



## COURS FRAMEWORK WEB

### CHAPITRE 3: FRAMEWORK LARAVEL - LES BASES

Enseignantes responsables:  
Nahla Haddar & Amal Bouaziz

Audiences: D-LSI-ADBD  
Année-universitaire: 2023-2024

### *Plan*

- Le routage dans Laravel
- Les contrôleurs
- Les vues
- Syntaxe Blade

## *L'outil Artisan*

- Lorsqu'on construit une application avec Laravel on a de nombreuses tâches à accomplir, comme par exemple créer des contrôleur, des modèles, vérifier les routes...
- C'est là qu'intervient Artisan.
- Il fonctionne en ligne de commande, donc à partir de la console.
- Pour obtenir la liste de ses possibilités, il suffit de se positionner dans le dossier racine du projet et d'utiliser la commande :

```
php artisan
```

3

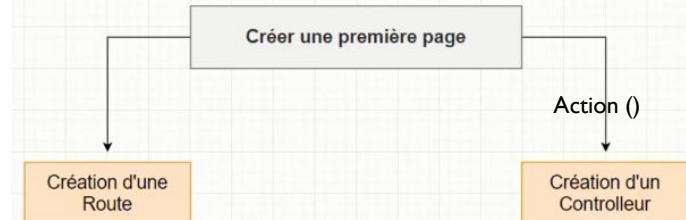
Chapitre 3:  
Framework laravel – LES BASES

## •° LE ROUTAGE

4

## *Création d'une nouvelle page => définition d'une route*

**Le rôle d'une route est tout simplement de lier une URL à une action définie dans un contrôleur ou à une fonction anonyme**



- Qu'est ce qu'une route ?



**Ainsi, en lançant la requête HTTP par cette route, on aura la réponse du contrôleur ou de la fonction.**

5

## *Le routage dans Laravel*

- C'est avec le fichier `routes/web.php` que la requête va être analysée et dirigée.
- Route** : c'est la classe qui fonctionne en tant que routeur,
- get** : est la méthode de classe Route pour recevoir des requête HTTP « **GET** », on verra plus tard qu'il en a d'autres méthodes (**post, put/patch, delete**)
- La route '/'** : signifie que l'url comporte uniquement le nom de domaine,
- Une route peut lancer**
  - l'exécution d'une réponse automatique à travers une fonction anonyme,
  - ou une méthode/action définie dans un contrôleur.

6

## Le routage dans Laravel

### Les réponses automatiques

(réponse sans recourt au modèle MVC)

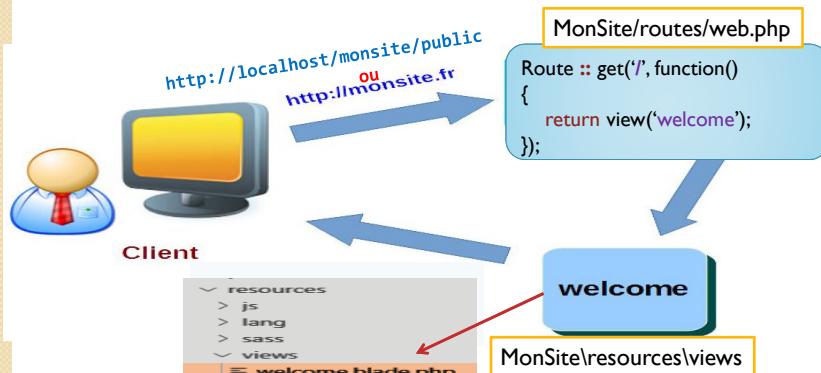
- Une réponse automatique est une réponse HTTP créée directement par une fonction anonyme (`function(){...}`) définie et exécutée par la méthode de classe `Route::get` dans le fichier `web.php`
- La requête est capturée avec la méthode `get` qui prend deux paramètres:
  - La route à exécuter (sous forme d'un string )
  - Une fonction anonyme permettant de retourner la réponse HTTP

