Cours Laravel 5.5 – les données – migrations et modèles

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.

Dans cette deuxième partie du cours je vais me référer à l'application d'exemple présente sur Github plutôt que d'élaborer des mini applications pas trop réalistes. On va ainsi progresser dans l'apprentissage de Laravel en décortiquant cette application très complète. Il vous faut donc l'installer en suivant la procédure indiquée…

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. Vous avez ainsi un suivi de vos modifications.

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 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
```

Ici nous avons les valeurs par défaut à l'installation de Laravel. Il faudra évidemment modifier ces valeurs selon votre contexte de développement et définir surtout le nom de la base, le nom de l'utilisateur et le mot de passe. Pour une installation de MySql en local en général l'utilisateur est **root** et on n'a pas de mot de passe.

Artisan

Nous avons déjà utilisé Artisan 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 :

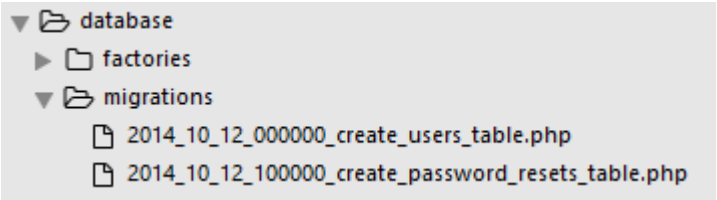
```
migrate
 migrate:fresh            Drop all tables and re-run all migrations
 migrate:install          Create the migration repository
 migrate:refresh          Reset and re-run all migrations
 migrate:reset            Rollback all database migrations
 migrate:rollback         Rollback the last database migration
 migrate:status           Show the status of each migration
```

On dispose de 6 commandes :

- **fresh** : supprime toutes les tables et relance la migration (commande apparue avec la version 5.5)
- **install** : crée et informe la table de référence des migrations
- **refresh** : réinitialise et relance les migrations
- **rollback** : annule la dernière migration
- **status** : donne des informations sur les migrations

Installation

Si vous regardez dans le dossier **database/migrations** il y a déjà 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.

Puisque ces migrations sont présentes autant les utiliser pour nous entrainer.

Commencez par créer une base MySQL et informez **.env**, par exemple :

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel5
DB_USERNAME=root
DB_PASSWORD=
```

Lancez alors la commande install :

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

On se retrouve alors avec une table **migrations** dans la base avec cette structure :

Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
id 🔑	int(10)		UNSIGNED	Non	*Aucun(e)*		AUTO_INCREMENT
migration	varchar(255)	utf8mb4_unicode_ci		Non	*Aucun(e)*		
batch	int(11)			Non	*Aucun(e)*		

Pour le moment cette table est vide, elle va se remplir au fil des migrations pour les garder en mémoire.

Vous n'aurez jamais à intervenir directement sur cette table qui est là juste pour la gestion effectuée par Laravel.

Constitution d’une migration

Si vous ouvrez le fichier **database/migrations/2014_10_12_000000_create_users_table.php** vous trouvez ce 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::dropIfExists('users');
    }
}
```

On dispose dans cette classe de deux fonctions :

- **up** : ici on a le code de création de la table et de ses colonnes
- **down** : ici on a le code de suppression de la table

Lancer une migration

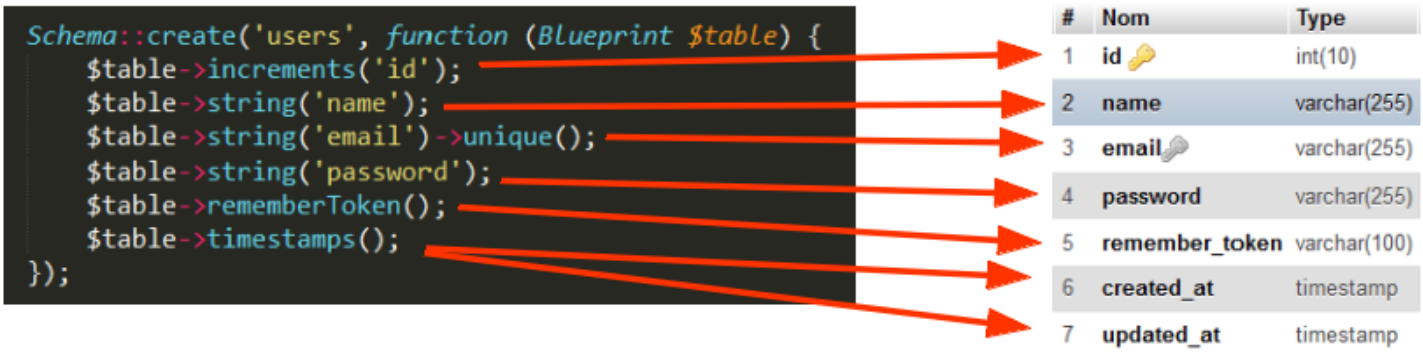
Pour lancer toutes les migrations on utilise la commande **migrate** :

