Les bases de données

1. Introduction

Nous allons parler des bases de données dans Laravel. Laravel propose une solution élégante/ simple que nous allons voir en partie. Comme d'habitude, n'hésitez pas à consulter le site Laravel pour la documentation officielle: Database: Getting Started

Pour vos tests, vous pourriez utiliser Laravel Sail qui permet d'avoir simplement un environnement dockerisé (utilisation d'un conteneur docker). Il contient: PHP et Mysql. Ceci ne rentre pas dans le cadre du cours mais sachez que c'est possible et vraiment très bien fait. Mais nous continuerons d'utiliser localement php, artisan et votre base de données Mysql.

2. Bases de données

Nous allons utiliser MySql car il est installé chez la plupart des personnes mais utiliser une autre base de données reviendrait au même: PosgreSQL, SQLite, Mariadb, etc.

Informations sur les différentes DB:

- MySQL 5.7+
- SQLite 3.8.8+
- PosgreSQL 9.6+
- Microsoft SQL Server 2017+: les extensions PHP seront nécessaires sqlsrv et pdo_sqlsrv. Le driver ODBC de Microsoft SQL peut aussi être nécessaire.

Dans Laravel on va utiliser le fichier .env pour indiquer nos credetials (login/password), le type de base de données, la base de données, l'ip/host et le port.

2.1 Mysql

Il faut que Mysql soit évidemment installé et que vous ayez toutes les informations de connexion.

Exemple d'entrées dans le fichier .env:

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

Adaptez votre fichier .env avec votre configuration de Mysql.

Si la base de données cupcake n'existe pas, créez-la:

```
mysql -u root -p
-- On va afficher les bases de données et on vérifie que cupcake n'existe pas
show databases;
-- Si cupcake n'existe pas.
create database cupcake;
-- on sort du client mysql
exit;
```

Il faut activer mysql_pdo dans le php.ini en décommentant (enlever le ; du début de ligne) la ligne extension=pdo_mysql

2.2 SQLite

Pour SQLite il faut simplement créer un fichier vide ou utiliser un fichier existant.
Pour les exemples, nous allons mettre ce fichier dans le répertoire database de Laravel et de le nommer par exemple: database.sqlite

Ensuite on met ces entrées dans le fichier .env

```
DB_CONNECTION=sqlite
DB_DATABASE=d:\monProjetLaravel\database\database.sqlite
```

Il faut activer pdo_sql dans le php.ini en décommentant (enlever le ; du début de ligne) la ligne extension=pdo_sqlite

3. Fichier .env

Le fichier .env est utilisé par Laravel pour déterminer un exemple de choses très importantes comme par exemples:

- APP_NAME: Le nom de l'application
- APP_ENV: L'environnement de l'application
- APP_KEY: La clef de l'application
- APP_DEBUG: L'application est-elle en mode debug ? Si on met à false on n'aura plus un dump détaillé quand une erreur survient. Ce qui est obligatoire avec un site en production !
- etc...

3.1 Accès aux variables

Toutes les variables contenues dans ce fichier seront chargées dans la variable PHP super globale $_ENV.

On peut y accéder via par exemple $_ENV['APP_NAME'] ou bien env('APP_NAME');

3.2 Protection

Il faut donc bien faire attention avec ce fichier car une partie des infos de notre application sont exposées. par exemple: http://monsiteLaravel/.env et vous pourriez avoir toute la config affichée dans un navigateur: mot de passe, addresse de la base de données, etc.

Pour éviter cela, on peut créer/modifier un fichier .htaccess.

Fichier .htaccess:

```
#Disable index view //On empêche de voir le contenu des répertoires
options -Indexes

#hide a Specifuc File // On cache le fichier .env
<Files .env>
order allow,deny
Deny from all
</Files>
```

De plus, si vous voulez utiliser git et que vous n'avez pas créé votre projet via la commande laravel:

```
laravel new monProjet --git
```

Vous devez ajouter avant votre premier commit le fichier .gitignore, ajoutez ce contenu généré par l'option --git

```
/node_modules
/public/hot
/public/storage
/storage/*.key
/vendor
.env
.env.backup
.phpunit.result.cache
docker-compose.override.yml
Homestead.json
Homestead.yaml
npm-debug.log
yarn-error.log
/.idea
/.vscode
```

Vous constatez que le fichier .env fait partie du fichier .gitignore. Lorsque vous pousserez vos modifications, .env ne sera pas envoyé et sera gardé précieusement. ☺

Le plus simple est tout de même de créer votre projet avec l'option --git si vous utilisez la

commande laravel.

