Cours Laravel 5.3 – les données – les ressources (2/2) et les erreurs

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 :

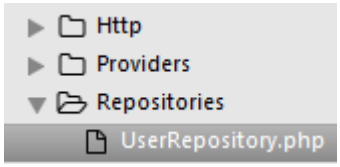
```
public function __construct(UserRepository $userRepository)
{
    $this->userRepository = $userRepository;
}
```

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;
    }

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

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

    public function update(User $user, Array $inputs)
    {
        $user->update($inputs);
    }

    public function destroy(User $user)
    {
        $user->delete();
    }
}
```

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



Le repository dans notre exemple est évidemment très léger et on pourrait se questionner sur sa pertinence. Par exemple quel intérêt d'appliquer la méthode delete dans le repository alors qu'on pourrait directement le faire sur le modèle qu'on a déjà dans le contrôleur ? On verra que ça deviendra plus évident quand on augmente la complexité d'une application. Il y a déjà eu des débats animés sur les repositories et le sujet n'est pas clos. Dans ce cours je privilégie leur utilisation, ensuite libre à vous de tout coder dans les contrôleurs ou les modèles. L'important est

de trouver une organisation du code qui vous convienne et qui reste lisible si vous devez le partager.

Le fonctionnement

Nous allons à présent analyser ses différentes actions en voyant comment sont articulés le contrôleur et le repository.

index

La page d'accueil (**index**) répond à l'url **…/user** avec la méthode GET.

On arrive dans la méthode **index** du contrôleur :

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

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

Il nous faut ici les renseignements sur les utilisateurs à afficher. Comme il peut y en avoir beaucoup on va limiter l'affichage en faisant de la pagination. La propriété **nbrPerPage** contient le nombre d'utilisateurs à afficher sur chaque page. On demande au repository de renvoyer ces informations dans la variable **$users**.

Dans le repository on a ce code :

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

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

```
}
```

Dès qu'on instancie la classe on définit une propriété **user** de type **App\User**.

Dans la méthode **getPaginate** on utilise la méthode **paginate** d'Eloquent pour renvoyer les informations paginées. Encore une fois c'est Eloquent qui s'occupe de tout !

show

La page pour voir les informations d'un utilisateur (**show**) répond à l'url **…/user/id** avec la méthode GET.

On arrive dans la méthode **show** du contrôleur :

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

Etant donné qu'on a la liaison implicite on n'a même pas besoin du repository, on envoie directement les informations dans la vue.

create

La page pour voir le formulaire de création d'un utilisateur (**create**) répond à l'url **…/user/create** avec la méthode GET.

On arrive dans la méthode **create** du contrôleur :

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

Ici on a juste une vue à retourner et aucune information à transmettre.

store

Pour recevoir la soumission du formulaire de création d'un utilisateur (**store**) on a l'url **…/user** avec la méthode POST.

On arrive dans la méthode **store** du contrôleur :

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

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

On faite appel au repository avec sa méthode **store** en lui transmettant toutes les information du formulaire (**$request->all()**).

Ensuite on redirige sur la route **user.index** en flashant une information en session (*ce qui veut dire qu'elle reste mémorisée seulement pour la prochaine requête*) avec les renseignements à afficher dans la vue, en l'occurrence ce sera une alerte informative.

Dans le repository on a ce code :

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

On commence par crypter le mot de passe (**bcrypt**). et ensuite on utilise la méthode **create** d'Eloquent pour créer l'utilisateur.

*Par sécurité ce type d'assignement de masse (on transmet directement un tableau de valeurs issues du client) est limité par une propriété au niveau du modèle qui désigne précisément les noms des colonnes susceptibles d'être modifiées. Si vous regardez dans le fichier **app/User.php** vous trouvez cette propriété :*

protected $fillable = ['name', 'email', 'password',];

*Ce sont les seules colonnes qui seront impactées par la méthode **create** (et équivalentes). Attention à cela lorsque vous avez un bug mystérieux avec des colonnes qui ne se mettent pas à jour !*

edit

La page pour voir le formulaire de modification d'un utilisateur (**edit**) répond à l'url …/user/id avec la méthode GET.

On arrive dans la méthode **edit** du contrôleur :

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

Comme pour **show**, étant donné qu'on a la liaison implicite on n'a même pas besoin du repository, on envoie directement les informations dans la vue.

update

Pour gérer la soumission du formulaire de modification d'un utilisateur (**update**) on a l'url …/user/id avec la méthode PUT.

On arrive dans la méthode **update** du contrôleur :