```
λ php artisan migrate
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table
```

On voit que les deux migrations présentes ont été exécutées et on trouve les deux tables dans la base :

Table
migrations
password_resets
users

Pour comprendre le lien entre migration et création de la table associée voici une illustration pour la table **users** :



La méthode **timestamps** permet la création des deux colonnes **created_at** et **updated-at**.

Annuler ou rafraichir une migration

Pour annuler une migration on utilise la commande **rollback** :

```
λ php artisan migrate:rollback
Rolling back: 2014_10_12_100000_create_password_resets_table
Rolled back:  2014_10_12_100000_create_password_resets_table
Rolling back: 2014_10_12_000000_create_users_table
Rolled back:  2014_10_12_000000_create_users_table
```

Les méthodes **down** des migrations sont exécutées et les tables sont supprimées.

Pour annuler et relancer en une seule opération on utilise la commande **refresh** :

```
λ php artisan migrate:refresh
Rolling back: 2014_10_12_100000_create_password_resets_table
Rolled back:  2014_10_12_100000_create_password_resets_table
Rolling back: 2014_10_12_000000_create_users_table
Rolled back:  2014_10_12_000000_create_users_table
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table
```

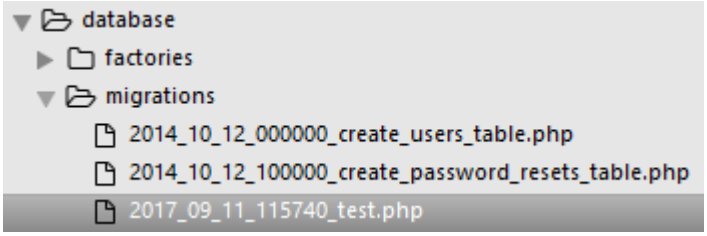
*Pour éviter d'avoir à coder ces méthode **down** la version 5.5 a prévu la commande **fresh** qui supprime automatiquement les tables concernées.*

Créer une migration

Il existe un commande d'artisan pour créer un squelette de migration :

```
λ php artisan make:migration test
Created Migration: 2017_09_11_115740_test
```

La migration est créée dans le dossier :



Avec ce code de base :

```
<?php
```

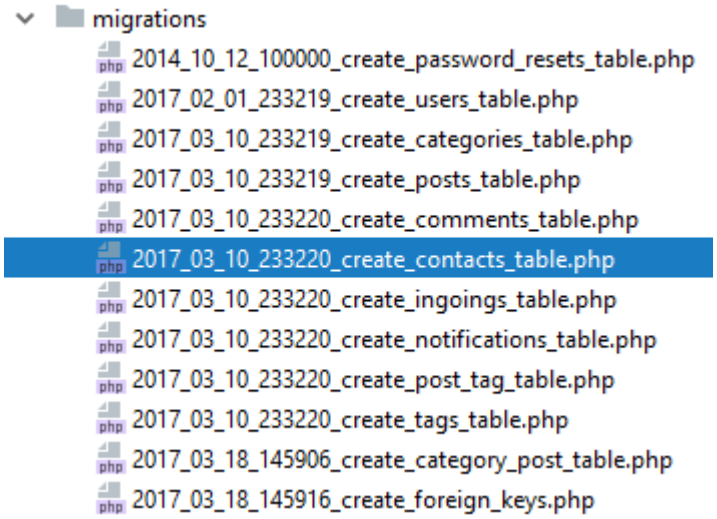
```
use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class Test extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        //
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        //
    }
}
```

Il faut ensuite compléter ce code selon vos besoins !

Le nom du fichier de migration commence par sa date de création, ce qui conditionne son positionnement dans la liste. L'élément important à prendre en compte est que l'ordre des migrations prend une grande importance lorsqu'on a des clés étrangères !

Les migrations de l'application d'exemple

L'application d'exemple comporte de nombreuse migrations :



Pour le présent chapitre nous allons nous intéresser à la migration de la table **contacts** :

