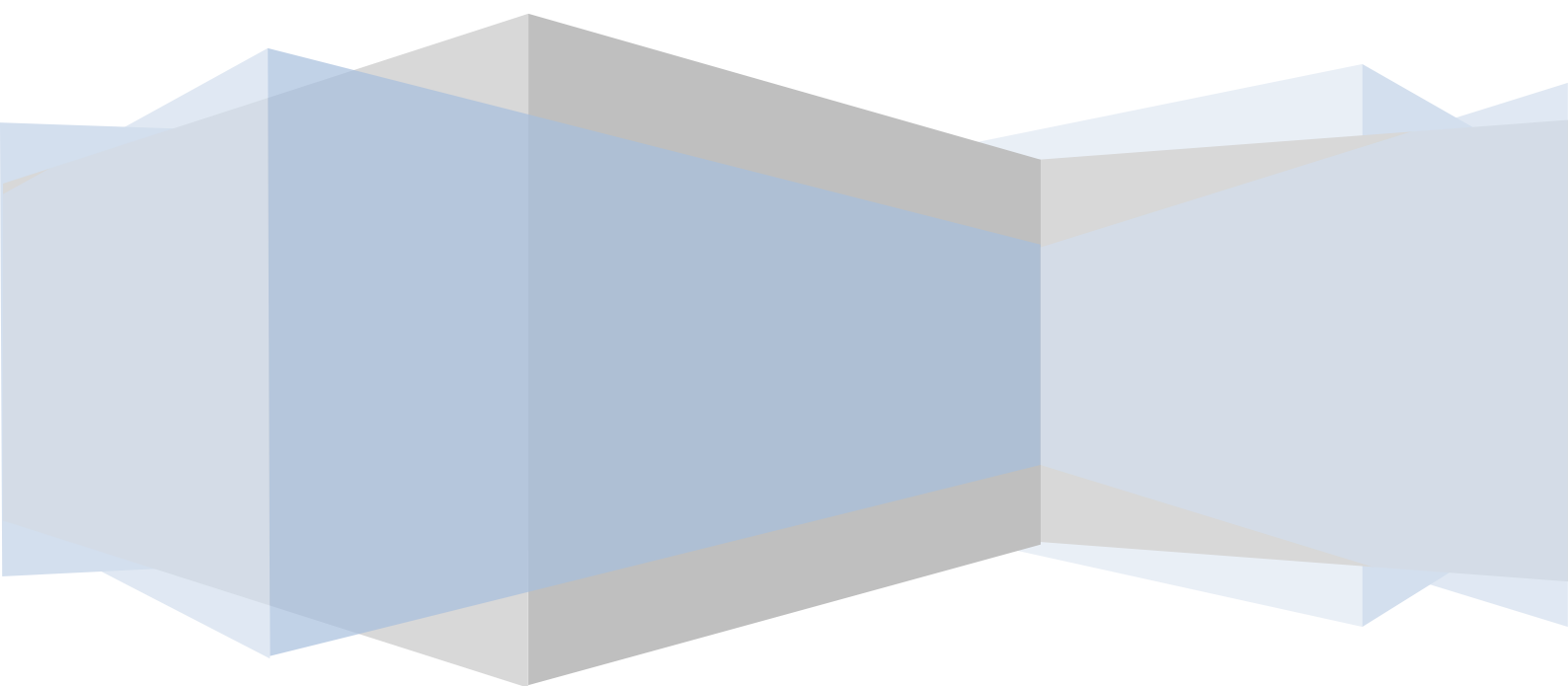


# SYMPHONY2

Manual de Programación web

Juan Antonio Japón de la Torre



## Contenido

1. Instalación en Windows .....	3
2. Creación de una Aplicación Symfony .....	4
3. Ejecutar una Aplicación en Symfony .....	4
4. Enrutamiento en Symfony .....	5
4. Pasando parámetros a través de Rutas.....	7
5. Configuración de la Base de Datos.....	8
6. Creación del Modelo de nuestro Proyecto (Entidades) .....	8
6.1 Creación de atributos en cada entidad .....	9
7. Generando la base de datos desde el CMD y comandos Importantes .....	10
Creando los otros atributos de cada entidad.....	12
GETTER Y SETTERS .....	12
Relaciones entre tablas .....	14
Autogeneración de Entidades teniendo una base de datos ya creada.....	18
8. Operaciones en Controlador .....	19
8.1 Insertando datos en la base de datos desde controlador.....	19
8.2 Buscando datos de la BD desde el controlador .....	20
8. Vistas (Archivos HTML.twig).....	22
8.1. Incluir archivos a nuestras vistas (css, js, jquery, imagenes).....	23
8.2 Incluir formularios en Vistas y trabajar con sus datos: GET Y POST.....	24