```
public function update(UserUpdateRequest $request, User $user)
{
    $this->userRepository->update($user, $request->all());
     return redirect()->route('user.index')->withOk("L'utilisateur
" . $request->name . " a été modifié.");
}
```

On fait appel à la méthode update du repository en lui transmettant :

- une instance de **App\User** pour l'utilisateur (liaison implicite),
- toutes les données du formulaire (**$request->all()**).

Ensuite on redirige sur la page d'accueil (**index**) en flashant une information en session (ce qui veut dire qu'elle reste mémorisée seulement pour la prochaine requête) avec les renseignements à afficher dans la vue, en l'occurrence ce sera une alerte informative.

destroy

Pour surprimer un utilisateur (**destroy**) on a l'url **…/user/id** avec la méthode DELETE.

On arrive dans la méthode **destroy** du contrôleur :

```
public function destroy(User $user)
{
    $this->userRepository->destroy($user);

    return back();
}
```

On fait appel à la méthode **destroy** du repository en lui transmettant le modèle généré par la liaison implicite :

```
public function destroy(User $user)
{
    $user->delete();
}
```

C'est la méthode **delete** d'Eloquent qui supprime effectivement l'enregistrement.

Le template

Nous aurons le même template pour toutes les vues, c'est celui que nous avons déjà utilisé dans les précédents chapitres (**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>Lettre d'information</title>
                                    {!!        Html::style('<a
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstra
p.min.css">https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/boo
tstrap.min.css</a>') !!}
                                    {!!        Html::style('<a
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstra
p-
theme.min.css">https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css
/bootstrap-theme.min.css</a>') !!}
    <!--[if lt IE 9]>
                                        {{      Html::style('<a
href="https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.js">ht
tps://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.js</a>') }}
                                        {{      Html::style('<a
href="https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js"
>https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js</a>')
}}
    <![endif]-->
  </head>
  <body>
    @yield('contenu')
  </body>
</html>
```

La vue index

Cette vue (**resources/views/index.blade.php**) 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-3 col-sm-6">
        @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']) !!}
```

```
        {!! $users->links() !!}
    </div>
@endsection
```

Structures de contrôle de Blade

Nous découvrons de nouvelles possibilités de Blade avec la structure de contrôle **@if** :

```
@if(session()->has('ok'))
     <div class="alert alert-success alert-dismissible">{!! session('ok') !!}</div>
@endif
```

On affiche l'alerte que si on a une clé **ok** en session.

On trouve aussi la structure **@foreach** qui permet d'itérer dans une collection (ici les utilisateurs) :

```
@foreach ($users as $user)
    ...
@endforeach
```

Helper link_to_route

Remarquez également l'utilisation de l'helper **link_to_route** pour créer un lien :

```
<td>{!! link_to_route('user.show', 'Voir', [$user->id], ['class' => 'btn btn-success btn-block']) !!}</td>
```

On précise le nom de la route (**user.show**) , le titre (**'Voir'**), un paramètre (**$user->id**) et un attribut (classes de Bootstrap pour obtenir une apparence de bouton).

Formulaire pour la suppression

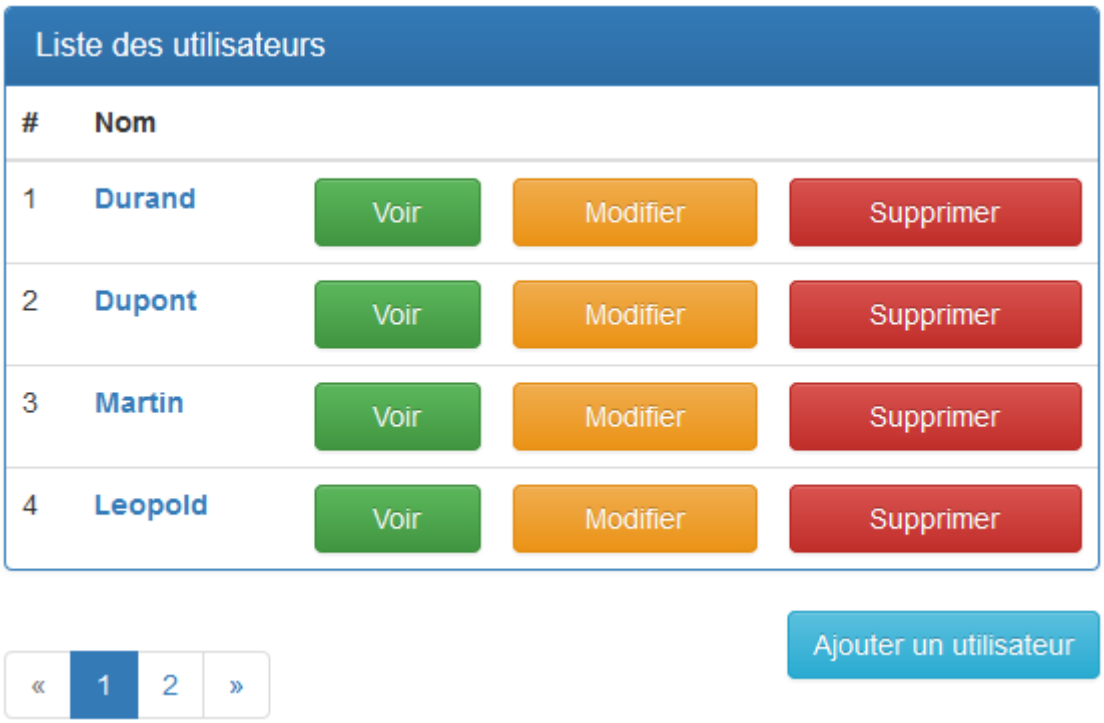
Pour supprimer un utilisateur on doit envoyer une requête avec la méthode DELETE. Les navigateurs ne savent pas faire ça alors on prévoit un formulaire :

