# Desenvolvemento dunha aplicación web cun framework: Symfony

Sitio: Aula Virtual do IES de Teis
Curso: Desarrollo web en entorno servidor 2023-24 (DAW-DUAL-A)
Libro: Desenvolvemento dunha aplicación web cun framework: Symfony

Impreso por: Abel Mahón Cortés
Data: Luns, 8 de Abril de 2024, 16:52

Táboa de contidos

# 1. Symfony

Symfony es un framework open-source de desarrollo en PHP. Un framework podría definirse como el conjunto de:

- **Una caja de herramientas o toolbox** : un conjunto de componentes de software prefabricados y rápidamente integrables.
  - Escribir menos código => menos riesgo de error, mayor productividad
- **Una metodología** : una arquitectura ya establecida para organizar las aplicaciones. Un enfoque estructurado permite trabajar de manera eficiente. Se promueve con ello el uso de *Mejores Prácticas* y patrones de diseño para garantizar la estabilidad, mantenimiento y capacidad de actualización de las aplicaciones.

Por lo tanto Symfony:

- Proporciona estructura el código fuente forzando al desarrollador a crear código más legible y más fácil de mantener.
- Simplifica el desarrollo de las aplicaciones, ya que automatiza muchos de los patrones utilizados para resolver las tareas comunes.
- Facilita la programación de aplicaciones sistematizando operaciones complejas en instrucciones sencillas.
- Facilita el mantenimiento, pues conforma un estándar a seguir por los desarrolladores
- Utiliza el patrón MVC: separa la lógica de negocio, la lógica de servidor y la presentación de la aplicación web.
- Utiliza componentes (Symfony components) que son bibliotecas (librerías) desacopladas para aplicaciones PHP. Para su instalación se recomienda el **gestor de dependencias Composer** que instalamos en la UD3 para trabajar con MongoDB

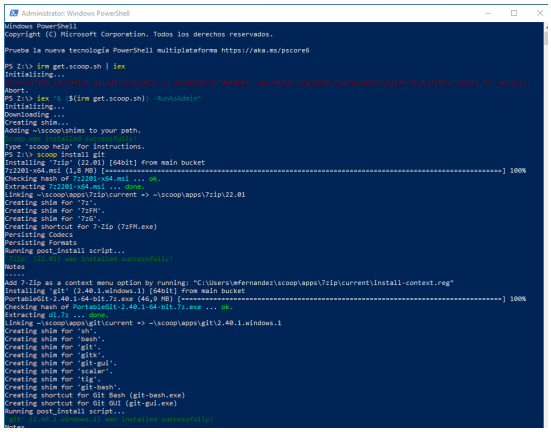
## 1.1. Instalación e configuración del entorno de desarrollo

Seguiremos las instrucciones de configuración de la documentación oficial: https://symfony.com/doc/current/setup.html

1) PHP8.2 y Composer deberían estar instalados con XAMPP y la UD3 con MongoDB

2) Symfony tiene una herramienta CLI (interfaz de línea de comandos) que vamos a utilizar. Para instalarla, necesitaremos a su vez Scoop (un instalador por línea de comandos para Windows) , tal y como indica la documentación.

Antes tendremos que cambiar la política de ejecución para que nos permita instalarlo.




Para PowerShell como administrador:

habrá que ejecutar

```
iex "& {$(irm get.scoop.sh)} -RunAsAdmin"
```

Fuente: GitHub - ScoopInstaller/Install: 🕹️ Next-generation Scoop (un)installer

3) Podemos instalar git con scoop si no lo tenemos instalado

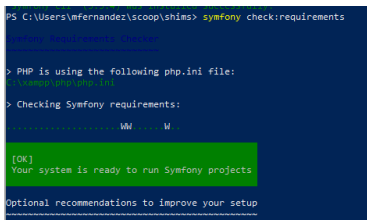
scoop install git

4 ) Instalaremos la aplicación Symfony-CLI con:

```
scoop install symfony-cli
```

Comprobaremos que el sistema tiene todos los requisitos para trabjar con symfony con:

```
symfony check:requirements
```



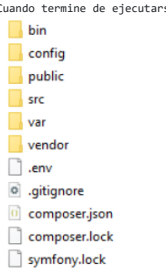

5) Para crear un nuevo proyecto web con Symfony, tenemos 2 opciones: una con Composer y otra con Symfony CLI. Ambas producen la misma estructura de ficheros y descarga de dependencias, pero Symfony CLI requiere instalar y configurar git con usuario.

Usaremos la opción de Composer.

