Luciano Romero
UNRaf – Departamento de Sistemas
Año 2021

CRUD BASICO DE SYMFONY

Índice:

Contenido

Luciano Romero
UNRaf – Departamento de Sistemas
Año 2021

Introducción:

Aclaro que esto se hace luego de haber creado un proyecto y haberle creado su base de datos. En caso de no haberlo hecho revisar la documentación de instalación de Symfony y en ella, solo crear el proyecto y crear la base de datos.

Otra cosa importante, es que voy a pegar imágenes forzándolos a escribir línea por línea para lograr un acostumbramiento a la sintaxis que utilizamos para programar.

Propósito:

Aprender de manera rápida y eficaz como realizar un *CRUD*(create, read, update, delete) o *ABM* (alta, baja, modificación), de cualquiera de las dos maneras se lo conoce. Esto nos servirá para cargar datos de una *ENTIDAD* a una base de datos a través de un *FORMULARIO* y que será controlada por un *CONTROLADOR*, que a su vez está diseñado en un *TEMPLATE*.

Explicación:

Primero partir por lo más importante, que es cada cosa que se menciona en el párrafo anterior:

- Entidad: Como estamos programando orientado a objetos, una entidad es como un objeto al que nosotros les daremos *ATRIBUTOS* que definen como es dicha entidad. Cada vez que se modifique una entidad (agregarle datos, borrarle, etc.) hay que actualizar la base de datos. *php bin/console doctrine:schema:update --force*

  Ejemplo:
  - Entidad: Perro
  - Atributos: color, raza, altura, largo, etc. (se pueden crear la cantidad que se desee o que el sistema requiera).
- Formulario: Se crean basados en una entidad, es decir que para crear un formulario necesitamos al menos una entidad previa. Es el encargado de controlar los atributos de la entidad y donde les podemos otorgar ciertas propiedades para validaciones.

- Controlador: Encargado de realizar la lógica del *CRUD*. Acá es donde implementamos el código necesario para realizar dichas acciones.
- Base de datos: Es la que alberga los datos de las entidades y sus atributos.
- Template: Es el encargado de la parte visual (frontend).

Sintaxis:

Se enumerarán los siguientes pasos a tener en cuenta y respetar si o si para programar nuestros sistemas.

- Entidades/Formularios/Controladores: Su primera letra SIEMPRE en mayúscula, ejemplo, Persona (Entidad), PersonaType (Formulario), PersonaController(Controlador).
- Atributos de las entidades: SIEMPRE en minúsculas, por ejemplo, nombre, apellido. En caso de ser más de una palabra seguir esta forma, por ejemplo, fechaNacimiento, documentoIdentidad.
- Programar en INGLÉS: si bien nuestros sistemas no están programados en inglés, la idea a futuro es hacerlo de esa manera, si está aprendiendo a programar no es necesario.
- NO perder tiempo con el frontend, es decir la parte estética y visual del sistema, priorizar la lógica del sistema(backend).

Funcionamiento:

Ya comprendido lo anterior empezaremos a crear un *CRUD* básico para aprender sobre Symfony.

Lo primero que debemos hacer es crear una nueva entidad en la terminal de nuestro Visual Studio Code (IDE que utilizaremos para programar si o si). Podemos abrirla de manera rápida con *"ctrl+shift+ñ"* simultáneamente o clickeando en la barra superior de arriba *"terminal/nueva terminal"*, allí escribimos lo siguiente.

