


# Cours Laravel 6 – les données – migrations et modèles

---

 [laravel.sillo.org/cours-laravel-6-les-donnees-migrations-et-modeles/](https://laravel.sillo.org/cours-laravel-6-les-donnees-migrations-et-modeles/)

bestmomo

August 30,  
2019

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.

## 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 :

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

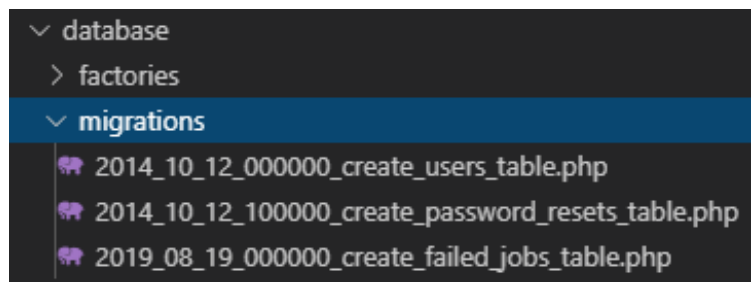
On dispose de 6 commandes :

- **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 3 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,
- table **failed\_jobs** : une migration qui concerne les queues.



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=laravel6
DB_USERNAME=root
DB_PASSWORD=
```

Lancez alors la commande **install** :

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

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

Name: migrations

Database: laravel6

Engine: INNODB

Columns

<input type="checkbox"/>	Name	Data Type	Unsigned	Auto Increment	Not Null	Default	Collation
<input type="checkbox"/>	id	INT(10)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
<input type="checkbox"/>	migration	VARCHAR(255)	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>		utf8mb4_unicode_ci
<input type="checkbox"/>	batch	INT(11)	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>		

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->bigIncrements('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $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 les migrations

---

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

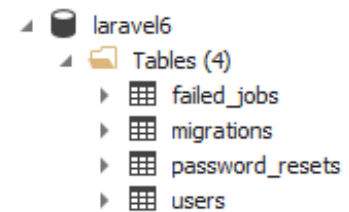
```

λ php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (0.55 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (0.59 seconds)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (0.32 seconds)

```

On voit que les 3 migrations présentes ont été exécutées et on trouve les 3 tables dans la base (en plus de celle de gestion des migrations) :

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



```
Schema::create('users', function (Blueprint $table) {
    $table->bigIncrements('id');
    $table->string('name');
    $table->string('email')->unique();
    $table->timestamp('email_verified_at')->nullable();
    $table->string('password');
    $table->rememberToken();
    $table->timestamps();
});
```

#	Nom	Type
1	id	bigint(20)
2	name	varchar(255)
3	email	varchar(255)
4	email_verified_at	timestamp
5	password	varchar(255)
6	remember_token	varchar(100)
7	created_at	timestamp
8	updated_at	timestamp

La méthode **timestamps** permet la création des deux colonnes **created\_at** et **updated\_at**.

## Annuler ou rafraîchir une migration

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

```
λ php artisan migrate:rollback
Rolling back: 2019_08_19_000000_create_failed_jobs_table
Rolled back: 2019_08_19_000000_create_failed_jobs_table (0.17 seconds)
Rolling back: 2014_10_12_100000_create_password_resets_table
Rolled back: 2014_10_12_100000_create_password_resets_table (0.2 seconds)
Rolling back: 2014_10_12_000000_create_users_table
Rolled back: 2014_10_12_000000_create_users_table (0.14 seconds)
```

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: 2019_08_19_000000_create_failed_jobs_table
Rolled back: 2019_08_19_000000_create_failed_jobs_table (0.16 seconds)
Rolling back: 2014_10_12_100000_create_password_resets_table
Rolled back: 2014_10_12_100000_create_password_resets_table (0.14 seconds)
Rolling back: 2014_10_12_000000_create_users_table
Rolled back: 2014_10_12_000000_create_users_table (0.14 seconds)
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (0.45 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (0.44 seconds)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (0.21 seconds)
```

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

```
λ php artisan migrate:fresh
Dropped all tables successfully.
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (0.52 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (0.63 seconds)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (0.28 seconds)
```

## Créer une migration

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

La migration est créée dans le dossier :

```
λ php artisan make:migration test
Created Migration: 2019_08_29_222938_test
```

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()
    {
        //
    }
}
```

