Desarrollo web en entorno servidor

Unidad 5: Generación dinámica de páginas web

ÍNDICE

INTRODUCCIÓN

Nos encontramos a mitad del temario y ya tenemos conocimientos suficientes para enfrentarnos a proyectos de cierta dificultad. A pesar de eso, aún creamos **código** de forma muy artesanal, y los **estándares de producción actuales** nos imprimen rapidez y nos exigen **escalabilidad**.

Por ello en esta unidad didáctica veremos un **patrón de diseño de software** que nos permitirá **dividir nuestro código** en estructuras más pequeñas y eficientes.

Para enviar datos entre el cliente y el servidor ya estudiamos los métodos, o verbos de acción, **GET** y **POST**. Pero realmente se utilizan muchos más: **PUT**, **PATCH**, **DELETE**, etc.

Cada verbo de acción modifica los datos de una entidad (**sustantivo**). La gestión de qué verbo modifica a según qué entidad se hace mediante rutas (**URL**).

Gestionar rutas de forma manual puede ser algo tedioso. Por eso en esta unidad veremos también herramientas, como los **frameworks**, que nos ayudarán en esta tarea.

OBJETIVOS / CAPACIDADES

En esta unidad de aprendizaje, las capacidades que más se van a trabajar son:

- ✔ **Instalar módulos analizando su estructura y funcionalidad para gestionar servidores de aplicaciones.**
- ✔ **Programar y realizar actividades para gestionar el mantenimiento de los recursos informáticos.**
- ✔ **Utilizar herramientas y lenguajes específicos, cumpliendo las especificaciones, para desarrollar e integrar componentes software en el entorno del servidor web.**
- ✔ **Utilizar lenguajes, objetos y herramientas, interpretando las especificaciones para desarrollar aplicaciones web con acceso a bases de datos.**

PROYECTO DE LA UNIDAD

Antes de empezar a trabajar el contenido, te presentamos la **actividad** que está relacionada con esta unidad de aprendizaje. Se trata de un **caso práctico** basado en una **situación real** con la que te puedes encontrar en tu puesto de trabajo. Con esta actividad se evaluará la puesta en práctica de los **criterios de evaluación** vinculados al resultado de aprendizaje que se trabaja en esta unidad. Para realizarla deberás hacer lo siguiente: lee el enunciado que te presentamos a continuación, dirígete al área general del módulo profesional, concretamente a la actividad de evaluación que se encuentra dentro de esta unidad, allí encontrarás todos los detalles sobre fecha y forma de entrega, objetivos... A lo largo de la unidad irás adquiriendo los conocimientos necesarios para ir elaborando este proyecto.

Enunciado:

Rendimiento de un empleado

Seguiremos evolucionando la aplicación para controlar el computo de horas de los trabajadores de una empresa. En la actividad de evaluación de la UD4 teníamos el fichero **empleados.json** con los siguientes datos en él:

Código: contenido del fichero empleados.json

```
{
  "Empleados":[{
    "Nombre":"Juan Palomarez Hernández",
    "Usuario": "jpalomares@empresa.es",
    "Contraseña":"jy3Rtbm",
    "Historial": [{
     "Lunes": 8,
        "Martes": 9,
        "Miércoles": 7,
        "Jueves": 7,
```

```
        "Viernes": 5
    }]
    },
    {
    "Nombre":"Ana Fernández Marín",
    "Usuario": "afernandez@empresa.es",
    "Contraseña":"sw9Ewph",
    "Historial": [{
      "Lunes": 6,
        "Martes": 6,
        "Miércoles": 8,
        "Jueves": 7,
        "Viernes": 8
    }]
  }]
}
```

Seguiremos utilizando este mismo fichero para el objetivo de esta actividad.

Lo que necesitamos ahora es que se cree una aplicación que cuando se acceda a la ruta raíz (/) muestre la página de *login* que creaste en la UD4.

Cuando un usuario se *loguee* correctamente deberán mostrarse sus datos.

Crea también la ruta empleados (/empleados) para que cualquier usuario que visite la ruta, independientemente si está *logueado* o no, pueda ver solamente una vista con el listado de empleados de la empresa (solamente sus nombres).

1. MECANISMOS DE SEPARACIÓN DE LA LÓGICA DE NEGOCIO. ARQUITECTURA MODELO VISTA - CONTROLADOR

*Entendemos la **lógica de negocio** como la parte de nuestro código que se encarga de codificar las reglas de negocio del mundo real que marcan como la información debe ser creada, almacenada y cambiada.*

Estas **reglas de negocio** son las necesidades que debemos cubrir con nuestros algoritmos. Un ejemplo de ellas son las actividades de evaluación que llevas desarrollando. En cada una de ellas se pide la **solución a un problema con unos requisitos**: calcular el salario, saber si un empleado ha trabajado de más o no, etc.

Hasta ahora hemos ido creando nuestro código de forma **secuencial**. Por ejemplo:

1. Abrimos el archivo.
2. Capturamos los datos.
3. Procesamos los datos.
4. Los mostramos por pantalla.
5. Escribimos los cambios en el archivo.

