



Symfony 5: Tutorial para desarrollar tu primera aplicación web

Escrito por [Rafa Morales](#) el 27 Noviembre 2019

► [TECNOLOGÍA](#)



Symfony

[Symfony](#) es un proyecto PHP de software libre que permite crear aplicaciones y sitios web rápidos y seguros de forma profesional.

En este tutorial podrás iniciarte en el desarrollo de aplicaciones web con Symfony siguiendo los pasos que te indicamos y adaptándolos al sitio que deseas construir.

Requisitos previos

Antes de instalar el ejecutable de Symfony, debemos tener instalado lo siguiente:

- [Control de versiones GIT](#).
- [Servidor web XAMPP](#) o similar.
- [Gestor de paquetes Composer](#).

La ruta de los ejecutables de los anteriores programas debe estar incluida en la variable de entorno PATH.

Instalación

Descargar el instalador de Symfony:

<https://symfony.com/download>

Si el instalador no lo ha hecho, incluir la carpeta de Symfony en la variable de entorno PATH del sistema:

- Linux: export PATH="\$HOME/.symfony/bin:\$PATH"
- Windows: "C:\Archivos de Programa\Symfony"

Comprobar la instalación y los requerimientos de Symfony:

```
symfony check:requirements
```

Creación de proyecto

[Privacidad](#)

Información ampliada

Crear el proyecto completo como aplicación web:

```
symfony new my_project_name --full
```

Comprobar las características del proyecto creado:

```
php bin/console about
```

Crear en el proyecto soporte para ser servido por un servidor Apache ([más información](#)):

```
composer require symfony/apache-pack
```

Crear un **VirtualHost** o **Alias** en Apache (*httpd.conf*) que apunte a la carpeta *public* del proyecto y visitar el proyecto desde un navegador web.

Ejemplo de VirtualHost:

```
#Se puede utilizar la barra "/" tanto para rutas en Windows como en Linux
<VirtualHost *:80>
    ServerName symfony.local
    DocumentRoot "/ruta/symfony/public/"
    <Directory "/ruta/symfony/public/">
        Options Indexes FollowSymLinks MultiViews
        AllowOverride All
        Require all granted
    </Directory>
    ErrorLog "logs/symfony.local-error.log"
    CustomLog "logs/symfony.local-access.log" common
</VirtualHost>
```

Recuerda que tendremos que introducir el dominio en nuestro fichero **hosts** del sistema operativo:

- Windows: C:\Windows\System32\drivers\etc\hosts
- Linux: /etc/hosts

```
127.0.0.1      symfony.local
```

Creación del controlador de página

Información ampliada

Crear una ruta (Route) para acceder a la página y un controlador (Controller) que construya la página.

Para crear las rutas utilizaremos anotaciones que nos permitirán definirlas en los comentarios dentro del propio controlador.

Instalar anotaciones:

```
composer require annotations
```

Crear el controlador en la carpeta del proyecto *src/Controller/*:

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

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

class LuckyController
{
    /**
     * @Route("/lucky/number")
     */
    public function number()
    {
        $number = random_int(0, 100);
        return new Response(
            '<html><body>Lucky number: ' . $number . '</body></html>'
        );
    }
}
```

Comprobar las rutas creadas en el proyecto:

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

Podemos automatizar la creación de un controlador y su plantilla mediante el siguiente comando:

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

Más información:

- [Toda la información sobre routas \(Routes\)](#)
- [Toda la información sobre controladores \(Controllers\)](#)

Renderización de plantillas

Podemos renderizar la salida de la página gracias al lenguaje [Twig](#), el cual está preparado para trabajar con plantillas, utilizar variables, sentencias de control, etc.

Instalar Twig:

```
composer require twig
```

Definir el controlador para que renderice la salida:

```

<?php
// src/Controller/LuckyController.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/number")
     */
    public function number()
    {
        $number = random_int(0, 100);
        return $this->render('lucky/number.html.twig', [
            'number' => $number,
        ]);
    }
}

```

Crear las plantillas de Twig en la carpeta *templates/*:

```

{# templates/lucky/number.html.twig #-}
<h1>Your lucky number is {{ number }}</h1>

```

Hacer que la plantilla herede el contenido de *base.html.twig*, que contiene la estructura general de un documento HTML. Hay que indicar en qué bloque de la plantilla padre se incluye el contenido:

```

{# templates/lucky/number.html.twig #-}
{% extends 'base.html.twig' %}

{% block body %}
    <h1>Your lucky number is {{ number }}</h1>
{% endblock %}

```

Más información:

- [Toda la información sobre plantillas \(Templates\)](#)

Limpieza de la caché

Symfony guarda en su propia caché las entidades, los formularios, sus validaciones, etc; para acelerar los procesos. En algunas ocasiones no se actualizan correctamente las acciones, por ejemplo las migraciones de la base de datos, por lo que es necesario limpiar la caché antes de seguir trabajando.

Limpiar la caché del entorno de desarrollo:

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

Limpiar la caché del entorno de producción:

```
php bin/console cache:clear --env=prod
```

Referencias a recursos

[Información ampliada](#)

El componente `Asset` nos permite generar URLs que nos dirijan a recursos estáticos de nuestra aplicación web como hojas de estilos CSS, archivos JavaScript o archivos de imágenes, así como controlar las distintas versiones de estos archivos. Sólo vamos a utilizar la funcionalidad que nos permite hacer referencia a archivos desde las plantillas Twig.

Instalar los paquetes necesarios:

```
composer require symfony/asset
```

Copiar los recursos (css, js, jpg, etc.) dentro de la carpeta `public`.

Hacer referencia a los recursos desde las plantillas, por ejemplo, `templates/base.html.twig`:

```
html>
  <head>
    <link href="{{ asset('css/main.css') }}" rel="stylesheet" />
```

Configuración

[Información ampliada](#)

Parámetros

Symfony permite almacenar parámetros y valores que podamos utilizar repetidamente en nuestro proyecto.

La definición se realizará bajo la clave `parameters` en el archivo `config/services.yaml`:

```
# config/services.yaml
parameters:
    # the parameter name is an arbitrary string (the 'app.' prefix is recommended
    # to better differentiate your parameters from Symfony parameters).
    app.admin_email: 'info@ticarte.com'
```

El uso de los parámetros se realizará mediante el siguiente método:

```
$adminEmail = $this->getParameter('app.admin_email');
```

Almacenamiento en base de datos

[Información ampliada](#)

Symfony utiliza el paquete Doctrine como ORM para el mapeo de datos entre objetos y bases de datos relacionales.

Instalar los paquetes necesarios:

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

Configurar el acceso a la base de datos en el fichero de variables del proyecto `.env`:

```
# .env
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name"
```

Crear la base de datos desde la consola:

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

Crear la entidad de datos con la que vamos a trabajar y añadimos todas las propiedades que poseerá. El comando creará la entidad dentro de la carpeta `src/Entity` y su repositorio dentro de `src/Repository`:

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

Preparar la migración de datos y ejecutar la migración para que se creen las tablas:

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

Para añadir nuevas propiedades a la entidad repetir los pasos anteriores, indicando al comienzo del comando el nombre de la entidad ya existente:

```
php bin/console make:entity
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

El **repositorio** de la entidad posee los métodos que nos permiten recuperar los elementos almacenados y filtrados. Es el lugar donde podemos incluir nuevas funciones como por ejemplo la de obtener el número de elementos totales o filtrados ([más sobre las consultas](#)):

```
class ProductRepository extends ServiceEntityRepository
{
    public function countAll()
    {
        return $this->createQueryBuilder('p')
            ->select('count(p.id)')
            ->getQuery()
            ->getSingleScalarResult();
    }
    // ...
}
```

Crear el sistema [CRUD](#) automáticamente para la entidad, lo que nos creará el formulario, el controlador y las plantillas:

```
php bin/console make:crud
```

