

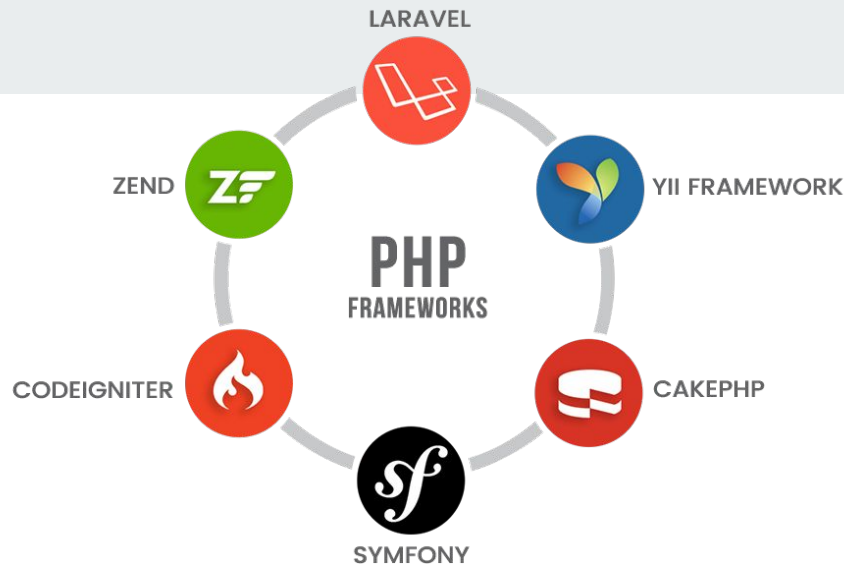


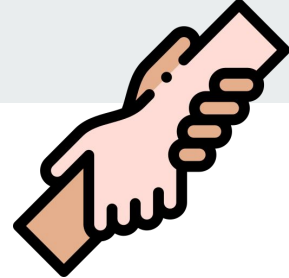
DESARROLLO DE APLICACIONES CON FRAMEWORKS

CONTENIDOS



- Introducción.
- Frameworks y Patrón MVC.
- Symfony:
 - Características, instalación y utilización.
- Laravel
 - Características, instalación y utilización.





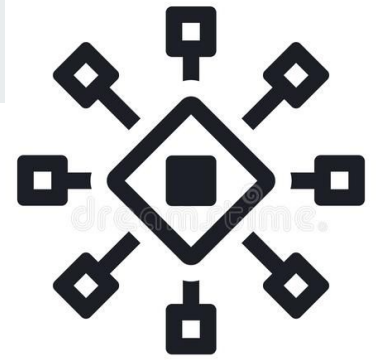
INTRODUCCIÓN

Desde el inicio de cualquier actividad la **evolución** pasa por el uso de herramientas que nos ayuden y faciliten la realización de dicho trabajo.

En el mundo del desarrollo de software se ha establecido desde hace años la utilización de ciertos **paquetes de software** que precisamente intentan facilitar la creación de código de una manera más rápida y de buena calidad (siguiendo buenas prácticas).

En esta evolución surge el concepto **framework** que se emplea en muchos ámbitos del desarrollo de sistemas software, incluidas las aplicaciones Web, y que son un añadido al concepto de **librería** (paquete de software que podemos utilizar en nuestro código) que se viene utilizando.

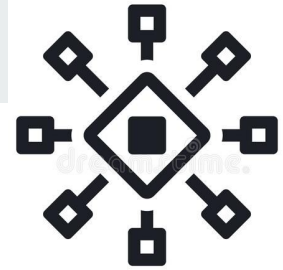
FRAMEWORKS



Los frameworks o entornos/marcos de trabajo son:

- En el mundo en general son **conjuntos estandarizados de conceptos, prácticas y criterios** para enfocar un tipo de problemática particular, para resolver nuevos problemas similares.
- En desarrollo de software es una **estructura** conceptual y tecnológica que ayuda mediante artefactos o módulos concretos de software y sirve de base para la organización y desarrollo de software.

Esta estructura es una arquitectura de software que modela las relaciones de todos los elementos y se añade una metodología de trabajo concreta para utilizar las aplicaciones existentes en su dominio.



FRAMEWORKS

Centrándonos en el mundo software:

- Framework es una **estructura** software.
- Está compuesto de **componentes personalizables** e intercambiables para el desarrollo de una aplicación.
- Se puede considerar como una **aplicación genérica** incompleta y configurable a la que podemos añadir piezas para construir una aplicación concreta.



FRAMEWORKS

Existen multitud de frameworks como:

- Para PHP: Symfony, Laravel, CodeIgniter, Yii...
- Para Java: Spring, Hibernate, JSF, Struts...
- Para Ruby: Ruby on Rails, Padrino, Sinatra...
- Para JavaScript: Angular, React, Vue, Meteor...
- Para Python: Django, Flask, Pyramid...
- Para ASP: ASP.NET, ASP.NET MVC, ReactiveUI...

Cada framework puede implementar un patrón, e incluso el mismo patrón se puede implementar de distinta forma



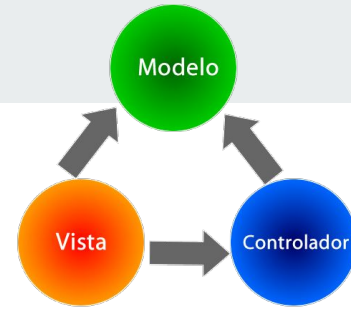
PATRONES EN EL MUNDO SOFTWARE



Debemos **diferenciar** entre patrones de diseño y patrones arquitectónicos.

- Los **patrones de diseño** se utilizan a un nivel más reducido y tratan de resolver un problema concreto: Hasta el minuto 4:30 aprox:
<https://www.youtube.com/watch?v=6BHOeDL8vls>
- Los **patrones arquitectónicos** son similares al patrón de diseño de software pero tienen un alcance más amplio. Algunos de ellos son el patrón de capas, **el patrón cliente-servidor**, el patrón maestro-esclavo, el patrón de filtro de tubería, el patrón de intermediario, el patrón de igual a igual, el patrón de bus de evento, **el patrón modelo - vista - controlador**, el patrón de pizarra o el patrón de intérprete.

PATRÓN MODELO - VISTA - CONTROLADOR



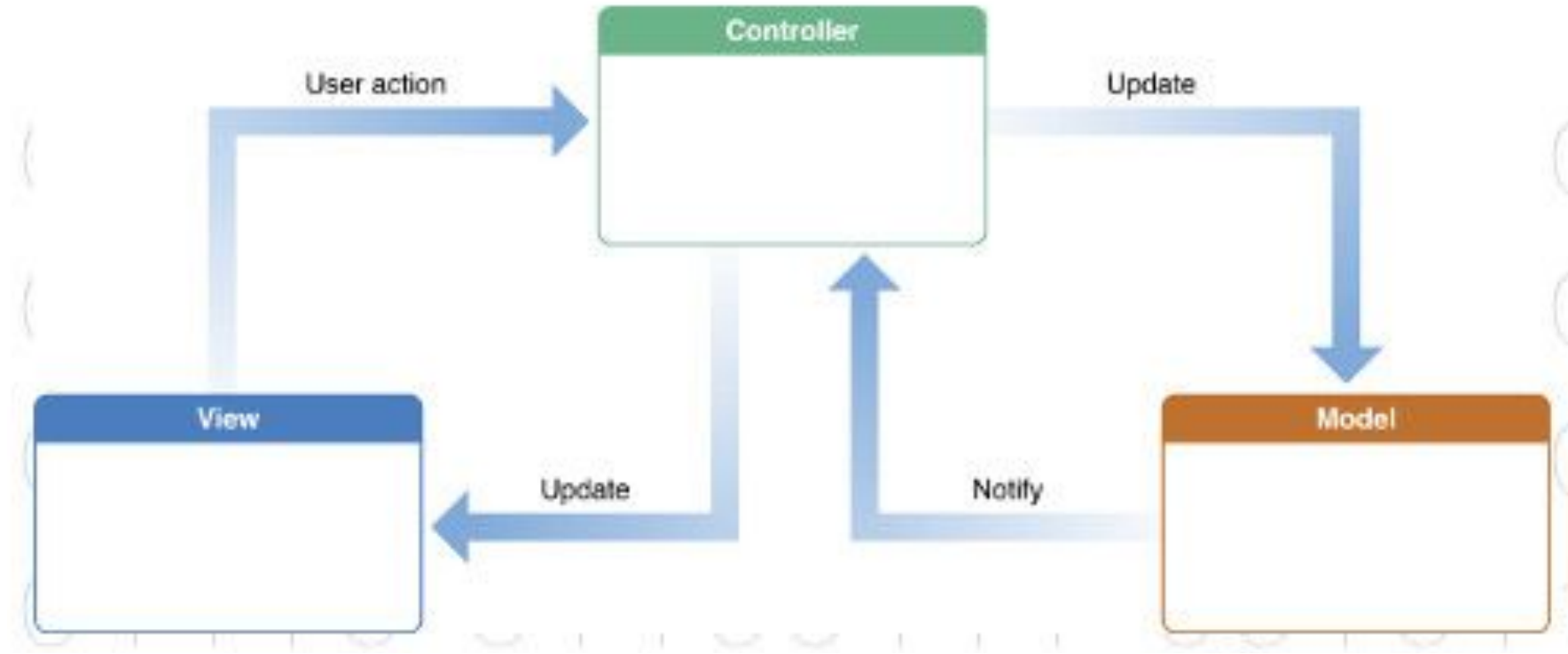
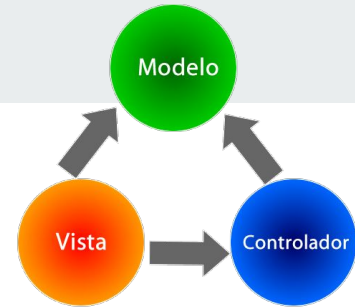
El patrón **MVC** consiste en dividir la aplicación en 3 capas:

