Cours LARAVEL partie 1
Gérer les données avec l'ORM ELOQUENT

Sommaire

C’est quoi un ORM

ELOQUENT est un ORM. Un ORM **(Object-Relational Mapping**) est un type de programme informatique qui se place en interface entre un programme applicatif et une base de données relationelle 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**.

Du fait de sa fonction, on retrouve ce type de programme dans un grand nombre de frameworks sous la forme de composant ORM.

Eloquent est inclus avec Laravel, il fournit une implémentation simple pour travailler avec votre base de données.

Chaque table de base de données a un modèle correspondant qui est utilisé pour interagir avec cette table.

Les modèles vous permettent de rechercher des données dans vos tables, ainsi que d'insérer de nouveaux enregistrements dans la table.

Créer la base de données

On crée une base de données **emploi** dans phpMyadmin

puis On renseigne le fichier **.env** en conséquence :

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

Créer un modèle et sa migration

Laravel pourrait communiquer directement avec les tables de la base de données. Cependant, il est plus intéressant d'utiliser des **modèles**, c'est-à-dire une représentation objet de chacune des tables.

A chaque modèle on associera une classe **migration**. 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.

Pour créer un modèle et sa migration on utilise une commande artisan :

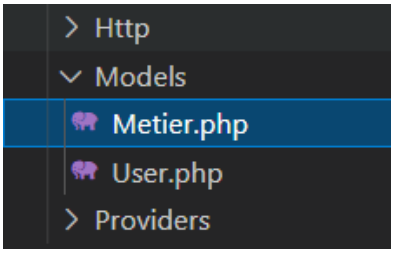
php artisan make:model Metier –-migration

Ici on vient de créer le **modèle Metier** en même temps que la classe de migration qui va gérer la table associée au modèle.

Nous avons donc obtenu 2 classes :

Metier.php (dans Models) et

Create_metiers_tables.php (dans migrations)




```
PS C:\xampp\htdocs\digijobs> php artisan make:model Metier --migration
Model created successfully.
Created Migration: 2022_03_29_113435_create_metiers_table
PS C:\xampp\htdocs\digijobs>
```

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Metier extends Model
{
    use HasFactory;
}
```

Le modèle

Le modèle **Metier** qu'on a créé est vide au départ

(il n'y a que use HasFactory).

On va se contenter de prévoir l'assignement de masse avec la propriété **$fillable** :

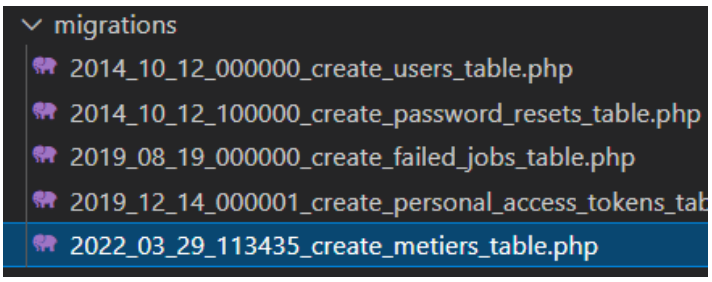
protected $fillable =

['nom', 'description', 'image', 'salaire', 'niveau'];

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Metier extends Model
{
    use HasFactory;
    protected $fillable = ['nom', 'description','image','salaire','niveau'];

}
```

La migration

Classe créée dans database->migrations pour gérer la création et la destruction de la table metiers dans phpMyadmin



On va rajouter des colonnes dans la fonction up de la classe migration qui gère la table **metiers**,

le nom et la description du metier

```
public function up()
{
Schema::create('metiers', function (Blueprint $table) {
$table->id();
$table->string('nom');
$table->text('description');
$table-> string ('image');
$table->text('salaire');
$table->string('niveau');
$table->timestamps();
```

Ensuite on lance la migration qui va créer physiquement la table metiers dans la base de données mySql :

php artisan migrate

```php
return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('metiers', function (Blueprint $table) {
            $table->id();
            $table->string('nom');
            $table->text('description');
            $table->string('image');
            $table->integer('salaire');
            $table->string('niveau');

            $table->timestamps();
        });
    }
```