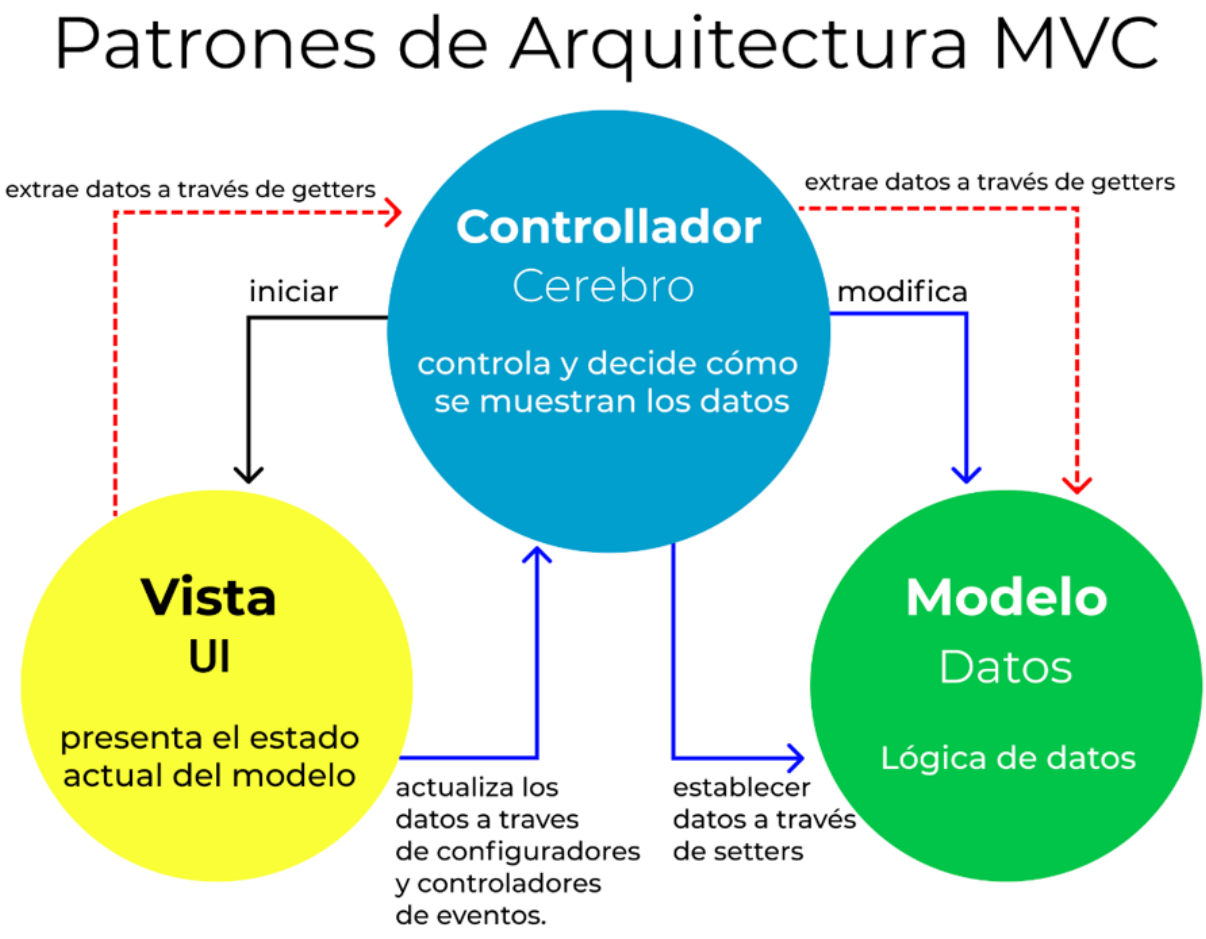
Todos estos **pasos** los hemos escrito uno debajo del otro:

- **I**. Como ya estamos desarrollando tareas de cierta complejidad nos hemos visto obligados, quizás, a utilizar **bloques de código para optimizarlo**. Y quizás también, para un mejor desarrollo, hemos **separado nuestros scripts en varios ficheros**.

  Pero fijémonos en una cosa: nuestro código mezcla la lógica de negocio, por ejemplo, saber el sueldo de un empleado si este ha trabajado más horas, con el almacenamiento de datos y con la presentación por pantalla de los mismos datos.

➔ **II**. Imagina que un día **cambiamos la forma como guardamos los datos**, por ejemplo, cambiamos el soporte y ya no los tenemos en un archivo sino en una base de datos. Con el modelo actual, tendremos que rebuscar entre el código qué parte pertenece al cálculo del sueldo, cosa que no cambia, y que parte pertenece al almacenamiento de datos.

➔ **III**. Separar la lógica de negocio significa compartimentar en módulos de responsabilidad cada tarea. Para ello utilizaremos más adelante un patrón muy famoso de arquitectura de software llamado **MVC: Modelo – Vista – Controlador**.



Elementos de la arquitectura MVC. Fuente: https://www.freecodecamp.org/. Esta imagen se reproduce acogiéndose al derecho de cita o reseña (art. 32 LPI), y está excluida de la licencia por defecto de estos materiales.

Vídeo: ¿qué es el patrón Modelo Vista - Controlador o MVC y cómo funciona?

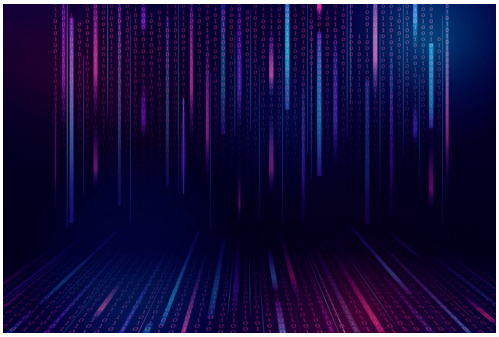
Visualiza el siguiente vídeo sobre el patrón de diseño MVC.
https://www.youtube.com/embed/zhSDjntidws

2. TECNOLOGÍAS ASOCIADAS BASADAS EN LA PROGRAMACIÓN ORIENTADA A OBJETOS Y MULTICAPA

Hasta ahora, hemos ido trabajando bajo un **paradigma imperativo** donde nosotros, como desarrolladores, hemos ido instruyendo a la máquina cómo cambiar su estado mediante la algoritmia de nuestros **scripts**. Seguiremos trabajando de esa misma manera.

A la vez y, para hilar más fino, también hemos ejecutado nuestro código de forma **procedimental**, utilizando procedimientos o funciones para dar instrucciones a la máquina. Podremos seguir también trabajando de esa forma.

Pero cuando intentemos separar la lógica de negocio, como hemos visto en el punto anterior, y asignar capas de responsabilidad, el **paradigma procedimental no es muy eficiente**. Para resolver este y otros desafíos existe también el **paradigma orientado a objetos**.



➔ **Paradigma procedimental**:

La POO es un paradigma de programación que intenta asemejarse a la realidad a la hora de imaginar conceptos.

