Cours Laravel 5.3 – les données – les ressources (1/2)

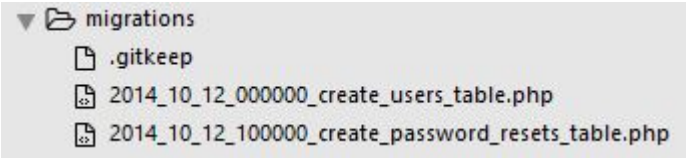
Dans ce chapitre nous allons commencer à étudier les ressources qui permettent de créer des routes « CRUD » (**C**reate, **R**ead, **U**pdate, **D**elete) adaptées à la persistance de données. Comme exemple pratique nous allons prendre le cas d'une table d'utilisateurs, une situation qui se retrouve dans la plupart des applications.

Les données

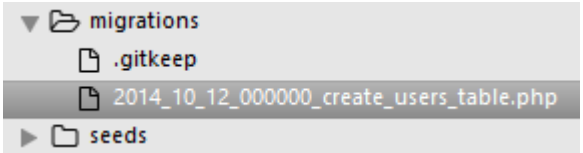
La migration

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.

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



Pour le moment on va garder seulement la première qui concerne la création de la table des utilisateurs :



En voici le code :

```
<?php

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

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

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

On a les champs :

- **id** : entier auto-incrémenté qui sera la clé primaire de la table,
- **name** : texte pour le nom,
- **email** : texte pour l'email (unique),
- **password** : texte pour le mot de passe (en fait une version cryptée du mot de passe),
- **remember_token** : généré par la méthode **rememberToken** 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 :

```
λ php artisan migrate
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
```

On doit se retrouver avec la table **users** créée :

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	**id**	int(10)		UNSIGNED	No	*None*		AUTO_INCREMENT
2	**name**	varchar(255)	utf8_unicode_ci		No	*None*		
3	**email**	varchar(255)	utf8_unicode_ci		No	*None*		
4	**password**	varchar(255)	utf8_unicode_ci		No	*None*		
5	**remember_token**	varchar(100)	utf8_unicode_ci		Yes	*NULL*		
6	**created_at**	timestamp			Yes	*NULL*		
7	**updated_at**	timestamp			Yes	*NULL*		

*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

C'est la commande qu'on a déjà vue pour créer un contrôleur avec en plus l'option **–resource**.

Vous trouvez comme résultat le contrôleur **app/Http/Controllers/UserController** :

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;

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
     */
    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 :

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

Les routes

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

```
Route::resource('user', 'UserController');
```

On va vérifier ces routes avec Artisan :

```
λ php artisan route:list
+--------+-----------+-----------------+-------------+-----------------------------------------+------------+
| Domain | Method    | URI             | Name        | Action                                  | Middleware |
+--------+-----------+-----------------+-------------+-----------------------------------------+------------+
|        | GET|HEAD  | user            | user.index  | App\Http\Controllers\UserController@index   | web        |
|        | POST      | user            | user.store  | App\Http\Controllers\UserController@store   | web        |
|        | GET|HEAD  | user/create     | user.create | App\Http\Controllers\UserController@create  | web        |
|        | GET|HEAD  | user/{user}     | user.show   | App\Http\Controllers\UserController@show    | web        |
|        | PUT|PATCH | user/{user}     | user.update | App\Http\Controllers\UserController@update  | web        |
|        | DELETE    | user/{user}     | user.destroy| App\Http\Controllers\UserController@destroy | web        |
|        | GET|HEAD  | user/{user}/edit| user.edit   | App\Http\Controllers\UserController@edit    | web        |
+--------+-----------+-----------------+-------------+-----------------------------------------+------------+
```

Vous trouvez 7 routes, avec chacune une méthode et une url, qui pointent sur les 7 méthodes du contrôleur. Notez également que chaque route a aussi un nom qui peut être utilisé par exemple pour une redirection. On retrouve aussi pour chaque route le middleware **web** dont je vous ai déjà parlé.

Le middleware web est automatiquement ajouté à toutes les routes.

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.

On peut limiter le nombre de routes si on n'a pas besoin de toutes les avoir.

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.

On pourrait se contenter d'une seule requête de formulaire en discriminant grâce à la méthode ou à la constitution de l'url. Mais il me semble plus clair d'en utiliser deux.

Pour la gestion une seule classe **UserRepository** suffira.

Si on considère ces injections voici le code du contrôleur :