- Capa de “**Modelo**”: lógica de negocio encargada de manejar la base de datos. Contiene la funcionalidad y los datos básicos
- Capa de “**Vista**”: interfaz gráfica para mostrar una parte del modelo al usuario. Es posible (y habitual) tener más de una vista.
- Capa de “**Controlador**”: recoge las acciones del usuario e interactúa con el modelo manejando los eventos del usuario.

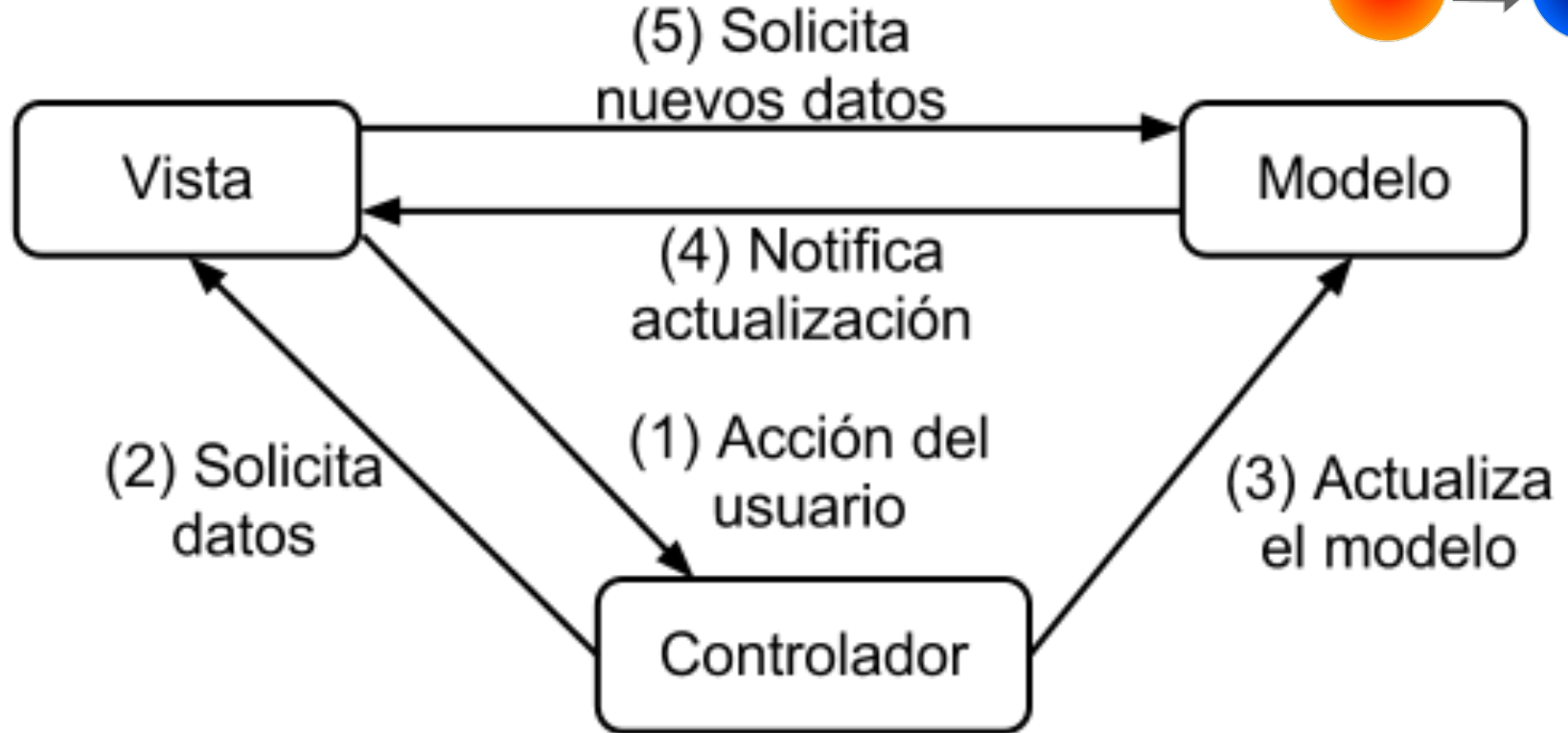
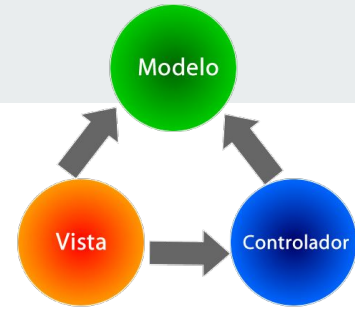
Al hacer esta división se consigue poder **reutilizar** el código y poder **desarrollar** la aplicación **en paralelo** con 3 equipos diferentes.

No es el único patrón de desarrollo aunque puede que sea el más extendido en el desarrollo de aplicaciones, incluidas las aplicaciones web.

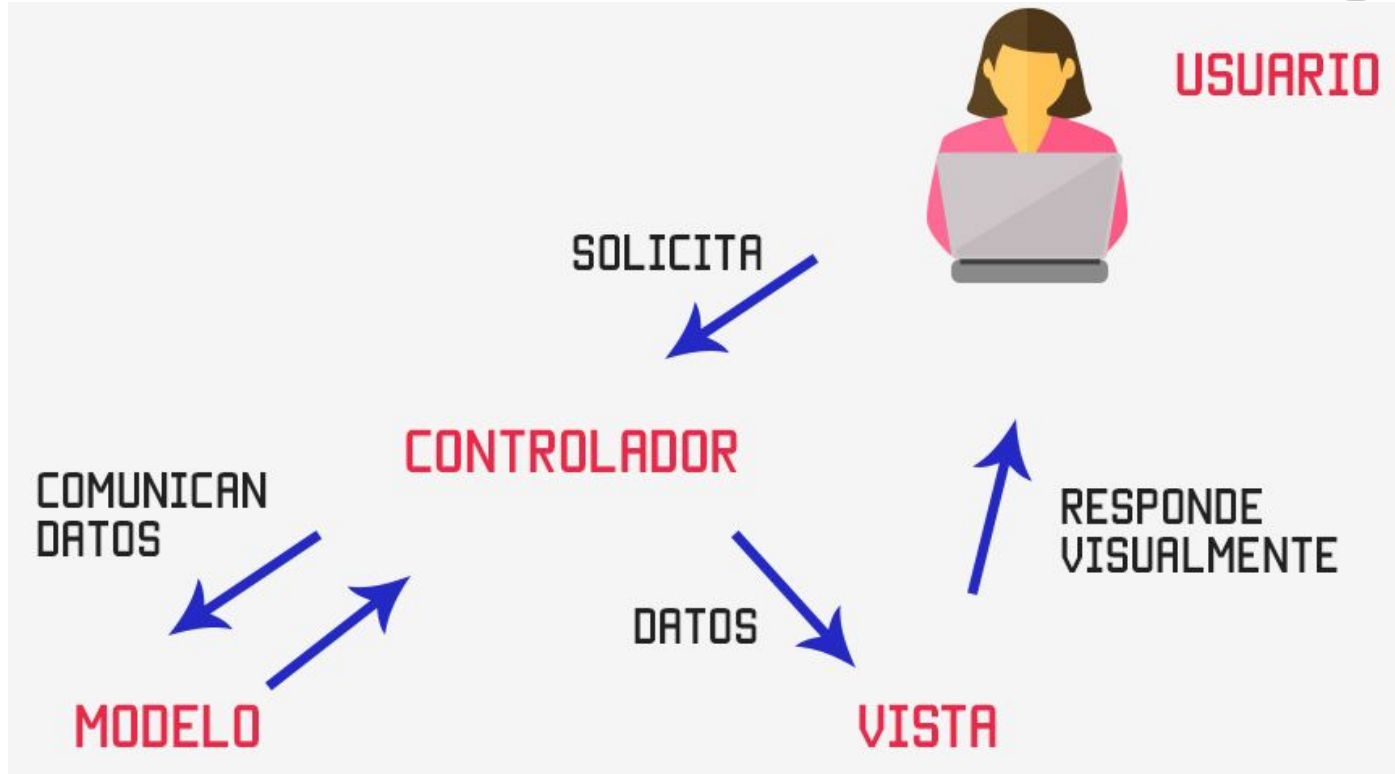
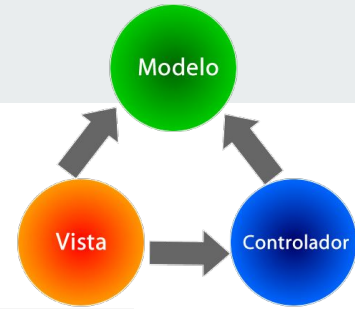
PATRÓN MVC.