En algún ejemplo hemos visto algo así:

```
<?php
function saludar($persona) {
      echo "$persona dice '¡Hola!'";
}

$persona1 = "Juan";
$persona2 = "Ana";

saludar("Juan");
saludar("Ana");
?>
```

El código anterior es un claro ejemplo de **paradigma procedimental**. Es fácil entender, pero en el mundo real esto no tiene sentido. No existen funciones donde las personas "nos metemos". Lo más natural es pensar que las personas tienen la capacidad de saludar.

➔ **Paradigma orientado a objetos**:

El mismo ejemplo, con el mismo resultado, pero pensado con **orientación a objetos** sería:

```
<?php
class Persona {
  public $nombre;

  function __construct($nombre) {
    $this->nombre = $nombre;
  }

  function saludar() {
    return "$this->nombre dice '¡Hola!";
  }
}

$persona1 = new Persona("Juan");
$persona2 = new Persona("Ana");
echo $persona1->saludar();
echo $persona2->saludar();
?>
```

Ejecutando el código, vemos que el resultado es el mismo.

Sí, parece más tedioso y largo, pero es mucho más natural y el código es más escalable y fácil de mantener en un futuro.

En los siguientes capítulos ahondaremos más en la programación orientada a objetos.

Actividad de aprendizaje 1: introducción a la programación orientada a objetos

Cambia el siguiente script, escrito de forma procedimental, a orientado a objetos.

```
<?php
function hablar($animal) {
      switch ($animal) {
            case "gato":
                  echo "Miau";
                  break;
            case "perro":
                  echo "Guau";
                  break;
            default:
                  echo "...";
      }
}

hablar("perro");
hablar("gato");
?>
```

Comparte tu idea con tus compañeros en el foro.

Cuestionario

Lee el enunciado e indica la opción correcta:

En la arquitectura MVC, quien se encarga de la lógica de negocio es:

a. Vista.

b. Modelo.

c. Controlador.

Lee el enunciado e indica la opción correcta:

En la arquitectura MVC, la misión del modelo es:

a. Hacer de interfaz de comunicación con la base de datos.

b. Crear objetos para añadirles lógica de negocio.

c. Presentar el estado actual de la aplicación.

Lee el enunciado e indica la opción correcta:

La programación orientada a objetos pertenece:

a. Al paradigma imperativo.

b. Al paradigma declarativo.

c. Al paradigma procedimental.

3. CONTROLES DE SERVIDOR

A continuación, hablaremos de los **controles de servidor**, así como de sus tareas, y de los frameworks:

- La tarea de los **controladores** es la de ser la **capa de lógica de negocio**, que capturará los deseos de los usuarios en forma de datos y acciones, los procesará y actuará acorde a dichas acciones.

  Una acción sería, por ejemplo, seleccionar un producto de un listado desplegable para ver su información. El dato es el propio producto. Con estos parámetros el controlador podrá relacionarse con la base de datos para consultar toda la información del producto y devolverla al cliente en forma de vista.

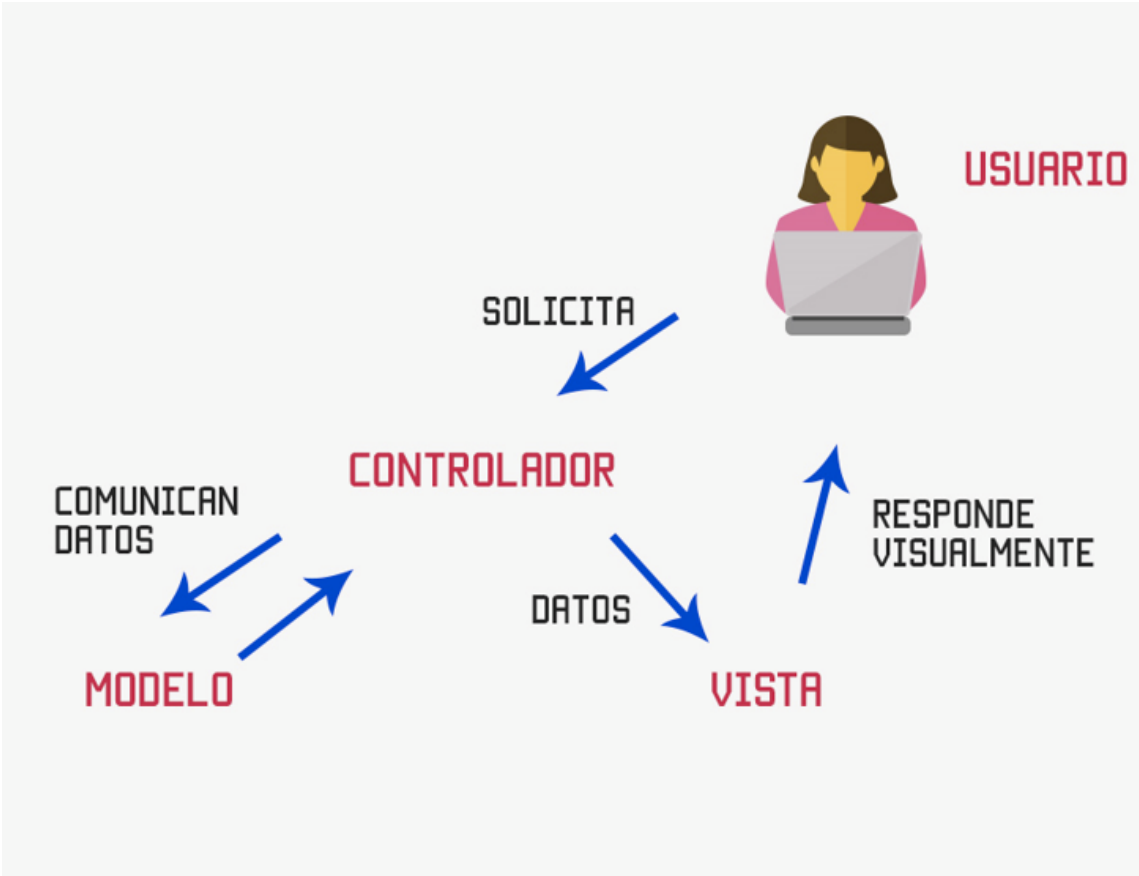
- Los controladores también realizan **tareas** de transformación de datos para hacer que la vista (el resultado visual) y el modelo (el componente encargado de la base de datos) se entiendan. Así, traducirán la información enviada desde la interfaz, por ejemplo, los valores de campos de un formulario recibidos mediante el protocolo HTTP, a objetos que puedan ser comprendidos por el modelo, como pueden las clases.

