LARAVEL

ROUTE, VIEWS, BLADE TEMPLATES

Routing

- Basic Routing
- Route Parameters
  - Required Parameters
  - Optional Parameters
  - Regular Expression Constraints
- Named Routes
- Route Groups
  - Middleware
  - Namespaces
  - Sub-Domain Routing
  - Route Prefixes
- Route Model Binding
  - Implicit Binding
  - Explicit Binding
- Form Method Spoofing
- Accessing The Current Route

Routing

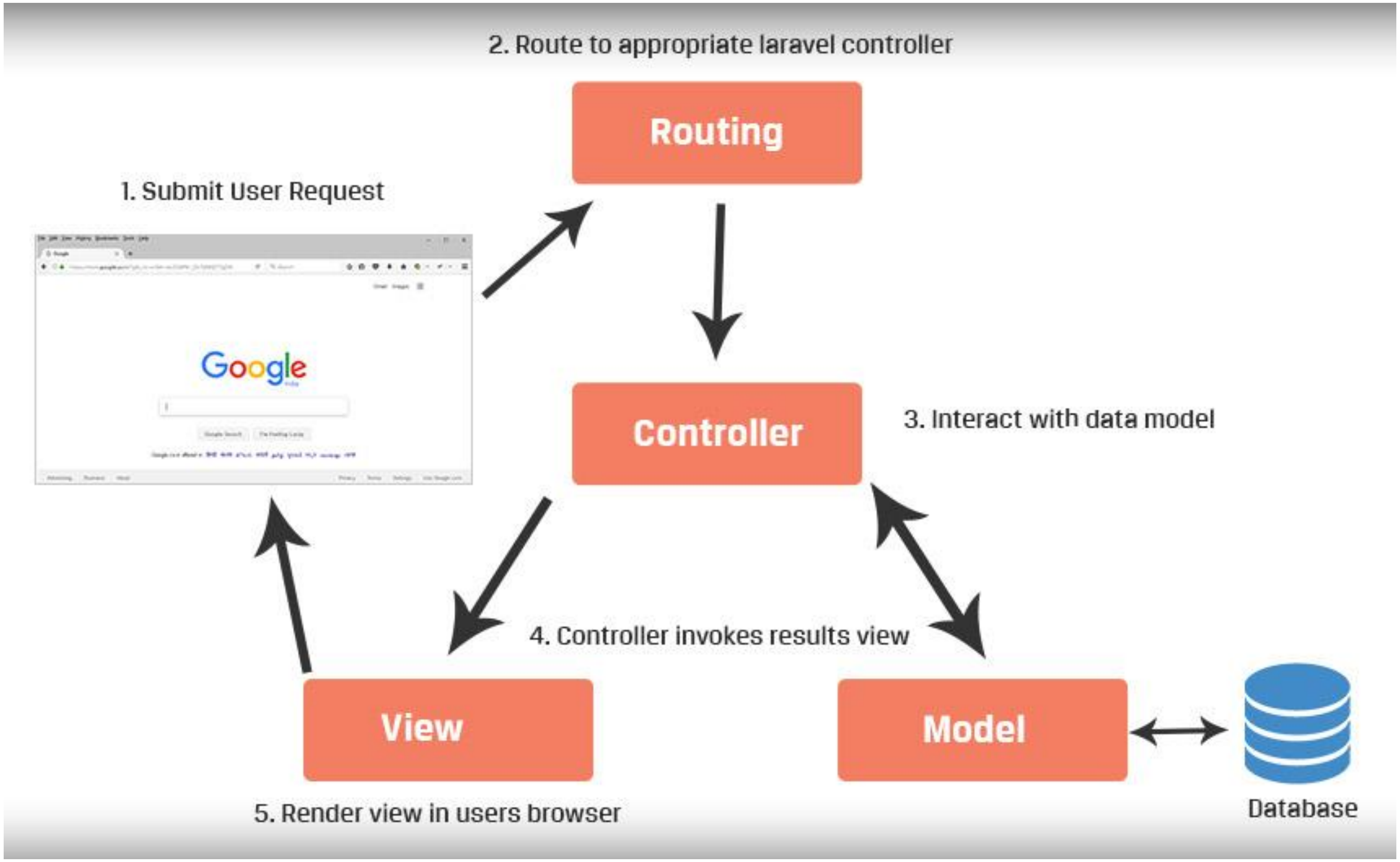

2. Route to appropriate laravel controller
Routing
1. Submit User Request
Google
Controller
3. Interact with data model
4. Controller invokes results view
View
Model
Database
5. Render view in users browser

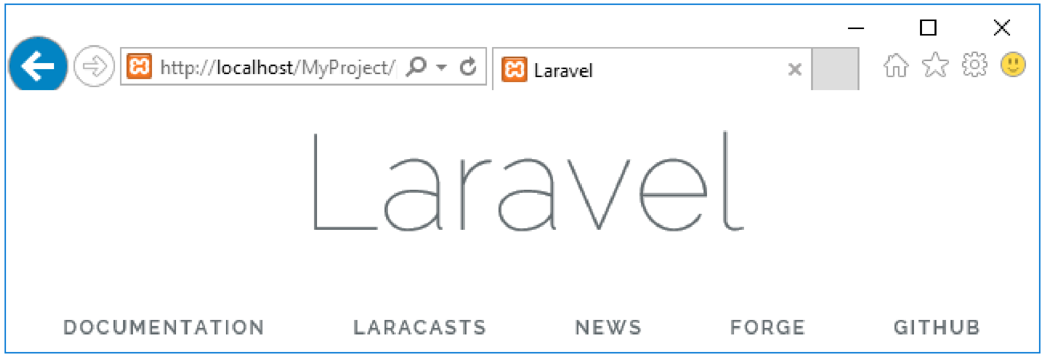
Image from: http://www.savecontactform7.com/everything-you-need-to-know-about-laravel-framework

Basic Routing

- Laravel routes: providing a very simple and expressive method of defining routes:

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

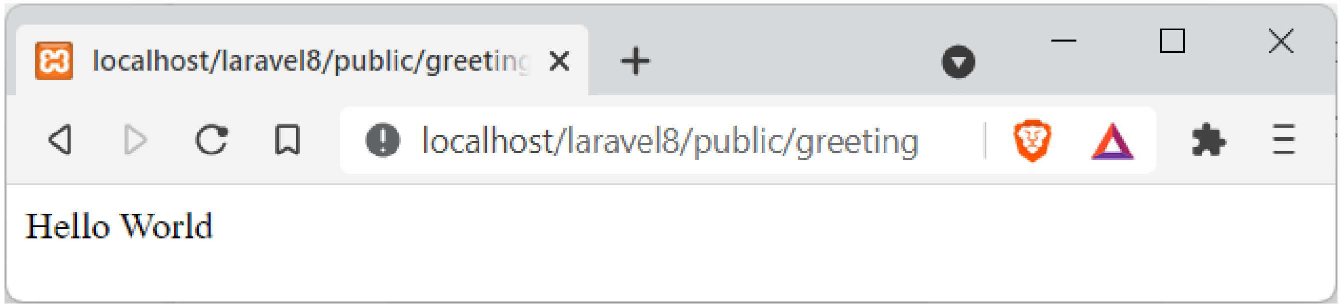
- For most applications, you will begin by defining routes in your **routes/web.php** file.
- Test: *http://localhost/MyProject/public/*



Basic Routing

```
Route::get ( 'greeting', function () {
      return 'Hello World';
} );
```

– Test: *http://localhost/laravel8/public/greeting*


localhost/laravel8/public/greeting

Hello World

Available Router Methods

- The router allows you to register routes that respond to any HTTP verb:

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

Responds to multiple HTTP

– Using the **match** method.