# SYMFONY2

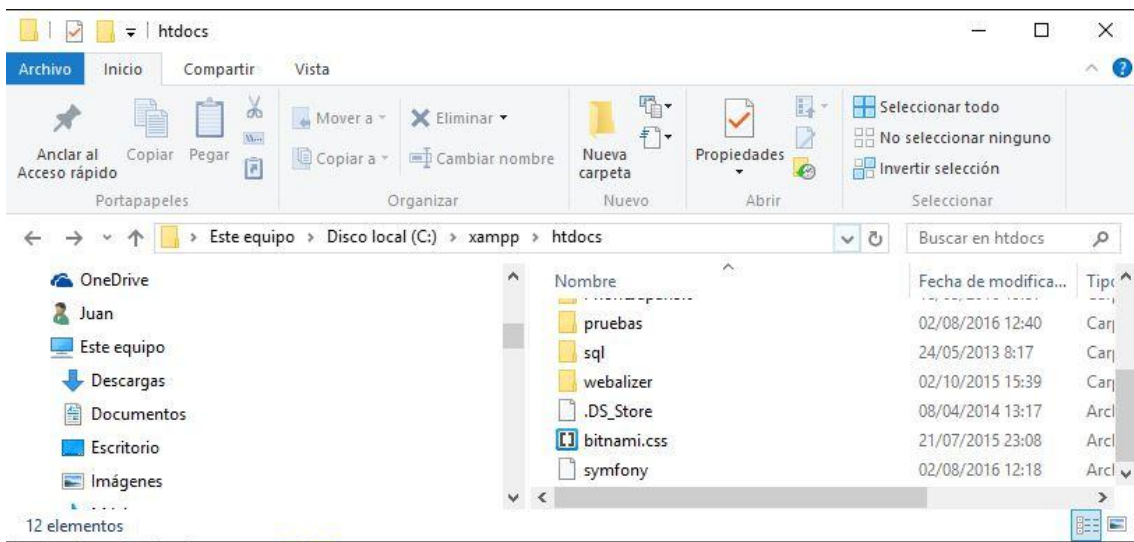
## 1. Instalación en Windows

Para comenzar debemos descargarnos symfony. Para ello usaremos el siguiente comando en el CMD abriéndolo como administrador y accediendo al directorio htdocs:

```
cd /xampp/htdocs
```

```
php -r "readfile('https://symfony.com/installer');" > symfony
```

Una vez realizada la descarga de symfony nos aparecerá el siguiente fichero llamado symfony en el directorio htdocs donde lo hemos descargado.



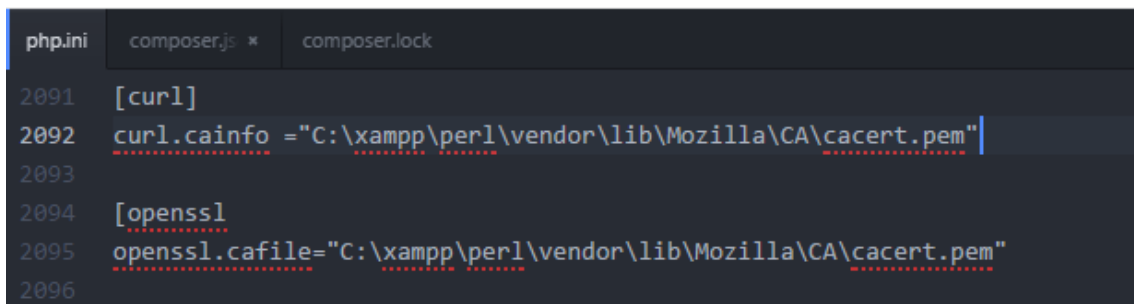
Ahora procederemos a configurar el fichero php.ini que se encuentra en **/xampp/php** y añadiremos las siguientes líneas que permitirá que podamos crear un proyecto symfony sin necesidad de utilizar el composer.