```
<?php

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

class CreateContactsTable extends Migration {

  public function up()
  {
    Schema::create('contacts', function(Blueprint $table) {
      $table->increments('id');
      $table->timestamps();
      $table->string('name');
      $table->string('email');
      $table->text('message');
    });
  }

  public function down()
  {
    Schema::drop('contacts');
  }
}
```

Pour cette table on a :

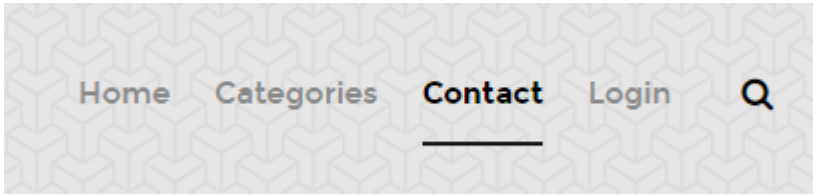
- une colonne clé incrémentée **id** (**increments**)
- les colonnes **created_at** et **updated_at** créées par la méthode

timestamps
- une colonne **name** de type **varchar** (**string**)
- une colonne **email** de type **varchar** (**string**)
- une colonne **text** de type **text** (**text**)

Lorsque vous lancez les migrations la table contacts est créée :

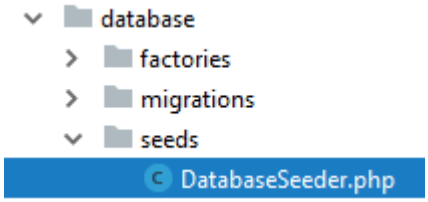
Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
id	int(10)		UNSIGNED	Non	*Aucun(e)*		AUTO_INCREMENT
created_at	timestamp			Oui	*NULL*		
updated_at	timestamp			Oui	*NULL*		
name	varchar(255)	utf8mb4_unicode_ci		Non	*Aucun(e)*		
email	varchar(255)	utf8mb4_unicode_ci		Non	*Aucun(e)*		
message	text	utf8mb4_unicode_ci		Non	*Aucun(e)*		

Cette table est destinée dans l'application à mémoriser les messages laissés par les visiteurs lorsqu'ils accèdent au formulaire avec cet item du menu :



La population (seeding)

En plus de proposer des migrations Laravel permet aussi la population (**seeding**), c'est à dire un moyen simple de remplir les tables d'enregistrements. Les classes de la population se trouvent dans le dossier **databases/seeds**:



On est libres d'organiser les classes comme on veut dans ce dossier. Dans l'application d'exemple il n'y a qu'une classe.

Lorsqu'on installe Laravel on dispose de la classe **DatabaseSeeder** avec ce code :

```
<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        // $this->call(UsersTableSeeder::class);
    }
}
```

On a un appel commenté à une classe **UserTableSeeder** qui pourrait être une classe pour remplir la table **users**.

Dans la fonction **run** on met tout le code qu'on veut pour remplir les tables. Dans l'application d'exemple on a :

```
User::create(
    [
        'name' => 'GreatAdmin',
        'email' => 'admin@la.fr',
        'password' => bcrypt('admin'),
        'role' => 'admin',
        'valid' => true,
        'confirmed' => true,
        'remember_token' => str_random(10),
    ]
);
```

Ici on crée l'enregistrement d'un utilisateur dans la table **users** avec l'ORM **Eloquent** dont je vais parler ci-dessous.

Je reviendrai dans ce cours sur la population quand on aura avancé notre connaissance des outils pour les bases de données. Je vais juste ajouter qu'on lance la population avec cette commande :

```
php artisan db:seed
```

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.

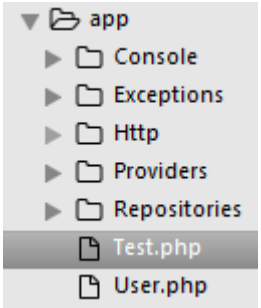
Quel intérêt ?

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**. On peut créer un modèle avec Artisan :