7

## Le routage dans Laravel

### Les réponses automatiques (suite...)

- En absence d'un contrôleur, la vue `welcome` est renvoyée à l'utilisateur en tant qu'une réponse HTTP (`return view('welcome')`), et cela à travers une fonction anonyme définie et exécutée par la méthode de classe `Route::get` dans le fichier `web.php`



8

## *Lister les routes d'un projet Laravel*

- Pointez le terminal sur le dossier du projet avec la commande Cd
- Puis exéutez la commande:
  - `php artisan route:list`



```
D:\laragon\www\monsitev10> php artisan route:list
+----+-----+-----+
| Method | URI | Controller |
+----+-----+-----+
| GET/HEAD | / | ... |
| POST | _ignition/execute-solution | Ignition\ExecuteSolution : Spatie\LaravelIgnition\ExecuteSolutionController |
| GET/HEAD | _ignition/health-check | Ignition\HealthCheck : Spatie\LaravelIgnition\HealthCheckController |
| POST | _ignition/update-config | Ignition\UpdateConfig : Spatie\LaravelIgnition\UpdateConfigController |
| GET/HEAD | api/user | ... |
| GET/HEAD | articles | ArticleController@show |
| GET/HEAD | category/{category}/products | ProductsController@index |
| GET/HEAD | contact | ... |
| POST | contact | CreateContact : ContactController@create |
| GET/HEAD | contact2 | ContactController@create |
| POST | contact2 | StoreContact : ContactController@store |
| POST | contact2 | Create2Contact : ContactController2@create |
| POST | contact2 | Store2Contact : ContactController2@store |
+----+-----+-----+
```

## *Activité 1*

- Créez et testez une route "/start" nommée "start" qui permet d'afficher le texte "Hello, i am your first route"
  - **Résultat d'affichage:** "Hello my name, i am your first route"
- Créez une vue nommé "start.blade.php" qui affiche le même message de la question précédente.
- Redéfinissez la route pour qu'elle retourne la vue "start" comme réponse Http.
- Listez les routes

10

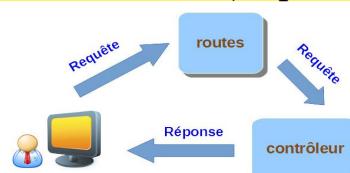
Chapitre 3:  
Framework laravel – LES BASES

## • ° LES CONTRÔLEURS

11

### *Les contrôleurs*

- **Rôle:** Il reçoit une requête et il interagit avec les différents composants d'une application **Laravel** (les vues, les modèles,...) pour retourner une réponse



- Chaque méthode (action) du contrôleur est associée à une **route**
- Dans un contrôleur, il n'y a que du code **PHP** (pas de **HTML**, ni **CSS**, ni **JS**)

12

## *Création d'un contrôleur avec l'outil Artisan (1/3)*

1. **Constitution:** pointez le terminal sur votre dossier de projet avec la commande **cd** puis entrez cette commande :

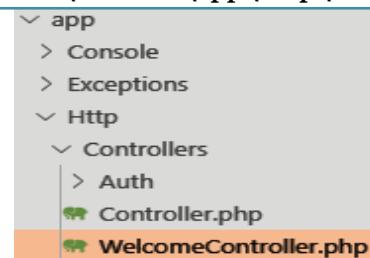
```
php artisan make:controller WelcomeController
```



Vous trouverez la classe générée dans le dossier  
C:\laragon\www\MonSite\app\Http\Controllers

```
D:\laragon\www
λ cd MonSite

D:\laragon\www\MonSite
λ php artisan make:controller WelcomeController
Controller created successfully.
```



13

## *Création d'un contrôleur avec l'outil Artisan (2/3)*

2. Dans le code généré, ajoutez la méthode **index** qui répond à l'URL de base '/':

```
namespace App\Http\Controllers;
class WelcomeController extends Controller
{
    public function index(){
        return view("home");
    }
}
```