  Del mismo modo que el controlador manda datos al modelo, también recibe respuesta de este, y esta respuesta deberá procesarla y adaptarla para mandarla a la vista.

- Estos controles normalmente no se construyen de cero, sino que los desarrolladores tienen a su disposición **frameworks** que ayudan a crear y relacionar esas piezas. Un *framework* de PHP muy popular, y es el que usaremos en este curso es **Laravel**.

Obviando muchos pormenores, podemos decir que los controladores son la pieza angular de la arquitectura MVC ya que pivotan el workflow de la acción entre usuario y servidor.

Veamos en la siguiente imagen un diagrama que muestra la interacción del controlador con el modelo y la vista:



Interacción del controlador con el modelo y la vista. Fuente: https://codigofacilito.com/articulos/mvc-model-view-controller-explicado. Esta imagen se reproduce acogiéndose al derecho de cita o reseña (art. 32 LPI), y está excluida de la licencia por defecto de estos materiales.

4. MANTENIMIENTO DEL ESTADO DE LOS CONTROLES

En el apartado anterior hemos explicado que utilizaremos **Laravel** como *framework* de PHP.

*Laravel guarda los controladores dentro de la ruta **app/Http/Controllers** y siguiendo este scaffolding, podemos organizarlos en subcarpetas si lo creemos necesario. Otras clases de Laravel están dentro del sistema de autocarga de clases por lo que estarán disponibles siempre que los necesitemos en la aplicación.*

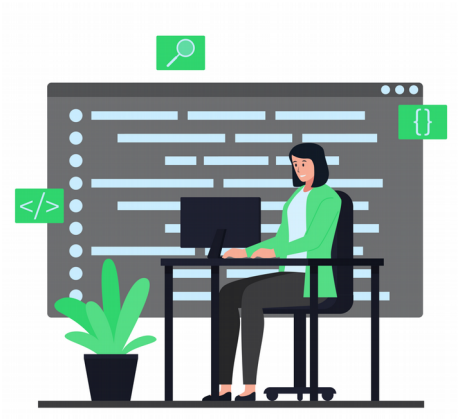
Veamos ahora el ejemplo propuesto en la documentación de Laravel como el **prototipo de controlador básico**:

Código: prototipo de controlador básico

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     *
     * @param  int  $id
     * @return \Illuminate\View\View
     */
    public function show($id)
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
```