```
php artisan make:model Test
```

On trouve le fichier ici :



Avec cette trame de base :

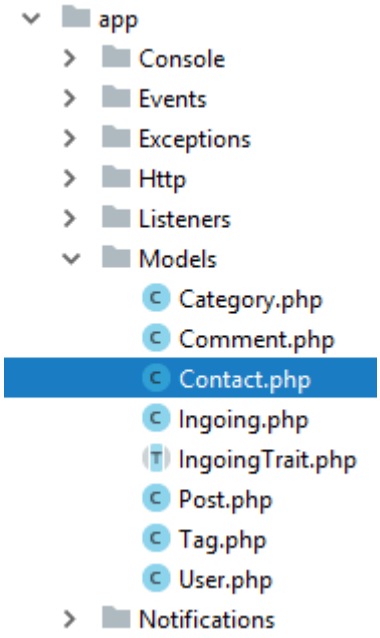
```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Test extends Model
{
    //
}
```

On peut créer un modèle en même temps que la migration pour la table avec cette syntaxe :

php artisan make:model Test -m

```
λ php artisan make:model Test -m
Model created successfully.
Created Migration: 2017_09_11_152337_create_tests_table
```

Dans l'application d'exemple les modèles sont rassemblés dans un dossier spécifique dans un souci d'organisation :



Ce modèle **Contact** correspond par défaut à la table **contacts**. Eloquent a quelques conventions de ce genre qui simplifient le codage. Si le nom de la table ne répond pas à la convention il faut prévoir une propriété **$table** pour préciser ce nom.

Routes et contrôleur

Routes

Pour les contacts dans l'application il faut deux routes :

- une route pour envoyer le formulaire à l'utilisateur
- une route pour la validation du formulaire

Si vous regardez dans le fichiers **routes/web.php** vous trouvez

cette ligne de code :

```
Route::resource('contacts', 'Front\ContactController', ['only' => ['create', 'store']]);
```

La méthode **resource** crée par défaut ces 7 routes **CRUD** :

VerbE	URI	Action	NOM DE LA ROUTE
GET	/contacts	index	contacts.index
GET	/contacts/create	create	contacts.create
POST	/contacts	store	contacts.store
GET	/contacts/{contact}	show	contacts.show
GET	/contacts/{contact}/edit	edit	contacts.edit
PUT/PATCH	/contacts/{contact}	update	contacts.update
DELETE	/contacts/{contact}	destroy	contacts.destroy

Il existe un commande d’artisan pour créer un contrôleur qui digère directement toutes ces routes :

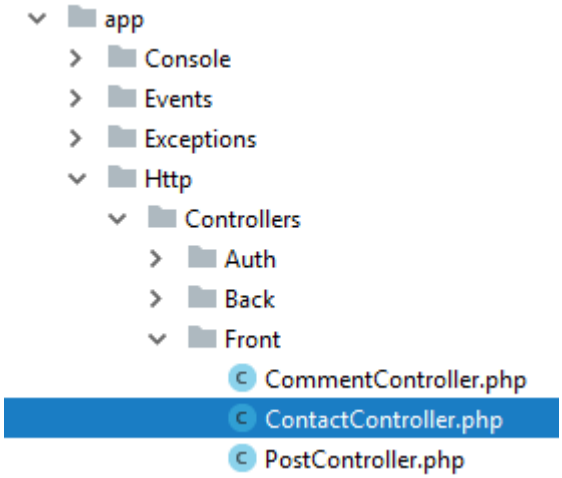
```
php artisan make:controller ContactController --resource
```

Je reviendrai sur ce point dans un chapitre ultérieur. Pour le moment notre contrôleur de contact se contentera de deux méthodes (**create** et **store**) c’est pour cette raison qu’on utilise le **only** dans la syntaxe de la route pour générer seulement les deux routes correspondantes. On aurait aussi pu détailler les deux routes :

```
Route::name('contacts.create')->get('contacts', 'Front\ContactController@create');
Route::name('contacts.store')->post('contacts', 'Front\ContactController@store');
```

Contrôleur

Le contrôleur est rangé ici :



Avec ce code :