PATRÓN MVC.



PATRÓN MVC.





SYMFONY.

Symfony es un **framework** de desarrollo que usa el patrón **MVC**.

Su objetivo es facilitar el desarrollo de aplicaciones web en PHP ofreciendo soluciones para las tareas más habituales.

Si se usa Symfony **debemos adaptarnos** a la forma predeterminada que se plantea para las aplicaciones web pero a cambio tendremos disponibles componentes para muchas tareas como formularios o seguridad que son librerías independientes y pueden utilizarse en otros proyectos.

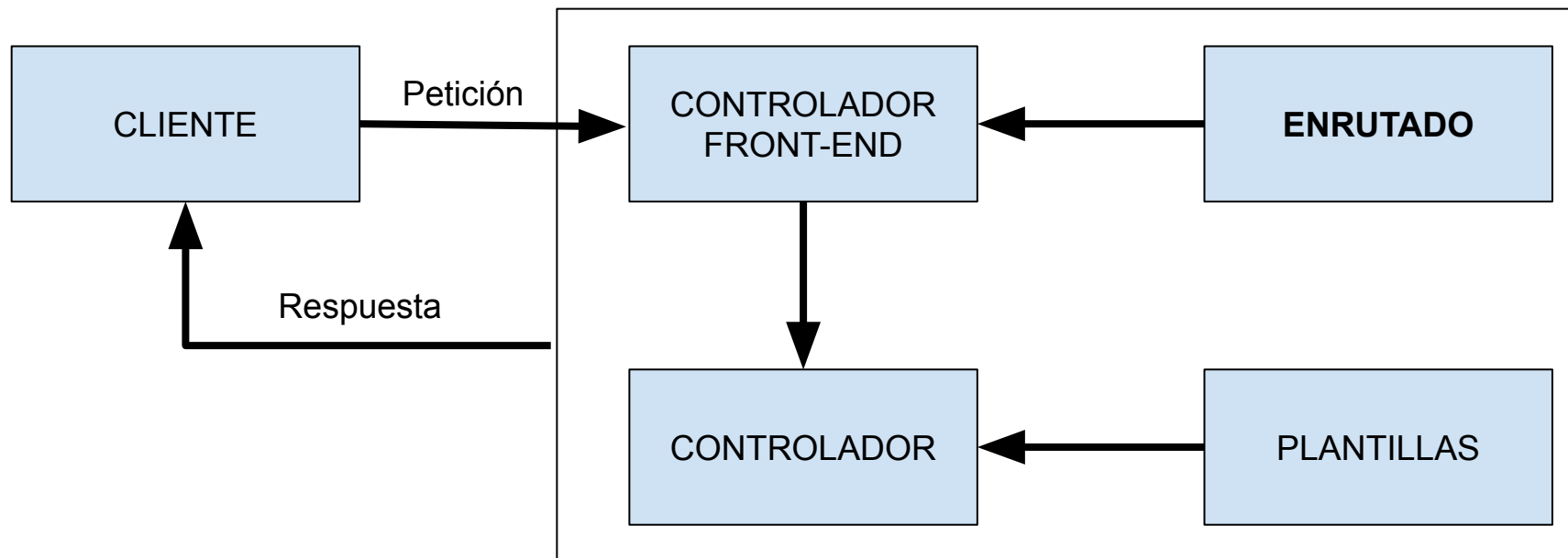
Es de código abierto y tiene detrás una comunidad de desarrolladores bastante amplia.

Alguna estadística de PHP: <https://w3techs.com/technologies/details/pl-php>



SYMFONY.

El procesamiento de una petición sería:





SYMFONY. INSTALACIÓN

Referencia oficial: <https://symfony.com/doc/current/setup.html>

Pasos:

1. Instalación de PHP (ya hecho a través de Xampp).
2. Instalación de composer (ya hecho).
3. (Opcional) Instalación de Symfony CLI (Interfaz de línea de comandos)
4. **Creación de nuevo proyecto Symfony:**
 - a. Ejecutar desde cmd: `composer create-project symfony/skeleton:"7.0.*" proyDir`
 - b. Con CLI sería `symfony new proyDir` aunque debemos tener git.

Donde “poyDir” será un nuevo directorio que se creará en el directorio en el que nos encontremos.



SYMFONY. INSTALACIÓN Symfony CLI

Symfony CLI es necesario para poder “levantar” el servidor local.

Instalación de Symfony CLI: <https://symfony.com/download>

Hay que instalar primero Scoop: <https://scoop.sh/>. Scoop sirve para hacer instalaciones de manera más sencilla.

Para instalarlo debemos hacer en PowerShell, Terminal en Windows 11 (NO administrador):












- Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser (Presionar S)
- Invoke-RestMethod -Uri <https://get.scoop.sh> | Invoke-Expression

Una vez instalado, hay que ejecutar en el cmd, “scoop install symfony-cli”. Si no funciona se puede hacer directamente la descarga.



SYMFONY. INSTALACIÓN

Obtenemos un directorio con todo lo necesario:

Nombre	
 bin	 .env
 config	 .gitignore
 public	 composer
 src	 composer.lock
 var	
 vendor	 symfony.lock



SYMFONY. INSTALACIÓN

Si nuestro proyecto no es un mero microservicio o una API, sino que es una **aplicación web** tradicional, debemos situarnos en el nuevo directorio creado y ejecutar el siguiente comando desde dentro del directorio que hemos creado antes (“cd proyDir”):

“composer require webapp”

Cuando nos pregunte sobre Docker debemos responder (no en este caso).

También se puede crear directamente (aunque no es exacto por alguna versión) ejecutando el comando “composer create-project symfony/website-skeleton proyDir” donde en lugar de usar skeleton se usa el website-skeleton.



SYMFONY. INSTALACIÓN

En este caso tenemos más cosas en el directorio:

Nombre

bin
config
public
src
var
vendor

.env
.gitignore
composer
composer.lock
symfony.lock



bin
config
migrations
public
src
templates
tests
translations
var
vendor

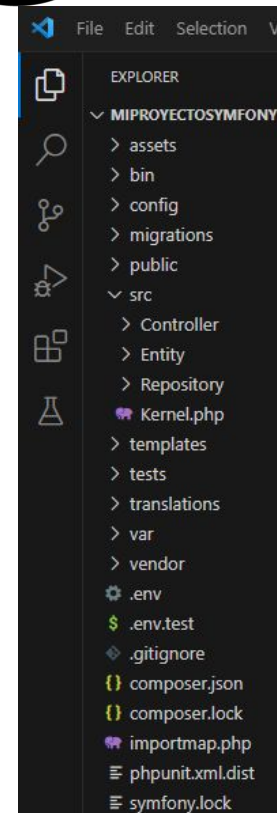
.env
.env.test
.gitignore
composer
composer.lock
phpunit.xml.dist
symfony.lock



SYMFONY. DIRECTORIOS.

Los directorios creados incluyen varias cosas:

- Existe un servidor propio en el directorio **bin** que se puede lanzar si ejecutamos “symfony server:start” en el directorio raíz del proyecto (debemos tener instalado el Symfony CLI) y podremos verlo en “<http://localhost:8000/>” si no está ocupado. Esto es útil si no tenemos servidor propio.
- Directorio **raíz**: con el fichero “.env” de configuración.
- En **/config/packages** tenemos configuraciones de seguridad.
- En **/src/Controller** estarán los controladores.
- En **/src/Entity** las entidades.
- En **/Templates** las plantilla





SYMFONY. ACCESO DESDE APACHE.

Aunque es mejor utilizar el propio servidor para desarrollar, se puede acceder a la aplicación en el Apache integrado en Xampp. Debemos realizar una configuración especial que puede verse en el enlace de más abajo, y podemos ver si symfony está instalado si hacemos lo siguiente:

- Para acceder a la aplicación debemos acceder al directorio “public” donde se aloja el “index”.
- Si queremos, podemos copiar dicho index al directorio del proyecto pero tendremos que cambiar “dirname” por “realpath”, para cambiar el directorio padre por el actual.

https://symfony.com/doc/current/setup/web_server_configuration.html



SYMFONY. CONTROLADORES.

