Cours Laravel 5.5 – les données – jouer avec Eloquent

On a commencé à voir Eloquent et les modèles dans le premier chapitre de cette partie du cours, mais on s'est limité pour le moment à créer un enregistrement. Dans ce chapitre on va aller plus loin en explorant les possibilités d'Eloquent, couplé à un puissant générateur de requêtes, pour manipuler les données.

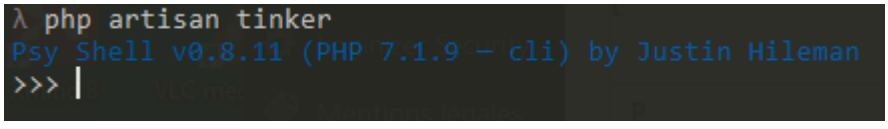
Pour effectuer les manipulations de ce chapitre vous aurez besoin d'une installation fraîche de Laravel complétée avec les éléments de l'authentification comme on l'a vu dans le chapitre correspondant. Les migrations devront également être effectuées pour qu'on puisse utiliser la base de données. Vous aurez également besoin de l'application d'exemple fonctionnelle.

Tinker

Artisan est plein de possibilités, l'une d'elle est un outil très pratique, un **REPL** (Read–Eval–Print Loop). C'est un outil qui permet d'entrer des expressions, de les faire évaluer et d'avoir le résultat à l'affichage et son nom est **Tinker**. Ce REPL est boosté par le package psysh. Il y a un bon article de présentation en anglais ici.

Pour lancer cet outil il y a une commande d'artisan :

php artisan tinker

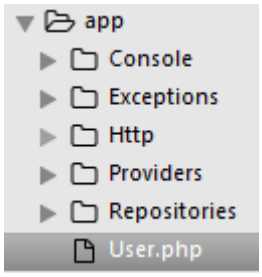


On va se servir de cet outil pour interagir avec la base de données dans ce chapitre.

Créer un enregistrement

Méthode create

On a déjà vu comment créer un enregistrement avec Eloquent. Comme a priori votre base de données est vide avec une installation fraiche de Laravel on va commencer par ajouter un utilisateur dans la table **users** à l'aide du modèle **User**. Pour mémoire ce modèle est le seul présent au départ :



Dans Tinker entrez cette expression :

```
User::create(['name' => 'Durand', 'email' => 'durand@chezlui.fr', 'password' => 'pass'])
```

```
>>> User::create(['name' => 'Durand', 'email' => 'durand@chezlui.fr', 'password' => 'pass'])
=> App\User {#759
     name: "Durand",
     email: "durand@chezlui.fr",
     updated_at: "2017-09-12 11:30:03",
     created_at: "2017-09-12 11:30:03",
     id: 1,
   }
```

*Remarquez qu'on n'a pas besoin d'entrer l'espace de nom complet **App\User**, Tinker crée automatiquement un alias.*

La méthode **create** renvoie le modèle avec tous les attributs créés comme les dates et l'id. Tinker nous les affiche gentiment. On retrouve évidemment notre enregistrement dans la table **users** :

id	name	email	password	remember_token	created_at	updated_at
1	Durand	durand@chezlui.fr	pass	*NULL*	2017-09-12 11:30:03	2017-09-12 11:30:03

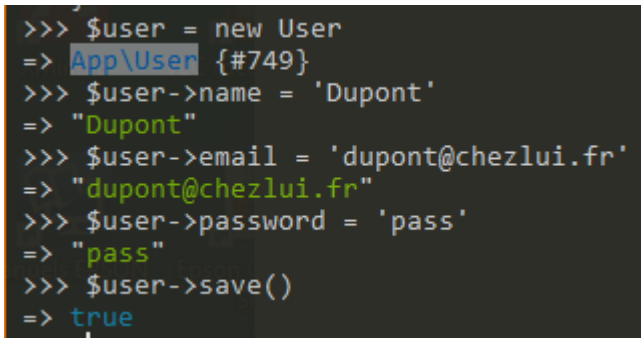
*Pour mémoire cette méthode **create** est un assignement de masse et implique les précautions que nous avons déjà vues.*

Méthode save

Une autre façon de créer un enregistrement avec Eloquent et de commencer par créer un modèle vide, renseigner les attributs, puis utiliser la méthode **save** pour enregistrer dans la table.