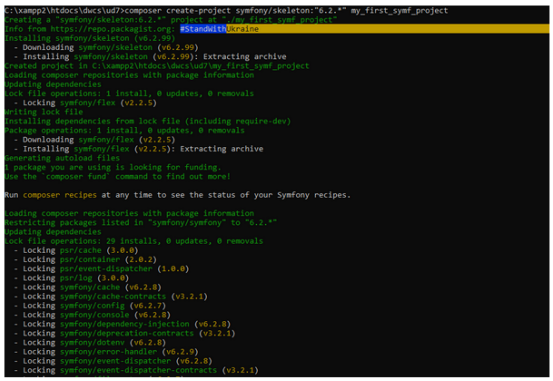
a) Crearemos un proyecto web Symfony nuevo con Composer tal y como indica la documentación navegando con la ventana de comandos en la ubicación que deseemos :

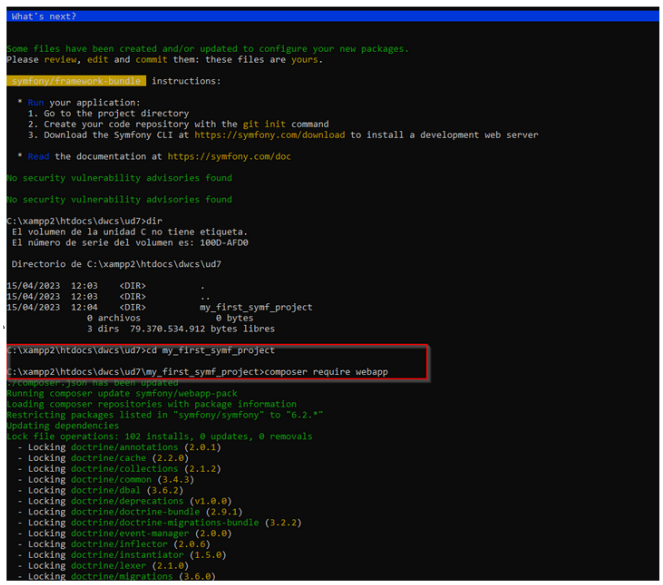
```
composer create-project symfony/skeleton:"7.0.*" my_first_symf_project
```

Cuando termine de ejecutarse, se habrá creado esta estructura de directorios:

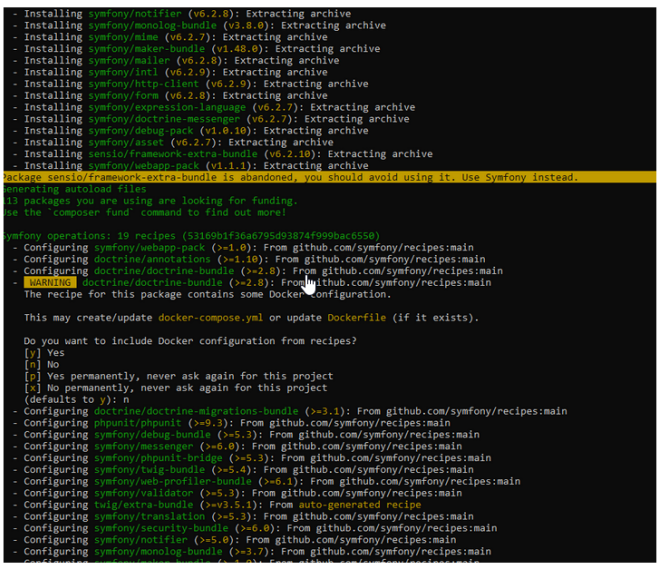


```
cd  my_first_symf_project
composer require webapp

Creará una estructura de directorios y descargará las librerías más habitualmente usadas en una aplicación web con Symfony
```