- Son el **elemento principal** en Symfony.
- Son **métodos** que reciben peticiones de cliente, las procesan y generan una salida o una redirección.
- Suelen ser un método de una clase, la “**clase controladora**”, y se almacenan en el directorio “**src/Controller**” y sus subdirectorios.
- Pueden heredar (para usar las funcionalidades presentes allí) de la clase “**AbstractController**” que se puede encontrar con “use `Symfony\Bundle\FrameworkBundle\Controller\AbstractController;`”
- **Obligatoriamente** se les añade a un espacio de nombres “**namespace**” por ejemplo “`namespace App\Controller`”.



SYMFONY. CONTROLADORES. EJEMPLO

Ejemplo para acceder desde “localhost:8000/hola”:

```
<?php // src/Controller/Ejemplo1.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class Ejemplo1 extends AbstractController{
    #[Route('/hola', name:'hola')]
    public function NombreFuncion(): Response{
        return new Response('<html><body>Hola</body></html>');
    }
}
```



SYMFONY. RUTAS.

- Como se ha mencionado, las **URL** con Symfony son las que nos marcan el **controlador** que se va a ejecutar. (CUIDADO con las mayúsculas)
- Éstas URL son diferentes a lo que estamos acostumbrados y debemos tener en cuenta que las rutas de los ficheros no son las utilizadas como **rutas** de acceso a los controladores.
- Se pueden utilizar las rutas para:
 - Pasar parámetros.
 - Introducir valores por defecto.
 - Redireccionar
 - Incluir rutas a nivel de clase.
- Podemos utilizar “**php bin/console debug:router**” desde el directorio del proyecto para ver las rutas disponibles.



SYMFONY. COMPONENTES.

Hemos dicho que utilizamos un framework para utilizar los componentes que nos ofrece. En el caso de Symfony podemos encontrar muchos:

- <https://symfony.com/components>

Veamos, mediante un ejemplo, como usar “clock” (en versión inferior a 7):

- Para instalarlo debemos ejecutar “composer require symfony/clock” desde el directorio del proyecto (si no lo habíamos ya incluido).
- Luego ya podremos utilizarlo como se ve en el ejemplo de la página siguiente.



SYMFONY. CONTROLADORES. EJEMPLO

```
<?php // src/Controller/Ejemplo2.php //Y se comprueba accediendo a "localhost:8000/hola2":
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Clock\Clock;
use Symfony\Component\Clock\MockClock;

class Ejemplo2 extends AbstractController{
    #[Route('/hola2', name:'hola2')]
    public function NombreFuncion(): Response{
        Clock::set(new MockClock()); //Configuración del reloj, no es obligatorio, pero si más rápido
        $clock = Clock::get(); //Instanciamos un reloj
        //$clock->withTimeZone('Europe/Madrid'); //Podemos marcar la zona
        $now = $clock->now(); //Cogemos la hora actual
        $dia=$now->format('h:i:s'); //La guardamos en una variable
        $clock->sleep(2.5); //Espera de 2,5 segundos
        $now = $clock->now(); // Volvemos a coger la hora actual
        $dia=$dia."->".$now->format('h:i:s'); //Añadimos a la variable la nueva hora
        return new Response("<html><body>$dia</body></html>");
    }
}
```



SYMFONY. RUTAS. PARÁMETROS

- El paso de parámetros en Symfony es diferente a lo que estamos acostumbrados con PHP.

`web.php?par1=v1&par2=v2` se transforma en `ruta/v1/v2`

- Los parámetros deben estar definidos en el controlador, como el método para multiplicar siguiente, y podríamos incluir esta función en la misma clase anterior:

```
#[Route("/producto/{num1}/{num2}", name:'producto')]
public function producto($num1, $num2){
    $producto = $num1 * $num2;
    return new Response( $producto);
}
```



SYMFONY. RUTAS. VALORES POR DEFECTO

- Dentro del método podemos realizar cualquier operación habitual de PHP, comprobaciones, y accesos a otros recursos.
- En el caso de los parámetros, si no vienen especificados nuestro controlador no funcionará correctamente.
- Para evitarlo podemos definir valores por defecto de dos formas:
 - Poniéndolo en la lista de argumentos de la función:

```
public function producto($num1=3, $num2=4){
```

- Poniéndolo en la anotación con el símbolo “?”:

```
#[Route("/producto/{num1?3}/{num2?4}")]
```



SYMFONY. RUTAS. PARÁMETROS



EJERCICIO:

Haz un controlador que reciba 1 parámetro y calcule el cuadrado y el cubo y los muestre oportunamente.

Además si el número es 0 o negativo debe indicarlo pero sin mostrar los resultados.



SYMFONY. RUTAS. REDIRECCIONES

Se puede hacer que una ruta haga una redirección a otra ruta (identificada en “name”) en lugar de devolver una página.

- Si la página no tiene parámetros se utiliza:

```
return $this->redirectToRoute("hola");
```

```
//return new RedirectResponse('/hola'); (Otra forma con la URL directa)
```

- Si la página tiene parámetros debemos pasar un array como segundo argumento con todos los parámetros (con su nombre) que queramos:

```
return $this->redirectToRoute("producto", array("num1"=>8,"num2"=>9));
```



SYMFONY. RUTAS. RUTAS A NIVEL DE CLASE

- Las rutas también pueden estar asignadas a una clase como anotaciones antes de declarar la propia clase.
- Con esto se consigue que haya una ruta común para todas aquellos controladores incluidos en dicha clase.

`#[Route("/rutaComun", name:'rutaComun')]`

- En este caso, para acceder a cualquier controlador interno de dicha clase lo debemos anteponer quedando:

`localhost:8000/rutaComun/rutaControlador`

SYMFONY. RUTAS. RUTAS A NIVEL DE CLASE



```
<?php // src/Controller/Ejemplo2.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;

#[Route("/rutaComun", name:'rutaComun')]
class Ejemplo2 extends AbstractController{
    #[Route('/holaComun', name:'holaComun')]
    public function holaComun(){
        return new Response('<html><body>HolaComun</body></html>');
    }
}
```



SYMFONY. PLANTILLAS.

Otro elemento muy importante en Symfony son las plantillas.

Las **plantillas** se utilizan para hacer **devoluciones** de información en lugar de los objetos “Response” que hemos utilizado en los ejemplos anteriores para devolver código HTML por ejemplo.

Las plantillas contienen una **parte estática** (sobre todo de presentación) y pueden recibir **argumentos** que serán los datos a mostrar.

Un caso habitual es cuando queremos devolver una tabla, en cuyo caso podríamos utilizar una plantilla que recibiese los datos en forma de array y generase la tabla oportuna.



SYMFONY. PLANTILLAS.

En Symfony la opción por defecto es la librería “**Twig**”.

Con esta librería podemos crear nuestras propias plantillas de manera independiente al resto de la aplicación y guardarlas en la carpeta “**templates**” de nuestro proyecto.

Las plantillas generadas tendrán la extensión “***.html.twig**” si son de HTML.

De esta manera se consigue **independizar** completamente la presentación de la lógica de negocio, que utiliza las plantillas sin preocuparse de cómo se realiza la presentación, sólo los datos que debe pasar como argumentos.

Además las plantillas son una herramienta potente de por sí, ya que admiten herencia (“**extends**”) e inclusiones entre ellas (“**include**”).

SYMFONY. PLANTILLAS.



En este ejemplo, guardado en la carpeta “templates” como “hola.html.twig” la parte dinámica sólo es “{{nombre}}”:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>HolaNombre</title>
  </head>
  <body>
    Hola {{ nombre }}
  </body>
</html>
```



SYMFONY. PLANTILLAS.

Cuando se utilice la plantilla anterior se deberá pasar un argumento que es el que se utilizará como “nombre”.

Para su utilización, desde el controlador, se debe hacer una llamada a “render” (de la clase AbstractController) con la plantilla y un array para los argumentos:

```
#[Route('/holaNombre/{nombre}', 'holaNombre')]
public function holaNombre($nombre){
    return $this->render('hola.html.twig', array('nombre' => $nombre));
}
```

Se debe llamar a dicha ruta con localhost:8000/holaNombre/Luis.



SYMFONY. PLANTILLAS.

En TWIG podemos configurar 3 tipos de etiquetas:

- `{{ ... }}` para introducir un parámetro o una expresión.
- `{% ... %}` para indicar lógica (como bucles o condicionales) o definir secciones.
- `{# ... #}` para comentarios.

También podemos usar las variables “app.user” (usuario de Symfony), “app.session” y “app.request”.

Con el mismo controlador vamos a comprobar la siguiente plantilla.

```
#[Route('/holaNombre2/{nombre}', name:'holaNombre2')]
public function holaNombre2($nombre){
    return $this->render('holaIF.html.twig', array ( 'nombre' => $nombre));
}
```

SYMFONY. PLANTILLAS.



Se debe llamar a dicha ruta con localhost:8000/holaNombre/Luis:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hola!F</title>
  </head>
  <body>
    {% if nombre=="Luis"%}
      Hola {{ nombre }}
    {%else%}
      Hola, no eres Luis.
    {%endif%}
  </body>
</html>
```



SYMFONY. PLANTILLAS.

