide de changer de framework je dois tout recommencer. Si je n'ai qu'un formulaire de ce genre ce n'est pas trop grave, si j'en ai 10 ou 20 ça devient embarrassant. De même si le framework évolue (ce qui est fréquent par exemple avec Bootstrap) et que je désire suivre cette évolution je vais être confronté au même problème.

L'idéal serait de pouvoir construire ma vue sans aucun lien avec un framework avec des méthodes neutres. Comment réaliser cela ? Il y a plusieurs façons de le faire mais la plus simple est certainement l'utilisation de macros.

Les macros

Une macro est un outil de substitution : on utilise un texte et ça en génère un autre avec possibilité de passer des paramètres. Laravel permet la création de macros. Regardez dans les dossiers du framework pour trouver ce fichier :



Le fichier pour les macros

Si vous regardez le code vous allez voir qu'il s'agit d'un trait qui expose quelques méthode dont :

```

<?php
public static function macro($name, callable $macro)
{
    static::$macros[$name] = $macro;
}

```

php

En utilisant ce trait on peut donc enregistrer des macros dans la propriété statique `$macros`. Il y a également des méthodes magiques permettant d'utiliser les macros mémorisées.

Il se trouve que le composant `LaravelCollective\Html` utilise ce trait. Par exemple dans la classe `FormBuilder` on trouve :

```
<?php
class FormBuilder {

    use Macroable, Componentable {
        Macroable::__call as macroCall;
        Componentable::__call as componentCall;
    }
}
```

Il en est de même pour la classe `HtmlBuilder`. Nous allons donc utiliser cette possibilité pour rendre notre vue plus propre...

On veut arriver avec une vue ainsi conçue :

```
@extends('template')

@section('contenu')
<br>
<div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-info">
        <div class="panel-heading">Ajout d'un article</div>
        <div class="panel-body">
            {!! Form::open(['route' => 'post.store']) !!}
            {!! Form::control('text', $errors, 'titre', 'Titre') !!}
            {!! Form::control('textarea', $errors, 'contenu', 'Contenu') !!}
            {!! Form::control('text', $errors, 'tags', 'Entrez les tags séparés par
des virgules') !!}
            {!! Form::button_submit('Envoyer !') !!}
            {!! Form::close() !!}
        </div>
    </div>
    {!! Html::button_back() !!}
</div>
@endsection
```

Nous n'avons plus aucune trace de notre framework au niveau du formulaire.

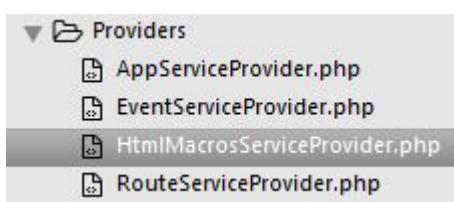
Maintenant la question est : où mettre les macros ?

Le service provider

Un service provider est le lieu idéal pour enregistrer des services. Laravel comporte de nombreux services providers. Nous allons en créer un pour nos macros. Encore une fois artisan va nous faciliter la tâche :

```
php artisan make:provider HtmlMacrosServiceProvider
```

On trouve notre provider dans le dossier correspondant :



Le provider pour les macros

Avec ce code :

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class HtmlMacrosServiceProvider extends ServiceProvider
{

    /**
     * Bootstrap the application services.
     *
     * @return void
     */
    public function boot()
    {
        //
    }

    /**
     * Register the application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

La méthode `register` permet d'enregistrer tout ce dont on a besoin, donc nos macros.

Info La méthode `boot` est exécutée une fois que tous les providers de l'application ont été enregistrés, ce qui permet alors de disposer de tous les services.

On va donc créer nos macros ici :

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Collective\Html\FormBuilder;
use Collective\Html\HtmlBuilder;

class HtmlMacrosServiceProvider extends ServiceProvider
{

    public function register()
    {
        $this->registerFormControl();
        $this->registerFormSubmit();
        $this->registerHtmlButtonBack();
    }

    private function registerFormControl()
    {
        FormBuilder::macro('control', function($type, $errors, $nom, $placeholder)
```

```

    {
        $valeur = \Request::old($nom) ? \Request::old($nom) : null;
        $attributes = ['class' => 'form-control', 'placeholder' => $placeholder];
        return sprintf(
            <div class="form-group %s">
                %s
                %s
            </div>',
            $errors->has($nom) ? 'has-error' : '',
            call_user_func_array(['Form', $type], [$nom, $valeur, $attributes]),
            $errors->first($nom, '<small class="help-block">:message</small>')
        );
    });
}

private function registerFormSubmit()
{
    FormBuilder::macro('button_submit', function($texte)
    {
        return FormBuilder::submit($texte, ['class' => 'btn btn-info pull-right']);
    });
}

private function registerHtmlButtonBack()
{
    HtmlBuilder::macro('button_back', function()
    {
        return '<a href="javascript:history.back()" class="btn btn-primary">
                    <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
                </a>';
    });
}
}

```

Ce n'est qu'un exemple et le codage pourrait être différent, c'est le principe qui importe. On pourrait aussi localiser ailleurs les macros et se contenter de les appeler à partir du provider. C'est alors juste une question d'organisation du code. Il est évident que les providers ne sont pas faits pour accueillir un code volumineux, comme toute classe d'ailleurs. Dans le cadre de ce cours on va se contenter de cette présentation.

Il ne reste plus qu'à informer Laravel que notre provider existe dans

`config/app.php` :

```

<?php
/*
 * Application Service Providers...
 */
App\Providers\AppServiceProvider::class,
App\Providers\AuthServiceProvider::class,
App\Providers\EventServiceProvider::class,
App\Providers\RouteServiceProvider::class,
Collective\Html\HtmlServiceProvider::class,
App\Providers\HtmlMacrosServiceProvider::class,

```

Normalement le formulaire de création d'un article devrait encore fonctionner :

Ajout d'un article

Titre

Contenu

Entrez les tags séparés par des virgules

Envoyer !

Le formulaire généré par les macros

On a ainsi pu assainir la vue au niveau du formulaire.

Les templates

Mais il reste encore des éléments du framework dans notre vue : la grille et le panel. Là une macro ne serait pas vraiment adaptée, il vaut mieux créer un template pour ce formulaire qui pourra servir pour d'autres formulaires

(`app/views/template_form.blade.php`) :

```
@extends('template')  
  
@section('contenu')  
    <br>  
    <div class="col-sm-offset-3 col-sm-6">  
        <div class="panel panel-info">  
            <div class="panel-heading">  
                @yield('titre')  
            </div>  
            <div class="panel-body">  
                @yield('formulaire')  
            </div>  
        </div>  
        {!! Html::button_back() !!}  
    </div>  
@stop
```

Je vous rappelle le contenu du template principal (

`app/views/template.blade.php`) :

```
<!DOCTYPE html>  
<html lang="fr">  
    <head>
```

```
<meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Mon joli site</title>
    {!! HTML::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css') !!}
    {!! HTML::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css') !!}
{!!
<!--[if lt IE 9]>
    {{ HTML::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.js') }}
    {{ HTML::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js') }}
<![endif]-->
<style> textarea { resize: none; } </style>
</head>
<body>
    <header class="jumbotron">
        <div class="container">
            <h1 class="page-header">{!! link_to_route('post.index', 'Mon joli blog') !!}</h1>
            @yield('header')
        </div>
    </header>
    <div class="container">
        @yield('contenu')
    </div>
</body>
</html>
```

Maintenant on peut avoir une vue très épurée pour notre formulaire :

```
html
@extends('template_form')

@section('titre')
    Ajout d'un article
@stop

@section('formulaire')
    {!! Form::open(['route' => 'post.store']) !!}
        {!! Form::control('text', $errors, 'titre', 'Titre') !!}
        {!! Form::control('textarea', $errors, 'contenu', 'Contenu') !!}
        {!! Form::control('text', $errors, 'tags', 'Entrez les tags séparés par des virgules') !!}
        {!! Form::button_submit('Envoyer !') !!}
    {!! Form::close() !!}
@stop
```

Plus aucune trace de framework. Quelles que soient les modifications d'interface ultérieures cette vue n'aura pas à être modifiée. Si on change de framework on a juste le template général et celui des formulaires à modifier.

Vous devez bien organiser vos templates pour avoir du code efficace, ce qui dépend évidemment de votre application.

Si vous avez beaucoup de vues il est judicieux de créer des dossiers pour les classer. Par exemple vous pouvez créer un dossier "templates".

Une autre organisation du code peut amener à créer des vues partielles à insérer dans la vue finale, on parle ainsi de "partials". C'est très utilisé par exemple pour les barres de message d'erreur. Il existe ainsi de nombreuses façons d'organiser les vues, l'essentiel est d'arriver à un code simple, lisible et facile à maintenir.

En résumé

- L'utilisation de macros permet d'avoir des vues indépendantes du framework utilisé.
- Un service provider permet de faire des initialisations.
- La création de templates permet de bien organiser le code des vues.
- L'utilisation de dossiers pour classer les vues est souvent nécessaire lorsqu'il y a beaucoup de vues.

[S'inscrire](#)

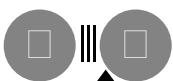
Se connecter



Accueil ▶ Cours ▶ Découvrez le framework PHP Laravel ▶ Des vues propres (1/2)

Découvrez le framework PHP Laravel

15 heures Moyenne



Des vues propres (1/2)

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

On a vu comment créer des classes bien organisées et se contentant d'effectuer leur tâche. Par contre au niveau des vues c'est une autre histoire. En général on utilise un framework CSS, par exemple Bootstrap dont je me suis servi dans les exemples de ce cours. Mais que se passe-t-il le jour où ce framework évolue ou si on décide d'en changer ?

Les macros

Le problème

Regardez la vue que nous avons créée dans l'exemple de blog personnel pour la création d'un article

(`resources/views/posts/add.blade.php`) :

```
html
@extends('template')

@section('contenu')
    <br>
    <div class="col-sm-offset-3 col-sm-6">
        <div class="panel panel-info">
            <div class="panel-heading">Ajout d'un article</div>
            <div class="panel-body">
                {!! Form::open(['route' => 'post.store']) !!}
                    <div class="form-group {!! $errors->has('titre') ? 'has-error' : '' !!}>
                        {!! Form::text('titre', null, ['class' => 'form-control',
'placeholder' => 'Titre']) !!}
                        {!! $errors->first('titre', '<small class="help-block">:message</small>') !!}
                    </div>
                    <div class="form-group {!! $errors->has('contenu') ? 'has-error' : '' !!}>
                        {!! Form::textarea ('contenu', null, ['class' => 'form-control',
'placeholder' => 'Contenu']) !!}
                        {!! $errors->first('contenu', '<small class="help-block">:message</small>') !!}
                    </div>
                
```

```

        </div>
        <div class="form-group {{ $errors->has('tags') ? 'has-error' : '' }}">
            {!! Form::text('tags', null, array('class' => 'form-control',
'placeholder' => 'Entrez les tags séparés par des virgules')) !!}
            {!! $errors->first('tags', '<small class="help-block">:message</small>') !!}
        </div>
        <a href="javascript:history.back()" class="btn btn-primary">
            <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
        </a>
    </div>
@endsection

```

Quel est le degré de dépendance entre Bootstrap et ce code ? Petit point :

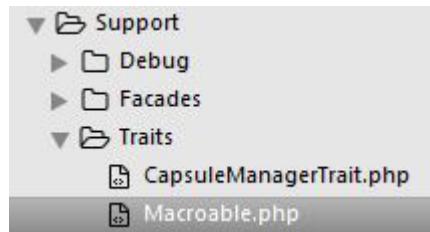
- mise en page avec la grille,
- utilisation du composant `panel`,
- utilisation des classes `btn` , `btn-info` , `form-control` , `form-group` .. pour le formulaire,
- utilisation de la classe `has-error` pour le retour de validation,
- ...

Si je décide de changer de framework je dois tout recommencer. Si je n'ai qu'un formulaire de ce genre ce n'est pas trop grave, si j'en ai 10 ou 20 ça devient embarrassant. De même si le framework évolue (ce qui est fréquent par exemple avec Bootstrap) et que je désire suivre cette évolution je vais être confronté au même problème.

L'idéal serait de pouvoir construire ma vue sans aucun lien avec un framework avec des méthodes neutres. Comment réaliser cela ? Il y a plusieurs façons de le faire mais la plus simple est certainement l'utilisation de macros.

Les macros

Une macro est un outil de substitution : on utilise un texte et ça en génère un autre avec possibilité de passer des paramètres. Laravel permet la création de macros. Regardez dans les dossiers du framework pour trouver ce fichier :



Le fichier pour les macros

Si vous regardez le code vous allez voir qu'il s'agit d'un trait qui expose quelques méthode dont :