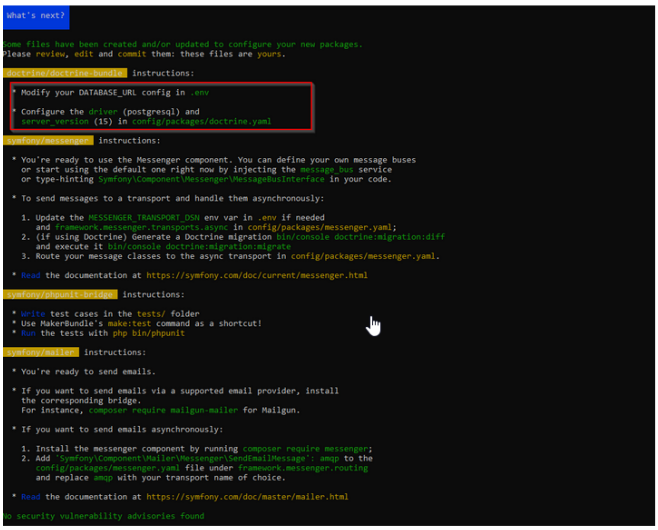


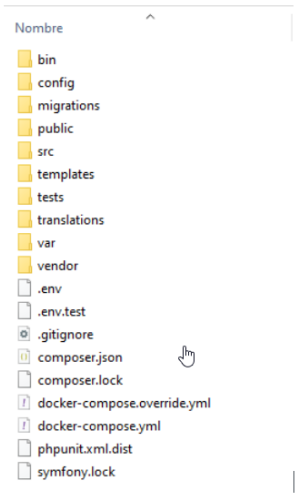

Indicaremos que no vamos a trabajar con Docker



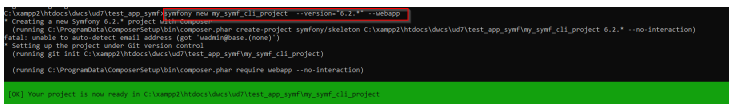

Aquí nos indica que tendremos que cambiar algunos ficheros de configuración de la BD, pero por el momento no lo vamos a hacer




Finalmente tendremos esta estructura de directorios:

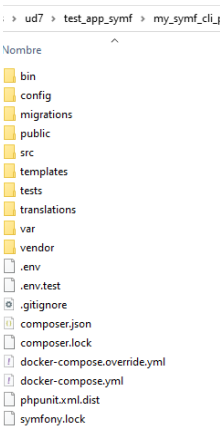



b) Crear un nuevo proyecto con Symfony CLI requiere configurar el usuario de git




Resultado con Symfony-CLI:

```
symfony new my_project_directory --version="7.0.*" --webapp
```




6) Instalaremos un componente que nos va a facilitar la creación de clases y configuraciones **maker-bundle**: https://symfony.com/bundles/SymfonyMakerBundle/current/index.html

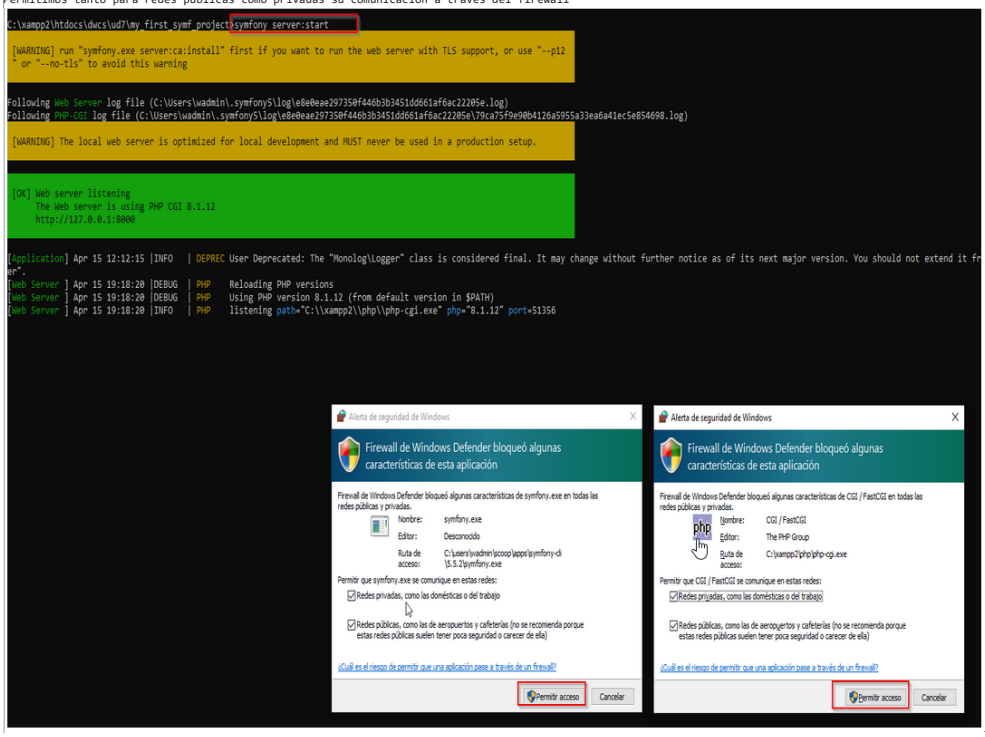
```
composer require --dev symfony/maker-bundle
```