Entrez cette suite d'expressions :

```
$user = new User
$user->name = 'Dupont'
$user->email = 'dupont@chezlui.fr'
suser->password = 'pass'
$user->save()
```



On commence par créer le modèle, puis on renseigne ses attributs, enfin on enregistre dans la base avec **save**.

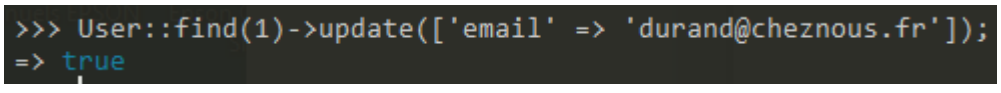
Modifier un enregistrement

Voyons maintenant comment modifier un enregistrement existant. Il y a aussi deux façons de procéder.

Méthode update

Entrez cette expression dans Tinker :

```
User::find(1)->update(['email' => 'durand@cheznous.fr'])
```



On commence par récupérer le premier enregistrement de la table avec **find(1)**, ça a pour effet de créer un modèle avec les

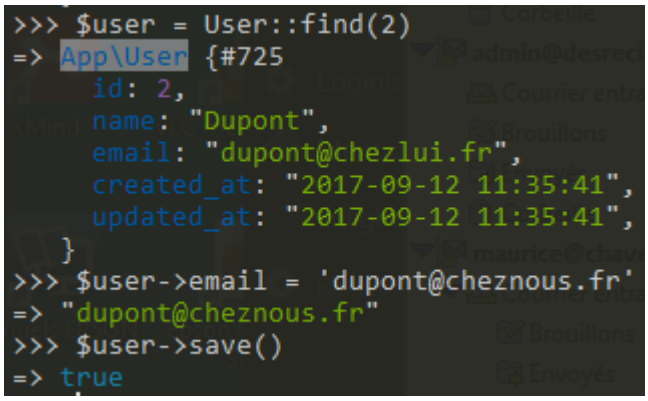
attributs renseignés pour Durand. Ensuite on utilise la méthode **update** sur ce modèle en précisant le nom et la valeur de chaque attribut qu'on veut modifier dans un tableau. ici on a changé l'adresse email pour Durand.

Avec cette méthode update on a encore le problème de l'assignement de masse.

Méthode save

On peut aussi utiliser la méthode **save** pour faire une modification. Entrez ces expressions dans Tinker :

```
$user = User::find(2)
$user->email = 'dupont@cheznous.fr'
$user->save()
```



Avec **find(2)** on crée un modèle avec les renseignements de Dupont (le deuxième enregistrement de la table), ensuite on change l'attribut **email**, enfin on enregistre dans la base.

On peut constater les modifications dans la table :

id	name	email	password	remember_token	created_at	updated_at
1	Durand	durand@cheznous.fr	pass	*NULL*	2017-09-12 11:30:03	2017-09-12 11:42:19
2	Dupont	dupont@cheznous.fr	pass	*NULL*	2017-09-12 11:35:41	2017-09-12 11:50:47

*Remarquez que les colonnes **created_at** et **updated_at** n'ont maintenant plus les mêmes valeurs puisqu'on a fait des modifications.*

Supprimer un enregistrement

Après la création et la modification vient tout naturellement la suppression avec la méthode **delete**. Si vous entrez cette expression dans Tinker :

User::find(2)->delete()

Vous supprimez l'utilisateur qui a l'index 2.

La suppression est définitive. Il existe aussi une possibilité de suppression provisoire (**soft delete**) mais il faut modifier le modèle en ajoutant un trait et une propriété. Cette partie de la documentation traite de ce sujet que je ne pense pas développer dans ce cours.

