

Cours Laravel 6 – les bases – les réponses

laravel.sillo.org/cours-laravel-6-les-bases-les-reponses/

bestmomo

August 28,
2019

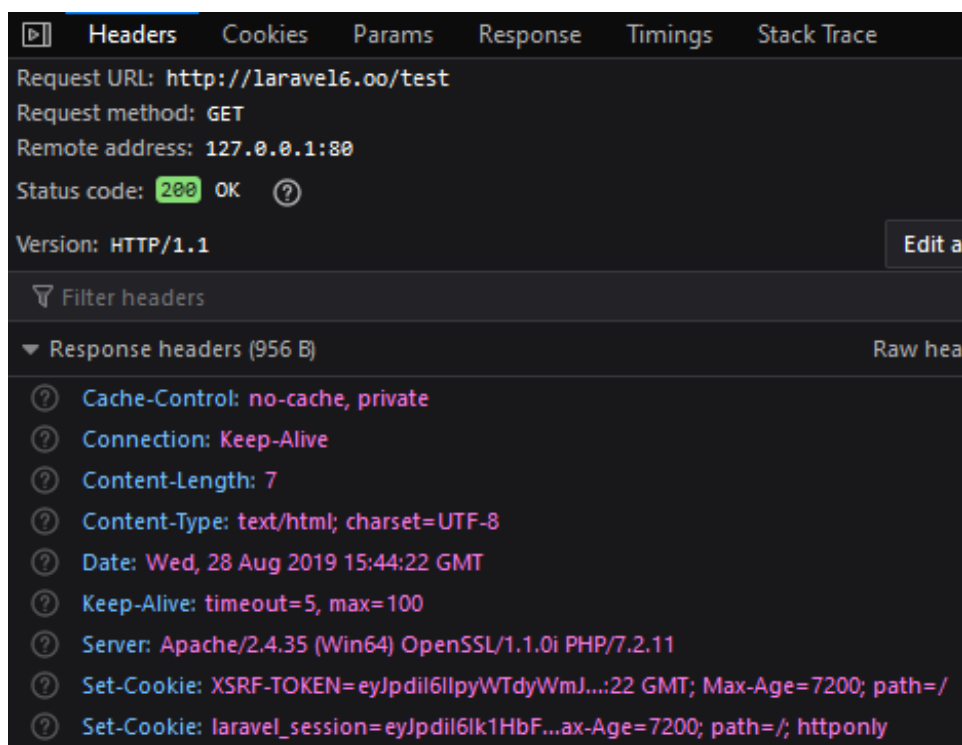
Nous avons vu précédemment comment la requête qui arrive est traitée par les routes. Voyons maintenant les réponses que nous pouvons renvoyer au client. Nous allons voir le système des vues de Laravel avec la possibilité de transmettre des paramètres. Nous verrons aussi comment créer des templates avec l'outil Blade.

Les réponses automatiques

Nous avons déjà construit des réponses lorsque nous avons vu le routage au chapitre précédent mais nous n'avons rien fait de spécial pour cela, juste renvoyé une chaîne de caractères comme réponse. Par exemple si nous utilisons cette route :

```
Route::get('test', function () {  
    return 'un test';  
});
```

Nous interceptons l'url **http://monsite/test** et nous renvoyons la chaîne de caractères « un test ». Mais évidemment Laravel en coulisse construit une véritable réponse HTTP. Voyons cela :



On se rend compte qu'on a une requête complète avec ses headers mais nous ne pouvons pas intervenir sur ces valeurs. Remarquez au passage qu'on a des cookies, on en reparlera lorsque nous verrons les sessions.

