# 1. Instalar composer

# 2. Instalar Symfony

DESCARGAR Ejecutable sympfnu

<http://symfony.com/installer>

BAJAR EL ARCHIVO PHAR. Y USAR PARA INSTALAR:

EJ

Php symfony.phar new new my\_project\_name 3.4

la recomendación en S3.4 y S4 es de **No usar Bundle**

**utilizar el App Bundle que se tiene para el proyecto.**

**Utilizar Routing por anotación**

**Templates plantillas demo con Twig**

**Usar Bootstrap**

**Imágenes free** [**https://unsplash.com/**](https://unsplash.com/)

**Rutas con parámetro**

**Ejemplo twig**

{{path('bares', {'ciudad': 'cal'})}}

**Controller:**

/\*\*

\* @Route("/bares/{ciudad}", name="bares")

\*/

Pasar un array al render de la vista

return $this->render('frontal/bares.html.twig', array("ciudad"=>$ciudad));

# LA BASE DE DATOS CON DOCTRINE

Se deben tener en cuenta los siguientes aspectos en mysql y en Sym

Configurar en parameters yml la base de datos

Mysql UTF 8 collatiion

Crear DB Con doctrine

php bin/console doctrine:database:create

Crear tablas

# Las Entity class

Para crear una entity se genera con el siguiente comando

php bin/console doctrine:generate:entity

1. Asignar nombre <bundle>:<entity>
2. Seleccionar metodo de formato de informacion adicional
3. El campo id se genera automaticamente\*
4. Se deberan introducir los campos, tipo de campo, tamaño
5. Al finalizar se crean dos archivos entity\tapa.php y repository\taaRepository.php

# Crear la DB usando doctrine

Para crear las tablas de la base de datos usando la consola con el siguiente comando:

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

# Empezar a leer la DB con DOCTRINE dentro del controlador

Lo primero es incorporar la primer línea de datos: en el controlador

Esta línea se pone en el action

Llamar al recurso

use AppBundle\Entity\Tapa;

// Capturar el repositorio de la tabla con la DB ,Tapa es la entidad q trae los datos

$repository = $this->getDoctrine()->getRepository(Product::class);

Cambiar Producto por el nombre de la entidad (Tapa) para este curso

Hacer un select de todo con

*// finds \*all\* products*

$tapas = $repository->findAll();

# Generacion del formulario en symfony

Crear un objeto de la entity.

$tapa = new Tapa();

Crear el constructor del formulario

$formBuilder = $this->createFormBuilder($tapa)

Agregar las clases que utiliza el formulario de Symfony

$formBuilder = $this->createFormBuilder($tapa);

Se puede iniciar cargando cada uno de las clases de tipos de datos en el formulario de la siguiente forma:

$tapa = new Tapa();

$formBuilder = $this->createFormBuilder($tapa);

$formBuilder->add('nombre', TextType::class);

$formBuilder->add('descripcion', TextareaType::class);

$formBuilder->add('fechaCreacion', DateType::class);

// para crear el boton SUBMIT

**$formBuilder->add('guardar', SubmitType::class, array('label'=>'Crear plato'));**

**$formBuilder->getForm();**

\*Nota: Agregar las clases de datos usadas en el encabezado del controlador

**use Symfony\Component\Form\Extension\Core\Type\TextType;**

**use Symfony\Component\Form\Extension\Core\Type\TextareaType;**

**use Symfony\Component\Form\Extension\Core\Type\DateType;**

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

Enviar a la vista el formulario renderizado

return $this->render('gestionPlatos/nuevoPlato.html.twig', array("form"=>$form->createView()));

Generar el formulario con el Framework

1CREAR carpeta de formulario en el bundle

AppBundle\Form

2 Crear archive TapaTYpe.php con los datos del formulario

namespace AppBundle\Form;

// Librerias para crear los formularios

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\FormBuilderInterface;

//Clases para los datos del Form

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

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

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

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

//Se crea una clase que cinstruira el formulario según los datos

class TapaType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder

->add('nombre', TextType::class)

->add('descripcion', TextareaType::class)

->add('guardar', SubmitType::class, array('label'=>'Crear plato'))

;

}

}

3 Modificar el controlador incluyendo el archivo Form/TapaType.php

use AppBundle\Form\TapaType;

4 En el controlador crear el formulario de la siguiente forma

//Creacion del formulario

//Crear nuevo objeto de tipo tapa para construir el formulario

$tapa = new Tapa();

//Construccion del formulario

$form = $this->createForm(TapaType::class, $tapa);