Trouver des enregistrements

Rafraichissez la table (pas dans Tinker cette fois) :

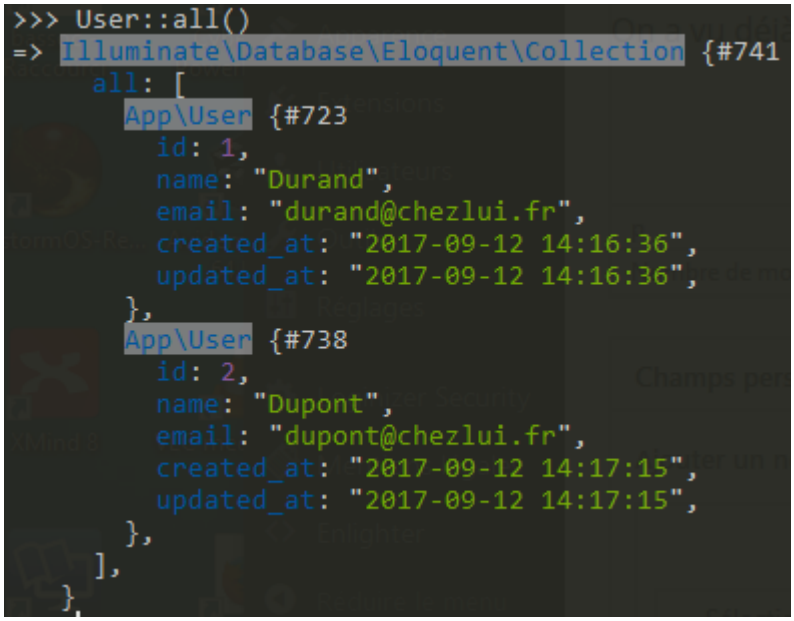
php artisan migrate:refresh

Et recréez avec Tinker les deux utilisateurs :

```
User::create(['name' => 'Durand', 'email' => 'durand@chezlui.fr', 'password' => 'pass'])
User::create(['name' => 'Dupont', 'email' => 'dupont@chezlui.fr', 'password' => 'pass'])
```

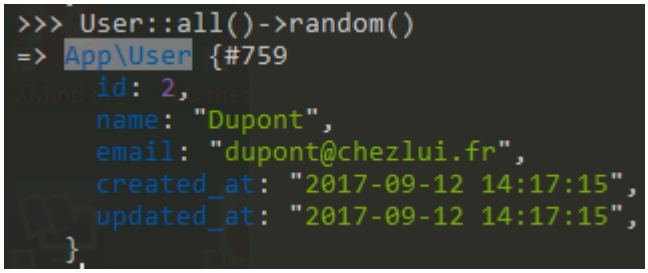
Tous les enregistrements

On a vu déjà la méthode **find** pour trouver un enregistrement à partir de son identifiant (on peut aussi passer plusieurs identifiants dans un tableau). On peut aussi tous les récupérer avec la méthode **all** :



Si vous êtes observateur vous avez remarqué que cette fois on n'obtient pas un modèle, comme c'était le cas avec **find** mais une collection (**Illuminate\Database\Eloquent\Collection**). Cette classe comporte un nombre considérable de méthodes, on peut donc effectuer de nombreux traitement à ce niveau et même les chaîner.

Par exemple ici j'extrais un modèle au hasard de la collection :

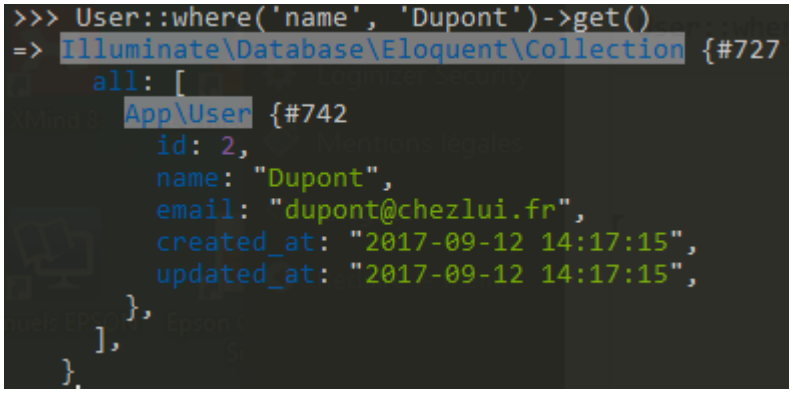


J'aurai l'occasion dans ce cours de montrer l'utilisation de plusieurs des méthodes disponibles.