[curl]

```
curl.cainfo="C:\xampp\perl\vendor\lib\Mozilla\CA\cacert.pem"
```

[openssl]

```
openssl.cafile="C:\xampp\perl\vendor\lib\Mozilla\CA\cacert.pem"
```

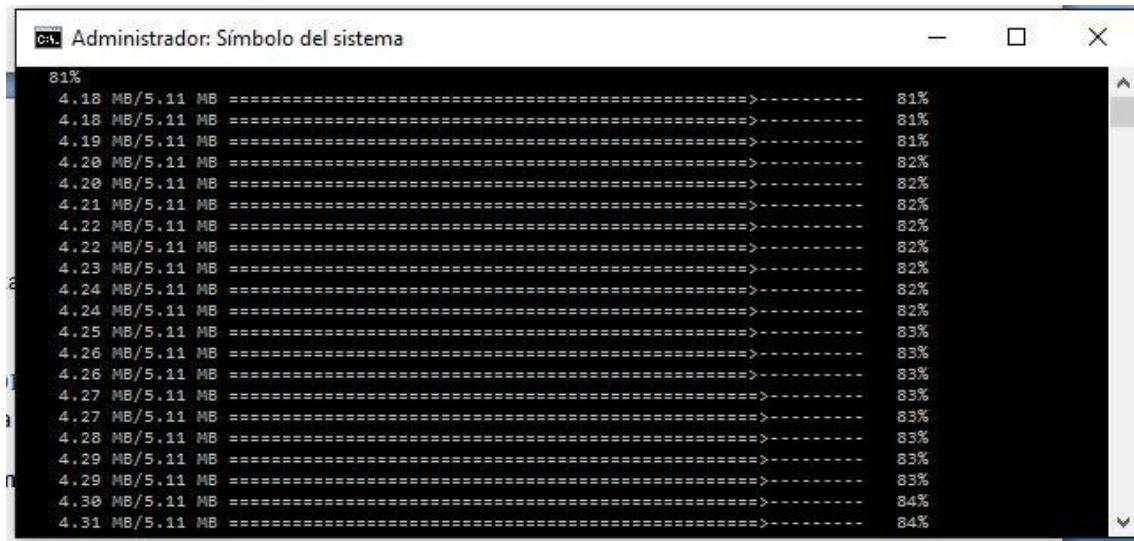


Una vez realizado estos cambios reiniciamos nuestro servidor PHP y Apache en XAMPP

## 2. Creación de una Aplicación Symfony

Para crear una aplicación usando symfony lo realizaremos con el comando:

**php symfony new [nombre\_del\_proyecto]**



Una vez creado el proyecto nos aparecerá en nuestro proyecto los siguientes ficheros creados:

pp > htdocs > ejemplo2				
Nombre	Fecha de modifica...	Tipo	Tamaño	
app	02/08/2016 12:27	Carpeta de archivos		
bin	02/08/2016 12:27	Carpeta de archivos		
src	02/08/2016 12:27	Carpeta de archivos		
tests	02/08/2016 12:27	Carpeta de archivos		
var	02/08/2016 12:27	Carpeta de archivos		
vendor	02/08/2016 12:27	Carpeta de archivos		
web	02/08/2016 12:27	Carpeta de archivos		
.gitignore	02/08/2016 12:27	Documento de tex...	1 KB	
composer.json	02/08/2016 12:27	Archivo JSON	3 KB	
composer.lock	02/08/2016 12:27	Archivo LOCK	73 KB	
phpunit.xml.dist	02/08/2016 12:27	Archivo DIST	1 KB	
README.md	02/08/2016 12:27	Archivo MD	1 KB	

## 3. Ejecutar una Aplicación en Symfony

Para ello accedemos a la carpeta de nuestro proyecto y ejecutamos el siguiente comando:

**cd /xampp/htdocs/ejemplo**

**php bin/console server:run**

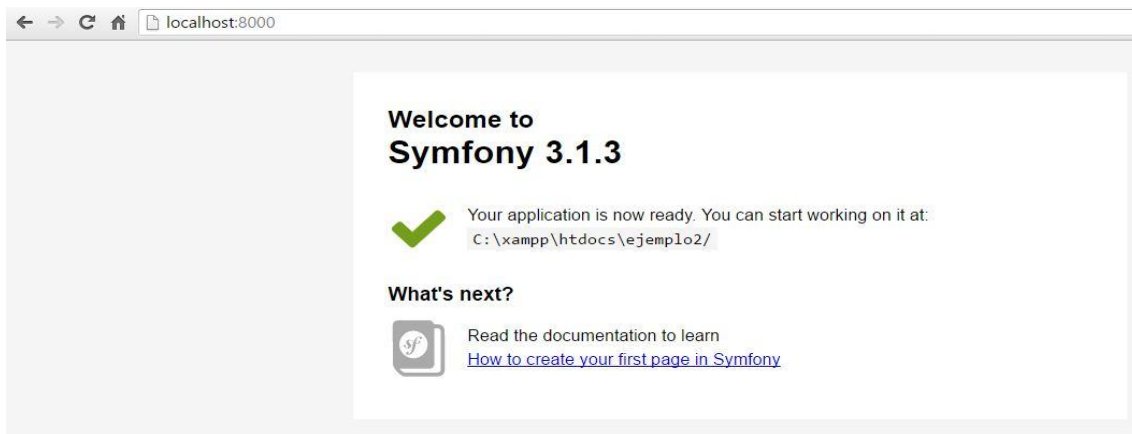
```
CA: Administrador: Símbolo del sistema - php bin/console server:run
server:run          Runs PHP built-in web server
server:start        Starts PHP built-in web server in the background
server:status       Outputs the status of the built-in web server for the given address
server:stop         Stops PHP's built-in web server that was started with the server:start command
swiftmailer:debug   Displays current mailers for an application
swiftmailer:email:send Send simple email message
swiftmailer:pool:send Sends emails from the spool
translation         Updates the translation file
translation:update

C:\xampp\htdocs\ejemplo2>php bin/console server:run
PHP: syntax error, unexpected $end, expecting ']' in C:\xampp\php\php.ini on line 2094

[OK] Server running on http://127.0.0.1:8000

// Quit the server with CONTROL-C.
```

E iniciamos en el navegador symfony usando como ruta la que nos devolvió el cmd a la hora de arrancar el servidor:



## 4. Enrutamiento en Symfony

Symfony se basa en un sistema de rutas para poder llamar a los diferentes métodos que vamos creando en nuestra aplicación.

Hay 2 maneras de usar rutas en symfony para llamar a un método que hayamos creado.

- **Creando las rutas en el fichero routing.yml:** En este caso en nuestro DefaultController (src/AppBundle/Controller) crearemos nuestro método. Obligatorio. Poner action en el nombre del método (prueba**Action**)

```
class DefaultController extends Controller
{
    public function pruebaAction(Request $request){
    }
}
```

Una vez este creado accederemos a nuestro fichero routing.yml (/config) y crearemos una ruta para llamar a este método:

```
prueba1:
  path: /prueba
  defaults: { _controller: AppBundle:Default:prueba }
```

Donde:

- Prueba1 : nombre que le daremos a nuestra ruta
  - Path: será la url para acceder a nuestra ruta (localhost:8000/proyecto/prueba)
  - Defaults: es donde le diremos que controlador tiene el método y que método llamaremos. No se le pone Action.
- **Creando las rutas en el mismo fichero de nuestro controlador:** En este caso en nuestro DefaultController (src/AppBundle/Controller) crearemos nuestro método. Obligatorio. Poner action en el nombre del método (prueba**Action**). Encima de él añadiremos la ruta para llamar a dicho método:

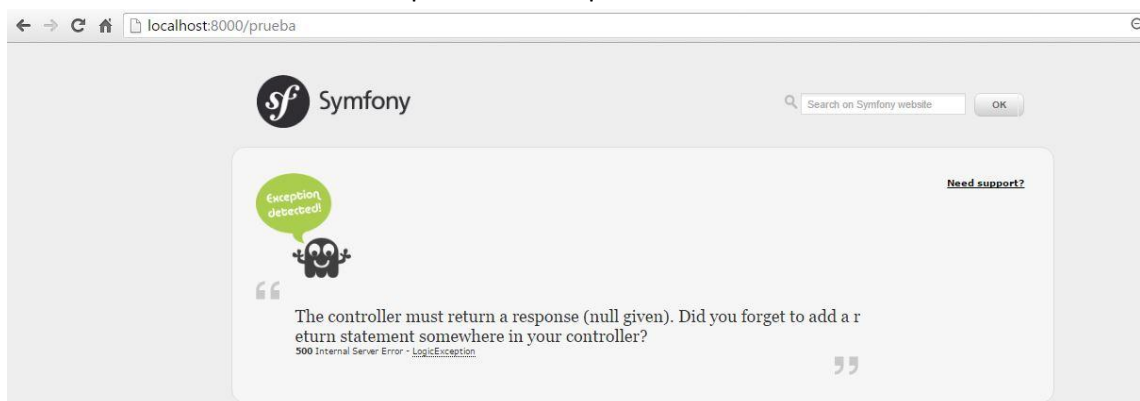
```
class DefaultController extends Controller
{

    /**
     * @Route("/prueba", name="primeraprueba")
     */
    public function pruebaAction(Request $request){

    }

}
```

