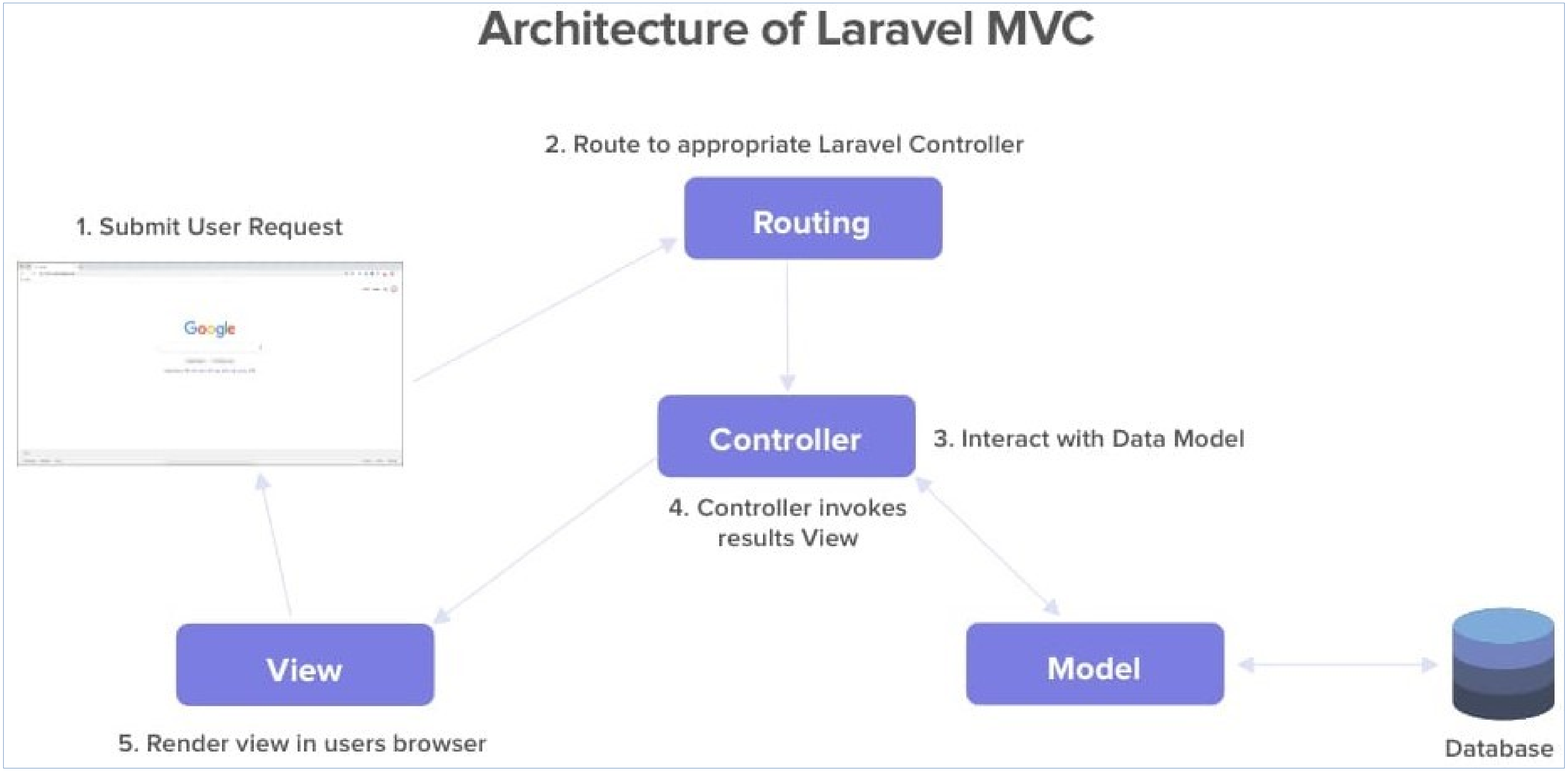
UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT
CHAPTER 2 : ROUTES, CONTROLLERS AND VIEWS