Le **content-type** indique le type **MIME** du document retourné, pour que le navigateur sache quoi faire du document en fonction de la nature de son contenu. Par exemple :

- **text/html** : page Html classique
- **text/plain** : simple texte sans mise en forme
- **application/pdf** : fichier pdf
- **application/json** : données au format JSON
- **application/octet-stream** : téléchargement de fichier

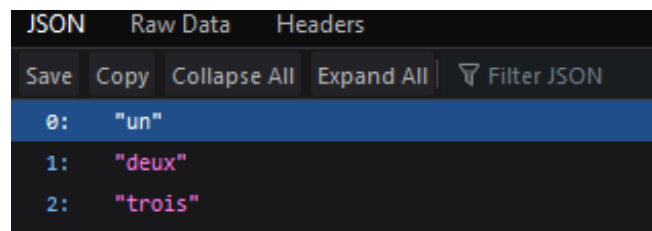
Pour une liste exhaustive c'est ici.

Maintenant voyons ce que ça donne si on renvoie un tableau :

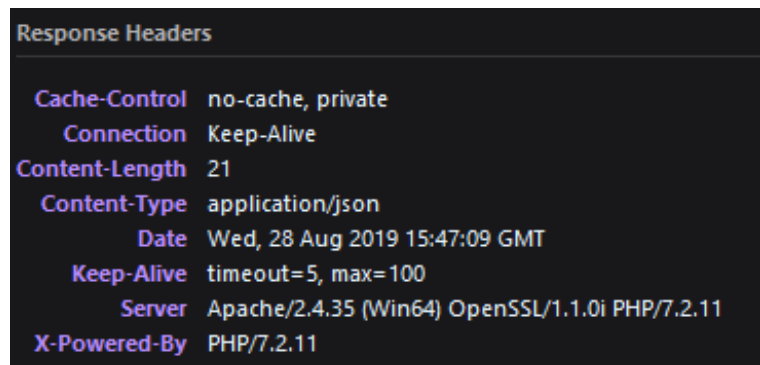
```
Route::get('test', function () {  
    return ['un', 'deux', 'trois'];  
});
```

Cette fois on reçoit une réponse JSON :

Donc si vous voulez renvoyer du JSON il suffit de retourner un tableau et Laravel s'occupe de tout !



JSON	Raw Data	Headers
Save	Copy	Collapse All Expand All Filter JSON
0:	"un"	
1:	"deux"	
2:	"trois"	



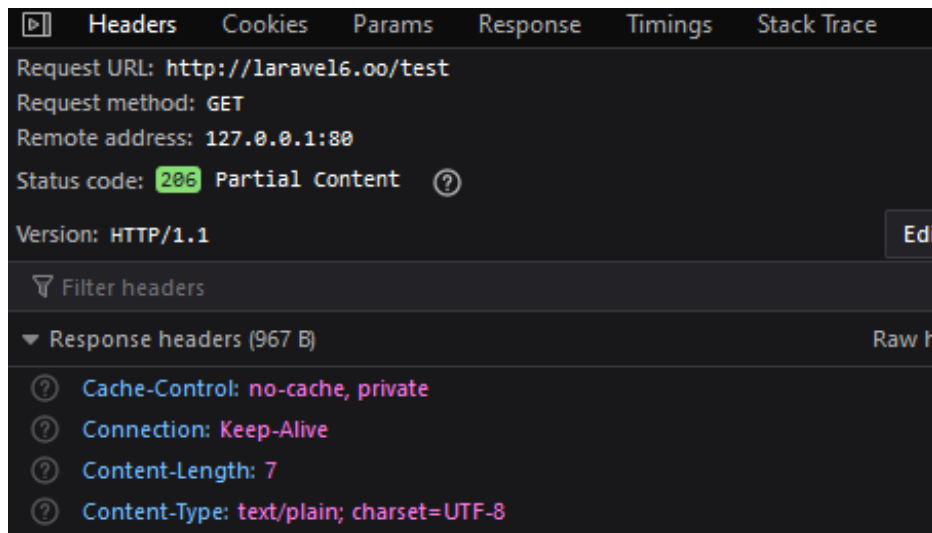
Response Headers	
Cache-Control	no-cache, private
Connection	Keep-Alive
Content-Length	21
Content-Type	application/json
Date	Wed, 28 Aug 2019 15:47:09 GMT
Keep-Alive	timeout=5, max=100
Server	Apache/2.4.35 (Win64) OpenSSL/1.1.0i PHP/7.2.11
X-Powered-By	PHP/7.2.11