On a donc les champs :

id : entier auto-incrémenté qui sera la clé primaire de la table,

nom : string pour le nom du métier,

description : text pour la description du métier,

image : string pour le chemin vers l'image

salaire : integer salaire moyen du métier,

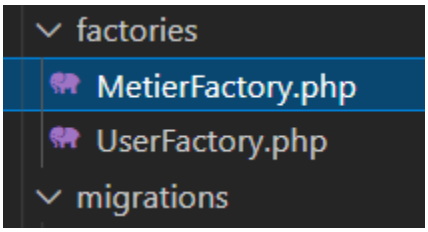
niveau : niveau requis pour le métier,

created_at et updated_at créés par la méthode **timestamps**

La population d'une table en ipsum lorem (ajout de valeurs fake)

Pour nos essais on va remplir un peu la table metiers qui se trouve dans la base de données « emploi » de phpMyadmin, Pour cela, on va créer un fichier factory :

php artisan make:factory MetierFactory --model=Metier



Afin de générer l'ipsum lorem dans la table metiers, il y a 2 fichiers à modifier :

MetierFactory.php.et **DatabaseSeeder.php**

Il faut rajouter du code dans la méthode **definition()** de **MetierFactory.php**.

Ce code va solliciter des fonctions de l'objet faker qui insére des valeurs dans la table metiers

```
public function definition()
{
return [
    'nom' => $this->faker->sentence(4, true),
    'description' => $this->faker->paragraph(),
    'image' => $this->faker->sentence(1, true),
    'salaire' => $this->faker-
>numberBetween(30000,80000) ;
    'niveau' => $this->faker->sentence(3, true),
    ];}
     }
```

```php
class MetierFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition()
    {
        return [
            'nom' => $this->faker->sentence(4, true),
            'description' => $this->faker->paragraph(),
            'image' => $this->faker->sentence(1, true),
            'salaire' => $this->faker->numberBetween(0, 10000),
            'niveau' => $this->faker->sentence(1, true),
        ];
    }
}
```

Il faut également changer le code de la fonction **run()** du fichier **DatabaseSeeder.php** qui se trouve dans le dossier seeds. On y indique le nombre d'enregistrements que l'on veut créer dans la table

```
public function run()
{
Metier::factory(10)->create();
}
```

```php
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        Metier::factory(10)->create();
    }
}
```

Il ne reste plus qu'à lancer la population :

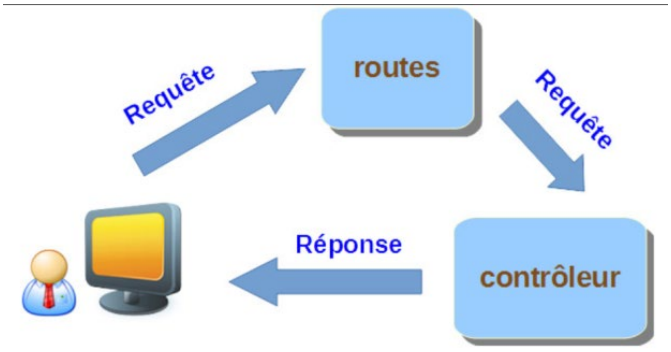
php artisan db:seed

Si tout va bien on se retrouve avec 10 enregistrements dans la table metiers

Le contrôleur

La tâche d'un contrôleur est de réceptionner une requête (qui a déjà été sélectionnée par une route) et de définir la réponse appropriée, rien de moins et rien de plus.