# Editor para los ingredientes

## En cmd:

composer require egeloen/ckeditor-bundle

agregar en el AppKernel en el segmento bundles

new Ivory\CKEditorBundle\IvoryCKEditorBundle(),

## Ejecutar en el CMD

php bin/console ckeditor:install

## Instalar en los assets

Php bin/console assets:install web

## Incorporar el Editor en el formulario

use Ivory\CKEditorBundle\Form\Type\CKEditorType;

Modificar la clase del formulario

->add('descripcion', CKEditorType::class)

Usar una barra de edición básica …

## Abrirl config.yml y agregar

ivory\_ck\_editor:

configs:

my\_config:

toolbar: basic

# Agregar manualmente una clase al formulario creado con Sym3

Se puede desglosar por partes individuales el formulario y a cada campo agregar una clase para su comoda presentación de los datos de la siguiente forma:

{{form\_start(form)}}

{{form\_errors(form)}}

<div class="control-group form-group">

{{form\_label(form.nombre)}}

{{form\_errors(form.nombre)}}

{{form\_widget(form.nombre)}}

</div>

<div class="control-group form-group">

{{form\_label(form.descripcion)}}

{{form\_errors(form.descripcion)}}

{{form\_widget(form.descripcion)}}

</div>

<div class="control-group form-group">

{{form\_label(form.ingredientes)}}

{{form\_errors(form.ingredientes)}}

{{form\_widget(form.ingredientes)}}

</div>

<div class="control-group form-group">

{{form\_label(form.top)}}

{{form\_errors(form.top)}}

{{form\_widget(form.top)}}

</div>

<div class="control-group form-group">

{{form\_widget(form.guardar)}}

</div>

{{form\_end(form)}}

# Agregar Automáticamente una clase al formulario creado con Sym3

Una de las formas de editar una clase al form de Symfony se debe realizar lo siguiente,

Revisar el theme del FORM de la ruta

[vendor/symfony/symfony/src/Symfony/Twig/Resources/](https://stackoverflow.com/questions/33160763/could-not-load-type-text-in-vendor-symfony-symfony-src-symfony-component-form)Views/Form

y Analizar el archivo:

*form\_div\_layout.html.twig*

\*El formulario debe quedar como se tenia inicialmente

{{form\_start(form)}}

{{form\_widget(form)}}

{{form\_end(form)}}

Crear una carpeta en Views/form

Crear archivo paltoform.html.twig y modificar su contenido

Usar como plantilla de modificación el siguiente demo

{%- block form\_row -%}

*<div>*

{{- form\_label(form) -}}

{{- form\_errors(form) -}}

{{- form\_widget(form) -}}

*</div>*

{%- endblock form\_row -%}

Modificar el DIV que encierra cada FORM por:

{%- block form\_row -%}

<div class="control-group form-group">

<div class="controls">

{{- form\_label(form) -}}

{{- form\_errors(form) -}}

{{- form\_widget(form) -}}

</div>

</div>

{%- endblock form\_row -%}

Agregar este tema de FORM a la plantilla de la vista de TWIG con editando lo siguiente en el nuevo producto.

{% form\_theme form 'form/platoForm.html.twig' %}

Para personalizar mas elementos del FORM se puede revisar el siguiente codifo y modificar

{%- block form\_widget\_simple -%}

{%- set type = type|default('text') -%}

<input type="{{ type }}" {{ block('widget\_attributes') }} {% if value is not empty %}value="{{ value }}" {% endif %}/>

{%- endblock form\_widget\_simple -%}

Agregar el DIV Con el Class form-coptrol

Modificar INPUT

{%- block form\_widget\_simple -%}

{%- set type = type|default('text') -%}

<input type="{{ type }}" class="form-control" {{ block('widget\_attributes') }} {% if value is not empty %}value="{{ value }}" {% endif %}/>

{%- endblock form\_widget\_simple -%}

Modificar TEXTAREA

{%- block textarea\_widget -%}

<textarea class="form-control" {{ block('widget\_attributes') }}>{{ value }}</textarea>

{%- endblock textarea\_widget -%}

# Insertar el input para subir archivos

AREGAR EL Type en el formulario

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

en el formulario agregar el Filetype

->add('foto', FileType::class)

Se puede agregar un evento JS a una etiqueta HTML de la siguiente manera:

->add('foto', FileType::class, array ('attr'=>array('onchange'=>'onChange(event)')))

Cerar una carpeta en web llamada /js y crear el archivo que llamara ese formulario

En el Twig del formulario incluir el siguiente bloque, el base Twig tiene un bloque con esta misma estructura

{% block javascripts %}

<script type="text/javascript" src="{{asset('js/nuevoPlato.js')}}"></script>

{% endblock %}

Crear un JS con los datos:

var tiposValidos =[

'image/jpg',

'image/jpeg',

'image/png',

]

function validarTIpos(file){

for(var i=0; i<tiposValidos.length;i++){

if(file.type===tiposValidos[i]){

return true

}

}

return false;

