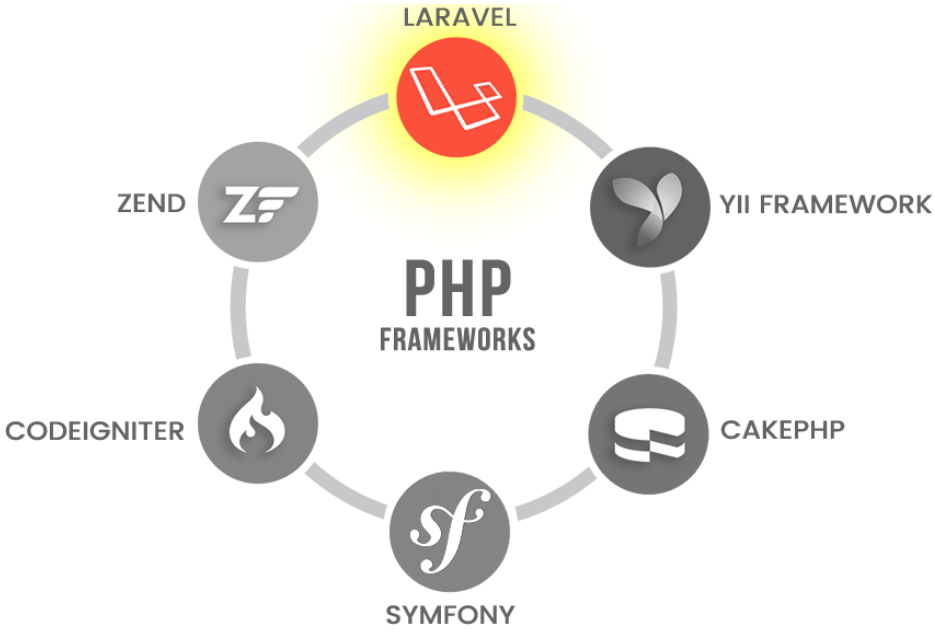
unidad didáctica 7

Framework Laravel



1. duración y criterios de evaluación

Duración estimada: ∞ sesiones.

Resultado de aprendizaje y criterios de evaluación:

2. instalar docker bitnami/Laravel

1. Lo primero de todo es crear una carpeta con el nombre del proyecto que vayamos a crear y nos metemos en ella.

   Por ejemplo, creamos el proyecto myapp-laravel dentro de nuestra carpeta de proyectos del módulo:

   ```
   $ mkdir ~/dwes/proyectos/myapp-laravel
   ```

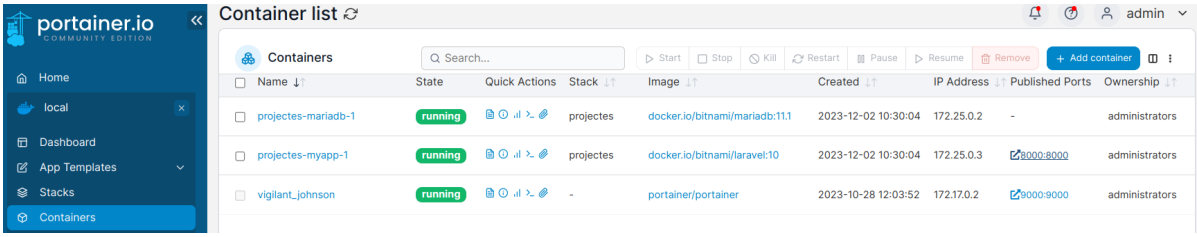
2. Accedemos dentro de la carpeta de este nuevo proyecto.
3. Vamos a utilizar la imagen de Bitnami ya preparada, así que lo que hacer ahora es descargar el archivo docker-compose.yml del repositorio de Github oficial.

