












# Índice

- **GUÍA BÁSICA DE SYMFONY** 
  - **Inicio de Proyecto: Instalación de Symfony** 
    - **Pre-requisitos** 
    - **Comprobación del entorno**
    - **¿Qué hacer si los comandos no devuelven nada?** 
    - **Instalar Symfony Installer**
    - **¿Cómo levantar el servidor?**
    - **Personalizar cómo se ejecuta el servidor**
  - **Composer** 
    - **¿Qué es Composer?**
    - **Gestión de dependencias**
    - **Ver dependencias**
    - **Eliminar dependencias**
  - **Symfony Flex** 
    - **¿Qué es Symfony Flex?**
    - **Instalación de paquetes con Flex**
    - **Recetas automáticas**
  - **Doctrine** 
    - **¿Qué es Doctrine?**
    - **Instalar Doctrine**
    - **Configurar conexión**
    - **Crear entidad**
    - **Migraciones**
    - **Otros comandos útiles de Doctrine**
  - **Comandos Symfony** 
    - **¿Qué son los comandos Symfony?**
    - **Listar comandos disponibles**
    - **Ejemplos útiles**
  - **Estructura de Directorios** 
  - **Twig** 
    - **¿Qué es Twig?**
    - **Instalación**
    - **Sintaxis básica**
    - **Extender plantillas**
    - **Filtros comunes**
    - **Funciones útiles**
    - **Condicionales**
    - **Herencia de bloques**
    - **Ciclos**
    - **Depuración**
  - **Otros Componentes Clave** 
    - **Mensajes Flash**
    - **Enrutamiento YAML**
    - **Componente Validador**





---

# GUÍA BÁSICA DE SYMFONY

---

## Inicio de Proyecto: Instalación de Symfony

### Pre-requisitos

-  PHP 8.1 o superior
-  Composer instalado globalmente
-  Extensiones PHP necesarias: `pdo`, `mbstring`, `tokenizer`
-  Node.js y npm

---

### Comprobación del entorno

- `php -v`
- `composer -v`
- `node -v`
- `npm -v`

### ¿Qué hacer si los comandos no devuelven nada?

#### 1. Instalar PHP:

- En Windows:

1. Descargar el instalador desde [php.net](https://www.php.net).
2. Configurar la variable de entorno `PATH` para incluir el directorio donde se encuentra el ejecutable de PHP:
  - Clic derecho en "Este equipo" o "Mi PC" / **Propiedades**.
  - En **Configuración avanzada del sistema** / **Opciones avanzadas**.
  - Clic en el botón **Variables de entorno**.
  - En las variables del sistema, / `Path` / **Editar**.
  - Añadir el directorio donde está instalado PHP, por ejemplo: `C:\php`.
  - Guardar y reiniciar cualquier terminal abierta.

- En Linux:

```
> sudo apt update
> sudo apt install php-cli
```

#### 2. Instalar Composer:

- Descargar Composer desde [getcomposer.org](https://getcomposer.org):

```
> php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"

```

```
> php composer-setup.php  
> php -r "unlink('composer-setup.php');" 
```

- Mover el archivo ejecutable a un directorio accesible:

```
> sudo mv composer.phar /usr/local/bin/composer
```

### 3. Instalar Node.js y npm:

- Descargar Node.js desde [nodejs.org](https://nodejs.org):
  - Seleccionar la versión LTS.
- Alternativamente, usar un gestor de paquetes:
  - En Linux:

```
> sudo apt update  
> sudo apt install nodejs npm
```

## Instalar Symfony Installer

- Symfony Installer es una utilidad que facilita la creación de nuevos proyectos Symfony sin tener que configurar manualmente las dependencias iniciales. Se puede ejecutar desde cualquier ubicación en la terminal.

```
> composer global require symfony/installer
```

- **Crear nuevo proyecto**

```
> symfony new my_project_name --webapp  
> cd my_project_name
```

- **Iniciar servidor de desarrollo**

- El servidor de desarrollo es una herramienta integrada que permite ejecutar la aplicación en un entorno local sin necesidad de configurar un servidor completo como Apache o Nginx.
- **Ventajas:**
  - **Autodetección de cambios:** Reinicia automáticamente si detecta modificaciones en los archivos.
  - **Integración con Profiler:** Muestra información detallada sobre las solicitudes HTTP, consumo de memoria, tiempo de ejecución...etc.