Personalizar el **formulario** de creación/edición de objetos de la entidad creado en `src/Form/ProductType.php`:

```

namespace App\Form;

use App\Entity\Product;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\NumberType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ProductType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('Name', TextType::class, [
                'label' => 'Name:',
                'help' => 'Make sure to add a valid product name',
            ])
            ->add('Price', NumberType::class, [
                'label' => 'Price:',
                'help' => 'Add a decimal value',
                'attr' => [
                    'class' => 'form-control',
                    'placeholder' => 'Product price'
                ],
            ])
            ->add('Description', TextareaType::class, [
                'label' => 'Description:',
                'help' => 'Explain your details',
                'required' => false,
            ])
            ->add('Date', DateType::class, [
                'label' => 'Date:'
                'widget' => 'single_text',
                'attr' => [
                    'class' => 'js-datepicker'
                ],
            ])
            ->add('OutStock', ChoiceType::class, [
                'label' => 'Out of stock:',
                'help' => 'Out of stocks',
                'required' => false,
                'choices' => [
                    'Yes' => 'yes',
                    'No' => 'no',
                ],
                'attr' => [
                    'class' => 'form-control'
                ],
            ]);
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Product::class,
        ]);
    }
}

```

Más información:

- [Toda la información sobre formularios \(Forms\)](#)
- [Toda la información sobre campos del formulario \(Form Types\)](#)

Personalizar el **controlador** de creación/edición de objetos de la entidad creado en *src/Controller/ProductController.php*. Por ejemplo, añadir una nueva ruta que inserte un nuevo objeto en la base de datos directamente sin utilizar el formulario:

```
/**
 * @Route("/new/example", name="product_new_example")
 */
public function newExample(): Response
{
    // you can fetch the EntityManager via $this->getDoctrine()
    // or you can add an argument to the action: createProduct(EntityManagerInterface $entityManager)
    $entityManager = $this->getDoctrine()->getManager();

    $product = new Product();
    // This will trigger an error: the column isn't nullable in the database
    $product->setName('Laptop Lenovo');
    // This will trigger a type mismatch error: an integer is expected
    $product->setPrice('500');
    $product->setDescription('Lenovo Ideapad 250');

    // tell Doctrine you want to (eventually) save the Product (no queries yet)
    $entityManager->persist($product);

    // actually executes the queries (i.e. the INSERT query)
    $entityManager->flush();

    return new Response('Saved new product with id '.$product->getId());
}
```

Personalizar las **plantillas** de las diferentes renderizaciones de páginas creadas en *templates/product/*.

Para renderizar la salida del formulario por ejemplo:

```
{{ form_start(form) }}
{{ form_row(form.Name) }}
{{ form_row(form.Price) }}
{{ form_row(form.Stock) }}
<div class="form-control">
    <i class="fa fa-calendar"></i> {{ form_label(form.Date) }}
    {{ form_widget(form.Date) }}
    <small>{{ form_help(form.Date) }}</small>
    <div class="form-error">
        {{ form_errors(form.Date) }}
    </div>
</div>
<button class="btn">{{ button_label|default('Save') }}</button>
{{ form_end(form) }}
```

Si queremos añadir código Javascript, como por ejemplo "**datepicker**" para seleccionar la fecha, debemos añadirlo al bloque correspondiente dentro de las plantillas *edit.html.twig* y *new.html.twig*:

```
{% block javascripts %}
<script>
$(document).ready(function() {
    $('.js-datepicker').datepicker({
        format: 'yyyy-mm-dd'
    });
});
</script>
{% endblock %}
```

Más información:

- [Toda la información sobre renderización de formularios \(Form rendering\)](#)

Validación de entidades y formularios

[Información ampliada](#)

Symfony posee su propio mecanismo para validar tanto entidades como formularios.