4. Les migrations

Les migrations permettent de créer/modifier une base de données: tables, index, relation, etc. Toute modification est enregistrée. Il y a donc un suivi de tout ce que vous avez fait sur vos tables. On peut même faire marche arrière sur une migration (rollback). C'est un peu le git de votre base de données. Ici, nous allons voir rapidement les migrations mais la documentation Laravel complétera cette introduction: Database: Migrations

Pour que nos accès à la base de données mysql fonctionnent, il faut que l'on active l'extension dans le fichier php.ini.
Lorsque vous lancerez vos migrations, si vous avez ce message d'erreur:PDOException:☹"could not find driver") vous devez activer l'extension pdo_mysql. Pour cela, ouvrez le fichier php.ini et enlevez le ; se trouvant en début de ligne de l'extension: extension=pdo_mysql

Dans tout nouveau projet Laravel, il y a des migrations qui n'ont pas été exécutées.
Celles-ci se trouvent dans le répertoire database\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_12_14_000001_create_personal_access_tokens_table.php

Nous allons les exécuter ces migrations qui permettront à Laravel de faire la gestion des utilisateurs.

```
php artisan migrate
```

Résultat de la commande:

```
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table (46.03ms)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table (45.00ms)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated:  2019_08_19_000000_create_failed_jobs_table (40.15ms)
Migrating: 2019_12_14_000001_create_personal_access_tokens_table
Migrated:  2019_12_14_000001_create_personal_access_tokens_table (112.35ms)
```

Regardons les tables qui ont été créées:

```
mysql -u root -p
use cupcake;
show tables;
```

Résultat:

```
+------------------------+
| Tables_in_cupcake      |
+------------------------+
| failed_jobs            |
| migrations             |
| password_resets        |
| personal_access_tokens |
| users                  |
+------------------------+
5 rows in set (0.00 sec)
```

Ces tables sont toutes vides sauf la table migrations. Regardons le contenu de celle-ci:

```
mysql -u root -p
use cupcake;
select * from migrations;
```

Résultat:

```
+----+-----------------------------------------------------+-------+
| id | migration                                           | batch |
+----+-----------------------------------------------------+-------+
|  1 | 2014_10_12_000000_create_users_table                |     1 |
|  2 | 2014_10_12_100000_create_password_resets_table      |     1 |
|  3 | 2019_08_19_000000_create_failed_jobs_table          |     1 |
|  4 | 2019_12_14_000001_create_personal_access_tokens_table |     1 |
+----+-----------------------------------------------------+-------+
4 rows in set (0.00 sec)
```

On voit que ces enregistrements correspondent aux noms des fichiers contenus dans le répertoire database\migrations.

4.1 Comparons table et migration

Prenons la table users et regardons la structure des champs:

```
mysql -u root -p
use cupcake;
show fields from users;
exit;
```

Résultat:

```
+-------------------+-----------------+------+-----+---------+----------------+
| Field             | Type            | Null | Key | Default | Extra          |
+-------------------+-----------------+------+-----+---------+----------------+
| id                | bigint unsigned | NO   | PRI | NULL    | auto_increment |
| name              | varchar(255)    | NO   |     | NULL    |                |
| email             | varchar(255)    | NO   | UNI | NULL    |                |
| email_verified_at | timestamp       | YES  |     | NULL    |                |
| password          | varchar(255)    | NO   |     | NULL    |                |
| remember_token    | varchar(100)    | YES  |     | NULL    |                |
| created_at        | timestamp       | YES  |     | NULL    |                |
| updated_at        | timestamp       | YES  |     | NULL    |                |
+-------------------+-----------------+------+-----+---------+----------------+
8 rows in set (0.00 sec)
```

Maintenant, regardons le fichier correspondant, c'est à dire 2014_10_12_000000_create_users_table.php

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

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->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');
    }
}
```

Je vous laisse 5 minutes pour l'analyser et me dire ce que vous avez constaté.

- Que fait la fonction up() ?
- Que fait la fonction down() ?

4.2 Options de migration

Voici les options de migration:

- php artisan migrate => Exécute toutes les migrations qui n'ont pas encore été exécutées du répertoire database\migrations
- php artisan migrate:rollback => Revient à l'état précédent l'actuelle migration.
- php artisan migrate:refresh => Fait un rollback de toutes les migrations via la méthode down() de chaque migration. Ensuite relance les migrations. ATTENTION A NE JAMAIS EXECUTER EN PRODUCTION !!!!
- php artisan migrate:fresh => Supprime toutes les tables et relance toutes les migrations. ATTENTION A NE JAMAIS EXECUTER EN PRODUCTION !!!!
- php artisan migrate:status => indique le status des migrations: exécutées ou non.

4.3 Valeur par défaut

Dans la fonction up() d'une migration on peut avoir des champs avec une valeur par défaut. Par exemple, une table employe avec un champ boolean 'renvoye' à false. En effet, normalement un nouvel employé ne peut être déjà renvoyé. 😀 Donc on peut mettre par défaut à false la valeur.