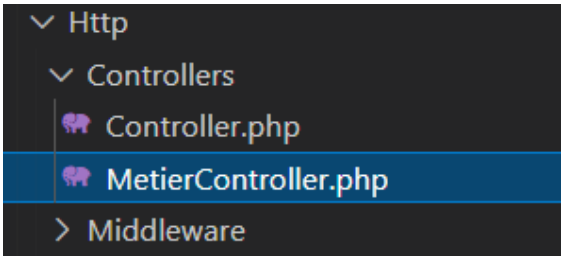
Voici une illustration du processus :



On va maintenant créer un contrôleur de ressource avec Artisan :

php artisan make:controller MetierController --resource

Un nouveau controller nommé MetierController.php a été créé dans http :



Découvrons le code de cette classe , il comprend 7 méthodes qui couvrent la gestion complète des Metiers

index : pour afficher la liste des métiers,

create : pour afficher et envoyer le formulaire de création d'un nouveau métier,

store : La méthode create submit les données à la méthode store qui les insère dans la base de donnée.

show : pour afficher les détails d'un métier, sans pouvoir modifier ses informations

edit : Affiche le formulaire contenant les données d'un metier en mode modifiables et permet d'envoyer formulaire pour la modification de ce métier,

update : La méthode edit submit les données à la méthode update qui fait la modification dans la base de données

destroy : pour supprimer un métier.

Les routes

Pour créer toutes les routes il suffit de rajouter ces lignes de code dans le fichier route/web.php :

```
use App\Http\Controllers\MetierController;
Route::resource('metiers', MetierController::class);
```

On va vérifier ces routes avec Artisan : **php artisan route:list**

```
GET|HEAD   metiers ..... metiers.index › MetierController@index
POST       metiers ..... metiers.store › MetierController@store
GET|HEAD   metiers/create ..... metiers.create › MetierController@create
GET|HEAD   metiers/{metier} ..... metiers.show › MetierController@show
PUT|PATCH  metiers/{metier} ..... metiers.update › MetierController@update
DELETE     metiers/{metier} ..... metiers.destroy › MetierController@destroy
GET|HEAD   metiers/{metier}/edit ..... metiers.edit › MetierController@edit
```

On constate que 7 routes on été créées. avec chacune une méthode et une url, qui pointent sur les 7 méthodes du contrôleur : index, create, store, show, edit, update, destroy.

Notez également que chaque route a aussi un nom qui peut être utilisé par exemple pour une redirection.

La liste des métiers (getAll = index())

La route

La liste des métiers correspond à cette route :

```
GET|HEAD        metiers ..... metiers.index › MetierController@index
```

Il faut ajouter la route qui indique la ressource ciblée dans web.php :

Route::resource('metiers', MetierController::class);

```php
//ROUTE PAR DEFAUT QUAND ON ARRIVE SUR LA PAGE D ACCUEIL
Route::get('/', function () {
    return view('template');
});

//ROUTE POUR RECUPERER TOUS LES METIERS
//ici on indique en premier argument de resource
//la table concernée dans la base de données
//et en second argument le nom du controller concerné
Route::resource('metiers', MetierController::class);
```

Le contrôleur

Dans le contrôleur c'est la méthode **index** qui est concernée. On va donc la coder :

```php
use App\Models\Metier;

class MetierController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $metiers = Metier::all();
        return view('ViewMetiers',compact('metiers'));
    }
```

...

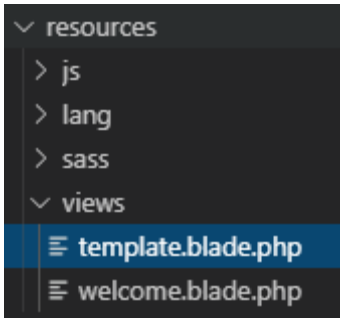
On va chercher tous les métiers avec la méthode **all** du modèle, on appelle la vue **ViewMetiers** et on lui transmet l'array de métiers.

Le template au niveau de la vue

Laravel s'occupe essentiellement du côté serveur et n'impose rien côté client, même s'il propose des choses.

Autrement dit on peut utiliser Laravel avec n'importe quel système côté client. Pour notre exemple je vous propose d'utiliser Bulma pour la mise en forme.