## ¿Cómo levantar el servidor?

Ubicado en el directorio raíz del proyecto, ejecutar:

```
> symfony serve
```

En el navegador, acceder a la URL:

```
> http://127.0.0.1:8000
```

## Personalizar cómo se ejecuta el servidor

Especificar un puerto diferente:

```
> symfony serve --port=8080
```

Ejecutar en segundo plano (modo demonio):

```
> symfony serve -d
```

Detener el servidor si está ejecutándose en segundo plano:

```
> symfony server:stop
```

---

## Composer

### ¿Qué es Composer?

Composer es una herramienta para gestionar las dependencias de un proyecto PHP. Permite instalar, actualizar y mantener librerías de forma sencilla, garantizando que tu proyecto siempre utilice las versiones correctas de las dependencias necesarias.

### Gestión de dependencias

- Instalar un paquete:

```
> composer require <paquete>
```

Este comando descarga e instala el paquete especificado y actualiza el archivo `composer.json` con la nueva dependencia.

- Actualizar las dependencias:

```
> composer update
```

Esto revisa las versiones de las dependencias y actualiza a las más recientes permitidas según las restricciones de `composer.json`.

## Ver dependencias

- Mostrar todas las dependencias instaladas:

```
> composer show
```

- Ver detalles de un paquete específico:

```
> composer show <paquete>
```

- Archivo `composer.json` y `composer.lock`**
- `composer.json`: Define las dependencias del proyecto y las versiones permitidas.
- `composer.lock`: Registra las versiones exactas de las dependencias instaladas, garantizando consistencia en diferentes entornos.

## Eliminar dependencias

- Para eliminar un paquete y sus referencias en `composer.json`:

```
> composer remove <paquete>
```

- Comandos adicionales útiles**

- Optimizar el autoload:

```
> composer dump-autoload -o
```

Esto genera un archivo de carga optimizado para mejorar el rendimiento.

- Comprobar dependencias obsoletas:

```
> composer outdated
```

Lista las dependencias que tienen nuevas versiones disponibles.

---

Symfony Flex ⚡

¿Qué es Symfony Flex?

Symfony Flex es una extensión para Composer que simplifica la instalación y configuración de paquetes en proyectos Symfony. Facilita la personalización del entorno de desarrollo mediante recetas predefinidas.

## Instalación de paquetes con Flex

- Cuando se instala un paquete, por ejemplo:

```
> composer require annotations
```

**Symfony Flex** aplica una "receta" asociada a ese paquete.

## Recetas automáticas

- **¿Qué son las recetas?**

Son configuraciones automáticas que Symfony Flex implementa para facilitar el uso de paquetes.

- **Ejemplo de receta:** Al instalar `annotations`, Flex:
  - Habilita las anotaciones como método de configuración en controladores.
  - Modifica el archivo `config/packages/framework.yaml` para incluir soporte de anotaciones:

```
framework:
  annotation:
    enabled: true
```

**Actualizar recetas:** Para actualizar recetas a sus últimas versiones:

```
> composer recipes:update
```

**Listar recetas disponibles:**

```
> composer recipes
```

---

## Doctrine

### ¿Qué es Doctrine?

Doctrine es un ORM (Object-Relational Mapper) que permite interactuar con bases de datos utilizando objetos PHP en lugar de realizar consultas SQL manuales. Facilita la creación, lectura, actualización y eliminación de datos (CRUD).

## Instalar Doctrine

- Instalar Doctrine y sus dependencias:

```
> composer require symfony/orm-pack
```

Esto también configura automáticamente los archivos necesarios, como `config/packages/doctrine.yaml`.

## Configurar conexión

- Editar el archivo `.env` para definir la conexión a la base de datos. Por ejemplo:

```
DATABASE_URL="mysql://user:password@127.0.0.1:3306/my_database"
```

- La base de datos debe estar creada antes de continuar. Si aún no está creada, se puede crear a partir del `.env` usando el siguiente comando:

```
> php bin/console doctrine:database:create
```

## Crear entidad

- Una entidad representa una tabla en la base de datos. Al crearla, se definen campos y sus tipos como propiedades de la clase, lo que Doctrine utiliza para contruir la estructura de la tabla.
- Comando para generar una nueva entidad:

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

- La terminal dará instrucciones para definir los campos de la entidad.