```
    }
}
```

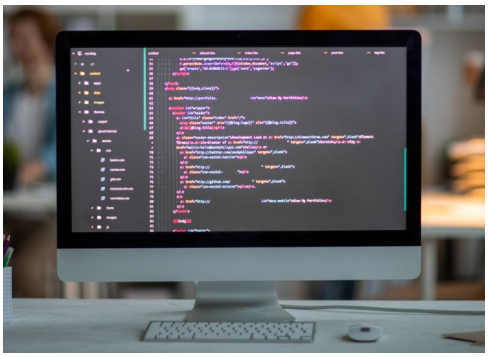
Vamos a ver los detalles de este **controlador**:



- **I**. Las tres primeras líneas de código nos indican en que espacio de nombres trabajamos y dónde se encuentran aquellas clases que utilizaremos (la clase **Controller** y la clase **User**).
- **II**. Este controlador se llama **UserController** y hereda características de una clase proporcionada por Laravel llamada **Controller**.
- **III. UserController** tiene una función llamada **show()** que consulta un usuario a partir del campo $id al modelo llamado **User**.
- **IV**. La función **show()** también devuelve la vista **user.profile** con los datos recogidos en el modelo.

5. MECANISMOS DE GENERACIÓN DINÁMICA DEL INTERFACE WEB. SISTEMAS DE PLANTILLAS

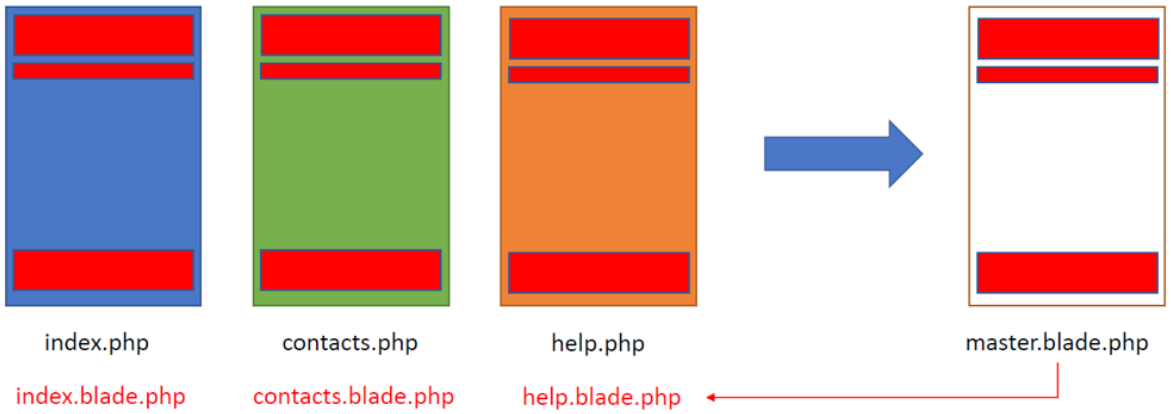
Las vistas de Laravel se guardan en el directorio **resources/views**. Los archivos de las vistas simplemente llevan un nombre y la extensión **.php**, pero hay algunos archivos que entre el nombre y la extensión, lleva la palabra **.blade**. Esto indica que usan el motor de plantillas Blade.



- **Blade** es el motor de plantillas de Laravel y sirve para escribir código de una forma más cómoda. Puedes conocer la sintaxis de Blade dirigiéndote al apartado recursos para ampliar: motor de plantillas Blade.

- **Función**.

  A parte de una sintaxis cómoda, Blade, como motor de plantillas nos permite reaprovechar y escalar código. Imaginemos que el *header*, el *navbar* y el *footer* de una web se repiten en cada página, como vemos en la siguiente imagen:

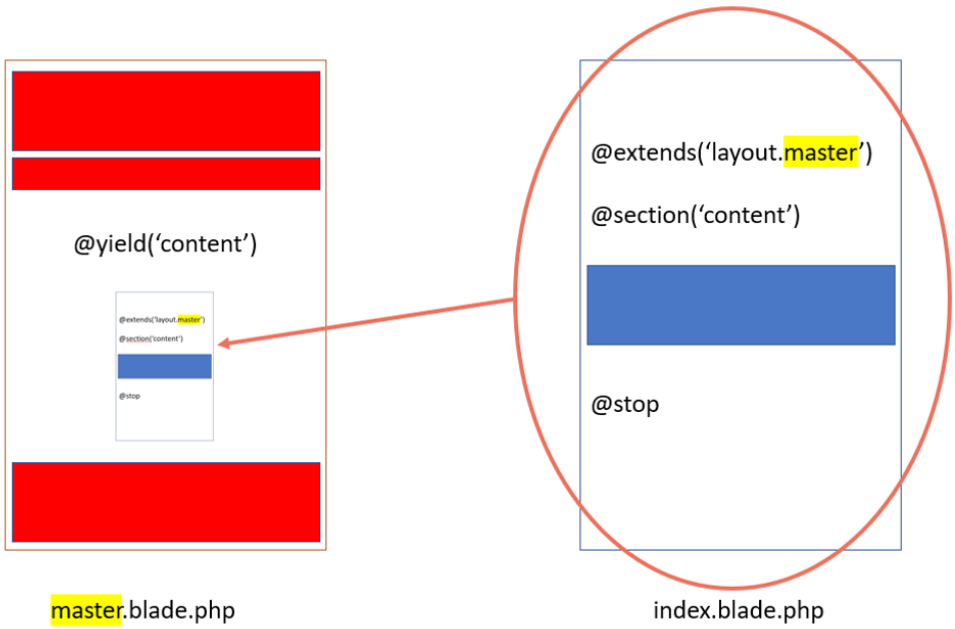


Estructura del sistema de plantillas de Blade

Es una buena idea que aquello común esté agrupado en una estructura, llamada en este ejemple **master.blade.php**. Es este archivo hay un campo

@yield que indica que habrá algún contenido que lo escribirá quien use la plantilla.

Y en cada una de las páginas (**index.php**, **contacts.php**, **help.php**, etc.) habrá sólo el contenido y la invocación de la plantilla que utilizan tal y como se ve en la imagen.



Ejemplo del sistema de plantillas de Blade

Actividad aprendizaje 2: primer proyecto Laravel

ACTIVIDAD COLABORATIVA

Genera un nuevo proyecto de Laravel llamado "instituto". Modifica la vista para que parezca la landing page de un centro educativo.

Comparte tu diseño con tus compañeros.

Cuestionario

Lee el enunciado e indica la opción correcta:

La tarea del controlador es:

a. Enrutar las peticiones de los clientes.

b. Procesar los datos de los usuarios y gestionar la comunicación con el modelo y las vistas.

c. Presentar las vistas.

Lee el enunciado e indica la opción correcta:

El scaffolding de Laravel es:

a. Los archivos del controlador.

b. El sistema de rutas y directorios.

c. Los archivos de configuración.

Lee el enunciado e indica la opción correcta:

Las vistas en Laravel se encuentran en:

a. /resources/views.

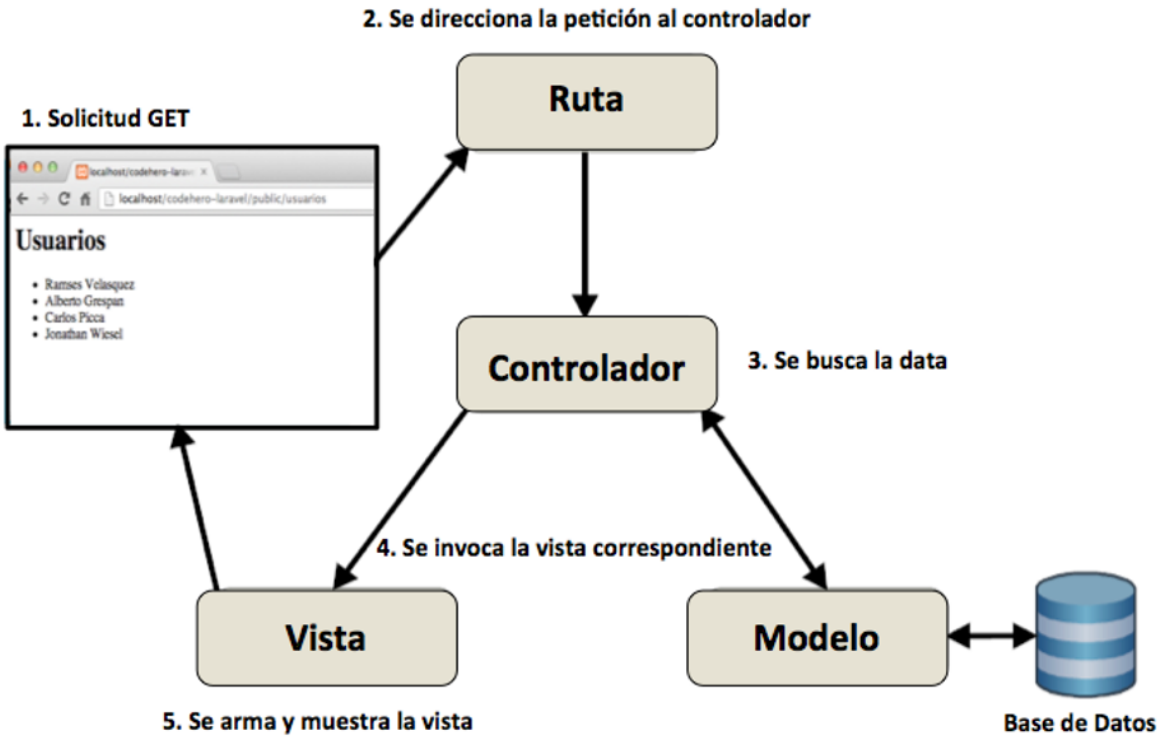
b. /public/views.

c. /views.

6. ELEMENTOS: CARGADORES Y RUTEADORES

Una ruta es el lugar al que dirigimos una petición requerida por un usuario. Nosotros podemos definir las rutas de nuestra aplicación en **routes/web.php**

Para entender el concepto de **ruta** podemos observar la siguiente imagen:



Workflow de una solicitud GET en Laravel. Fuente: https://richos.gitbooks.io/. Esta imagen se reproduce acogiéndose al derecho de cita o reseña (art. 32 LPI), y está excluida de la licencia por defecto de estos materiales.

Vamos a ver algunos **ejemplos de ruta**:

- Ejemplo de **ruta básico** que nos devolverá el texto "¡Hola Mundo!":