```
<?php

namespace App\Http\Controllers\Front;

use App\ {
    Http\Controllers\Controller,
    Http\Requests\ContactRequest,
    Models\Contact
};

class ContactController extends Controller
{
    /**
     * Create a new ContactController instance.
     *
     */
    public function __construct()
    {
        $this->middleware('guest');
    }

    /**
     * Show the form for creating a new contact.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        return view ('front.contact');
    }
```

```
    /**
     * Store a newly created contact in storage.
     *
     * @param  ContactRequest $request
     * @return \Illuminate\Http\Response
     */
    public function store(ContactRequest $request)
    {
        Contact::create ($request->all ());

         return back ()->with ('ok', __('Your message has been recorded, we will respond as soon as possible.'));
    }
}
```

La méthode **create** ne présente aucune nouveauté, elle se contente de renvoyer une vue avec le formulaire.

La méthode **store** est chargée de mémoriser les informations dans la table **contacts**. On voit qu'on utilise la méthode **create** du modèle pour le réaliser. Il suffit de donner comme paramètres à cette méthode les entrées du formulaire.

Remarquez qu'on utilise une requête de formulaire (**ContactRequest**) pour la validation avec des règles classiques :

```
public function rules()
{
    return [
        'name' => 'bail|required|max:255',
        'email' => 'bail|required|email',
        'message' => 'bail|required|max:1000'
    ];
}
```

Le modèle en détail

Modifiez ainsi le code dans le contrôleur **ContactController** :

```
dd(Contact::create ($request->all ()));
```

L'helper **dd** est bien pratique il regroupe un **var_dump** et un **die**.

Ici si on soumet le formulaire on va observer de plus près le modèle créé parce que la méthode **create** renvoie ce modèle :

```
Contact {#302 ▼
 #dispatchesEvents: array:1 [▶]
 #fillable: array:3 [▼
   0 => "name"
   1 => "email"
   2 => "message"
 ]
 #connection: "mysql"
 #table: null
 #primaryKey: "id"
 #keyType: "int"
 +incrementing: true
 #with: []
 #withCount: []
 #perPage: 15
 +exists: true
 +wasRecentlyCreated: true
 #attributes: array:6 [▼
   "name" => "Durand"
   "email" => "durand@chezlui.fr"
   "message" => "Mon message"
   "updated_at" => "2017-09-11 15:28:59"
   "created_at" => "2017-09-11 15:28:59"
   "id" => 7
 ]
 #original: array:6 [▶]
 #changes: []
 #casts: []
 #dates: []
 #dateFormat: null
 #appends: []
 #observables: []
 #relations: []
 #touches: []
 +timestamps: true
 #hidden: []
 #visible: []
 #guarded: array:1 [▶]
}
```

On a pas mal de propriétés mais surtout des attributs (**attributes**)

et on se rend compte que chacun de ces attributs correspond à une colonne de la table contact :

id	created_at ▾ 1	updated_at	name	email	message
7	2017-09-11 15:28:59	2017-09-11 15:28:59	Durand	durand@chezlui.fr	Mon message

Méthode create et assignement de masse

Par sécurité ce type d'assignement de masse (on transmet directement un tableau de valeurs issues du client avec la méthode **create**) est limité pour la sécurité 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/Models/Contact.php** vous trouvez cette propriété :