   ```
   curl -LO
   https://raw.githubusercontent.com/bitnami/containers/main/bitnami/lara
   vel/docker-compose.yml
   ```

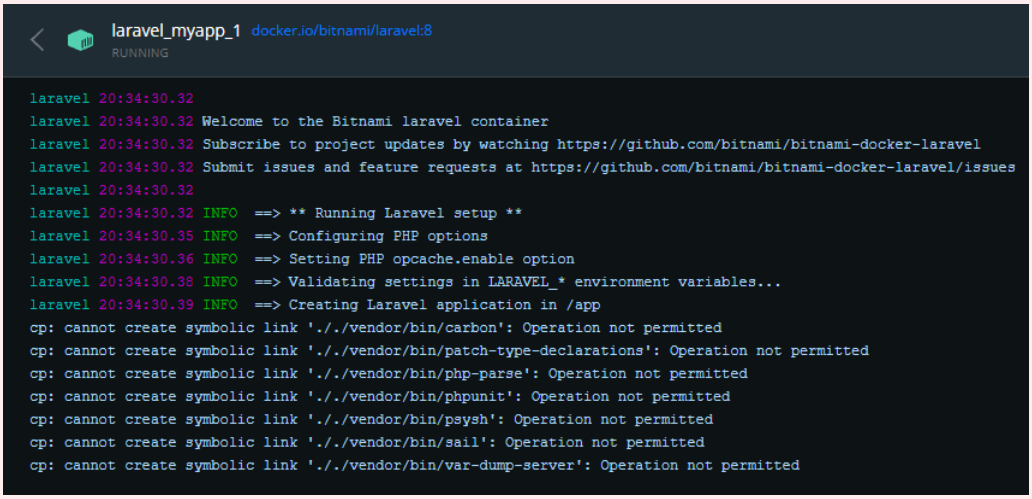
4. Una vez descargado el archivo en nuestra carpeta que acabamos de crear con el nombre del proyecto, lanzamos el siguiente comando por consola para instalar todas las dependencias y crear las imágenes de Docker correspondientes.

   ```
   sudo docker-compose up -d
   ```

5. Si utilizamos el contenedor `Portainer` para la gestión de nuestros contenedores, podremos observar que estarán en marcha nuestros dos contenedores (pertenecientes al servidor web y servidor de bases de datos):



Si por alguna extraña razón estás en Windows y no te funciona una de las 2 imágenes, puede ser debido a la instalación de composer dentro de la imagen de Laravel.



Para solucionarlo, nos vamos a la carpeta del proyecto que se te habrá creado por defecto al hacer docker-compose; en este caso, y si no has modificado el archivo .yml, la carpeta del proyecto sera `my-proyect` y dentro de ella eliminamos la carpeta vendor.

Acto seguido instalar Composer de manera global en nuestro sistema Windows (bájate el instalador desde este enlace).

Una vez lo instales ya serás capaz de lanzar el comando composer desde cualquier consola de Windows.

2.1. VSCode extensiones

Recomendable instalar los siguientes plugins para Visual Studio Code.

Referentes a PHP:

- *PHP Intelephense*
- PHP IntelliSense
- *PHP Namespace Resolver*

Referentes a Laravel:

- Laravel Blade Snippets
- *Laravel Snippets*
- Laravel goto view
- *Laravel Extra Intellisense*

Referentes a CSS:

- *Tailwind CSS IntelliSense*

Aporte

Un aporte, o instalación, a tener en cuenta, podría ser la de instalar `Tailwind CSS`. Este software nos va a proporcionar, de manera sencilla y cómoda, una opción de utilizar CSS.

Para ello, seguir las instrucciones del anexo I - instalación de Tailwind CSS.

3. carpeta en Laravel

Al crear un nuevo proyecto con este framework, Laravel crea una serie de carpetas por defecto. Esta estructura de carpetas es la recomendada para utilizar Laravel.

3.1. public

Esta es la carpeta más importante ya que es donde se ponen todos los archivos que el cliente va a mostrar al usuario cuando introduzcamos la URL de nuestro sitio web. Normalmente se carga el archivo index.php por defecto.

3.2. routes

Otra de las carpetas que más vamos a usar a lo largo de este curso de Laravel. En ella se albergan todas las rutas (redirecciones web) de nuestro proyecto, pero más concretamente en el archivo web.php

```
Dada una ruta → se cargará una vista
```

3.3. resources

Esta es nuestra carpeta de recursos donde guardaremos los siguientes archivos, que también, están separados por sus carpetas... como cada nombre indica:

- `css` Archivos CSS.
- `js` Archivos JS (JavaScript).
- `lang` Archivos relacionados con el idioma del sitio (variables & strings).
- `views` Archivos de nuestras vistas, lo que las rutas cargan.