LOO YIM LING
ylloo@utar.edu.my

Previously - Laravel Framework Architecture


Architecture of Laravel MVC
2. Route to appropriate Laravel Controller
1. Submit User Request
Routing
Google
Controller
3. Interact with Data Model
4. Controller invokes results View
View
Model
5. Render view in users browser
Database

Information available on https://www.netsolutions.com/insights/laravel-framework-benefits/

Request LifeCycle – First Steps

- The entry point for all requests to a Laravel application is the `public/index.php` file.
- The `index.php` file doesn't contain much code. Rather, it is a starting point for loading the rest of the framework.
- The `index.php` file loads the Composer generated autoloader definition, and then retrieves an instance of the Laravel application from `bootstrap/app.php`.
- The first action taken by Laravel itself is to create an instance of the application / service container.

Request LifeCycle – Http / Console Kernels

- Next, the incoming request is sent to either the HTTP kernel or the console kernel. For now, let's just focus on the HTTP kernel; `app/Http/Kernel.php`.
- The HTTP kernel extends the `Illuminate\Foundation\Http\Kernel` class, running an array of `bootstrappers` before request is executed. ***These bootstrappers configure error handling, configure logging, detect the application environment, and perform other tasks that need to be done before the request is actually handled.*

Request LifeCycle – Http / Console Kernels

- The HTTP kernel also defines a list of HTTP middleware that all requests must pass through before being handled by the application.
- These middleware handle reading and writing the HTTP session, determining if the application is in maintenance mode, verifying the CSRF token, and more.
- The method signature for the HTTP kernel's handle method is quite simple: it receives a Request and returns a Response. Feed it HTTP requests and it will return HTTP responses.

Request LifeCycle – Service Providers

- One of the most important kernel bootstrapping actions is loading the service providers for your application.
- All of the service providers for the application are configured in the `config/app.php` configuration file's `providers` array.
- Laravel will iterate through this list of providers and instantiate each of them. After instantiating the providers, the `register` method will be called on all of the providers. Then, once all of the providers have been registered, the `boot` method will be called on each provider.

Request LifeCycle – Routes

- One of the most important service providers in your application is the `App\Providers\RouteServiceProvider`. This service provider loads the route files contained within your application's `routes` directory.
- Once the application has been bootstrapped and all service providers have been registered, the `Request` will be handed off to the router for dispatching. The router will dispatch the request to a route or controller, as well as run any route specific middleware.

Request LifeCycle – Routes

- **Once the application has been bootstrapped and all service providers have been registered, the `Request` will be handed off to the router for dispatching.**
- **The router will dispatch the request to a route or controller, as well as run any route specific middleware.**