maker bundle

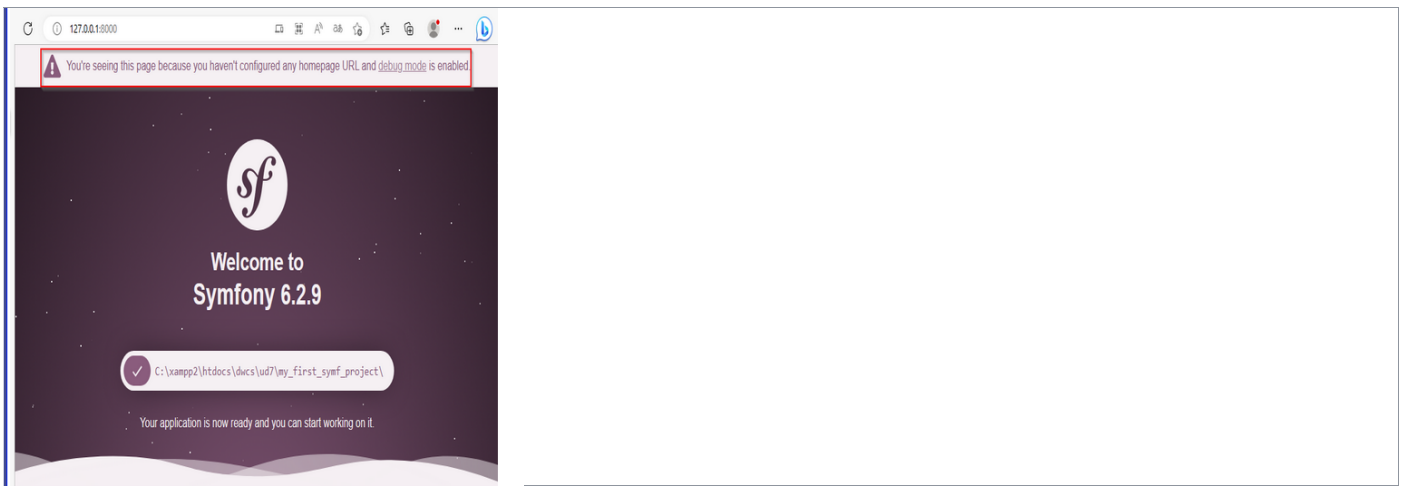
7) Symfony viene con un servidor de desarrollo integrado. Se puede arrancar con el comando:

```
symfony server:start
```

Permitimos tanto para redes públicas como privadas su comunicación a través del firewall



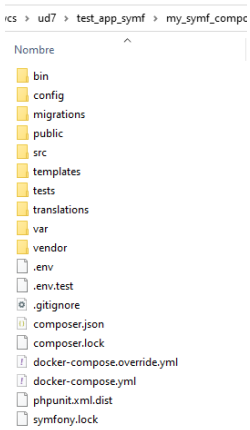

Visitamos con el navegador la url http://127.0.0.1:8000 y deberíamos ver la página por defecto de Symfony cuando nuestra aplicación no tiene una página de inicio




Cuando se quiera parar el servidor se puede usar CTRL+c o **con symfony server:stop**

## 1.2. Estructura de directorios de Symfony

La estructura de carpetas y directorios que se ha creado con Composer es la siguiente:




A grandes rasgos, los contenidos de los directorios más relevantes son:

**`bin/`**
Ficheros ejecutables (e.g. `bin/console`).

**`config/`**
Ficheros de configuración por defecto en formato .yaml (soporta XML y php)

**`migrations/`**
Ficheros que permitirán actualizar la BD

**`public/`**
El directorio raíz del proyecto donde situaremos los activos css y jss

**`src/Controller`**
Incluirá las clases que ejerzan el rol de controlador

**`src/Entity`**
Incluirá las clases que tengan correspondencia en BD

**`src/Repository`**
Incluirá las clases permitirán consultar la BD por cada entidad y realizar operaciones CRUD de una entidad específica.

**`templates/`**
Donde se situarán las plantillas Twig con el marcado html de las vistas. Twig es un motor de plantilla para el lenguaje de programación PHP. Los motores de plantillas o templates ayudan a dividir el código HTML en partes mas pequeñas, donde podemos reutilizar en otros archivos HTML y además tienen la capacidad de utilizar variables, con la finalidad de simplificar el código.

`tests/`
Pruebas automáticas (e.g. Unit tests).

**`translations/`**
Ficheros de internacionalización i18n

**`var/`**
Ficheros generados (cache, logs, etc.).

**`vendor/`**
Librerías de terceros

**`.env`**
Fichero donde se almacena la configuración de la BD y de otros paquetes como el correo electrónico. Puede tener variantes dependiendo de los entornos, .env.local (que sobreescribe las variables declaradas para una máquina concreta) o .env.test para un entorno de pruebas.