- Le contrôleur est définie dans l'espace de nom (**App\Http\Controllers**), on le trouve ainsi dans ce dossier.
- Le contrôleur hérite de la classe **Controller** qui se trouve dans le même dossier et qui permet de factoriser des actions communes à tous les contrôleurs

14

## *Création d'un contrôleur avec l'outil Artisan (3/3)*

3. Liaison avec les routes : dans le fichier **routes/web.php**, ajoutez ce code

→ Déclaration de l'emplacement de la classe contrôleur

```
use App\Http\Controllers\WelcomeController;
```

```
Route::get('/', [WelcomeController::class, 'index']);
```

Route

```
use App\Http\Controllers\WelcomeController;
Route::get('home', [WelcomeController::class, 'index']);
```

désigner le contrôleur et la méthode dans un tableau

```
class WelcomeController extends Controller {
    public function index(){
        return view("home");
    }
}
```

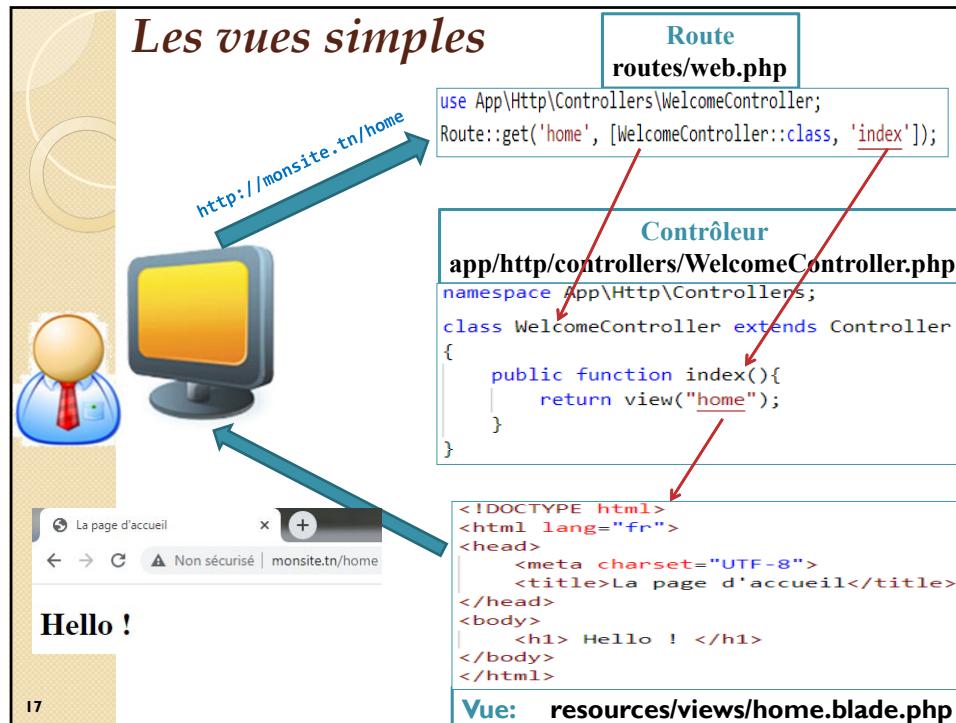
Contrôleur

15

Chapitre 3:  
Framework laravel – LES BASES

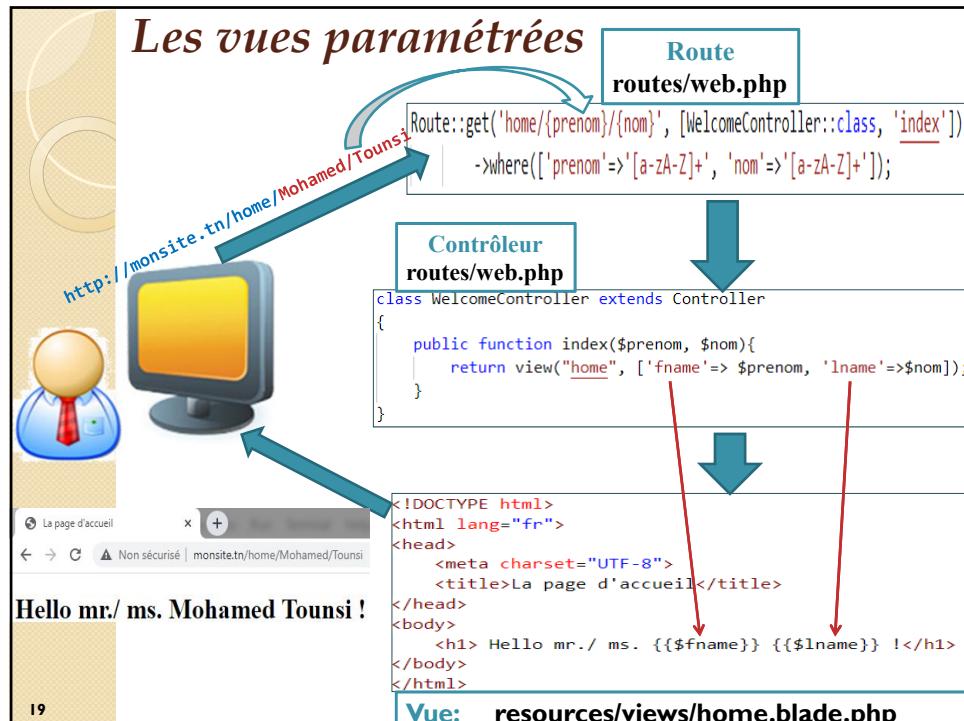
### •° LES VUES

16



## Activité 2

- Créez un Contrôleur nommé `TestController`:
- Utilisez "`TestController`", la route "`start`" et la vue "`start.blade.php`" pour afficher le message "Hello, i am your first route"



19

## Les vues paramétrées Remarques importantes!

- Au niveau du contrôleur :**

- On peut transmettre les paramètres à la vue selon 3 syntaxes (vous avez le choix entre elles):

- With **Avec un tableau associatif:**

```
return view("home", ['fname'=>$prenom, 'lname'=>$nom]);
```

Dans ce cas les variables envoyées à la vue `home` sont `$fname` et `$lname`

- With **Avec la fonction compact:**