Sélectionner des enregistrements

En général on ne veut pas tous les enregistrements mais justes certains qui correspondent à certains critères. Eloquent est couplé à un puissant générateur de requêtes SQL (**Query Builder**). On peut par exemple utiliser la méthode **where**. Entrez cette expression dans Tinker :

```
User::where('name', 'Dupont')->get()
```



On récupère une collection qui ne contient que le modèle qui correspond à Dupont.

Ajoutez cet utilisateur :

User::create(['name' => 'Martin', 'email' => 'martin@chezlui.fr', 'password' => 'pass'])

On a maintenant 3 utilisateurs :

name	email	password
Durand	durand@chezlui.fr	pass
Dupont	dupont@chezlui.fr	pass
Martin	martin@chezlui.fr	pass

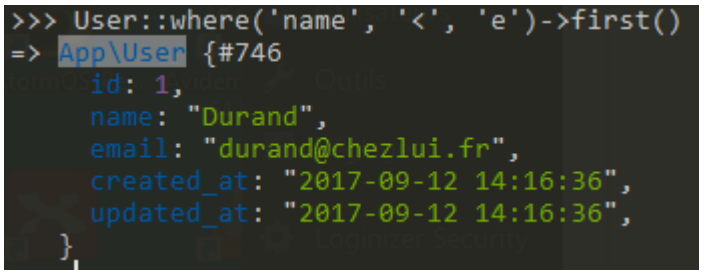
Maintenant entrez cette expression :

User::where('name', '<', 'e')->get()

Cette fois on a sélectionné à partir des noms dont la première

lettre doit être avant « e » dans l'alphabet et on obtient une collection avec deux modèles.

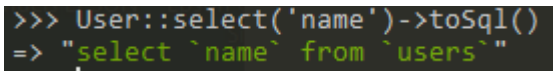
On dispose de la méthode **first** si on veut récupérer juste le premier :



On peut aussi limiter les colonnes retournées avec la méthode **select** :

Ici on ne retourne que les noms.

Souvent on a envie de connaître la requête générée, pour la voir il suffit d'utiliser la méthode **toSql** :



On dispose avec le générateur de requête de riches possibilités que nous verrons progressivement, par exemple on peut facilement trier les enregistrements avec la méthode **orderBy** :

```
>>> User::select('name')->orderBy('name', 'desc')->get()
=> Illuminate\Database\Eloquent\Collection {#746
     all: [
       App\User {#741
         name: "Martin",
       },
       App\User {#745
         name: "Durand",
       },
       App\User {#750
         name: "Dupont",
       },
     ],
   }
```

Souvent il existe un syntaxe plus simple, par exemple on peut obtenir le même résultat avec **latest** :

```
User::select('name')->latest('name')->get()
```

Trouver ou créer

Que se passe-t-il si l'enregistrement qu'on veut ne se trouve pas dans la table ? Faisons un essai :

```
>>> User::find(4)
=> null
```

On a **null** en retour et on peut gérer cette valeur. Mais des fois on se trouve dans une situation où on veut récupérer un enregistrement ou le créer s'il n'existe pas. On peut le faire en deux étapes mais on peut aussi le faire en une seule action avec la méthode **firstOrCreate** :

```
>>> User::firstOrCreate(['name' => 'Martin', 'email' => 'martin@chezlui.fr', 'password' => 'pass'])
=> App\User {#758
     id: 3,
     name: "Martin",
     email: "martin@chezlui.fr",
     created_at: "2017-09-12 14:35:14",
     updated_at: "2017-09-12 14:35:14",
   }
```

