

SYMPHONY 5



ÍNDICE

Introducción	3
Instalación y configuración	4
Modelo - Vista - Controlador	6
Definición	6
Estructura	6
Controladores	7
Enrutamiento	9
TDD	10
Vistas	11
Proyecto 'Mascotas'	13
Bibliografía	21

Introducción

Según la página oficial de 'Symfony', este es un framework PHP full-stack de software libre que permite crear aplicaciones y sitios web rápidos y seguros de forma profesional.

Este framework es muy popular entre los programadores PHP de Europa, especialmente en España.

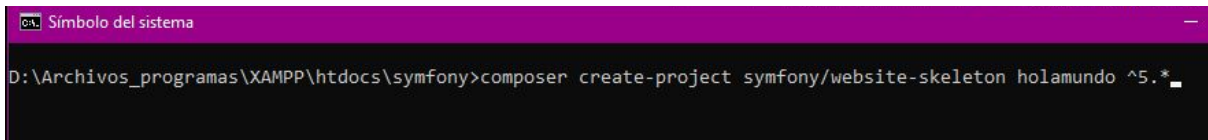
En este informe se redacta información de interés sobre el mismo, así como la creación de un proyecto básico.

Instalación y configuración

Para poder realizar un proyecto 'Symfony' es necesario cumplir los siguientes pasos:

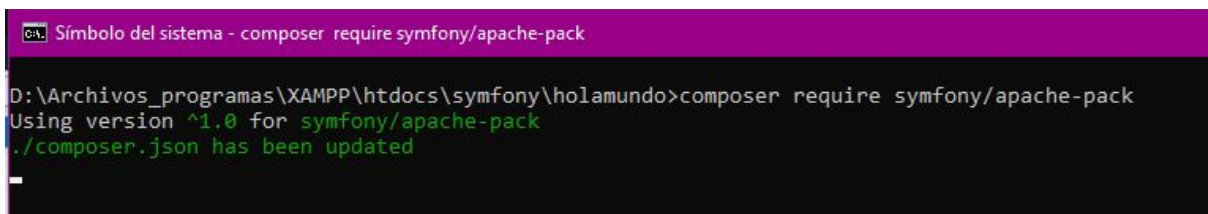
- Tener instalado un servidor de aplicaciones web local con, por lo menos, PHP 7.2. En mi caso, utilizaré XAMPP.
- Tener instalado Composer.
- En la carpeta 'xampp/htdocs' se ejecuta el siguiente comando. Yo creé una carpeta 'symfony' para una mejor organización. Esto nos crea la estructura de carpetas del proyecto.

```
composer create-project symfony/website-skeleton nombre-proyecto ^5.*
```