Por último llamaremos al método pruebaAction que hemos creado con anterioridad



## 4. Pasando parámetros a través de Rutas

Definimos el parámetro que queramos pasarle en el método que hayamos creado.

```
public function productosAction($id, Request $request){  
    |  
}  
}
```

Para pasar parámetros a través de una ruta en symfony tenemos las siguientes opciones:

- Pasar un parámetro: si queremos pasar un parámetro a través de una ruta tenemos que definirlo en ella usando **{nombre\_parametro}**

```
productos:  
    path: /productos/{id}  
    defaults: {_controller: AppBundle:Default:productos}
```

- Pasar un parámetro por defecto en caso de no enviar nada: si el usuario no enviara el parámetro y en nuestra ruta tenemos puesto que vamos a enviar un parámetro podemos asignarle un valor por defecto. Este valor se lo asignamos al final de defaults:

```
productos:  
    path: /productos/{id}  
    defaults: {_controller: AppBundle:Default:productos, id:1}
```

Si queremos decirle que por defecto acceda por GET o por post solo añadiríamos debajo de default una nueva línea:

**Methods: [POST]**

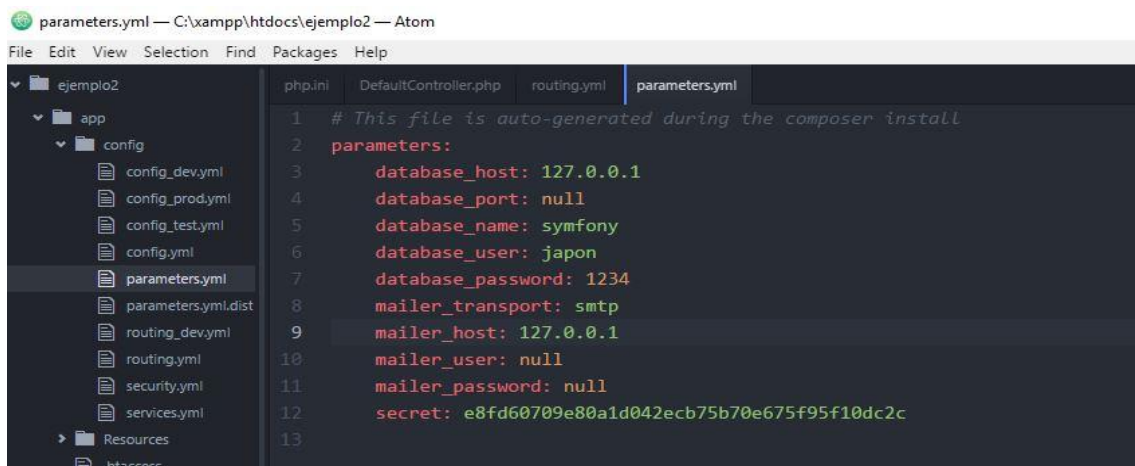
**Methods: [GET , POST]**

## 5. Configuración de la Base de Datos

Para configurar la base de datos que vamos a usar para trabajar y que usuario va a conectarse a ella tendremos que crearnos primero en nuestro phpmyadmin:

- Base de datos: en mi caso le he llamado symfony a la que he creado
- Usuario: japon, Contraseña:1234

Una vez creado el usuario y la base de datos en nuestro proyecto symfony le decimos que vamos a usar esa base de datos y el usuario que se va a conectar en un fichero llamado **parameters.yml** que se encuentra en **/app/Config**



```
parameters.yml — C:\xampp\htdocs\ejemplo2 — Atom
File Edit View Selection Find Packages Help
ejemplo2
└─ app
   └─ config
      ├── config_dev.yml
      ├── config_prod.yml
      ├── config_test.yml
      ├── config.yml
      └── parameters.yml
         ├── parameters.yml.dist
         ├── routing_dev.yml
         ├── routing.yml
         ├── security.yml
         └── services.yml
Resources
ntarcess

1 # This file is auto-generated during the composer install
2 parameters:
3     database_host: 127.0.0.1
4     database_port: null
5     database_name: symfony
6     database_user: japon
7     database_password: 1234
8     mailer_transport: smtp
9     mailer_host: 127.0.0.1
10    mailer_user: null
11    mailer_password: null
12    secret: e8fd60709e80a1d042ecb75b70e675f95f10dc2c
13
```

