

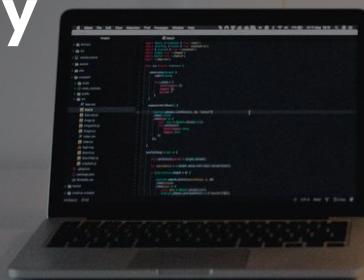


Symfony

PHP



Symfony



El objetivo de este fastbook es que aprendas a dar tus primeros pasos con Symfony, un framework que se utiliza en una amplia variedad de proyectos web, desde sitios web simples y aplicaciones móviles hasta sistemas empresariales complejos. Es una muy buena opción para desarrolladores de PHP que buscan una base sólida y herramientas eficientes.

Autor: Jesús Donoso

-  **Introducción**
-  **Proyectos en Symfony**
-  **Diseño MVC (modelo-vista-controlador)**
-  **Enrutamiento y controladores**
-  **Plantillas en Symfony**
-  **Acceso de bases de datos con Doctrine**
-  **Formularios**
-  **Conclusiones y recursos de interés**

Introducción



Como ya sabes un framework es un conjunto de herramientas y componentes predefinidos que permiten a los desarrolladores construir aplicaciones web de manera más eficiente al proporcionar una estructura organizada y reutilizable para el desarrollo.

Symfony es un framework de desarrollo web de código abierto diseñado para facilitar la creación de aplicaciones web robustas, escalables y de alto rendimiento en PHP.

¿Sabías que...? Symfony fue creado en 2005 por Fabien Potencier, un desarrollador de software francés. Potencier estaba buscando una forma más eficiente de desarrollar aplicaciones web y decidió crear su propio framework PHP.

Aquí te presento algunas de las **características destacables de Symfony**:

Arquitectura MVC

Symfony sigue el patrón de arquitectura modelo-vista-controlador (MVC), lo que significa que separa la lógica de la aplicación en tres componentes principales: el modelo, que representa los datos y la lógica de negocio; la vista, que maneja la presentación; y el controlador que se encarga de manejar las solicitudes y las respuestas del usuario. Esto ayuda a mantener el código organizado y facilita la colaboración en equipos de desarrollo.

Componentes reutilizables

Symfony se basa en un conjunto de componentes reutilizables que pueden ser utilizados de forma independiente. Estos componentes son bibliotecas PHP que ofrecen funcionalidades específicas, como el manejo de formularios, la autenticación, el enrutamiento, el acceso a bases de datos, etc. Esto facilita la construcción de aplicaciones personalizadas y permite a los desarrolladores aprovechar las partes necesarias del framework.

Flexibilidad y escalabilidad

Symfony es altamente personalizable y permite a los desarrolladores elegir las herramientas y componentes que mejor se adapten a sus necesidades. Además, es adecuado para proyectos de cualquier tamaño, desde pequeñas aplicaciones web hasta grandes sistemas empresariales.

Rendimiento

También ofrece un alto rendimiento. A lo largo de las versiones, se han realizado mejoras significativas en la velocidad de ejecución y la eficiencia de los recursos, lo que lo hace adecuado para aplicaciones web de alto tráfico.

Comunidad activa

Symfony, como no podría ser de otra manera, cuenta con una comunidad activa de desarrolladores, lo que significa que hay una gran cantidad de recursos, documentación, complementos y paquetes disponibles. Participar en estas comunidades te ayudará a mejorar tu aprendizaje y el desarrollo de aplicaciones con este framework.

Seguridad

Symfony se preocupa por la seguridad y proporciona características y mejores prácticas para proteger las aplicaciones web contra amenazas comunes, como los ataques XSS (*Cross-Site Scripting*) y CSRF (*Cross-Site Request Forgery*).

Proyectos en Symfony



En este apartado, vamos a estudiar cómo crear un proyecto con esta herramienta paso a paso. ¡Empezamos!

Instalación de Symfony CLI

Symfony CLI es una herramienta adicional que es útil para crear y administrar proyectos Symfony de manera más sencilla, por eso considero que es una opción muy recomendada para principiantes.

Se instala a través de Composer, un administrador de dependencias para proyectos PHP. Los requisitos previos a la instalación son los siguientes:

Tener [PHP](#) previamente instalado en el sistema.

Instalar Composer (puedes descargarte e instalarlo desde su [web oficial de Composer](#)).