```
▼ migrations
  2014_10_12_000000_create_users_table.php
  2014_10_12_100000_create_password_resets_table.php
  2019_08_19_000000_create_failed_jobs_table.php
  2019_08_29_222938_test.php
```

Il faut ensuite compléter ce code selon vos besoins !

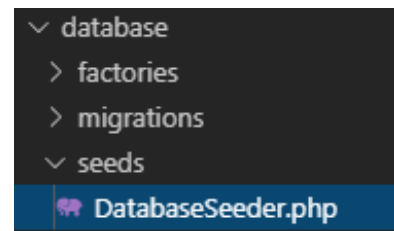
*Le nom du fichier de migration commence par sa date de création, ce qui conditionne son positionnement dans la liste. L'élément important à prendre en compte est que l'ordre des migrations prend a grande importance lorsqu'on a des clés étrangères !*

## 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 : en garder une seule ou en faire plusieurs pour être mieux organisé.

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. Prenons un exemple pour la table **users** :

```

public function run()
{
    App\User::create(
        [
            'name' => 'Dupont',
            'email' => 'dupont@la.fr',
            'password' => bcrypt('pass'),
        ]
    );
}

```

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. On lance la population avec cette commande :

```
php artisan db:seed
```

On vérifie dans la table :

```

λ php artisan db:seed
Database seeding completed successfully.

```

id	name	email	email_verified_at	password	remember_token	created_at	updated_at
UNSIGNED BIGINT(20)	VARCHAR(255)	VARCHAR(255)	TIMESTAMP	VARCHAR(255)	VARCHAR(100)	TIMESTAMP	TIMESTAMP
1	Dupont	dupont@la.fr	(null)	\$2y\$10\$uIF4aKooU1dr0iopXpoSK.Rmv8SNG.Z9JTowsxj1SD7q/EP5u9eK	(null)	29/08/2019 22:37:22	29/08/2019 22:37:22

## 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 :

```

λ php artisan make:model Test
Model created successfully.

```



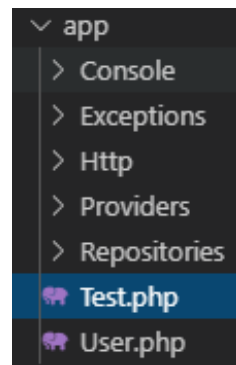
```
<?php
```

```
namespace App;
```

```
use Illuminate\Database\Eloquent\Model;
```

```
class Test extends Model
```

```
{  
    //  
}
```

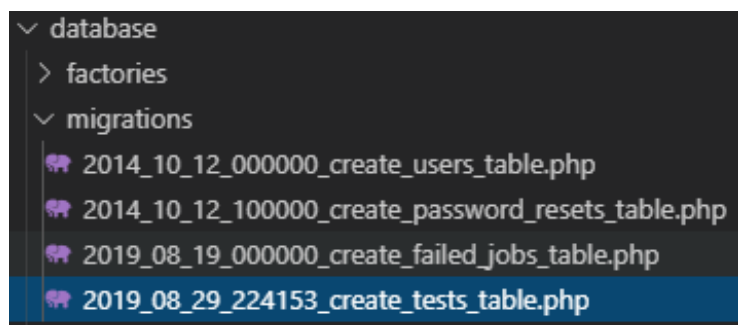


Lorsque les modèles deviennent nombreux il est conseillé de créer un dossier pour les ranger.

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: 2019_08_29_224153_create_tests_table
```



## Un exemple

Nous allons encore prendre un exemple simple de gestion de contacts mais cette fois en allant plus loin avec Eloquent.

## Le contrôleur

On va créer un contrôleur avec cette commande :

```
php artisan make:controller ContactsController
```

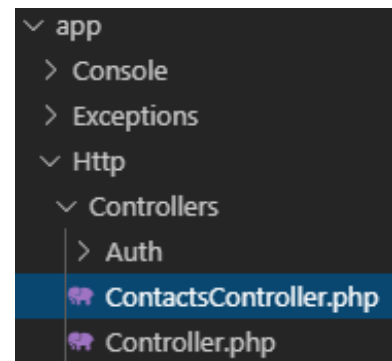
```
λ php artisan make:controller ContactsController  
Controller created successfully.
```

On va créer ces deux méthodes :

- **create** : pour appeler la vue avec le formulaire de création
- **store** : pour valider la saisie et enregistrer le contact dans la base

```
class ContactsController extends Controller
{
    public function create()
    {
        //
    }

    public function store(Request $request)
    {
        //
    }
}
```



Il va falloir coder ces méthodes...