```
protected $fillable = ['name', 'email', 'message'];
```

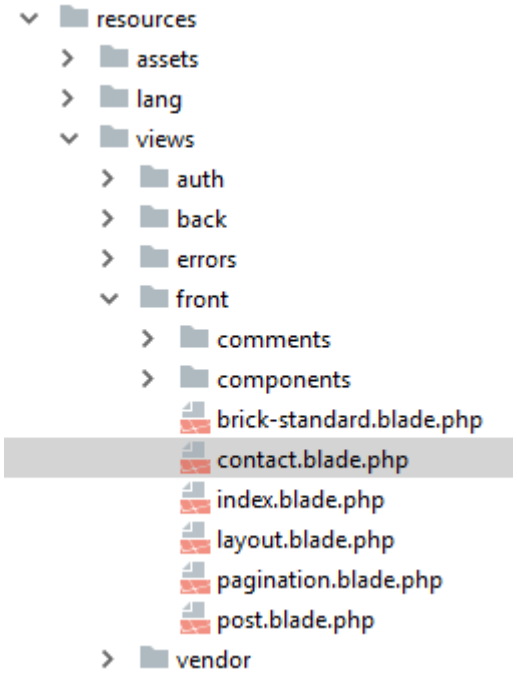
Ce sont les seules colonnes qui seront impactées par la méthode **create** (et équivalentes). Attention à cela lorsque vous avez un bug mystérieux avec des colonnes qui ne se mettent pas à jour !

Mais quel est le risque ?

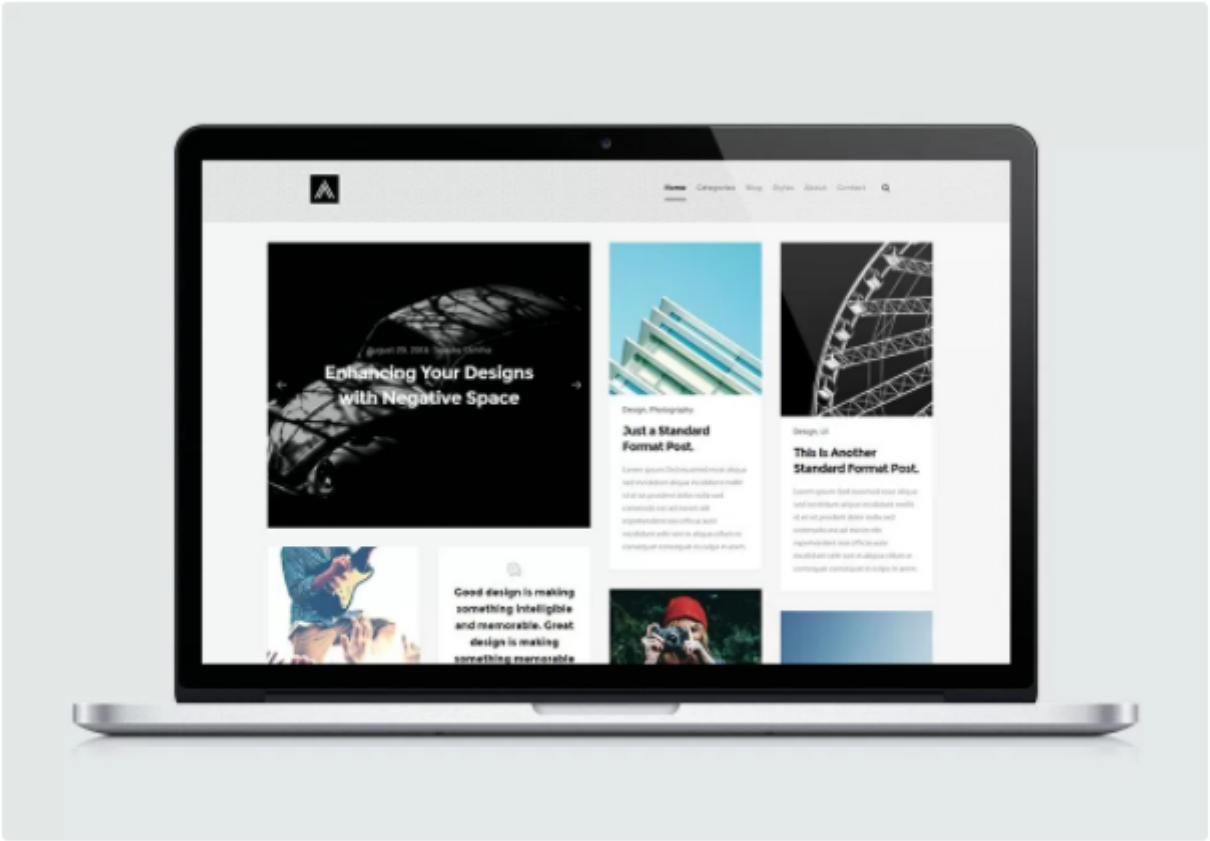
Imaginez qu'il y ait une autre colonne avec des données sensibles et non prévue dans le formulaire mais qu'un petit malin l'ajoute à la requête, cette colonne serait mise à jour en même temps que les autres !

Les vues