4. rutas

Las rutas en Laravel (y en casi cualquier Framework) sirven para redireccionar al cliente (o navegador) a las vistas que nosotros queramos.

Estas rutas se configuran en el archivo `public/routes/web.php` donde se define la ruta que el usuario pone en la URL después del dominio y se retorna la vista que se quiere cargar al introducir dicha dirección en el navegador.

```php
<?php
    // Ruta por defecto para cargar la vista welcome cuando el usuario introduce simplemente el dominio
    Route::get('/', function () {
        return view('welcome');
    });
```

En el ejemplo de arriba vamos a cargar la vista llamada welcome que hace referencia a la vista `resources/views/welcome.blade.php`.

4.1. alias

Es interesante darle un alias o un nombre a nuestras rutas para poder utilizar dichos alias en nuestras plantillas de Laravel que veremos más adelante.

Para ello, basta con utilizar la palabra `name` al final de la estructura de la ruta y darle un nombre que queramos; normalmente descriptivo y asociado a la vista que tiene que cargar el enroutador de Laravel.

```php
<?php
    Route::get('/users', function () {
        return view('users');
    }) -> name('usuarios');
```

Después veremos que es muy útil ya que a la hora de refactorizar o hacer un cambio, si tenemos enlaces o menús de navegación que apuntan a esta ruta, sólo tendríamos que cambiar el parámetro dentro del get() y no tener que ir archivo por archivo.

Laravel nos proporciona una manera más cómoda a la hora de cargar una vista si no queremos parámetros ni condiciones. Tan sólo definiremos la siguiente línea que hace referencia la ruta datos en la URL y va a cargar el archivo `usuarios.php` de nuestra carpeta views como le hemos indicado en el segundo parámetro.

```php
<?php
    /* http://localhost/datos/ */
    Route::view('datos', 'usuarios');
```

Pero no sólo podemos retornar una vista, sino, desde un simple string, a módulos propios de Laravel.

4.2. parámetros

Ya hemos visto que con PHP podemos pasar parámetros a través de la URL, como si fueran variables, que las recuperábamos a través del método GET o POST.

Con Laravel también podemos introducir parámetros pero de una forma más vistosa y ordenada, de tal manera que sea visualmente más cómodo de recordar y de indexar por los motores de búsqueda como Google.

```
http://localhost/cliente/324
```

Para configurar este tipo de rutas en nuestro archivo de rutas `public/routes/web.php` haremos lo siguiente.

```
<?php
    Route::get('cliente/{id}', function($id) {
        return('Cliente con el id: ' . $id);
    });
```

¿Qué pasa si no introducimos un id y sólo navegamos hasta cliente/ ? ... Nos va a devolver un `404 | NOT FOUND`.

Para resolver esto, podemos definir una ruta por defecto en caso de que el id (o parámetro) no sea pasado. Para ello usaremos el símbolo `?` en nuestro nombre de ruta e inicializaremos la variable con el valor que queramos.

```
<?php
    Route::get('cliente/{id?}', function($id = 1) {
        return ('Cliente con el id: ' . $id);
    });
```

Ahora tenemos otro problema, porque estamos filtrando por id del cliente que, normalmente es un número; pero si metemos un parámetro que no sea un número, vamos a obtener un resultado no deseado.

Para resolver este caso haremos uso de la cláusula where junto con una expresión regular númerica.

```
<?php
    Route::get('cliente/{id?}', function($id = 1) {
        return ('Cliente con el id: ' . $id);
    }) -> where('id', '[0-9]+');
```

Además, podemos pasarle variables a nuestra URL para luego utilzarlas en nuestros archivos de plantillas o en archivos .php haciendo uso de un array asociativo. Veamos un ejemplo con la forma reducida para ahorrarnos código.

```
<?php
    Route::view('datos', 'usuarios', ['id' => 5446]);
```

... y el archivo `resources/views/usuarios.php` debe tener algo parecido a ésto:

```
<!-- Estructura típica de un archivo HTML5 -->
<!-- ... -->
<p>Usuario con id: <?= $id ?></p>
<!-- ... -->
```

> Con las plantillas de Laravel **blade.php** veremos cómo simplificar aún más nuestro código.