EJERCICIO:

Vamos a crear una plantilla, y su correspondiente controlador, que pueda decidir entre sacar el resultado de una división o un error por división entre 0.

El controlador debe enviar un array con 2 argumentos a la plantilla, un valor booleano TRUE si hay error y el resultado.

La plantilla debe elegir qué mostrar según el valor booleano.



SYMFONY. PLANTILLAS.

Otro ejemplo con “for” y una tabla:

El controlador debe devolver un array con los datos:

```
#[Route('/tabla', name:'tabla')]
public function tabla(){
    return $this->render('tabla.html.twig', array("filas"=> array (
        array("dni"=>5, "nombre"=>"Juan"),
        array("dni"=>7, "nombre"=>"Pedro"),
        array("dni"=>12, "nombre"=>"Manuel")
    )));
}
```

SYMFONY. PLANTILLAS.



```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Tabla</title>
  </head>
  <body>
    <table>
      <tr><th>DNI</th><th>Nombre</th></tr>
      {%for fila in filas %}
        <tr><td>{{ fila.dni }}</td><td>{{ fila.nombre }}</td></tr>
      {% endfor %}
    </table>
  </body>
</html>
```




SYMFONY. PLANTILLAS.

Además, en las plantillas podemos utilizar rutas de otros controladores que tengamos definidos.

Ésto puede ser útil para incluirlo dentro de un enlace (href). Para ello debemos usar la función “path”, “{{ path('ruta') }}" y, como es habitual si hay parámetros, acompañado de un array definido de ésta manera: “{{ path('ruta', {'p1':"v1", 'p2':"v2"}) }}"

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Enlace</title>
  </head>
  <body>
    <a href="{{ path('holaNombre', {'nombre': nombre}) }}">Enlace</a>
  </body>
</html>
```



SYMFONY. PLANTILLAS.

Con su correspondiente controlador:

```
#[Route('/enlace/{nombre}', name: 'enlace')]
public function enlace($nombre){
    return $this->render('enlace.html.twig', array ( 'nombre' => $nombre));
}
```

En este caso se debe indicar que el nombre utilizado en la función “path” se corresponde con el que se haya consignado en la definición de la ruta (se debe insertar el “name: 'holaNombre'” en la ruta correspondiente si no se hizo) en el apartado de “name”, que normalmente hemos puesto que sean iguales a la ruta, pero que podría ser diferente.



SYMFONY. SERVICIOS.

Otro elemento a tener en cuenta en Symfony son los “**servicios**” que son objetos que dan una funcionalidad útil para nuestros desarrollos.

Para utilizarlos en nuestros códigos debemos añadirlos como parámetros de los métodos que definamos con la clase adecuada en cada caso.

Algunos de estos objetos son:

- **Request:** que da información sobre la petición recibida en el servidor.
- **SessionInterface:** que da información de las variables de sesión y también nos da funcionalidad para gestionar la propia sesión.

Se pueden crear nuevos servicios y también se pueden ver todos los **servicios disponibles** con “php bin/console debug:autowiring”.



SYMFONY. SERVICIOS.

Un ejemplo para el Request (atención a que al indicar como parámetro se hace saber a Symfony que se va a usar pero no es necesario nada más):

Hay que incluir en el controlador: “*use Symfony\Component\HttpFoundation\Request;*”

```
#[Route('/testRequest', name: 'testRequest')]
public function testRequest(Request $request){
    $ip = $request->getClientIp();
    //$algo = $request->get...
    return new Response(
        '<html><body>IP: '.$ip.'</body></html>'
    );
}
```



SYMFONY. SERVICIOS.

Un ejemplo para el SessionInterface:

Hay que incluir en el controlador: *"use Symfony\Component\HttpFoundation\Session\SessionInterface;"*

```
#[Route("/sesionSet", name: "sesionSet")]
public function sesionSet(SessionInterface $session){
    $session->set("var", 100);
    return $this->redirectToRoute("sesionGet");
}

#[Route("/sesionGet", name: "sesionGet")]
public function sesionGet(SessionInterface $session){
    $var = $session->get("var");
    return new Response('<html><body>'.$var.'</body></html>');
}
```

SYMFONY. BD.



Como ya se ha hablado, Symfony utiliza por defecto Doctrine como ORM aunque podríamos utilizar otros.

La configuración la podemos encontrar en el fichero “.env”. Por defecto encontraremos las siguientes 2 líneas de ejemplo:

```
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8&charset=utf8mb4"  
DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=15&charset=utf8"
```

Debemos dejar sólo aquella que conecte con nuestra base de datos.

`DATABASE_URL="mysql://root:1234@localhost:3306/doctrine"`

Podemos, como hemos hecho anteriormente, generar la base de datos y las entidades (en el directorio src/Entity) pero hay un sistema más sencillo.

SYMFONY. BD.



Una vez configurado “.env” podemos realizar el proceso de creación de la base de datos a través de la consola con los comandos siguientes:

php bin/console doctrine:database:create

Creada la base de datos indicada en “.env” podemos agregar entidades:

php bin/console make:entity

Nos guiará paso a paso para crear una entidad, por ejemplo “Product” con un “identificador” de tipo “integer” (que no será el identificador de la tabla puesto que Symfony genera uno nuevo) y un “name” de tipo “string”.

También genera un “ProductRepository” para gestionar las ocurrencias de la entidad de manera más cómoda.

SYMFONY. BD.



Después debemos realizar el mapeo, que aquí se llama migración:

php bin/console make:migration

Con este comando se autogenerará un fichero php (que podremos ver) que creará las tablas oportunas para hacer el mapeo (la de la entidad generada antes y las que necesita Symfony para funcionar). Para ejecutar la “migración” debemos hacer:

php bin/console doctrine:migrations:migrate

Si accedemos a la base de datos veremos las nuevas tablas. Hay que tener en cuenta que por cada entidad se genera en la tabla correspondiente un “id” de tipo auto incremental.

SYMFONY. BD.



Si se necesita alguna modificación basta ejecutar de nuevo los 3 comandos siguientes y completar con la información pertinente (por ejemplo otra columna):

```
php bin/console make:entity
```

```
php bin/console make:migration
```

```
php bin/console doctrine:migrations:migrate
```

En el caso de que necesitemos comenzar de nuevo una entidad anterior debemos ejecutar:

```
php bin/console make:entity --regenerate
```

Y si queremos que se regeneren los get y set podemos utilizar:

```
php bin/console make:entity --overwrite
```

SYMFONY. BD.



Ahora podemos ya utilizar un nuevo controlador (y el objeto “\$EntityManager” sin tener que crear un bootstrap) para insertar elementos. El siguiente ejemplo inserta una fila cada vez que es llamado (id incremental):

```
use Doctrine\Persistence\ManagerRegistry;
use App\Entity\Product;

...
#[Route("/insertarProducto", name:"insertarProducto")]
public function insertarProducto(ManagerRegistry $doctrine){
    $entityManager = $doctrine->getManager();
    $product = new Product();
    $product->setIdentificador(100);
    $product->setName('n1');
    $entityManager->persist($product);
    $entityManager->flush();
    return new Response('Insertado un elemento con id: '.$product->getId());
}
```

SYMFONY. BD.



Ya también podremos mostrar el contenido de la tabla de varias formas:

- Con el “entityManager” y el método find:

```
#[Route("/mostrarProducto/{id}", name:"mostrarProducto")]  
public function mostrarProducto(ManagerRegistry $doctrine, $id){  
    $entityManager = $doctrine->getManager();  
    $p = $entityManager->find(Product::class, $id);  
    $nom = $p->getName();  
    return new Response('<html><body>'.$id.'-'. $nom . '</body></html>');  
}
```

SYMFONY. BD.



- Con el “getRepository(“Product”)” y el método find:

```
#[Route("/mostrarProducto2/{id}", name:"mostrarProducto2")]  
public function mostrarProducto2(ManagerRegistry $doctrine, $id){  
    $p = $doctrine->getRepository(Product::class)->find($id);  
    $nom = $p->getName();  
    return new Response('<html><body>'.$id.'-'. $nom . '</body></html>');  
}
```

SYMFONY. BD.



- Utilizando el “ProductRepository” que se genera por defecto:

```
use App\Repository\ProductRepository;
```

```
...
```

```
#[Route("/mostrarProducto3/{id}", name:"mostrarProducto3")]
```

```
public function mostrarProducto3(ProductRepository $pR, $id){
```

```
    $p = $pR->find($id);
```

```
    $nom = $p->getName();
```

```
    return new Response('<html><body>'.$id.'"'. $nom . '</body></html>');
```

```
}
```

SYMFONY. BD.



- Existe una característica de Symfony de “fetch automático” (autowiring) en la que podemos hacer que el “id” de la ruta se convierta en un “Product” automáticamente para utilizarlo en la función sólo por estar definido así, pero en Symfony 7 no funciona (aunque debería):