Construire une réponse

Le fonctionnement automatique c'est bien mais des fois on veut imposer des valeurs. Dans ce cas il faut utiliser une classe de Laravel pour construire une réponse. Comme la plupart du temps on a un helper qui nous évite de déclarer la classe en question (en l'occurrence c'est la classe **Illuminate\Http\Response** qui hérite de celle de Symfony : **Symfony\Component\HttpFoundation\Response**).

```
Route::get('test', function () {  
    return response('un test', 206)->header('Content-Type', 'text/plain');  
});
```

Cette fois j'impose un code (**206** : envoi partiel) et un type MIME (**text/plain**) :



Dans le protocole HTTP il existe des codes pour spécifier les réponses. Ces codes sont classés par grandes catégories. Voici les principaux :

- **200** : requête exécutée avec succès,
- **403** : ressource interdite,
- **404** : la ressource demandée n'a pas été trouvée,
- **503** : serveur indisponible.

Pour une liste complète c'est ici.

En fait vous aurez rarement la nécessité de préciser les headers parce que Laravel s'en charge très bien, mais vous voyez que c'est facile à faire.

On peut aussi ajouter un cookie avec la méthode **cookie**.

Les vues

Dans une application réelle vous retournerez rarement la réponse directement à partir d'une route, vous passerez au moins par une vue. Dans sa version la plus simple une vue est un simple fichier avec du code Html :

```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Ma première vue</title>
</head>
<body>
  Je suis une vue !
</body>
</html>
```

Il faut enregistrer cette vue (j'ai choisi le nom « vue1 ») dans le dossier **resources/views** avec l'extension php :

Même si vous ne mettez que du code Html dans une vue vous devez l'enregistrer avec l'extension php.

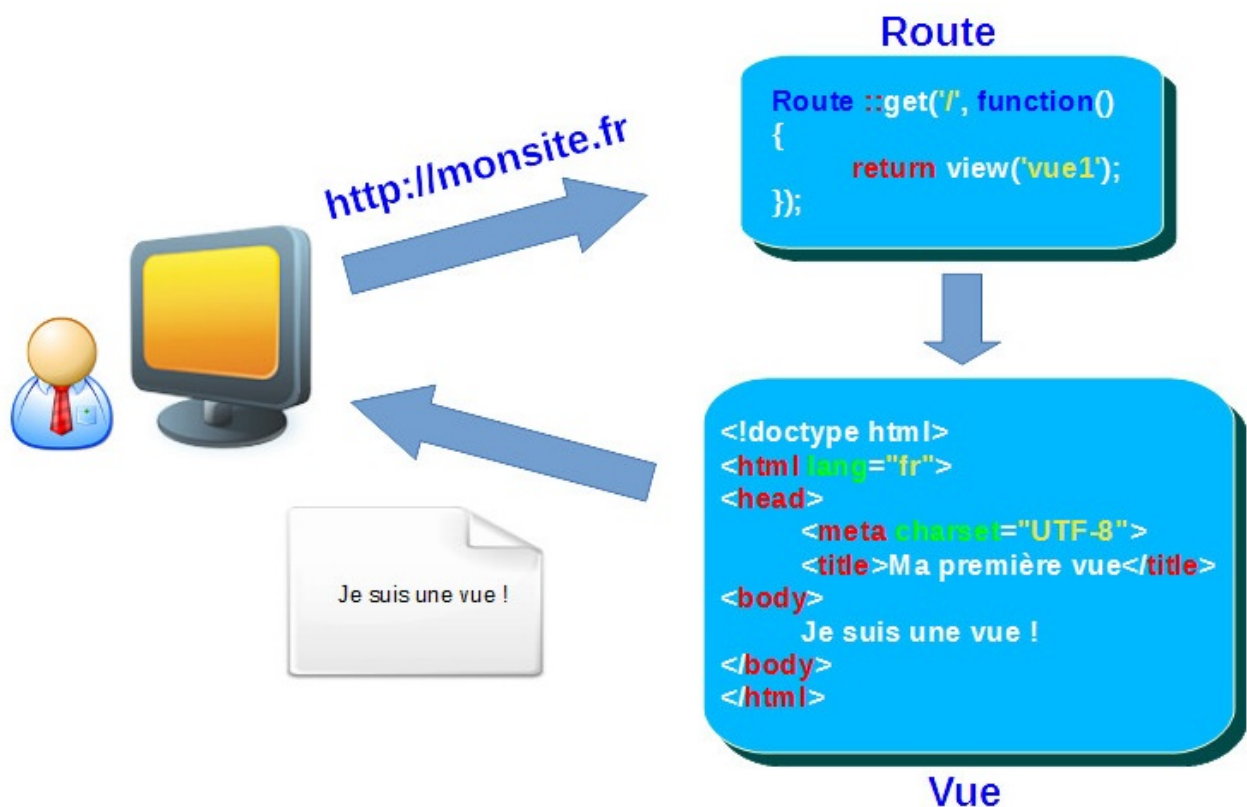
On peut appeler cette vue à partir d'une route avec ce code :

```
Route::get('/', function() {  
    return view('vue1');  
});
```