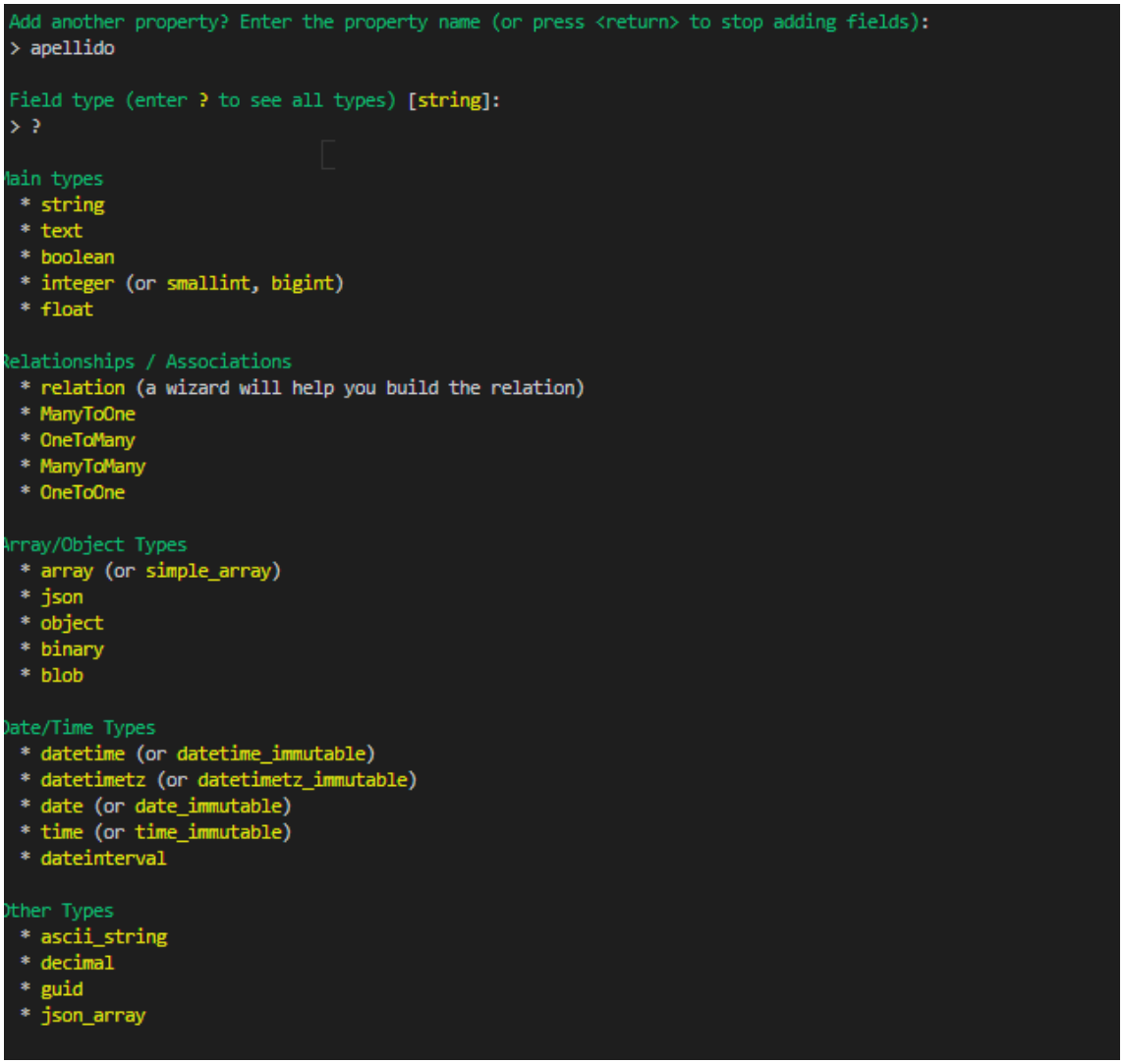
php bin/console make:entity

```
Class name of the entity to create or update (e.g. OrangeGnome):
> Persona

created: src/Entity/Persona.php
created: src/Repository/PersonaRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

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

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

Field length [255]:
> 255

Can this field be null in the database (nullable) (yes/no) [no]:
> no

updated: src/Entity/Persona.php
```

Lo que está entre corchetes son valores por defecto, sino escribimos nada, por defecto tomara esos.

NUNCA CREAR UNA PROPIEDAD QUE SE LLAME ID, SYMFONY LA CREA AUTOMATICAMENTE.

En los tipos de datos podemos apretar *" ? "* para ver todos los tipos de datos disponibles

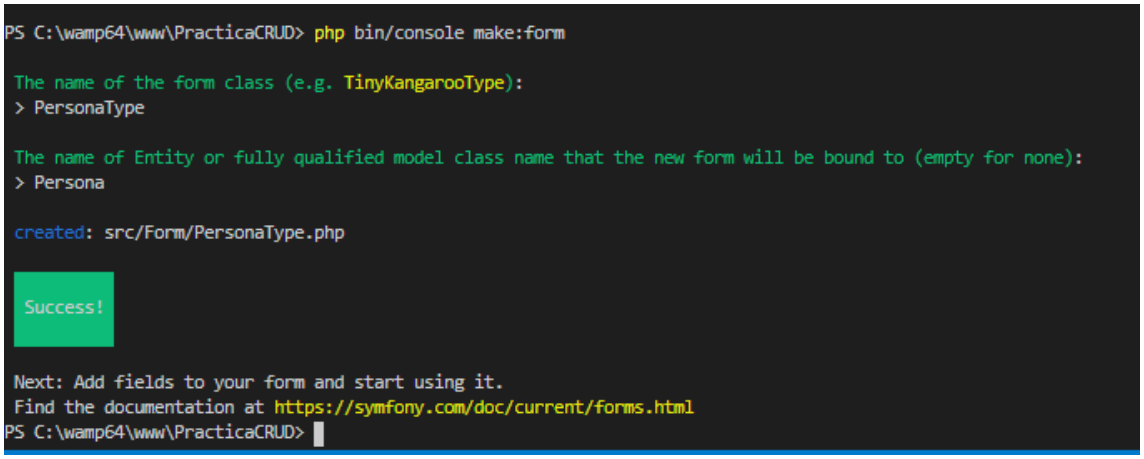


Elegir el que se desee y crear cuantos atributos desee, mayormente se utilizara *"STRING"* por el hecho de que luego es mas facil trabajar con este tipo de dato.Una vez creado todo, actualizar la base de datos.