Para realizar la descarga e instalación de Symfony CLI te recomiendo también que sigas los pasos que se detallan en la [web oficial de Symfony](#). Igualmente se puede crear un proyecto de Symfony sin el cliente, a través de Composer.

Creación de un proyecto de Symfony

El cliente de Symfony se utiliza mediante un terminal en Linux o MacOs y mediante el CMD para Windows. Antes de crear un nuevo proyecto, se puede comprobar que el cliente está instalado correctamente mediante el comando que se puede observar a continuación:

```
symfony check:requirements
```

En caso de existir errores de la configuración de PHP se mostrarían en el terminal tras la ejecución del comando anterior. Es importante corregir los errores antes de continuar.

Para crear un proyecto en Symfony haciendo uso de Symfony CLI utiliza el siguiente comando:

```
symfony new nombre_del_proyecto --full
```

Y si quieres crear un proyecto en Symfony haciendo uso de Composer utiliza este otro comando:

```
composer create-project symfony/skeleton nombre_del_proyecto
```

Ambos comandos resultan en la creación de una estructura básica de un proyecto Symfony en un directorio llamado 'nombre_del_proyecto'.

iToma nota! Composer está preconfigurado con un repositorio Packagist. De forma predeterminada, solo se pueden instalar los paquetes listados en este repositorio. Los paquetes instalados para el proyecto se definen en el archivo 'composer.json', dentro de la raíz del proyecto en las secciones 'require' y 'requiere-dev'.

Para ejecutar un servidor web de desarrollo incorporado y probar el proyecto de Symfony creado es necesario navegar hasta el directorio y ejecutar comando start.

```
cd nombre_de_tu_proyecto  
symfony server:start --port=8000
```

Symfony, por defecto, utiliza el puerto 8000, pero si se necesita se puede modificar y utilizar otro. Esto inicia el servidor web de desarrollo y proporciona la URL en la que se puede acceder a la aplicación Symfony en el navegador.

Al abrir un navegador y acceder a la URL proporcionada por el servidor web de desarrollo (generalmente, `http://127.0.0.1:8000`) nos encontramos con la página de inicio de Symfony.

Estructura de un proyecto con Symfony

Symfony sigue una estructura de proyecto bien definida que ayuda a organizar los archivos y directorios de manera efectiva para facilitar el desarrollo y el mantenimiento de aplicaciones.

A continuación, se detallan algunos de los directorios y ficheros más comunes en un proyecto de Symfony:

- bin**: contiene el archivo `console`. Este archivo es la entrada de línea de comandos para ejecutar tareas relacionadas con Symfony, como generar controladores, ejecutar migraciones de base de datos, etc.
- config**: se almacena la configuración de la aplicación, incluidos los archivos de configuración de rutas, servicios y paquetes.
- public**: contiene los archivos públicos de la aplicación, como imágenes, hojas de estilo y JavaScript. El archivo `index.php` es el punto de entrada principal de la aplicación.
- src**: en este directorio se encuentra almacenada la mayor parte del código de la aplicación. Contiene subdirectorios como Controller, Entity o Form entre otros donde se pueden organizar las clases.

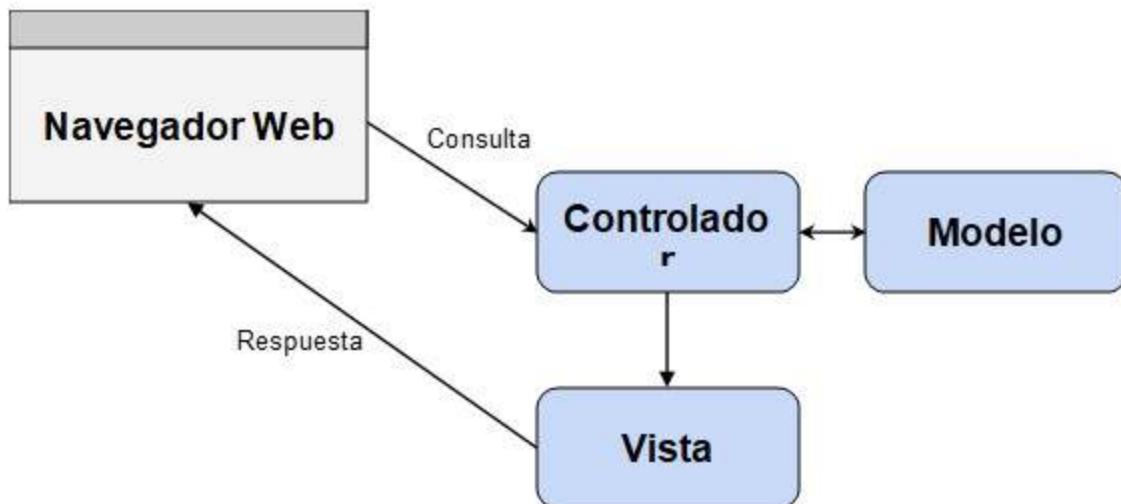
- templates**: aquí localizamos las plantillas Twig utilizadas para renderizar las vistas de la aplicación.
- translations**: se pueden almacenar archivos de traducción en este directorio para admitir múltiples idiomas en la aplicación.
- var**: este directorio contiene archivos generados dinámicamente por Symfony, como caché y registros de eventos.
- vendor**: es donde Composer instala las dependencias del proyecto.
- .env** y **.env.local**: estos archivos contienen variables de entorno que configuran la aplicación. Y las variables de entorno específicas del entorno local se pueden configurar en **.env.local**.
- phpunit.xml.dist**: la configuración para PHPUnit, una herramienta para realizar pruebas unitarias en Symfony.
- README.md**: un archivo de documentación que generalmente contiene información sobre cómo configurar y ejecutar el proyecto.