Pour que ça fonctionne commentez ou supprimez la route de base de l'installation.

Je vous rappelle la belle sémantique de Laravel qui se lit comme de la prose : je retourne (**return**) une vue (**view**) à partir du fichier de vue « vue1 ».

Voici une illustration du processus :

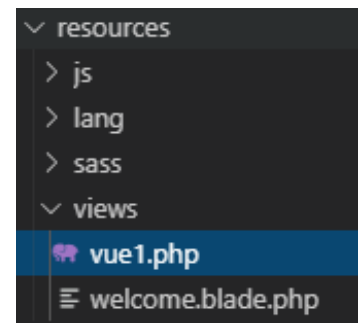


Vue paramétrée

En général on a des informations à transmettre à une vue, voyons à présent comment mettre cela en place. Supposons que nous voulions répondre à ce type de requête :

`http://monsite.fr/article/n`

Le paramètre **n** pouvant prendre une valeur numérique . Voyons comment cette url est constituée :





- la base de l'url est constante pour le site, quelle que soit la requête,
- la partie fixe ici correspond aux articles,
- la partie variable correspond au numéro de l'article désiré (le paramètre).

Route

Il nous faut une route pour intercepter ces urls :

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

On transmet la variable à la vue avec la méthode **with**.

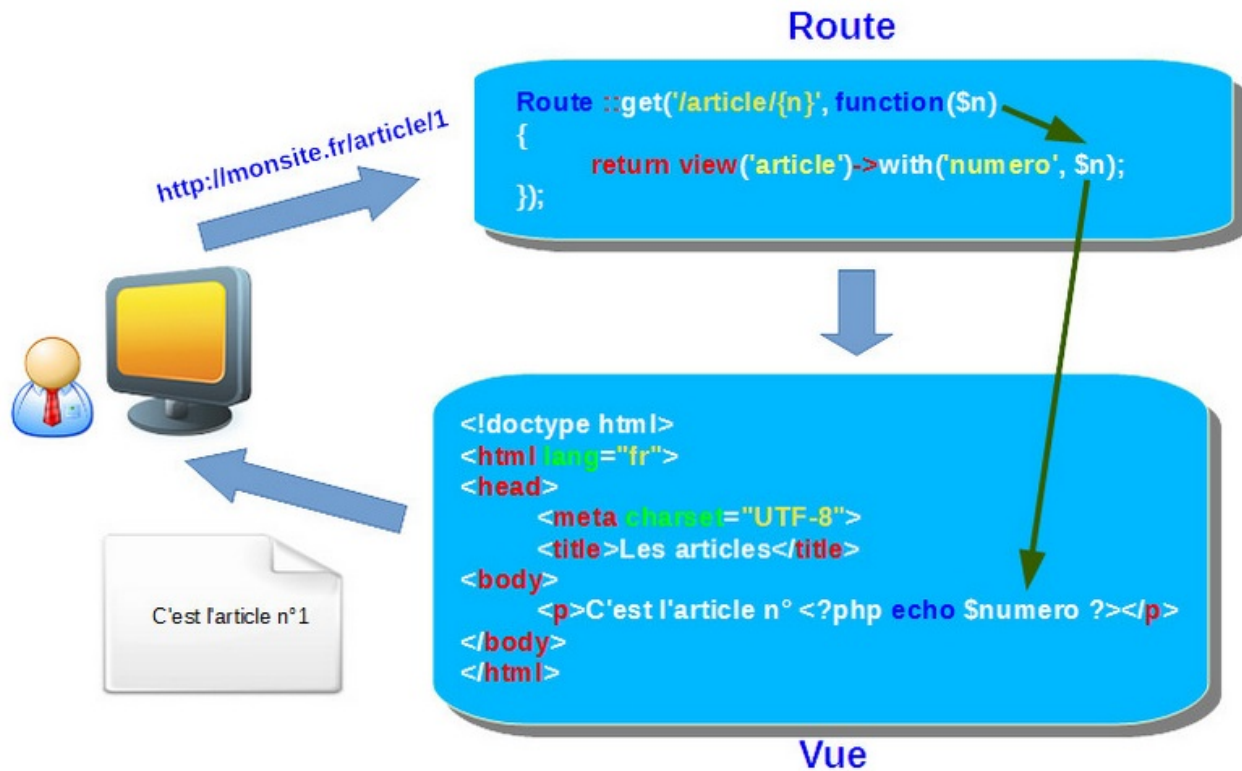
Vue

Il ne nous reste plus qu'à créer la vue **article.php** dans le dossier **resources/views** :

```
<!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>
```

Pour récupérer le numéro de l'article on utilise la variable **\$numero**.