}

function onChange(event){

var file= event.target.files[0]

if(validarTIpos(file)){

var mini=document.getElementById('platoThumb');

mini.src=window.URL.createObjectURL();

} else {

}

console.log(file);

}

window.URL.createObjectURL();

Crea una URL de un objeto pasado por el parametro, en este caso como se psa un elemento tipo File, crea un (blob: objeto binario de tamaño grande) del objeto en la ventana que fue creada,

# Iniciar la carga udpatFile en el controlador

Utilizar el FileTYpe y sus métodos para obtener el nombre del archivo, el tipo de extensión demás datos

$fototype=$tapa->getFoto();

$fileName= $this->generateUniqueFileName().'.'. $fototype->guessExtension();

Crear una función única que modifique el nombre del archivo

*/\*\**

*\* @return string*

*\*/*

private function generateUniqueFileName()

{

*// md5() reduces the similarity of the file names generated by*

*// uniqid(), which is based on timestamps*

return md5(uniqid());

}

El controlador quedará asi:

public function nuevoPlatoAction(Request $request){

$tapa = new Tapa();

$form = $this->createForm(TapaType::class, $tapa);

$form->handleRequest($request);

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

$tapa=$form->getData();

$fototype=$tapa->getFoto();

$fileName= $this->generateUniqueFileName().'.'. $fototype->guessExtension();

$fototype->move($this->getParameter('plato\_img\_directory'),

$fileName

);

$tapa->setFoto($fileName);

$tapa->setFechaCreacion(new \DateTime());

$em = $this->getDoctrine()->getManager();

$em->persist($tapa);

$em->flush();

return $this->redirectToRoute('plato', array('id'=>$tapa->getId()));

}

$repository = $this->getDoctrine()->getRepository(Tapa::class);

$tapas = $repository->findAll();

return $this->render('gestionPlatos/nuevoPlato.html.twig', array("form"=>$form->createView()));

}

En config.yml crear un parámetro con la ruta absoluta: La seguna es una variable global que se crea para que todo sea mas generico

parameters:

locale: en

plato\_img\_directory: '%kernel.project\_dir%/web/img/uploads'

En la vista usar el render que visualice Opc1:

<img class="img-fluid" src="{{asset('img/uploads')}}/{{plato.foto}}" alt="">

otra forma mas elegante de realizar la injeccion de variables globales es:

# Twig Configuration

twig:

debug: '%kernel.debug%'

strict\_variables: '%kernel.debug%'

globals:

plato\_img: 'img/uploads/'

En la vista usar el render que visualice Opc2:

<img class="img-fluid" src="{{asset( plato\_img~plato.foto )}}" alt="">

Concatenar en twig es con var1~var2

Imágenes {{asset(plato\_img~tapa.foto) }}

# Relaciones many to one

Primero se debe crear la realacion en las entities de Productos(platos) para que sea relacional

Esto significa que muchos productos pueden tener solo una categoría. Many to one.

Para modificar y crear una relación se cera el campo en el entity que se desea modificar.

Agregar después de $top el código de ejemplo

En tapa.php

*/\*\**

*\* @ORM\ManyToOne(targetEntity="Category", inversedBy="products")*

*\* @ORM\JoinColumn(name="category\_id", referencedColumnName="id")*

*\*/*

private $category;

y modifuicarlo segun uestra entidad Categoria asi:

/\*\*

\* @ORM\ManyToOne(targetEntity="Categoria", inversedBy="products")

\* @ORM\JoinColumn(name="categoria\_id", referencedColumnName="id")

\*/

private $categoria;

# Relaciones one to many

A continuacion se debe modificar la relacion Oneto Many ya que una categoria puede tener muchos platoa asociados

En categoria.php

Ejemplo

*/\*\**

*\* @ORM\OneToMany(targetEntity="Product", mappedBy="category")*

*\*/*

private $products;

Modificarlo por:

/\*\*

\* @ORM\OneToMany(targetEntity="Tapa", mappedBy="categoria")

\*/

private $tapas;

A continuaciion crear el constructor puede ponerse debajo de

public function \_\_construct()

{

$this->tapas = new ArrayCollection();

}

Incluir la librería

use Doctrine\Common\Collections\ArrayCollection;

Finalmente traspasar la configuracion a la base de datos. Con la consola

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

Ahora se debe generar los seters para los item nuevos… CMC

php bin/console doctrine:generate:entities AppBundle

Generating entities for bundle "AppBundle"

> backing up Categoria.php to Categoria.php~

> generating AppBundle\Entity\Categoria

> backing up Tapa.php to Tapa.php~

> generating AppBundle\Entity\Tapa