php bin/console doctrine:schema:update –force

Paso siguiente: Crear el formulario de esa entidad

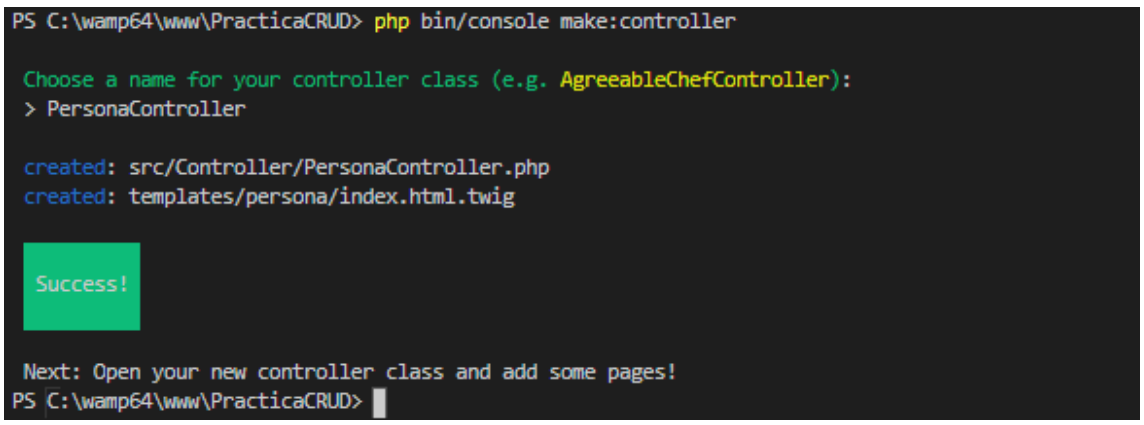
php bin/console make:form



Es importante agregarle el TYPE al nombre del formulario, asi sea PERRO, deberia llamarse PerroType.

Paso siguiente: Crear el controlador para esta entidad.

php bin/console make:controller

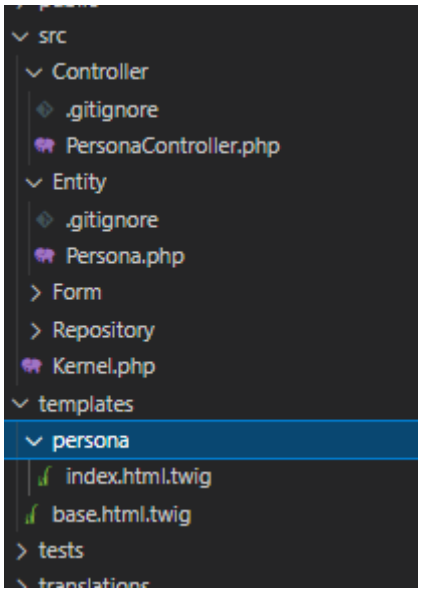


Es muy importante siempre seguir un hilo de nombres claros, es decir, si vamos a tratar sobre una entidad Persona, nuestro formulario y controlador que sean adecuados a esa

entidad, no usemos nombres genericos. Mal no está solamente es que a futuro va a tener muchas mas entidades,controladores y formularios, entonces se va a prestar a confusion.

Como vemos, el controlador nos creo un template junto con el, que vendria a ser, controlador (backend), template (frontend).

Dentro de la carpeta template, se encuentra una carpeta persona que alli es donde crearemos todos los “html.twig” para la parte visual. Podemos crear cuantos templates querramos, si vamos a hablar de persona dentro de persona y sino en otra carpeta (que tambien podemos crear las que queramos).



Una aclaracion antes de empezar a programar es que si nosotros queremos agregar mas datos a nuestra entidad lo podemos hacer tirando el mismo comando para crear una entidad, en caso de no recordarlo buscar mas arriba, y cuando nos pida un nombre de entidad, tipeamos Persona. Nos va a decir que ya existe y que agreguemos atributos.

Comienzo de CRUD o ABM:

Voy a dividirlo en las 4 partes correspondientes, *CREAR*, *LEER* o *VER*, *MODIFICAR*, *ELIMINAR*.

Nos paramos en PersonaController y vamos a ver lo siguiente:

```php
<?php

namespace App\Controller;

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

class PersonaController extends AbstractController
{
    /**
     * @Route("/persona", name="persona")
     */
    public function index(): Response
    {
        return $this->render('persona/index.html.twig', [
            'controller_name' => 'PersonaController',
        ]);
    }
}
```

Para poner en contexto de que es todo esto, lo voy a hacer en partes breves:

- Las primeras dos oraciones, van siempre, la primera es para inicializar php *(<? php*) y la segunda "*namespace…*" para indicar que esto es un controlador.
- Todas las oraciones que se encuentren por debajo del namespace y por encima del class, son *LIBRERIAS* o datos que necesitemos *IMPORTAR*, como veremos mas adelante(son faciles de identificar por la palabra USE delante).
- Todo el codigo que vayamos a escribir en este controlador ira dentro de las llaves "{}" de la clase PersonaController. Por cierto una clase es como un plano de una casa, es donde se encuentra programada la logica para crear posteriormente objetos (en nuestro caso cargar personas al sistema).
- Funciones: Como se ve en la captura, las funciones son esctructuras de codigo que cumplen con ciertos objetivos. Arriba de ellas veremos un texto en verde que nos indican dos cosas:
    - Route: Es la ruta a la que accedemos en el navegador.
    - Name: Es como vamos a llamar a este controlador mediante el template.