```

<?php
public static function macro($name, callable $macro)
{
    static::$macros[$name] = $macro;
}

```

php

En utilisant ce trait on peut donc enregistrer des macros dans la propriété statique `$macros`. Il y a également des méthodes magiques permettant d'utiliser les macros mémorisées.

Il se trouve que le composant `LaravelCollective\Html` utilise ce trait. Par exemple dans la classe `FormBuilder` on trouve :

```
<?php
class FormBuilder {

    use Macroable, Componentable {
        Macroable::__call as macroCall;
        Componentable::__call as componentCall;
    }
}
```

Il en est de même pour la classe `HtmlBuilder`. Nous allons donc utiliser cette possibilité pour rendre notre vue plus propre...

On veut arriver avec une vue ainsi conçue :

```
@extends('template')

@section('contenu')
<br>
<div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-info">
        <div class="panel-heading">Ajout d'un article</div>
        <div class="panel-body">
            {!! Form::open(['route' => 'post.store']) !!}
            {!! Form::control('text', $errors, 'titre', 'Titre') !!}
            {!! Form::control('textarea', $errors, 'contenu', 'Contenu') !!}
            {!! Form::control('text', $errors, 'tags', 'Entrez les tags séparés par
des virgules') !!}
            {!! Form::button_submit('Envoyer !') !!}
            {!! Form::close() !!}
        </div>
    </div>
    {!! Html::button_back() !!}
</div>
@endsection
```

Nous n'avons plus aucune trace de notre framework au niveau du formulaire.

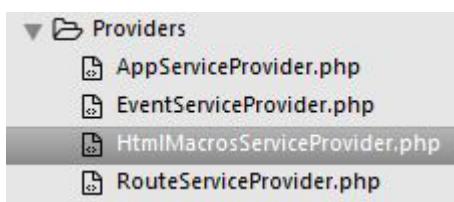
Maintenant la question est : où mettre les macros ?

Le service provider

Un service provider est le lieu idéal pour enregistrer des services. Laravel comporte de nombreux services providers. Nous allons en créer un pour nos macros. Encore une fois artisan va nous faciliter la tâche :

```
php artisan make:provider HtmlMacrosServiceProvider
```

On trouve notre provider dans le dossier correspondant :



Le provider pour les macros

Avec ce code :

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class HtmlMacrosServiceProvider extends ServiceProvider
{

    /**
     * Bootstrap the application services.
     *
     * @return void
     */
    public function boot()
    {
        //
    }

    /**
     * Register the application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

php

La méthode `register` permet d'enregistrer tout ce dont on a besoin, donc nos macros.



La méthode `boot` est exécutée une fois que tous les providers de l'application ont été enregistrés, ce qui permet alors de disposer de tous les services.

On va donc créer nos macros ici :

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Collective\Html\FormBuilder;
use Collective\Html\HtmlBuilder;

class HtmlMacrosServiceProvider extends ServiceProvider
{

    public function register()
    {
        $this->registerFormControl();
        $this->registerFormSubmit();
        $this->registerHtmlButtonBack();
    }

    private function registerFormControl()
    {
        FormBuilder::macro('control', function($type, $errors, $nom, $placeholder)
```

php

```

    {
        $valeur = \Request::old($nom) ? \Request::old($nom) : null;
        $attributes = ['class' => 'form-control', 'placeholder' => $placeholder];
        return sprintf(
            <div class="form-group %s">
                %s
                %s
            </div>',
            $errors->has($nom) ? 'has-error' : '',
            call_user_func_array(['Form', $type], [$nom, $valeur, $attributes]),
            $errors->first($nom, '<small class="help-block">:message</small>')
        );
    });
}

private function registerFormSubmit()
{
    FormBuilder::macro('button_submit', function($texte)
    {
        return FormBuilder::submit($texte, ['class' => 'btn btn-info pull-right']);
    });
}

private function registerHtmlButtonBack()
{
    HtmlBuilder::macro('button_back', function()
    {
        return '<a href="javascript:history.back()" class="btn btn-primary">
                    <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
                </a>';
    });
}
}

```

Ce n'est qu'un exemple et le codage pourrait être différent, c'est le principe qui importe. On pourrait aussi localiser ailleurs les macros et se contenter de les appeler à partir du provider. C'est alors juste une question d'organisation du code. Il est évident que les providers ne sont pas faits pour accueillir un code volumineux, comme toute classe d'ailleurs. Dans le cadre de ce cours on va se contenter de cette présentation.

Il ne reste plus qu'à informer Laravel que notre provider existe dans

`config/app.php` :

```

<?php
/*
 * Application Service Providers...
 */
App\Providers\AppServiceProvider::class,
App\Providers\AuthServiceProvider::class,
App\Providers\EventServiceProvider::class,
App\Providers\RouteServiceProvider::class,
Collective\Html\HtmlServiceProvider::class,
App\Providers\HtmlMacrosServiceProvider::class,

```

Normalement le formulaire de création d'un article devrait encore fonctionner :

Ajout d'un article

Titre

Contenu

Entrez les tags séparés par des virgules

Envoyer !

Le formulaire généré par les macros

On a ainsi pu assainir la vue au niveau du formulaire.

Les templates

Mais il reste encore des éléments du framework dans notre vue : la grille et le panel. Là une macro ne serait pas vraiment adaptée, il vaut mieux créer un template pour ce formulaire qui pourra servir pour d'autres formulaires

(`app/views/template_form.blade.php`) :

```
@extends('template')  
  
@section('contenu')  
    <br>  
    <div class="col-sm-offset-3 col-sm-6">  
        <div class="panel panel-info">  
            <div class="panel-heading">  
                @yield('titre')  
            </div>  
            <div class="panel-body">  
                @yield('formulaire')  
            </div>  
        </div>  
        {!! Html::button_back() !!}  
    </div>  
@stop
```

Je vous rappelle le contenu du template principal (

`app/views/template.blade.php`) :

```
<!DOCTYPE html>  
<html lang="fr">  
    <head>
```

html

```
<meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Mon joli site</title>
    {!! HTML::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css') !!}
    {!! HTML::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css') !!}
    {!!
        <!--[if lt IE 9]>
            {{ HTML::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.js') }}
            {{ HTML::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js') }}
        <![endif]-->
        <style> textarea { resize: none; } </style>
    </head>
    <body>
        <header class="jumbotron">
            <div class="container">
                <h1 class="page-header">{!! link_to_route('post.index', 'Mon joli blog') !!}</h1>
                @yield('header')
            </div>
        </header>
        <div class="container">
            @yield('contenu')
        </div>
    </body>
</html>
```

Maintenant on peut avoir une vue très épurée pour notre formulaire :

```
html
@extends('template_form')

@section('titre')
    Ajout d'un article
@stop

@section('formulaire')
    {!! Form::open(['route' => 'post.store']) !!}
        {!! Form::control('text', $errors, 'titre', 'Titre') !!}
        {!! Form::control('textarea', $errors, 'contenu', 'Contenu') !!}
        {!! Form::control('text', $errors, 'tags', 'Entrez les tags séparés par des virgules') !!}
        {!! Form::button_submit('Envoyer !') !!}
    {!! Form::close() !!}
@stop
```

Plus aucune trace de framework. Quelles que soient les modifications d'interface ultérieures cette vue n'aura pas à être modifiée. Si on change de framework on a juste le template général et celui des formulaires à modifier.

Vous devez bien organiser vos templates pour avoir du code efficace, ce qui dépend évidemment de votre application.

Si vous avez beaucoup de vues il est judicieux de créer des dossiers pour les classer. Par exemple vous pouvez créer un dossier "templates".

Une autre organisation du code peut amener à créer des vues partielles à insérer dans la vue finale, on parle ainsi de "partials". C'est très utilisé par exemple pour les barres de message d'erreur. Il existe ainsi de nombreuses façons d'organiser les vues, l'essentiel est d'arriver à un code simple, lisible et facile à maintenir.

En résumé

- L'utilisation de macros permet d'avoir des vues indépendantes du framework utilisé.
 - Un service provider permet de faire des initialisations.
 - La création de templates permet de bien organiser le code des vues.
- L'utilisation de dossiers pour classer les vues est souvent nécessaire lorsqu'il y a beaucoup de vues.

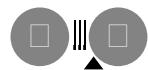
Découvrez le framework PHP Laravel



15 heures

Moyenne

Licence



Des vues propres (2/2)

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Nous allons continuer notre démarche de nettoyage des vues qui nous sert en fait de prétexte pour explorer le conteneur de dépendances qui constitue le cœur de Laravel. Pourquoi conteneur ? Parce que c'est une sorte de boîte dans laquelle on peut mettre plein de choses. Je vous ai déjà parlé de l'injection de dépendance quand on a vu les contrôleurs. C'est le conteneur qui permet de gérer facilement ces injections. Nous avons déjà utilisé le conteneur pour gérer la résolution des dépendances. Dans ce chapitre je vais poursuivre l'exemple des macros vu au chapitre précédent mais en utilisant maintenant une classe.

La nouvelle vue

Où en sommes-nous du nettoyage du code du formulaire de création d'un article ? Dans le précédent chapitre on l'a séparé en deux avec un template (`resources/views/template_form.blade.php`) :

```
@extends('template')

@section('contenu')
<br>
    <div class="col-sm-offset-3 col-sm-6">
        <div class="panel panel-info">
            <div class="panel-heading">
                @yield('titre')
            </div>
            <div class="panel-body">
                @yield('formulaire')
            </div>
        </div>
        {!! Html::button_back() !!}
    </div>
@endsection
```

Et le formulaire lui-même épuré grâce aux macros (`resources/views/post/add.blade.php`):

```
@extends('template_form')

@section('titre')
    Ajout d'un article
@endsection

@section('formulaire')
    {!! Form::open(['route' => 'post.store']) !!}
        {!! Form::control('text', $errors, 'titre', 'Titre') !!}
        {!! Form::control('textarea', $errors, 'contenu', 'Contenu') !!}
        {!! Form::control('text', $errors, 'tags', 'Entrez les tags séparés par des virgules') !!}
    {!! Form::button_submit('Envoyer !') !!}
    {!! Form::close() !!}
@endsection
```

Plus loin avec Laravel

1. Déploiement
 2. Des vues propres (1/2)
 - ▶ **3. Des vues propres (2/2)**
 4. La localisation
 5. Ajax
 6. Les tests unitaires
 7. Événements et autorisations
- Quiz : Quiz 3
 Activité : Perfectionnez votre site de sondages

[Accéder au forum](#)



@endsection

Ce que je vous propose maintenant c'est de remplacer les macros par des méthodes d'une classe dédiée et d'ajouter le traitement du panneau. Le but étant de se passer du template et d'aboutir à cette vue :

```
@extends('template')

@section('contenu')
<br>
<div class="col-sm-offset-3 col-sm-6">
    {!! Panel::head(
        'Ajout d\'un article'
    ) !!}
    ->body(
        Form::open(['route' => 'post.store']).!!
        Form::control('text', $errors, 'titre', 'Titre').!!
        Form::control('textarea', $errors, 'contenu', 'Contenu').!!
        Form::control('text', $errors, 'tags', 'Entrez les tags séparés
par des virgules').!!
        Form::button_submit('Envoyer !').!!
        Form::close()
    ) !!}
    ->type('primary')
    !!}
    {!! Html::button_back() !!}
</div>
@endsection
```

html

On ne voit évidemment aucune différence pour le formulaire lui-même puisque celle-ci va se situer au niveau de l'intendance. Par contre apparaît une nouvelle classe (`Panel`) avec des méthodes pour construire le panneau.

Évidemment l'aspect de la vue lui ne doit pas changer :

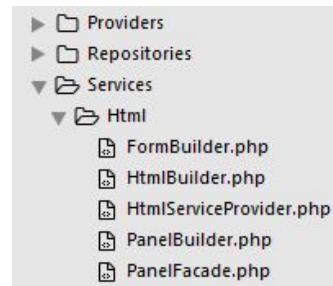
La vue de création d'un article

Organisation du code

Dans le précédent chapitre on a mis le code des macros dans le service provider. C'était simple et efficace. Cette solution est envisageable lorsque le code n'est pas trop fourni ou complexe, sinon il devient vite indispensable de le répartir dans plusieurs autres classes.