```
{!! 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() !!}
```

Remarquez la possibilité dans l'ouverture du formulaires de désigner une route (**user.destroy**) et un paramètre (**$user->id**).

Enfin voici l'aspect de la vue :



La vue show

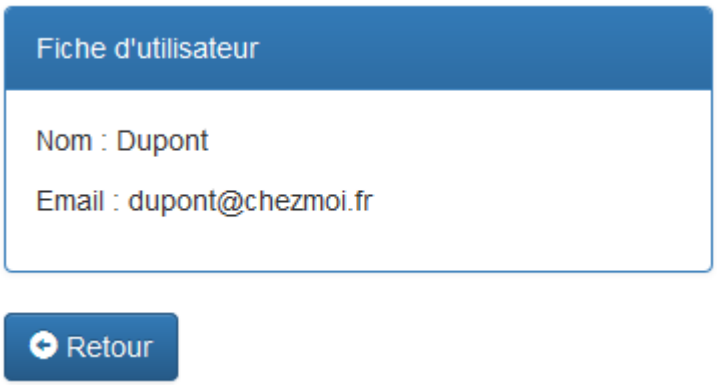
La vue **show** sert à afficher la fiche d'un utilisateur avec son nom et son adresse email :

```
@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>
            </div>
        </div>
          <a href="javascript:history.back()" class="btn btn-
primary">
              <span class="glyphicon glyphicon-circle-arrow-
left"></span> Retour
        </a>
    </div>
@endsection
```

Le code ne présente aucune particularité.

Voici son aspect :



La 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-
```

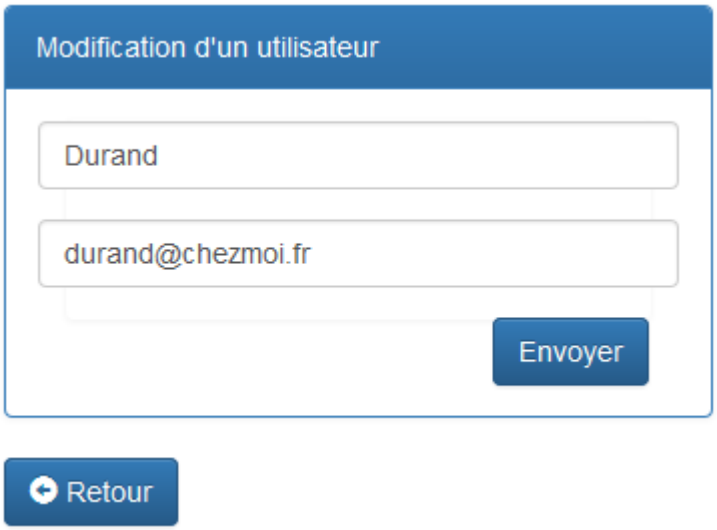
```
horizontal panel']) !!}
                                <div class="form-group {!!
$errors->has('name') ? 'has-error' : '' !!}">
                                {!! Form::text('name', null, ['class'
=> 'form-control', 'placeholder' => 'Nom']) !!}
                                 {!! $errors->first('name', '<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>
                             {!! Form::submit('Envoyer', ['class' =>
'btn btn-primary pull-right']) !!}
                        {!! Form::close() !!}
                    </div>
                </div>
            </div>
              <a href="javascript:history.back()" class="btn btn-
primary">
                   <span class="glyphicon glyphicon-circle-arrow-
left"></span> Retour
            </a>
        </div>
@endsection
```