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

– Or, register a route that responds to all HTTP verbs using the **any** method.

```
Route::any ('foo', function () {
        return 'Hello World';
} );
```

CSRF Protection

– Any HTML forms pointing to POST, PUT, or DELETE routes that are defined in the web routes file should include a CSRF token field.
– Otherwise, the request will be rejected.

```
<form method="POST" action="/profile">
    @csrf
    ...
</form>
```

Redirect Routes

- If you are defining a route that redirects to another URI, you may use the Route::redirect method.

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

- By default, Route::redirect returns a **302** status code.
- Customize the status code using the optional third parameter:

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

301: chuyển hướng vĩnh viễn.
302: chuyển hướng tạm thời.

View Routes

- Route::view: return a view
- The view method accepts a URI as its first argument and a view name as its second argument.

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

In addition, you may provide an <u>array of data </u>to pass to the view as an optional <u>third argument</u>:

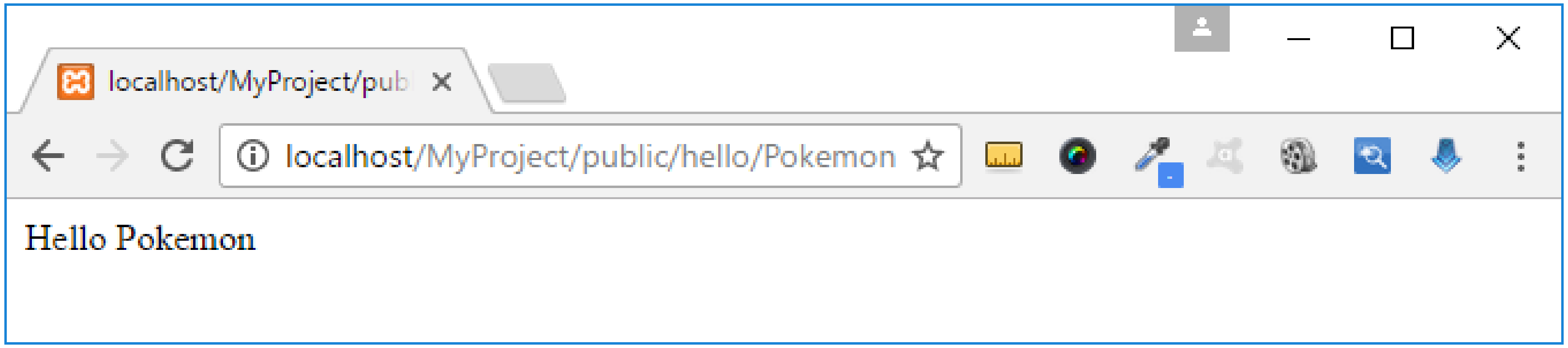
```
Route::view('/welcome', 'welcome', ['name' => ‘Tom']);
```

Route Parameters

– You may need to capture a user's ID from the URL. You may do so by defining route parameters:

```
Route::get ( 'hello/{name}', function ($name) {
        return 'Hello ' . $name;
} );
```

– EX: http://.../public/hello/Pokemon



```
localhost/MyProject/public/hello/Pokemon
```

```
Hello Pokemon
```

Route Parameters

– You may define as many route parameters as required by your route:

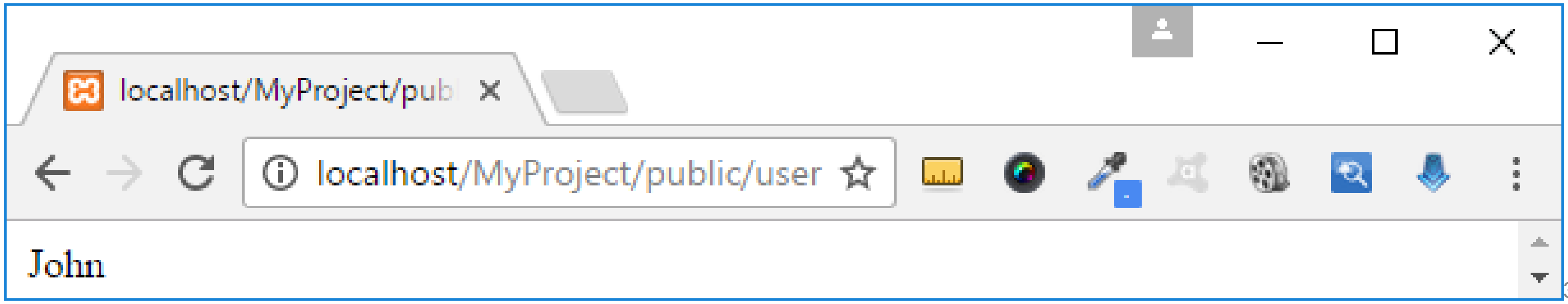
```
Route::get ( 'posts/{post}/comments/{comment}',
function ($postId, $commentId) {
      //
} );
```

– Note:
  - Route parameters are always encased within **{}** braces and should consist of alphabetic characters.
  - Route parameters may **not** contain a **-** character. Use an underscore (**_**) instead.

Optional Parameters

– Placing a **?** mark after the parameter name. Make sure: a default value

```
Route::get ( 'user/{name?}', function ($name = null) {
       if ($name == null)
              //Response to …
       else
              //Response to …
} );

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



John

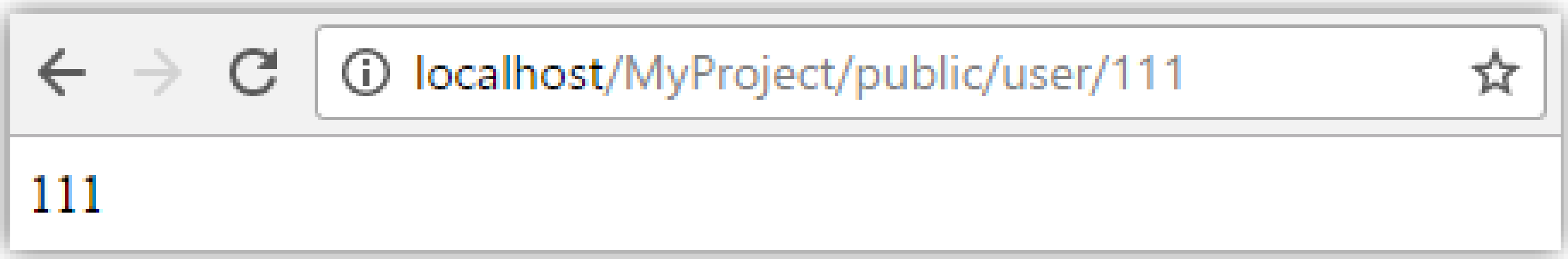
Regular Expression Constraints

– Constrain the format of your route parameters using the **where** method on a route instance.