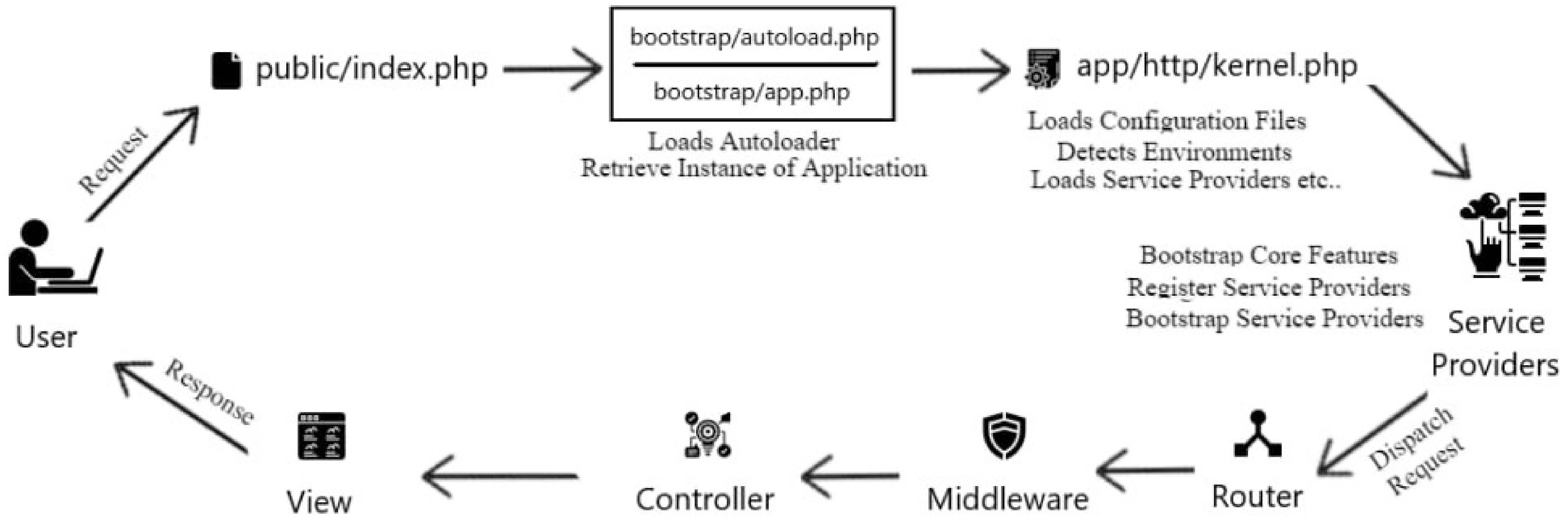
Request LifeCycle – Routes

- Middleware provide a convenient mechanism for filtering or examining `HTTP requests` entering your application.
- If the request passes through all of the matched route's assigned middleware, the route or controller method will be executed and the `response` returned by the `route` or `controller` method will be sent back through the route's chain of middleware.

Request LifeCycle – Finishing Up

- Once the response travels back through the middleware, the HTTP kernel's `handle` method returns the response object and the `index.php` file calls the `send` method on the returned response.
- The `send` method sends the response content to the user's web browser.

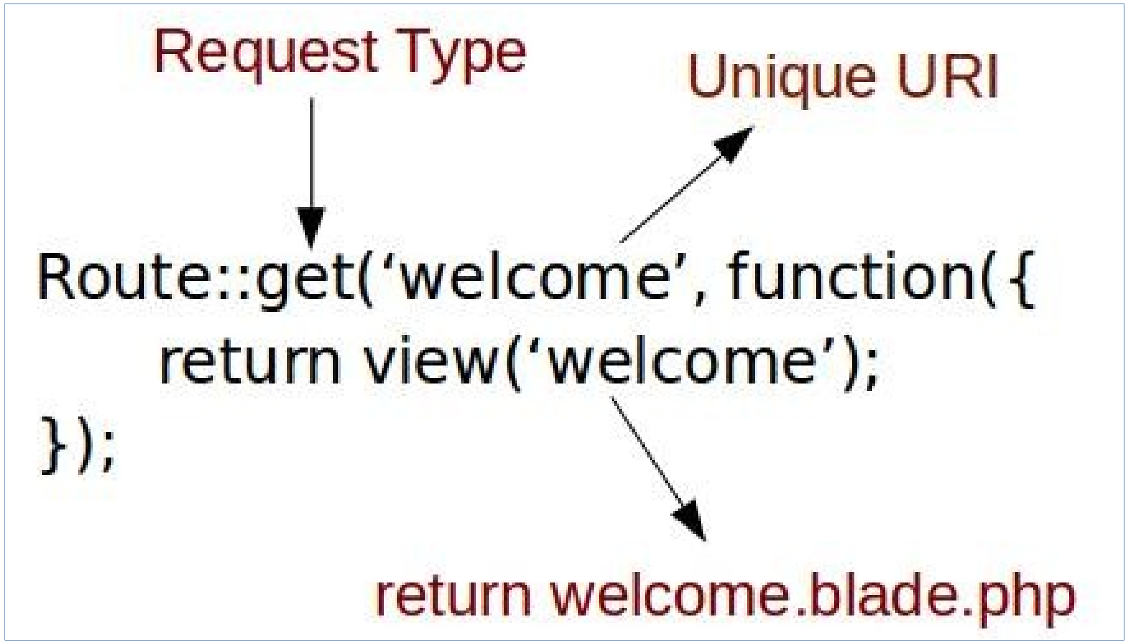
Request LifeCycle – Overview



Information available on https://dev.to/patelparixit07/laravel-request-lifecycle-195e

Routes

The most basic Laravel routes accept a URI and a closure, providing a very simple and expressive method of defining routes and behavior without complicated routing configuration files:



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

Routes – Default Route Files

- All Laravel routes are defined in `route` files, which are located in the routes directory.
- Files are automatically loaded by application's `App\Providers\RouteServiceProvider`.
- The `routes/web.php` file defines routes that are for web interface. These routes are assigned the `web` middleware group, which provides features like session state and CSRF protection.
- The routes in `routes/api.php` are stateless and are assigned the `api` middleware group.