```
return view("home", compact('prenom', 'nom'));
# équivalent à:
return view("home", ['prenom'=>$prenom, 'nom'=>$nom]);
```

Dans ce cas les variables envoyées à la vue `home` sont `$nom` et `$prenom`

- With **Avec la fonction with:**

```
return view("home")->with(['nom'=>$nom, 'prenom'=>$prenom]);
# si on a une seule variable, disant $nom, on écrit:
return view("home")->with('nom', $nom);
```

20

## Activité 3

- Modifiez le contrôleur "TestController", la route "start" et la vue "start.blade.php" pour afficher le message "Hello \*\*\*\*, your age is \*\*\*\* years"

21

Chapitre 3:  
Framework laravel – LES BASES

## •° LES FORMULAIRES

22

## Les formulaires

- Scénario et routes:



- On va donc avoir besoin de deux routes dans MonsSite/routes/web.php :

```
use App\Http\Controllers\UsersController;

Route::get('users', [UsersController::class, 'create']);
Route::post('users', [UsersController::class, 'store']);
```

23

## Les formulaires

### 1. Création

- Soit la vue `resources/views/infos.blade.php` qui affiche le formulaire:

```
Entrez votre nom :  Envoyer !
<form action="{{ url('users') }}" method="POST">
    @csrf
    <label for="nom">Entrez votre nom : </label>
    <input type="text" name="nom" id="nom">
    <input type="submit" value="Envoyer !">
</form>
```

- Les helpers `url` ou `route` sont utilisés pour générer l'url complète pour l'action du formulaire à partir de la route.

```
use App\Http\Controllers\UsersController;
Route::get('users', [UsersController::class, 'create'])->name('create_user');
Route::post('users', [UsersController::class, 'store'])->name('store_user');
```