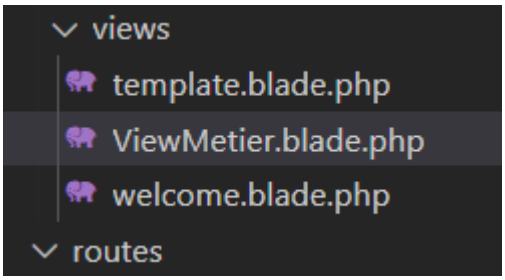
Voici un template Bulma qui va nous servir pour toutes nos vues :




```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<nom>Metiers</nom>
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.9.0/css/bulma.min.css">
@yield('css')
</head>
<body>
<main class="section">
<div class="container">
@yield('content')
</div>
</main>
</body>
</html>
```

La vue ViewMetiers pour afficher tous les métiers

On crée la vue **ViewMetiers** :



Avec ce code :

```
@extends('template')

@section('content')
@if(session()->has('info'))
<div class="notification is-success">
{{ session('info') }}
</div>
@endif
 <div class="card">
<header class="card-header">
<p class="card-header-nom">metiers</p>
</header>
<div class="card-content">
<div class="content">

<div class="card">
      <header class="card-header">
         <p class="card-header-title">Metiers</p>
         <a class="button is-info" href="{{
route('metiers.create') }}">Créer un metier</a>
      </header>
      <div class="card-content">
         <div class="content">

         <table class="table is-hoverable">
<thead>
<tr>
<th>#</th>
<th>Nom</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</tr>
<tbody>
@foreach($metiers as $metier)
<tr>
<td>{{ $metier->id }}</td>
<td><strong>{{ $metier->nom }}</strong></td>
<td><a class="button is-primary" href="{{
route('metiers.show', $metier->id)
}}">Voir</a></td>
<td><a class="button is-warning" href="{{
route('metiers.edit', $metier->id)
}}">Modifier</a></td>
<td>
<form action="{{ route('metiers.destroy', $metier-
>id) }}" method="post">
@csrf
@method('DELETE')
<button class="button is-danger"
type="submit">Supprimer</button>
</form>
</td>
</tr>
@endforeach
</tbody>
</table>
</div>

</div>
</div>
@endsection
```

Trier les métiers

Pour le moment les films sont listés dans l'ordre de leur identifiant, ce qui n'est pas très heureux. Il serait plus judicieux d'avoir la liste dans l'ordre alphabétique des noms de films. Pour le faire il suffit d'intervenir dans le contrôleur :

```
public function index()
{
$metiers = Metier::oldest('nom')->paginate(5);
 }
```

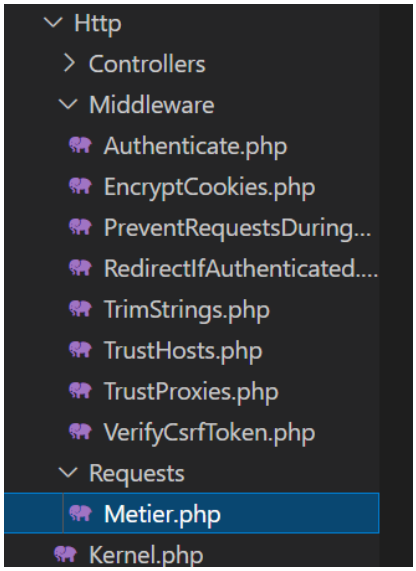
Il faut aussi rajouter un footer dans la view pour voir la pagination :

```
<footer class="card-footer is-centered">
    {{ $metiers->links() }}
 </footer>
```

La validation de formulaires

On va créer une requête de formulaire pour la modification d'un métier , elle sera nécessaire pour la méthode update du controller.

php artisan make:request Metier



Le fichier Metier.php dans Request sert à poster les champs du formulaire et à vérifier leur validité

On prévoie comme règles :

nom : le champ est obligatoire (**required**), ça doit être du texte (**string**), le nombre maximum de caractères (**max**) doit être de 200

description : ça doit être du texte (**string**),

image : ça doit être du texte (**string**),

salaire : ça doit être du numérique (**numeric**),

niveau : ça doit être du texte (**string**),

On pourra injecter cette classe dans les méthodes du contrôleur.

```php
class Metier extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {

        return [
            'nom' => ['required', 'string', 'max:150'],
            'description' => ['required', 'string', 'max:2500'],
            'image' => ['required', 'string', 'max:250'],
            'salaire' => ['required', 'numeric'],
            'niveau' => ['required', 'string', 'max:100']
        ];
    }
}
```