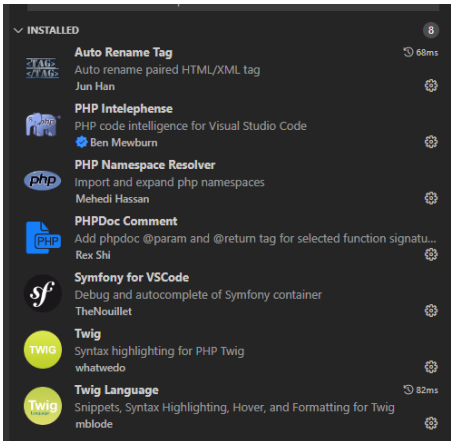
**`symfony.lock`**
**Muchos paquetes/bundles de Symfony definen "recetas" o *recipes*, que son un conjunto de instrucciones automatizadas para instalar y habilitar paquetes en las aplicaciones de Symfony. Flex realiza un seguimiento de las recetas que instaló en un `symfony.lock` archivo**

`composer.json`
Un fichero en formato JSON que indica las dependencias del proyecto. Se sitúa normalmente en la raíz del proyecto.

`composer.lock`
Composer apunta en el archivo `composer.lock` la versión exacta que se ha instalado de cada librería. De esta forma, el proyecto se fija a unas determinadas versiones.

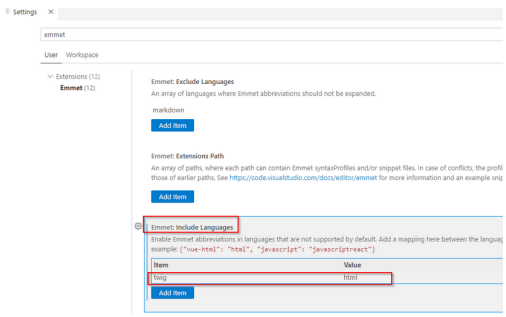
## 1.3. Algunas extensiones de utilidad

Instalaremos las extensiones que se ven en la imagen siguiente por el momento




La extensión de Twing Language tiene una versión 2, que es la que usaremos

3- En Archivo > Preferencias>Configuración. Buscamos Emmet, en la sección de Include Languages añadimos twig y html

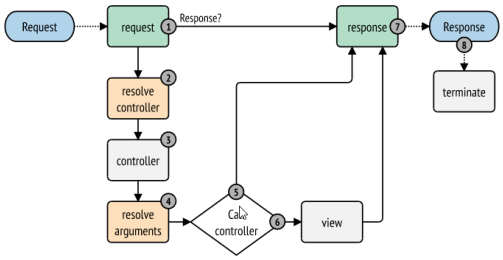


## 1.4. Flujo de una petición-respuesta en Symfony

El componente HttpKernel proporciona un proceso estructurado para convertir un `Request`en un `Response`.

El proceso de comunicación petición - respuesta es el siguiente:

1. El **usuario** solicita un **recurso** en un **navegador**
2. El **navegador** envía una **petición** al **servidor**
3. **Symfony** le da a la **aplicación** un objeto **Request** que encapsula la petición **HTTP**
4. La **aplicación** procesará la petición utilizando los datos del objeto **Request** y posteriormente genera un objeto **Response**
5. El **servidor** devuelve la **respuesta** al **navegador**
6. El **navegador** muestra el **recurso** al **usuario**




El proceso, aunque más sofisticado y robusto, se parece en cierto modo al FrontController.php que hemos implementado.

A partir de la ruta de la URL:

- se deciden aspectos de seguridad como la autorización de un usuario a la ruta
- se deduce el controlador que ha de atender la petición
- se extraen los parámetros de la URL para que el controlador tenga acceso a ellos
- se invoca el método correspondiente en el controlador, que procesará o delegará el procesado de la respuesta en otros componentes
- con la respuesta del controlador se crea el objeto Response de Symfony

## 1.5. Creación de un controlador

- Los controladores son clases que se colocarán en src/Controller
- Normalmente extenderán de una clase Symfony AbstractController que nos permitirá beneficiarnos de algunos métodos predefinidos para generar objetos Response.
- Se pueden crear manualmente, pero podemos hacerlo de forma automatizada ayudándonos de MakerBundle.