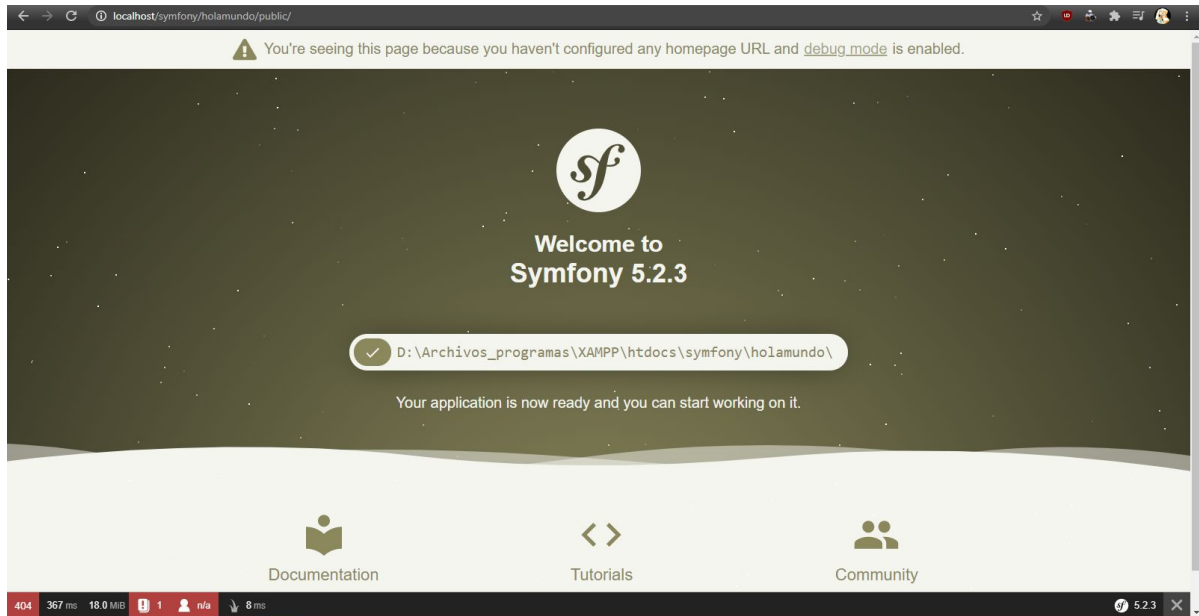


- Para que el proyecto de 'Symfony' funcione en un servidor Apache, ejecutamos lo siguiente dentro del proyecto:

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



- Para comprobar que se ha generado correctamente, en un navegador escribimos 'localhost/ruta-proyecto/public'. En mi caso sería 'localhost/symfony/holamundo/public'. Si todo ha funcionado correctamente, el sitio web debería lucir de esta forma:



Modelo - Vista - Controlador

Definición

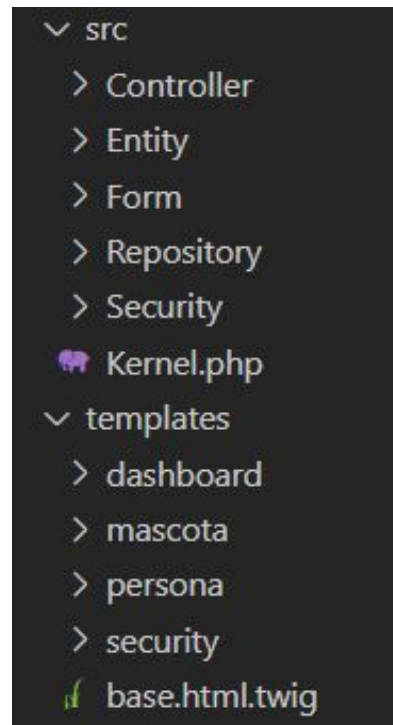
Esta arquitectura de software surge de la necesidad de crear código más robusto y reutilizable, cuyo mantenimiento sea más cómodo. Se separa en tres capas:

- Modelo: aquí se trabaja con los datos almacenados en la base de datos correspondiente.
- Vista: aquí está el código que produce la visualización de las interfaces de usuario.
- Controlador: aquí está el código que responde a las acciones que solicite el usuario sobre los datos.