Créer un métier

Les routes

2 routes correspondent à la création d'un métier :

une route GET appeler la méthode create du contrôleur

une route POST appeler la méthode store du contrôleur

```
use App\Http\Controllers\ MetierController;

Route::get('metier', [MetierController::class, 'create'])->name('metier.create');

Route::post('metier', [MetierController::class, 'store'])->name('metier.store');
```

(à noter que ces routes sont redondantes avec celle de la ressource qui pointe sur toutes les méthodes du controller)

Le contrôleur

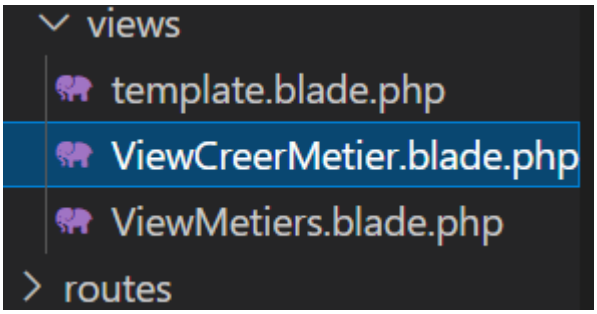
Dans le contrôleur ce sont les méthodes create et store qui sont concernées. On va donc les coder :

```
use App\Http\Requests\Metier as MetierRequest;
...
public function create()
{
return view('ViewCreerMetier');
}
```

```
public function store(Request $metierRequest)
{
Metier::create($metierRequest->all());
return redirect()->route('metiers.index')->with('info', 'Le metier a bien été créé');
}
```

La vue ViewCreerMetier pour afficher le formulaire de création d'un métier

On crée la vue **ViewCreerMetier** :



Avec ce code :

```
@extends('template')
@section('content')
  <div class="card">
    <header class="card-header">
      <p class="card-header-nom">Création d'un métier</p>
    </header>
    <div class="card-content">
      <div class="content">
        <form action="{{ route('metiers.store') }}" method="POST">
            @csrf
            <div class="field">
              <label class="label">Nom du métier</label>
              <div class="control">
                <input class="input @error('nom') is-danger @enderror" type="text" name="nom" value="{{ old('nom') }}" placeholder="Titre du métier">
              </div>
              @error('nom')
                <p class="help is-danger">{{ $message }}</p>
              @enderror
            </div>
            <div class="field">
              <label class="label">Salaire moyen</label>
              <div class="control">
                <input class="input" type="number" name="salaire" value="{{ old('salaire') }}" min="30000" max="80000">
              </div>
              @error('salaire')
                <p class="help is-danger">{{ $message }}</p>
              @enderror
            </div>
            <div class="field">
              <label class="label">Niveau d'études</label>
              <div class="control">
                <input class="input @error('niveau') is-danger @enderror" type="text" name="niveau" value="{{ old('niveau') }}" placeholder="niveau du métier">
              </div>
              @error('nom')
                <p class="help is-danger">{{ $message }}</p>
              @enderror
            </div>            <div class="field">
              <label class="label">Description</label>
              <div class="control">
                <textarea class="textarea" name="description" placeholder="Description du métier">{{ old('description') }}</textarea>
              </div>
              @error('description')
                <p class="help is-danger">{{ $message }}</p>
              @enderror
            </div>
            <div class="field">
              <div class="control">
                <button class="button is-link">Envoyer</button>
              </div>
            </div>
          </form>
        </div>
      </div>
    </div>
@endsection
```

Modifier un métier

Les routes

Pour la modification d'un métier on va avoir deux routes :

Ce sont les routes suivantes :

Route::get('metier', [MetierController::class, 'edit'])->name('metier.edit');