Instalar los paquetes necesarios:

```
composer require symfony/validator  
composer require doctrine/annotations
```

Incluir las validaciones en la entidad mediante las anotaciones `@Assert` ([listado de validaciones - Constraints](#)):

```
namespace App\Entity;  
  
use Doctrine\ORM\Mapping as ORM;  
use Symfony\Component\Validator\Constraints as Assert;  
  
/**  
 * @ORM\Entity(repositoryClass="App\Repository\ProductRepository")  
 */  
class Product  
{  
    /**  
     * @ORM\Id()  
     * @ORM\GeneratedValue()  
     * @ORM\Column(type="integer")  
     */  
    private $id;  
  
    /**  
     * @Assert\NotBlank  
     * @Assert\Length(  
     *      min = 2,  
     *      max = 255,  
     *      minMessage = "Your first name must be at Least {{ limit }} characters Long",  
     *      maxMessage = "Your first name cannot be Longer than {{ limit }} characters"  
     * )  
     * @Assert\NotEqualTo(  
     *      "Tablet",  
     *      message = "This value should not be equal to {{ compared_value }}"  
     * )  
     * @ORM\Column(type="string", Length=255)  
     */  
    private $Name;  
  
    /**  
     * @Assert\Positive(  
     *      message = "This value should be positive"  
     * )  
     * @ORM\Column(type="decimal", precision=10, scale=0, nullable=true)  
     */  
    private $Price;  
...}
```

Para que el formulario nos muestre los mensajes personalizados de estas validaciones debemos añadir como mínimos a los campos del formulario en `src/Form/ProductType.php` el tipo de campo que es. En caso contrario, simplemente nos mostrará un mensaje indicando que existe un error.

```
$builder  
->add('Name', TextType::class);
```

De esta manera, al enviar el formulario de creación/edición, nos mostrará los errores bajo el *label* del campo.

Crear nuestras propias validaciones (Constraints)

Crear la validación:

```
<?php  
// src/Validator/Constraints/ContainsAlphanumeric.php  
namespace App\Validator\Constraints;  
  
use Symfony\Component\Validator\Constraint;  
  
/**  
 * @Annotation  
 */  
class ContainsAlphanumeric extends Constraint  
{  
    public $message = 'The string "{{ string }}" contains an illegal character: it can only contain letters or  
numbers.';  
}
```

Crear la lógica de la validación:

```

<?php
// src/Validator/Constraints/ContainsAlphanumericValidator.php
namespace App\Validator\Constraints;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;
use Symfony\Component\Validator\Exception\UnexpectedTypeException;
use Symfony\Component\Validator\Exception\UnexpectedValueException;

class ContainsAlphanumericValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        if (!$constraint instanceof ContainsAlphanumeric) {
            throw new UnexpectedTypeException($constraint, ContainsAlphanumeric::class);
        }

        // custom constraints should ignore null and empty values to allow
        // other constraints (NotBlank, NotNull, etc.) take care of that
        if (null === $value || '' === $value) {
            return;
        }

        if (!is_string($value)) {
            // throw this exception if your validator cannot handle the passed type so that it can be marked as
invalid
            throw new UnexpectedValueException($value, 'string');

            // separate multiple types using pipes
            // throw new UnexpectedValueException($value, 'string|int');
        }

        if (!preg_match('/^[a-zA-Z0-9 ]+$/', $value, $matches)) {
            $this->context->buildViolation($constraint->message)
                ->setParameter('{{ string }}', $value)
                ->addViolation();
        }
    }
}

```

Aplicar la validación:

```

<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;
use App\Validator\Constraints as AcmeAssert;

/**
 * @ORM\Entity(repositoryClass="App\Repository\ProductRepository")
 */
class Product
{
    ...
    /**
     * @AcmeAssert\ContainsAlphanumeric
     */
    private $Name;
    ...
}

```

Almacenamiento de archivos mediante formularios

Información ampliada

Symfony nos permite recibir archivos de los usuarios mediante campos de formulario y almacenarlos en nuestra carpeta `public` del proyecto. En la base de datos se almacenará solamente el nombre del archivo o la ruta de acceso al archivo si lo añadimos.