Remarquez l’utilisation de **Form::model** :

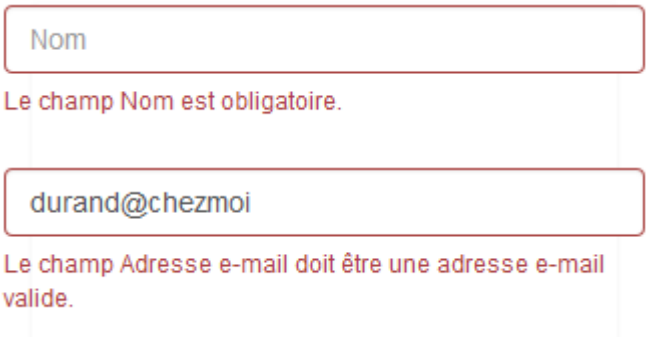
```
{!! Form::model($user, ['route' => ['user.update', $user->id],
'method' => 'put', 'class' => 'form-horizontal panel']) !!}
```

Cette méthode 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.

Voici son aspect :



En cas d'erreur dans la validation on a l'affichage correspondant :



La 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>
                                 {!! Form::submit('Envoyer', ['class' => 'btn btn-primary pull-right']) !!}
                                {!! Form::close() !!}
                            </div>
                        </div>
                    </div>
                      <a href="javascript:history.back()" class="btn btn-primary">
                            <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
                    </a>
                </div>
@endsection
```

Au niveau de la validation on a pour le mot de passe deux contrôles pour avoir une double saisie, ce qui est classique. Il

faut évidemment vérifier la concordance des deux. Dans la requête de formulaire c'est prévu ainsi :

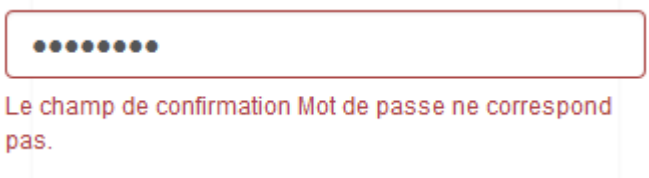
```
'password' => 'bail|required|confirmed|min:8'
```

On a mis **confirmed.**

Mais comment Laravel connait-il le nom du contrôle qui sert pour la confirmation ?

Par convention il cherche un contrôle dont le nom est celui du contrôle primaire auquel on ajoute « _confirmation ». Ici comme le contrôle s'appelle **password**, le contrôle pour la confirmation porte le nom de **password_confirmation**.

Si vous faites un essai avec deux entrées différentes vous devriez obtenir ce message :



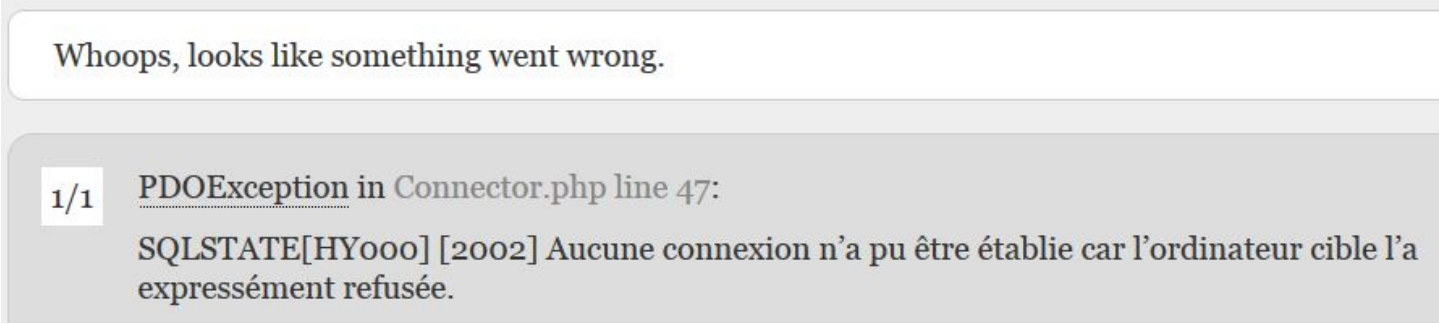
Vous devez donc avoir ces 5 vues :

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 veux ouvrir la page d'accueil :



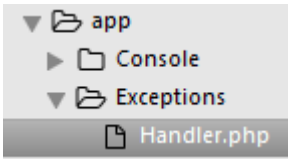
Je n'ai affiché ici que la partie supérieure.

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.

On a le détail parce qu'on est en mode de débogage (variable **APP_DEBUG** avec la valeur **true** dans le fichier **.env**). En mode production on aura seulement la partie supérieure du message qui dit laconiquement que quelque chose ne va pas.

On a déjà vu qu'on peut mettre dans le dossier **views/errors** une vue avec comme nom le code de l'erreur, on l'a déjà fait pour l'erreur 404. Ici on a un code d'erreur 500 (erreur interne du serveur), on pourrait donc créer une vue **views/errors/500.blade.php**, mais elle serait commune à toutes les erreurs 500.

Pour personnaliser la page d'erreurs de façon plus fine il faut aller un peu toucher au fichier **app/Exceptions/Handler.php** :



Voici son code :