```
public function up()
{
    Schema::create('employe', function (Blueprint $table) {
        $table->id();
        $table->string('nom');
        $table->string('email')->unique();
        $table->string('password');
        $table->boolean('renvoye')->default(false);
        $table->timestamps();
    });
}
```

5. Eloquent

Eloquent est l'ORM utilisé par Laravel:

> Un mapping objet-relationnel (en anglais object-relational mapping ou ORM) est un type de programme informatique qui se place en interface entre un programme applicatif et une base de données relationnelle pour simuler une base de données orientée objet.
>
> Ce programme définit des correspondances entre les schémas de la base de données et les classes du programme applicatif. On pourrait le désigner par là, « comme une couche d'abstraction entre le monde objet et monde relationnel ».
>
> Du fait de sa fonction, on retrouve ce type de programme dans un grand nombre de frameworks sous la forme de composant ORM qui a été soit développé, soit intégré depuis une solution externe.

Source Wikipedia.

5.1 Eloquent et le design pattern 'Active record'

Dans le design pattern Active record est utilisé pour faire correspondre une table à une classe. Les attributs de la table sont encapsulés dans une classe. De cette manière, un enregistrement d'une table correspond à un objet.

5.3 Tinker

Tinker veut dire littéralement "bidouilleur". Donc c'est un outil qui permet de bidouiller du code dans son contexte en ligne de commande.
L'intérêt est de ne pas mettre du code de tests dans notre application.
C'est rapide et on ne perd pas trop de temps à tester via un navigateur.

5.3 Création d'un user via tinker

Tinker est très utile quand vous voulez interagir avec votre base de données très rapidement sans faire de programme. Et il est parfait pour vous présenter Eloquent et le design patter Active record.

Nous allons créer un premier user dans notre base de données avec tinker:

```
php artisan tinker
//Nouvelle instance de User
$user = new App\Models\User;
//On remplit les attributs name, emai, password
$user->name='Piette';
$user->email='johnny.piette@gmail.com';
$user->password=bcrypt('!password');
//On sauve notre objet dans la base de données :)
$user->save();
//Affichons le contenu de notre objet après save()
$user;
```

Résultat de $user dans tinker:

```
=> App\Models\User {#3510
     name: "Piette",
     email: "johnny.piette@gmail.com",
     #password: "$2y$10$Yl/YfrtiykRVbnghPhuj8.iKB/s16gPJNnpya0g6niIPQ769QaFam",
     updated_at: "2021-10-26 01:30:03",
     created_at: "2021-10-26 01:30:03",
     id: 1,
   }
```

Ne fermez pas votre console tinker, nous allons avoir besoin plus loin de $user.

Vérifions que l'utilisateur a bien été créé et qu'il correspond à $user:

```
mysql -u root -p
use cupcake;
select * from users;
exit;
```

Résultat:

```
+----+--------+-------------------------+------------------+----------------------------
| id | name   | email                   | email_verified_at | password
+----+--------+-------------------------+------------------+----------------------------
|  1 | Piette | johnny.piette@gmail.com | NULL              | $2y$10$Yl/YfrtiykRVbnghPhuj8
+----+--------+-------------------------+------------------+----------------------------
```

Et oui, ça correspond ! 😀

Notez au passage que l'on a utilisé bcrypt qui a crypté notre mot de passe car dans une DB, il ne faut jamais stocker un mot de passe en clair: just a reminder. ☺

5.4 Mise à jour de User via tinker

Nous allons changer le nom de l'objet $user:

```
-- On change la valeur de l'attribut 'name'
$user->name='Defalque';
$user->save();
$user;
```

Résultat:

```
=> App\Models\User {#3510
     name: "Defalque",
     email: "johnny.piette@gmail.com",
     #password: "$2y$10$Yl/YfrtiykRVbnghPhuj8.iKB/s16gPJNnpya0g6niIPQ769QaFam",
     updated_at: "2021-10-26 01:42:20",
     created_at: "2021-10-26 01:30:03",
     id: 1,
   }
```

5.5 méthodes find, all

Nous allons avant créer un nouveau User pour cette partie:

```
//Nouvelle instance de User
$user = new App\Models\User;
//On remplit les attributs name, emai, password
$user->name='Piettus';
$user->email='djoni.piettus@gmail.com';
$user->password=bcrypt('!password');
//On sauve notre objet dans la base de données :)
$user->save();
```

On va rechercher l'utilisateur avec l'id = 1 et ensuite id=2. Enfin, on affichera tous les utilisateurs.

- Utlisateur avec l'id = 1 et ensuite l'id = 2

  ```
  App\Models\User::find(1);
  App\Models\User::find(2);
  ```

- Afficher tous les Users

  ```
  App\Models\User::all();
  ```

  Résultat de tous les Users:

```
=> Illuminate\Database\Eloquent\Collection {#4446
    all: [
    App\Models\User {#4445
        id: 1,
        name: "Piette Jacques",
        email: "johnny.piette@gmail.com",
        email_verified_at: null,
        #password: "$2y$10$Yl/YfrtiykRVbnghPhuj8.iKB/s16gPJNnpya0g6niIPQ769QaFa
        #remember_token: null,
        created_at: "2021-10-26 01:30:03",
        updated_at: "2021-10-26 01:42:20",
    },
    App\Models\User {#4444
        id: 2,
        name: "Piettus",
        email: "djoni.piettus@gmail.com",
        email_verified_at: null,
        #password: "$2y$10$oLp1C499P8IAlDlXjANiBe5X2H/qIDB7RKd3aj2/i47IWlegWV08
        #remember_token: null,
        created_at: "2021-10-26 02:03:00",
        updated_at: "2021-10-26 02:03:00",
    },
    ],
}
```

J'ai affiché le résultat de tous les Users car il est intéressant de remarquer qu'Eloquent nous indique que ce qui est retourné est une Collection: un tableau.

C'est beau non ? 😀

5.6 pluck méthode

Maintenant, j'ai envie de connaître tous les noms d'utilisateurs et rien d'autre.

```
User::pluck('name');
```

Résultat:

```
=> Illuminate\Support\Collection {#4391
     all: [
       "Piette Jacques",
       "Piettus",
     ],
   }
```

A nouveau une collection où l'on voit tous les noms.
Pour afficher le premier résultat de pluck, je peux le faire de deux manières:

```
    -- Première manière
    App\Models\User::pluck('name')[0];
    -- Seconde manière
    App\Models\User::pluck('name')->first();
```

5.7 Mettre à jour un enregistrement

La mise à jour peut se faire de deux manières soit via la méthode update() ou la méthode save().

5.7.1 via la méthode save()

Le principe est simple, on récupère l'enregistrement et on met à jour les attributs qui doivent être changés.
Ensuite, on utilise la méthode save() et c'est mis à jour.

```
$utilisateur = App\Models\User::find(1); //On recherhe l'utilisateur avec l'id = 1
$utilisateur->name='Pietta';
$utilisateur->email='johnny.pietta@gmail.com';
$utilisateur->save();
```

5.7.2 via la méthode update()

On récupère aussi l'enregistrement et dans la méthode update on donne un tableau associatif où l'on affectera les attributs de la classe.

```
App\Models\User::find(1)->update(['name'=>'Pietta','email'=>'johnny.pietta@gmail.com']);
```

5.8 Les conditions

Si on veut utiliser la clause where SQL, on peut le faire via la méthode where() que l'on va illustrer par différents exemples. La syntaxe pourra vous sembler étrange mais on s'y fait assez vite. Vous aurez plus d'infos dans la documentation Laravel.

Pour les prochains exemples on va réinitialier la base de données:

```
php artisan migrate:fresh
```

via tinker, nous allons ajouter 5 enregistrements grâce à la méthode insert de la classe User:

```
php artisan tinker
use App\Models\User;
    User::insert([
        [
            'name' => 'Piette',
            'email' => 'johnny.piette@gmail.com',
            'password' => bcrypt('L353nf4nt5')
        ],
        [
            'name' => 'Jacques',
            'email' => 'jacques.veronique@tutu.be',
            'password' => bcrypt('p@ssw0rd')
        ],
        [
            'name' => 'Jacques',
            'email' => 'jacques.andre@piette.be',
            'password' => bcrypt('p@ssw0rd')
        ],
        [
            'name' => 'Perezzz',
            'email' => 'perezzz.alvaro@bank.com',
            'password' => bcrypt('myc4s4')
        ],
        [
            'name' => 'jemina',
            'email' => 'trum.jemina@home.br',
            'password' => bcrypt('t01lets')
        ]
    ]);
```

5.8.1 Opérateurs de comparaison

En Laravel, pour effectuer des requêtes, on va voir comment utiliser les opérateurs SQL suivants:

- > et <
- >= et <=
- = et <>
- like
- and

- or
- between et not between
- in et notIn
- null et not null

Afficher les utilisateurs dont le mail est 'johnny.piette@gmail.com'. Dans l'exemple j'affecte le résultat de la méthode get() dans la variable $user;