Mas adelante esta breve informacion se complementa con contenido y se comprendera de una mejor manera.

Registrar Persona (crear):

Lo primero que debemos hacer es pararnos en la seccion de arriba, donde se encuentran los USE y pegamos estas dos lineas de codigo:

use App\Entity\Persona;

use App\Form\PersonaType;

Como se mencionaba, estos use son librerias o informacion recopilada de otras partes del sistema, este es el caso. Necesitamos recopilar los datos de la ENTIDAD Persona y los datos del FORM Persona.

Ahora lo siguiente es acomodar la sintaxis de la funcion de esta manera:

```
/**
 * @Route("/registrarPersona", name="registrarPersona")
 */
public function registrarPersona()
{

}
```

Se modifican los nombres por lo mencionado paginas arriba, de mantener una misma logica respecto a los nombres, si en este caso vamos a registrar personas, lo ideal es que todos tengan el nombre de esa manera.

Dentro de la función escribimos lo siguiente:

```php
/**
 * @Route("/registrarPersona", name="registrarPersona")
 */
public function registrarPersona(Request $request)
{
    $persona = new Persona();

    $formulario = $this -> createForm(PersonaType::class, $persona);
    $formulario -> handleRequest($request);


}
```

Antes de empezar una aclaracion importante es que PHP crea las variables de la siguiente manera:

$algo

Siempre se utiliza un signo $ (pesos) delante de la palabra de nuestra variable.

Y otra aclaracion mas es que TODAS las sentencias que escribamos terminan con un punto y coma.

- La primer linea de la funcion es una INSTANCIA DE UNA CLASE, con esto estamos diciendo que la entidad Persona la vamos a manipular a traves de la variable $persona.
- En las siguientes lineas estamos creando un formulario para esta entidad utilizando el que hemos creado PersonaType y tambien referenciando a $persona ya que vamos a manejar sus atributos. Y debajo el "*handle request*" controla y maneja las solicitudes que haremos con este formulario.
- Si observamos bien ahora la funcion registrarPersona tiene algo escrito entre parentesis, estos se llaman PARAMETROS los cuales se reciben para ser utilizados. Probablmente ahora nos este indicando un error el request pero para solucionarlo solamente debemos IMPORTAR una LIBRERÍA nueva. Como ya sabemos donde van ubicadas las librerias, la importamos de esta manera *use Symfony\Component\HttpFoundation\Request;*

Ahora debemos hacer lo siguiente, ya es un poco mas complejo pero se ira comprendiendo a medida que se avance.

--imagen en la siguiente página--

```
if($formulario -> isSubmitted() && $formulario -> isValid()){
    $entityManager = $this -> getDoctrine() -> getManager();
    $entityManager -> persist($persona);
    $entityManager -> flush();

    $this -> addFlash('correcto', 'La persona se registró correctamente');
    return $this -> redirectToRoute('verPersonas');
}

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

- Verificamos con un IF(sentencia condicional, en caso de no conocerla aprender sobre ella en YouTube, pero en resumen determina un caso u otro, dependiendo una condición) si el formulario se envió(*isSubmitted*) y es válido (*isValid*).
- Las 3 lineas del manager hacen lo siguiente:
    - Inicializamos el manager (SIEMPRE se hace de esta manera) es el encargado de manejar las acciones con nuestros datos hacia la DB(base de datos).
    - Carga los datos de la variable persona en la base(*persist*).
    - Actualiza los campos creados de cada atributo a la DB.¡OJO!, NO CONFUNDIR CON ACTUALIZAR LA BASE DE DATOS DESDE LA TERMINAL. Esto solo actualiza los campos de cada atributo, un campo de un atributo es lo siguiente:
        - Atributo: nombre
        - Campo: Luciano.
- Las ultimas dos son:
    - *Addflash:* un cartel con un mensaje (este mensaje se puede modificar a gusto). Atención acá también, el mensaje modificable es la segunda parte

"la persona se….", la parte de "correcto" se deja asi, se explicará mas adelante.

- Y la linea debajo del *addflash*, nos retorna (*return*) hacia otro controlador.

Todo esto se realiza siempre y cuando EL FORMULARIO SE HAYA ENVIADO Y SEA VALIDO, RECORDAR ESO.

Como se observa en la imagen, la llave del IF termina ahí, y hay una ultima sentencia, esta es la que nos redirecciona al template de esta funcion, que es donde se maneja el frontend(parte visual del sistema).

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

En nuestra carpeta template, dentro de persona, tenemos un archivo llamado *"index.html.twig"*, lo renombramos a como lo vemos arriba en color naranja.

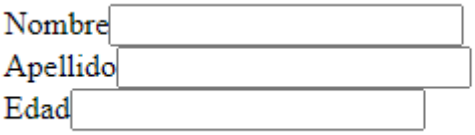
Nos posicionamos en ese template, borramos todo el contenido y escribimos lo siguiente

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

{% block title %}Registrar Persona{% endblock %}

{% block body %}
    {{parent()}}
    {{form(formulario)}}
{% endblock %}
```

Con la linea escrita dentro de las llaves BODY, le indicamos a este template que nos dibuje el formulario con los atributos de la entidad.