Je ne vais pas détailler les vues dans ce chapitre consacré aux données. Je signale juste que les vues sont rangées dans des dossiers :



Le template pour les vues du **front** est **layout.blade.php**. Le template utilisé est issu de [cette source](#) :

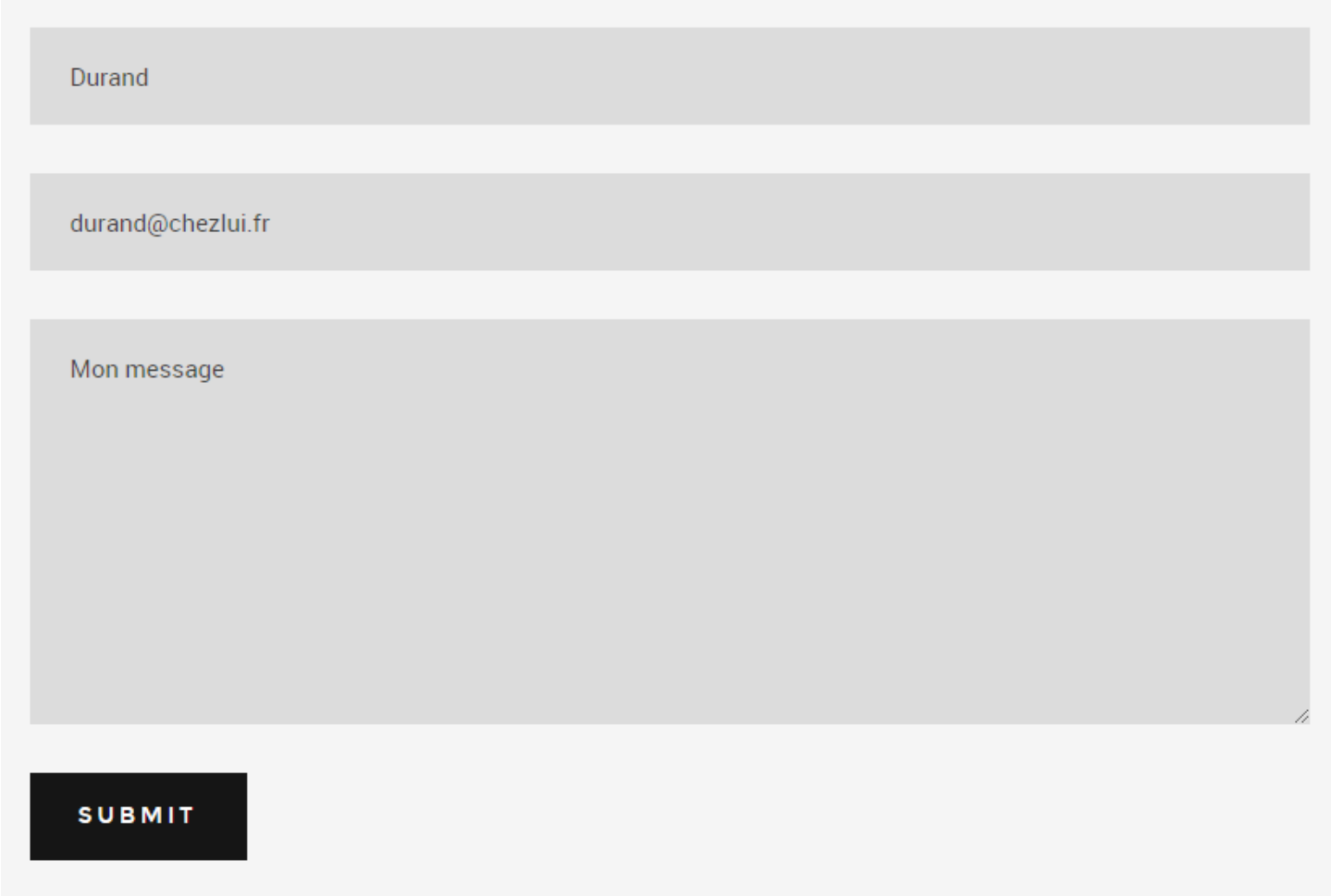


Abstract.

Je l’ai évidemment adapté pour Laravel mais je consacrerai un chapitre sur le sujet parce que de nombreuse possibilités de Blade

ont été utilisées.

Le formulaire de contact a cet aspect :



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.
- L’assignement de masse est limité par la propriété **$fillable**.