La notion de "service" est suffisamment générale pour inclure la plupart des tâches à réaliser dans une

application. Je vous propose de créer un service pour gérer ces fonctionnalités pour les vues. On va créer les dossiers `Services` et `Html` pour mettre tout le code dont nous allons avoir besoin pour ce chapitre. Au final nous aurons cette organisation :



L'organisation des fichiers

Voyons un peu à quoi tout ça va servir :

- **FormBuilder** : va contenir les méthodes `control` et `button_submit`, les deux qui concernent le formulaire,
- **HtmlBuilder** : va contenir la méthode `button_back`,
- **HtmlServiceProvider** : c'est le service provider qui va initialiser tous les composants,
- **PanelBuilder** : va contenir les méthodes pour le panneau,
- **PanelFacade** : va générer la façade pour le panneau, on pourra ainsi utiliser une syntaxe simplifiée.

Les builders

FormBuilder et HtmlBuilder

Pour créer ces deux builders on va juste reprendre le code des macros en le mettant dans deux classes séparées. Donc pour les deux méthodes du formulaire on aura la classe `FormBuilder` :

```

<?php
namespace App\Services\Html;

use Collective\Html\FormBuilder as CollectiveFormbuilder;
use Request;

class FormBuilder extends CollectiveFormbuilder
{
    public function control($type, $errors, $nom, $placeholder)
    {
        $valeur = Request::old($nom) ? Request::old($nom) : null;
        $attributes = ['class' => 'form-control', 'placeholder' => $placeholder];
        return sprintf(
            <div class="form-group %s">
                %s
                %s
            </div>,
            $errors->has($nom) ? 'has-error' : '',
            call_user_func_array(['Form', $type], [$nom, $valeur, $attributes]),
            $errors->first($nom, '<small class="help-block">:message</small>')
        );
    }

    public function button_submit($texte)
    {
        return parent::submit($texte, ['class' => 'btn btn-info pull-right']);
    }
}
  
```

Vous remarquez que je me contente d'étendre la classe `FormBuilder` du package pour en conserver toutes les méthodes.

Et pour ce qui est du html la classe `HtmlBuilder` :

```

<?php

namespace App\Services\Html;

use Collective\Html\HtmlBuilder as CollectiveHtmlBuilder;

class HtmlBuilder extends CollectiveHtmlBuilder
{
    public function button_back()
    {
        return '<a href="javascript:history.back()" class="btn btn-primary">
            <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
        </a>';
    }
}

```

Ici aussi je me contente d'étendre la classe du package pour ajouter ma méthode.

PanelBuilder

Pour le panneau on va aussi créer une classe (`PanelBuilder`) :

```

<?php

namespace App\Services\Html;

class PanelBuilder
{
    protected $type;
    protected $head;
    protected $body;
    protected $foot;

    public function __construct($type = 'default', $head = null, $body = null, $foot = null)
    {
        $this->type = $type;
        $this->head = $head;
        $this->body = $body;
        $this->foot = $foot;
    }

    public function __toString()
    {
        $s = '<div class="panel panel-'.$this->type.'">';
        if ($this->head)
        {
            $s .= '<div class="panel-heading">'.$this->head.'</div>';
        }
        if ($this->body)
        {
            $s .= '<div class="panel-body">'.$this->body.'</div>';
        }
        if ($this->foot)
        {
            $s .= '<div class="panel-footer">'.$this->foot.'</div>';
        }
        $s .= "</div>";
        return $s;
    }

    public function type($type)
    {
        $this->type = $type;
        return $this;
    }

    public function head($head)
    {
        $this->head = $head;
        return $this;
    }

    public function body($body)
    {
        $this->body = $body;
    }
}

```

```

    return $this;
}

public function footer($foot)
{
    $this->foot = $foot;
    return $this;
}

}

```

J'ai prévu de pouvoir transmettre les paramètres soit dans le constructeur, soit au moyen de méthodes dédiées pour un peu généraliser le code.

Il faut aussi créer la façade pour la classe `Panel` :

```

<?php
php

namespace App\Services\Html;

use Illuminate\Support\Facades\Facade;

class PanelFacade extends Facade
{

    protected static function getFacadeAccessor() { return 'panel'; }

}

```

La syntaxe d'une façade est toute simple, on renvoie juste la référence 'panel'. Nous allons voir bientôt d'où sort cette référence et comment on la nomme.

Service provider et façade

C'est dans le provider que va se situer l'intendance pour le framework. En voici le code :

```

<?php
php

namespace App\Services\Html;

use Illuminate\Support\ServiceProvider;

class HtmlServiceProvider extends ServiceProvider
{

    public function register()
    {
        $this->registerHtmlBuilder();
        $this->registerFormBuilder();
        $this->registerPanelBuilder();
    }

    protected function registerHtmlBuilder()
    {
        $this->app->singleton('html', function ($app) {
            return new HtmlBuilder($app['url'], $app['view']);
        });
    }

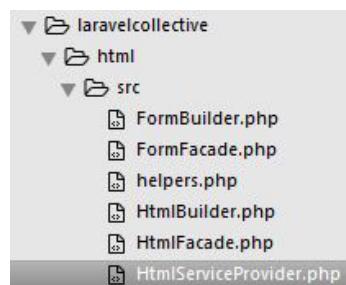
    protected function registerFormBuilder()
    {
        $this->app->singleton('form', function ($app) {
            $form = new FormBuilder($app['html'], $app['url'], $app['view'], $app['session.store']->getToken());
            return $form->setSessionStore($app['session.store']);
        });
    }

    protected function registerPanelBuilder()
    {
        $this->app->singleton('panel', function()
        {
            return new PanelBuilder();
        });
    }
}

```

{}

Je vous ai déjà dit que les providers servent à enregistrer des choses : des liaisons de dépendance, des événements, des éléments de configuration. Ici nous enregistrons les méthodes pour les formulaires et le html. Il existe déjà un provider dans le package :



Le provider du package

Mais je suis obligé de surcharger la plupart de ses méthodes.

La méthode la plus importante d'un provider est `register`. Lorsqu'une requête est prise en charge par Laravel on a besoin de tout un tas de choses : configuration des erreurs, détection de l'environnement, définition des middlewares que la requête devra traverser, vérification du CSRF... A un moment on va aussi passer en revue les providers et exécuter leur méthode `register`, les uns après les autres dans leur ordre d'inscription. On en profite pour déclarer toutes les dépendances dans le conteneur.

Le conteneur est un peu le grand sac de l'application qui permet de tout ranger et de tout retrouver. Par exemple avec cette méthode du provider :

```
<?php
protected function registerPanelBuilder()
{
    $this->app->singleton('panel', function()
    {
        return new PanelBuilder();
    });
}
```

php

On va dire au conteneur : tu vas te souvenir de 'panel' et si j'en ai besoin j'aurais juste à te dire 'panel' et tu me renverras une instance de `PanelBuilder`. Le conteneur va donc créer un lien (`singleton`) entre la référence 'panel' et la classe `PanelBuilder`.



Si on utilise `singleton` c'est qu'on veut une unique résolution, sinon on utilisera `bind`

Du coup vous allez mieux comprendre le code de la façade :

```
<?php
class PanelFacade extends Facade
{
    protected static function getFacadeAccessor() { return 'panel'; }
}
```

php

Là on dit que si on a la façade `Panel` on doit aller chercher dans le conteneur l'instance de la classe qui correspond à 'panel'. Et comme on a pris le soin dans le provider de créer un lien entre cette référence et le nom de la classe ça va fonctionner !

Maintenant si j'écris :

```
<?php
Panel::head()
```

php

Laravel va comprendre que je parle de la façade, que je veux une instance de la classe `PanelBuilder` et que je veux utiliser la méthode `head`.

Ça fonctionnerait aussi avec ce code :

```
<?php  
app('panel')->head()
```

Ici je vais chercher directement dans le conteneur une instance de la classe. Mais je pourrais aussi l'écrire ainsi :

```
<?php  
App::make('panel')->head()
```

Cette fois j'utilise la façade de l'application (donc du conteneur parce que l'application en est une extension) et je demande de créer une instance de la classe référencée par 'panel'.

Il ne reste plus qu'à informer Laravel que notre provider existe. Cela se fait dans `config/app.php`. Pour installer le package on a écrit :

```
<?php  
Collective\Html\HtmlServiceProvider::class,
```

On va changer cette ligne pour pointer maintenant sur notre provider :

```
<?php  
App\services\Html\HtmlServiceProvider::class,
```

Il ne nous reste plus qu'à renseigner la façade pour le panel :

```
<?php  
'Panel'      => App\services\Html\PanelFacade::class,
```

Et tout devrait bien fonctionner !

Si vous rencontrez des difficultés, surtout si vous avez une erreur sur le fait qu'une classe n'existe pas, faites :

```
composer dumpautoload  
php artisan clear-compiled
```

Les composants

Une évolution récente du package LaravelCollective autorise la création de composants. C'est une simplification de l'approche qui consiste à utiliser ce genre de syntaxe :

```
<?php  
Form::component('button_back', 'components.form.button_back', ['nom']);
```

Cette déclaration doit être insérée dans la méthode **boot** d'un Service Provider (pour que la façade **Form** soit active).

Le premier paramètre est le nom de l'élément.

Le second paramètre est destiné à localiser le fichier contenant le template, ici il sera donc localisé en **resources/views/components/form/button_back**.

Le dernier paramètre est un tableau qui permet de transmettre toutes les valeurs pour renseigner la vue. On peut définir des valeurs par défaut.

On peut donc facilement créer le template à partir de tout ça, par exemple :

```
<a href="javascript:history.back()" class="btn btn-primary">  
  <span class="glyphicon glyphicon-circle-arrow-left"></span> {{ $nom }}  
</a>
```

Ensuite on peut directement l'utiliser dans une vue :

```
{{ Form::button_back('Retour') }}
```

Ce qui devrait se traduire au final par ce code :

```
<a href="javascript:history.back()" class="btn btn-primary">  
  <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
```


C'est une approche plus légère que celle proposée dans ce chapitre mais qui présente moins de souplesse et de possibilités. Vous avez donc le choix selon vos besoins.

En résumé

- Le conteneur de dépendances est la clé du fonctionnement de Laravel.
- On enregistre un composant avec un service provider.
- On simplifie la syntaxe d'accès à une classe avec une façade.



Des vues propres (1/2)



La localisation



L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

OpenClassrooms

Qui sommes-nous ?

Fonctionnement de nos cours

Recrutement

Nous contacter

En plus

Créer un cours

CourseLab

Conditions Générales d'Utilisation

□ Professionnels

Affiliation

Entreprises

Universités et écoles

□ Suivez-nous

Le blog OpenClassrooms



English

Español

[S'inscrire](#)[Se connecter](#)

Accueil ▶ Cours ▶ Découvrez le framework PHP Laravel ▶ La localisation

Découvrez le framework PHP Laravel

15 heures Moyenne



La localisation

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Lorsqu'on crée un site c'est souvent dans une optique multi-langages. On parle alors d'internationalisation (`i18n`) et de localisation (`L10n`). L'internationalisation consiste à préparer une application pour la rendre potentiellement adaptée à différents langages. La localisation consiste quant à elle à ajouter un composant spécifique à une langue.

C'est un sujet assez complexe qui ne se limite pas à la traduction des textes mais qui impacte aussi la représentation des dates, la gestion des pluriels, parfois la mise en page...

Qu'a à nous proposer Laravel dans ce domaine ? Nous allons le voir en rendant notre petit blog disponible en français et en anglais.



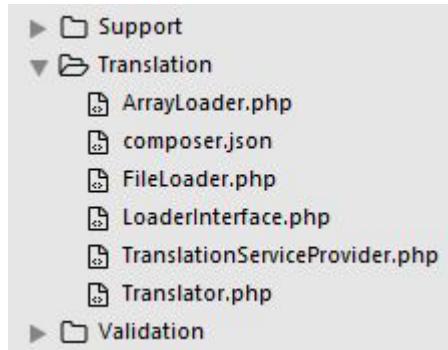
i18n parce qu'il y a 18 lettres entre le "i" et le "n" de internationalisation et **L10n** parce qu'il y a 10 lettres entre le "L" et le "n" de Localisation.

Le principe

La façade Lang et la classe Translator

Laravel est équipé de la façade `Lang` pour générer des chaînes de caractères dans plusieurs langues. Mais maintenant vous savez qu'une façade cache une classe. Laquelle dans notre cas ?