```
<?php

namespace App\Exceptions;

use Exception;
use Illuminate\Auth\AuthenticationException;
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 = [
        \Illuminate\Auth\AuthenticationException::class,
        \Illuminate\Auth\Access\AuthorizationException::class,
\Symfony\Component\HttpKernel\Exception\HttpException::class,
\Illuminate\Database\Eloquent\ModelNotFoundException::class,
        \Illuminate\Session\TokenMismatchException::class,
        \Illuminate\Validation\ValidationException::class,
    ];

    /**
     * Report or log an exception.
     *
     * This is a great spot to send exceptions to Sentry, Bugsnag,
etc.
     *
     * @param  \Exception  $exception
     * @return void
     */
    public function report(Exception $exception)
    {
        parent::report($exception);
    }

    /**
     * Render an exception into an HTTP response.
     *
```

```
     * @param  \Illuminate\Http\Request  $request
     * @param  \Exception  $exception
     * @return \Illuminate\Http\Response
     */
    public function render($request, Exception $exception)
    {
        return parent::render($request, $exception);
    }

    /**
     * Convert an authentication exception into an unauthenticated response.
     *
     * @param  \Illuminate\Http\Request  $request
        * @param   \Illuminate\Auth\AuthenticationException $exception
     * @return \Illuminate\Http\Response
     */
         protected  function  unauthenticated($request, AuthenticationException $exception)
    {
        if ($request->expectsJson()) {
                    return  response()->json(['error'  => 'Unauthenticated.'], 401);
        }

        return redirect()->guest('login');
    }
}
```

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 le dossier **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 :

[2016-09-01 15:27:27] local.ERROR: PDOException: 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.

On a vu plus haut que lorsqu’on arrête le service MySql on génère une erreur **PDOException**.

Comment intercepter cette erreur ?

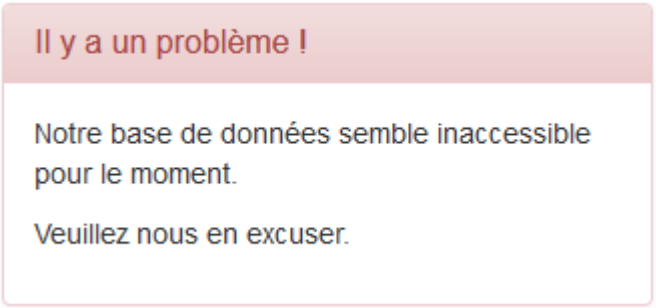
Voilà une solution :

```
public function render($request, Exception $exception)
{
    if($exception instanceof \PDOException) {
        return response()->view('errors.pdo', [], 500);
    }
    return parent::render($request, $exception);
}
```

Il suffit maintenant de créer une vue **resources/views/error/pdo.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>Notre base de données semble inaccessible pour le moment.</p>
                <p>Veuillez nous en excuser.</p>
            </div>
        </div>
    </div>
@endsection
```

Maintenant on va afficher ceci :



C’est quand même plus explicite et élégant !

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.
- 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.