Vues

- Un fichier qui contient du code HTML, PHP et ou les directives du système de Template Blade
- Les vues se trouvent dans le dossier resources/views et ont pour extension .php ou .blade.php
- Pour créer une vue, soit on crée manuellement un fichier dans resources/views ou avec la commande artisan:

```
php artisan make:view NomVue;
```

- Dans une vue, il est possible d'insérer du code php d'où l'extension .php
- Pour retourner une vue, on utilise la fonction **view** qui prend en paramètre le **nom** de la vue sans son extension

Ex. Route::get('/', function(){

return view('accueil')});

Dans cet exemple, quand un visiteur se rend sur la racine du domaine, le fichier 'accueil.php' présent dans resources/views est exécuté et le résultat sera retourné à l'utilisateur

Ex. Route::view('/', 'accueil'); équivalente à la précédente

- Ex. Vue simple

```
<!doctype html>
<html lang="fr">
    <head>
        <meta charset="UTF-8">
        <title>Ma première vue</title>
    </head>
    <body>
        <h1> Bonjour! </h1>
    </body>
</html>
```

- Question: quelle est l'extension de ce fichier ?
- Réponse: .php ou . Blade.php

Vue paramétrée

- Transmettre des informations à une vue avec la méthode **with**

Ex. Route

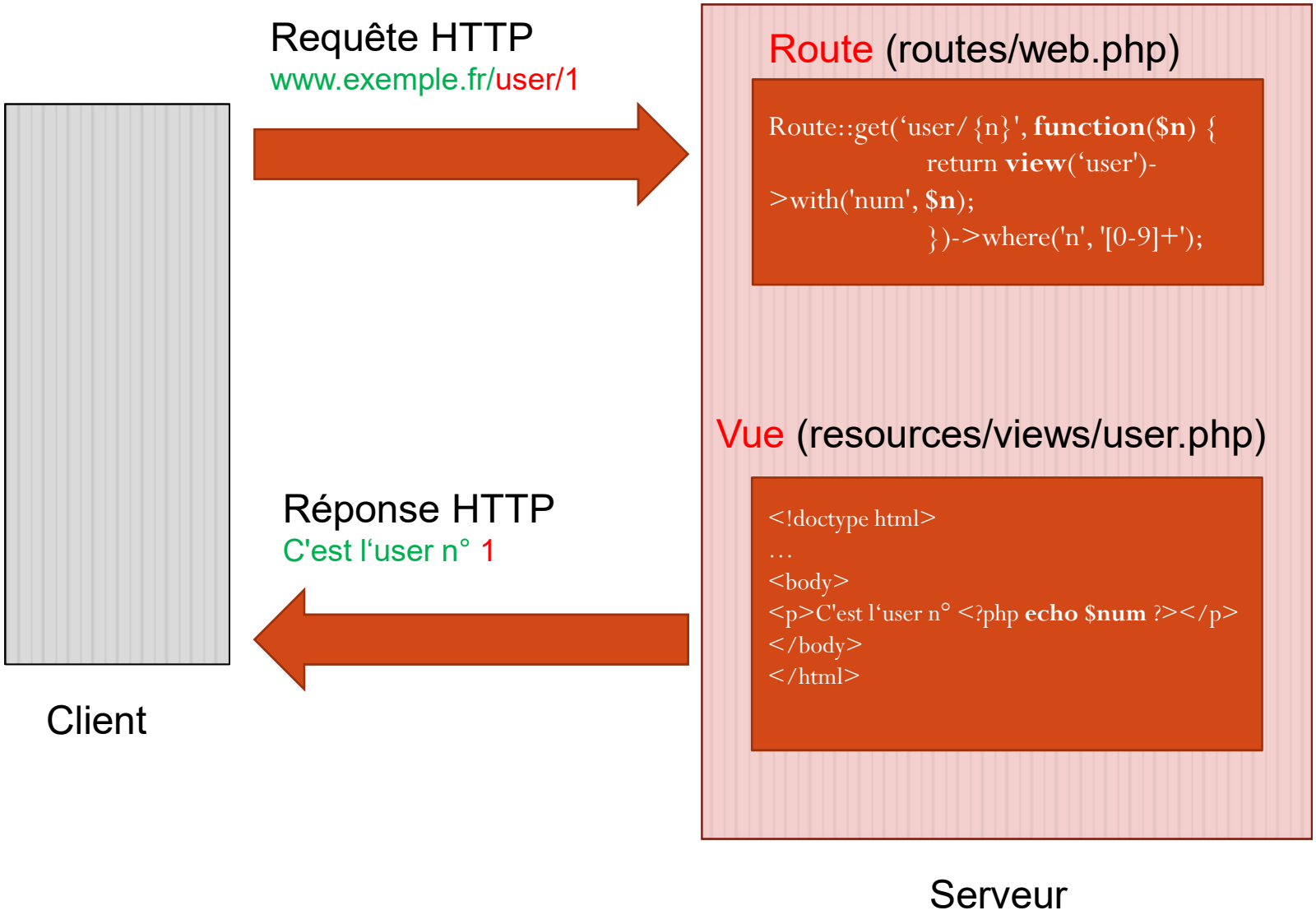
```
Route::get('user/{n}', function($n) {
return view('user')->with('num', $n);
})->where('n', '[0-9]+');
```