Voici une schématisation du fonctionnement :



Il existe une méthode « magique » pour transmettre un paramètre, par exemple pour transmettre la variable **numero** comme je l'ai fait ci-dessus on peut écrire le code ainsi :

```
return view('article')->withNumero($n);
```

Il suffit de concaténer le nom de la variable au mot clé **with**.

On peut aussi transmettre un tableau comme paramètre :

```
return view('article', ['numero' => $n]);
```

Blade

Laravel possède un moteur de template élégant nommé Blade qui nous permet de faire pas mal de choses. La première est de nous simplifier la syntaxe. Par exemple au lieu de la ligne suivante que nous avons prévue dans la vue précédente :

```
<p>C'est l'article n° <?php echo $numero ?></p>
```

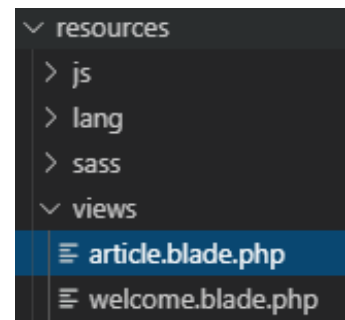
On peut utiliser cette syntaxe avec Blade :

```
<p>C'est l'article n° {{ $numero }}</p>
```

Tout ce qui se trouve entre les doubles accolades est interprété comme du code PHP. Mais pour que ça fonctionne il faut indiquer à Laravel qu'on veut utiliser Blade pour cette vue. Ça se fait simplement en modifiant le nom du fichier :

Il suffit d'ajouter « blade » avant l'extension « php ». Vous pouvez tester l'exemple précédent avec ces modifications et vous verrez que tout fonctionne parfaitement avec une syntaxe épurée.