Nos dirigmos a una pestaña nueva en Chrome(o el navegador que use) y mediante este enlace veremos el formulario http://localhost/PracticaCRUD/public/index.php/registrarPersona

PracticaCRUD es el nombre de MI proyecto, reemplazarlo por el suyo, y registrarPersona es el NAME del controlador que tenemos en PersonaController.



Quedaría algo como eso (si tiene mas atributos apareceran la cantidad que haya creado).

Por ahora todo claro, pero falta un botón para cargar estos datos, acá es donde el FORM entra en juego, nos vamos a la carpeta *Form/PersonaType* y nos paramos alli:

```php
namespace App\Form;

use App\Entity\Persona;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class PersonaType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            ->add('apellido')
            ->add('edad')
        ;
    }

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

Dentro de *$builder*, veremos TODOS los atributos de la clase Persona. Ahora falta agregar un boton para enviar estos datos, lo agregamos de la siguiente manera, debajo del ultimo atributo creado, en mi caso debajo de *->add('edad'):*

->add('Cargar', SubmitType::class)

Importante escribirlo a mano y no copiar y pegar, esto es para que se nos *IMPORTE AUTOMATICAMENTE* la librería *SUBMIT*. En caso de que esta sentencia esté tirando un error o aparezca en rojo y no se haya importado, en la seccion *USE*, agregamos lo

siguiente:

use Symfony\Component\Form\Extension\Core\Type\SubmitType;

Recargamos la página que teniamos abierta y ahora veremos el boton.

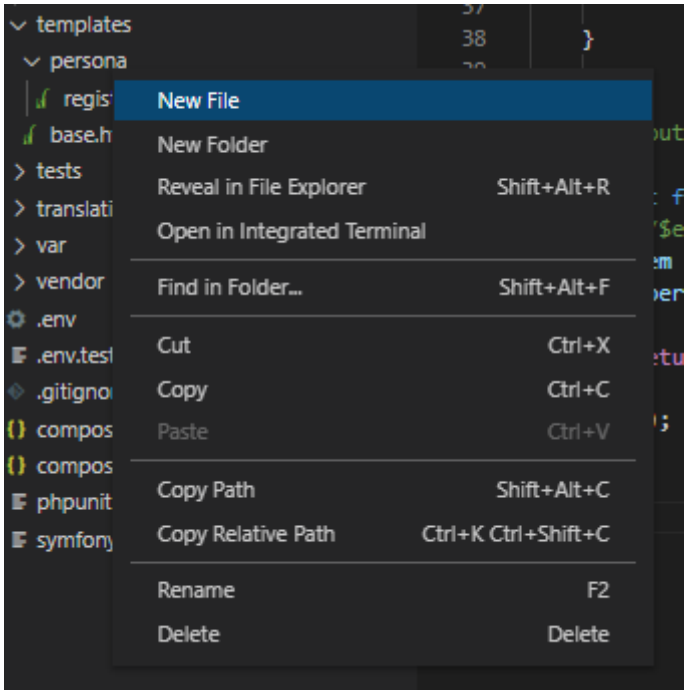
Antes de cargar algun dato, debemos crear nuestro segundo controlador que es aquel que nos va a mostrar los datos cargados.

Ver o Leer:

```php
/**
 * @Route("/verPersonas", name="verPersonas")
 */
public function verPersonas(){
    //$em o $entityManager es lo mismo, le puede dar el nombre que usted desee, tambien puede llamarse $entityManager
    $em = $this -> getDoctrine() ->getManager();
    $personas = $em -> getRepository(Persona::class)->findAll();

    return $this -> render('persona/verPersonas.html.twig', [
        'personas' => $personas
    ]);
}
```

Lo que esta en verde debajo del nombre de la funcion es un comentario, que les deje, no hace falta que lo copien.

En esta funcion podemos ver que nos esta retornando *(return)* hacia otro *TEMPLATE*, llamado *verPersonas.html.twig* que NO tenemos creado, crearlo es facil, vamos a la carpeta personas, click derecho crear archivo nuevo y lo creamos con ese nombre.



Una vez creado, nos paramos en ese template y copiamos lo siguiente:

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

{% block title %}Ver Personas cargadas{% endblock %}

{% block body %}
{{parent()}}
    <table class="table table-borderless">
        <thead>
          <tr style="color:#0F9FA8; background-color: #EDEDED;">
            <th scope="col">Nombre</th>
            <th scope="col">Apellido</th>
            <th scope="col">Modificar</th>
            <th scope="col">Eliminar</th>
          </tr>
        </thead>
        <tbody>
            {% for persona in personas %}
                <tr>
                    <td>{{ persona.nombre }}</td>
                    <td>{{ persona.apellido }}</td>
                    <td><a href="#">Modificar</a></td>
                    <td><a href="#">Eliminar</a></td>
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

Por lo que nos quedaría una tabla de la siguiente manera:

Nombre	Apellido	Modificar	Eliminar
Luciano	Romero	Modificar	Eliminar
Luciano	Romero	Modificar	Eliminar

Es muy probable que no la vean de esta manera todavía, la diferencia es que yo ya tengo incluido *BOOTSTRAP*, ver la explicacion de abajo para implementarlo.

Template BASE:

Como ultimo paso, dentro de la carpeta *TEMPLATE*, tenemos un archivo que se llama *"base.html.twig",* aca es donde importaremos librerias a utilizar como Bootstrap, Jquery,FontAwesome ya que es *"TEMPLATE PADRE".* Es muy util ya que nos ahorra importar este tipo de librerias en cada html.twig que hagamos, directamente HEREDAR de el y listo. Si prestamos atencion en la imagen de arriba hay una linea que es la que se utiliza para esta herencia: *{{parent()}}.*

Debajo voy a dejar todo el codigo del *BASE* y este SI lo copian y pegan.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>{% block title %}Base{% endblock %}</title>

    {% block stylesheets %}
      <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css"
integrity="sha384-
```