Vue:

```
<!doctype html>
…
<body>
<p>C'est l'user n° <?php echo $num ?></p>
</body>
</html>
```

Question: Quelle est la valeur de $num?

$num=$n


Requête HTTP
www.exemple.fr/user/1
Route (routes/web.php)
Route::get('user/{n}', function($n) {
return view('user')->with('num', $n);
})->where('n', '[0-9]+');
Vue (resources/views/user.php)
<!doctype html>
…
<body>
C'est l'user n° <?php echo $num ?>
</body>
</html>
Réponse HTTP
C'est l'user n° 1
Client
Serveur

Moteur de template blade

- Dans une vue, il est préférable d'utiliser le système de template blade inspiré du syteme Razor de .NET

- Le moteur de template est caractérisé par:
  - Syntaxe concise, simplifiée et Simple
  - Sécurité
  - Lisibilité et bonne organisation du code
  - Modèle d'héritage puissant
- Un code blade est **compilé** en code php puis mis en cache

Code php vs code blade

- Ex.
  - PHP

```
<p>C'est l'user n° <?php echo $num ?></p>
```

  - Blade

```
<p>C'est l'user n° {{$num }}</p>
```

NB: Pour que cela fonctionne, le fichier doit avoir pour extension .blade.php

Code php vs code blade

- Ex.
  - **Php**:

```
<? php if (date('d/m'=='01/01')){?>
    <p> Bonne Année</p>
<?php } else {?>
   <p> Attendre!</p>
<?php }?>
```

  - **Blade**:

```
@if (date('d/m'=='01/01'))
   <p> Bonne Année</p>
@else
  <p> Attendre!</p>
@endif
```

Syntaxe blade

- Un fichier blade est un fichier html avec l'extension .blade.php
- Un tel fichier contient deux types d'éléments qui:

❑ **affichent les données** introduits entre accolades et remplace 'echo' de php

Ex. {{$var}} => affichage protégé utilise les entités HTML pour protéger les données de l'insertion de script malicieux

- Laravel remplace ce code blade par le code php suivant:

<?php echo htmlspecialchars($var, EN_QUOTES, 'UTF-8', false); ?>

Ex. {!!$var} => affichage non protégé c'est à eviter

Directives blade

- **structurent et contrôlent l'affichage**: précédés par l'arobase @ appelés directives

  Ex. @if ($var==0)

  …

  @endif

- Une directive blade est un *raccourci* vers les structures de contrôles et instructions disponibles en php

- Elles permettent une meilleure **lisibilité** du code

Structures de contrôle

- **Conditionnelles:**
  - **@if @elseif @else @endif**

@if ($condition)

  - **@unless et @endunless (**L'inverse de @if)

@if (!$condition)

Structures de contrôle

- Boucles: afficher les éléments d'une liste

- **@for @endfor**

- **@foreach @endforeach**

- **@while @endwhile**

- **@forelse @endforelse:** comme foreach qui ajoute une condition si l'objet est **vide**

- @loop: cette variable permet notamment de connaitre l'itération de boucle en cours dans n'importe quelle boucle

Ex. $loop->first: booléen pour voir si 1ere itération ?

$loop->last: booléen pour voir si dernière itération ?

$loop->count: nombre d'éléments dans la boucle

$loop->remaining: nombre d'items restant dans la boucle

Héritage de template

- Les pages d'un même site web partagent souvent des éléments communs tels que:
  - En-tête
  - Un logo
  - Un menu de navigation
  - Un pied de page
  - Une feuille de style
- Ces éléments ne doivent pas être répétés dans chaque page

  => factoriser les éléments communs
- Blade propose un système **d'héritage** simple

Base (layout)

- La base est **étendu** par les autres pages avec extends
- Pour la créer, on doit créer un fichier de vue dans resources/views
- Par convention, la base est crée dans un sous dossier appelé 'layouts'
- La base contient en général le **DOCTYPE**, la balise **html** **head**, **meta** et **body** et tous ce qui est **commun** aux autres pages
- On utilise la directive **@yield** pour indiquer à quel endroit doit apparaitre le contenu des pages elles-mêmes.