```
Route::get ( 'user/{name}', function ($name) {
       return $name;
} )->where ( 'name', '[A-Za-z]+' );


Route::get ( 'user/{id}', function ($id) {
       return $id;
} )->where ( 'id', '[0-9]+' );


Route::get ( 'user/{id}/{name}', function ($id, $name) {
       return $id . '' . $name;
} )->where ( [ 'id' => '[0-9]+','name' => '[a-z]+' ] );
```

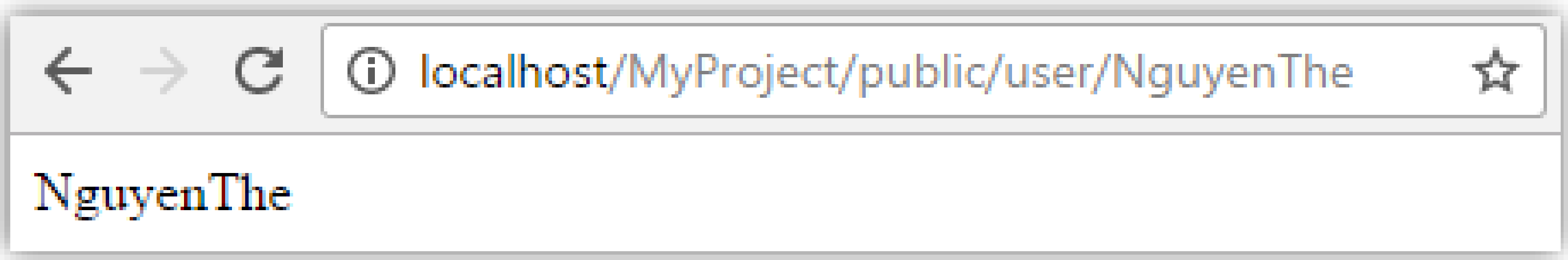
Regular expression editor: https://rubular.com/

Regular Expression Constraints



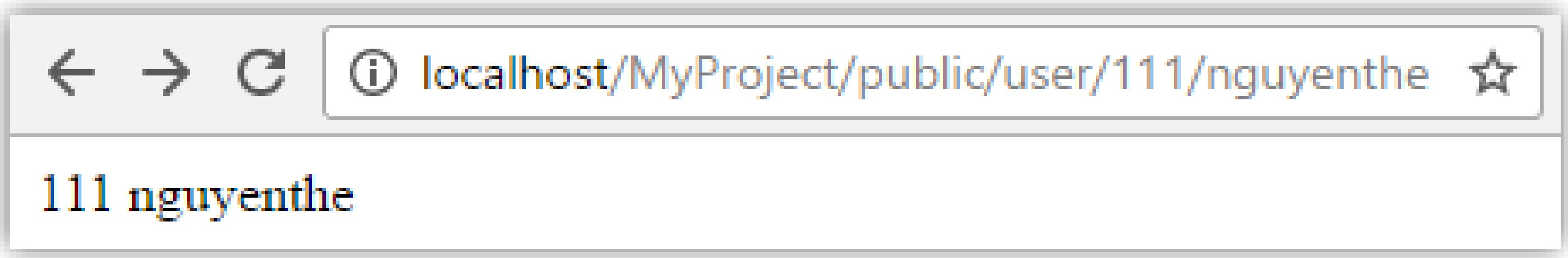
```
localhost/MyProject/public/user/111
```

```
111
```



```
localhost/MyProject/public/user/NguyenThe
```

```
NguyenThe
```



```
localhost/MyProject/public/user/111/nguyenthe
```

```
111 nguyenthe
```

Regular Expression Constraints

[abc]	A single character of: a, b, or c
[^abc]	Any single character except: a, b, or c
[a-z]	Any single character in the range a-z
[a-zA-Z]	Any single character in the range a-z or A-Z
^	Start of line
$	End of line
\A	Start of string
\z	End of string

Regular Expression Constraints

.	Any single character
\s	Any whitespace character
\S	Any non-whitespace character
\d	Any digit
\D	Any non-digit
\w	Any word character (letter, number, underscore)
\W	Any non-word character
\b	Any word boundary

Regular Expression Constraints

(...)	Capture everything enclosed
(a|b)	a or b
a?	Zero or one of a
a*	Zero or more of a
a+	One or more of a
a{3}	Exactly 3 of a
a{3,}	3 or more of a
a{3,6}	Between 3 and 6 of a

options:
i case insensitive
m make dot match newlines
x ignore whitespace in regex

o perform #{...} substitutions only once

Regular Expression Constraints - helper methods

Helper methods that allow you to quickly add pattern constraints

```
Route::get('/user/{id}/{name}', function ($id, $name) {
    //
})->whereNumber('id')->whereAlpha('name');

Route::get('/user/{name}', function ($name) {
    //
})->whereAlphaNumeric('name');

Route::get('/user/{id}', function ($id) {
    //
})->whereUuid('id');
```

where($name, '[0-9]+')

where($name, '[a-zA-Z]+')

where($name, '[a-zA-Z0-9]+')

where($name, '[\da-fA-F]{8}-[\da-fA-F]{4}-[\da-fA-F]{4}-[\da-fA-F]{4}-[\da-fA-F]{12}') (định dạng UUID)

Global Constraints

– A route parameter to always be constrained by a given regular expression, use the **pattern** method.

– Define these patterns in the **boot** method of your RouteServiceProvider: *app\Providers\RouteServiceProvider.php*

```
public function boot(){
     Route::pattern('id', '[0-9]+');
     parent::boot();
}
```

– Once the pattern has been defined, it is automatically applied to all routes using that parameter name:

```
Route::get('user/{id}', function ($id) {
    // Only executed if {id} is numeric...
});
```

Named Routes

– The convenient generation of URLs or redirects for specific routes.
– **name** method:

```
Route::get ( 'user/profile', function () {
        //
} )->name ( 'profile' );
```

– You may also specify route names for **controller** actions:

```
Route::get('user/profile',
'UserController@showProfile')->name('profile');
```

Generating URLs To Named Routes

– Use the route's name when generating URLs or redirects via the global **route** function:

```
// Generating URLs...
$url = route('profile');

// Generating Redirects...
return redirect()->route('profile');
```

– If the named route defines parameters, you may pass the parameters as the second argument to the **route** function.

```
Route::get('user/{id}/profile', function ($id) {
//
})->name('profile');
$url = route('profile', ['id' => 1]);
```

Generating URLs To Named Routes

– Parameters in the array: those key / value pairs will automatically be added to the generated URL's query string

```
Route::get('/user/{id}/profile', function ($id) {
    //
})->name('profile');

$url = route('profile', ['id' => 1, 'photos' => 'yes']);
```

.../user/1/profile?photos=yes

Route Groups

- Share route attributes:
  - Middleware,
  - Namespaces,
  - Sub-Domain Routing,
  - Route Prefixes
- Shared attributes are specified in an array format as the first parameter to the **Route::group()** method.

Route Prefixes

- The **prefix** group attribute may be used to prefix each route in the group with a given URI.
  - For example, you may want to prefix all route URIs within the group with product.

Laravel 5.x:

```
Route::group(['prefix' => 'product'], function () {
    Route::get('add', function () {
        echo 'Add content';
    });
    Route::get('edit', function () {
        echo 'Edit content';
    });
    Route::get('del', function () {
        echo 'Delete content';
    });
});
```

http://localhost/laravel/public/**product/add**

Route Prefixes

– Laravel 7.x, 8.x

```
Route::prefix('admin')->group(function () {
    Route::get('users', function () {
        // Matches The "/admin/users" URL
    });
});
```

Route Prefixes

```
Route::prefix('admin')->group(function () {
   Route::prefix('users')->group(function () {
      Route::get('/', function () {
         return view('admin.users.index');
      });
      Route::get('add', function () {
         echo "admin/users/add";
      });
      Route::get('edit', function () {
         echo "admin/users/edit";
      });
      Route::get('delete', function () {
         echo "admin/users/delete";
      });
   });
   Route::get('category', function () {
      echo "admin/category";
   });
   Route::get('news', function () {
      echo "admin/news";
   });
});
```

```
/*
 * admin/users
 * admin/users/add
 * admin/users/edit
 * admin/users/delete
 * admin/category
 * admin/news
 */
```

Route Name Prefixes

- Prefix each route name in the group with a given string
- The trailing . character in the prefix

```
Route::name('admin.')->group(function () {
   Route::get('/users', function () {
      // Route assigned name "admin.users"...
   })->name('users');
});
```

Middleware

– To assign middleware to all routes within a group, you may use the **middleware** key in the group attribute array.