```
#[Route("/mostrarProducto4/{id}", name:"mostrarProducto4")]  
public function mostrarProducto4(Product $p){  
    $id = $p->getId();  
    $nombre = $p->getName();  
    return new Response('<html><body>'.$id.'-'. $nombre . '</body></html>');  
}
```



SYMFONY. FORMULARIOS.

No podía faltar el uso de formularios en Symfony que se realiza a través de un objeto “FormBuilder”.

1. Se usa la función “createFormBuilder()” de “AbstractController”.
2. Se añaden controles con “add()” al formulario.
3. Se obtiene el formulario con getForm().

Por defecto los formularios se envían a la propia ruta y para diferenciar el estado de envío o recibido se usa “\$form->isSubmitted()”



SYMFONY. FORMULARIOS.

- Utilizando el “ProductRepository” que se genera por defecto:

```
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
...
#[Route("/formulario", name: "formulario")]
public function formulario(Request $request)    {
    $f = $this->createFormBuilder();
    $f->add('dato1', TextType::class);           $f->add('dato2', TextType::class);
    $f->add('Enviar', SubmitType::class, array('label' => 'Saludar'));
    $form = $f->getForm();
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $datos = $form->getData();
        $d1 = $datos['dato1'];                  $d2 = $datos['dato2'];
        return new Response('<html><body>Hola '. $d1 . ' '. $d2 . '</body></html>');
    }
    return $this->render('formulario.html.twig', array('form' => $form->createView()));
}
```




SYMFONY. FORMULARIOS.

La plantilla “formulario.html.twig” es bastante simple al utilizar los comandos twig propios para formularios.

```
{{ form_start(form) }}
```

```
{{ form_widget(form) }}
```

```
{{ form_end(form) }}
```



SYMFONY. BD Y FORMULARIOS

EJERCICIO:

Vamos a intentar seguir la siguiente referencia para crear la implementación de la base de datos para profesores e institutos ya utilizada:

<https://symfony.com/doc/current/doctrine/associations.html>

Se debe hacer todo con los comandos de consola vistos y los que aparecen en dicho enlace.

Además debemos crear un formulario para crear nuevos profesores y otro para crear nuevos institutos.



SYMFONY. CORREO ELECTRÓNICO.

En Symfony se utiliza la librería “Mailer” para poder enviar correos. Se incluye por defecto con la opción website y se debe configurar en “.env”.

<https://symfony.com/doc/current/mailer.html>

Por defecto aparece una línea con: “# MAILER_DSN=null://null”

La sintaxis correcta es:

MAILER_DSN=smtp://user:pass@smtp.example.com:port

Por ejemplo para un correo de Gmail podríamos poner:

MAILER_DSN=gmail+smtp://myadresse@gmail.com:mypassword@localhost

Debemos tener en cuenta las mismas cosas que cuando se trabajó con PHP en cuanto a los permisos de enviar correos desde terceras aplicaciones.



SYMFONY. SEGURIDAD.

En cuanto al tema de seguridad, es bastante extenso.

En Symfony tendremos un fichero de configuración (carpeta /config/packages) en el cual se procede a configurar un sistema de “usuarios y roles” similar a los vistos en otros ámbitos (por ejemplo en MySQL).

Podremos abrir sesión, controlar el acceso y cerrar sesión.

<https://symfony.com/doc/current/security.html>



EJERCICIO DE REPASO

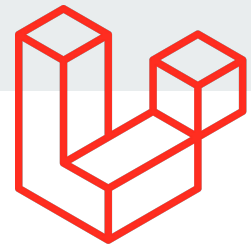
Realizar, mediante Symfony, la siguiente aplicación creando los controladores que se indican cada uno con una ruta diferenciada:

- La base de datos:
 - Entidad juego: con “nombre”, “tipo”, “estudio”, “edad recomendada”
 - Entidad estudio: creadora de los juegos con “nombre” y “sede”.
- Los siguientes controladores:
 - Controlador que inserte un estudio mediante un formulario.
 - Controlador que inserte un juego mediante formulario.
 - Controlador que liste todos los estudios a través de una plantilla.
 - Controlador que liste todos los juegos a través de una plantilla.

EJERCICIO DE REPASO



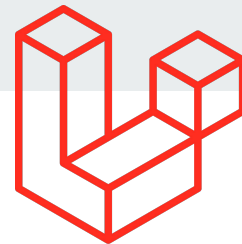
- Controlador que saque los juegos de un estudio (accesible desde cualquier sitio donde se listen los estudios).
- Controlador que permita ver la información de un juego en particular (accesible desde cualquier sitio donde se liste el juego).
- Controlador que permita editar un estudio (accesible desde su página de lista de juegos).
- Controlador que permita editar un juego (accesible desde su página de información).
- Controlador que permita filtrar a través de un formulario según la edad recomendada de los juegos.
- Controlador inicial con menú para los controladores pertinentes.



LARAVEL.

Laravel es otro framework de desarrollo de código abierto con soporte para el patrón MVC usando PHP como lenguaje de servidor y aunque llegó más tarde, en 2011 frente a Symfony (del cual posee muchas dependencias) de 2005 o CodeIgniter de 2006, les ha superado en cuanto a número de usuarios.

La filosofía de Laravel es desarrollar código PHP “de forma elegante y simple, evitando el "código espagueti"” (muchas ramas o hilos que se entremezclan y no son entendibles). Tiene una gran influencia de frameworks como Ruby on Rails, Sinatra y ASP.NET MVC.



LARAVEL. INSTALACIÓN

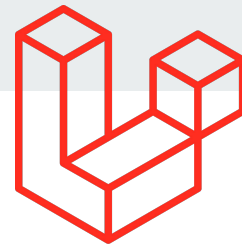
Referencia oficial: <https://laravel.com/docs/10.x>

Al igual que en Symfony, es necesario tener instalados PHP y Composer

Una vez hecho esto debemos crear el nuevo proyecto Laravel ejecutando desde cmd: “*composer create-project laravel/laravel proyDir*” donde “proyDir” será un nuevo directorio que se creará en el directorio en el que nos encontremos.

También podemos instalar Laravel a nivel global en nuestro ordenador ejecutando el comando “*composer global require laravel/installer*” y luego sólo tendríamos que ejecutar “*laravel new proyDir2*”

En el caso de Laravel existe la recomendación de tener instalado [Node.js](https://nodejs.org/).



LARAVEL. SERVIDOR LOCAL

En ambos casos para lanzar el servidor local debemos ir al directorio del nuevo proyecto y ejecutar “*php artisan serve*”.

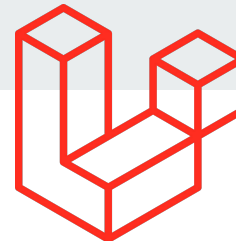
El servidor será accesible a través de la dirección “localhost:8000”.

The screenshot shows the Laravel website homepage. At the top is the Laravel logo. Below it is a grid of four cards:

- Documentation**: Laravel has wonderful, thorough documentation covering every aspect of the framework. Whether you are new to the framework or have previous experience with Laravel, we recommend reading all of the documentation from beginning to end.
- Laracasts**: Laracasts offers thousands of video tutorials on Laravel, PHP, and JavaScript development. Check them out, see for yourself, and massively level up your development skills in the process.
- Laravel News**: Laravel News is a community driven portal and newsletter aggregating all of the latest and most important news in the Laravel ecosystem, including new package releases and tutorials.
- Vibrant Ecosystem**: Laravel's robust library of first-party tools and libraries, such as [Forge](#), [Vapor](#), [Nova](#), and [Envoyer](#) help you take your projects to the next level. Pair them with powerful open source libraries like [Cashier](#), [Dusk](#), [Echo](#), [Horizon](#), [Sanctum](#), [Telescope](#), and more.

At the bottom, there is a footer with a shopping cart icon and the text "Shop Sponsor", and the version information "Laravel v9.48.0 (PHP v8.1.6)".

LARAVEL. DIRECTORIOS



> Este equipo > Disco local (C:) > Laravel > proyDir2

Nombre	Fecha de modificación	Tipo	Tamaño
app	11/01/2023 16:50	Carpeta de archivos	
bootstrap	11/01/2023 16:50	Carpeta de archivos	
config	11/01/2023 16:50	Carpeta de archivos	
database	11/01/2023 16:50	Carpeta de archivos	
lang	11/01/2023 16:50	Carpeta de archivos	
public	11/01/2023 16:50	Carpeta de archivos	
resources	11/01/2023 16:50	Carpeta de archivos	
routes	11/01/2023 16:50	Carpeta de archivos	
storage	11/01/2023 16:50	Carpeta de archivos	
tests	11/01/2023 16:50	Carpeta de archivos	
vendor	23/01/2023 0:52	Carpeta de archivos	

.editorconfig	11/01/2023 16:50	Archivo de origen ...	1 KB
.env	23/01/2023 0:52	Archivo ENV	2 KB
.env.example	23/01/2023 0:52	Archivo EXAMPLE	2 KB
.gitattributes	11/01/2023 16:50	Archivo de origen ...	1 KB
.gitignore	11/01/2023 16:50	Archivo de origen ...	1 KB
artisan	11/01/2023 16:50	Archivo	2 KB
composer.json	11/01/2023 16:50	Archivo de origen ...	2 KB
composer.lock	23/01/2023 0:51	Archivo LOCK	280 KB
package.json	11/01/2023 16:50	Archivo de origen ...	1 KB
phpunit.xml	11/01/2023 16:50	Archivo XML	2 KB
README.md	11/01/2023 16:50	Archivo de origen ...	5 KB
vite.config.js	11/01/2023 16:50	Archivo de origen ...	1 KB

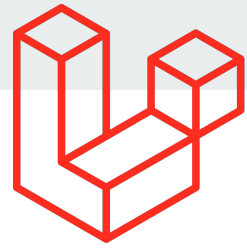
LARAVEL. DIRECTORIOS



Los directorios más importantes en Laravel son (<https://laravel.com/docs/9.x/structure>):

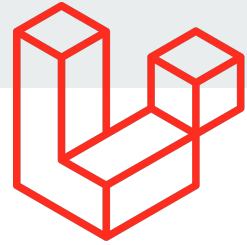
- El directorio **raíz** que contiene archivos como “.env” con opciones de configuración como por ejemplo la base de datos (por defecto MySQL).
- El directorio **“app”** que contendrá el código de nuestra aplicación. Contiene el código de nuestro proyecto “incompleto” como el “Http/Kernel.php” que es el receptor real de las peticiones de cliente y que detecta las configuraciones realizadas. Se mapea con el “namespace” App. Contiene a su vez varios directorios como:
 - “Console” con comandos creados para la consola “artisan”.
 - “Http” con los controladores, peticiones y otro middleware.
 - “Models” con las clases que modelan Eloquent (ORM).

LARAVEL. DIRECTORIOS



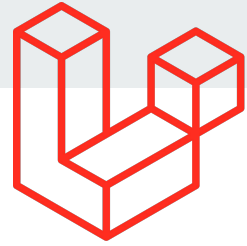
- El directorio “**bootstrap**” que contiene “app.php” que será el encargado de gestionar el arranque de la aplicación. Habitualmente no se debe cambiar nada aquí.
- El directorio “**config**” con ficheros de configuración como el “app.php” en donde podemos cambiar de idioma o la URL por defecto y realizar configuraciones acerca de los “service providers”.
- El directorio “**database**” que contiene las migraciones de la base de datos y otros elementos.
- El directorio “**lang**” para los ficheros de idioma.
- El directorio “**public**” en donde encontramos el “index.php” que por defecto es la entrada para cualquier petición. Aquí tendremos nuestros códigos Javascript, CSS e imágenes.

LARAVEL. DIRECTORIOS



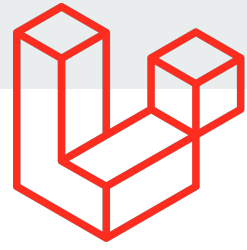
- El directorio “**routes**” que contiene las definiciones de las rutas de la aplicación. Por defecto se incluyen:
 - “web.php” que contiene rutas del “routeServiceProvider” que pertenecen al grupo web
 - “api.php” que contiene rutas del “routeServiceProvider” que pertenecen al grupo api.
 - “console.php” que contiene definiciones para los comandos basados en consola.
 - “channels.php” en donde podemos registrar canales de eventos “broadcasting” soportados

LARAVEL. DIRECTORIOS



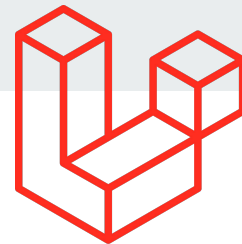
- El directorio “**resources**” que contiene las vistas, otro concepto importante en Laravel.
- El directorio “**storage**” que contiene logs, plantillas Blade (similares a Twig), cachés y otros ficheros autogenerados por Laravel. Se puede utilizar parte de este directorio (“storage/app/public”) para hacer públicos determinados ficheros que se generen en la aplicación como un logo o un avatar.
- El directorio “**tests**” que contiene pruebas automatizadas, por ejemplo para PHPUnit.
- El directorio “**vendor**” que contiene las dependencias “Composer”.

LARAVEL. FRONT-END



Con Laravel podemos construir el front-end con varias tecnologías como por ejemplo:

- Desde PHP mediante plantillas Blade (parecidas a Twig) y vistas utilizando los comandos “echo” necesarios.
- Introduciendo algún framework JavaScript como Vue, React, Livewire o Alpine.js
- Utilizando “Inertia” (plantillas almacenadas en “resources/js/Pages”) que formatean el código) como puente entre Laravel y Vue o React.

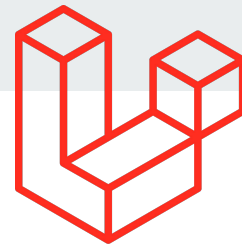


LARAVEL. BD

Para usar la base de datos desde Laravel tenemos que configurar primero en el archivo “.env” el servidor, puerto, usuario y contraseña (en las propiedades DB_CONNECTION y siguientes), como hemos hecho en Symfony, aunque aquí hay otro fichero en el directorio config que tiene también parámetros de configuración de las conexiones a bases de datos.

En cuanto al ORM utilizado por defecto, en Laravel es Eloquent. El proceso sigue los mismos pasos (<https://laravel.com/docs/10.x/eloquent>) de creación del modelo que hemos visto en Symfony.

Como curiosidad, si queremos utilizar otro gestor de base de datos podemos migrar desde la base de datos anterior con “php artisan migrate”.

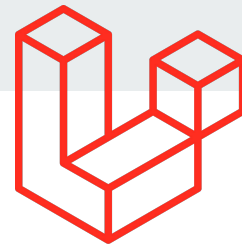


LARAVEL. RUTAS

En Laravel las rutas tienen una complejidad mucho mayor.

Debemos quedarnos con que podemos configurar muchas rutas desde el archivo “routes/web.php” con el formato:

```
<?php  
  
class Service{  
  
    ...  
  
}  
  
Route::get('/', function (Service $service) {  
    die(get_class($service));  
  
});
```

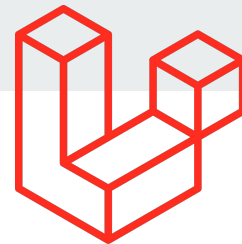


LARAVEL. SERVICE CONTAINERS

Para el despliegue de proyectos Laravel utiliza lo que se llama **contenedores de servicio**. Se encargan de gestionar dependencias entre clases.

Los contenedores de servicio son donde un desarrollador une todo lo necesario para ejecutar una aplicación Laravel, es decir, aquí podemos empezar a crear lo que hemos llamado “**controladores**” en Symfony.

Por ejemplo en un “Service Container” podemos crear una ruta.



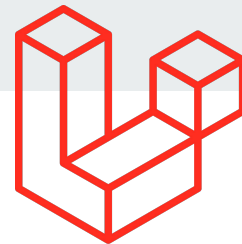
LARAVEL. SERVICE PROVIDERS

Otro concepto muy importante en Laravel son los proveedores de servicio o “**service providers**”. Son responsables de iniciar los componentes instalados, como la base de datos, la caché o el enrutamiento.

Todos ellos deben estar configurados dentro del fichero “config/app.php”.

Uno de los servicios más importantes es precisamente el de enrutamiento que podemos encontrar en “App\Providers\RouteServiceProvider”. Este servicio se encarga de cargar las rutas de los ficheros o los controladores de la aplicación una vez recibida una petición.

En la misma ruta, en el fichero “AppServiceProvider” podremos crear nuestros propios servicios.



LARAVEL. FACADES

El último concepto que veremos, que también es muy importante en Laravel son las fachadas o “**facades**” que son interfaces estáticas de acceso a nuestros contenedores de servicio.

En una aplicación una fachada será una clase que da acceso al objeto creado por el controlador.

En el siguiente ejemplo de un controlador, se usa la “facade” Cache para hacer el get:

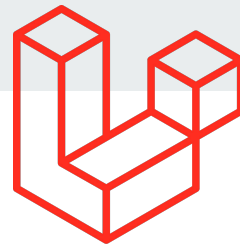
```
... use Illuminate\Support\Facades\Cache; ...
```

```
public function showProfile($id){
```

```
    $user = Cache::get('user:'.$id);
```

```
    return view('profile', ['user' => $user]);
```

```
}
```



LARAVEL. CÓMO EMPEZAR

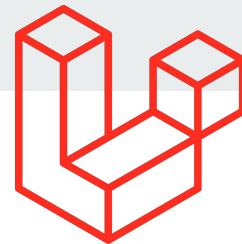
Documentación: <https://laravel.com/docs/10.x/starter-kits>

La idea de estos kits de iniciación es tener ya implementado algún tipo de componente que ayuden a iniciar nuestra aplicación de una manera más rápida y ágil.

Hay un par de Starter kits para trabajar con PHP y Livewire:

- Breeze, un sistema de autenticación, que utiliza plantillas Blade.
- Jetstream, también con autenticación, que utiliza plantillas Tailwind.

Después de instalar estos starter kits, se migraría la base de datos y se instalarán otros paquetes de dependencias con npm.



LARAVEL. STARTER KITS. BREEZE

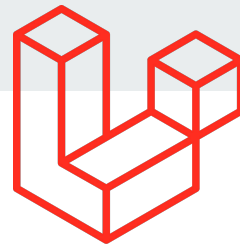
Breeze es un proyecto muy simple pero con componentes de autenticación como login, registro, etc con plantillas Blade y con Tailwind CSS. También se puede usar con Vue/React e Inertia.

Para descargar breeze tenemos que ejecutar el siguiente comando sobre el proyecto ya creado (*composer create-project laravel/laravel proyDir*):