## Les routes

Il nous faut deux routes :

- une route **GET** appeler la méthode **create** du contrôleur
- une route **POST** appeler la méthode **store** du contrôleur

```
Route::get('contact', 'ContactsController@create')->name('contact.create');
Route::post('contact', 'ContactsController@store')->name('contact.store');
```

On nomme les routes pour pouvoir les utiliser plus facilement.

## Le modèle et la migration

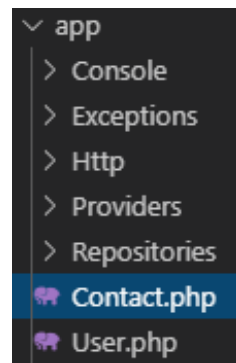
On va créer le modèle et la migration tant qu'on y est pour la table **contacts** :

php artisan make:model Contact -m

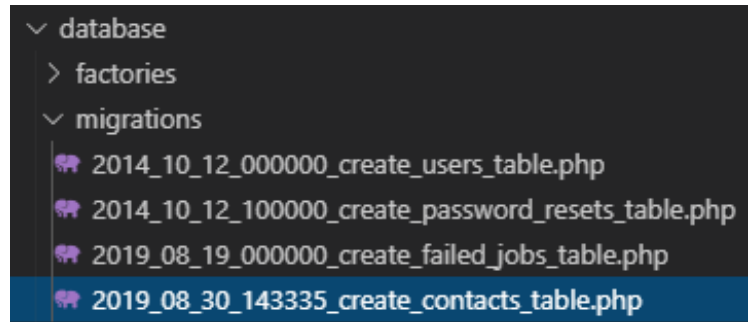
```
λ php artisan make:model Contact -m
Model created successfully.
Created Migration: 2019_08_30_143335_create_contacts_table
```

Dans le code généré pour la migration on va ajouter deux colonnes (**message** et **email**) :

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



Et on lance la migration pour la table **contacts** :



```
λ php artisan migrate
Migrating: 2019_08_30_143335_create_contacts_table
Migrated: 2019_08_30_143335_create_contacts_table (0.42 seconds)
```

Par convention le modèle **Contact** sait qu'il est relié à la table **contacts**. Si on sortait de la convention de nommage il faudrait ajouter une propriété au modèle pour spécifier le nom de la table.

## Le formulaire

On va continuer à utiliser le template déjà vu dans les précédents articles (**resources/views/template.blade.php**) :

```

<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <title>Mon joli site</title>
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/ijTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
    <style>
      textarea { resize: none; }
      .card { width: 25em; }
    </style>
  </head>
  <body>
    @yield('contenu')
  </body>
</html>

```

On crée la vue du formulaire (**contact.blade.php**) :

```

@extends('template')

@section('contenu')
  <br>
  <div class="container">
    <div class="row card text-white bg-dark">
      <h4 class="card-header">Contactez-moi</h4>
      <div class="card-body">
        <form action="{{ route('contact.create') }}" method="POST">
          @csrf
          <div class="form-group">
            <input type="email" class="form-control @error('email') is-invalid @enderror"
name="email" id="email" placeholder="Votre email" value="{{ old('email') }}">
            @error('email')
              <div class="invalid-feedback">{{ $message }}</div>
            @enderror
          </div>
          <div class="form-group">
            <textarea class="form-control @error('message') is-invalid @enderror"
name="message" id="message" placeholder="Votre message">{{ old('message') }}</textarea>
            @error('message')
              <div class="invalid-feedback">{{ $message }}</div>
            @enderror
          </div>
          <button type="submit" class="btn btn-secondary">Envoyer !</button>
        </form>
      </div>
    </div>
  </div>
@endsection