```
Route::group ( [ 'middleware' => 'auth' ], function () {
	Route::get ( '/', function () {
		// Uses Auth Middleware
	} );

	Route::get ( 'user/profile', function () {
		// Uses Auth Middleware
	} );
} );
```

Middleware

– Laravel 7.x

```
Route::get('/', function () {
    //
})->middleware('web');

Route::group(['middleware' => ['web']], function () {
    //
});

Route::middleware(['web', 'subscribed'])->group(function ()
{
    //
});
```

Middleware

- Create a new middleware:

```
php artisan make:middleware EnsureTokenIsValid
```

- This command will place a new **EnsureTokenIsValid** class within your **app/Http/Middleware** directory.

- In this middleware:
  - We will only allow access to the route if the supplied **token** input **matches** a specified value.
  - Otherwise, we will redirect the users **back** to the home URI

Middleware

```
namespace App\Http\Middleware;
use Closure;

class EnsureTokenIsValid
{
   /**
    * Handle an incoming request.
    * @param  \Illuminate\Http\Request  $request
    * @param  \Closure  $next
    * @return mixed
    */
    public function handle($request, Closure $next)
    {
       if ($request->input('token') !== 'my-secret-token') {
          return redirect('home');
       }

       return $next($request);
    }
}
```

If the given token does not match our secret token, the middleware will return an HTTP redirect to the client;

Otherwise, the request will be passed further into the application, call the $next callback with the $request.

Middleware & Responses

```
namespace App\Http\Middleware;

use Closure;

class BeforeMiddleware
{
   public function handle($request, Closure $next)
   {
      // Perform action

      return $next($request);
   }
}
```

A middleware can perform tasks before or after passing the request deeper into the application.

For example, the following middleware would perform some task **before** the **request** is handled

Middleware & Responses

```
namespace App\Http\Middleware;

use Closure;

class AfterMiddleware
{
    public function handle($request, Closure $next)
    {
        $response = $next($request);

        // Perform action

        return $response;
    }
}
```

This middleware would perform its task **after** the request is handled by the application

Namespaces

– Use-case for route groups is assigning the same PHP namespace to a group of controllers using the **namespace** parameter in the group array:

```
Route::group ( [ 'namespace' => 'Admin' ], function () {
        // Controllers Within The
        //"App\Http\Controllers\Admin" Namespace
} );
```

– Default: the RouteServiceProvider includes your route files within a namespace group, allowing you to register controller routes without specifying the full App\Http\Controllers namespace prefix.

Namespaces

– Laravel 7.x

```
Route::namespace('Admin')->group(function () {
    // Controllers Within The "App\Http\Controllers\Admin" Namespace
});
```

Sub-Domain Routing

- Route groups may also be used to handle sub-domain routing.
- The sub-domain may be specified using the **domain** key on the group attribute array:

```
Route::group(['domain' => '{account}.myapp.com'], function () {
    Route::get('user/{id}', function ($account, $id) {
        //
    });
});
```

Sub-Domain Routing

– Laravel 7.x, 8.x

```
Route::domain('{account}.myapp.com')->group(function () {
   Route::get('user/{id}', function ($account, $id) {
      //
   });
});
```

Implicit Binding

- Laravel automatically resolves Eloquent models defined in routes or controller actions whose variable names match a route segment name. For example:

```
Route::get('api/users/{user}', function (App\User $user) {
        return $user->email;
});
```

- In this example:
  - Since the Eloquent $user variable defined on the route matches the {user} segment in the route's URI,
  - Laravel will automatically inject the model instance that has an ID matching the corresponding value from the request URI.

Customizing The Key Name

– Model binding to use a database column other than id when retrieving a given model class:
  - Override the **getRouteKeyName** method on the Eloquent model:

```
// Get the route key for the model.
// @return string

public function getRouteKeyName(){
    return 'slug';
}
```

Explicit Binding

- Use the router's **model** method.
  - In the **boot** method of the **RouteServiceProvider** class:

```
public function boot(){
        parent::boot();
        Route::model('user', App\User::class);
}
```

- Next, define a route that contains a {user} parameter:

```
Route::get ( 'profile/{user}', function (App\User $user) {
        //
} );
```

- Since we have bound all {user} parameters to the App\User model,
  - A User instance will be injected into the route.
  - For example, a request to profile/1 will inject the User instance from the database which has an ID of 1.

Customizing The Resolution Logic

- Use your own resolution logic
  - Use the **Route::bind** method.

```
public function boot(){
	parent::boot();
	Route::bind('user', function ($value) {
		return App\User::where('name', $value)->first();
	});
}
```

Form Method Spoofing

- HTML forms do not support PUT, PATCH or DELETE actions.
  - So, when defining PUT, PATCH or DELETE routes that are called from an HTML form, you will need to add a hidden _method field to the form.
  - The value sent with the _method field will be used as the HTTP request method.

```
<form action="/foo/bar" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

- Use the method_field helper to generate the _method input:

```
{{ method_field('PUT') }}
```

Accessing The Current Route

– To access information about the route handling the incoming request.

```
$route = Route::current ();
$name = Route::currentRouteName ();
$action = Route::currentRouteAction ();
```

Views

1. Creating Views
2. Passing Data To Views
3. Sharing Data With All Views
4. View Composers

Views

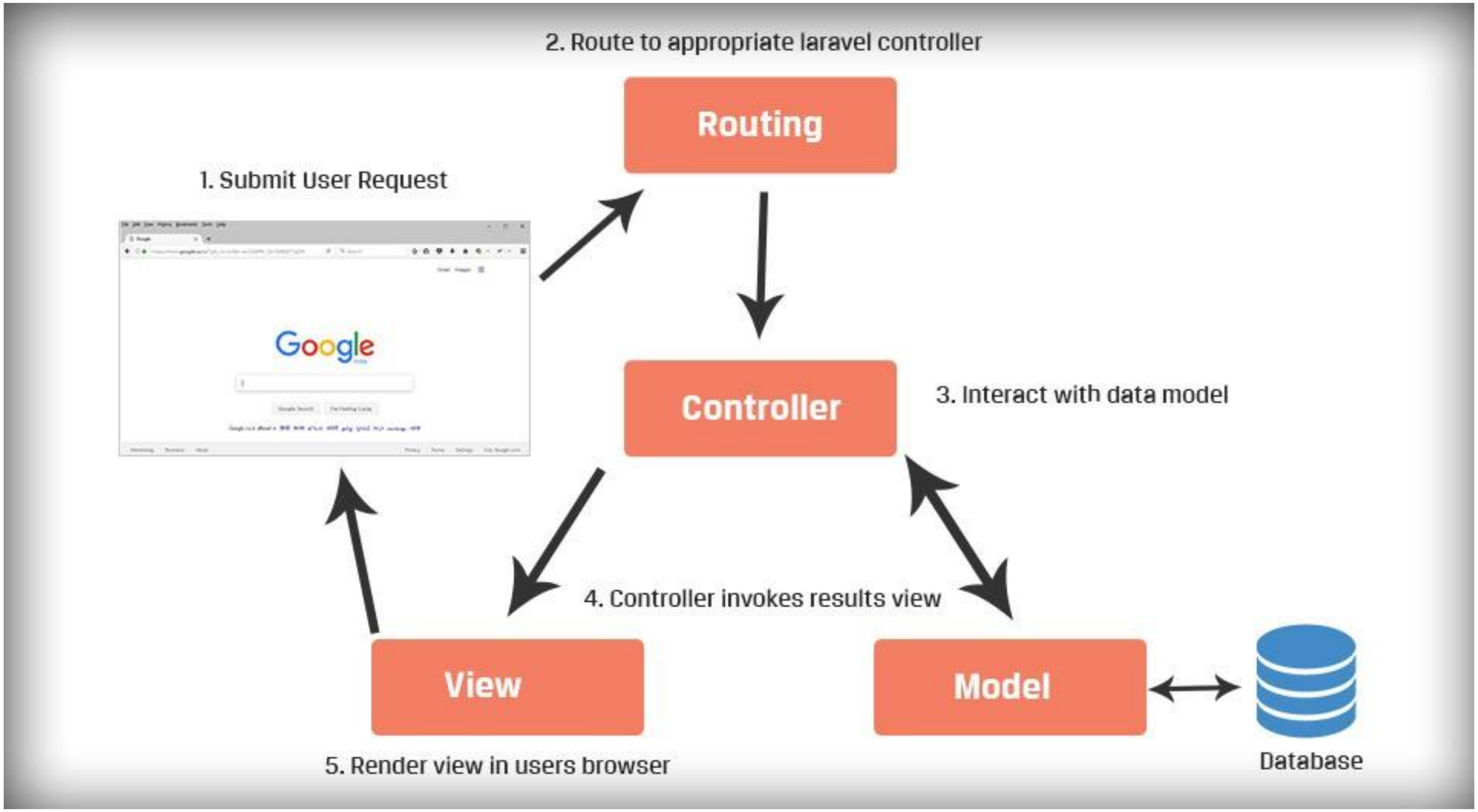

2. Route to appropriate laravel controller
Routing
1. Submit User Request
Google
Controller
3. Interact with data model
4. Controller invokes results view
View
Model
Database
5. Render view in users browser