php bin/console make:controller

Esto creará dos ficheros nuevos:

```
PS C:\xampp2\htdocs\dwcs\ud7\test_app_symf\my_symf_composer_project> php bin/console make:controller

 Choose a name for your controller class (e.g. GrumpyElephantController):
 > LuckyController

 created: src/Controller/LuckyController.php
 created: templates/lucky/index.html.twig

  Success!

 Next: Open your new controller class and add some pages!
```

- LuckyController.php

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class LuckyController extends AbstractController
{
    #[Route('/lucky', name: 'app_lucky')]
    public function index(): Response
    {
        return $this->render('lucky/index.html.twig', [
            'controller_name' => 'LuckyController',
        ]);
    }
}

```

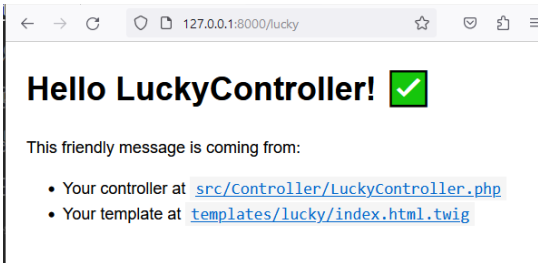
- templates/lucky/index.html.twig

```twig
{% extends 'base.html.twig' %}

{% block title %}Hello LuckyController!{% endblock %}

{% block body %}
<style>
    .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5 sans-serif; }
    .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>

<div class="example-wrapper">
    <h1>Hello {{ controller_name }}! ✅</h1>

    This friendly message is coming from:
    <ul>
        <li>Your controller at <code><a href="{{ 'C:/xampp2/htdocs/dwcs/ud7/test_app_symf/my_symf_composer_project/src/Controller/LuckyController.php'|file_link(0) }}">src/Controller/LuckyControl
        <li>Your template at <code><a href="{{ 'C:/xampp2/htdocs/dwcs/ud7/test_app_symf/my_symf_composer_project/templates/lucky/index.html.twig'|file_link(0) }}">templates/lucky/index.html.twig<
    </ul>
</div>
{% endblock %}
```

Veremos que si arrancamos el servidor con

```
symfony server:start
```

Y visitamos la URL: http://127.0.0.1:8000/lucky veremos que se nos muestra el contenido de la plantilla



# 1.6. Rutas

Las rutas en Symfony pueden configurarse en distintos modos, pero Symfony recomienda el uso de attributes de PHP 8

Los atributos en PHP 8 podrían asemejarse conceptualemente a anotaciones de Java, pero usan la sintaxis #[nombreAtributo] en lugar de @atributo

El nombre del atributo es Route y entre paréntesis van sus argumentos: el path y el nombre con el que se conocerá a esta ruta (Podrían utilizarse más)

```
#[Route('/lucky', name: 'app_lucky')]
  public function index(): Response
  {
     return $this->render('lucky/index.html.twig', [
        'controller_name' => 'LuckyController',
     ]);
  }
}
```

Se indica que se añade una ruta de forma que cuando llegue una petición a la URL /lucky, esta será atendida por el controlador LuckyController y el método index. Además a la ruta se le asigna un *name* que será útil para referirse a esa ruta (crear enlaces o generar redirecciones).

Por defecto, en la creación del proyecto, se ha configurado para que las rutas se hagan a través de atributos (config/routes.yaml)

my_symf_composer_project > config > ! routes.yaml

```yaml
controllers:
    resource:
        path: ../src/Controller/
        namespace: App\Controller
    type: attribute

```

## Rutas con parámetros

- Los parámetros van entre llaves{}
- El nombre del parámetro se usa para crear una variable con el mismo nombre que se pasa por parámetro al Controlador

```php
// src/Controller/LuckyController.php
namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class LuckyController
{
    #[Route('/lucky/number/{max}', name: 'app_lucky_number')]
    public function number(int $max): Response
    {
        $number = random_int(0, $max);

        return new Response(
            '<html><body>Lucky number: '.$number.'</body></html>'
        );
    }
}
```

- Se pueden añadir parámetros con valores por defecto (si no están presentes en la ruta, se asignará el valor por defecto). Se puede usar cualquiera de las siguientes formas:
  - `public function list(int $max = 10): Response`
  - `#[Route('lucky/number/{max?10}', name: 'app_lucky_number')]`
- Para forzar que el parámetro esté presente se usa ! antes del parámetro
  - `/lucky/number/{!max}`
- Se pueden añadir restricciones a los parámetros de dos formas
  - Con requirements:

```
#[Route('/lucky/number/{max}', name: 'app_lucky_number',
requirements: ['max' => '\d+'])]
```

  - Inline:

```
#[Route('/lucky/number/{max<\d+>}', name: 'app_lucky_number')]
```

- Por defecto, todos los métodos responden a cualquier método HTTP: GET, POST, PUT, HEAD, etc. Se puede especificar qué método HTTP va a servir un método PHP
- `#[Route('/lucky/number/{!max<\d+>}', name: 'app_lucky_number', methods:['GET', 'HEAD'])]`

Documentación:
https://symfony.com/doc/current/routing.html#parameters-validation

## 1.7. Plantillas Twig

Symfony utiliza, en la creación del proyecto por defecto, Twig.

- Twig es un motor de plantillas que mezcla código estático y dinámico.
- Estático con contenido HTML invariable y dinámico porque permite introducir mecanimos de herencia, paso de argumentos, instrucciones de control que permiten la comunicación con el controlador y la creación personalizada de una vista.
- Se puede encontrar una introducción de las posibilidades que ofrece Twig para diseñadores en https://twig.symfony.com/doc/3.x/templates.html. Las expresiones y estructuras más destacables son las siguientes:

```
{# Esto es un comentario #}

{{ var }} {# Muestra el contenido de una variable var #}

{% block body %}
   Crea un bloque llamado body que podrá ser sobrescrito en caso de herencia
{% endblock %}

{% extends 'base.html.twig' %}{# Indica que esta plantilla hereda de otra llamada base.html.twig  #}


{{ include('_sidebar.html.twig') }} {# Incluye el contenido de una plantilla dentro de otra #}



Se pueden crear y asignar valores a variables:
{% set foo = 'foo' %}
{% set foo = [1, 2] %}
{% set foo = {'foo': 'bar'} %}



Se pueden aplicar filtros a variables. Se usa la barra vertical o pipe | después de la variable y antes del filtro:


{{ 'my first car'|title }}

{# outputs 'My First Car' #}

Se pueden aplicar filtros a extractos de código con apply:
{% apply upper %}
    This text becomes uppercase
{% endapply %}

Todos los filtros disponibles están en la URL: https://twig.symfony.com/doc/3.x/filters/index.html
```

### Estructuras de control

For

```
<h1>Members</h1>
<ul>
    {% for user in users %}
        <li>{{ user.username|e }}</li>
    {% else %}
        <li><em>no user found</em></li>
    {% endfor %}
</ul>
```

Estructura If

```
{% if product.stock > 10 %}
   Available
{% elseif product.stock > 0 %}
   Only {{ product.stock }} left!
{% else %}
   Sold-out!
{% endif %}
```

# 1.8. AssetMapper

## AssetMapper

- Todos los archivos dentro de `assets/` se ponen a disposición pública y **se versionan**.
- Se puede hacer referencia al archivo `assets/images/product.jpg` en una plantilla Twig con `{{ asset('images/product.jpg') }}`.
- La URL final incluirá una versión hash, como `/assets/images/product-3c16d9220694c0e56d8648f25e6035e9.jpg`. Si se cambias el archivo, la parte de la versión de la URL también cambiará automáticamente.

En **assets/app.js** se importan:

- scripts: Con import seguido de la ruta al archivo .js: import './bootstrap.js';
- estilos: Con import seguido de la ruta al archivo .cs: import './styles.app.css';

```
import './bootstrap.js';
/*
 * Welcome to your app's main JavaScript file!
 *
 * This file will be included onto the page via the importmap() Twig function,
 * which should already be in your base.html.twig.
 */
import './styles/app.css';

console.log('This log comes from assets/app.js - welcome to AssetMapper! 🎉');
```

Este archivo se usa en **base.html.twig**. Gracias a la función de Twig `{{ importmap('app') }}`, el archivo `assets/app.js` se carga y se ejecuta en el navegador.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.
        {% block stylesheets %}
        {% endblock %}

        {% block javascripts %}
            {% block importmap %}{{ importmap('app') }}{% endblock %}
        {% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

### ¿Qué e un módulo de JavaScript?

Módulos en JavaScript: https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Modules

Un módulo es un **archivo de JavaScript que agrupa funciones, clases, variables que luego pueden ser exportadas y utilizadas en otras partes de nuestra aplicación**. Un módulo permite ocultar funcionalidad del mismo y solo exportar aquello para lo que ha sido implementado.