Estructura

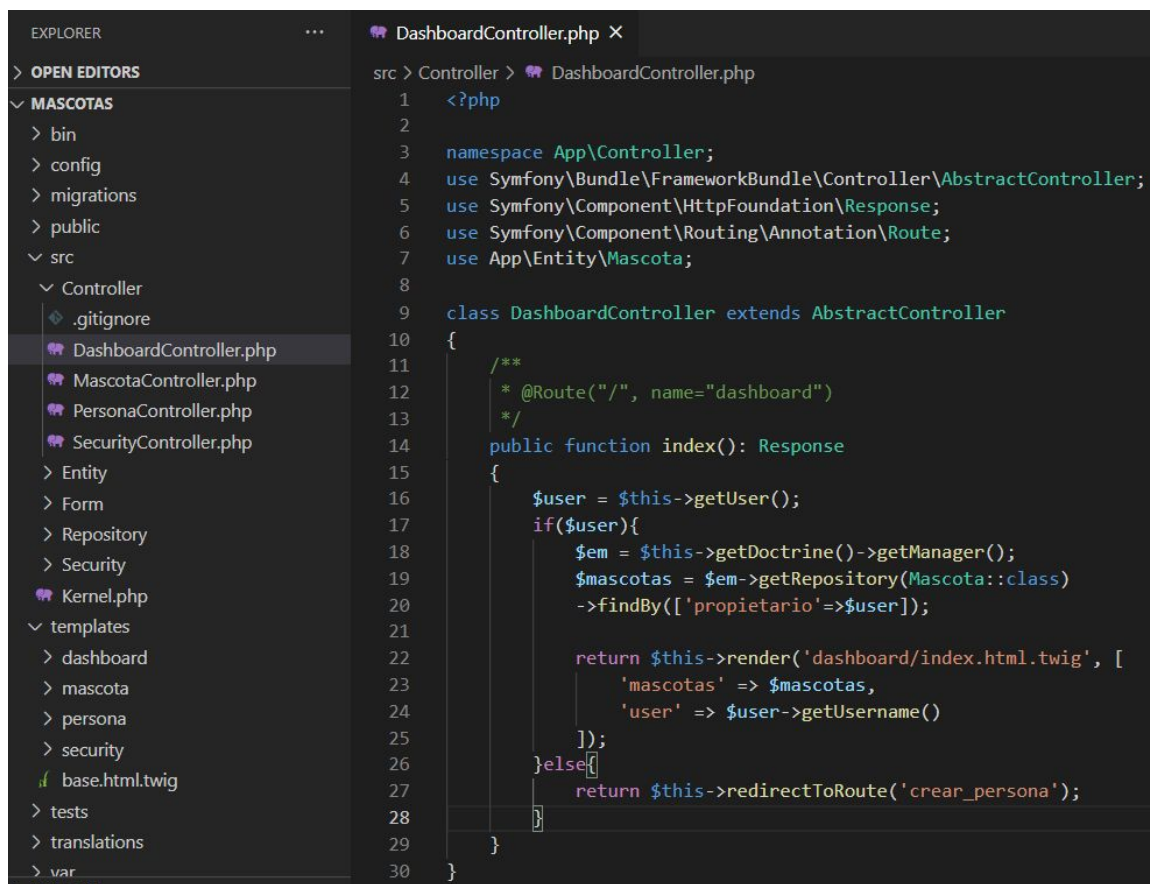
Symfony trabaja en MVC, pero con una estructura de archivos un poco diferente:

- En la carpeta 'src' se guardan los **controladores**, las entidades (con un comando se convierten en tablas de la base de datos), los formularios y los repositorios, que son las clases que tienen los métodos para manipular los datos (**modelo**).
- En la carpeta 'templates' se guardan las **vistas** necesarias en formato '.twig'. Estas reciben variables con los datos desde el controlador y las utilizan para pintar la información con el uso de bucles y condicionales (siempre que sea necesario).



Controladores

Para entender cómo funciona un controlador en Symfony, vamos a ver como ejemplo el controlador de un 'dashboard'. Este es el usado en el proyecto práctico que se presenta más adelante.



Para crear un controlador, hacemos uso del siguiente comando:

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

Después de rellenar los datos solicitados, se nos genera un archivo .php en la carpeta 'Controller' con su estructura ya definida.

Según la ruta en la que estemos, se ejecutará un método u otro. En este caso, cuando estemos en el inicio, comprobará si ya hay un usuario logueado y

pasará a recoger datos relacionados con este usuario y enviarlos a la vista correspondiente. Si aún no se ha iniciado sesión, irá a la página donde se crean los usuarios.

El objetivo de un controlador es siempre el mismo: crear y devolver un objeto Response.

Cuando es el controlador de un formulario, donde recibimos datos, se recibe un objeto del tipo 'Request'.

```
/**
 * @Route("/guardar/mascota", name="guardar-mascota")
 */
public function index(Request $request): Response
{
    $mascota = new Mascota();
    $form = $this->createForm(MascotaType::class, $mascota);
    $form->handleRequest($request);

    if($form->isSubmitted() && $form->isValid()){
        $persona = $this->getUser();
```

Este ejemplo es del controlador de mascota, para guardar una en la base de datos. Se crea el formulario indicando el tipo de dato y el objeto donde se guardarán los datos. Una vez comprobemos que se ha rellenado el formulario con el 'if', hacemos lo necesario.

Para mostrar el formulario, lo añadimos a los datos a renderizar:

```
return $this->render('mascota/index.html.twig', [
    'formulario' => $form->createView(),
    'persona' => $this->getUser()
]);
```


Enrutamiento

Es muy importante generar las rutas con sentido, pues es lo que se ve en la URL del proyecto. Estas se instancian en los controladores, por encima del método que contiene el código a ejecutar en esa dirección.