Toutes les classes qui sont concernées par la traduction se trouvent dans le dossier `Illuminate\Translation` :

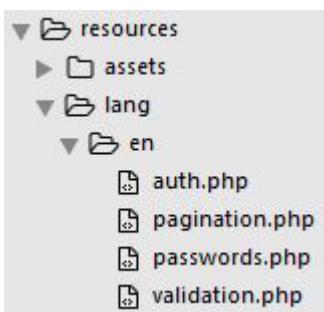


Le dossier pour les traductions

On y trouve très logiquement un provider pour faire toutes les initialisations. Un fichier `composer.json` pour les dépendances. Une classe `FileLoader` avec son interface `LoaderInterface` pour gérer les fichiers de traduction. Une classe `ArrayLoader` pour charger les messages. Et enfin la classe principale `Translator` qui est mise en action par la façade.

Les fichiers de langage

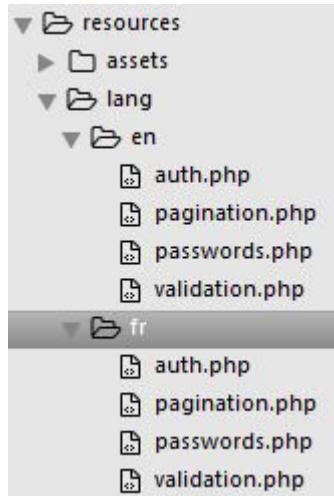
On a déjà parlé des fichiers de langage. Lorsque Laravel est installé on a cette architecture :



Les dossiers de langage

Il n'est prévu au départ que la langue anglaise (`en`) avec 4 fichiers. Pour ajouter une langue il suffit de créer un nouveau dossier, par exemple `fr` pour le Français. Et il faut évidemment avoir le même contenu comme nous allons le voir.

Vous n'allez pas être obligé de faire toute cette traduction vous-même ! Certains s'en sont déjà chargés avec [ce package](#). Nous l'avons déjà utilisé dans ce cours. Téléchargez le et copiez le dossier du français ici si vous ne l'avez pas encore ou si vous l'avez supprimé :



Le dossier du français

Vous disposez alors de deux versions linguistiques des messages de base de Laravel.

Voyons maintenant comment sont constitués ces fichiers. Prenons le plus léger (`pagination.php`) :

```
<?php  
  
return [  
  
    'previous' => '&laquo; Previous',  
    'next'      => 'Next &raquo;',  
  
];
```

On voit qu'on se contente de renvoyer un tableau avec des clés et des valeurs. On ne pourrait pas faire plus simple ! Si on prend le même fichier en version française :

```
<?php  
  
return [  
  
    'previous' => '&laquo; Précédent',  
    'next'      => 'Suivant &raquo;',  
  
];
```

On retrouve évidemment les mêmes clés avec des valeurs adaptées au langage.

Le fonctionnement

On dispose de quelques méthodes pour tester et récupérer ces valeurs.

Avec :

```
<?php  
Lang::get('pagination.previous');
```

Je vais obtenir :

```
« Previous
```

Donc la version anglaise. Comment faire pour obtenir la version française ? Regardez dans le fichier `config/app.php` :

```
<?php  
'locale' => 'en',
```

C'est ici qu'est fixé le langage en cours. Changez la valeur :

```
<?php  
'locale' => 'fr',
```

Maintenant avec le même code vous allez obtenir :

« Précédent

Evidemment on va pouvoir changer cette valeur dans le code avec la méthode `setLocale` :

```
<?php  
App::setLocale('fr');
```

Comme vous aurez de multiples endroits dans le code où vous allez récupérer des valeurs il existe un helper :

```
<?php  
trans('pagination.previous');
```

On peut aussi vérifier qu'une clé existe avec la méthode `has` :

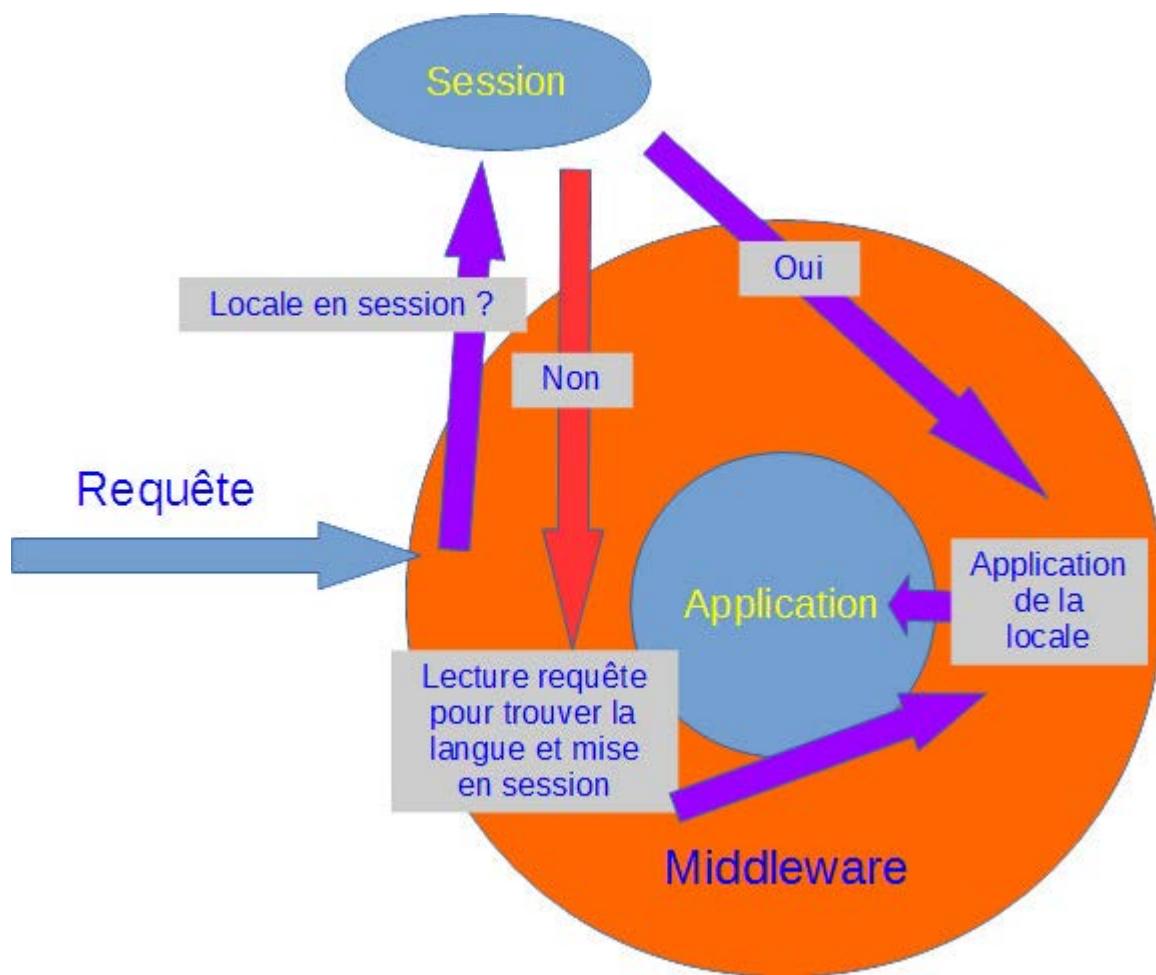
```
<?php  
Lang::has('pagination.previous');
```

Ce sont les méthodes fondamentales qui seront suffisantes pour le présent chapitre, vous pouvez trouver des compléments d'information [dans la documentation](#).

Le middleware

Quand un utilisateur choisit une langue il faut s'en rappeler, on va donc utiliser la session pour mémoriser son choix. Lorsqu'un utilisateur débarque sans avoir encore choisi il faut lui présenter une langue, mais laquelle ? Il est possible d'aller chercher l'information de la langue utilisée au niveau de la requête.

Maintenant la question est : où allons nous placer le code pour gérer tout ça ? C'est un middleware qui va logiquement s'en occuper. Voici un schéma :



Gestion de la locale

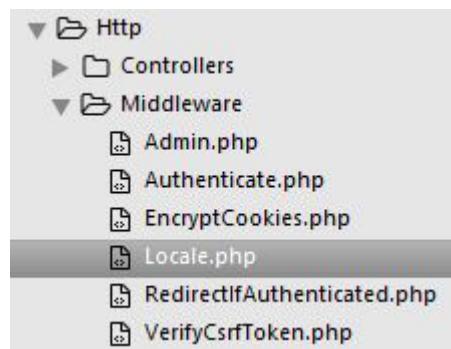
Lorsque la requête arrive elle est prise en charge par le middleware. Là on va regarder si l'utilisateur a une session active avec une localisation, si c'est le cas on applique cette locale et on envoie la requête à l'étape suivante. Si ce n'est pas le cas on regarde dans le header de la requête la langue envoyée par le navigateur, on la mémorise dans la session et on l'applique. On envoie alors la requête à t'étape suivante.

Nous allons créer ce middleware avec Artisan :

```
php artisan make:middleware Locale
```

text

On le trouve dans le dossier des middlewares :



Le nouveau middleware

Je l'ai nommé `Locale` pour représenter sa fonction. Voici le code généré par défaut :

```
<?php
```

php

```
namespace App\Http\Middleware;

use Closure;

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

On va donc ajouter notre code :

```
<?php

namespace App\Http\Middleware;

use Closure;

class Locale
{
    protected $languages = ['en','fr'];

    public function handle($request, Closure $next)
    {
        if(!session()->has('locale'))
        {
            session()->put('locale', $request->getPreferredLanguage($this->languages));
        }

        app()->setLocale(session('locale'));

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

Le code reprend strictement le schéma vu ci dessus.

Il ne nous reste plus qu'à dire à Laravel de prendre en compte ce middleware dans `app/Http/Kernel.php` :

```
<?php
protected $middlewareGroups = [
    'web' => [
        ...
        \App\Http\Middleware\Locale::class,
    ],
    ...
];
```

Les dates

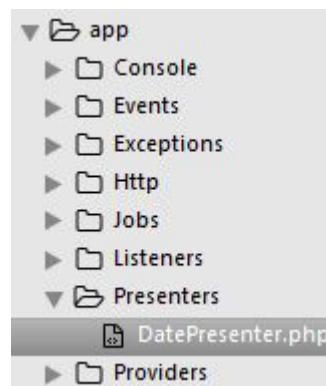
Il ne vous a sans doute pas échappé que les dates ne sont pas présentées de la même manière selon les langues utilisées. En particulier entre le français et l'anglais. En France nous utilisons le

format `jj/mm/aaaa` alors que les Américains utilisent le format `mm/jj/aaaa`. Il va donc falloir faire quelque chose pour que les dates de création des articles apparaissent au bon format dans chaque cas.

Comme ce formatage doit être appliqué pour toutes les dates la façon la plus simple et élégante de procéder est de créer un "accessor" sur la propriété. Ainsi chaque fois qu'on va extraire une date à partir du modèle on va le formater au passage. On pourrait faire ceci directement dans notre modèle `Post`. Mais pour généraliser la chose on va imaginer qu'on peut avoir besoin de la même chose pour d'autres modèles.

Dans ce genre de situation la création d'un trait est pertinente pour introduire une fonctionnalité sans passer par l'héritage. On va utiliser une version simplifiée du Design pattern Décorateur (decorator). Dans Laravel on va appeler cela un "Model Presenter".

On va créer un dossier pour les presenters et créer celui dont nous avons besoin :



Le presenter pour les dates

Avec ce code :

```
<?php
namespace App\Presenters;

use Carbon\Carbon;

trait DatePresenter
{
    public function getCreatedAtAttribute($date)
    {
        return $this->getDateFormated($date);
    }

    public function getUpdatedAtAttribute($date)
    {
        return $this->getDateFormated($date);
    }

    private function getDateFormated($date)
    {
        return Carbon::parse($date)->format(config('app.locale') == 'fr' ? 'd/m/Y' : 'm/d/Y');
    }
}
```

On voit que selon la valeur de la locale on va appliquer le formatage adapté de la date extraite.