- Donc, pour définir la valeur de l'attribut `action` du formulaire, on procède comme suit:

```
<form action="{{url('users')}}" method="post">
```

- Ou:

```
<form action=" {{route('store_user')}} " method="post">
```

24

## Les formulaires

### 2. Association à un contrôleur

- Utilisez Artisan pour générer un contrôleur :

```
php artisan make:controller UsersController
```

- Modifiez ensuite son code ainsi :

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UsersController extends Controller
{
    public function create()
    {
        return view('infos');
    }

    public function store(Request $request)
    {
        return 'Le nom est ' . $request->input('nom');
    }
}
```

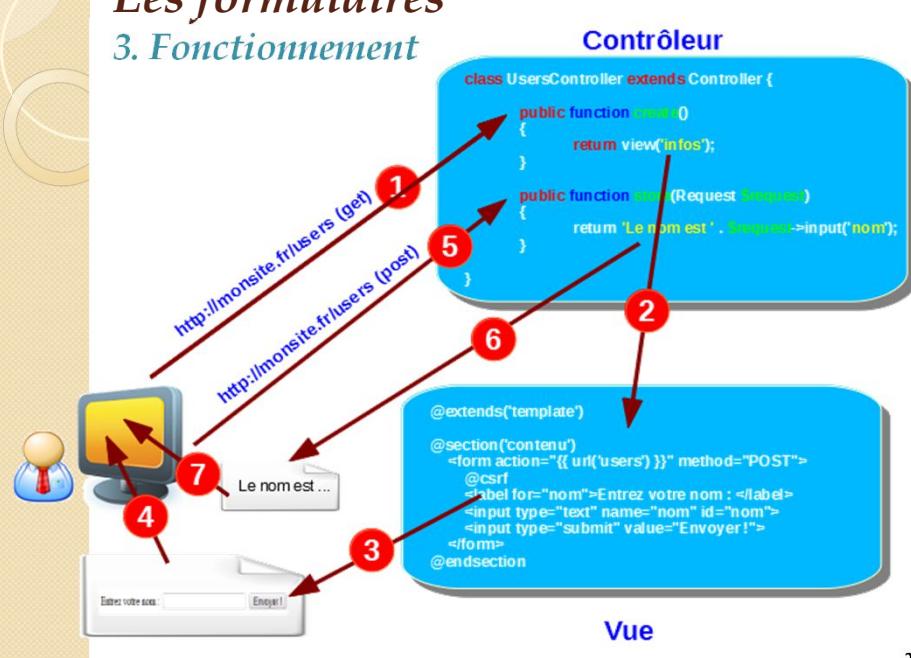
Pour récupérer le paramètre de la requête

Ou tout simplement on écrit:  
\$request->nom

25

## Les formulaires

### 3. Fonctionnement



26

## Les formulaires

### 3. Fonctionnement (explication)

- **Explication:**

1. le client envoie la requête de demande du formulaire qui est transmise au contrôleur par la route,
2. le contrôleur appelle la vue « infos »,
3. la vue « infos » crée le formulaire,
4. le formulaire est envoyé au client,
5. le client soumet le formulaire, le contrôleur reçoit la requête de soumission par l'intermédiaire de la route (non représentée sur le schéma),
6. le contrôleur génère la réponse,
7. la réponse est envoyée au client.

27

## Les formulaires

### 4. En cas de plusieurs champs de saisie

- Comment récupérer les valeurs saisies dans le contrôleur et les affichées dans une vue?

- **La solution la plus simple:**

- Au niveau du contrôleur: utiliser la fonction compact

```
public function store(Request $request)
{
    return view("afficheForm", compact('request'));
}
```

- Au niveau de la vue afficheForm.blade.php

Le nom est:  `{{$request->input('nom')}}`

Le prénom est:  `{{$request->input('prenom')}}`

Ou on écrit directement:  
 `{{ $request->nom }}`

28

## Les formulaires

### 5. La protection CSRF (@csrf)

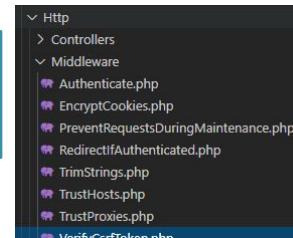
- Si on regarde le code généré on trouve quelque chose dans ce genre :