B0vP5xmATw1+K9KRQjQERJvTumQW0nPEzvF6L/Z6nronJ3oUOFUFpCjEUQouq2+l" crossorigin="anonymous">

{% endblock %}

{% block javascripts %}

<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUew+OrCXaRkfj" crossorigin="anonymous"></script>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/js/bootstrap.bundle.min.js" integrity="sha384-Piv4xVNRyMGpqkS2by6br4gNJ7DXjqk09RmUpJ8jgGtD7zP9yug3goQfGII0yAns" crossorigin="anonymous"></script>

{% endblock %}

</head>

<body>

{% block body %}

<nav class="navbar navbar-expand-lg navbar-light bg-light">

Navbar

<button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">

**

</button>

* *

```
            <ul class="navbar-nav mr-auto">
              <li class="nav-item dropdown">
                <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
                Personas
                </a>
                <div class="dropdown-menu" aria-labelledby="navbarDropdown">

                    <a class="dropdown-item" href="#">Registrar persona</a>
                    <div class="dropdown-divider"></div>
                    <a class="dropdown-item" href="#">Ver Personas</a>
                </div>
              </li>
            </ul>
          </div>
        </nav>
        {% for message in app.flashes('correcto') %}
          <div class="container alert alert-success">
            <a href="#" class="close" data-dismiss="alert" aria-label="close">&times;</a>
            {{ message }}
          </div>
        {% endfor %}
    {% endblock %}
```

```
    </body>

</html>
```

Con esto ya podemos crear y ver a gusto la cantidad de personas registradas modificando la estructura del base, que lo haremos a continuación.

Como se mencionaba puntos anteriores respecto a los addFlash, acá en el base es donde entran en juego, ya que antes de cerrar la etiqueta body los incluimos. También esta el motivo del porqué les hice dejar la palabra "correcto", ya que es la que llama a esa funcion que dice

```
for message in app.flashes('correcto')
```

Ahora es el momento de modificar esa estructura dentro de nuestro base. Nos dirigimos hacia donde dice *Registrar Persona y Ver Persona* en este codigo, vemos que estan dentro de una etiqueta <a>. Dentro del "href" que ahora tiene un "#" lo reemplazamos por lo siguiente:

- En registrar persona: *href = "{{path('registrarPersona')}}"*
- En ver persona: *href = "{{path('verPersonas')}}"*

A diferencia de cuando usabamos *HTML* puro, lo que se escribe dentro del *HREF* ahora usando *HTML.TWIG* es diferente, debemos utilizar la sentencia *"path"* para que nos redirija hacia el controlador que queremos mediante un click en esa etiqueta. Otro dato importante es que lo que escribimos entre los parentesis, es el *NAME* del controlador, es decir, lo que esta en verde arriba de cada funcion.

Con esto realizado hasta el momento, deberiamos tener lo siguiente:

¡IMPORTANTE NO SEGUIR SI TIENE ALGÚN ERROR O NO TIENE ALGO DE LA LISTA QUE SE ENCUENTRA DEBAJO!

- Una entidad: Persona, con 3 atributos, nombre, apellido, edad (en mi caso).
- Un form: PersonaType (en mi caso).
- Dos controladores: altaPersonas y verPersonas.
- Dos templates: uno para registrarPersonas y otro para verPersonas, ambos dentro de la carpeta "persona".
- Template BASE: con el Navbar(barra de navegación) y Bootstrap incluido. Nos daremos cuenta sino esta incluido porque la barra de navegacion o la tabla, no se veran con un estilo diferente.
- Una barra de navegacion superior con un boton que nos lleva a REGISTRAR PERSONAS y a VER PERSONAS.