Image from: https://medium.com/@dannyhuang_75970/learning-laravel-controllers-101

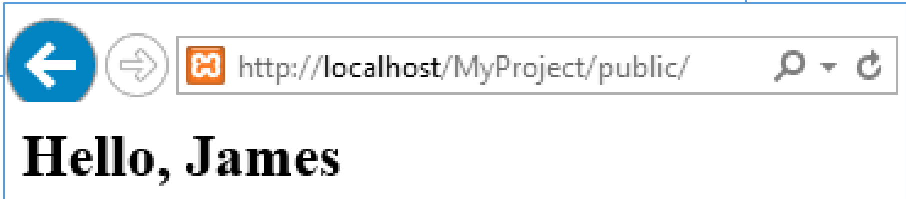
Views

- Views contain the HTML served by your application and separate your controller / application logic from your presentation logic.
- Views are stored in the resources/views directory.

```
<!-- View stored in resources/views/greeting.blade.php -->
<html>
   <body>
      <h2>Hello, {{ $name }}</h2>
   </body>
</html>
```

- This view is stored at resources/views/greeting.blade.php

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


http://localhost/MyProject/public/

Hello, James

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

- The first argument: the name of the view.
- The second argument is an array of data that should be made available to the view.

❖ Views may also be nested within sub-directories of the resources/views directory.

- For example, if your view is stored at resources/views/admin/profile.blade.php, you may reference it like so:

```
return view('admin.profile', $data);
```

Determining If A View Exists

– If you need to determine if a view exists, you may use the **View** facade. The **exists** method will return **true** if the view exists:

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

if (View::exists ( 'emails.customer' )) {
      //
}
```

Passing Data To Views

– An array of data to views:

```
return view('greetings', [  'name' => 'Victoria',
                            'job'=>'Developer,
                            'more_data'=> $data
]);
```

– Data should be an array with **key/value** pairs.
  - Inside your view, you can then access each value using its corresponding key, such as <?php echo **$key**; ?>.
  - You may use the **with** method to add individual pieces of data to the view:

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

Sharing Data With All Views

– Share a piece of data with all views that are rendered by your application.
  - Using the view facade's **share** method within a service provider's **boot** method.
  - Add them to the **AppServiceProvider.**

```
namespace App\Providers;
use Illuminate\Support\ServiceProvider;
class AppServiceProvider extends ServiceProvider{
    public function boot() {
        View::share('key', 'value');
    }
    public function register() {
        //
    }
}
```

Sharing Data With All Views - Example

– **Step 1** – Add the following line in routes/web.php

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

Route::get('/test2', function(){
    return view('test2');
});
```

– **Step 2** – Create two view files — **test.php** and **test2.php** with the same code. These are the two files which will share data.

resources/views/test.php & resources/views/test2.php

```
<html>
<body>
    <h1><?php echo $name; ?></h1>
</body>
</html>
```

```
<html>
<body>
    <h1><?php echo $name; ?></h1>
</body>
</html>
```

Sharing Data With All Views - Example

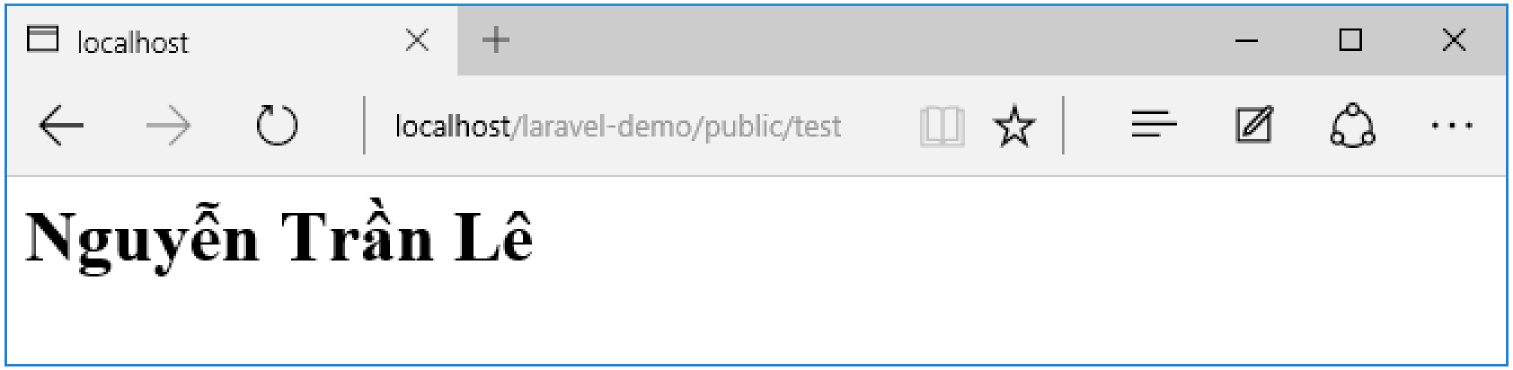
- **Step 3** – Change the code of boot method in the file app/Providers/AppServiceProvider.php as shown below.
- (Here, we have used share method and the data that we have passed will be shared with all the views.)
- app/Providers/AppServiceProvider.php

```
namespace App\Providers;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider{
        public function boot(){
                view()->share('name', 'Nguyễn Trần Lê');
        }

        public function register() {
                //
        }
}
```

Sharing Data With All Views - Example

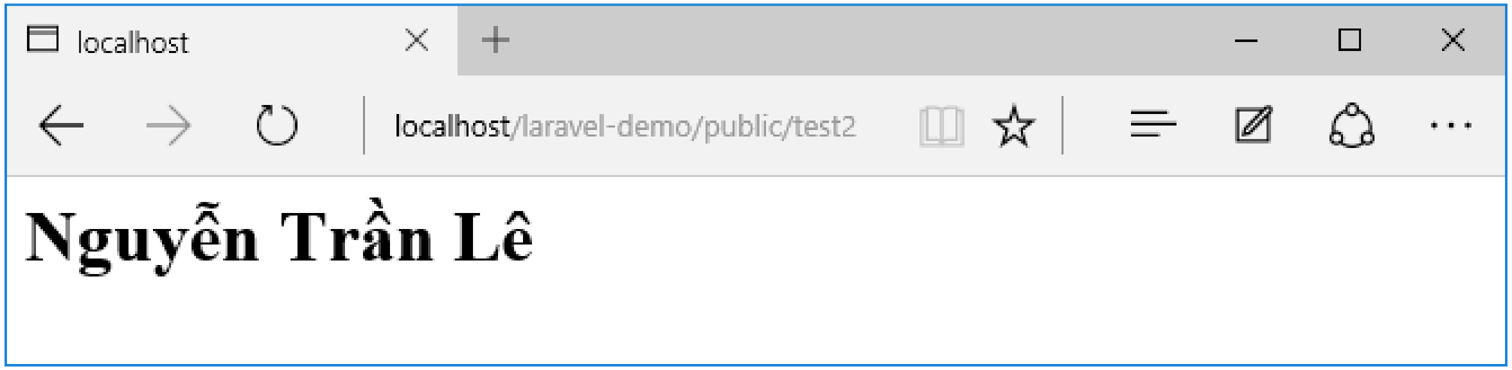
– **Step 4** – Visit the following URLs.