```
$user = User::where('email','johnny.piette@gmail.com')->get();
//Ou bien mais plus long...
$user = User::where('email','=','johnny.piette@gmail.com')->get();
```

Afficher les utilisateurs dont l'id > 1

```
App\Models\User::where('id','>',1)->get();
```

Afficher les utilisateurs dont l'id >= 2

```
App\Models\User::where('id','>=',2)->get();
```

Afficher les utilisateurs dont l'id <= 4

```
App\Models\User::where('id','<=',4)->get();
```

Afficher les utilisateurs dont le nom n'est pas 'Pietta' et que le mail contient 'gmail.com'. On chaîne les where tout simplement. 😀

```
App\Models\User::where('name','<>','Pietta ')->where('email','like','%gmail.com')->get();
```

Afficher les utilisateurs dont le nom est 'Pietta' ou 'Jacques'. Pour le 'or' on va chaîner avec un orWhere:

```
App\Models\User::where('name','Pietta ')->orWhere('name','Jacques')->get();
```

Afficher les utilisateurs dont l'id est compris entre 2 et 4:

```
App\Models\User::whereBetween('id',['2','4 '])->get();
```

Afficher les utilisateurs dont l'id n'est pas compris entre 2 et 4:

```
App\Models\User::whereNotBetween('id',['2','4 '])->get();
```

Afficher les utilisateurs dont l'id a une des valeurs suivantes:1,3 ou 5

```
App\Models\User::whereIn('id',['1','3','5'])->get();
```

Afficher les utilisateurs dont l'id n'a pas une des valeurs suivantes:1,3 ou 5

```
App\Models\User::whereNotIn('id',['1','3','5'])->get();
```

6. Création de la table Cupcake

```
php artisan make:migration create_cupcakes_table
```

Le fichier de migration a été créé dans le répertoire database\migrations
Ouvrez ce fichier et modifiez-le de la sorte:

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

class CreateCupcakesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('cupcakes', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('description');
            $table->string('imageName');
            $table->decimal('price');
            $table->integer('stock');
            $table->timestamps();
        });
    }

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

Laravel a compris grâce au verbe create qu'il doit créer une table cupcakes et a ajouté tout seul le nom de la table. De base, il ajoute un id et un timestamps.

Lançons notre migration:

```
php artisan migrate
```

La table cupcakes a été créée avec les champs correspondants à ceux que nous avons définis.

7. Création du modèle Cupcake

Nous avons créé notre table cupcakes. Maintenant, nous allons créer la classe Cupcake qui permettra d'appliquer le design pattern Active record.

```
php artisan make:model Cupcake
```

La classe Cupcake a été créée dans le répertoire App\Models

On peut aussi créer un modèle et une migration en même temps:

```
php artisan make:model Tutu -m
```

8. Ajout d'un cupcake via Tinker

```
php artisan tinker
$cupcake = new App\Models\Cupcake;
$cupcake->name = 'Le Pinky';
$cupcake->description = 'Bla bla bla bla...';
$cupcake->imageName = 'cupcake1.jpg';
$cupcake->stock = 10;
$cupcake->price = 2.5;
$cupcake->save();
```

Et voilà ! Notre object $cupcake est devenu **persistant** dans le base de données. D'un objet, il est devenu un enregistrement dans la table cupcakes.

Si notre programme se termine, l'objet disparait de la mémoire mais l'enregistrement, lui restera/ persistera dans la base de données.

9. Stocker un Mot de passe en DB

Ceci est un point ajouté à la demande d'un étudiant:"Monsieur comment faire la gestion des mots de passe en Laravel ?"

Premièrement, il ne faut pas stocker votre mot de passe en clair dans une base de données. Je l'ai fait il y a très très longtemps par le passé mais c'était très mauvais, je dois le reconnaître. ☺
De nos jours, il serait suicidaire de le faire...

Pour résumer très rapidement, de votre mot de passe on fait des calculs et on obtient un résultat. Mais on ne peut pas faire l'opération inverse pour retrouver le mot de passe.

Pour stocker votre mot de passe, utilisez simplement la commande password_hash de php. Pour Laravel, utilisez soit le helper bcrypt ou bien fonction statique make de la classe Hash.

Avec password_hash en php