Los directorios y ficheros vistos son los más típicos en un proyecto de Symfony, sin embargo, pueden variar según la versión específica de Symfony y las personalizaciones que se deseen realizar en el proyecto.

Diseño MVC (modelo-vista-controlador)



Symfony sigue **el patrón de diseño modelo-vista-controlador** (MVC) para organizar y estructurar aplicaciones web. Este patrón separa la lógica de la aplicación en esos tres componentes principales: el modelo, la vista y el controlador.



Estudiaremos, a continuación, cada uno de ellos.

Modelo

El modelo en Symfony representa la capa de acceso a datos y la lógica de negocio de la aplicación.

Esto incluye la gestión de la base de datos, la recuperación y el almacenamiento de datos, y las reglas de negocio.

Symfony utiliza el componente Doctrine ORM (*Object Relational Mapping*) para **administrar la interacción con la base de datos**. Las entidades Doctrine representan las tablas de la base de datos y definen cómo se relacionan entre sí. Se pueden definir estas entidades en el directorio `src/Entity`.

Otro elemento importante del modelo son los repositorios. Estos son clases que se utilizan para consultar y manipular datos en la base de datos. Symfony proporciona un mecanismo fácil para crear repositorios personalizados para las entidades.

Vista

Las vistas en Symfony se encargan de mostrar los datos del modelo y de proporcionar una interfaz amigable para el usuario final.

Se puede utilizar el motor de plantillas Twig para crear y renderizar las vistas. Las plantillas Twig son archivos HTML enriquecidos con código Twig que se utilizan para definir la estructura y el diseño de las páginas web. Podemos encontrar estas plantillas en el directorio *templates* del proyecto.

Controlador

El controlador en Symfony maneja las solicitudes de los usuarios y actúa como intermediario entre el modelo y la vista. **Los controladores son clases PHP que contienen métodos que responden a las solicitudes web.** Estos métodos se llaman acciones y contienen las anotaciones de enrutamiento que asocian una URL a una acción específica.

El controlador recopila datos del modelo, decide qué plantilla Twig se debe usar para la respuesta y devuelve una respuesta HTTP. Symfony utiliza un enrutador para mapear las URL a controladores y acciones específicas. El enrutador se configura en el directorio *config/routes*.

A continuación, quiero que veas el proceso de manejo, por parte de Symfony, de las solicitudes HTTP para la generación de una respuesta:

1

Solicitud HTTP del cliente. El proceso comienza cuando un cliente (por ejemplo, un navegador web) realiza una solicitud HTTP a la aplicación Symfony.

2

Enrutamiento (Routing). Symfony utiliza un componente de enrutamiento para mapear la URL de la solicitud a un controlador específico. El archivo de rutas (*config/routes.yaml*) define cómo se deben manejar las URL y qué controlador y acción deben responder a cada ruta.

3

Controlador (Controller). Una vez que el enrutador ha determinado qué controlador y acción deben manejar la solicitud se procede a la ejecución del controlador correspondiente.

4

Modelo (Model). El controlador es una clase PHP que contiene uno o más métodos llamados acciones. Las acciones son responsables de procesar la solicitud y preparar una respuesta.