- http://localhost/laravel-demo/public/test



Nguyễn Trần Lê

- http://localhost/laravel-demo/public/test2



Nguyễn Trần Lê

View Composers

- Callbacks or class methods that are called when a view is rendered.
- For this example:
  - Let's register the view composers within a service provider.
  - We'll use the **View** facade to access Illuminate\Contracts\View\Factory.
  - Laravel does not include a default directory for view composers.
    - For example, you could create an App\Http\ViewComposers directory:

View Composers

– Create an App\Http\ViewComposers directory:

```
namespace App\Providers;
use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;
class ComposerServiceProvider extends ServiceProvider {
        public function boot() {
                // Using class based composers...
                View::composer ( 'profile', 'App\Http\ViewComposers\ProfileComposer' );

                // Using Closure based composers...
                View::composer ( 'dashboard', function ($view) {
                //
                } );
        }
        public function register() {
                //
        }
}
```

View Composers

– Now that we have registered the composer,

– The **ProfileComposer@compose** method will be executed each time the profile view is being rendered:

```
namespace App\Http\ViewComposers;
use Illuminate\View\View;
use App\Repositories\UserRepository;
class ProfileComposer{
        protected $users;
        public function __construct(UserRepository $users){
                // Dependencies automatically resolved by service container...
                $this->users = $users;
        }
        public function compose(View $view)  {
                $view->with('count', $this->users->count());
        }
}
```

Attaching A Composer To Multiple Views

– Attach a view composer to multiple views:
  - the first argument to the **composer** method:

```
View::composer(
['profile', 'dashboard'],
'App\Http\ViewComposers\MyViewComposer'
);
```

– The ***** character as a wildcard, attach a composer to all views:

```
View::composer('*', function ($view) {
//
});
```

View Creators

- View creators are very similar to view composers;
- They are executed immediately after the view is instantiated instead of waiting until the view is about to render.

```
View::creator('profile',
 'App\Http\ViewCreators\ProfileCreator');
```

BLADE TEMPLATES

1. Introduction
2. Template Inheritance
   - Defining A Layout
   - Extending A Layout
3. Displaying Data
   - Blade & JavaScript Frameworks
4. Control Structures
   - If Statements
   - Loops
   - The Loop Variable
   - Comments
   - PHP
5. Including Sub-Views
   - Rendering Views For Collections
6. Stacks

BLADE TEMPLATES

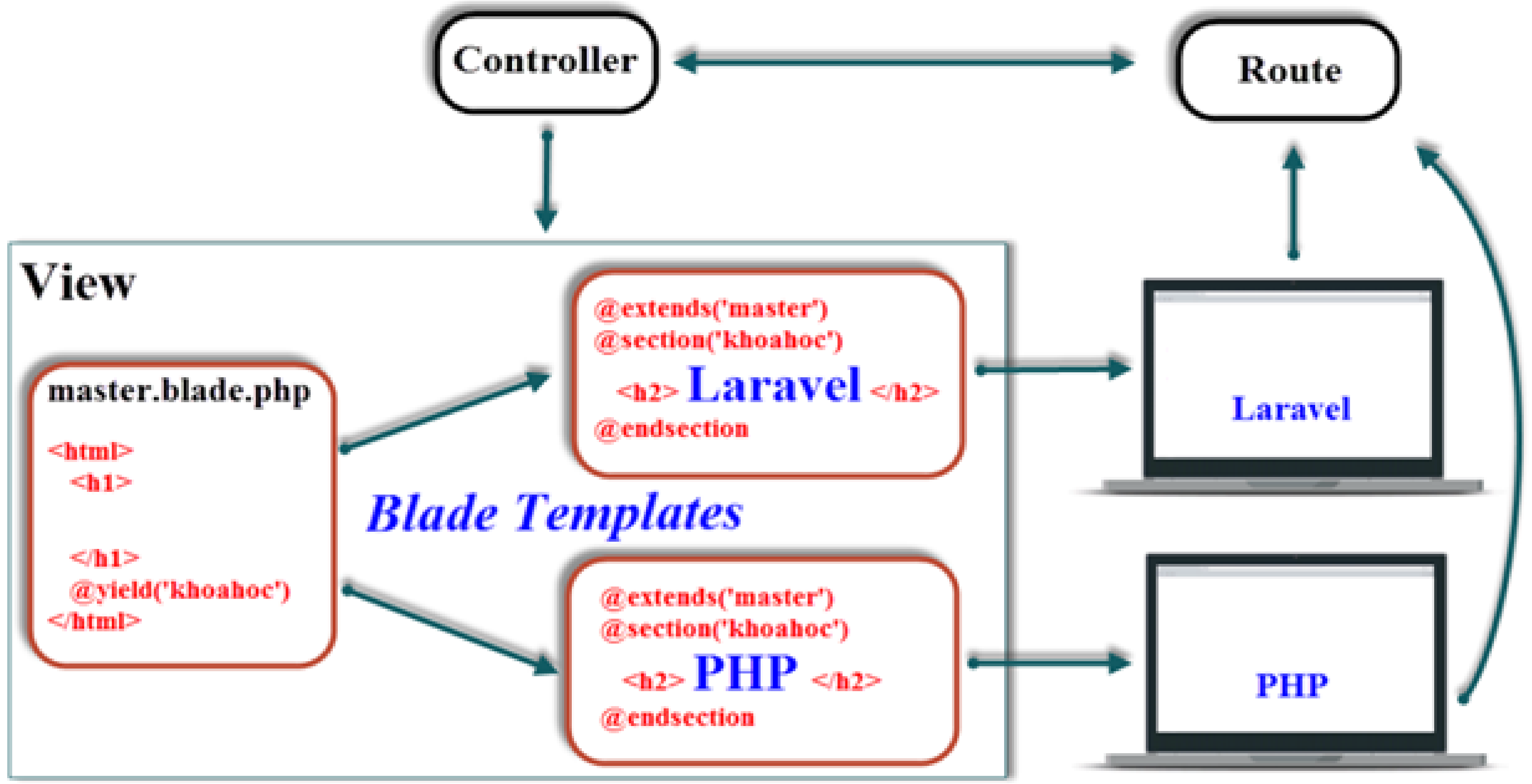


Image from: khoapham.vn

Giới thiệu Blade Templates

– Cho phép sử dụng code PHP thuần ở trong view.
– Các Blade view được compiled từ code PHP và được cache cho đến khi chúng được chỉnh sửa => không làm tăng thêm bộ nhớ.
– Sử dụng đuôi **.blade.php**
  - Lưu trong resources/views.

Layout

2 lợi ích khi sử dụng Blade: template inheritance và sections.

```
<!-- Stored in resources/views/layouts/app.blade.php -->
<html>
  <head>
    <title>App Name - @yield('title')</title>
  </head>
  <body>
    @section('sidebar')
      This is the master sidebar.
    @show

    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```

@yield('title') is used to display the **value** of the title
@section('sidebar') is used to define a section **named** sidebar
@show is used to display the **contents** of a section
@yield('content') is used to display the **contents** of content

Kế thừa một layout

- Khi tạo một trang con, sử dụng Blade **@extends** directive để chỉ ra layout của trang con này "inherit" từ đâu.
- Views kế thừa một Blade layout có thể inject nội dung vào trong sections using **@section** directives của layout.