```
$password = password_hash('votremotdepasse',PASSWORD_DEFAULT);
```

- PASSWORD_DEFAULT (='2y'): utilise l'algorithme bcrypt par défaut. Pourrait peut-être changé dans le futur si un algo plus robuste est implémenté par défaut.
- PASSWORD_BCRYPT (='2y'): utilise l'algorithme bcrypt.
- PASSWORD_ARGON2I (='argon2i'): utilise l'algorithme argon2i.
- PASSWORD_ARGON2ID (='argon2id'): utilise l'algorithme argon2id.

password_algos() vous retournera les différents algorithmes de cryptages que vous pouvez utilisez pour vos mots de passe.

Le hashage de votre mot de passe pourrait être: $2y10
MV6iXclPOt0R6vFh0xPoq.xQGLl0A7upIXXMdaYc1.7MCT7K48era
Mais si vous utilisez bcrypt avec le même mot de passe, le résultat de $password changera.
Donc il ne faut pas croire qu'à un mot de passe crypté sera toujours le même.

Donc à un mot de passe peut correspondre plusieurs hashs.
Mais à un hash devrait correspondre un seul mot de passe. Sauf en cas de collision. C'était le cas de certains algorithmes de hashages qui avaient ce type de bug: sha1, md5. Linkedin en 2012 stockait ses mots de passe avec du sha1 et il a fallu 3 jours pour récupérer 90% des mots

de passe. 😀

Avec Hash::make de Laravel

```
use Illuminate\Support\Facades\Hash;
$password = Hash::make('votremotdepasse');
```

Avec le helper bcrypt de Laravel

```
use Illuminate\Support\Facades\Hash;
$password = bcrypt('votremotdepasse');
```

Pour Hash::make, par défaut, Laravel va utiliser comme driver de hashage 'bcrypt'. On pourrait aussi utiliser argon et argon2id.

En regardant les sources de Laravel, j'ai vu que c'était indiqué dans le fichier: config\hashing.php.

10. Vérifier un Mot de passe

Pour le mot de passe, on ne décrypte pas un mot de passe avec une fonction.

Donc si vous avez perdu le mot de passe, l'administrateur ne pourra pas vous le fournir s'il a été crypté. Il devra vous en faire un nouveau qui sera lui aussi crypté. Ou bien, vous recevrez par email un lien vers un formulaire où vous pourrez mettre un nouveau mot de passe. Ca dépend du contexte...

On a en php la fonction password_verify et en Laravel la fonction Hash::check

Avec password_verify en php

```
$password = 'votremotdepasse';
if(password_verify($password,'$2y$10$MV6iXclPOt0R6vFh0xPoq.xQGLl0A7upIXXMdaYc1.7MCT7K48era
    echo "Le mot de passe est correct";
} else {
    echo "Le mot de passe n'est pas correct !";
}
```

Avec Hash::check en Laravel

```
use Illuminate\Support\Facades\Hash;
$password = 'votremotdepasse';
if(Hash::check($password,'$2y$10$MV6iXclPOt0R6vFh0xPoq.xQGLl0A7upIXXMdaYc1.7MCT7K48era')){
    echo "Le mot de passe est correct";
} else {
    echo "Le mot de passe n'est pas correct !";
}
```

11. Ajouter une colonne

On va ajouter une colonne 'arrete' pour indiquer qu'un produit n'est plus produit.

On crée la migration:

```
php artisan make:migration add_arrete_to_cupcakes --table=cupcakes
```

On modfie la migration

```
    public function up()
    {
        Schema::table('cupcakes', function (Blueprint $table) {
            $table->boolean('arrete');
        });
    }

    public function down()
    {
        if (Schema::hasColumn('cupcakes', 'arrete')) {
            Schema::table('cupcakes', function (Blueprint $table) {
                $table->dropColumn('arrete');
            });
        }
    }
```

On exécute la migration

```
php artisan migrate
```

12. Mettre à jour une colonne

Nous aurons besoin du package doctrine/dbal pour pouvoir mettre à jour. Nous verrons pourquoi après un essai de modification de la colonne 'nom' qu'un migrate va provoquer une erreur.

Création de la migration

```
php artisan make:migration update_stock_to_cupcakes --table=cupcakes
```

Modification de la migration