```

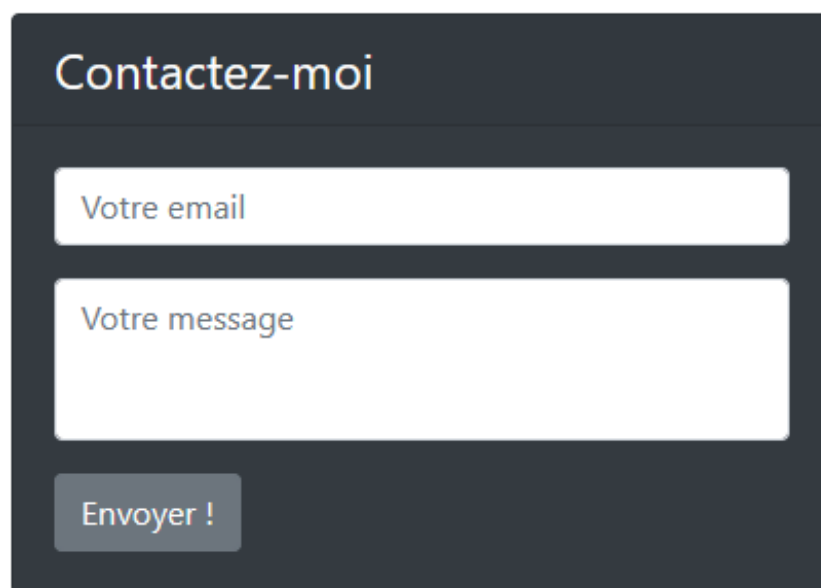
Rien de nouveau ici puisque c'est pratiquement le même code qu'on avait déjà vu. Remarquez l'attribut action du formulaire dans lequel on utilise le nom de la route ainsi que l'helper **route** :

```
<form action="{{ route('contact.create') }}" method="POST">
```

Il ne reste plus qu'à coder le contrôleur pour envoyer la vue :

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

Et on devrait l'obtenir à l'adresse **.../contact** :



The image shows a contact form with a dark background. At the top, the title 'Contactez-moi' is displayed in a light blue font. Below the title, there are two white input fields. The first field is labeled 'Votre email' and the second field is labeled 'Votre message'. At the bottom of the form, there is a grey button with the text 'Envoyer !' in white.

---

## L'enregistrement

---

Maintenant on doit coder le traitement de la soumission du formulaire dans le contrôleur :

```
public function store(Request $request)
{
    $this->validate($request, [
        'email' => 'bail|required|email',
        'message' => 'bail|required|max:500'
    ]);

    $contact = new \App\Contact;
    $contact->email = $request->email;
    $contact->message = $request->message;
    $contact->save();

    return "C'est bien enregistré !";
}
```

Je ne reviens pas sur la validation qu'on a vu dans un précédent article.

On crée une nouvelle instance du modèle, on renseigne ses propriétés et on finit avec un **save** pour l'enregistrer dans la base. Et si tout se passe bien on doit trouver l'enregistrement dans la table :

id UNSIGNED BIGINT(20)	message TEXT	email VARCHAR(255)	created_at TIMESTAMP	updated_at TIMESTAMP
1	Mon message à moi	dupont@la.fr	30/08/2019 14:58:59	30/08/2019 14:58:59

## Méthode create et assignement de masse

On peut aussi utiliser la méthode **create** du modèle pour enregistrer dans la base :

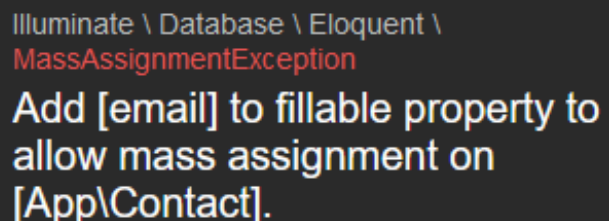
```
\App\Contact::create([
    'email' => $request->email,
    'message' => $request->message,
]);
```

Si vous utilisez ce code vous allez tomber sur cette erreur :

Par sécurité ce type d'assignement de masse (on transmet directement un tableau de valeurs issues du client avec la méthode create) est limité par une propriété au niveau du modèle qui désigne précisément les noms des colonnes susceptibles d'être modifiées.

Dans le modèle Contact on va ajouter ce code :

```
class Contact extends Model
{
    protected $fillable = ['email', 'message'];
}
```



Illuminate \ Database \ Eloquent \  
MassAssignmentException  
Add [email] to fillable property to  
allow mass assignment on  
[App\Contact].

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 !

## Le modèle en détail

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

```
dd(\App\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 {#196 ▼
  #fillable: array:2 [▼
    0 => "email"
    1 => "message"
  ]
  #connection: "mysql"
  #table: "contacts"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: true
  #attributes: array:5 [▼
    "email" => "dupont@la.fr"
    "message" => "Mon message à moi"
    "updated_at" => "2019-08-30 15:09:21"
    "created_at" => "2019-08-30 15:09:21"
    "id" => 3
  ]
  #original: array:5 [▶]
  #changes: []
  #casts: []
  #dates: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #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.

## 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**.