```
<!-- Stored in resources/views/child.blade.php -->
@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
   @parent

   <p>This is appended to the master sidebar.</p>
@endsection


@section('content')
   <p>This is my body content.</p>
@endsection
```

Kế thừa một layout

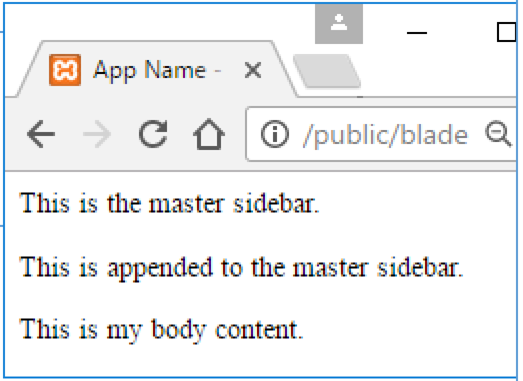
– Blade views có thể được trả về từ routes bằng cách sử dụng hàm global view

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

```
<!– layouts/app.blade.php -->

<html>
  <head>
    <title>App Name - @yield('title')</title>
  </head>
  <body>
    @section('sidebar')
      This is the master sidebar.
    @show

    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```

```
<!-- child.blade.php -->

@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
  @parent

  <p>This is appended to the master
sidebar.</p>
@endsection

@section('content')
  <p>This is my body content.</p>
@endsection
```

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



This is the master sidebar.

This is appended to the master sidebar.

This is my body content.

```
<html>
   <head>
      <title>App Name - Page Title</title>
   </head>
   <body>
        This is the master sidebar.

        <p>This is appended to the master sidebar.

      <div class="container">
           <p>This is my body content.</p>
      </div>
   </body>
</html>
```

Hiển thị dữ liệu

- Truyền dữ liệu vào Blade views bằng cách đặt biến trong cặp ngoặc nhọn.

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

- Hiển thị nội dung của biến name variable như sau:

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

Hiển thị dữ liệu nếu tồn tại

– Cú pháp kiểm tra biến:

```
{{ isset($name) ? $name : 'Default' }}
```

– Hoặc:

```
{{ $name or 'Default' }}
```

Hiện dữ liệu chưa Unescaped

- Mặc định, cặp {{ }} được tự động gửi qua hàm htmlentities của PHP để ngăn chặn tấn công XSS.
- Nếu không muốn dữ liệu bị escaped, sử dụng cú pháp:

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

```
<?php echo $name; ?>
```

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

```
<?php echo htmlentities($name); ?>
```

Blade & JavaScript Frameworks

– Vì nhiều JavaScript frameworks cũng sử dụng cặp "ngoặc nhọn" để cho biết một biểu thức cần được hiển thị lên trình duyệt.
  - Có thể sử dụng biểu tượng **@** để nói cho Blade biết được biểu thức này cần được giữ lại.

```
<h1>Laravel</h1> Hello, @{{ name }}.
```

The @verbatim Directive

– Nếu muốn hiển thị biến JavaScript trong phần lớn template
  ➔ Bọc chúng trong khối directive

```
@verbatim
    <div class="container"> Hello, {{ name }}. </div>
```

Cấu trúc điều kiện

– Cấu trúc if:
  - @if, @elseif, @else, và @endif.

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

```
<?php if ($records === 1){?>
…
<?php }?>
```

Vòng lặp

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

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

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

Continue

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

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

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

Continue

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

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

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

Biến vòng lặp

– Trong vòng lặp:
  - Một biến $loop sẽ tồn tại bên trong vòng lặp.
  - Cho phép truy cập một số thông tin hữu ích của vòng lặp như index của vòng lặp hiện tại, vòng lặp đầu, vòng lặp cuối

```
@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
```

Biến vòng lặp

– Nếu vòng lặp lồng nhau, truy cập biến $loop của vòng lặp tra qua thuộc tính parent:

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

Biến $loop

Thuộc tính	Miêu tả
$loop->index	Chỉ số index hiện tại của vòng lặp (starts at 0).
$loop->iteration	Các vòng lặp hiện tại (starts at 1).
$loop->remaining	Số vòng lặp còn lại.
$loop->count	Tổng số vòng lặp.
$loop->first	Vòng lặp đầu tiên.
$loop->last	Vòng lặp cuối cùng.
$loop->depth	Độ sâu của vòng lặp hiện tại.
$loop->parent	Biến parent loop của vòng lặp trong 1 vòng lặp lồng.

Comments

- Blade cho phép comment trong view.

```
{{-- This comment will not be present in the rendered HTML --}}
```

Including Sub-Views

– @include: chèn một Blade view từ một view khác.
– Tất cả các biến tồn tại trong view cha đều có thể sử dụng ở view chèn thêm.

```
<div>
  @include('shared.errors')

  <form>
    <!-- Form Contents -->
  </form>
</div>
```

– Truyền một mảng dữ liệu bổ sung cho view

```
@include('view.name', ['some' => 'data'])
```

Rendering Views cho Collections

– Có thể kết hợp vòng lặp và view chèn thêm trong một dòng với **@each** directive

```
@each('view.name', $jobs, 'job')
```

- Tham số thứ nhất là tên của view partial để render các element trong mảng hay collection.
- Tham số thứ hai là một mảng hoặc collection mà bạn muốn lặp
- Tham số thứ ba là tên của biến được gán vào trong vòng lặp bên view.

Stacks

– Để xác định thư viện JavaScript libraries cần cho view con:
  - Blade cho phép đẩy tên stack để cho việc render ở một vị trí nào trong view hoặc layout khác.

```
@push('scripts')
   <script src="/example.js"></script>
@endpush
```

– Có thể đẩy một hoặc nhiều vào stack.

```
<head>
   <!-- Head Contents -->
   @stack('scripts')
</head>
```

Mở rộng Blade

– Tùy biến directives bằng phương thức directive. Khi trình viên dịch của Blade gặp directive, nó sẽ gọi tới callback được cung cấp với tham số tương ứng.

```
namespace App\Providers;
use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider {
        public function boot() {
                Blade::directive('datetime', function($expression) {
                   return "<?php echo $expression->format('m/d/Y H:i'); ?>";
                });
        }
        public function register() {
                //
        }
}
```

Forms & HTML

- Laravel provides various in built tags to handle HTML forms easily and securely.
- All the major elements of HTML are generated using Laravel.
- To support this, we need to add HTML package to Laravel using composer.

Forms & HTML - Instalation

- Begin by installing this package through Composer. Run the following from the terminal:

```
composer require "laravelcollective/html":"^5.3.0"
```

- Next, add your new provider to the providers array of config/app.php:

```
'providers' => [
  // ...
  Collective\Html\HtmlServiceProvider::class,
  // ...
],
```

- Finally, add two class aliases to the aliases array of config/app.php:

```
'aliases' => [
  // ...
   'Form' => Collective\Html\FormFacade::class,
   'Html' => Collective\Html\HtmlFacade::class,
  // ...
],
```

Forms & HTML - Opening A Form

```
{!! Form::open(['url' => 'foo/bar']) !!}
   //
{!! Form::close() !!}
```

– By default, a POST method will be assumed; however, you are free to specify another method:

```
echo Form::open(['url' => 'foo/bar', 'method' => 'put'])
```

– **Note**: Since HTML forms only support POST and GET, PUT and DELETE methods will be spoofed by automatically adding a _method hidden field to your form.