```
    public function up()
    {
        Schema::table('cupcakes', function (Blueprint $table) {
            $table->integer('stock')->default(0)->change();
        });
    }

    public function down()
    {
        if (Schema::hasColumn('cupcakes', 'stock')) {
            Schema::table('cupcakes', function (Blueprint $table) {
                $table->integer('stock')->default(NULL)->change();
            });
        }
    }
```

Exécution de la migration

```
php artisan migrate
```

Et bardaf c'est l'embardée, on reçoit le message d'erreur suivant:'Changing columns for table "%s" requires Doctrine DBAL. Please install the doctrine/dbal package.'

Le message est on ne peut plus clair: on doit installer le package doctrine/dbal si on veut modifier une colonne.

Installation de la dépendance via composer

```
composer require doctrine/dbal
```

Maintenant votre migration devrait fonctionner.

13. Supprimer une colonne

Pour l'exercice, on a va ajouter une colonne inutile: le prix TVA comprise. En effet, on ne met pas de colonne qui pourrait être calculée en DB. C'est une perte inutile en base de données. Si vous 1 milliard d'enregistrements, vous avez 1 milliard de colonnes inutiles...

On va faire la migration

```
php artisan make:migration add_pricevat_to_cupcakes --table=cupcakes
```

Modification de la migration:

```
    public function up()
    {
        Schema::table('cupcakes', function (Blueprint $table) {
            $table->decimal('pricevat');
        });
    }

    public function down()
    {
        if (Schema::hasColumn('cupcakes', 'pricevat')) {
            Schema::table('cupcakes', function (Blueprint $table) {
                $table->dropColumn('pricevat');
            });
        }
    }
```

On exécute la migration

```
php artisan migrate
```

Comme cette colonne est inutile, on va la supprimer.
Deux manières:

- On fait une nouvelle migration avec le dropColumn dans la méthode up() et ajouter une colonne dans la méthode down();
- On fait un rollback et comme nous appelons la méthode dropColumn dans la méthode down(). Ensuite on supprime le fichier de migration. (A voir si c'est une bonne méthode)

Créons la migration

```
php artisan make:migration drop_pricevat_to_cupcakes --table=cupcakes
```

Modifions la migration

```
    public function up()
    {
        if (Schema::hasColumn('cupcakes', 'pricevat')) {
            Schema::table('cupcakes', function (Blueprint $table) {
                $table->dropColumn('pricevat');
            });
        }
    }

    public function down()
    {
        if (Schema::hasColumn('cupcakes', 'pricevat')) {
            Schema::table('cupcakes', function (Blueprint $table) {
                $table->decimal('pricevat');
            });
        }
    }
```

On exécute la migration

```
php artisan migrate
```

14. Les relations

Dans une base de données, il y a des tables. Et il existe aussi des liens entre ces tables qu'on appelle relations. Relation de 1 à plusieurs (One to many), de plusieurs à plusieurs (many to many).

Nous allons voir comme imprémenter ces relations en Laravel.

Avant d'aller plus loin, nous allons créer le modèle Categorie et la migration qui va créer la table categories en une seule commande. Nnous l'avons déjà fait précédemment.

```
php artisan make:model Category --m
```

Il est intéressant de noter que la classe Category est au singulier et que Laravel a mis au pluriel Category quand il a créé la migration: create_categories_table.php et lors de l'appel de la fonction statique create.

Modifions cette migration avec:

```
class CreateCategoriesTable extends Migration
{
    public function up()
    {
        Schema::create('categories', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('categories');
    }
}
```

Exécutons la migration:

```
php artisan migrate
```

14.1 One to many

Nous prendrons le cas de notre table cupcakes. A un cupcake va correspondre une catégorie. Et une catégorie correspond à un ou plusieurs cupcakes (voire aucun).

Nous pourrons par exemple voir directement d'une categorie quels sont les cupcakes liés:

```
//On prend par exemple la première catégorie de la table Categories
$cat = Category::first();
$cupcakes = $cat->cupcakes;
//Ou en une ligne
$cupcakes = Category::first()->cupcakes;
```

Ou à l'inverse voir à quelle catégorie appartient un cupcake:

```
$cat = Cupcake::first()->category;
```

Mais nous allons construire la mécanique avant d'arriver à cela...

Créons donc un lien entre la table cupcakes et la table categories:

```
php artisan make:migration add_categoryid_to_cupcakes
```

Modifions cette migration pour ajouter la clef étrangère:

```
class AddCategoryidToCupcakes extends Migration
{
    public function up()
    {
        Schema::table('cupcakes', function (Blueprint $table) {
            $table->foreignId('category_id')->constrained();
        });
    }

    public function down()
    {
        if (Schema::hasColumn('cupcakes', 'category_id')) {
            Schema::table('cupcakes', function (Blueprint $table) {
                $table->dropColumn('category_id');
            });
        }
    }
}
```

On va exécuter la migration:

```
php artisan migrate
```

Voici le code SQL généré par Larvel pour la version finale de la table cupcakes. On voit qu'il a bien ajouté la clef étrangère:

```
CREATE TABLE `cupcakes` (
  `id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `description` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `imageName` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `stock` int NOT NULL,
  `created_at` timestamp NULL DEFAULT NULL,
  `updated_at` timestamp NULL DEFAULT NULL,
  `category_id` bigint unsigned NOT NULL,
  PRIMARY KEY (`id`),
  KEY `cupcakes_category_id_foreign` (`category_id`),
  CONSTRAINT `cupcakes_category_id_foreign` FOREIGN KEY (`category_id`) REFERENCES `categ
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

C'est beau non ? 😄

Maintenant, il faut que nous ajoutions une fonction dans la classe Category et une fonction dans la classe Cupcake.

Nous devons définir le fait qu'un Cupcake a une Categorie (belongsTo). Et qu'une Categorie a un ou plusieurs Cupcakes (HasMany).

La classe Cupcake:

```
class Cupcake extends Model
{
    use HasFactory;

    public function categorie(){
        return $this->belongsTo(Categorie::class);
    }

}
```

La classe Category:

```php
class Category extends Model
{
    use HasFactory;


    public function cupcakes(){
        return $this->hasMany(Cupcake::class);
    }
}
```

Créons des catégories et des cupcakes via tinker

```php
use App\Models\Category;
use App\Models\Cupcake;

Category::insert([
    [
        'name' => 'Moelleux'
    ],
    [
        'name' => 'Croquant'
    ],
    [
        'name' => 'Glacé'

    ]
]);

Cupcake::insert([
    [
        'name' => 'Le Pinky',
        'description' => 'Le pinky ce cupcake rose et délicieux !',
        'imageName' => 'pinky.png',
        'price' => 2.5,
        'stock' => 50,
        'category_id' => 1
    ],
    [
        'name' => 'Le Brown',
        'description' => 'Sa couleur brune vient d\'un chocolat du Brésil où les grains or
        'imageName' => 'brown.png',
        'price' => 3,
        'stock' => 175,
        'category_id' => 2
    ],
    [
        'name' => 'Le Caramel',
        'description' => 'Fait à partir de Caramel beurre salé fait maison, ravira les pap
        'imageName' => 'caramel.png',
        'price' => 2.75,
        'stock' => 99,
```

```
        'category_id' => 3
    ],
    [
        'name' => 'Le Blue',
        'description' => 'Le pink ce cupcake rose et délicieux !',
        'imageName' => 'blue.png',
        'price' => 2.5,
        'stock' => 12,
        'category_id' => 2
    ],
    [
        'name' => 'Le Rainbow',
        'description' => 'Une explosion de goûts et de couleur viendont titiller vos sens
        'imageName' => 'rainbow.png',
        'price' => 2,
        'stock' => 33,
        'category_id' => Category::where('name','Croquant')->first()->id
    ]
]);
```

Regardez le dernier cupcake ajouté dans la méthode statique insert de la classe Cupcake. Maintenant, amusons-nous un peu 😀

J'ai envie de savoir quels Cupcakes font partie de la catégorie 'Croquant'

```
Category::find(2)->cupcakes;
```

Laravel a fait pour vous une requête qui devrait ressembler à ceci:

```
SELECT cupcakes.*
FROM cupcakes
INNER JOIN categories ON cupcakes.category_id = categories.id
WHERE categories.id = 2;
```

J'ai envie de savoir le nom de la Catégorie du capcake dont l'id = 3

```
Cupcake::find(3)->category->name
```

Laravel a fait pour vous une requête qui devrait ressembler à ceci:

```sql
SELECT categories.name
FROM categories
INNER JOIN cupcakes ON categories.id = cupcakes.category_id
WHERE cupcakes.id = 3;
```

Il y a juste une chose à savoir c'est que la méthode category de la classe Cupcake est appelée comme un attribut sans les ().
Idem pour la méthode cupcakes de la classe Category qui est appelée sans les ().

J'espère que vous avez vu la facilité avec laquelle nous avons pu faire nos relation 'One to Many' avec Laravel 😀

⬅Revenir à la théorie.