*Il y a aussi la version avec la syntaxe {!! ... !!}. La différence entre les deux versions est que le texte entre les doubles accolades est échappé ou purifié (on utilise en interne **htmlspecialchars** pour éviter les attaques XSS). Donc soyez prudent si vous utilisez la syntaxe {!! ... !!} !*



Un template

Une fonction fondamentale de Blade est de permettre de faire du templating, c'est à dire de factoriser du code de présentation. Poursuivons notre exemple en complétant notre application avec une autre route chargée d'intercepter des urls pour des factures. Voici la route :

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

Et voici la vue :

```
<!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>
```

On se rend compte que cette vue est pratiquement la même que celle pour les articles. Il serait intéressant de placer le code commun dans un fichier.

C'est justement le but d'un template d'effectuer cette opération.

Voici le template :

```

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

```

J'ai repris le code commun et prévu deux emplacements repérés par le mot clé **@yield** et nommés « titre » et « contenu ». Il suffit maintenant de modifier les deux vues. Voilà pour les articles :

```

@extends('template')

@section('titre')
  Les articles
@endsection

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

```

Et voilà pour les factures :

```

@extends('template')

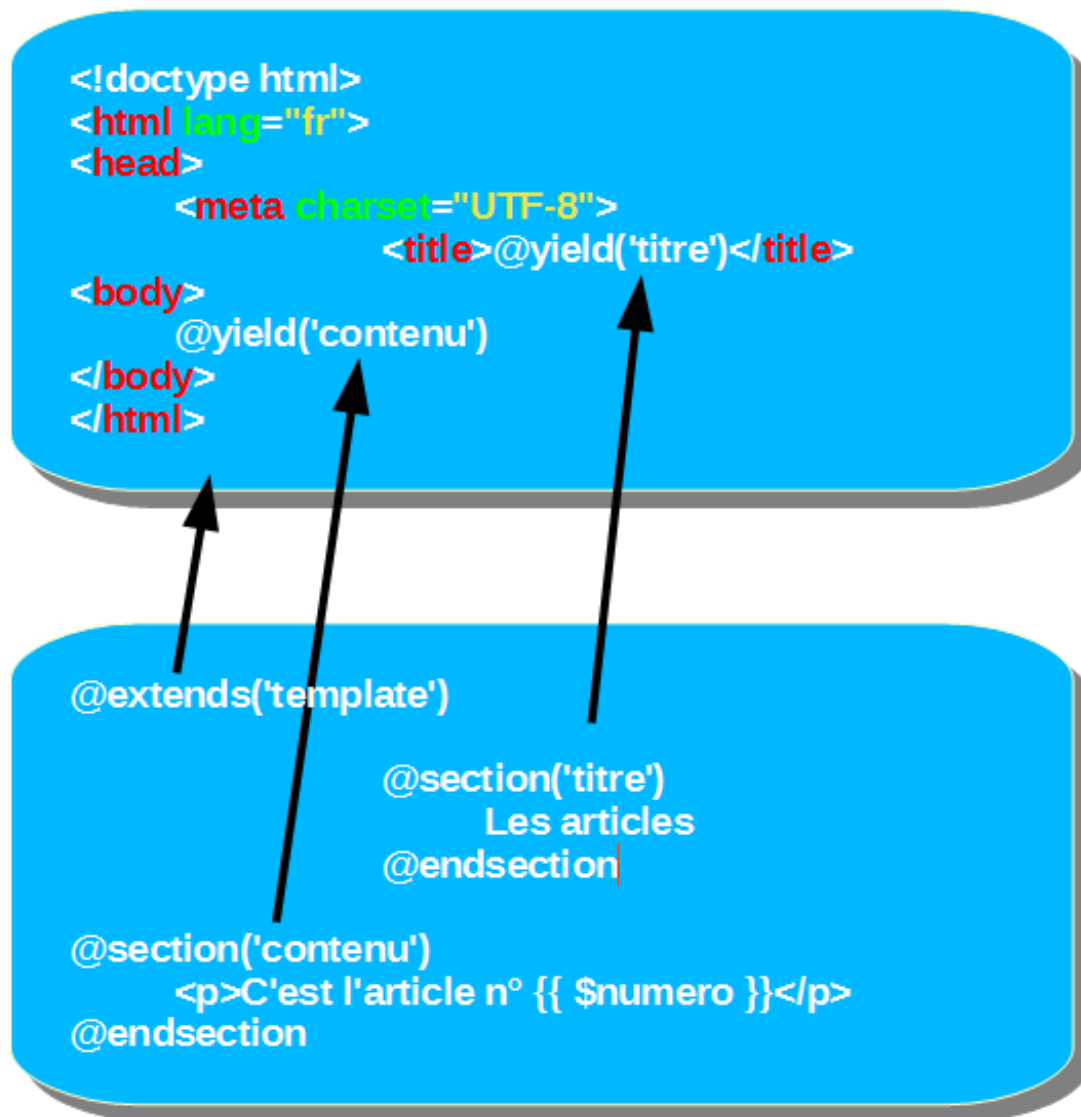
@section('titre')
  Les factures
@endsection

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