Para más información acerca de las rutas, parámetros y expresiones regulares en las rutas puedes echar un vistazo a la documentación oficial de rutas que contiene numerosos ejemplos.

5. plantillas o templates

A través de las plantillas de Laravel vamos a escribir menos código PHP y vamos a tener nuestros archivos mejor organizados.

Blade es el sistema de plantillas que trae Laravel, por eso los archivos de plantillas que guardamos en el directorio de *views* llevan la extensión `blade.php`.

De esta manera sabemos inmediatamente que se trata de una plantilla de Laravel y que forma parte de una vista que se mostrará en el navegador.

5.1. directivas

Laravel tiene un gran número de directivas que podemos utilizar para ahorrarnos mucho código repetitivo entre otras funciones.

Digamos que las directivas son pequeñas funciones ya escritas que aceptan parámetros y que cada una de ellas hace una función diferente dentro de Laravel.

- `@yield` Define el contenido dinámico que se va a cargar. Se usa conjuntamente con `@section`.
- `@section` y `@endsection` bloque de código dinámico.
- `@extends` importa el contenido de una plantilla ya creada.

En en anexo III - uso de directivas tienes un ejemplo.

5.2. separando código

Veamos un ejemplo de cómo hacer uso del poder de Laravel para crear plantillas y no repetir código.

Supongamos que tenemos ciertas estructuras HTML repetidas como puede ser una cabecera header, un menú de navegación nav y un par de secciones que hacen uso de este mismo código.

Supongamos que tenemos 2 apartados en la web:

- Blog
- Fotos

Primero de todo tendremos que generar un archivo que haga de plantilla de nuestro sitio web.

Para ello creamos el archivo plantilla.blade.php dentro de nuestro directorio de plantillas resources/views.

Dicho archivo va a contener el típico código de una página simple de HTML y en el body añadiremos nuestros contenido estático y dinámico.

```
1  <body>
2     <!-- CONTENIDO ESTÁTICO PARA TODAS LAS SECCIONES -->
3     <h1>Bienvenid@s a Laravel</h1>
4     <hr>
5
6     <!-- MENÚ -->
7     <nav>
```

```
        <a href={{ route('noticias') }}>Blog</a> |
        <a href={{ route('galeria') }}>Fotos</a>
    </nav>

    <!-- CONTENIDO DINÁMICO EN FUNCIÓN DE LA SECCIÓN QUE SE VISITA -->
    <header>
        @yield('apartado')
    </header>
</body>
```

Cada sección que haga uso de esta plantilla contendrá el texto estático Bienvenid@s a Laravel seguido de un menú de navegación con enlaces a cada una de las secciones y el contenido dinámico de cada sección.

Ahora crearemos los archivos dinámicos de cada una de las secciones, en nuestro caso blog.blade.php y fotos.blade.php

```
<?php
  // blog.blade.php

  @extends('plantilla')

  @section('apartado')
    <h2>Estás en BLOG</h2>
  @endsection
```

Importamos el contenido de plantilla bajo la directiva @extends para que cargue los elementos estáticos que hemos declarado y con la directiva @section y @endsection definimos el bloque de código dinámico, en función de la sección que estemos visitando.

Ahora casi lo mismo para la sección de fotos

```
<?php
  // fotos.blade.html

  @extends('plantilla')

  @section('apartado')
    <h2>Estás en FOTOS</h2>
  @endsection
```

El último paso que nos queda es configurar el archivo de rutas routes/web.php

```
<?php
  // web.php

  Route::view('blog', 'blog') -> name('noticias');
  Route::view('fotos', 'fotos') -> name('galeria');
```

De esta manera podremos hacer uso del menú de navegación que hemos puesto en nuestra plantilla y gracias a los alias noticias y galeria, la URL será más amigable.

5.3. estructuras de control

Como en todo buen lenguaje de programación, en Laravel también tenemos estructuras de control.

En Blade (plantillas de Laravel) siempre que iniciemos un bloque de estructura de control DEBEMOS cerrarla.