5

Plantilla (View). Dentro de la acción del controlador se puede interactuar con el modelo de la aplicación para realizar operaciones con base de datos.

6

Respuesta HTTP. La plantilla Twig genera una respuesta HTML que incluye el contenido de la página web que se envía de vuelta al cliente.

7

Cliente recibe la respuesta. El navegador web o el cliente que realizó la solicitud recibe la respuesta y muestra la respuesta en página web o la procesa según corresponda.

Enrutamiento y controladores



En Symfony, el enrutamiento y los controladores son **dos conceptos estrechamente relacionados que trabajan juntos** para manejar las solicitudes HTTP y generar respuestas. Vamos a entrar a detallarlos en los siguientes subapartados.

Enrutamiento

En Symfony las rutas se definen en **un archivo de configuración**. Este archivo suele ser `routes.yaml` aunque se pueden usar otros formatos como XML o PHP. Cada ruta se identifica por un nombre único y se define con un patrón de URL, una referencia al controlador y la acción que deben manejarla.

```
app_homepage:  
    path: /inicio  
    controller: App\Controller\HomeController::index
```

En este ejemplo se puede observar que el patrón de URLs "/inicio", la referencia al controlado es 'App\Homepage\HomeController' y la acción es el método 'index' del controlador 'HomeController'. En este caso, cuando un usuario acceda a la URL '/inicio' desde un navegador web se ejecuta el método 'index' del controlador 'HomeController'.

Una ventaja clave del enrutamiento en Symfony es la capacidad de generar URLs de manera dinámica en las vistas y controladores, utilizando el nombre de la ruta y los parámetros necesarios. Symfony se encarga de construir la URL correcta según la definición de la ruta.

A continuación, te muestro un ejemplo de la generación de una URL en una plantilla Twig.

```
<a href="{{ path('app_homepage') }}">Ir a la página de inicio</a>
```

Se pueden definir rutas con parámetros variables en las URLs. Estos parámetros se indican dentro de llaves **en el patrón de la URL** y se pasan al controlador correspondiente. Te comparto este ejemplo para que lo entiendas mejor:

```
app_blog_show:  
    path: /blog/{parametro}  
    controller: App\Controller\BlogController::show
```

En este caso, `{slug}` es un parámetro variable que se pasa al controlador cuando la URL coincide con el patrón especificado en el `path`. El controlador tiene la lógica necesaria para procesar dicho parámetro y actuar en consecuencia.

Symfony permite **agregar restricciones y requisitos** a las rutas para asegurarse de que cumplan ciertos criterios. Por ejemplo, se puede especificar que un parámetro en la URL debe ser un número o seguir un patrón de expresión regular. Un ejemplo de restricción podría ser este:

```
app_product:  
    path: /producto/{id}  
    controller: App\Controller\ProductController::show  
    requirements:  
        id: \d+ # Requiere que "id" sea un número.
```

Controladores

Los controladores en Symfony son clases de PHP que se encargan de manejar las solicitudes HTTP y generar respuestas. Son una parte fundamental en la arquitectura del framework Symfony y se utilizan para implementar la lógica de las aplicaciones web.

Las clases PHP se almacenan, generalmente, en el directorio `src/Controller` del proyecto Symfony. **Un controlador puede tener varias acciones** correspondientes a métodos públicos en la clase.

Ahora, observa este ejemplo básico de un controlador:

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

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends AbstractController
{
    public function index()
    {
        return $this->render('default/index.html.twig');
    }
}
```

Las acciones del controlador son métodos que se encargan de manejar solicitudes específicas. Estas acciones pueden **realizar diversas tareas**, como recuperar datos de una base de datos, procesar formularios y generar respuestas. En el ejemplo anterior, la acción *index* devuelve una respuesta renderizando una plantilla Twig.

Los controladores pueden **devolver respuestas** que se generan a partir de plantillas Twig, archivos JSON, XML, o cualquier otro formato de respuesta requerido. En el ejemplo, *\$this->render()* se utiliza para renderizar una plantilla Twig y generar una respuesta HTML.

Por otro lado, los controladores en Symfony pueden **aprovechar la inyección de dependencias para acceder a servicios y componentes necesarios en la aplicación**. Pues bien, Symfony proporciona el mecanismo de inyección de dependencias para facilitar el acceso a recursos como bases de datos, sesiones, correo electrónico, etc.

Esto se logra a través de la inyección de dependencias automática en las acciones del controlador. En el ejemplo que te comparto, puedes ver la inyección del servicio *entityManager* para acceder a la base de datos.