Il ne nous reste plus qu'à utiliser ce trait dans le modèle `Post` :

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use App\Presenters\DatePresenter;

class Post extends Model
{
    use DatePresenter;

    protected $fillable = ['titre', 'contenu', 'user_id'];

    public function user()
    {
        return $this->belongsTo('App\User');
    }

    public function tags()
    {
        return $this->belongsToMany('App\Tag');
    }
}
```

Route et contrôleur

L'utilisateur doit pouvoir changer la langue s'il le désire. On va donc ajouter une route :

```
<?php
Route::get('language', 'PostController@language');
```

Et voici le contrôleur modifié :

```
<?php

namespace App\Http\Controllers;

use App\Repositories\PostRepository;
use App\Repositories\TagRepository;
use App\Http\Requests\PostRequest;

class PostController extends Controller
{

    protected $postRepository;

    protected $nbrPerPage = 4;

    public function __construct(PostRepository $postRepository)
    {
        $this->middleware('auth', ['except' => ['index', 'indexTag', 'language']]);
        $this->middleware('admin', ['only' => 'destroy']);

        $this->postRepository = $postRepository;
    }

    public function index()
    {
        $posts = $this->postRepository->getWithUserAndTagsPaginate($this->nbrPerPage);
        $links = $posts->render();
    }
}
```

```

        return view('posts.liste', compact('posts', 'links'));
    }

    public function create()
    {
        return view('posts.add');
    }

    public function store(PostRequest $request, TagRepository $tagRepository)
    {
        $inputs = array_merge($request->all(), ['user_id' => $request->user()->id]);

        $post = $this->postRepository->store($inputs);

        if(isset($inputs['tags']))
        {
            $tagRepository->store($post, $inputs['tags']);
        }

        return redirect(route('post.index'));
    }

    public function destroy($id)
    {
        $this->postRepository->destroy($id);

        return redirect()->back();
    }

    public function indexTag($tag)
    {
        $posts = $this->postRepository->getWithUserAndTagsForTagPaginate($tag, $this->nbrPerPage);
        $links = $posts->render();

        return view('posts.liste', compact('posts', 'links'))
            ->with('info', trans('blog.search') . $tag);
    }

    public function language()
    {
        session()->set('locale', session('locale') == 'fr' ? 'en' : 'fr');

        return redirect()->back();
    }
}

```

On voit une nouvelle fonction `language`. D'autre part la fonction `indexTag` a été modifiée pour tenir compte des langues.

Donc avec l'url

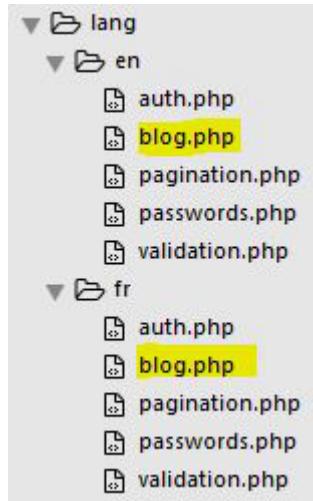
`.../language`

php

On va pouvoir passer d'une langue à l'autre.

La localisation

Il nous faut aussi créer les fichiers de localisation :



Les fichiers de localisation

Avec ce contenu pour `resources/lang/en/blog.php` :

```
<?php
return [
    'site' => 'My nice site',
    'title' => 'My nice blog',
    'creation' => 'Create an article',
    'login' => 'Login',
    'logout' => 'Logout',
    'delete' => 'Delete this post',
    'confirm' => 'Really delete this post ?',
    'on' => 'on',
    'search' => 'Results for tag : '
];
```

Et celui-ci pour `resources/lang/fr/blog.php` :

```
<?php
return [
    'site' => 'Mon joli site',
    'title' => 'Mon joli blog',
    'creation' => 'Créer un article',
    'login' => 'Se connecter',
    'logout' => 'Déconnexion',
    'delete' => 'Supprimer cet article',
    'confirm' => 'Vraiment supprimer cet article ?',
    'on' => 'le',
    'search' => 'Résultats pour la recherche du mot-clé : '
];
```

Les vues

Il ne nous reste plus qu'à modifier les vues pour que ça fonctionne. D'abord le template (`views/template.php`) :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>{{ trans('blog.site') }}</title>
        {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css') !!}

```

```
{!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css') !!}
<!--[if lt IE 9]>
    {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.js') }}
    {{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js') }}
<![endif]-->
<style> textarea { resize: none; } </style>
</head>
<body>
    <header class="jumbotron">
        <div class="container">
            <h1 class="page-header">{!! link_to_route('post.index', trans('blog.title')) !!}
</h1>
            @yield('header')
        </div>
    </header>
    <div class="container">
        @yield('contenu')
    </div>
</body>
</html>
```

Il est modifié pour la traduction du titre du site dans le header :

```
<title>{{ trans('blog.site') }}</title>
```

Et le titre/lien sur la page :

```
<h1 class="page-header">{!! link_to_route('post.index', trans('blog.title')) !!}</h1>
```

Et voici le blog transformé (`views/posts/liste.blade.php`) :

```
@extends('template')

@section('header')
    <div class="btn-group pull-right">
        {!! link_to('language', session('locale') == 'fr' ? 'English' : 'Français', ['class' => 'btn btn-primary']) !!}
        @if(Auth::check())
            {!! link_to_route('post.create', trans('blog.creation'), [], ['class' => 'btn btn-info']) !!}
            {!! link_to('logout', trans('blog.logout'), ['class' => 'btn btn-warning']) !!}
        @else
            {!! link_to('login', trans('blog.login'), ['class' => 'btn btn-info pull-right']) !!}
        @endif
    </div>
@endsection

@section('contenu')
    @if(isset($info))
        <div class="row alert alert-info">{{ $info }}</div>
    @endif
    {!! $links !!}
    @foreach($posts as $post)
        <article class="row bg-primary">
            <div class="col-md-12">
                <header>
                    <h1>{{ $post->titre }}</h1>
                    <div class="pull-right">
                        @foreach($post->tags as $tag)
                            {!! link_to('post/tag/' . $tag->tag_url, $tag->tag, ['class' => 'btn btn-xs btn-info']) !!}
                        @endforeach
                    </div>
                </header>
            </div>
        </article>
    @endforeach
@endforeach
```

```

        </header>
        <hr>
        <section>
            <p>{{ $post->contenu }}</p>
            @if(Auth::check() and Auth::user()->admin)
                {!! Form::open(['method' => 'DELETE', 'route' => ['post.destroy',
$post->id]]) !!}
                    {!! Form::submit(trans('blog.delete'), ['class' => 'btn
btn-danger btn-xs '], 'onclick' => 'return confirm(\'' . trans('blog.confirm') . '\')')) !!}
                {!! Form::close() !!}
            @endif
            <em class="pull-right">
                <span class="glyphicon glyphicon-pencil"></span> {{ $post->user-
>name . ' ' . trans('blog.on') . ' ' . $post->created_at }}
            </em>
        </section>
    </div>
</article>
<br>
@endforeach
{!! $links !!}
@endsection

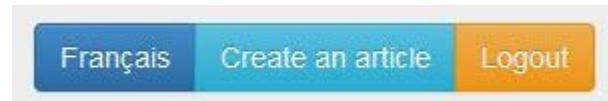
```

Les modifications portent sur tous les textes. Je vous laisse analyser le code. Le résultat est qu'en français rien ne change par rapport à ce que nous avions. Par contre quand on passe en anglais en cliquant sur le bouton "English" :



Le bouton de changement de langue

Déjà les boutons changent d'aspect :



Les boutons en anglais

Le titre de la page est en anglais :



Le titre de la page en anglais

Le titre/lien est en anglais :



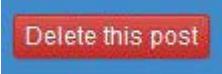
Le titre en anglais

Les dates sont formatées correctement :

 Nom on 11-26-2015

Date formatée

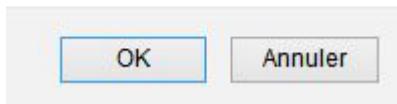
Le bouton de suppression est aussi en anglais :



Le bouton de suppression

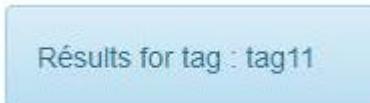
De même que le message :

Really delete this post ?



Le message d'alerte en anglais

Le message de la recherche par tag est aussi adapté :



Le message de la recherche par tag

 Il est parfois nécessaire de vider le dossier `storage/framework/views` pour purger le cache des vues.
Toutefois ne supprimez pas le fichier `.gitignore`.

Noms des contrôles

Il faudrait aussi traiter la vue de création d'un article mais c'est exactement le même principe. Il y a toutefois un élément à prendre en compte, c'est le nom des contrôles de saisie. Vous vous rappelez qu'on les a appelés `titre`, `contenu` et `tags`. Ces noms ne sont pas visibles tant qu'il n'y a pas de souci de validation, ils sont alors transmis dans le texte de l'erreur. On voit mal arriver en langue anglaise "The titre field is required.". Alors comment faire ?

Si vous regardez dans le fichier `resources/lang/fr/validation.php` vous trouvez ce tableau :

```
<?php  
'attributes' => [  
    "name" => "Nom",  
    "username" => "Pseudo",  
    ...  
,]
```

php

Ici on fait correspondre le nom d'un attribut avec sa version linguistique pour justement avoir quelque chose de

cohérent au niveau du message d'erreur. Donc dans notre situation, étant donné que nous avons francisé le nom des contrôles, il faut ajouter la version anglaise dans le fichier `resources/lang/en/validation.php` :

```
<?php
'attributes' => [
    "titre" => "Title",
    "contenu" => "Content",
],
```

php

Evidemment si on avait prévu des noms anglais, ce que l'on fait en général, il aurait fallu faire l'inverse, et prévoir des appellations françaises dans le fichier `resources/lang/fr/validation.php`.

 Le but de ce chapitre est de montrer comment localiser l'interface d'une page web. S'il s'agit d'adapter le contenu selon la langue c'est une autre histoire et il faut alors intervenir au niveau de l'url. En effet un principe fondamental du web veut que pour une certaine url on renvoie toujours le même contenu.

En résumé

- Laravel possède les outils de base pour la localisation.
- Il faut créer autant de fichiers de localisation qu'on a de langues à traiter.
- La localisation doit s'effectuer au niveau d'un middleware.
- Le formatage des dates peut s'effectuer facilement avec un "presenter".
- Il faut prévoir la version linguistique des attributs.



Des vues propres (2/2)



Ajax

L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

OpenClassrooms



Qui sommes-nous ?

Fonctionnement de nos cours

Recrutement

Nous contacter

Découvrez le framework PHP Laravel



15 heures

Moyenne

Licence

-



Ajax

[Connectez-vous](#) ou [inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Ajax est une technologie Javascript fort répandue qui permet d'envoyer des requêtes au serveur et de recevoir des réponses sans rechargement de la page. Il est par ce moyen possible de modifier dynamiquement le DOM, donc une partie de la page.

Dans ce chapitre nous allons voir comment mettre en œuvre Ajax avec Laravel. Nous allons partir d'une nouvelle installation de Laravel et mettre en place l'authentification prévue avec Artisan. Nous allons ensuite la modifier pour utiliser Ajax pour l'enregistrement d'un utilisateur. Pour ajouter de l'intérêt à l'exemple nous allons placer le formulaire dans une page modale.

Donc faites une nouvelle installation puis utilisez la commande Artisan :

```
php artisan make:auth
```

Créez aussi une base de données et effectuez la migration.

Vérifiez que tout fonctionne correctement. Ensuite on va un peu changer le code...

Les vues

Le template

Dans le template par défaut (`resources/views/layouts/app.blade.php`) on va apporter deux modifications. Voici le code résultant :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">

  <title>Laravel</title>

  <!-- Fonts -->
  <link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.4.0/css/font-awesome.min.css" rel='stylesheet' type='text/css'>
  <link href="https://fonts.googleapis.com/css?family=Lato:100,300,400,700" rel='stylesheet' type='text/css'>

  <!-- Styles -->
  {{-- <link href="{{ elixir('css/app.css') }}" rel="stylesheet"> --}}
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css" rel="stylesheet">

<style>
```

html

Plus loin avec Laravel

1. Déploiement
2. Des vues propres (1/2)
3. Des vues propres (2/2)
4. La localisation

► 5. Ajax

5. Ajax
 6. Les tests unitaires
 7. Événements et autorisations
- Quiz : Quiz 3
 Activité : Perfectionnez votre site de sondages

[Accéder au forum](#)



```

body {
    font-family: 'Lato';
}

.fa-btn {
    margin-right: 6px;
}

```

```
</style>
```

```
</head>
```

```
<body id="app-layout">
```

```
<nav class="navbar navbar-default">
```

```
<div class="container">
```

```
<div class="navbar-header">
```

```
<!-- Collapsed Hamburger -->
```

```
<button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#spark-navbar-collapse">
```