```
Route::get('hola', function() { return "¡Hola mundo!"; });
```

- Ejemplo de una **ruta con paso de parámetros**. Esta petición GET hacia el recurso "nombre" nos devolverá "Hola" y el nombre que le pasemos por parámetro:

```
Route::get('nombre/{nombre}', function($nombre) { return "Hola ".
$nombre; });
```

➔ Ejemplo de una **ruta que buscará la función show()** de un controlador llamado UserController a partir de una petición GET hacia el recurso "user".

```
Route::get('user', 'UserController@show');
```

Hasta ahora, hemos visto sólo peticiones **GET**, pero también usamos otros métodos. Mirando la documentación de Laravel y tomando como ejemplo un tipo de controlador llamado **Controlador de Recursos**, tenemos:

Actions Handled By Resource Controller

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

Acciones manejadas por el controlador de recursos. Fuente: Laravel. Esta imagen se reproduce acogiéndose al derecho de cita o reseña (art. 32 LPI), y está excluida de la licencia por defecto de estos materiales.

7. EXTENSIONES: LIBRERÍAS Y CLASES

Más allá del núcleo de un *framework* como Laravel, también existen toda una serie de **servicios integrados**, soportados por Laravel a través de librerías de terceros. Estas librerías están enfocadas al desarrollo como partes importantes de una aplicación. Estos servicios, aunque son de **uso habitual**, no forman parte del ***core* de Laravel**, por lo que unas aplicaciones pueden requerir usarlos y otras no.

A continuación, veremos:

- **Para usar un service provider** tenemos que tener presente:
  - El archivo **config/app.php** presente en los proyectos Laravel, contiene un array llamado providers.
  - Este array es un listado de todos los *service providers* que serán cargados en nuestra aplicación. Cada *service provider* es una clase.
  - No todos los proveedores son cargados cuando nuestra aplicación resuelve una petición. Muchos de ellos son cargados únicamente cuando se requieren.
- Cuando desarrollamos una aplicación muy puntual, generalmente tenemos secciones bastante simples, donde basta con validar y registrar datos. En estos casos no es necesario usar *service providers*. Pero a veces, hay **aplicaciones que van más allá de registrar datos**. Por ejemplo:
  - Cuando debemos subir y procesar imágenes.
  - Cuando debemos enviar notificaciones, de distintos tipos (vía email, sms, *push notifications*, etc.).

- ✔ Cuando debemos generar gráficos e informes.

➔ Aquí es cuando entran en juego los *service providers* y tenemos varias formas de **implementar service providers**:

- ✔ Usando como base un paquete ya escrito por alguien.
- ✔ Escribiendo nuestra propia lógica.
- ✔ Combinando el uso de distintos paquetes y nuestros propios algoritmos según se requiera.

➔ **¿Es necesario implementar service providers?** Para saber si hace falta implementar *service providers* o no, debemos estudiar:

- ✔ Si la necesidad a resolver es pequeña, quizás con un método en un controlador sea suficiente.
- ✔ Si la solución a desarrollar es pequeña, no requiere de una configuración inicial, y debe usarse en distintos lugares, podemos definir un *helper* (una función que estará disponible en todo nuestro proyecto).
- ✔ Sin embargo, si la solución requiere de una configuración inicial (depende de otras clases, y requiere parámetros específicos), lo más recomendable es crear un *service provider* (un proveedor, para que se encargue de la configuración de este servicio, y nosotros simplemente lo usemos).