```
export { name, draw, reportArea, reportPerimeter };
```

La declaración `import`, seguida de una lista separada por comas de las características que se desean importar entre llaves, seguida de la palabra clave `from`, seguida de la ruta al archivo del módulo:

import { name, draw, reportArea, reportPerimeter } from "./modules/square.js";

Ejemplos en GitHub: https://github.com/mdn/js-examples/tree/main/module-examples/basic-modules/modules

### Añadir paquetes de terceros de javascript

En la raíz del proyecto tenemos **importmap.php**, ahí figuran los mapeados de nombres lógicos con su ubicación física. Un entrypoint es el fichero principal que cargará el navegador para comenzar por defecto.
En este caso vemos que cargará el fichero **/assets/app.js:**

importmap.php (Working Tree) M

```php
<?php

/**
 * Returns the importmap for this application.
 *
 * - "path" is a path inside the asset mapper system. Use the
 *     "debug:asset-map" command to see the full list of paths.
 *
 * - "entrypoint" (JavaScript only) set to true for any module that will
 *     be used as an "entrypoint" (and passed to the importmap() Twig function).
 *
 * The "importmap:require" command can be used to add new entries to this file.
 */
return [
    'app' => [
        'path' => './assets/app.js',
        'entrypoint' => true,
    ],
    '@hotwired/stimulus' => [
        'version' => '3.2.2',
    ],
    '@symfony/stimulus-bundle' => [
        'path' => './vendor/symfony/stimulus-bundle/assets/dist/loader.js',
    ],
    '@hotwired/turbo' => [
        'version' => '7.3.0',
    ],
```

Por defecto, la creación de un paquete webapp de Symfony ha añadido paquetes de Stimulus, Turbo y Hotwired. Los tres componentes trabajan en un conjunto de técnicas complementarias para acelerar los cambios de página con JavaScript y envíos de formularios, dividiendo páginas complejas en componentes, y transmitir actualizaciones de páginas parciales a través de WebSocket. Nosotros no los vamos a usar en el módulo y los podríamos eliminar con los comandos:

- php bin/console importmap:remove @hotwired/stimulus
- php bin/console importmap:remove @hotwired/turbo
- php bin/console importmap:remove @symfony/stimulus-bundle

Añadir paquete de JavaScript de Bootstrap:

Si añadimos el paquete de js de bootstrap usaremos el comando:

```
php bin/console importmap:require bootstrap
```

Se modificará el contenido de importmap.php para añadir las dependencias de forma transitiva.

importmap.php (Working Tree) M

```php
<?php

/**
 * Returns the importmap for this application.
 *
 * - "path" is a path inside the asset mapper system. Use the
 *     "debug:asset-map" command to see the full list of paths.
 *
 * - "entrypoint" (JavaScript only) set to true for any module that will
 *     be used as an "entrypoint" (and passed to the importmap() Twig function).
 *
 * The "importmap:require" command can be used to add new entries to this file.
 */
return [
    'app' => [
        'path' => './assets/app.js',
        'entrypoint' => true,
    ],
    '@hotwired/stimulus' => [
        'version' => '3.2.2',
    ],
    '@symfony/stimulus-bundle' => [
        'path' => './vendor/symfony/stimulus-bundle/assets/dist/loader.js',
    ],
    '@hotwired/turbo' => [
        'version' => '7.3.0',
    ],
+   'bootstrap' => [
+       'version' => '5.3.3',
+   ],
+   '@popperjs/core' => [
+       'version' => '2.11.8',
+   ],
+   'bootstrap/dist/css/bootstrap.min.css' => [
+       'version' => '5.3.3',
+       'type' => 'css',
+   ],
];

```

Todos los paquetes en `importmap.php` se descargan en el directorio `assets/vendor/`, que debe ser ignorado por git. Cuando se descarge de un repositorio central de Git, tendrá que ejecutarse el siguiente comando para descargar los archivos en otros equipos si algunos faltan:

```
php bin/console importmap:install
```

```
Después se podrá importar el  paquete de bootstrap en cualquier archivo de JavaScript con:

import { Alert } from 'bootstrap';
// ...
```

Añadir el paquete de CSS de Bootstrap

Por defecto, **app.js** añade el fichero ./styles/app.css a través de la instrucción import modificada especialmente por AssetMapper que lo traducirá a una etiqueta  <link> dentro del HTML.

Para añadir estilos de terceros, como de Bootstrap:

```
php bin/console importmap:require bootstrap/dist/css/bootstrap.min.css
```

```
(aunque ya se había añadido como dependencia transitiva previamente cuando instalamos JavaScript de Bootstrap)

Para poder usar los estilos de Bootstrap, los añadimos en app.js:
```

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

```
Más información sobre la configuración de Assets y hashing:
https://symfony.com/doc/current/reference/configuration/framework.html#assets
```

leer:
https://backbeat.tech/blog/asset-cache-busting-in-symfony-applications