```
    <span class="sr-only">Toggle Navigation</span>
```

```
    <span class="icon-bar"></span>
```

```
    <span class="icon-bar"></span>
```

```
    <span class="icon-bar"></span>
```

```
</button>
```

```
<!-- Branding Image -->
```

```
<a class="navbar-brand" href="{{ url('/') }}>
```

```
    Laravel
```

```
</a>
```

```
</div>
```

```
<div class="collapse navbar-collapse" id="spark-navbar-collapse">
```

```
<!-- Left Side Of Navbar -->
```

```
<ul class="nav navbar-nav">
```

```
    <li><a href="{{ url('/') }}>Home</a></li>
```

```
</ul>
```

```
<!-- Right Side Of Navbar -->
```

```
<ul class="nav navbar-nav navbar-right">
```

```
<!-- Authentication Links -->
```

```
@if (Auth::guest())
```

```
    <li><a href="{{ url('/login') }}>Login</a></li>
```

```
@else
```

```
    <li class="dropdown">
```

```
        <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button"
```

```
aria-expanded="false">
```

```
            {{ Auth::user()->name }} <span class="caret"></span>
```

```
</a>
```

```
        <ul class="dropdown-menu" role="menu">
```

```
            <li><a href="{{ url('/logout') }}><i class="fa fa-btn fa-sign-out">
```

```
</i>Logout</a></li>
```

```
        </ul>
```

```
    </li>
```

```
@endif
```

```
</ul>
```

```
</div>
```

```
</div>
```

```
</nav>
```

```
@yield('content')
```

```
<!-- JavaScripts -->
```

```
{--><script src="{{ elixir('js/app.js') }}></script> -->}}
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
```

```
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"></script>
```

```
@yield('scripts')
```

```
</body>
```

```
</html>
```

Les deux modifications sont :

- la suppression du code pour l'option "Register" dans le menu, on ne garde plus que "Login" :

```

@if (Auth::guest())
    <li><a href="{{ url('/login') }}>Login</a></li>
@else

```

html

- une nouvelle section pour insérer un script :

```
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"></script>
@yield('scripts')
```

La barre de menu va donc avoir cet aspect lorsque personne n'est authentifié :

La barre de menu sans "Register"

La vue login

C'est cette vue (`resources/views/auth/login.blade.php`

) qui va subir le plus de changements. Voici le code de cette vue :

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

@section('content')



You are now registered, you can login.



Login



{!! csrf_field() !!}



<label class="col-md-4 control-label">E-Mail Address</label>

    <div class="col-md-6">
        <input type="email" class="form-control" name="email" value="{{ old('email') }}>

        @if ($errors->has('email'))
            <span class="help-block">
                <strong>{{ $errors->first('email') }}</strong>
            </span>
        @endif
    </div>



<label class="col-md-4 control-label">Password</label>

    <div class="col-md-6">
        <input type="password" class="form-control" name="password">

        @if ($errors->has('password'))
            <span class="help-block">
                <strong>{{ $errors->first('password') }}</strong>
            </span>
        @endif
    </div>



<div class="col-md-6 col-md-offset-4">


```

html

```

        <div class="checkbox">
            <label>
                <input type="checkbox" name="remember" /> Remember Me
            </label>
        </div>
    </div>

    <div class="form-group">
        <div class="col-md-6 col-md-offset-4">
            <button type="submit" class="btn btn-primary">
                <i class="fa fa-btn fa-sign-in"></i>Login
            </button>
        </div>
    <a class="btn btn-link" id="register" href="#">Register</a>
    <a class="btn btn-link" href="{{ url('/password/reset') }}>Forgot Your
Password?</a>

        </div>
    </div>
</form>
</div>
</div>
</div>

<!-- Modal --&gt;
&lt;div class="modal fade" id="myModal" tabindex="-1" role="dialog" aria-labelledby="myModalLabel" aria-
hidden="true"&gt;
    &lt;div class="modal-dialog"&gt;
        &lt;div class="modal-content"&gt;
            &lt;div class="modal-header"&gt;
                &lt;button type="button" class="close" data-dismiss="modal" aria-label="Close"&gt;&lt;span aria-
hidden="true"&gt;×&lt;/span&gt;&lt;/button&gt;
                &lt;h4 class="modal-title" id="myModalLabel"&gt;Register&lt;/h4&gt;
            &lt;/div&gt;
            &lt;div class="modal-body"&gt;

                &lt;form id="formRegister" class="form-horizontal" role="form" method="POST" action="{{
url('/register') }}"&gt;
                    {{!! csrf_field() !!}}


                    &lt;div class="form-group"&gt;
                        &lt;label class="col-md-4 control-label"&gt;Name&lt;/label&gt;
                        &lt;div class="col-md-6"&gt;
                            &lt;input type="text" class="form-control" name="name"&gt;
                            &lt;small class="help-block"&gt;&lt;/small&gt;
                        &lt;/div&gt;
                    &lt;/div&gt;

                    &lt;div class="form-group"&gt;
                        &lt;label class="col-md-4 control-label"&gt;E-Mail Address&lt;/label&gt;
                        &lt;div class="col-md-6"&gt;
                            &lt;input type="email" class="form-control" name="email"&gt;
                            &lt;small class="help-block"&gt;&lt;/small&gt;
                        &lt;/div&gt;
                    &lt;/div&gt;

                    &lt;div class="form-group"&gt;
                        &lt;label class="col-md-4 control-label"&gt;Password&lt;/label&gt;
                        &lt;div class="col-md-6"&gt;
                            &lt;input type="password" class="form-control" name="password"&gt;
                            &lt;small class="help-block"&gt;&lt;/small&gt;
                        &lt;/div&gt;
                    &lt;/div&gt;

                    &lt;div class="form-group"&gt;
                        &lt;label class="col-md-4 control-label"&gt;Confirm Password&lt;/label&gt;
                        &lt;div class="col-md-6"&gt;
                            &lt;input type="password" class="form-control" name="password_confirmation"&gt;
                        &lt;/div&gt;
                    &lt;/div&gt;

                    &lt;div class="form-group"&gt;
                        &lt;div class="col-md-6 col-md-offset-4"&gt;
                            &lt;button type="submit" class="btn btn-primary"&gt;
                                Register
                            &lt;/button&gt;
                        &lt;/div&gt;
                    &lt;/div&gt;
                &lt;/form&gt;
            &lt;/div&gt;
        &lt;/div&gt;
    &lt;/div&gt;
&lt;/div&gt;
</pre>

```

```

        </div>
    </div>
</form>

</div>
</div>
</div>
</div>

@endsection

@section('scripts')

<script>

$(function(){

    $('#register').click(function() {
        $('#myModal').modal();
    });

    $(document).on('submit', '#formRegister', function(e) {
        e.preventDefault();

        $('input+small').text('');
        $('input').parent().removeClass('has-error');

        $.ajax({
            method: $(this).attr('method'),
            url: $(this).attr('action'),
            data: $(this).serialize(),
            dataType: "json"
        })
        .done(function(data) {
            $('.alert-success').removeClass('hidden');
            $('#myModal').modal('hide');
        })
        .fail(function(data) {
            $.each(data.responseJSON, function (key, value) {
                var input = '#formRegister input[name=' + key + ']';
                $(input + '+small').text(value);
                $(input).parent().addClass('has-error');
            });
        });
    });
});

})>

</script>

@endsection

```

Voyons les changements :

- ajout d'une barre d'information cachée par défaut (hidden) :

```

<div class="alert alert-success alert-dismissible hidden">
    You are now registered, you can login.
</div>

```

Elle aura cet aspect :

You are now registered, you can login.

La barre d'information

- ajout d'un lien pour l'enregistrement :

```

<a class="btn btn-link" id="register" href="#">Register</a>
<a class="btn btn-link" href="{{ url('/password/reset') }}>Forgot Your Password?</a>

```

On a donc un nouveau lien pour s'enregistrer dans ce formulaire :

Login

E-Mail Address

Password

Remember Me

Register [Forgot Your Password?](#)

Le lien pour s'enregistrer

- ajout du code de la page modale qui héberge le formulaire :

```
html
<div class="modal fade" id="myModal" tabindex="-1" role="dialog" aria-labelledby="myModalLabel" aria-hidden="true">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <button type="button" class="close" data-dismiss="modal" aria-label="Close"><span aria-hidden="true">&amptimes</span></button>
                <h4 class="modal-title" id="myModalLabel">Register</h4>
            </div>
            <div class="modal-body">

                <form id="formRegister" class="form-horizontal" role="form" method="POST" action="{{ url('/register') }}>
                    {{ csrf_field() }}

                    <div class="form-group">
                        <label class="col-md-4 control-label">Name</label>
                        <div class="col-md-6">
                            <input type="text" class="form-control" name="name">
                            <small class="help-block"></small>
                        </div>
                    </div>

                    <div class="form-group">
                        <label class="col-md-4 control-label">E-Mail Address</label>
                        <div class="col-md-6">
                            <input type="email" class="form-control" name="email">
                            <small class="help-block"></small>
                        </div>
                    </div>

                    <div class="form-group">
                        <label class="col-md-4 control-label">Password</label>
                        <div class="col-md-6">
                            <input type="password" class="form-control" name="password">
                            <small class="help-block"></small>
                        </div>
                    </div>

                    <div class="form-group">
                        <label class="col-md-4 control-label">Confirm Password</label>
                        <div class="col-md-6">
                            <input type="password" class="form-control" name="password_confirmation">
                        </div>
                    </div>

                    <div class="form-group">
                        <div class="col-md-6 col-md-offset-4">
                            <button type="submit" class="btn btn-primary">
                                Register
                            </button>
                        </div>
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>
```

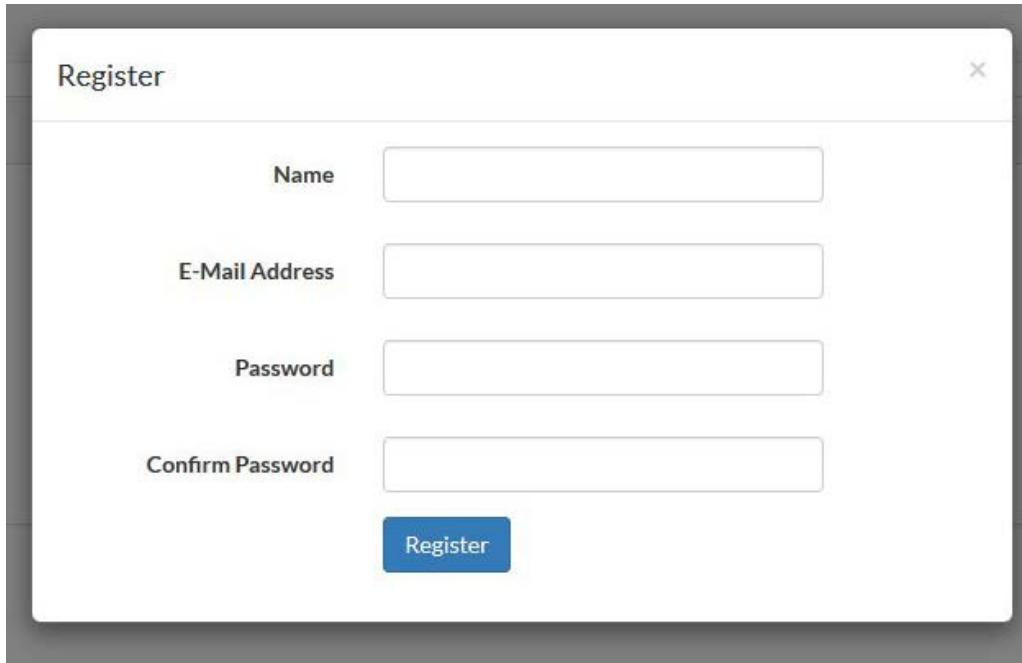
</div>

C'est du codage classique avec Bootstrap. Remarquez pour chaque contrôle du formulaire la partie qui va servir pour l'affichage éventuel des erreurs de validation :

<small class="help-block"></small>

html

On aura cet aspect pour cette fenêtre modale :



La fenêtre modale d'enregistrement

- ajout du code JavaScript pour la gestion du formulaire d'enregistrement :

javascript

```
$(function(){
    $('#register').click(function() {
        $('#myModal').modal();
    });

    $(document).on('submit', '#formRegister', function(e) {
        e.preventDefault();

        $('input+small').text('');
        $('input').parent().removeClass('has-error');

        $.ajax({
            method: $(this).attr('method'),
            url: $(this).attr('action'),
            data: $(this).serialize(),
            dataType: "json"
        })
        .done(function(data) {
            $('.alert-success').removeClass('hidden');
            $('#myModal').modal('hide');
        })
        .fail(function(data) {
            $.each(data.responseJSON, function (key, value) {
                var input = '#formRegister input[name=' + key + ']';
                $(input + '+small').text(value);
                $(input).parent().addClass('has-error');
            });
        });
    });
})
```

Le Javascript

On va regarder de plus près la partie JavaScript...