Routes – Available Route Methods

- `Route::get($uri, $callback);`
- `Route::post($uri, $callback);`
- `Route::put($uri, $callback);`
- `Route::patch($uri, $callback);`
- `Route::delete($uri, $callback);`
- `Route::options($uri, $callback);`

Routing with combination of methods:

```
Route::match(['get', 'post'], '/', function () {
});
```

Routing with any methods:

```
Route::any('/', function () {
});
```

Routes – Redirect Routes

- Defining a route that redirects to another URI, one may use the `Route::redirect` method.
- This method provides a convenient shortcut so that one do not have to define a full route or controller for performing a simple redirect:

**Redirect to specific page:

```
Route::redirect('/here', '/there');
```

**Redirect with 301 status code

```
Route::redirect('/here', '/there', 301);
Route::permanentRedirect('/here', '/there');
```

Routes – View Routes

- If the route only needs to return a view, the `Route::view` method may be used.
- The view method accepts a URI as its first argument and a view name as its second argument.

**Routing to a view:

```
Route::view('/welcome', 'welcome');
```

**Routing to a view and passing parameter / data to the view

```
Route::view('/welcome', 'welcome',
    ['name' => 'Taylor']);
```

**When using route parameters in view routes, the following parameters are reserved by Laravel and cannot be used: `view`, `data`, `status`, and `headers`.

Routes – Controller Routes

- If the route returns a controller, the Controller need to be imported into `routes/web.php`.

**Importing Controller into routes (Laravel 8):

```
use App\Http\Controllers\Users;
```

**Routing to a controller in Laravel 8

```
Route::get("users/{user}",
 [Users::class,'index']);
```

**Routing to a controller in older Laravel versions

```
Route::get("users", "Users@index");
```

Routes – Parameters

- Sometimes, segments of the URI within route need to be captured. For example, a user's ID from the URL.
- Occasionally, there is a need to specify a route parameter that may not always be present in the URI. A ‘?’ mark may be placed after the parameter name.

**Routing to capture a parameter / data:

```
Route::get('/user/{id}', function ($id) {
    return 'User '.$id;
});
```

**Routing to capture an optional parameter / data

```
Route::get('/user/{name?}', function ($name = 'John')
{
    return $name;
});
```

Routes – Groups: Middleware

- Route groups allow sharing of route attributes, such as middleware, across a large number of routes without needing to define those attributes on each individual route.
- To assign middleware to all routes within a group, the middleware method is used before defining the group. Middleware are executed in the order they are listed in the array:

```
Route::middleware(['first', 'second'])->group(function ()
{
    Route::get('/', function () {
        // Uses first & second middleware...
    });
    Route::get('/user/profile', function () {
        // Uses first & second middleware...
    });
});
```

Controllers – Basic Information

- Instead of defining all of request handling logic as closures in route files, logics may be defined in "controller" classes.
- For example, a `UserController` class might handle all incoming requests related to users, including showing, creating, updating, and deleting users. By default, controllers are stored in the `app/Http/Controllers` directory.

**Create Controller using Artisan CLI

```
php artisan make:Controller UserController
```

Controllers – Basic Controllers

**Controller Class

```
class UserController extends Controller
{
    public function show($id)
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

**Controller Route

```
use App\Http\Controllers\UserController;
Route::get('/user/{id}', [UserController::class, 'show']);
```

Controllers – Controller Middleware

****Route defining middleware**

```
Route::get('profile', [UserController::class, 'show'])
->middleware('auth');
```

****Controller defining middleware**

```
public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('log')->only('index');
        $this->middleware('subscribed')->except('store');
    }
```

OR