## Migraciones

- Las migraciones son archivos generados por Doctrine que contienen las instrucciones SQL necesarias para sincronizar la base de datos con las entidades definidas en el código.
- Generar una migración para aplicar los cambios en la base de datos:

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

Este comando genera un archivo que describe los cambios detectados

- Aplica la migración generada:

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

- **Consultar datos**

Consultar datos implica interactuar con la base de datos para realizar operaciones como:

- Crear nuevos registros.
  - Leer datos existentes.
  - Actualizar registros.
  - Eliminar registros.
- **Uso del EntityManager:** El **EntityManager** es un componente central de Doctrine que se encarga de gestionar las entidades y sincronizarlas con la base de datos.
  - **Ejemplo práctico:**

```
// Crear una nueva entidad de tipo Product
$product = new Product();
$product->setName('Producto 1');
$product->setPrice(19.99);

// Obtener el EntityManager
$entityManager = $doctrine->getManager();

// Persistir la entidad en memoria
$entityManager->persist($product);

// Guardar los cambios en la base de datos
$entityManager->flush();
```

**Descripción del proceso:**

1. **Crear la entidad:** Se instancia una nueva entidad (**Product** en este caso) y se asignan valores a sus propiedades.
  2. **Persistir:** El método **persist** indica al EntityManager que esta entidad debe ser insertada o actualizada en la base de datos.
  3. **Flush:** El método **flush** guarda los cambios pendientes en la base de datos.
- **Leer datos existentes:** Puedes usar el repositorio asociado a una entidad para buscar registros:

```
// Obtener el repositorio de la entidad Product
$repository = $doctrine->getRepository(Product::class);

// Buscar un producto por su ID
$product = $repository->find(1);

// Buscar todos los productos
$products = $repository->findAll();
```



```
// Buscar productos con criterios específicos
$filteredProducts = $repository->findBy(['price' => 19.99]);
```

- **Actualizar registros:**

```
$product = $repository->find(1);
if ($product) {
    $product->setPrice(29.99);
    $entityManager->flush();
}
```

- **Eliminar registros:**

```
$product = $repository->find(1);
if ($product) {
    $entityManager->remove($product);
    $entityManager->flush();
}
```

## Otros comandos útiles de Doctrine

- Mostrar información del esquema:

```
php bin/console doctrine:schema:validate
```

- Crear la base de datos:

```
php bin/console doctrine:database:create
```

- Borrar la base de datos:

```
php bin/console doctrine:database:drop --force
```

---

## Comandos Symfony

### ¿Qué son los comandos Symfony?

Symfony proporciona una herramienta de línea de comandos (**bin/console**) que permite realizar tareas administrativas y de desarrollo de manera eficiente. Esta herramienta es una parte esencial del framework y

está diseñada para automatizar tareas comunes, como la generación de controladores, la limpieza de caché y la gestión de bases de datos.

## Listar comandos disponibles

- Para ver todos los comandos:

```
> php bin/console list
```

Esto mostrará una lista categorizada de comandos disponibles junto con una breve descripción de cada uno.

## Ejemplos útiles

### 1. Crear un controlador:

- Crea un nuevo controlador con un método inicial y una ruta configurada automáticamente:

```
> php bin/console make:controller NombreDelControlador
```

Esto generará un archivo en el directorio `src/Controller` con un método básico y una ruta asociada.

### 2. Limpiar la caché:

- Limpia la caché del proyecto para aplicar cambios recientes:

```
> php bin/console cache:clear
```

Útil después de cambios en configuraciones o en el entorno.

### 3. Crear entidades:

- Genera una nueva entidad interactivamente:

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

Te pedirá definir los campos, sus tipos y restricciones.

### 4. Generar migraciones:

- Crea un archivo de migración basado en los cambios realizados en las entidades:

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

Esto permite sincronizar la base de datos con las entidades de tu proyecto.

### 5. Ejecutar migraciones:

- Aplica las migraciones pendientes en la base de datos:

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

### 6. Ver rutas definidas:

- Muestra una lista de todas las rutas configuradas en tu proyecto:

```
> php bin/console debug:router
```

Incluye detalles como el nombre de la ruta, el método HTTP y el controlador asociado.

### 7. Validar el esquema de la base de datos:

- Comprueba si las entidades están sincronizadas con la base de datos:

```
> php bin/console doctrine:schema:validate
```

### 8. Generar contraseñas codificadas:

- Genera una contraseña codificada utilizando el sistema de codificación del proyecto:

```
> php bin/console security:encode-password
```

Ideal para pruebas o configuraciones iniciales de usuarios.