Ex. fichier Base

- Resources/views/layouts/base.blade.php

```
<!DOCTYPE>
<html lang='fr'>
<head>
    <meta charset='utf-8'>
    <title> @yield('title', 'Page d'accueil')</title>
<body>
    <div class='main'>
            @yield('content')
    </div>
     @section('UnScript')
            <script src='app.js'></script>
      @show
</body>
</html>
```

On a 3 directives

Héritage

- Les fichiers qui héritent de la base ainsi créée doivent contenir le code @extends('layouts.base')

Ex.

```
@extends('layouts.base')
@section('title', 'contact')
@endsection
@section('content')
        <h1> email: monsite@google.com  </h1>
@endsection
@section('UnScrip')
@endsection
```

Directives Parent

- @yield: indique à quel endroit doit apparaitre le contenu des pages qui héritent de la base

- @section et @show: définir ou réserver une place pour une section dans le parent

Directives Fils

- @extend: étend une autre vue de base
- @section et @endsectoin: définir un contenu dans le fils
- @parent

Contrôleur

- Un contrôleur contient la **logique** concernant les actions effectuées par l'utilisateur
- Il permet d'**alléger** les routes du code qu'elles contiennent dans les fonctions anonymes
- Il est matérialisé par une **classe** où chaque méthode représente une action qui correspond à une route
- Un contrôleur est une **classe** qui étend la classe de base **Controller**
- Par défaut, tous les contrôleurs de l'application sont placés dans le répertoire app/Http/Controllers

- Pour créer un contrôleur avec artisan, on utilise la commande:

  php artisan make:controller NomControleur

Ex.
```
<?php
    namespace App/Http/Controllers;
    class NomControleur extends Controller
        {
            public function index()    {
                return 'bienvenue'; }
        }
?>
```

- Les méthodes d'un contrôleur retournent une réponse de type chaine de caractère, fichier JSON ou une vue

Lier une route à un contrôleur

- Définir une route vers la méthode de contrôleur

Ex. use App\Http\Controllers\NomController;

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

- Lorsqu'une requête entrante correspond à l'URI '/' de la route spécifiée, la méthode 'index' de la classe App\Http\Controllers\NomController est invoquée

Contrôleur à action unique

- Si une action du contrôleur est particulièrement complexe, il peut être utile de lui consacrer une classe entière. Pour ce faire, une seule méthode __invoke sera définie au sein du contrôleur

```
Ex. class NomController extends Controller{
        public function __invoke() {
        … } }
```

- Pas besoin de spécifier le nom de la méthode lorsqu'on définit une route pour ce genre de contrôleur

Ex. Route::get('/', NomControleur::class)

- Ce contrôleur est généré avec l'option invokable

Ex. php artisan make:controller NomController --invokable

Contrôleur/middleware

- Un midleware peut être assigné à une route du contrôleur ou défini dans le constructeur du contrôleur ou en ligne dans une **closure**

Ex. Route::get('profile', [NomController::class, index'])->middleware('auth');

Ex. class NomController extends Controller{

```
    public function __construct(){
        $this->middleware('auth');}
    }
```

Contrôleur de ressources

- Ce type de contrôleur peut être généré avec la commande:

  php artisan make:controller NomController –resource

- Il est utilisé pour gérer les opérations CRUD (Create, Read, Update, Delete) sur une ressource spécifique (modèle)

- Laravel crée automatiquement 07 méthodes de base prédéfinies pour effectuer les opérations CRUD sur une ressource telles que la création, la lecture, la mise à jour et la suppression d'objets
- les méthodes générées sont personnalisées en fonction des besoins

- Définir une route de ressource qui pointe vers ce contrôleur

Ex. Route::resource('produit', NomController::class);

get /produit =>method index

get/produit /create =>method create

post /produit =>method store

get /produit /{produit} =>method show

post /produit /{produit}/edit =>method edit

put/patch /produit /{produit} =>method update

delete /produit /{produit} =>method destroy

Ex. Route::resource(['produit'=> NomController::class,

'commande'=> NomController::class]);

- Ex.

```
class NomController extends Controller{
    // Afficher la liste des ressources (READ)
    public function index() {}
    // Afficher le formulaire de création (CREATE)
    public function create() {}
    // Enregistrer une nouvelle ressource (CREATE)
    public function store(Request $request) {}
    // Afficher une ressource spécifique (READ)
    public function show($id) {}
    // Afficher le formulaire de mise à jour (UPDATE)
    public function edit($id) {}
      // Mettre à jour une ressource spécifique (UPDATE)
    public function update(Request $request, $id) {}
  // Supprimer une ressource spécifique (DELETE)
    public function destroy($id){}}
```