- @foreach ~ @endforeach lo usamos para recorrer arrays.
- @if ~ @endif para comprobar condiciones lógicas.
- @switch ~ @endswitch en función del valor de una variable ejecutar un código.
- @case define la casuística del switch.
- @break rompe la ejecución del código en curso.
- @default si ninguna casuística se cumple.

```
<?php

  $equipo = ['María', 'Alfredo', 'William', 'Verónica'];

  @foreach ($equipo as $nombre)
     <p> {{ $nombre }} </p>
  @endforeach
```

Acordaros que podemos pasar variables a través de las rutas como si fueran parámetros. Pero en este caso, vamos a ver otra directiva más; el uso de @compact.

```
<?php
  // Uso de @compact
  $equipo = ['María', 'Alfredo', 'William', 'Verónica'];

  // Route::view('nosotros', ['equipo' => 'equipo']);
  Route::view('nosotros', @compact('equipo'));
```

6. controladores

Los controladores son el lugar perfecto para definir la lógica de negocio de nuestra aplicación o sitio web.

Hace de intermediario entre la vista (lo que vemos con nuestro navegador o cliente) y el servidor donde la app está alojada.

Por defecto, los controladores se guardan en una carpeta específica situada en app/Http/Controllers y tienen extensión .php.

Para crear un controlador nuevo debemos hacer uso de nuestro querido autómata artisan donde le diremos que cree un controlador con el nombre que nosotros queramos.

Abrimos la consola y nos situamos en la raíz de nuestro proyecto

```
php artisan make:controller PagesController
```

Si todo ha salido bien, recibiremos un mensaje por consola con que todo ha ido bien y podremos comprobar que, efectivamente se ha creado el archivo PagesController.php con una estructura básica de controlador, dentro de la carpeta Controllers que hemos descrito anteriormente.

Ahora podemos modificar nuestro archivo de rutas web.pbp para dejarlo limpio de lógica y trasladar ésta a nuestro nuevo controlador.

La idea de ésto es dejar el archivo web.php tan limpio como podamos para que, de un vistazo, se entienda todo perfectamente.

RECUERDA que sólo movemos la lógica, mientras que las cláusulas como where y name las seguimos dejando en el archivo de rutas web.php

Veamos cómo quedaría un refactor del archivo de rutas utilizando un Controller como el que acabamos de crear.

Ahora nos quedaría de la siguiente manera:

```
<?php

  // web.php (v2.0) ▒▒▒▒ Refactorizado

  use App\Http\Controllers\PagesController;
  use Illuminate\Support\Facades\Route;

  Route::get('/', [ PagesController::class, 'inicio' ]);
  Route::get('datos', [ PagesController::class, 'datos' ]);
  Route::get('cliente/{id?}', [ PagesController::class, 'cliente' ]) ->
where('id', '[0-9]+');
  Route::get('nosotros/{nosotros?}', [ PagesController::class, 'nosotros']) -
> name('nosotros');
```

y en nuestro archivo controlador lo dejaríamos de la siguiente manera:

```
<?php
```

```php
// PagesController.php

namespace App\Http\Controllers;

class PagesController extends Controller
{
   public function inicio() { return view('welcome'); }

   public function datos() {
      return view('usuarios', ['id' => 56]);
   }

   public function cliente($id = 1) {
      return ('Cliente con el id: ' . $id);
   }

   public function nosotros($nombre = null) {
      $equipo = [
          'Paco',
          'Enrique',
          'Maria',
          'Veronica'
      ];

      return view('nosotros', @compact('equipo', 'nombre'));
   }
}
```

7. anexo I - instalación de Tailwind CSS

Si hemos decidido instalar `Tailwind CSS` para que nos eche una mano con nuestro css, deberemos de seguir estos pasos:

1. Hay que comprobar la versión de npm y node:

```
npm -v
node -v
```

2. Si la versión de nodejs es inferior a la versión 14 (a la hora de crear este documento) **antes de seguir habremos de reinstalarlo**. Para ello habrá que ir al anexo II - reinstalación de node en este mismo documento.

> **no continuar si la versión de node es inferior a 14**.