```
<input type="hidden" name="_token" value="iIjW9PMNsV6VKT2sIc16ShoTf6SdVVZolVUGsxDI">
```

- CSRF signifie Cross-Site Request Forgery.** C'est une attaque qui consiste à faire envoyer par un client une requête à son insu.
- Pour se prémunir contre ce genre d'attaque Laravel génère une valeur aléatoire (**token**) associée au formulaire de telle sorte qu'à la soumission cette valeur est vérifiée pour être sûre de l'origine.

*Vous vous demandez peut-être où se trouve ce middleware CSRF ?*

Il est bien rangé dans le dossier:  
**app/Http/Middleware**



29

## Activité 4

- Créez la vue info.blade.php, qui affiche un formulaire avec deux zones de texte nom, prenom, age, sexe (zone de sélection simple), et loisirs (liste de checkbox)
- Définissez les routes nécessaires à l'affichage et la soumission du formulaire
- Associez le formulaire à un contrôleur "TestFormController", qui définit deux actions:
  - Create(): permet d'afficher le formulaire
  - Store(): permet de récupérer les valeurs saisies dans le formulaire et de les afficher dans la vue "afficheInfo.blade.php"

30

Chapitre 3:  
Framework laravel – LES BASES

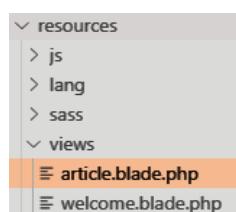
• **SYNTAXE BLADE**

31

## *Blade*

- Laravel possède un moteur de template élégant nommé Blade qui nous permet de simplifier la syntaxe des vues.

- **Enregistrement:**



32

## Règle simple

- Tout ce qui se trouve entre les doubles accolades est interprété comme du code PHP.
- Par exemple, au lieu d'écrire:  

```
&copy;isims 2020-<?php echo date('Y'); ?>
```
- On écrit simplement en Blade:  

```
&copy;isims 2020-{{date('Y')}}
```
- On aura toujours le même affichage:

©isims 2020-2022

33

## Affichage des variables:

### Variable de type primitif

- Toujours en utilisant les {{ ... }}
- Par exemple au lieu de la ligne suivante :  

```
<h1> Hello mr./ ms. <?php echo '$prenom $nom !';?> </h1>
```
- On peut utiliser cette syntaxe avec Blade :  

```
<h1> Hello mr./ ms. {{$prenom}} {{$nom}} !</h1>
```
- On peut aussi utiliser @empty et @isset pour vérifier les variables comme suit:

```
@isset($nom)
<p>Hello mr./ ms.{{$nom}} </p>
@endisset
```

34

## Affichage des variables:

### Variable de type Objet

- On suppose que le contrôleur a envoyé à une vue blade, un objet de type user ayant un id, un nom et un prénom:

```
$user=new User(23, 'foulen', 'ben foulen');
return view("home", compact('user'));
```

- L'affichage des détails de cet utilisateur dans la vue est comme suit:

```
L'id est : {{$user->id}}
Son prenom: {{$user->prenom}}
Son nom: {{$user->nom}}
```

35

## Les structures conditionnelles

- La structure de if:

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

Attention, la fonction isset  
dans un if est sans @

```
@if(isset($user))
    L'id est : {{$user->id}}
    Son prenom: {{$user->prenom}}
    Son nom: {{$user->nom}}
@endif
```

Équivalent à

- La structure de switch:

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break

    @default
        Default case...
@endswitch
```

```
@isset($user)
    L'id est : {{$user->id}}
    Son prenom: {{$user->prenom}}
    Son nom: {{$user->nom}}
@endisset
```