Al `@Route` se le guardan dos valores, la ruta en tipo 'string' y el nombre de la ruta. Esta última se utiliza para indicar la ruta cuando queremos hacer un 'redirect' desde otro lado. De esta forma, si cambiamos esa parte de URL, no tenemos que modificarlo en todos los sitios donde se llama.

```
/**
 * @Route("/guardar/mascota", name="guardar-mascota")
 */
public function index(Request $request): Response
{
}
```

Los redirect se hacen de esta forma:

```
$em->flush();
return $this->redirectToRoute("dashboard");
}
```

Para recibir valores de la ruta, es necesario indicarlo entre llaves (`{ }`) y poner dentro el nombre de variable. El método recibirá como parámetro esa variable para poder operar con ella.

```
/**
 * @Route("/mascota/{id}", name="ver-mascota")
 */
public function verMascota($id){
    $em = $this->getDoctrine()->getManager();
    $mascota = $em->getRepository( Mascota::class)->find($id);
    return $this->render('mascota/verMascota.html.twig', [
        'mascota' => $mascota
    ]);
}
```

TDD

Las siglas TDD significan 'Test Driven Development', Desarrollo Dirigido mediante Tests, en español. Los tests nos ayudan a crear código de calidad y sencillo de mantener.

Esto es una técnica basada en realizar tests que describen la funcionalidad antes de desarrollarla. De esta manera, el código final debe ir consiguiendo pasar los test y avanzar mediante refactorización.

Para programar de esta forma, seguiremos la teoría descrita por James Shore:

1. Pensar qué test es más adecuado para dirigirnos a la finalidad del código.
2. Escribimos código de prueba y ejecutamos el test. Seguramente el test falle, así que prestamos atención a los mensajes de error.
3. Escribimos un poco de código de producción, ejecutamos los test y deberían ser correctos.
4. Refactorizar el código y vemos que los tests siguen siendo correctos.

Para hacer esto en nuestro proyecto de Symfony, es necesario primero instalar lo siguiente:

```
composer require --dev symfony/phpunit-bridge
```

Luego, lanzamos este otro comando, la primera vez que lo ejecutemos se instalará todo lo necesario para lanzar nuestra batería de pruebas:

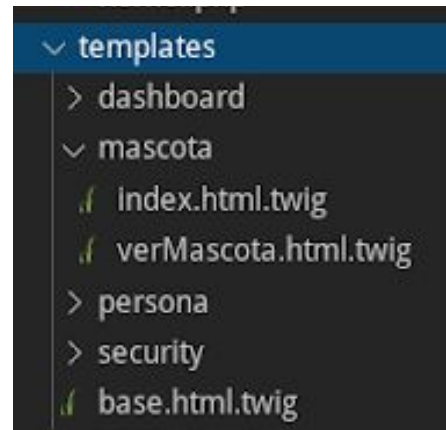
```
php bin/phpunit
```

A partir de aquí, ya podemos crear nuestros tests de cualquier tipo y aplicar el TDD.

Vistas

Las vistas se crearán en la carpeta 'templates'. Se generan carpetas para cada controlador creado, de esta forma, está todo más organizado.

Son en formato html.twig para poder tener un "html vitaminado": es posible hacer el uso de bucles y condicionales.



Aquí podemos ver un ejemplo:

```

./ index.html.twig •
templates > dashboard > ./ index.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block title %}{{parent()}}{% endblock %}
4
5  {% block body %}
6  {{parent()}}
7      <div class="container-fluid">
8          <h1 class="text-center my-3">Mascotas de {{user}}</h1>
9          <div class="row text-center">
10             <div class="col-4 offset-4">
11                 {% for mascota in mascotas %}
12                     <div class="bg-info p-4 m-4"
13                     onclick="window.location.href= '{{path('ver-mascota', {id: mascota.id})}}'">
14                         <div class="">
15                             <h3>{{mascota.nombre}}</h3>
16                         </div>
17                     </div>
18                 {% endfor %}
19             </div>
20         </div>
21     </div>
22
23 {% endblock %}
24

```

En el 'block title' se utiliza la función 'parent()' para llamar así al valor que tenga el html padre, que sería el base.html.twig. De esta forma, nos ahorramos tener que escribir el título en todas las vistas cada vez que lo

queramos modificar. En el 'block body' también se llama a su padre para poder recibir la barra de navegación.