Forms & HTML - Opening A Form

– You may also open forms that point to named routes or controller actions:

```
echo Form::open(['route' => 'route.name'])
echo Form::open(['action' => 'Controller@method'])
```

– You may pass in route parameters as well:

```
echo Form::open(['route' => ['route.name', $user->id]])
echo Form::open(['action' => ['Controller@method',
$user->id]])
```

– If your form is going to accept file uploads, add a files option to your array:

```
echo Form::open(['url' => 'foo/bar', 'files' => true])
```

Forms & HTML - Label

– Generating A Label Element

```
echo Form::label('email', 'E-Mail Address');
```

– Specifying Extra HTML Attributes

```
echo Form::label('email', 'E-Mail Address', ['class' =>
'awesome']);
```

– ***Note***: *After creating a label, any form element you create with a name matching the label name will automatically receive an ID matching the label name as well.*

Forms & HTML - Text Input

- Generating A Text Input

```
echo Form::text('username');
```

- Specifying A Default Value

```
echo Form::text('email', 'example@gmail.com');
```

- ***Note**: The hidden and textarea methods have the same signature as the text method.*

Forms & HTML - Password Input

– Generating A Password Input

```
echo Form::password('password', ['class' => 'awesome']);
```

– Generating Other Inputs

```
echo Form::email($name, $value = null, $attributes = []);
echo Form::file($name, $attributes = []);
```

Forms & HTML - Checkbox Or Radio Input

– Generating A Checkbox Or Radio Input

```
echo Form::checkbox('name', 'value');
echo Form::radio('name', 'value');
```

– Generating A Checkbox Or Radio Input That Is Checked

```
echo Form::checkbox('name', 'value', true);
echo Form::radio('name', 'value', true);
```

Forms & HTML – Number, Date, File

- Generating A Number Input
```
echo Form::number('name', 'value');
```

- Generating A Date Input
```
echo Form::date('name', \Carbon\Carbon::now());
```

- Generating A File Input
```
echo Form::file('image');
```

Forms & HTML – Drop-Down Lists

- Generating A Number Input

```
echo Form::number('name', 'value');
```

- Generating A Drop-Down List

```
echo Form::select('size', ['L' => 'Large', 'S' => 'Small']);
```

- Generating A Drop-Down List With Selected Default

```
echo Form::select('size', ['L' => 'Large', 'S' => 'Small'], 'S');
```

- Generating a Drop-Down List With an Empty Placeholder
  - *This will create an <option> element with no value as the very first option of your drop-down.*

```
echo Form::select('size', ['L' => 'Large', 'S' => 'Small'], null,
['placeholder' => 'Pick a size...']);
```

Forms & HTML – Drop-Down Lists

– Generating a List With Multiple Selectable Options

```
echo Form::select('size', ['L' => 'Large', 'S' => 'Small'], null, ['multiple'
=> true]);
```

– Generating A Grouped List

```
echo Form::select('animal',[
    'Cats' => ['leopard' => 'Leopard'],
    'Dogs' => ['spaniel' => 'Spaniel'],
]);
```

– Generating A Drop-Down List With A Range

```
echo Form::selectRange('number', 10, 20);
```

– Generating A List With Month Names

– `echo Form::selectMonth('month');`

Forms & HTML – Buttons

– Generating A Submit Button

```
echo Form::submit('Click Me!');
```

– ***Note**: Need to create a button element? Try the button method. It has the same signature as submit.*

Forms & HTML – Generating URLs

- Generate a HTML link to the given URL.

  ```
  echo link_to('foo/bar', $title = null, $attributes = [], $secure = null);
  ```

- Generate a HTML link to the given asset.

  ```
  echo link_to_asset('foo/bar.zip', $title = null, $attributes = [], $secure =
  null);
  ```

- Generate a HTML link to the given named route.

  ```
  echo link_to_route('route.name', $title = null, $parameters = [],
  $attributes = []);
  ```

- Generate a HTML link to the given controller action.

  ```
  echo link_to_action('HomeController@getIndex', $title = null,
  $parameters = [], $attributes = []);
  ```

Forms & HTML – Example

– resources/views/form.php

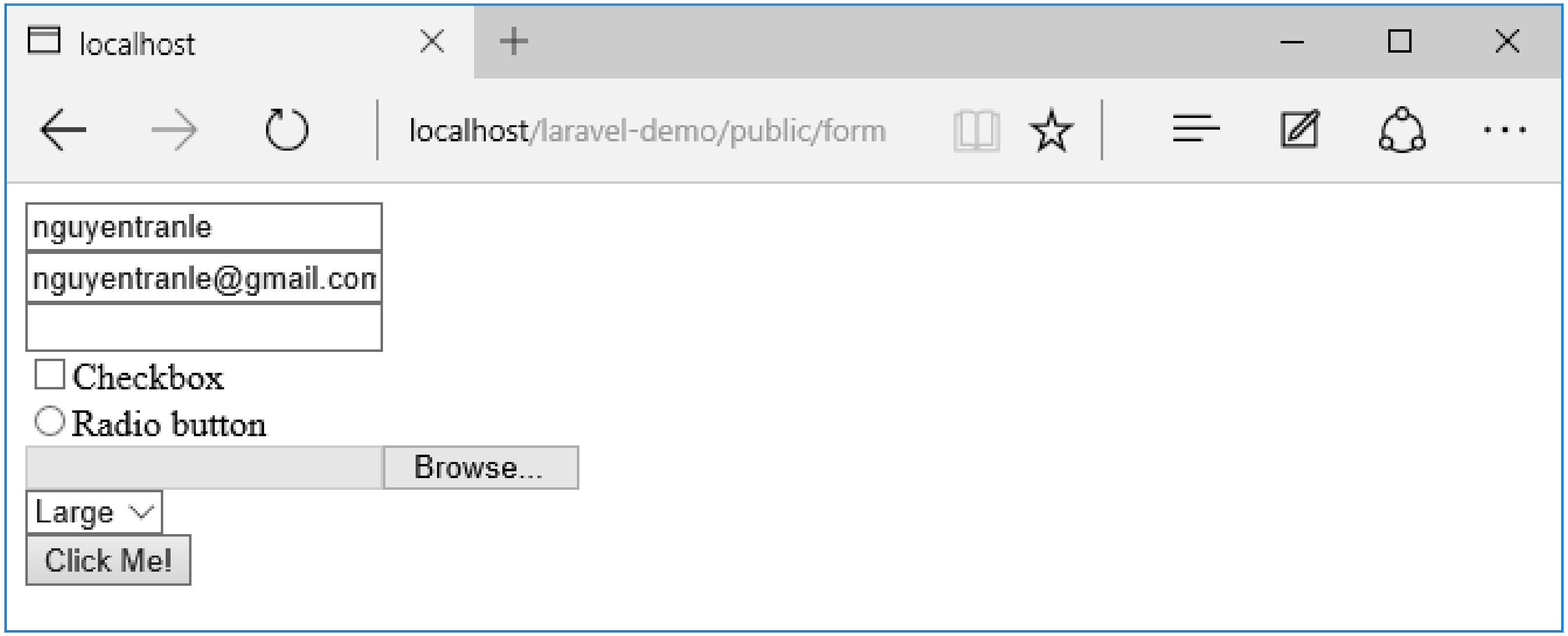
```
<html>
  <body>
    <?php
     echo Form::open(array('url' => 'foo/bar'));
        echo Form::text('username','nguyentranle') . '<br/>';

        echo Form::text('email', 'nguyentranle@gmail.com') . '<br/>';

        echo Form::password('password') . '<br/>';

        echo Form::checkbox('name', 'value') . 'Checkbox<br/>';

        echo Form::radio('name', 'value') . 'Radio button<br/>';

        echo Form::file('image') . '<br/>';

        echo Form::select('size', array('L' => 'Large', 'S' => 'Small')) . '<br/>';

        echo Form::submit('Click Me!');
     echo Form::close();
    ?>
  </body>
</html>
```

Forms & HTML – Example

– Routes/web.php

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

– Test: http://localhost/laravel-demo/public/form



nguyentranle
nguyentranle@gmail.con

☐ Checkbox
○ Radio button