À la soumission du formulaire il faut éviter d'avoir l'envoi normal :

javascript

```
e.preventDefault();
```

Ensuite on supprime toute trace d'erreur précédente :

javascript

```
$(‘input+small’).text(‘’);
$(‘input’).parent().removeClass(‘has-error’);
```

Il faut ensuite envoyer la requête avec les valeurs saisies en se simplifiant la vie avec une sérialisation :

javascript

```
$.ajax({
    method: $(this).attr(‘method’),
    url: $(this).attr(‘action’),
    data: $(this).serialize(),
    dataType: “json”
})
```

Si la validation est correcte il faut afficher la barre d'information et cacher la fenêtre modale :

javascript

```
.done(function(data) {
    $('.alert-success').removeClass(‘hidden’);
    $('#myModal').modal(‘hide’);
})
```

En cas d'erreur dans la validation on balaye les erreurs et on met en place le style :

javascript

```
.fail(function(data) {
    $.each(data.responseJSON, function (key, value) {
        var input = ‘#formRegister input[name=’ + key + ‘]’;
        $(input + ‘+small’).text(value);
        $(input).parent().addClass(‘has-error’);
    });
});
```

Par exemple si on ne remplit pas le contrôle du nom :

Name

The name field is required.

Nom non présent

Le traitement

Le contrôleur

Voici le nouveau contrôleur `app/Http\Controllers\Auth\AuthController` :

php

```
<?php

namespace App\Http\Controllers\Auth;

use App\User;
use Validator;
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Foundation\Auth\ThrottlesLogins;
use Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers;

class AuthController extends Controller
{
    /*
    |--------------------------------------------------------------------------
    | Registration & Login Controller
    |--------------------------------------------------------------------------
    |
    | This controller handles the registration of new users, as well as the
    | authentication of existing users. By default, this controller uses
    | a simple trait to add these behaviors. Why don't you explore it?
    |
    */
}
```

```

use AuthenticatesAndRegistersUsers, ThrottlesLogins;

/**
 * Where to redirect users after login / registration.
 *
 * @var string
 */
protected $redirectTo = '/home';

/**
 * Create a new authentication controller instance.
 *
 * @return void
 */
public function __construct()
{
    $this->middleware('guest', ['except' => 'logout']);
    $this->middleware('ajax', ['only' => 'register']);
}

/**
 * Get a validator for an incoming registration request.
 *
 * @param array $data
 * @return \Illuminate\Contracts\Validation\Validator
 */
protected function validator(array $data)
{
    return Validator::make($data, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255|unique:users',
        'password' => 'required|confirmed|min:6',
    ]);
}

/**
 * Create a new user instance after a valid registration.
 *
 * @param array $data
 * @return User
 */
protected function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => bcrypt($data['password']),
    ]);
}

/**
 * Handle a registration request for the application.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function register(Request $request)
{
    $validator = $this->validator($request->all());

    if ($validator->fails()) {
        $this->throwValidationException(
            $request, $validator
        );
    }

    $this->create($request->all());

    return response()->json();
}
}

```

La méthode `register` se situe normalement dans le trait. Mais comme nous avons besoin d'en modifier le code on surcharge cette méthode dans le contrôleur. Les seules différences avec le code du trait sont :

- pas d'authentification immédiate à l'enregistrement,
- réponse JSON.

Le middleware

On peut aussi remarquer dans le contrôleur la présence d'un nouveau middleware `ajax` :

```
<?php
$this->middleware('ajax', ['only' => 'register']);
```

php

En effet on veut que notre méthode `register` ne soit accessible que si on reçoit une requête ajax. Ce middleware n'existe pas par défaut dans Laravel, il faut donc le créer

(`app/Http/Middlewares/Ajax.php`) :

```
<?php
namespace App\Http\Middleware;

use Closure;

class Ajax
{

    public function handle($request, Closure $next)
    {
        if ($request->ajax())
        {
            return $next($request);
        }

        abort(404);
    }
}
```

php

La méthode `ajax` de la requête nous permet facilement de savoir s'il s'agit d'une requête de ce type, si ce n'est pas le cas on revoie une erreur 404.

Il faut informer Laravel que notre middleware existe (`app/Http/Kernel.php`) :

```
<?php
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'ajax' => \App\Http\Middleware\Ajax::class,
];
```

php

En résumé

- Ajax est facile à mettre en œuvre avec Laravel.
- La sérialisation permet de mettre en forme les entrées du formulaire.
- On peut prévoir un middleware particulier pour les requêtes Ajax.



[La localisation](#)

[Les tests unitaires](#)



L'auteur

Maurice Chavelli

Développeur retraité qui se consacre à l'enseignement des technologies du web.

[S'inscrire](#)[Se connecter](#)[Accueil](#) ▶ [Cours](#) ▶ [Découvrez le framework PHP Laravel](#) ▶ [Les tests unitaires](#)

Découvrez le framework PHP Laravel

15 heures Moyenne



Les tests unitaires

[Connectez-vous ou inscrivez-vous](#) pour bénéficier de toutes les fonctionnalités de ce cours !

Les développeurs PHP n'ont pas été habitués à faire des tests pour leurs applications. Cela est dû à l'histoire de ce langage qui n'était au départ qu'une possibilité de scripter au milieu du code HTML mais qui s'est peu à peu développé comme un langage de plus en plus évolué. Les créateurs de frameworks ont initié une autre façon d'organiser le code de PHP, en particulier ils ont mis en avant la séparation des tâches qui a rendu la création de tests possible.

Laravel a été pensé pour intégrer des tests. Il comporte une infrastructure élémentaire et des helpers. Nous allons voir dans ce chapitre cet aspect de Laravel. Considérez ce que je vais vous dire ici comme une simple introduction à ce domaine qui mériterait à lui seul un cours spécifique. Je vais m'efforcer de vous démontrer l'utilité de créer des tests, comment les préparer et comment les isoler.

Lorsqu'on développe avec PHP on effectue forcément des tests au moins manuels. Par exemple si vous créez un formulaire vous allez l'utiliser, entrer diverses informations, essayer des fausses manœuvres... Imaginez que tout ça soit automatisé et que vous n'ayez qu'à cliquer pour lancer tous les tests. C'est le propos de ce chapitre.

Vous pouvez aussi vous dire qu'écrire des tests conduit à du travail supplémentaire, que ce n'est pas toujours facile, que ce n'est pas nécessaire dans tous les cas. C'est à vous de voir si vous avez besoin d'en créer ou pas. Pour des petites applications la question reste ouverte. Par contre dès qu'une application prend de l'ampleur ou lorsqu'elle est conduite par plusieurs personnes alors il devient vite nécessaire de créer des tests automatisés.

L'intendance des tests

PHPUnit

Laravel utilise [PHPUnit](#) pour effectuer les tests. C'est un framework créé par Sebastian Bergmann qui fonctionne à partir d'assertions.

Ce framework est installé comme dépendance de Laravel en mode développement :

```
"require-dev": {
    "fzaninotto/faker": "~1.4",
    "mockery/mockery": "0.9.*",
    "phpunit/phpunit": "~4.0",
    "symfony/css-selector": "2.8.*|3.0.*",
    "symfony/dom-crawler": "2.8.*|3.0.*"
},
```

json

Mais vous pouvez aussi utiliser le fichier `phar` [que vous pouvez trouver sur cette page](#) et le placer à la racine de votre application et vous êtes prêt à tester !



La version 5 de PHPUnit nécessite PHP 5.6

Vous pouvez vérifier que ça fonctionne en entrant cette commande :

```
php phpunit.phar -h
```

text

Vous obtenez ainsi la liste de toutes les commandes disponibles.

Si vous utilisez la version installée avec Laravel ça donne :

```
php vendor\phpunit\phpunit\phpunit -h
```

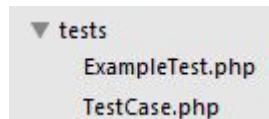
text

Je vous conseille de vous faire un alias :).

Dans tous les exemples de ce chapitre j'utiliserai le fichier `phar`.

L'intendance de Laravel

Si vous regardez les dossiers de Laravel vous allez trouver un dossier qui est consacré aux tests :



Les dossier des tests

Vous avez déjà deux fichiers. Voilà `TestCase.php` :

```
<?php

class TestCase extends Illuminate\Foundation\Testing\TestCase
{
    /**
     * The base URL to use while testing the application.
     *
     * @var string
     */
    protected $baseUrl = 'http://localhost';

    /**
     * Creates the application.
     *
     * @return \Illuminate\Foundation\Application
     */
    public function createApplication()
```

php

```
{
    $app = require __DIR__.'/../bootstrap/app.php';

    $app->make(Illuminate\Contracts\Console\Kernel::class)->bootstrap();

    return $app;
}
```

Cette classe est chargée de créer une application pour les tests dans un environnement spécifique (ce qui permet de mettre en place une configuration adaptée aux tests).

Elle étend la classe `Illuminate\Foundation\Testing\TestCase` qui elle-même étend la classe `PHPUnit_Framework_TestCase` en lui ajoutant quelques fonctionnalités bien pratiques, comme nous allons le voir.

Toutes les classes de test que vous allez créer devront étendre cette classe présent `ExampleTest.php` :

`TestCase`. Il y a un exemple de test déjà

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
            ->see('Laravel 5');
    }
}
```

php

Sans entrer pour le moment dans le code sachez simplement qu'on envoie une requête pour la route de base et qu'on attend une réponse. Pour lancer ce test c'est très simple, entrez cette commande :

```
php phpunit.phar
PHPUnit 4.8.21 by Sebastian Bergmann and contributors.

.

Time: 3290ms, Memory: 21.50Mb

OK (1 test, 2 assertions)
```

On voit qu'ont été effectués un test et 2 assertions et que tout s'est bien passé.

L'environnement de test

Je vous ai dit que les tests s'effectuent dans un environnement particulier, ce qui est bien pratique. Où se trouve cette configuration ? Regardez le fichier `phpunit.xml` :

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit backupGlobals="false"
       backupStaticAttributes="false"
```

xml

```

    bootstrap="bootstrap/autoload.php"
    colors="true"
    convertErrorsToExceptions="true"
    convertNoticesToExceptions="true"
    convertWarningsToExceptions="true"
    processIsolation="false"
    stopOnFailure="false">
<testsuites>
    <testsuite name="Application Test Suite">
        <directory suffix="Test.php">./tests</directory>
    </testsuite>
</testsuites>
<filter>
    <whitelist processUncoveredFilesFromWhitelist="true">
        <directory suffix=".php">./app</directory>
        <exclude>
            <file>./app/Http/routes.php</file>
        </exclude>
    </whitelist>
</filter>
<php>
    <env name="APP_ENV" value="testing"></env>
    <env name="CACHE_DRIVER" value="array"></env>
    <env name="SESSION_DRIVER" value="array"></env>
    <env name="QUEUE_DRIVER" value="sync"></env>
</php>
</phpunit>

```

On trouve déjà 4 variables d'environnement :

- APP_ENV : là on dit qu'on est en mode testing,
- CACHE_DRIVER : en mode "array" ce qui signifie qu'on ne va rien mettre en cache pendant les tests (par défaut on a **file**),
- SESSION_DRIVER : en mode "array" ce qui signifie qu'on ne va pas faire persister la session (par défaut on a **file**),
- QUEUE_DRIVER : je n'évoque pas les tâches dans ce cours, on ne va donc pas tenir compte de cette variable.

On peut évidemment ajouter les variables dont on a besoin. Par exemple si pendant les tests je ne veux plus **MySql** mais **sqlite**. Il y a une variable dans **config/database.php** :

```

<?php
'default' => env('DB_CONNECTION', 'mysql'),

```

Il faut donc la renseigner dans **.env** :

```

DB_CONNECTION=mysql

```

Du coup dans **phpunit.xml** je peux écrire :

```

<env name="DB_CONNECTION" value="sqlite"/>

```

Maintenant pour les tests je vais utiliser **sqlite**.

Construire un test

Les trois étapes d'un test

Pour construire un test on procède généralement en trois étapes :

1. on initialise les données,
2. on agit sur ces données,
3. on vérifie que le résultat est conforme à notre attente.