Como se puede ver en la línea 11 de código, se utiliza un bucle for con la variable mascotas, esta es recibida desde el controlador.

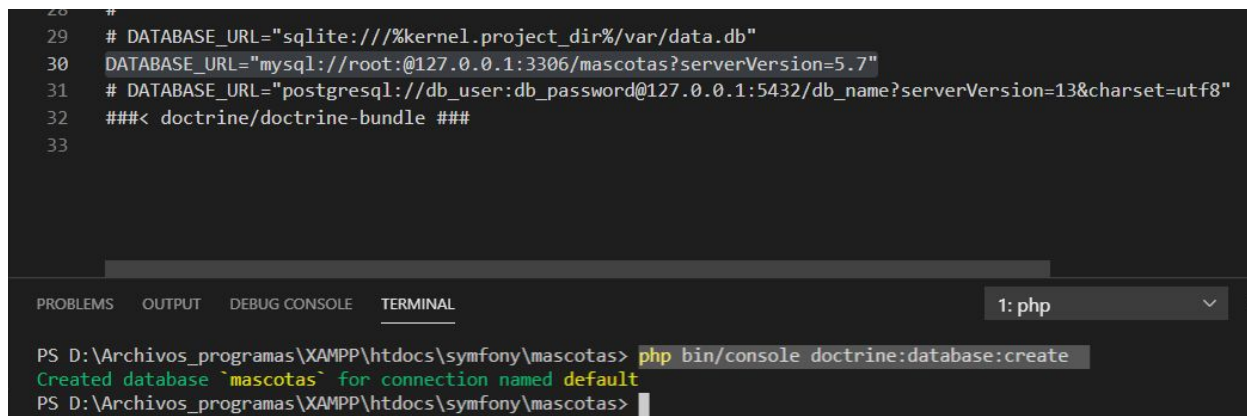
Gracias a estas funciones y más, es posible ahorrar líneas de código, quedando todo mucho más legible y dinámico.

Proyecto 'Mascotas'

Como ejemplo he decidido hacer una aplicación bastante sencilla donde un usuario pueda registrar sus mascotas.

Primero creamos el proyecto como bien se explicó en el apartado [Instalación y configuración](#). En el archivo `.env` definimos los datos de nuestra base de datos, la cual llamaremos 'mascotas'. En la terminal ejecutamos el siguiente comando (dentro del directorio del proyecto) para así crear la base de datos en el phpmyadmin. En el caso de que ya la tengamos creada, no es necesario este paso.

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



The screenshot shows a code editor with a dark theme. The top part displays the `.env` file with the following content:

```
28 #  
29 # DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"  
30 DATABASE_URL="mysql://root:@127.0.0.1:3306/mascotas?serverVersion=5.7"  
31 # DATABASE_URL="postgresql://db_user:db_password@127.0.0.1:5432/db_name?serverVersion=13&charset=utf8"  
32 ###< doctrine/doctrine-bundle ###  
33
```

The bottom part shows a terminal window with the following content:

```
PS D:\Archivos_programas\XAMPP\htdocs\symfony\mascotas> php bin/console doctrine:database:create  
Created database `mascotas` for connection named default  
PS D:\Archivos_programas\XAMPP\htdocs\symfony\mascotas>
```

Una vez creada la base de datos, procedemos a generar las tablas:

- Persona: id, nombre
- Mascota: id, nombre, raza, propietario

Para la tabla persona, que vendría a ser un usuario, creamos la entidad con el siguiente comando, lo ejecutamos y rellenamos los datos solicitados:

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

```

PS D:\Archivos_programas\XAMPP\htdocs\symfony\mascotas> php bin/console make:user

The name of the security user class (e.g. User) [User]:
> Persona

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
> yes

Enter a property name that will be the unique "display" name for the user (e.g. email, username, uuid) [email]:
> username

Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be checked/hashed
tem (e.g. a single sign-on server).

Does this app need to hash/check user passwords? (yes/no) [yes]:
> yes

created: src/Entity/Persona.php
created: src/Repository/PersonaRepository.php
updated: src/Entity/Persona.php
updated: config/packages/security.yaml

Success!