Añadir a la **entidad** un campo de texto que almacene el nombre y extensión del archivo:

```
php bin/console make:entity
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

Añadir al **formulario** de la entidad un campo del tipo `FileType` para que el navegador muestre el controlador de subida de archivos, pero indicar que dicho campo no está mapeado *unmapped* directamente en la base de datos, además de indicar una validación para el mismo, ya que los campos que no están mapeados no se pueden validar mediante anotaciones:

```
// src/Form/ProductType.php
namespace App\Form;

use App\Entity\Product;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\FileType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Validator\Constraints\File;

class ProductType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            // ...
            ->add('brochure', FileType::class, [
                'label' => 'Brochure (PDF file)',

                // unmapped means that this field is not associated to any entity property
                'mapped' => false,

                // make it optional so you don't have to re-upload the PDF file
                // everytime you edit the Product details
                'required' => false,

                // unmapped fields can't define their validation using annotations
                // in the associated entity, so you can use the PHP constraint classes
                'constraints' => [
                    new File([
                        'maxSize' => '1024k',
                        'mimeTypes' => [
                            'application/pdf',
                            'application/x-pdf',
                        ],
                        'mimeTypesMessage' => 'Please upload a valid PDF document',
                    ])
                ],
            ])
        ;
    }
//...
}
```

Crear un **parámetro** en el proyecto para indicar el directorio donde se almacenarán los archivos:

```
# config/services.yaml  
  
# ...  
parameters:  
    brochures_directory: '%kernel.project_dir%/public/uploads/brochures'
```

Actualizar el **controlador** para manejar la respuesta del formulario:

```

// src/Controller/ProductController.php
namespace App\Controller;

use App\Entity\Product;
use App\Form\ProductType;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\File\Exception\FileException;
use Symfony\Component\HttpFoundation\File\UploadedFile;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;

class ProductController extends AbstractController
{
    /**
     * @Route("/product/new", name="app_product_new")
     */
    public function new(Request $request)
    {
        $product = new Product();
        $form = $this->createForm(ProductType::class, $product);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            /** @var UploadedFile $brochureFile */
            $brochureFile = $form['brochure']->getData();

            // This condition is needed because the 'brochure' field is not required
            // so the PDF file must be processed only when a file is uploaded
            if ($brochureFile) {
                $originalFilename = pathinfo($brochureFile->getClientOriginalName(), PATHINFO_FILENAME);
                // This is needed to safely include the file name as part of the URL
                // Enable "Intl" extension in "php.ini"
                // https://stackoverflow.com/questions/33869521/how-can-i-enable-php-extension-intl
                $safeFilename = transliterator_transliterate('Any-Latin; Latin-ASCII; [^A-Za-z0-9_] remove;
Lower()', $originalFilename);
                $newFilename = $safeFilename.'-'.$_.uniqid().'.'.$brochureFile->guessExtension();

                // Move the file to the directory where brochures are stored
                try {
                    $brochureFile->move(
                        $this->getParameter('brochures_directory'),
                        $newFilename
                    );
                } catch (FileException $e) {
                    // ... handle exception if something happens during file upload
                }

                // Updates the 'brochureFilename' property to store the PDF file name
                // instead of its contents
                $product->setBrochureFilename($newFilename);
            }
        }

        // ... persist the $product variable or any other work

        return $this->redirect($this->generateUrl('app_product_list'));
    }

    return $this->render('product/new.html.twig', [
        'form' => $form->createView(),
    ]);
}

```

Actualizar la plantilla de la visualización de la entidad para incluir un enlace al archivo, o en el caso de una imagen, poder mostrarla:

```
{# templates/product/show.html.twig #}
<a href="{{ asset('uploads/brochures/' ~ product.brochureFilename) }}>View brochure (PDF)</a>
```

Actualizar la **plantilla del formulario**. Hay que tener en cuenta que el formulario será diferente al crear un nuevo elemento o al editararlo, ya que al editararlo puede que tenga un archivo previo asignado. Por ejemplo, mostrar un enlace al archivo en caso de que exista al editar el elemento:

```
{# templates/product/_form.html.twig #}
{{ form_start(form) }}
{# ... #}

{{ form_row(form.brochure) }}
{% if product.brochure %}
    <a href="{{ asset('uploads/brochures/' ~ product.brochureFilename) }}>View brochure (PDF)</a>
{% endif %}
{{ form_end(form) }}
```

Relaciones en base de datos

Información ampliada

Symfony nos permite realizar relaciones entre diferentes entidades y que estas relaciones se reflejen en la base de datos.

Existen dos **tipos de relaciones** principales:

- ManyToOne / OneToMany: Crea en una de las tablas de la base de datos una clave foránea al identificador de la otra tabla.
- ManyToMany: Crear una nueva tabla en la base de datos que relaciona los identificadores de dos tablas.

Vamos a implementar un ejemplo en el que la entidad producto posea una relación del tipo *ManyToOne* hacia la entidad categoría, de tal modo que un producto se relacione con una categoría, y una categoría se relacione con muchos productos.

Partimos de que ya poseemos la entidad categoría creada en nuestro proyecto.

Añadir a la entidad producto la nueva **relación**:

```
php bin/console make:entity

Class name of the entity to create or update (e.g. BraveChef):
> Product

New property name (press <return> to stop adding fields):
> category

Field type (enter ? to see all types) [string]:
> relation

What class should this entity be related to?:
> Category

Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
> ManyToOne

Is the Product.category property allowed to be null (nullable)? (yes/no) [yes]:
> no

Do you want to add a new property to Category so that you can access/update Product objects from it - e.g.
$category->getProducts()? (yes/no) [yes]:
> yes

New field name inside Category [products]:
> products

Do you want to automatically delete orphaned App\Entity\Product objects
(orphanRemoval)? (yes/no) [no]:
> no

New property name (press <return> to stop adding fields):
>
(press enter again to finish)
```

Este proceso creará en la entidad producto un nuevo campo categoría, y si lo hemos elegido, los métodos necesarios para que desde la categoría podamos listar los productos.

Comprobar que el campo categoría posea la anotación *ManyToOne* y *Join*. Si es así, preparar la **migración** de datos y ejecutar la migración para que se cree la clave foránea:

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

Añadir nuevo campo al **formulario** de producto para que nos liste las categorías disponibles:

```

<?php

namespace App\Form;

use App\Entity\Category;
use App\Entity\Product;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\CollectionType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

```

```

class ProductType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            //...
            ->add('category', EntityType::class, array(
                // looks for choices from this entity
                'class' => Category::class,
                // uses the Product.Name property as the visible option string
                'choice_label' => 'Name',
                // used to render a select box, check boxes or radios
                'multiple' => false,
                // used to send properties to HTML
                'attr' => array('class' => 'form-control')
            ))
        ;
    }

//...
}

```

Rediseñar la **plantilla** para acceder a las propiedades de la clase referenciada, por ejemplo en *show.html.twig*:

```
<td>{{ product.category.Name }}</td>
```

Carga de datos fijos

Información ampliada

Symfony permite cargar datos fijos en la base de datos a modo de prueba.

Instalar los paquetes necesarios:

```
composer require --dev doctrine/doctrine-fixtures-bundle
```

Por defecto se nos crea un Fixture en *src/DataFixtures/AppFixtures* para hacer una carga de datos general a la aplicación.

Si queremos, podemos crear otros Fixtures adicionales, por ejemplo por entidad. Crear la clase *ProductFixtures*:

```
php bin/console make:fixtures
```

Este proceso creará el fichero *src/DataFixtures/ProductFixtures.php*. Añadir la entidad que se utiliza (use) ya que por defecto no se incluye. Modificar el fichero para añadir los datos fijos que deseemos:

```
<?php

namespace App\DataFixtures;

use App\Entity\Product;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;

class ProductFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        $product = new Product();
        $product->setName('Keyboard');
        $product->setPrice(12);

        $manager->persist($product);
        $manager->flush();
    }
}
```

Ejecutar la creación de datos fijos. Por defecto vacía las tablas y ejecuta todos los Fixtures del proyecto.