Route::post('metier', [MetierController::class, 'update'])->name('metier.update');

(à noter que ces routes sont redondantes avec celle de la ressource qui pointe sur toutes les méthodes du controller donc il n'est pas nécessaire de les écrire dans web.php si vous avez déjà la route « ressource »)

Le contrôleur

Dans le contrôleur ce sont les méthodes create et store qui sont concernées. On va donc les coder :

```
use App\Http\Requests\Metier as MetierRequest;
...
public function edit(Metier $metier)
{
return view('ViewModifierMetier', compact('metier'));
}
```

```
public function update(MetierRequest $metierRequest, Metier $metier)
{
$metier ->update($metierRequest ->all());
return redirect()->route('metiers.index')->with('info', 'Le métier a bien été modifié');
}
```

La vue ViewModifierMetier pour afficher le formulaire de modification d'un métier

On crée la vue **ViewModifierMetier, qui est la même que la vue ViewCreerMetier, mais on affiche des valeurs (de l'objet métier choisi) dans les value des input text. Et la route de l'action du formulaire est différente**

```
@extends('template')
@section('content')
<div class="card">
   <header class="card-header">
      <p class="card-header-nom">Modification d'un metier</p>
   </header>
   <div class="card-content">
      <div class="content">
         <form action="{{ route('metiers.update', $metier->id) }}" method="POST">
            @csrf
            @method('put')
            <div class="field">
               <label class="label">Nom du métier</label>
               <div class="control">
                  <input class="input @error('nom') is-danger @enderror" type="text" name="nom"
value="{{ old('nom', $metier->nom) }}" placeholder="Titre du metier">
               </div>
               @error('nom')
               <p class="help is-danger">{{ $message }}</p>
               @enderror
            </div>
            <div class="field">
               <label class="label">Salaire moyen</label>
               <div class="control">
                  <input class="input" type="number" name="salaire" value="{{ old('salaire',
$metier->salaire) }}" min="30000" max="80000">
               </div>
               @error('salaire')
               <p class="help is-danger">{{ $message }}</p>
               @enderror
            </div>
            <div class="field">
               <label class="label">Niveau d'études</label>
               <div class="control">
                  <input class="input @error('niveau') is-danger @enderror" type="text" name="niveau"
value="{{ old('niveau', $metier->niveau) }}" placeholder="niveau du métier">
               </div>
               @error('niveau')
               <p class="help is-danger">{{ $message }}</p>
               @enderror
            </div>
               <div class="field">
               <label class="label">Description</label>
               <div class="control">
                  <textarea class="textarea" name="description" placeholder="Description du metier">{{
old('description', $metier->description) }}</textarea>
               </div>
               @error('description')
               <p class="help is-danger">{{ $message }}</p>
               @enderror
            </div>
            <div class="field">
               <div class="control">
                  <button class="button is-link">Envoyer</button>
               </div>
            </div>
         </form>
      </div>
   </div>
</div>
@endsection
```

Supprimer un métier

Le contrôleur

Dans le contrôleur c'est la méthode **destroy** qui est concernée. On va donc la coder :

```
public function destroy(Metier $metier)
{
$metier ->delete();
return back()->with('info', 'Le film a bien été supprimé dans la base de données.');
}
```

C'est un peu brutal comme suppression il faudrait fournir un message de confirmation pour l'utilisateur. C'est un pop up à coder côté front

Par contre après la suppression il faut afficher quelque chose pour dire que l'opération s'est réalisée correctement. On voit qu'il y a une redirection avec la méthode **back** qui renvoie la même page. D'autre part la méthode **with** permet de flasher une information dans la session. Cette information ne sera valide que pour la requête suivante. Dans notre vue **index** on va prévoir quelque chose pour afficher cette information :

```
@section('content')
@if(session()->has('info'))
<div class="notification is-success">
{{ session('info') }}
</div>
@endif
<div class="card">
```

La directive **@if**(session()->has('info')) permet de déterminer si une information est présente en session, et si c'est le cas de l'afficher .