Comme tout ça est un peu abstrait prenons un exemple. Remplacez le code de la méthode `testBasicExample` par celui-ci :

```
<?php
public function testBasicExample()
{
    $data = [10, 20, 30];
    $result = array_sum($data);
    $this->assertEquals(60, $result);
}
```

On trouve nos trois étapes. On initialise les données :

```
<?php
$data = [10, 20, 30];
```

On agit sur ces données :

```
<?php
$result = array_sum($data);
```

On teste le résultat :

```
<?php
$this->assertEquals(60, $result);
```

La méthode `assertEquals` permet de comparer deux valeurs, ici 60 et `$result`. Si vous lancez le test vous obtenez :

```
OK (1 test, 1 assertion)
```

Vous voyez à nouveau l'exécution d'un test et d'une assertion. Le tout s'est bien passé. Changez la valeur 60 par une autre et vous obtiendrez ceci :

```
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Vous connaissez maintenant le principe de base d'un test et ce qu'on peut obtenir comme renseignement en cas d'échec.

Assertions et appel de routes

Les assertions

Les assertions constituent l'outil de base des tests. On en a vu une ci-dessus et il en existe bien d'autres. Vous pouvez en trouver [la liste complète ici](#).

Voici l'utilisation de quelques assertions et l'utilisation d'un helper de Laravel que l'on teste au passage :

```
<?php
public function testBasicExample()
{
```

```
$data = 'Je suis petit';
$this->assertTrue(starts_with($data, 'Je'));
$this->assertFalse(starts_with($data, 'Tu'));
$this->assertSame(starts_with($data, 'Tu'), false);
$this->assertStringStartsWith('Je', $data);
$this->assertStringEndsWith('petit', $data);
}
```

Lorsqu'on lance le test on obtient ici :

```
OK (1 test, 5 assertions)
```

Un test et 5 assertions correctes.

Appel de routes et test de réponse

Il est facile d'appeler une route pour effectuer un test sur la réponse. Modifiez la route de base pour celle-ci :

```
<?php
Route::get('/', function()
{
    return 'coucou';
});
```

On a donc une requête avec l'url de base et comme réponse la chaîne "coucou". Nous allons tester que la requête aboutit bien, qu'il y a une réponse correcte et que la réponse est "coucou" :

```
<?php
public function testBasicExample()
{
    $response = $this->call('GET', '/');
    $this->assertResponseOk();
    $this->assertEquals('coucou', $response->getContent());
}
```

L'assertion `assertResponseOk` nous assure que la réponse est correcte. Ce n'est pas une assertion de PHPUnit mais une spécifique de Laravel. La méthode `getContent` permet de lire la réponse. On obtient :

```
OK (1 test, 2 assertions)
```

Les vues et les contrôleurs

Les vues

Qu'en est-il si on retourne une vue ? Mettez ce code pour la route :

```
<?php
Route::get('/', function()
{
    return view('welcome')->with('message', 'Vous y êtes !');
});
```

Ajoutez dans cette vue ceci :

```
{{ $message }}
```

Maintenant voici le test :

```
<?php
public function testBasicExample()
{
```

```

$response = $this->call('GET', '/');
$view = $response->original;
$this->assertEquals('Vous y êtes !', $view['message']);
}

```

On envoie la requête et on récupère la réponse. Ensuite la méthode `original` nous permet de récupérer la vue. On peut alors tester la valeur de la variable `$message` dans la vue.

Laravel propose une autre assertion pour effectuer la même chose :

```

<?php
public function testBasicExample()
{
    $response = $this->call('GET', '/');
    $this->assertViewHas('message');
    $this->assertViewHas('message', 'Vous y êtes !');
}

```

L'assertion `assertViewHas` permet de vérifier qu'une vue reçoit bien une donnée, on peut aussi grâce au second paramètre vérifier sa valeur.

Les contrôleurs

Créez ce contrôleur :

```

<?php
namespace App\Http\Controllers;

class WelcomeController extends Controller
{
    public function index()
    {
        return view('welcome');
    }
}

```

Vous pouvez tester facilement les actions d'un contrôleur. Créez cette route pour mettre en oeuvre le contrôleur ci-dessus :

```

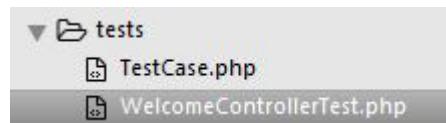
<?php
Route::get('welcome', 'WelcomeController@index');

```

Vérifiez que ça fonctionne (vous aurez peut-être besoin de retoucher la vue où nous avons introduit une variable).

Supprimez le fichier `ExampleTest.php` qui ne va plus nous servir.

Créez cette architecture de dossiers et le fichier `WelcomeControllerTest.php` :



Le dossier des tests

Et mettez ce code dans le fichier `WelcomeControllerTest.php` :

php

```
<?php

class WelcomeControllerTest extends TestCase
{

    public function testIndex()
    {
        $response = $this->action('GET', 'WelcomeController@index');
        $this->assertResponseOk();
    }

}
```

Il y a une commande d'artisan pour créer une classe de test :

```
php artisan make:test WelcomeControllerTest
```

Si vous lancez le test ça devrait bien se passer :

```
OK (1 test, 1 assertion)
```

text

Donc pour appeler une action d'un contrôleur il suffit d'utiliser la méthode `action` et de préciser le verbe, le nom du contrôleur et le nom de la méthode associée. D'autre part il est important de bien organiser les tests pour s'y retrouver. Une bonne façon de procéder est d'adopter la même architecture de dossiers que celle prévue pour l'application.

Isoler les tests

Nous allons maintenant aborder un aspect important des tests qui ne s'appellent pas unitaires pour rien. Pour faire des tests efficaces il faut bien les isoler, donc savoir ce qu'on teste, ne tester qu'une chose à la fois et ne pas mélanger les choses. Ceci est possible si le code est bien organisé, ce que je me suis efforcé de vous montrer depuis le début de ce cours.

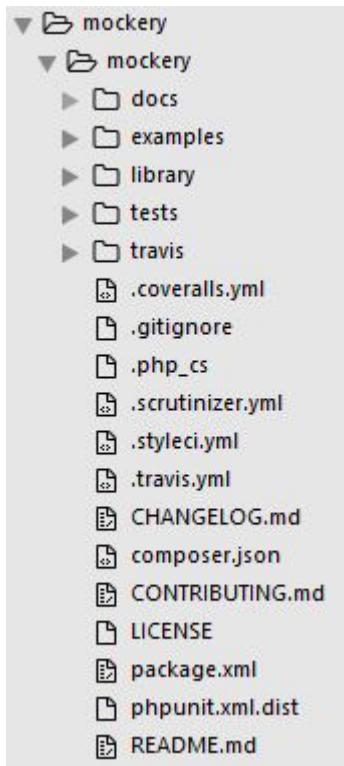
Avec PHPUnit chaque test est effectué dans une application spécifique, il n'est donc pas possible de les rendre dépendants les uns des autres.

En général on utilise [Mockery](#), un composant qui permet de simuler le comportement d'une classe. Il est déjà prévu dans l'installation de Laravel en mode développement :

```
"require-dev": {
    ...
    "mockery/mockery": "0.9.*"
},
```

json

Le fait de prévoir ce composant uniquement pour le développement simplifie ensuite la mise en œuvre pour le déploiement. Normalement vous devriez trouver ce composant dans vos dossiers :



Le composant Mockery

Simuler une classe

Nous allons voir maintenant comment l'utiliser mais pour cela on va mettre en place le code à tester. Ce ne sera pas trop réaliste mais c'est juste pour comprendre le mécanisme de fonctionnement de Mockery. Remplacez le code du contrôleur `WelcomeController` par celui-ci :

```

<?php
namespace App\Http\Controllers;

use Livre;

class WelcomeController extends Controller
{
    public function __construct()
    {
        $this->middleware('guest');
    }

    public function index(Livre $livre)
    {
        $livres = $livre->all();
        return view('welcome', compact('livres'));
    }
}

```

J'ai prévu l'injection d'un modèle dans la fonction `index`. Ce n'est pas forcément très judicieux d'un point de vue conceptuel et il vaudrait mieux passer par un repository comme nous l'avons vu plusieurs fois dans ce cours mais on va prendre cette situation simple pour exécuter un test. D'ailleurs la classe `Livre` n'a pas besoin de vraiment exister puisqu'on ne va pas vraiment l'utiliser.

La difficulté ici réside dans la présence de l'injection d'une classe. Comme on veut isoler les tests l'idéal serait de pouvoir

simuler cette classe. C'est justement ce que permet de faire Mockery.

Voici la classe de test que nous allons utiliser :

```
<?php

use Illuminate\Database\Eloquent\Collection;

class WelcomeControllerTest extends TestCase
{

    public function testIndex()
    {
        // Création de la collection
        $collection = new Collection;
        $i = 2;
        while($i--) {
            $collection->add((object) ['titre' => 'Titre' . $i]);
        }

        // Création Mock
        $mock = Mockery::mock('Livre');
        $mock ->shouldReceive('all')
            ->once()
            ->andReturn($collection);

        // Cr éation lien
        $this->app->instance('Livre', $mock);

        // Action
        $response = $this->call('GET', 'welcome');

        // Assertions
        $this->assertResponseOk();
        $this->assertViewHas('livres');

    }

    public function tearDown()
    {
        Mockery::close();
    }
}
```

Et voici le code à ajouter dans la vue pour faire réaliste :

```
@foreach ($livres as $livre)
    <p>Titre : {{ $livre->titre }}</p>
@endforeach
```

Si je lance le test j'obtiens :

```
OK (1 test, 2 assertions)
```

Un test et deux assertions correctes. Voyons de plus près ce code.

Au départ on crée une collection de livres :

```
<?php
// Cr éation de la collection
$collection = new Collection;
$i = 2;
```

```
while($i--) {
    $collection->add((object) ['titre' => 'Titre' . $i]);
}
```

On crée ensuite un objet Mock en lui demandant de simuler la classe `Livre` :

```
<?php
$mock = Mockery::mock('Livre');
```

Ensuite on définit le comportement que l'on désire pour cet objet :

```
<?php
$mock ->shouldReceive('all')
    ->once()
    ->andReturn($collection);
```

On lui dit que s'il reçoit l'appel de la méthode "all" (`shouldReceive`) une fois (`once`) il doit retourner la collection `$collection` (`andReturn`).

Ensuite on informe le conteneur de Laravel de la liaison entre la classe `Livre` et notre objet Mock :

```
<?php
$this->app->instance('Livre', $mock);
```

C'est une façon de dire à Laravel : chaque fois que tu auras besoin de la classe `Livre` tu iras plutôt utiliser l'objet Mock.

Ensuite on fait l'action, ici la requête :

```
<?php
$response = $this->call('GET', 'welcome');
```

Pour finir on prévoit deux assertions, une pour vérifier qu'on a une réponse correcte et la seconde pour vérifier qu'on a bien les livres dans la vue :

```
<?php
$this->assertResponseOk();
$this->assertViewHas('livres');
```

Pour finir on prévoit pour travailler proprement de supprimer l'objet Mock à l'issue du test dans la méthode `tearDown` qui sert justement à effectuer des actions après un test :

```
<?php
public function tearDown()
{
    Mockery::close();
}
```

Vous connaissez maintenant le principe de l'utilisation de Mockery. Il existe de vastes possibilités avec ce composant.

Il n'y a pas vraiment de règle quant à la constitution des tests, quant à ce qu'il faut tester ou pas. L'important est de comprendre comment les faire et de juger ce qui est utile ou pas selon les circonstances. Une façon efficace d'apprendre à réaliser des tests tout en comprenant mieux Laravel est de [regarder comment ces tests ont été conçus](#).

Tester une application

Laravel va encore plus loin dans la convivialité pour les tests en offrant la possibilité de tester facilement une

application.

Partez d'un Laravel tout neuf et ajoutez l'authentification avec la commande :

```
php artisan make:auth
```

Prévoyez une base de données pour que ça fonctionne et entrez l'utilisateur Toto, avec l'email "toto@chez.fr" et le mot de passe "password". On va maintenant créer un test pour voir si cet utilisateur peut bien s'authentifier.

Commencez par supprimer le fichier `ExampleTest.php`.

Créez un nouveau fichier `AuthControllerTest.php` avec Artisan :

```
php artisan make:test AuthControllerTest
```

Complétez ainsi le code :

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class AuthControllerTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testRegister()
    {
        $this->visit('/')
            ->click('Login')
            ->type('toto@chez.fr', 'email')
            ->type('password', 'password')
            ->press('Login')
            ->see('Dashboard');
    }
}
```

php

Vous voyez que ça se lit comme de la prose ! Si vous lancez le test vous obtiendrez :

```
OK(2 tests, 5 assertions)
```

Je ne vais pas détailler ici toutes les possibilités, reportez vous à [la documentation](#).

En résumé

- Laravel utilise PHPUnit pour effectuer les test unitaires.
- En plus des méthodes de PHPUnit on dispose d'helpers pour intégrer les tests dans une application réalisée avec Laravel.
- Le composant Mockery permet de simuler le comportement d'une classe et donc de bien isoler les tests.
- Laravel propose des commandes conviviales pour tester une application.



Ajax



Événements et autorisations