Ici comme l'enregistrement existe il est retourné normalement mais dans le cas suivant il n'existe pas et donc il est créé :

```
>>> User::firstOrCreate(['name' => 'Lunel', 'email' => 'lunel@chezlui.fr', 'password' => 'pass'])
=> App\User {#742
     name: "Lunel",
     email: "lunel@chezlui.fr",
     updated_at: "2017-09-12 15:15:12",
     created_at: "2017-09-12 15:15:12",
     id: 4,
   },
```

On se retrouve donc avec 4 enregistrements dans la table :

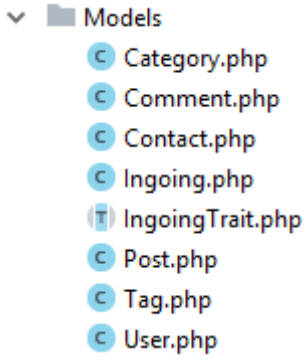
id	name	email	password
1	Durand	durand@chezlui.fr	pass
2	Dupont	dupont@chezlui.fr	pass
3	Martin	martin@chezlui.fr	pass
4	Lunel	lunel@chezlui.fr	pass

Il existe aussi la méthode **firstOrNew**, si l'enregistrement existe il est retourné comme avec **firstOrCreate**, par contre s'il n'existe pas il est juste créé une instance du modèle mais aucune action dans la base n'est réalisée.

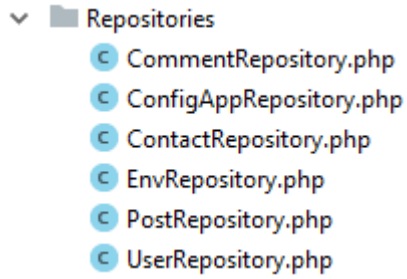
De la même manière il existe la méthode **updateOrCreate** basée sur le même principe.

L'application d'exemple

Dans l'application d'exemple il y a 7 modèles et un trait :



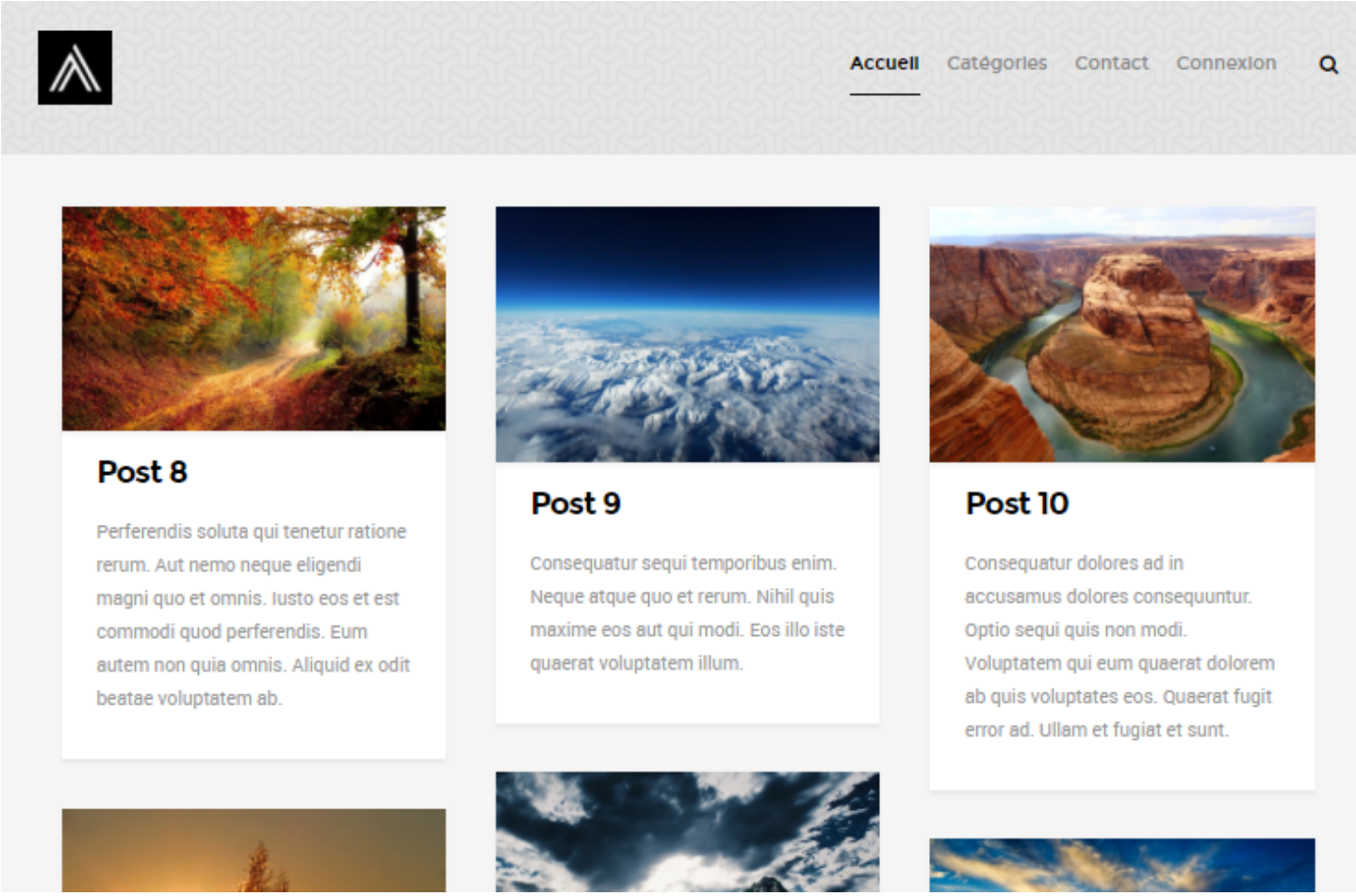
La plupart des opérations sur la base sont déléguées à des repositories :