36

## Les boucles

- Boucle for:

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor
```

- Boucle foreach:

```
@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach
```

- Boucle while:

```
@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

37

## Les boucles

- Lorsque vous utilisez des boucles, vous pouvez également terminer la boucle ou ignorer l'itération en cours à l'aide des directives @continue et @break :

```
@foreach ($users as $user)
@if ($user->type == 1)
    @continue
@endif

<li>{{ $user->name }}</li>

@if ($user->number == 5)
    @break
@endif
@endforeach
```

Équivalent à

```
@foreach ($users as $user)
@continue($user->type == 1)

<li>{{ $user->name }}</li>

@if ($user->number == 5)
    @break($user->number == 5)
@endif
@endforeach
```

38

## Les templates (1/3)

- Une fonction fondamentale de Blade est de permettre de faire du templating, c'est à dire de factoriser du code de présentation.
- Voyons les deux exemples suivants:**

### Exemple 1:

```
Route::get('article/{n}', function($n) {
    return view('article')->with('numero', $n);
})->where('n', '[0-9]+');

<!doctype html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Les articles</title>
</head>
<body>
    <p>C'est l'article n° <?php echo $numero ?></p>
</body>
</html>
```

### Exemple 2:

```
Route::get('facture/{n}', function($n) {
    return view('facture')->withNumero($n);
})->where('n', '[0-9]+');

<!doctype html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Les factures</title>
</head>
<body>
    <p>C'est la facture n° {{ $numero }}</p>
</body>
</html>
```

39

## Les templates (2/3)

### • Solution : tempalte.blade.php

### articles.blade.php

```
<!doctype html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>@yield('titre')</title>
</head>
<body>
    @yield('contenu')
</body>
</html>
```

@extends('template')

```
@section('titre')
    Les articles
@endsection

@section('contenu')
    <p>C'est l'article n° {{ $numero }}</p>
@endsection
```

factures.blade.php

```
@extends('template')

@section('titre')
    Les factures
@endsection

@section('contenu')
    <p>C'est la facture n° {{ $numero }}</p>
@endsection
```

40

## *Les templates (3/3)*

- Explication:

- **tempalte.blade.php** comporte la structure globale des pages et est déclaré comme parent par les autres vues : `@extends('template')`
- Dans le template on prévoit un emplacement (**@yield**) pour que les vues enfants puissent placer leur code :

```
<title>@yield('titre')</title>
<body>@yield('contenu')</body>
```

- Ainsi dans les vues enfants (articles et factures) on utilise cet emplacement :

```
@section('contenu')
// Code de la vue
@endsection
```

41

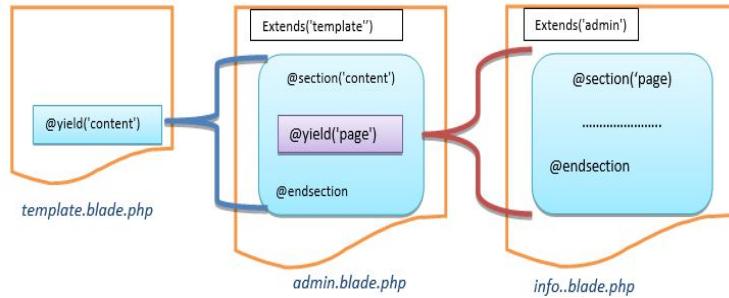
## *Front-end avec BootStrap*

- L'intégration des feuilles de styles et des fichiers JS se fait dans **tempalte.blade.php**
- Nous allons faire une démonstration en classe en se basant sur le site de documentation de Bootstrap:  
<https://getbootstrap.com/docs/5.1/getting-started/introduction/>

42

## Activité 5

- essayez de créer cette hiérarchie de pages blade, en respectant ce qui est introduit dans le schéma suivant:



- Dans la page admin.blade.php, ajoutez un navbar contenant les éléments "TestController" et "TestForm", pour tester le fonctionnement décrit dans les activités 2 et 4 à travers des liens.

43