```

Para las mascotas, ejecutamos el siguiente comando y rellenamos los datos igual que en el anterior:

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

Una vez hecho esto, deberíamos tener los siguientes archivos creados:

```

src
├── Controller
├── Entity
│   ├── .gitignore
│   ├── Mascota.php
│   └── Persona.php
├── Repository
│   ├── .gitignore
│   ├── MascotaRepository.php
│   └── PersonaRepository.php
├── Kernel.php
└── templates

```

Para actualizar la base de datos con las tablas, ejecutamos este comando (sólo si la base de datos a usar no tiene las tablas de antes):

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

Creamos las relaciones entre las tablas, una persona puede tener una o más mascotas, mientras que una mascota solo puede tener un propietario. En la página de symfony, están detallados y hay ejemplos sobre cómo hacer las relaciones.

En personas :

```
class Persona implements UserInterface
{
    /**
     * @ORM\OneToMany(targetEntity="App\Entity\Mascota", mappedBy="propietario")
     */
    private $mascotas;
```

En mascotas:

```
class Mascota
{
    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\Persona", inversedBy="mascotas")
     */
    private $propietario;
```

Para el login del usuario, usamos un bundle de seguridad y encriptar las contraseñas:

```
composer require symfony/security-bundle
```

Generamos el login:

```
php bin/console make:auth
```

Para crear los los formularios:

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

Una vez hecho todo esto, pasamos a la creación de controladores. En ellos definimos los métodos necesarios:

- En el dashboard queremos mostrar todas las mascotas del usuario iniciado.

```
<?php

namespace App\Controller;

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

class DashboardController extends AbstractController
{
    /**
     * @Route("/", name="dashboard")
     */
    public function index(): Response
    {
        $user = $this->getUser();

        if($user){
            $em = $this->getDoctrine()->getManager();
            $mascotas = $em->getRepository(Mascota::class)->findBy(['propietario'=>$user]);

            return $this->render('dashboard/index.html.twig', [
                'mascotas' => $mascotas,
                'user' => $user->getUsername()
            ]);
        }else{
            return $this->redirectToRoute('crear_persona');
        }
    }
}
```


- Con las mascotas, queremos guardar nuevas mascotas y ver una individual.

```
class MascotaController extends AbstractController
{
    /**
     * @Route("/guardar/mascota", name="guardar-mascota")
     */
    public function index(Request $request): Response
    {
        $mascota = new Mascota();
        $form = $this->createForm(MascotaType::class, $mascota);
        $form->handleRequest($request);

        if($form->isSubmitted() && $form->isValid()){
            $persona = $this->getUser();
            $mascota->setPropietario($persona);
            $em = $this->getDoctrine()->getManager();
            $em->persist($mascota);
            $em->flush();
            return $this->redirectToRoute("dashboard");
        }

        return $this->render('mascota/index.html.twig', [
            'formulario' => $form->createView(),
            'persona' => $this->getUser()
        ]);
    }

    /**
     * @Route("/mascota/{id}", name="ver-mascota")
     */
    public function verMascota($id){
        $em = $this->getDoctrine()->getManager();
        $mascota = $em->getRepository( Mascota::class)->find($id);
        return $this->render('mascota/verMascota.html.twig', [
            'mascota' => $mascota
        ]);
    }
}
```

- Creación de usuarios.

```
3 namespace App\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6 use Symfony\Component\HttpFoundation\Response;
7 use Symfony\Component\Routing\Annotation\Route;
8 use App\Entity\Persona;
9 use App\Form\PersonaType;
10 use Symfony\Component\HttpFoundation\Request;
11 use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
12
13
14 class PersonaController extends AbstractController
15 {
16     /**
17      * @Route("/crear-usuario", name="crear_persona")
18      */
19     public function index(Request $request,
20         UserPasswordEncoderInterface $passwordEncoder): Response
21     {
22         $persona = new Persona();
23         $form = $this->createForm(PersonaType::class, $persona);
24         $form->handleRequest($request);
25
26         if($form->isSubmitted() && $form->isValid()){
27             $em = $this->getDoctrine()->getManager();
28             $persona->setPassword($passwordEncoder->encodePassword(
29                 $persona,
30                 $form['password']->getData()
31             ));
32
33             $em->persist($persona);
34             $em->flush();
35             return $this->redirectToRoute("app_login");
36         }
37
38         return $this->render('persona/index.html.twig', [
39             'formulario' => $form->createView()
40         ]);
41     }
42 }
```