### 9. Ver servicios registrados:

- Lista todos los servicios disponibles en el contenedor de Symfony:

```
> php bin/console debug:container
```

Buscar servicios específicos añadiendo un término de búsqueda:

```
> php bin/console debug:container --search=doctrine
```

## 10. Iniciar el servidor de desarrollo:

- Arrancar un servidor local para pruebas:

```
> symfony serve
```

## Estructura de Directorios

```
proyecto/
├── src/                                # Código fuente de la aplicación (controladores,
entidades, servicios, etc.)
│   ├── Controller/                   # Controladores del proyecto
│   ├── Entity/                       # Entidades del proyecto (mapeo a tablas de la base de
datos)
│   ├── Repository/                   # Repositorios para consultas personalizadas
│   └── ...
├── templates/                         # Archivos Twig para las vistas
│   ├── base.html.twig                # Plantilla base
│   └── ...
├── config/                           # Archivos de configuración del proyecto
│   ├── packages/                     # Configuración de paquetes instalados
│   ├── routes.yaml                   # Rutas de la aplicación
│   └── ...
├── public/                           # Punto de entrada público de la aplicación (accesible
desde el navegador)
│   ├── index.php                     # Punto de entrada principal
│   └── ...
├── migrations/                       # Migraciones para el manejo de la base de datos
│   └── ...
├── var/                              # Archivos generados dinámicamente (caché, logs, etc.)
│   ├── cache/                        # Archivos de caché
│   ├── log/                          # Archivos de registro
│   └── ...
├── vendor/                           # Dependencias instaladas mediante Composer
├── .env                              # Variables de entorno del proyecto
└── composer.json                     # Archivo de configuración de dependencias
```

[!NOTE] Esta estructura refleja los elementos principales que se encuentran en un proyecto Symfony

## Twig

### ¿Qué es Twig?

Twig es el motor de plantillas oficial de Symfony. Permite generar vistas dinámicas utilizando una sintaxis simple y potente, optimizada para el desarrollo web. Twig se encarga de separar la lógica de presentación de la lógica del negocio en la aplicación.

## Instalación

- Twig se instala como una dependencia en Symfony:

```
> composer require twig
```

- Symfony configura automáticamente Twig como motor de plantillas y crea el directorio `templates/` para almacenar los archivos `.twig`.

## Sintaxis básica

```
{# templates/example.html.twig #}
<h1>{{ title }}</h1>
<ul>
    {% for item in items %}
        <li>{{ item }}</li>
    {% else %}
        <li>No hay elementos</li>
    {% endfor %}
</ul>
```

- `{{ variable }}`: Imprime el valor de una variable.
- `{% ... %}`: Delimita bloques de control como bucles y condicionales.
- `{# ... #}`: Define comentarios que no serán visibles en el HTML generado.

## Extender plantillas

Twig permite reutilizar partes de código mediante la extensión de plantillas base:

```
{# templates/base.html.twig #}
<html>
<body>
    {% block content %}{% endblock %}
</body>
</html>

{# templates/child.html.twig #}
{% extends 'base.html.twig' %}

{% block content %}
    <h1>Contenido personalizado</h1>
{% endblock %}
```

## Filtros comunes

Los filtros en Twig se aplican con el símbolo `|` para transformar datos antes de mostrarlos:

```
{{ 'hola mundo'|upper }} {# Resultado: HOLA MUNDO #}  
{{ price|number_format(2, '.', ',') }} {# Formatea números: 1,234.56 #}  
{{ '2024-01-01'|date('d/m/Y') }} {# Convierte fechas: 01/01/2024 #}
```

## Funciones útiles

Twig ofrece funciones integradas para operaciones comunes:

- **include**: Inserta otra plantilla.

```
{% include 'partials/menu.html.twig' %}
```

- **path**: Genera URLs basadas en rutas definidas.

```
<a href="{{ path('homepage') }}">Inicio</a>
```

- **asset**: Genera URLs para archivos estáticos.

```

```

## Condicionales

Controla el flujo en las plantillas con **if** y **elseif**:

```
{% if is_logged_in %}  
    <p>Bienvenido, {{ user.name }}!</p>  
{% elseif show_login %}  
    <p>Por favor, inicia sesión.</p>  
{% else %}  
    <p>Acceso restringido.</p>  
{% endif %}
```

## Herencia de bloques

Se pueden extender y modificar bloques específicos en plantillas secundarias:

```
{# templates/base.html.twig #}  
<html>  
<head>  
    {% block title %}Título por defecto{% endblock %}  
</head>
```

```
<body>
    {% block content %}{% endblock %}
</body>
</html>

{# templates/child.html.twig #}
{% extends 'base.html.twig' %}

{% block title %}Título personalizado{% endblock %}

{% block content %}
    <p>Este es el contenido específico de esta página.</p>
{% endblock %}
```