3. Si nuestra versión de nodejs es correcta (o hemos procedido a reinstalar node en el punto 2), instalaremos en desarrollo estas tres dependencias:

```
npm install -D tailwindcss postcss autoprefixer
```

4. Generamos ahora el fichero `tailwindcss.config.js` que aparecerá en la raíz del proyecto:

```
npx tailwindcss init
```

5. Editar el fichero del proyecto Laravel `tailwindcss.config.js` que se ha generado en el directorio raíz del proyecto y donde indicaremos dónde vamos a utilizarlo:

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./resources/**/*.blade.php",
    "./resources/**/*.js",
    "./resources/**/*.vue"
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

6. Ahora, en el fichero /resources/css/`app.css` agregar las siguientes líneas:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

7. Desde el terminal (y siempre dentro de nuestro proyecto), vamos a ejecutar:

```
npm run dev
```

```
abc@abc-pc:~/Escritorio/IES/DWES/projectes/my-project$ npm run dev

> dev
> vite

  VITE v4.5.0  ready in 470 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h to show help

  LARAVEL v10.34.2  plugin v0.8.1

  ➜  APP_URL: http://localhost
```

8. En el fichero /resources/views/layouts/`app.blade.php` hay que indicarle que va a utilizar el fichero /resources/css/`app.css`, para ello hay que añadirlo en:

```
@vite('resources/css/app.css')
```

resources > views > layouts > app.blade.php

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>Mi proyecto - @yield('titulo')</title>

        @vite(resources/css/app.css')

    </head>
    <body>
        <h1>@yield('titulo')</h1>
        <hr>
        @yield('contenido')
    </body>
</html>
```

A partir de ahora, y con este ejemplo, podemos observar que se nos muestra el css:

```
    <body>
        <nav>
            <a href="/">Principal</a>
            <a href="/nosotros">Nosotros</a>
            <a href="/tienda">Tienda virtual</a>
        </nav>


        <h1 class="">@yield('titulo')</h1>
        <hr>
        @yield('con
    </body>
</html>
```

```
{} first-letter:  &::first-letter
{} first-line:
{} marker:
{} selection:
{} file:
{} placeholder:
{} backdrop:
{} before:
{} after:
{} first:
{} last:
{} only:
```

8. anexo II - reinstalación de node

Para reinstalar nodejs:

1. Desinstalar node por completo:

```
sudo apt-get purge --auto-remove nodejs
```

2. Eliminar todo resto de node y npm:

   a. Antes que nadas, debe ejecutar el siguiente comando desde el terminal:

```
sudo rm -rf /usr/local/bin/npm /usr/local/share/man/man1/node*
/usr/local/lib/dtrace/node.d ~/.npm ~/.node-gyp /opt/local/bin/node
opt/local/include/node /opt/local/lib/node_modules
```

   b. Eliminar los directorios node o node_modules de /usr/local/lib con la ayuda del siguiente comando:

```
sudo rm -rf /usr/local/lib/node*
```

   c. Eliminar los directorios node o node_modules de /usr/local/include con la ayuda del siguiente comando:

```
sudo rm -rf /usr/local/include/node*
```

   d. Eliminar cualquier archivo de nodo o directorio de /usr/local/bin con la ayuda del siguiente comando:

```
sudo rm -rf /usr/local/bin/node
```

3. Instalar otra vez nvm:

   a. Instalar NvM (Node Version Manager), desde el directorio de usuario `~` :

```
curl -o- https://raw.githubusercontent.com/nvm-
sh/nvm/v0.39.0/install.sh | bash
```

   b. Actualiza el archivo .bashrc:

```
source .bashrc
```

   c. Confirma que el directorio local está configurado:

```
echo $NVM_DIR
/home/username/.nvm
```

4. Instalar node:

a. Revisar qué versiones de Node.js están disponibles:

```
nvm ls-remote
```

b. Instalar la versión que desees (elige la v20.10.0):

```
nvm install v20.10.0
```

5. Comprobar que la nueva versión de node es superior a 14:

```
node -v
```

9. anexo III - uso de directivas

10. referencias

- Tutorial de Composer
- Web Scraping with PHP – How to Crawl Web Pages Using Open Source Tools
- PHP Monolog
- Unit Testing con PHPUnit — Parte 1, de Emiliano Zublena.