Modificar:

Nos dirigimos nuevamente a nuestro controlador PersonaController, y alli crearemos una nueva función con el mismo criterio que las dos anteriores.

```php
/**
 * @Route("/modificarPersona/{id}", name="modificarPersona")
 */
public function modificarPersona(Request $request, $id){
    $em = $this -> getDoctrine() ->getManager();

    $persona = $em -> getRepository(Persona::class) -> find($id);

    $formulario = $this -> createForm(PersonaType::class, $persona);
    $formulario -> handleRequest($request);

    if($formulario -> isSubmitted() && $formulario -> isValid()){

        $em -> persist($persona);
        $em -> flush();

        $this -> addFlash('correcto', 'La persona se modificó correctamente');
        return $this -> redirectToRoute('verPersonas');
    }
    return $this -> render('persona/modificarPersona.html.twig', [
        'formulario' => $formulario -> createView()
    ]);
}
```

La funcion contendra todo esto, debajo voy a explicar aspectos particulares que la diferencian de crear una nueva persona.

Aspectos a tener en cuenta:

- Como podemos ver, en la estructura en verde arriba del nombre de la funcion, tenemos el "Route" con algo diferente a lo que veniamos haciendo, tiene un id entre llaves */{id}*. Esto es porque la funcion recibe un ID.
    - Un id es un identificador UNICO que posee cada persona que vayamos a cargar en la entidad. Por ejemplo si tenemos dos personas cargadas, por mas que tengan datos iguales, tendran un ID diferente y asi sucesivamente con todos los datos que deseemos cargar.
        - *Luciano Romero 23 (id = 1)*
        - *Luciano Romero 23(id = 2)*
- Respecto al ID, no sólo se escribe en el Route sino que tambien llega como parámetro entre paréntesis a la derecha del nombre de la función.
- Y lo ultimo diferente a como cargamos una persona a base de datos son las dos primeras líneas de la función.
    - Inicializamos el manager: *$em = $this .....*
    - Esta sentencia es importante ya que en la variable $persona, mediante el $em (manager) está buscando en TODA LA ENTIDAD, una persona con el ID que llega por parámetro.

Lo que resta de la funcion es igual a cuando cargamos o damos de alta una persona, pero no debemos olvidarnos que está retornando a un template diferente, asi que lo que debemos hacer ahora, es crear un template para esta nueva funcion.

Dentro de la carpeta *templates/persona* creamos un archivo nuevo llamado, *modificarPersona.html.twig* y allí escribimos lo siguiente.

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

{% block title %}Modificar Persona{% endblock %}

{% block body %}
    {{parent()}}
    {{form(formulario)}}
    <a href=  "{{path('verPersonas')}}">Volver</a>
{% endblock %}
```

Verán que es igual al template de registroPersona.html.twig (salvo el title y una etiqueta <a> que simula un boton de volver atrás), pero a pesar de que sean iguales necesitamos uno para cada controlador, tanto para cargar como para modificar.

Antes de probar que esto realmente funcione, nos dirigimos al template verPersona.html.twig, también dentro de la carpeta templates/persona.

Dentro de la etiqueta <tbody> tenemos dos etiquetas <a> con un href = "#". Acá vamos a repetir el proceso que utilizamos cuando modificamos el BASE. Dentro del "href" de la etiqueta <a> que tiene el texto "Modificar" , copiaremos lo siguiente:

```
{% for persona in personas %}
    <tr>
        <td>{{ persona.nombre }}</td>
        <td>{{ persona.apellido }}</td>
        <td><a href="{{path('modificarPersona', {'id': persona.id})}}">Modificar</a></td>
        <td><a href="#">Eliminar</a></td>
    </tr>
{% endfor %}
```

Prestar atención a la estructura para evitar errores.

Eliminar:

Por último, pero no por eso menos importante, tenemos la parte final de eliminar este registro que acabamos de crear, o cuantos querramos borrar. Para ello necesitamos volver a PersonaController y crear una nueva funcion.

```php
/**
 * @Route("/eliminarPersona/{id}", name="eliminarPersona")
 */

public function eliminarPersona($id){
    $em = $this -> getDoctrine() -> getManager();
    $persona = $em -> getRepository(Persona::class) -> find($id);

    $em ->remove($persona);
    $em -> flush();

    $this -> addFlash('correcto', 'La persona se eliminó correctamente');
    return $this->redirectToRoute("verPersonas");
}
```

Es bastante sencilla y sigue la logica de modificar una persona, con esto me refiero a que recibe un ID (en el route y por parámetro) y en base a el, recupera la persona que hayamos clickeado para eliminar, la diferencia es que utilizamos una sentencia nueva con el manager llamada *"remove"*, la cual nos elimina esta persona cargada.

Ahora vamos a la carpeta *templates/persona* y nos paramos nuevamente en *verPersonas.html.twig*, acá la sentencia es un poco diferente ya que, si dentro del href, llamamos directamente a este controlador, va a borrar el campo directamente sin consultarlo.

```twig
<tbody>
    {% for persona in personas %}
        <tr>
            <td>{{ persona.nombre }}</td>
            <td>{{ persona.apellido }}</td>
            <td><a href="{{path('modificarPersona', {'id': persona.id})}}">Modificar</a></td>
            <td><a href="{{path('eliminarPersona', {'id': persona.id})}}">Eliminar</a></td>

        </tr>
    {% endfor %}
</tbody>
```

Funcionar funciona, lo único preocupante es que no nos consulta antes de eliminarlo en caso de haber clickeado por error y no es una buena práctica. Para lograr que nos consulte si queremos eliminarlo, debemos reemplazar la sentencia de eliminar por una diferente.