```
composer require laravel/breeze --dev
```

Después de la descarga debemos instalarlo en el proyecto ejecutando:

```
php artisan breeze:install
```



LARAVEL. STARTER KITS. JETSTREAM

Por su parte, **Jetstream** pretende ir un paso más añadiendo los componentes de autenticación pero usando en este caso el llamado “Laravel Sanctum” (una librería más de Laravel). Permite usar plantillas Tailwind CSS y también se puede usar con Lifewire e Inertia.

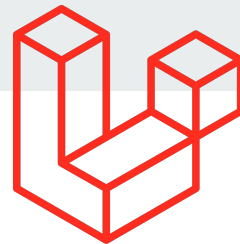
Para instalar Jetstream también tenemos que tener creado un proyecto (*composer create-project laravel/laravel proyDir*) y ejecutar:

```
composer require laravel/jetstream
```

Y después de la creación del proyecto debemos ejecutar igualmente:

```
php artisan jetstream:install
```

LARAVEL. BOOTCAMP. PROYECTO CHIRPER



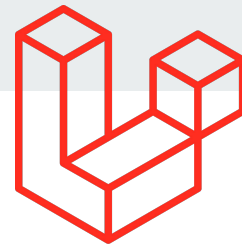
Bootcamp de laravel: <https://bootcamp.laravel.com/>

El bootcamp nos sirve para seguir paso a paso la creación de un proyecto Laravel. Vamos a hacer una prueba creando un proyecto “Chirper”

```
composer create-project laravel/laravel chirper
```

Se debe cambiar la base de datos a usar: `DB_CONNECTION=sqlite`

Luego podremos instalar los componentes que queramos, como Breeze, y arrancar todos los procesos para crear “chirps” que son mensajes enviados por los usuarios pero antes debemos crear el modelo de la base de datos, las rutas, las plantillas, etc.



LARAVEL. HOMESTEAD

Laravel Homestead es un entorno de desarrollo que permite el desarrollo en una máquina virtual proporcionando una caja Vagrant pre-empaquetada. Homestead incluye todo lo necesario para el desarrollo de aplicaciones Laravel, incluyendo PHP, MySQL, Nginx, Composer, Redis, y mucho más, por lo que el desarrollador no necesita instalar estos paquetes en su máquina local. Homestead requiere la instalación de Vagrant y VirtualBox o Parallels.

<https://laravel.com/docs/10.x/homestead>

WEBGRAFÍA:



Frameworks y patrones:

- Explicación de patrones de diseño: <https://devexperto.com/patrones-de-diseno-software/>
- Explicación de patrones arquitectónicos: <https://medium.com/@maniakhitoccori/los-10-patrones>
- Teoría de frameworks: https://developer.mozilla.org/es/docs/Learn/Server-side/First_steps/Web
- Evolución de uso: <https://www.youtube.com/watch?v=S9Jcl254DZI>
- Comparación de frameworks PHP: <https://kinsta.com/es/blog/frameworks-php/>

Symfony:

- Referencia oficial de Symfony: <https://symfony.com/>
- Instalación: <https://symfony.com/download#step-1-install-symfony-cl>
- Componentes HTTPFoundation: https://symfony.com/doc/current/components/http_foundation.html
- Componentes Session: <https://symfony.com/doc/current/session.html>

Laravel:

- Referencia oficial de Laravel: <https://laravel.com/>
- Documentación oficial: <https://laravel.com/docs/10.x>
- Empezar de 0 con Laravel Homestead: <https://laravel.com/docs/10.x/homestead>
- <https://kinsta.com/es/base-de-conocimiento/que-es-laravel/>



DESARROLLO DE APLICACIONES CON FRAMEWORKS. EJERCICIO



EJERCICIO 27-1-2023

Realizar, mediante Symfony, la siguiente aplicación creando los controladores que se indican cada uno con una ruta diferenciada:

- La base de datos:
 - Entidad coche con datos matrícula, marca, modelo, color.
- Los siguientes controladores:
 - Controlador que inserte un coche mediante un formulario. Tras la inserción se debe cargar el listado de todos los coches que se pide a continuación.
 - Controlador que extraiga datos de todos los coches y los muestre en forma de tabla a través de una plantilla.

EJERCICIO 27-1-2023



Más controladores:

- Controlador que de un listado de marcas.
- Controlador que pida una marca y muestre los coches de esa marca.
- Controlador que liste todos los coches y que permita seleccionar uno y mostrar sus datos por medio de una plantilla.
- Controlador que permita editar los datos de un coche.
- Menú para los controladores anteriores



EJERCICIO 27-1-2023

La entrega se realizará mediante un archivo comprimido con los siguientes ficheros **exclusivamente**.

- Fichero de texto con la lista de comandos utilizados en consola para la creación del proyecto y de la base de datos (la parte de las tablas se puede copiar completa con el diálogo de creación).
- Fichero “.env”.
- Fichero/s de “src/Controller” creado para tus controladores.
- Fichero/s de “Templates” creados para tus plantillas.
- De “src/Entity” no hará falta ya que se crearon a partir de los comandos.



EJERCICIO 27-1-2023

Rúbrica de evaluación:

Cada petición resuelta correctamente se obtendrá el apartado entero, si hay deficiencias la mitad y si es visiblemente erróneo será 0:

- Creación del proyecto y configuración correcta: 0,5 puntos.
- Base de datos creada con todos los datos: 1 punto.
- Controlador de inserción con formulario: 1,5 puntos.
- Controlador de extracción con tabla y plantilla: 1,5 puntos.
- Controlador de marcas: 1 punto.
- Controlador de lista de coches y envía a la información de un coche: 1 punto.
- Controlador para edición: 1,5 puntos.
- Controlador de búsqueda por marca: 1 punto.
- Menú de controladores: 1 punto.



DESARROLLO DE APLICACIONES CON FRAMEWORKS. EJERCICIO DE RECUPERACIÓN



EJERCICIO DE RECUPERACIÓN

Realizar, mediante Symfony, la siguiente aplicación creando los controladores que se indican cada uno con una ruta diferenciada:

- La base de datos:
 - Entidad provincia: con “denominación” y “extensión”.
 - Entidad ciudad: con “nombre”, “población” y “provincia”.
- Los siguientes controladores:
 - Controlador que inserte una provincia mediante un formulario.
 - Controlador que inserte una ciudad mediante formulario.
 - Controlador que liste todas las provincias a través de una plantilla.
 - Controlador que liste todas las ciudades a través de una plantilla.



EJERCICIO DE RECUPERACIÓN

- Controlador que saque las ciudades de una provincia (accesible desde cualquier sitio donde se listen las provincias).
- Controlador que permita ver la información de una ciudad en particular (accesible desde cualquier sitio donde se liste la ciudad).
- Controlador que permita editar una provincia (accesible desde su página de lista de ciudades).
- Controlador que permita editar una ciudad (accesible desde su página de información).
- Controlador que permita listar provincias a través de un formulario filtrando según la extensión de la provincia.
- Controlador que permita listar ciudades a través de un formulario filtrando según la población de la ciudad.
- Controlador inicial con menú para los controladores pertinentes.



EJERCICIO DE RECUPERACIÓN

La entrega se realizará mediante un archivo comprimido con los siguientes ficheros **exclusivamente**.

- Fichero de texto con la lista de comandos utilizados en consola para la creación del proyecto y de la base de datos (la parte de las tablas se puede copiar completa con el diálogo de creación).
- Fichero “.env”.
- Fichero/s de “src/Controller” creado para tus controladores.
- Fichero/s de “Templates” creados para tus plantillas.
- De “src/Entity” no hará falta ya que se crearon a partir de los comandos.



EJERCICIO DE RECUPERACIÓN

Rúbrica de evaluación:

Cada petición resuelta correctamente se obtendrá el apartado entero, si hay deficiencias la mitad y si es visiblemente erróneo será 0:

- Creación del proyecto y configuración correcta: (0,5 p).
- Base de datos creada con todos los datos (1 p).
- Controladores de inserción con formulario (1,5 p).
- Controladores de listados con plantillas (1,5 p).
- Controlador de ciudades de una provincia (1 p).
- Controlador de información de una ciudad (0,5 p).
- Controladores de edición de provincia y de ciudad (1,5 p).
- Controladores de filtro por extensión y por población (1,5 p).
- Controlador inicial de menú (1 p).