```
php bin/console doctrine:fixtures:load
```

Podemos especificar si deseamos añadir los datos y ejecutar solamente un Fixture concreto:

```
php bin/console doctrine:fixtures:load --append --group=ProductFixtures
```

Implementación de menús

Aunque hay paquetes que ayudan a crear menús más complejos, a continuación vamos a mostrar como implementar un simple menú utilizando sólo un controlador y su plantilla.

Crear el **controlador** que construya el menú. Se utilizará un array asociativo para almacenar los elementos que tenga el menú, junto con todas las opciones que deseemos definirle a cada elemento. Nos ayudaremos del nombre de ruta (Route) para conocer en qué página estamos y así poder marcar alguno de los elementos como activo. Este nombre de ruta nos vendrá desde Twig como veremos a continuación:

```

<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class MenuController extends AbstractController
{

/**
 * @var String $route_name
 * Machine name of a route
 */
public function mainMenu(String $route_name)
{
    $items['home']['title'] = 'Home';
    $items['home']['url'] = $this->generateUrl('home');
    if ($route_name == 'home') {
        $items['home']['class'] = "active";
    }

    $items['products']['title'] = 'Products';
    $items['products']['url'] = $this->generateUrl('producto_index');
    if (in_array($route_name, ['producto_index', 'producto_show', 'producto_new', 'producto_edit'])) {
        $items['products']['class'] = "active";
    }

    return $this->render('menu/_main.html.twig', [
        'items' => $items,
    ]);
}

}

```

Crear la **plantilla** del menú:

```

<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="#">TicArte Symfony</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav mr-auto">
            {% for item in items %}
                <li class="nav-item {% if item.class is defined %}{{ item.class }}{% endif %}">
                    <a class="nav-link" href="{{ item.url }}>{{ item.title }}</a>
                </li>
            {% endfor %}
        </ul>
    </div>
</nav>

```

Llamamos a la **plantilla** desde otra plantilla para renderizar el menú, incluyendo como variable el nombre de la ruta en la que se está visualizando:

```

{{ render(controller('App\\Controller\\MenuController::mainMenu', {'route_name': app.request.attributes.get('_route')})) }}

```

Administración de usuarios en base de datos

Información ampliada

Symfony puede trabajar con autenticación de usuarios desde diversas fuentes de datos. En este apartado nos centramos en el almacenamiento de los usuarios en una base de datos local y en su posterior uso en los diversos controladores del sitio.

Muchos de los paquetes que ya hemos instalado anteriormente son necesarios para el siguiente proceso, por lo que hay que tenerlo en cuenta si comenzamos un nuevo proyecto directamente en este apartado.

Instalar los nuevos paquetes necesarios:

```
composer require symfony/security-bundle  
composer require --dev doctrine/doctrine-fixtures-bundle
```

Crear la entidad para almacenar los usuarios:

```
php bin/console make:user
```

Este comando nos creará la entidad del usuario que implementará la interfaz [UserInterface](#) con la lógica de control necesaria y configurará en el paquete config/packages/security.yaml el [User Providers](#) y los encoders para encriptar las contraseñas.

Preparar la migración de datos y ejecutar la migración para que se creen las tablas:

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

Añadir más campos a nuestra entidad en caso de que sea necesario:

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

Preparar la migración de datos y ejecutar la migración para que se creen las tablas:

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

Crear usuarios demo con el paquete [DoctrineFixtureBundle](#), llamando a la clase *UserFixtures*.

```
php bin/console make:fixtures
```

Este proceso creará el fichero *src/DataFixtures/UserFixtures.php*. Modificar el fichero con el siguiente contenido para añadir los usuarios que deseemos y se codifique su contraseña:

```

<?php

namespace App\DataFixtures;

use App\Entity\User;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;

class UserFixtures extends Fixture
{
    private $passwordEncoder;

    public function __construct(UserPasswordEncoderInterface $passwordEncoder)
    {
        $this->passwordEncoder = $passwordEncoder;
    }

    public function load(ObjectManager $manager)
    {
        $user = new User();
        $user->setEmail('info@ticarte.com');
        $user->setPassword($this->passwordEncoder->encodePassword(
            $user,
            'ticarte.com'
        ));
        $manager->persist($user);
        $manager->flush();
    }
}

```

Ejecutar la creación de datos demo añadiendo a los existentes (aunque ahora mismo tenemos la tabla vacía) y ejecutando sólamente los datos de usuario:

```
php bin/console doctrine:fixtures:load --append --group=UserFixtures
```

Crear el formulario de inicio de sesión. El comando nos preguntará sobre la autenticación y si deseamos crear el cierre de sesión también:

```

php bin/console make:auth

What style of authentication do you want? [Empty authenticator]:
[0] Empty authenticator
[1] Login form authenticator
> 1

The class name of the authenticator to create (e.g. AppCustomAuthenticator):
> LoginFormAuthenticator

Choose a name for the controller class (e.g. SecurityController) [SecurityController]:
> SecurityController

Do you want to generate a '/logout' URL? (yes/no) [yes]:
> yes

```

Este comando creará el formulario de login, su controlador y su plantilla.

Todo es funcional excepto la página a la que se redirige el usuario al iniciar sesión, que debemos modificar en *src/Security/LoginFormAuthenticator.php*:

```
// ...
public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
{
    // ...

    - throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
+ // redirect to some "app_homepage" route - or wherever you want
+ return new RedirectResponse($this->urlGenerator->generate('my_name_route'));
}
```

Y la página a la que se redirige el usuario al cerrar sesión, que debemos modificar en *config/packages/security.yaml*:

```
security:
    firewalls:
        main:
            logout:
                path: app_logout
                target: app_login
```

Crear el formulario de registro:

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

Asegurar el acceso a plantillas de URLs desde el fichero *config/packages/security.yaml*:

```
security:
    access_control:
        # matches /admin/users/*
        - { path: '^/admin/users', roles: ROLE_SUPER_ADMIN }
        # matches /admin/* except for anything matching the above rule
        - { path: '^/admin', roles: ROLE_ADMIN }
```

Asegurar el acceso al controlador en general y a sus métodos en particular mediante anotaciones, por ejemplo:

```
// src/Controller/AdminController.php
// ...

use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;

/**
 * Require ROLE_ADMIN for *every* controller method in this class.
 *
 * @IsGranted("ROLE_ADMIN")
 */
class AdminController extends AbstractController
{
    /**
     * Require ROLE_ADMIN for only this controller method.
     *
     * @IsGranted("ROLE_ADMIN")
     */
    public function adminDashboard()
    {
        // ...
    }
}
```

Asegurar el acceso a zonas del código de manera simple:

```
if( $this->get('security.authorization_checker')->isGranted('IS_AUTHENTICATED_FULLY') ) {  
    // Only for authenticated users  
}
```

Asegurar el acceso a zonas del código, si no se está logueado se lanzará una excepción y se redirigirá a la página de inicio de sesión, si se está logueado pero no se posee el rol se lanzará una excepción y se mostrará un error 403, por ejemplo:

```
// src/Controller/AdminController.php  
  
public function adminDashboard()  
{  
    $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');  
  
    // or add an optional message - seen by developers  
    $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'User tried to access a page without having ROLE_ADMIN');  
}
```

Asegurar el acceso desde una plantilla, por ejemplo:

```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}  
    <p>Hello {{ app.user.name }}</p>  
{% endif %}
```

Bibliografía

- [Documentación oficial](#). Symfony.com
- [Symfony Tutorial: Building a Blog](#). Auth0



[Symfony 5: Tutorial para desarrollar tu primera aplicación web](#) de [Rafa Morales](#) está bajo una licencia [Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional](#)

SYMFONY

PHP

[Inicie sesión](#) o [regístrate](#) para enviar comentarios

Artículos similares

- ❑ [Manuales y recursos de PHP](#)
- ❑ [Cómo implementar un servidor REST en PHP con SlimFramework](#)
- ❑ [Ejercicios prácticos de PHP](#)
- ❑ [Nuestro servidor web propio: XAMPP](#)

Suscríbete a TicArte

Tu correo electrónico

Acepto recibir en mi correo electrónico los artículos publicados [Términos y Condiciones de Uso](#) *

[Suscríbeme](#)

Síguenos





Administración de Sistemas Operativos



Gestión de Bases de Datos



Lenguajes de Marcas y Sistemas de Gestión de Información



Planificación y Administración de Redes



Sistemas Operativos en Red