```twig
{% for persona in personas %}
    <tr>
        <td>{{ persona.nombre }}</td>
        <td>{{ persona.apellido }}</td>
        <td><a href="{{path('modificarPersona', {'id': persona.id})}}">Modificar</a></td>
        <td><a onclick="DeleteFunction({{persona.id}})">Eliminar</a></td>
    </tr>
{% endfor %}
```

Como podemos observar está llamando a una funcion mediante el "onclick", esta "funcion" a la que llama es de JavaScript, la cual crearemos a continuación.

En la izquierda de nuestro visual studio, donde tenemos todas las carpetas, nos dirigimos a public y alli crearemos una nueva carpeta llamada *"js" o "javascripts"* como desee. Dentro de ella crearemos un archivo llamado, *eliminarPersona.js* . Dentro de este archivo copiaremos esto:

```
function DeleteFunction(id) {

    if (confirm("¡Atencion!¿Está seguro/a que desea eliminar?")) {
        window.location.href="http://localhost/PracticaCRUD/public/index.php/eliminarPersona/"+id+"";
    }

}
```

La direccion que yo escribo, es en mi caso de ejemplo, se debe reemplazar por sus datos, por ejemplo, donde dice PracticaCRUD, deberá poner el nombre de SU proyecto, lo demas queda igual.

Con esto realizado, volvemos al template de verPersonas.html.twig e importaremos este archivo JS a nuestro template. Lo haremos de la siguiente manera:

```
{% block title %}Ver Personas cargadas{% endblock %}

{% block javascripts %}
    {{parent()}}
    <script src = "{{asset('js/eliminarPersona.js')}}"></script>
{% endblock %}

{% block body %}
{{parent()}}
```

Como podemos observar, lo estamos haciendo arriba del "block body". Con importar me refiero a que esta parte haga uso del archivo JS que creamos previamente. Tambien hay algo particular en esta sentencia que dentro de "src" se escribe de una manera particular debido a TWIG, utilizando *"{{asset()}}.*

Ahora probaremos todo el ciclo del CRUD, desde nuestro navegador.

CRUD o ABM:

1. *Registrar una Persona.*
2. *Verla reflejada en la tabla.*
3. *Modificarle algun dato.*

4. *Eliminarla.*

Con todos los pasos explicados en esta documentación, debería poder hacer sin ningún problema, el ciclo natural de un CRUD o ABM.

Debajo voy a dejar el código de los controladores, por si llegan a tener algún problema, aunque no recomiendo venir hasta este apartado ya que la idea es que los escriban ustedes.

```
  /**
     * @Route("/registrarPersona", name="registrarPersona")
     */
    public function registrarPersona(Request $request)
    {
        $persona = new Persona();

        $formulario = $this -> createForm(PersonaType::class, $persona);
        $formulario -> handleRequest($request);

        if($formulario -> isSubmitted() && $formulario -> isValid()){
            $entityManager = $this -> getDoctrine() -> getManager();
            $entityManager -> persist($persona);
            $entityManager -> flush();

            $this -
> addFlash('correcto', 'La persona se registró correctamente');
            return $this -> redirectToRoute('verPersonas');
        }
```

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

    }

     /**
     * @Route("/verPersonas", name="verPersonas")
     */
    public function verPersonas(){
        //$em o $entityManager es lo mismo, le puede dar el nombre que us
ted desee, tambien puede llamarse $entityManager
        $em = $this -> getDoctrine() ->getManager();
        $personas = $em -> getRepository(Persona::class)->findAll();

        return $this -> render('persona/verPersonas.html.twig', [
            'personas' => $personas
        ]);
    }

    /**
     * @Route("/modificarPersona/{id}", name="modificarPersona")
     */
    public function modificarPersona(Request $request, $id){
        $em = $this -> getDoctrine() ->getManager();

        $persona = $em -> getRepository(Persona::class) -> find($id);

        $formulario = $this -> createForm(PersonaType::class, $persona);
        $formulario -> handleRequest($request);

        if($formulario -> isSubmitted() && $formulario -> isValid()){

            $em -> persist($persona);
            $em -> flush();

            $this -
> addFlash('correcto', 'La persona se modificó correctamente');
            return $this -> redirectToRoute('verPersonas');
        }
        return $this -> render('persona/modificarPersona.html.twig', [
            'formulario' => $formulario -> createView()
        ]);
    }


    /**
```

```
     * @Route("/eliminarPersona/{id}", name="eliminarPersona")
     */

    public function eliminarPersona($id){
        $em = $this -> getDoctrine() -> getManager();
        $persona = $em -> getRepository(Persona::class) -> find($id);

        $em ->remove($persona);
        $em -> flush();

        $this -
> addFlash('correcto', 'La persona se eliminó correctamente');
        return $this->redirectToRoute("verPersonas");
    }
```