Middleware

- composant qui permet de filtrer les requêtes HTTP entrantes et sortantes de l'application
- souvent utilisé pour effectuer des tâches telles que l'authentification, la gestion des sessions, la vérification des autorisations, la journalisation, la sécurité, la compression de données, la gestion des erreurs, etc
- Laravel propose des middleware intégrés tels que auth (authentification), web (gestion de sessions et protection contre les attaque CSRF), etc
- pour créer un middleware personnalisé, on utilise la commande

  php artisan make:middleware NomMiddleware
- Le fichier crée sera placé dans **app/Http/Middleware**

- on peut attribuer des alias aux middleware dans le fichier app/Http/Kernel.php et les utiliser dans les routes

```
protected $middlewareAliases = [
    'auth' => \App\Http\Middleware\Authenticate::class,..
```

- Ex. route::get('admin', [AdminController::class, 'login')->middleware('auth');

⇒ Il vérifie si un utilisateur est authentifié

- On peut assigner **plusieurs** middlewares à une **même** route

```
Ex. Route::get('/', function () { …})->middleware([First::class, Second::class]);
```

- Si on souhaite qu'un middleware soit exécuté lors de chaque requête HTTP adressée à l'application, on indique sa classe dans la propriété $middleware de la classe app/Http/Kernel.php.

- Ex. un middleware de journalisation peut enregistrer toutes les demandes entrantes adressées à l'application

- Le middleware doit être enregistré dans le noyau de l'application app/Http/Kernel.php

- L'application d'un middleware à une route se fait avec la méthode middleware dans le fichier routes/web.php ou routes/api.php
- Ex. route::get('admin', [AdminController::class, 'login')->middleware(Authenticate::class);

Migration

- Gère la structure de la base de données
- facilite la collaboration avec d'autres développeurs
- La création d'une migration se fait avec la commande artisan make:migration

  Ex. php artisan make:migration create_étudiants_table

  => crée la table étudiant

- Le fichier généré sera placé dans le répertoire database/migrations

- Chaque **nom** de fichier contient une estampille qui permet à Laravel de déterminer l'ordre des migrations

- Editer le fichier créé pour définir la structure de la base. Les migrations utilisent schema pour créer et modifier les tables et les colonnes de la base de données
- Exécuter la migration et créer la table dans la base de données avec la commande

  php artisan migrate

- Laravel utilise le nom de la migration pour tenter de deviner le nom de la table
- La structure de la migration contient deux méthode:

- up() : ajouter de nouvelles tables, colonnes et index à la base de données

- down(): assurer les opérations inverses de up()

Ex.
```
public function up(): void   {
    Schema::create('tasks', function (Blueprint $table) {
      $table->id();
      $table->string('titre');
      $table->timestamps();
      });   }
  public function down(): void   {
   Schema::dropIfExists('tasks');  }
```

- Pour connaitre l'ensemble des migrations qui ont été exécutées, on utilise la commande

  php artisan migrate:status
- L'annulation de la dernière migration se fait avec

  php artisan rollback
- Pour annuler N migrations à la fois, on exécute

  php artisan migrate:rollback --step=N
- L'annulation de toutes les migrations se fait avec

  php artisan migrate:reset

- La commande php artisan migrate:refresh permet de réinitialiser la base de données en **annulant** toutes les migrations et en les **exécutant** à nouveau.

- La commande php artisan migrate:refresh --step=N annule et exécute à nouveau les N dernières migrations

- La commande php artisan migrate:fresh supprime toutes les tables de la base de données et exécute la commande migrate

Schéma de la base de données

❑ Table

- **Création :** Pour créer une nouvelle table, on utilise la méthode create de la façade Schema qui prend deux arguments: le nom de la table et une closure
- Ex. Schema::create('users', function (Blueprint $table) {…}

- **Mise à jour** se fait avec la méthode table qui prend aussi deux arguments
- Ex. Schema::table('users', function (Blueprint $table) {…}

- **Suppression d'une table** se fait avec la méthode drop ou dropIfExists

  Ex. Schema::drop('users');

  Schema::dropIfExists('users');

- **Renommage d'une table** se fait avec la méthode rename qui prend deux arguments: l'ancien et le nouveau nom

Ex. Schema::rename($ancien, $nouveau);

❑ Colonne

- **Création d'une colonne s**e fait comme suit:

  $table->typeCol(nomCol'); où typeCol, nomCol désigne le type, le nom de la colonne

- Ex. Types de colonne: integer, string, float, increments, id, etc

- **Maj des colonnes** se fait avec la méthode change
- Ex. $table->string('adresse', 50)->change();

- **Renommage** se fait avec renameColumn
- Ex. $table->renameColumn('ancien', 'nouveau');

- **Suppression** des colonnes se fait avec dropColumn
- Ex. $table->dropColumn('adresse');