```

Dans un premier temps on dit qu'on veut utiliser le template avec **@extends** et le nom du template « template ». Ensuite on remplit les zones prévues dans le template grâce à la syntaxe **@section** en précisant le nom de l'emplacement et en fermant avec **@endsection**. Voici un schéma pour bien visualiser tout ça avec les articles :

Template

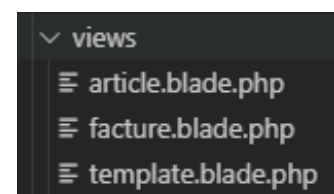


Vue

Au niveau du dossier des vues on a donc les trois fichiers :

Blade permet de faire bien d'autres choses, nous verrons cela dans les prochains chapitres.

Lorsqu'elles deviendront nombreuses on organisera nos vues dans des dossiers.



Les redirections

Souvent il ne faut pas envoyer directement la réponse mais rediriger sur une autre url. Pour réaliser cela on a l'helper **redirect** :

```
return redirect('facture');
```

Ici on redirige sur l'url **http://monsite/facture**.

On peut aussi rediriger sur une route nommée. par exemple vous avez cette route :

```
Route::get('users/action', function() {  
    return view('users.action');  
})->name('action');
```

Cette route s'appelle **action** et elle correspond à l'url **http://monsite/users/action**. On peut simplement rediriger sur cette route avec cette syntaxe :

```
return redirect()->route('action');
```

Si la route comporte un paramètre (ou plusieurs) on peut aussi lui assigner une valeur. Par exemple avec cette route :

```
Route::get('users/action/{type}', function($type) {  
    return view('users.action');  
})->name('action');
```

On peut rediriger ainsi en renseignant le paramètre :

```
return redirect()->route('action', ['type' => 'play']);
```

Des fois on veut tout simplement recharger la même page, par exemple lors de la soumission d'un formulaire avec des erreurs dans la validation des données, il suffit alors de faire :

```
return back();
```

On verra de nombreux exemples de redirections dans les prochains chapitres.

En résumé

- Laravel construit automatiquement des réponses HTTP lorsqu'on retourne une chaîne de caractère ou un tableau.
- Laravel offre la possibilité de créer des vues.
- Il est possible de transmettre simplement des paramètres aux vues.
- L'outil Blade permet de créer des templates et d'optimiser ainsi le code des vues.
- On peut facilement effectuer des redirections avec transmission éventuelle de paramètres.