Ces modèles et repositories sont intensivement utilisés et nous allons voir quelques illustrations concrètes des éléments théoriques vus ci-dessus.

Les articles sur la page d'accueil

La page d'accueil présentent un résumé des articles avec une image et un texte d'introduction :



Il faut aller extraire ces informations de la base. D'autre part on a aussi une pagination pour limiter le nombre d'articles visible sur une page mais je développerai ce point particulier dans un chapitre spécifique parce que c'est un sujet assez long.

Dans le fichier **PostRepository** vous trouvez ce code :

```
public function __construct(Post $post, Tag $tag, Comment $comment)
{
    $this->model = $post;
    ...
}


protected function queryActiveOrderByDate()
{
    return $this->model
        ->select('id', 'title', 'slug', 'excerpt', 'image')
        ->whereActive(true)
        ->latest();
}


public function getActiveOrderByDate($nbrPages)
{
    return $this->queryActiveOrderByDate()->paginate($nbrPages);
}
```

Dans la fonction **queryActiveOrderByDate** vous avez des choses qu'on a vues ci-dessus :

- **select** : pour ne sélectionner que les colonnes qui nous intéressent, en effet on ne veut pas toutes les informations des articles pour cette page d'accueil,
- **whereActive** : on ne sélectionne que les articles « actifs », c'est la colonne **active** qui nous l'indique,
- **latest** : on trie les article du plus récent au plus ancien (équivalent à **orderBy('created_at', desc)**)

Ensuite dans la fonction **getActiveOrderByDate** on se contente d'appliquer la pagination.

Au niveau du contrôleur **PostController** le code est classique :

```
public function index()
{
                                        $posts          =
```

```
$this->postRepository->getActiveOrderByDate($this->nbrPages);

    return view('front.index', compact('posts'));
}
```

On retourne la vue **front.index** (qui donc se trouve en **resources/views/front/index.blade.php**) en lui transmettant le résultat de la requête, c'est à dire un tableau qui contient une collection de tous les enregistrements de articles.