```
public function show(EntityManagerInterface $entityManager)
{
    // Utilizar EntityManager para interactuar con la base de datos.
}
```

Es importante recordar que, **cada acción del controlador** se asocia a una ruta específica que se define en el enrutamiento de Symfony. Las rutas se configuran en un archivo de enrutamiento, generalmente en *config/routes.yaml*, y se vinculan a los métodos de controlador correspondientes.

Symfony permite **definir las rutas directamente** en las clases de controlador mediante anotaciones. Se puede utilizar la anotación `@Route` para especificar la ruta y el método HTTP asociado con una acción de controlador.

```
/**
 * @Route("/pagina", name="mi_pagina")
 */
public function myPage()
{
    // Lógica de la acción de controlador.
}
```

Como hemos visto, los controladores en Symfony son componentes clave para el manejo de solicitudes y la generación de respuestas en las aplicaciones. Son útiles para organizar la lógica de una aplicación de manera estructurada aprovechando el sistema de enrutamiento de Symfony para vincular rutas a acciones específicas.

Vamos a dar un paso más, veamos cómo trabajar con plantillas en Symfony.

Plantillas en Symfony



La gestión de plantillas en Symfony se refiere a cómo el framework maneja y renderiza las vistas de la aplicación. Symfony utiliza el motor de plantillas Twig por defecto; una herramienta potente y flexible para generar contenido HTML, XML, JSON y otros formatos.

Symfony contiene Twig preinstalado.

En caso de que no estuviera, se puede añadir como dependencia de Composer en el proyecto Symfony.

```
composer require twig
```

Symfony, por tanto, configura Twig de forma predeterminada, es decir, no se necesita realizar **ninguna configuración adicional**. Las plantillas en Symfony se almacenan en el directorio *templates* de la aplicación. Es posible crear plantillas Twig utilizando la extensión *.html.twig*. Lo vas a entender mejor, con un ejemplo que muestra cómo generar una plantilla.

```
{# templates/mi_plantilla.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ content }}</h1>
</body>
</html>
```

En los controladores de Symfony, se pueden renderizar plantillas Twig utilizando el método `$this->render()`. Esto devuelve una respuesta HTTP con el contenido de la plantilla. En el ejemplo que comparto a continuación fíjate en cómo se renderiza la plantilla que se ha creado previamente.

```
public function myAction()
{
    return $this->render('mi_plantilla.html.twig', [
        'title' => 'Título de la Página',
        'content' => 'Contenido de la Página',
    ]);
}
```

Se puede observar cómo la función `render()` tiene dos parámetros que se pasan como argumento. El primero de ellos es el nombre de la plantilla Twig que se ha creado. El segundo es un **array asociativo** en el que se incluyen los datos. En la plantilla Twig se puede acceder a estos datos haciendo uso de la sintaxis `{{ variable }}`.

Existe la posibilidad de **incluir una plantilla dentro de otra usando la directiva `{%include%}`**. Esto es útil para reutilizar código común en múltiples plantillas.

```
{% include 'header.html.twig' %}  
<h1>{{ content }}</h1>  
{% include 'footer.html.twig' %}
```

Twig admite la herencia de plantillas. Se puede definir una plantilla base con secciones que pueden ser sobreescritas en las plantillas hijas. Esto facilita la creación de un diseño consistente en la aplicación web.

En los fragmentos de código, que muestro a continuación, puedes observar la implementación de una plantilla base y una plantilla hija.

```
{# PLANTILLA PADRE #}  
<!DOCTYPE html>  
<html>  
<head>  
    <title>{% block title %}Título por Defecto{% endblock %}</title>  
</head>  
<body>  
    {% block content %}{% endblock %}  
</body>  
</html>
```

```
{# PLANTILLA HIJA #}
{% extends 'base.html.twig' %}

{% block title %}Título Personalizado{% endblock %}

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

Y quédate con esto: **Twig ofrece una amplia variedad de filtros y funciones integrados** que pueden ser utilizados para manipular datos en las plantillas.

```
{{ variable|filter }}
{{ function(variable) }}
```

En definitiva, la gestión de plantillas en Symfony, a través de Twig, nos ayuda a construir vistas de manera sencilla y eficiente. También puedes organizar y personalizar el contenido de manera efectiva, lo que facilita la creación de interfaces de usuario atractivas y funcionales en las aplicaciones Symfony en las que estás trabajando.