## Ciclos

Los bucles son fundamentales para iterar sobre datos en Twig:

```
<ul>
    {% for product in products %}
        <li>{{ product.name }} - {{ product.price|number_format(2) }}€</li>
    {% endfor %}
</ul>
```

- Puedes usar la variable especial `loop` dentro de los bucles para obtener detalles:

```
{% for product in products %}
    <p>Índice: {{ loop.index }} - Nombre: {{ product.name }}</p>
{% endfor %}
```

## Depuración

Para inspeccionar datos en las plantillas, se utiliza la función `dump` (requiere que la depuración esté activada):

```
{{ dump(variable) }}
```

Esto imprimirá el contenido de `variable` en la barra de depuración o directamente en la plantilla (si la barra no está activa).

- **Activar la depuración:**

1. El entorno debe ser `dev`:

```
symfony serve --env=dev
```

## 2. Configurar las variables en el archivo `.env`:

```
APP_ENV=dev
APP_DEBUG=1
```

## 3. Verificar que la barra de depuración esté visible al pie de las páginas.

### • Ventajas de Twig

- **Rendimiento:** Twig genera plantillas compiladas en PHP, mejorando el rendimiento.
- **Seguridad:** Escapa automáticamente las variables para prevenir ataques de inyección de código.
- **Flexibilidad:** Soporta extensiones personalizadas para agregar filtros, funciones y pruebas según las necesidades del proyecto.

---

## Otros Componentes Clave

### Mensajes Flash

- **¿Qué son los mensajes flash?** Los mensajes flash son una forma temporal de pasar mensajes entre las peticiones HTTP. Se utilizan para mostrar notificaciones como "Operación exitosa" o "Error en el formulario" tras realizar una acción.
- **Enviar un mensaje flash desde un controlador:**

```
$this->addFlash('success', 'Mensaje guardado con éxito!');
```

- `success`: Es el tipo de mensaje. Puede ser `error`, `warning`, `info`, etc.
- `'Mensaje guardado con éxito!'`: Es el contenido del mensaje.

- **Mostrar mensajes flash en una plantilla Twig:**

```
{% for message in app.flashes('success') %}
    <div class="alert alert-success">{{ message }}</div>
{% endfor %}
```

- `app.flashes('success')`: Devuelve todos los mensajes flash del tipo `success`.

### Enrutamiento YAML

- **¿Qué es el enrutamiento?**

El enrutamiento define cómo se asocian las URLs a los controladores. En Symfony, esto se configura generalmente en `config/routes.yaml`.

- **Ejemplo básico de configuración de una ruta:**



```
home:
  path: /
  controller: App\Controller\HomeController::index
```

- **home**: Nombre único de la ruta.
  - **path**: URL asociada a la ruta (/ es la raíz del sitio).
  - **controller**: Método del controlador que se ejecutará cuando se acceda a esta URL.
- **Generar rutas dinámicas**: Pasar parámetros en la URL:

```
user_profile:
  path: /user/{id}
  controller: App\Controller\UserController::profile
```

- **{id}**: Es un parámetro dinámico que se pasará al método del controlador.

## Componente Validador

- **¿Qué es el componente validador?**

Es una herramienta de Symfony que permite validar datos, como entradas de formularios o propiedades de objetos.

- **Ejemplo de validaciones en una entidad:**

```
use Symfony\Component\Validator\Constraints as Assert;

class User
{
    #[Assert\NotBlank]
    #[Assert\Length(min: 3, max: 50)]
    private $name;

    #[Assert\Email]
    private $email;

    #[Assert\GreaterThan(18)]
    private $age;
}
```

- **#[Assert\NotBlank]**: Asegura que el campo no esté vacío.
  - **#[Assert\Length(min: 3, max: 50)]**: Restringe la longitud del texto.
  - **#[Assert\Email]**: Verifica que el valor tenga un formato de email válido.
  - **#[Assert\GreaterThan(18)]**: Requiere que el valor sea mayor que 18.
- **Validar manualmente un objeto:**

```
use Symfony\Component\Validator\Validation;
use Symfony\Component\Validator\Constraints as Assert;

$validator = Validation::createValidator();
$violations = $validator->validate($value, [
    new Assert\NotBlank(),
    new Assert\Length(['min' => 3]),
]);

if (count($violations) > 0) {
    foreach ($violations as $violation) {
        echo $violation->getMessage();
    }
}
```

- **Validation::createValidator()**: Crea un validador.
- **validate**: Aplica las restricciones definidas a un valor.