Au niveau de la vue l'instruction **@foreach** de Blade permet d'itérer dans les collections pour toutes les récupérer :

```
@foreach($posts as $post)
    @include('front.brick-standard', compact('$post'))
@endforeach
```

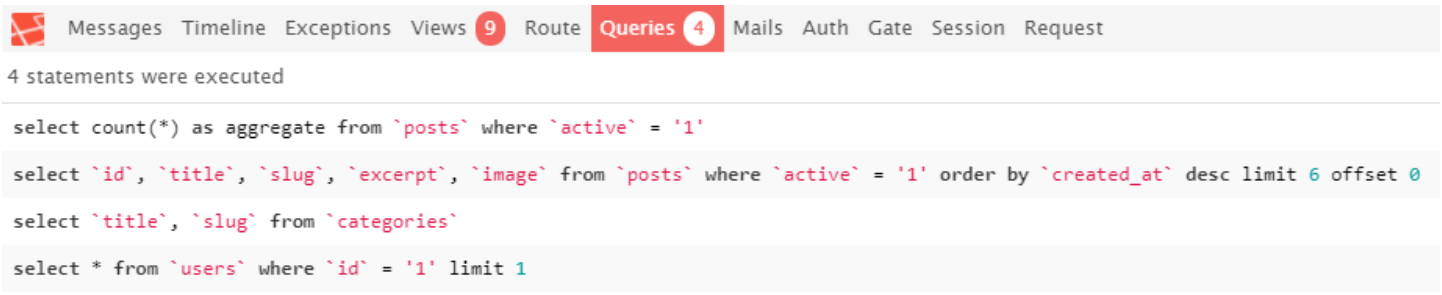
Avec Blade on peut inclure une vue dans une autre vue avec l'instruction **@include**. C'est ce qui est réalisé ici avec l'inclusion de la vue **front.brick-standard** (qui donc se trouve en **resources/views/front/brick-standard.blade.php**) :

```
<article class="brick entry format-standard animate-this">

    <div class="entry-thumb">
        <a href="{{ url('posts/' . $post->slug) }}" class="thumb-
link"><img src="{{ $post->image }}"></a>
    </div>

    <div class="entry-text">
        <div class="entry-header">
             <h1 class="entry-title"><a href="{{ url('posts/' .
$post->slug) }}">{{ $post->title }}</a></h1>
        </div>
        <div class="entry-excerpt">
            {{ $post->excerpt }}
        </div>
    </div>

</article>
```

Pour chaque article ce code est généré et on voit que les propriétés sont récupérées de façon classique, par exemple pour le

titre avec **$post->title**.