8. INSTALACIÓN DE FRAMEWORK

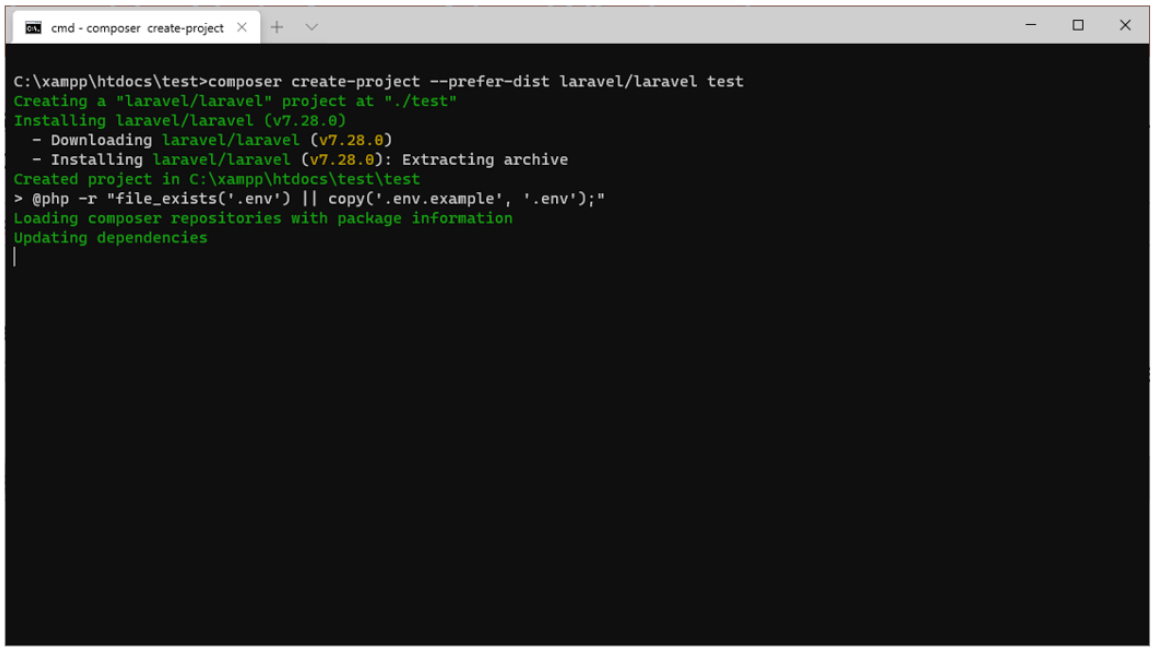
Laravel se dota de un gestor de dependencias PHP llamado **Composer (getcomposer.org)**. Por lo tanto, lo más cómodo es instalar Composer (vía terminal o vía ejecutable).

A continuación, veremos cómo **instalar Composer**:

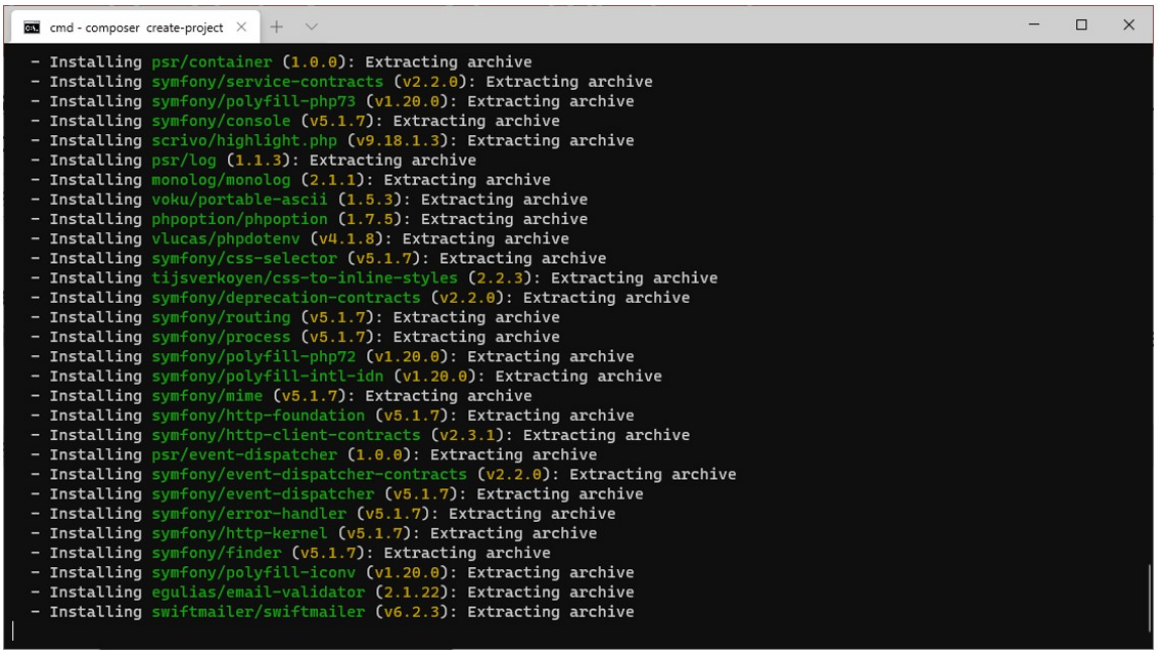
- Composer hará una serie de **preguntas durante la instalación**. La pregunta clave es la ruta en la que tenemos el **binario de PHP**. Si utilizas un stack de servidor web como XAMPP o LAMP la ruta más habitual es:
  - Windows: `C:/xampp/php`
  - Linux: `/usr/bin/php`

Comprueba la ruta donde tienes instalado PHP. Si no te acuerdas repasa la unidad didáctica 1.

- Laravel nos propone **dos formas de instalar** o crear un nuevo proyecto.
  - Por un lado, ejecutar: `composer global require "laravel/installer"`
  - Y después en el directorio donde queramos trabajar escribir: `laravel new test`
- La otra opción de instalar o **crear un nuevo proyecto** (y quizás la más habitual) es utilizando comandos de Composer. Ésta es la que utilizaremos. Nos situamos en la carpeta www o htdocs de nuestro servidor y vía Composer creamos el primer proyecto: `composer create-project --prefer-dist laravel/laravel test`



Proceso de creación de un proyecto Laravel (I)



Proceso de creación de un proyecto Laravel (II)

El uso del argumento –prefer-dist le dice a Composer que intente montar un proyecto de Laravel con la versión más reciente. En nuestro caso nos interesaría la versión 8. Pero es necesario que se cumplan los requisitos que

esta versión pide, como por ejemplo PHP7.3. Si tenemos una versión más antigua, Composer nos ajustará los parámetros y seguramente nuestro proyecto estará montado sobre una versión 7.X de Laravel.

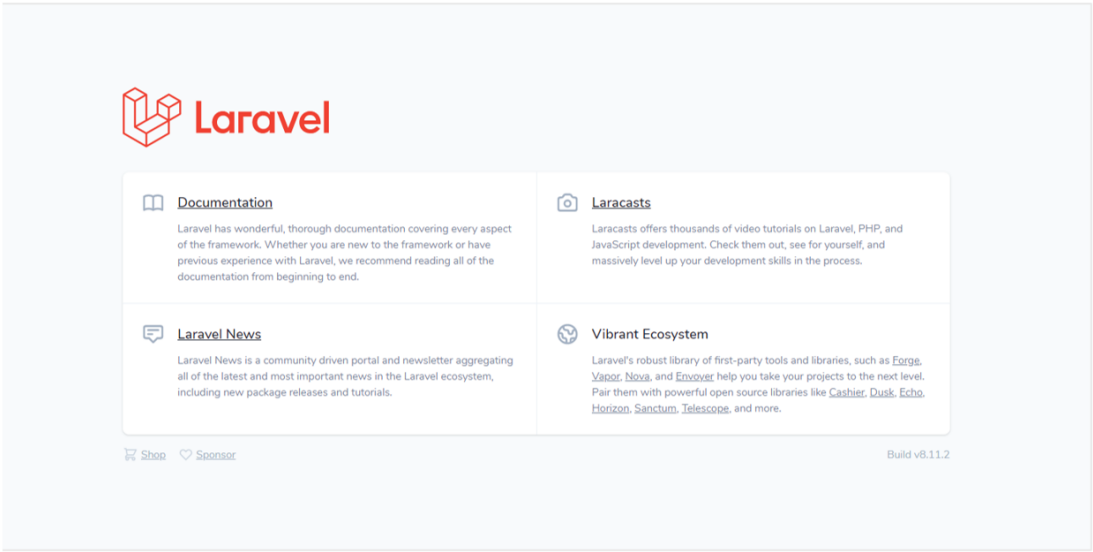
Vídeo: creación de un proyecto de Laravel

Visualiza el siguiente vídeo en el que verás cómo crear un proyecto de Laravel.

9. CONFIGURACIÓN DE UNA APLICACIÓN WEB Y DESARROLLO DE APLICACIONES EN FRAMEWORK

En el apartado anterior hemos creado un **nuevo proyecto de Laravel**. Este proyecto ya se puede visualizar con los archivos que el *framework* nos monta por defecto:

- **I**. Todo lo que será accesible se guarda en la carpeta **public**. En fase de producción deberemos enlazarlo o crear un subdominio para verlo correctamente.
- **II**. De momento podemos ir a nuestro navegador y, si hemos seguido las rutas y nombres del ejemplo, dirigirnos a `http://localhost/test/public` y deberíamos ver algo así:



Página de bienvenida de Laravel

También podemos utilizar un servidor al vuelo, escribiendo el comando `php artisan serve` en la carpeta de nuestro proyecto. Esto nos levantará un servidor web escuchando en el puerto 8000, por tanto, podemos ir a `http://127.0.0.1:8000`.

- **III**. Cuando hemos creado el proyecto en Laravel se han añadido una serie de carpetas y archivos en nuestra ruta. Esto es lo que se llama el *scaffolding*. Todas las configuraciones se encuentran en la carpeta **config**.
- **IV**. Dentro de la carpeta **config** encontramos el archivo **app.php** con una serie de configuraciones sobre la aplicación: si está en producción o *debug*, la zona horaria, el idioma, etc. De la misma forma encontramos el archivo **database.php** con configuraciones por las bases de datos: BD por defecto, migraciones, etc.