```
<?php
```

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests\UserCreateRequest;
use App\Http\Requests\UserUpdateRequest;
use App\Repositories\UserRepository;
use App\User;

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);

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

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

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

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

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

```
    }

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

    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.");
    }

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

        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.

Liaison implicite

Il me faut toutefois préciser déjà un point important. Dans la version du contrôleur générée par défaut on voit que l’utilisateur au niveau des arguments des fonctions est référencé par son identifiant, par exemple :

```
public function show($id)
```

La variable **id** contient la valeur passée dans l'url. Par exemple **…/user/8** indique qu'on veut voir les informations de l'utilisateur d'identifiant 8. Il suffit donc ensuite d'aller chercher dans la base l'utilisateur correspondant.

Si vous regardez la signature dans le contrôleur définitif vous avez ceci :

public function show(User $user)

L'argument cette fois est une instance du modèle **App\User**. Etant donné qu'il rencontre ce type Laravel va automatiquement livrer une instance du modèle pour l'utilisateur concerné ! C'est ce qu'on appelle liaison implicite (**Implicit Bindind**). Vous voyez encore là à quel point Laravel nous simplifie la vie .

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 Illuminate\Foundation\Http\FormRequest;

class UserCreateRequest extends FormRequest
{
    /**
     * 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 [
            'name' => 'bail|required|max:255',
            'email' => 'bail|required|email|max:255|unique:users',
            'password' => 'bail|required|confirmed|min:8'
        ];
    }
}
```

Avec ces règles :

- **bail** : on s'arrête à la première erreur,
- **name** : requis, longueur maximale de 255 caractères,
- **email** : requis, adresse valide, longueur maximale de 255 caractères, et unique dans la table **users**,
- **password** : requis, longueur minimale de 8 caractères et doit correspondre à ce qui est entré dans le champ de confirmation du mot de passe (**confirmed**).

Modification d'un utilisateur

Pour la modification d'un utilisateur nous allons avoir un petit souci. En effet on veut conserver l'unicité de l'email dans la base, il est donc judicieux de prévoir une règle « unique ». Mais comme la valeur de l'email existant est déjà dans la base on va avoir un échec de la validation en cas de non modification de cette valeur, ce qui est probable.

Comment nous en sortir ?

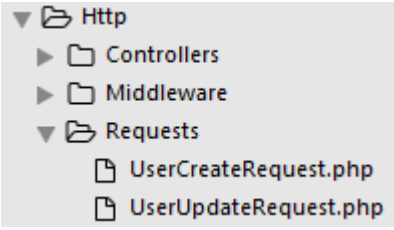
Étant donné que nous disposons d'une fonction on peut effectuer

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 Illuminate\Foundation\Http\FormRequest;

class UserUpdateRequest extends FormRequest
{
    /**
     * 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 [
            'name' => 'bail|required|max:255',
                                                    'email'    =>
'bail|required|email|max:255|unique:users,email,'       .
$this->user->id,
        ];
    }
}
```

On récupère l'instance du modèle pour l'utilisateur dans l'url (on sait qu'il y a une liaison implicite). Ensuite on utilise une possibilité d'exclusion de la règle « unique ».

Vous devez donc avoir ces 2 fichiers pour 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.
- On peut mettre en place dans le routage une liaison implicite pour générer automatiquement une instance de la classe dont l'identifiant est passée dans l'url.
- Pour une ressource la validation est toujours différente entre la création et la modification et il faut adapter le code pour en tenir compte.