Acceso de bases de datos con Doctrine



Qualentum Lab

En Symfony, Doctrine es el ORM (mapeo objeto-relacional) predeterminado que se utiliza para **interactuar con bases de datos relacionales**. Doctrine facilita la interacción con la base de datos a través de consultas orientadas a objetos.

Doctrine ORM viene preinstalado en Symfony 4 y versiones posteriores. Si se trabaja con una versión anterior se debe instalar como una dependencia de Composer utilizando este comando:

```
composer require doctrine
```

En Symfony la configuración de la conexión a la base de datos se realiza en el archivo `config/packages/doctrine.yaml`. Se pueden configurar **diferentes conexiones para entornos de desarrollo, prueba y producción**. Te planteo un ejemplo en el que se utiliza MySQL como motor de base de datos. Igualmente se pueden configurar otros motores compatibles.

```

doctrine:
    dbal:
        driver: 'pdo_mysql'
        server_version: '5.7'
        host: '%env(resolve:DATABASE_URL)%'
        charset: utf8mb4
        default_table_options:
            charset: utf8mb4
            collate: utf8mb4_unicode_ci
    orm:
        auto_generate_proxy_classes: true
        naming_strategy:
            doctrine.orm.naming_strategy.underscore_number_aware
            auto_mapping: true

```

Es importante recordar que en una base de datos relacional como MySQL uno de los elementos fundamentales son las tablas. En Symfony las entidades son clases de PHP que representan las tablas en la base de datos y cada propiedad de una entidad se corresponde con una columna en la tabla. Y es en el directorio `src/Entity` del proyecto donde podemos definir las entidades.

Echa un vistazo a este ejemplo y descubre cómo realizar la configuración de este directorio.

```

// src/Entity/Producto.php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="productos")
 */
class Producto
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $nombre;

    // Getters y Setters
}

```

Para generar tablas en la base de datos basadas en las definiciones de entidades especificadas en el directorio se puede hacer uso del siguiente comando por consola:

```
php bin/console doctrine:schema:update --force
```

Doctrine proporciona repositorios para cada entidad que permiten **realizar consultas y recuperar datos de la base de datos de manera orientada a objetos**. En el siguiente ejemplo, se recupera el repositorio de la clase Producto y se realiza la búsqueda del elemento de la tabla Producto que tenga id 1.

```
$repository = $this->getDoctrine()->getRepository(Producto::class);
$producto = $repository->find(1);
```

Igualmente, podemos hacer uso de SQL para realizar consultas más complejas a través de objetos (observa el ejemplo).

```
$em = $this->getDoctrine()->getManager();
$query = $em->createQuery(
    'SELECT p FROM App\Entity\Producto p WHERE p.precio > :precio'
)->setParameter('precio', 50);

$productos = $query->getResult();
```

Doctrine también permite persistir y actualizar entidades en la base de datos. En este ejemplo que te comarto, verás el uso de dos funciones importantes: persist() y flush() para guardar el registro.

```
$em = $this->getDoctrine()->getManager();
$producto = new Producto();
$producto->setNombre('Nuevo Producto');
$em->persist($producto);
$em->flush();
```

Otra operación que se puede realizar es **el borrado de registros**. Para ello disponemos de dos funciones:

- *remove()* para eliminar el registro;
- y *flush()* para guardar dicho borrado.

```
$em = $this->getDoctrine()->getManager();
$producto = $em->getRepository(Producto::class)->find(1);
$em->remove($producto);
$em->flush();
```

Ahora que ya hemos visto cómo gestionar nuestras bases de datos con Doctrine, solo nos falta desgranar un proceso más: la interacción con los usuarios, y para ello tenemos que estudiar cómo trabajar los formularios con Symfony.

Formularios



Qualentum Lab

Los formularios son un elemento esencial para interactuar con los usuarios en aplicaciones.

Symfony proporciona un componente de formularios que permite crear estos elementos y gestionarlos de manera eficiente.

Existen varias maneras de crear formularios en Symfony. Una forma habitual es mediante la consola de comandos haciendo uso del siguiente comando.

```
php bin/console make:form
```

Tras la ejecución del comando **se abre un asistente interactivo** que ayuda al desarrollador en la creación de un formulario y la definición de los campos que se van a incluir. Como resultado se genera una clase formulario en el directorio *src/Form*.