```
$this->middleware(function ($request, $next) {
    return $next($request);
});
```

Controllers – Resource Controllers

- Think of each Eloquent model in web application as a "resource", then it is typical to perform the same sets of actions against each resource in the web application.
- For example, imagine the web application contains a `Photo` model and a `Movie` model. It is likely that users can create, read, update, or delete these resources.
- Laravel resource routing assigns the typical create, read, update, and delete ("CRUD") routes to a controller with a single line of code.

Controllers – Resource Controllers

**Create Resource Controller using Artisan CLI

```
php artisan make:controller PhotoController --resource
```

OR

```
php artisan make:controller PhotoController --resource --
model=Photo
```

**Declaring Resource Routes

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);
```

OR

```
Route::resources([
    'photos' => PhotoController::class,
    'movies' => MovieController::class,
]);
```

Controllers – Actions Handled by Resource Controllers

VERB	URI	ACTION	ROUTE NAME
`GET`	`/photos`	`index`	`photos.index`
`GET`	`/photos/create`	`create`	`photos.create`
`POST`	`/photos`	`store`	`photos.store`
`GET`	`/photos/{photo}`	`show`	`photos.show`
`GET`	`/photos/{photo}/edit`	`edit`	`photos.edit`
`PUT/PATCH`	`/photos/{photo}`	`update`	`photos.update`
`DELETE`	`/photos/{photo}`	`destroy`	`photos.destroy`

Controllers – Resource Controllers

**Create an API resource controller that does not include the create or edit methods

```
php artisan make:controller PhotoController --api
```

**Declaring API Resource Routes

```
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\MovieController;

Route::apiResources([
    'photos' => PhotoController::class,
    'movies' => MovieController::class,
]);
```

Views – Basic Information

- Views provide a convenient way to place all HTML in separate files.
- Views separate controller / application logic from presentation logic and are stored in the `resources/views` directory.

**A simple View (`greeting.blade.php`)

```
<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>
```

Views – Basic Information

- In order to route to the previous simple view:

**Using global view helper

```
Route::get('/', function () {
    return view('greeting', ['name' => 'Taylor']);
});
```

**Using view facade

```
use Illuminate\Support\Facades\View;

return View::make('greeting',
['name' => 'Taylor']);
```

Views – Using Facades

- Facades can be used for rendering and checking a view

**To render first view (useful in midst of an array of views)

```
use Illuminate\Support\Facades\View;
return     View::first(['custom.admin',     'admin'],
$data);
```

**To check existence of a view

```
use Illuminate\Support\Facades\View;
if (View::exists('emails.customer')) {
}
```

Views – Passing Data to View

- Parameters / Data may be passed or shared to view:

**Pass data through routes

```
return view('greeting')
            ->with('name', 'Victoria')
            ->with('occupation', 'Astronaut');
```

**Share data to all views through `App\Providers\AppServiceProvider` class

```
public function boot()
    {
        View::share('key', 'value');
    }
```

View – Blade Templates

- Unlike some PHP templating engines, Blade does not restrict usage of plain PHP codes.
- In fact, all Blade templates are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to the web application.
- Blade template files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

View – Blade Templates

- Variables or any other PHP scripts can be included in Blade views by wrapping them variable in curly braces.
- Javasripts can be included in Blade views by wrapping with `<scripts> </scripts>` and `@json()`.

**PHP

```
Hello, {{ $name }}.
```

**JS

```
<script>
    var app = @json($array);
</script>
```

View – Blade Templates (Directives)

- **PHP directive statements such as “if...else”, “switch” and “loop” can be easily scripte in blade templates:**

****Combination of loop and conditional statements.**

```
@foreach ($users as $user)
    @if ($loop->first)
        This is the first iteration.
    @endif

    @if ($loop->last)
        This is the last iteration.
    @endif

    <p>This is user {{ $user->id }}</p>
@endforeach
```

View – Components

- Components can be created as a reusable in views.
- Example of components: header, footer, layout, etc.

**Use of Artisan CLI to create layout reusable component
```
php artisan make:component layout
```
**Use of layout component in view.
```
<x-layout>
    @foreach ($tasks as $task)
        {{ $task }}
    @endforeach
</x-layout>
```

END OF LECTURE 03