- **V**. A partir de aquí ya podemos empezar a construir controladores, modelos y vistas. Podemos hacerlo manualmente, pero Laravel incluye una interfaz de línea de comandos llamada artisan que nos facilitará estas tareas. Por ejemplo:
  - Para crear un nuevo controlador: `php artisan make:controller UserController`.
  - Para crear un nuevo modelo: `php artisan make:model User`.
  - Para listar las rutas creadas: `php artisan route:list`.
  - Si se quieren saber todas las opciones de artisan: `php artisan -h`.

Vídeo: configuración de un proyecto de Laravel

Visualiza el siguiente vídeo para aprender a cómo configurar un proyecto de Laravel.

Actividad aprendizaje 3: primer proyecto Laravel II

A partir de la actividad anterior crea una nueva ruta que deberá responder peticiones a /schedule. Esta ruta deberá dirigirte a la función schedule() de un controlador llamado InstitutoController. La función schedule() deberá devolver una vista con un horario de asignaturas.

Cuestionario

Lee el enunciado e indica la opción correcta:

Las rutas para páginas web en Laravel se encuentran en:

a. /routes/web.php.

b. /routes/api.php.

c. /resources/routes.php.

Lee el enunciado e indica la opción correcta:

¿Qué es composer?

a. Un framework.

b. Un gestor de dependencias.

c. Un ejecutable.

Lee el enunciado e indica la opción correcta:

La interfaz de línea de comandos de Laravel se llama:

a. Artisan.

b. Composer.

c. Php.

RESUMEN

En esta unidad hemos visto cómo organizar nuestro código de una forma **escalable**, delegando responsabilidades en distintos módulos o capas gracias a la arquitectura MVC (Modelo Vista-Controlador).

Esta organización en un principio la hemos hecho de forma manual, pero enseguida nos hemos adentrado en el mundo de los ***frameworks***, ya que estos nos facilitan mucho el desarrollo de software.

Finalmente hemos creado nuestro primer proyecto con un *framework* de PHP, **Laravel** en este caso, y hemos ido viendo los primeros conceptos de **rutas**, **controladores** y **vistas**. En unidades posteriores completaremos nuestro aprendizaje incluyendo **bases de datos** y el **mapeo de objetos** con la ayuda de los modelos.

RECURSOS PARA AMPLIAR

PÁGINAS WEB

- Motor de plantillas Blade: *https://laravel.com/docs/9.x/blade* [Consulta noviembre 2022].
- Service providers: *https://programacionymas.com/blog/service-providers-en-laravel* [Consulta noviembre 2022].

BIBLIOGRAFÍA

PÁGINAS WEB

- Documentación de MDN sobre MVC: *https://developer.mozilla.org/es/docs/Glossary/MVC* [Consulta noviembre 2022].
- Referencia de Laravel: *https://laravel.com/* [Consulta noviembre 2022].
- Referencia oficial de PHP: *https://www.php.net/* [Consulta noviembre 2022].

GLOSARIO

- **Controlador**: dentro de la arquitectura MVC, el controlador contiene una lógica que actualiza el modelo y / o vista en respuesta a las entradas de los usuarios de la aplicación.
- **Framework**: es un conjunto estandarizado de conceptos, prácticas y criterios que ayuda al desarrollo de software de imprimiéndole fiabilidad y escalabilidad.
- **Modelo**: dentro de la arquitectura MVC, el modelo es la pieza que se relaciona con la base de datos y define qué datos debe contener la aplicación.
- **Scaffolding**: literalmente se traduce como andamio y es la estructura de subcarpetas que un *framework* necesita para construir un proyecto.
- **Vista**: dentro de la arquitectura MVC, la vista define cómo se deben mostrar los datos de la aplicación.