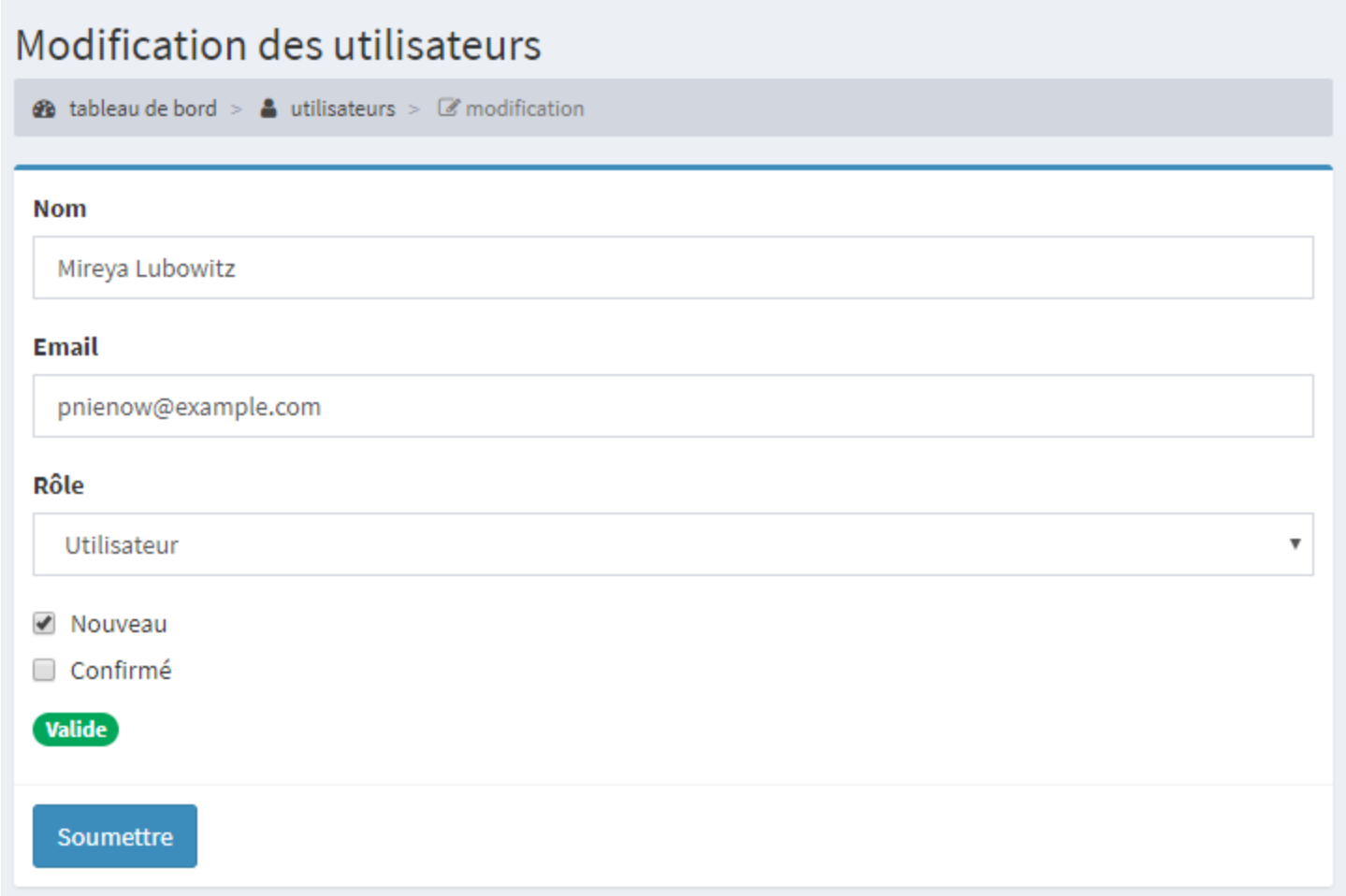
L'application utilise le package barryvdh/laravel-debugbar qui permet en particulier de surveiller les requêtes générées :



On aura de nombreuses occasions d'utiliser cette barre et n'hésitez pas à en explorer les possibilités. Elle ne fonctionne que lorsqu'on est en mode DEBUG.

La modification des utilisateurs

Quand on se connecte en tant qu'administrateur et qu'on va dans l'administration, les utilisateurs et qu'on en édite un on se retrouve avec ce formulaire :



Si vous allez dans le fichier **UserRepository** vous trouvez ce code :

```
public function update($request, $user)
{
    $inputs = $request->all ();

    if (isset($inputs['confirmed'])) {
        $inputs['confirmed'] = true;
    }

    if (isset($inputs['valid'])) {
        $inputs['valid'] = true;
    }

    $user->update($inputs);

    ...
}
```

Voici les actions réalisées :

- on récupère toutes les entrées du formulaire **$request->all()** dans le tableau **$inputs**,
- si la case à cocher « Confirmé » est cochée on ajoute la colonne **confirmed** avec la valeur **true**, (on sait que si une case n'est pas cochée dans un formulaire aucune information n'est envoyée),
- si la case à cocher « Valide » est cochée on ajoute la colonne **valid** avec la valeur **true**,
- on met l'utilisateur à jour dans la table avec la méthode **update**.

On continuera à explorer cette application dans les prochains chapitres mais n'hésitez pas à fouiller dans le code !

En résumé

- On peut créer des enregistrements avec Eloquent avec la méthode **create**.
- On peut modifier des enregistrements avec Eloquent avec la méthode **update**.
- On peut créer ou modifier des enregistrements avec Eloquent avec la méthode **save**.
- Le résultat d'une requête générée avec Eloquent est soit un modèle soit une collection.
- Eloquent est associé à un puissant générateur de requêtes (Query Builder).
- Des méthodes spéciales permettent de faire plusieurs actions comme aller chercher un enregistrement et le créer s'il n'existe pas.