Se pueden **definir los campos de un formulario** en la clase de formulario que se ha generado. Cada campo representa una propiedad de la clase. En siguiente ejemplo, vemos que, en el parámetro `$builder`, se añaden los campos del formularios con el método `add` indicando, como parámetros, el nombre del campo y el tipo.

```
Debes crear un controlador que se encargue de mostrar y procesar el formulario. // src/Form/NombreDelFormularioType.php
```

```
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;

class NombreDelFormularioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('campo1', TextType::class, [
                'label' => 'Campo 1',
            ])
            ->add('campo2', TextType::class, [
                'label' => 'Campo 2',
            ])
            ->add('guardar', SubmitType::class, [
                'label' => 'Guardar',
            ]);
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            // Opciones por defecto.
        ]);
    }
}
```

Es importante también crear un controlador que se encargue de mostrar y procesar el formulario.

```
// src/Controller/FormularioController.php

use App\Form\NombreDelFormularioType;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class FormularioController extends AbstractController
{
    public function crearFormulario(Request $request)
    {
        $formulario = $this->createForm(NombreDelFormularioType::class);
        $formulario->handleRequest($request);

        if ($formulario->isSubmitted() && $formulario->isValid()) {
            // Procesar los datos del formulario, por ejemplo, guardar en
            la base de datos.
        }

        return $this->render('formulario.html.twig', [
            'formulario' => $formulario->createView(),
        ]);
    }
}
```

En este ejemplo que acabas de ver, **la creación de un formulario (\$formulario)** se realiza a través del método *CreateForm*. Este método es utilizado dentro de la función 'crearFormulario' de la clase 'FormularioController'. Adicionalmente, se comprueba la validez del formulario con el método *isValid()*.

¡Seguimos!

Se puede desarrollar una plantilla Twig para renderizar el formulario. En la plantilla, se puede utilizar el objeto formulario generado por el controlador para mostrar los campos del formulario.

```
{# templates/formulario.html.twig #}
<form method="post">
    {{ form(formulario) }}
</form>
```

Es importante remarcar que es necesario la definición de la ruta dentro del archivo de enrutamiento *routes.yaml* para acceder al controlador que contiene la creación del formulario.

```
formulario:
    path: /formulario
    controller: App\Controller\FormularioController::crearFormulario
```

Otro dato importante es que Symfony maneja la validación de los datos del formulario automáticamente según las reglas definidas en la clase del formulario. Se le pueden agregar restricciones de validación a los campos del formulario. En el ejemplo que muestro a continuación se observa la adición de la clave *constraints* dentro del array de atributos del formulario.

```
// src/Form/NombreDelFormularioType.php

use Symfony\Component\Validator\Constraints as Assert;

public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('campo1', TextType::class, [
            'label' => 'Campo 1',
            'constraints' => [
                new Assert\NotBlank(),
                new Assert\Length(['min' => 3]),
            ],
        ])
        // ...
    ;
}
```

Dentro de *constraints* se añaden **las restricciones** que se consideren necesarias. En nuestro ejemplo las restricciones son que el campo nunca puede estar vacío y deba tener una longitud mínima de 3 caracteres.

Por último, me gustaría destacar Symfony proporciona una amplia gama de opciones y funcionalidades para trabajar con formularios, incluyendo la gestión de envío de formularios, protección CSRF, manejo de archivos y mucho más.

Conclusiones y recursos de interés



Qualentum Lab

Como ya supones o quizás lo has investigado por tu cuenta, uno de los motivos de la popularidad de Symfony, en el desarrollo de aplicaciones con PHP, es que nos permite escalar nuestros proyectos para que vayan creciendo, aunque contengan mucho código. Además, gracias a su estructura MVC, puede cubrir progresivamente las necesidades de negocio. Su gran variedad de plantillas también lo elevan en el pódium de frameworks flexibles y eficientes.

No voy a repetir de nuevo todas sus fortalezas, pero sí quiero insistir en que entres con Symfony por tu cuenta para mejorar tu aprendizaje.



Te comarto aquí los enlaces para comenzar a trabajar con esta herramienta:
[instalación de Symfony](#), [documentación oficial de Composer](#) e [instalación Symfony CLI](#).

¡Enhorabuena! Fastbook superado



Qualentum.com