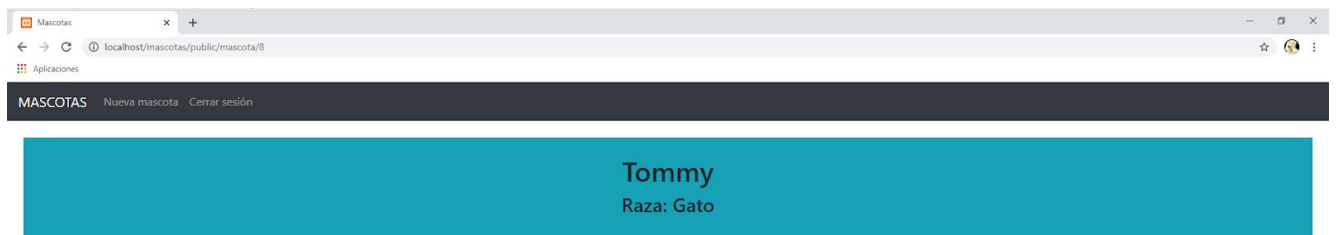
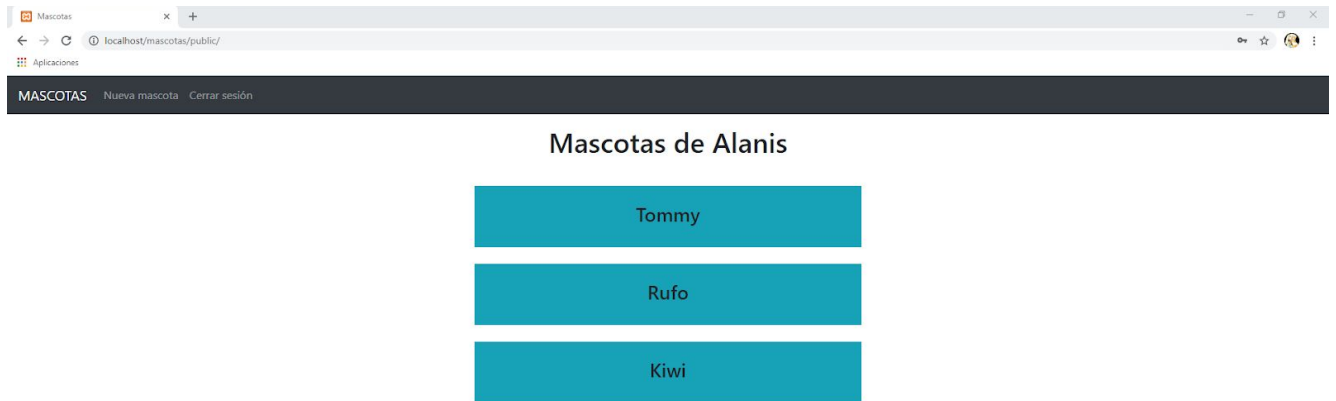
En la barra de navegación, ponemos enlaces que nos lleven al dashboard, a crear una nueva mascota y para cerrar sesión.

```
<body>
  {% block body %}
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
      <a class="navbar-brand" href="{{path('dashboard')}}">MASCOTAS</a>
      <ul class="navbar-nav mr-auto">
        <li class="nav-item">
          <a class="nav-link" href="{{path('guardar-mascota')}}">Nueva mascota</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="{{path('app_logout')}}">Cerrar sesión</a>
        </li>
      </ul>
    </nav>
  {% endblock %}
</body>
```

Al hacer esto y aplicarle unos pocos estilos, terminaríamos con el siguiente programa:

Formulario 'Crear cuenta' con campos para 'Username' y 'Password'. Incluye un botón azul 'Registrar' y un botón gris 'Ya tengo cuenta'.

Formulario 'Iniciar sesión' con campos para 'Username' (conteniendo 'Alanis') y 'Password' (conteniendo '****'). Incluye un botón azul 'Iniciar sesión'.



Bibliografía

- [¿Qué es symfony?](#)
- [Symfony 5: Novedades e Instalación desde cero y paso a paso](#)
- [Controllers en Symfony](#)
- [Symfony: tutorial 15: los tests automáticos, funcionales y unitarios](#)