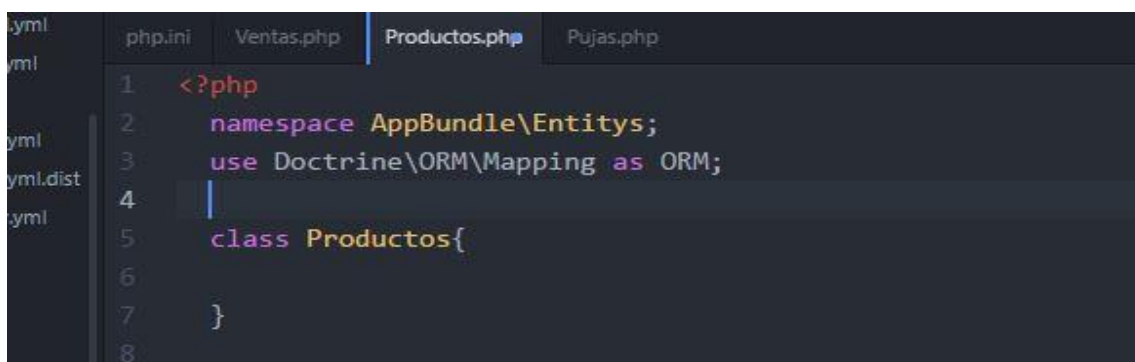
## 6. Creación del Modelo de nuestro Proyecto (Entidades)

Una vez que tenemos desarrollado nuestro modelo en papel o en cualquier herramienta como por ejemplo lucidchart comenzamos a programarlo en nuestro proyecto symfony.

Lo primero que haremos será crear en el directorio **src/appBundle** un sistema de directorios que tenga las siguientes carpetas:

- **Entity**: será el directorio donde creemos nuestras entidades
- **Resources**: directorio que tendrá las vistas de nuestra aplicación
- **Controller (creado por defecto)**: contendrá los controladores que usaremos que tendrá acciones

Para crear una entidad accederemos al directorio Entitys y crearemos la entidad:



```
ym
ym
ym
ym.dist
ym

php.ini Ventas.php Productos.php Pujas.php

1 <?php
2     namespace AppBundle\Entity;
3     use Doctrine\ORM\Mapping as ORM;
4
5     class Productos{
6
7     }
8
```



Tendremos que tener en cuenta lo siguiente:

- **Namespace AppBundle\Entitys:** en esta línea le decimos que es una entidad que se encuentra en el directorio Entitys que está en AppBundle
- **Use Doctrine\ORM\Mapping as ORM:** con esta línea permitiremos a doctrine gestionar la entidad de tal manera que no tengamos que realizar o establecer las consultas nosotros. Podremos usar todos los métodos que tiene Doctrine

Una vez hayamos añadido Doctrine podremos usar las anotaciones que tiene:

```
<?php
namespace AppBundle\Entity;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 *
 * @ORM\Table(name="productos")
 */
class Productos{

}
```

Como por ejemplo el caso de arriba. En el estamos diciéndole con **@ORM\Entity** que esa clase va a ser una entidad y con **@ORM\Table(name="nombre\_tabla")** que va a hacer referencia a una tabla de nuestra base de datos. De esta manera conseguiremos tener persistencia

Una vez añadida las anotaciones usando el comando:

**php bin/console doctrine:schema:create:** creara directamente en nuestra base de datos MySQL las tablas a partir de las entidades que creamos en nuestra aplicación

```
C:\xampp\htdocs\ejemplo2>php bin/console doctrine:schema:create

[Doctrine\ORM\Mapping\MappingException]
No identifier/primary key specified for Entity "AppBundle\Entity\Contactos"
. Every Entity must have an identifier/primary key.

doctrine:schema:create [--dump-sql] [--em [EM]] [-h|--help] [-q|--quiet] [-v|vv|
vvv|--verbose] [-V|--version] [--ansi] [--no-ansi] [-n|--no-interaction] [-e|--e
nv ENV] [--no-debug] [--] <command>

C:\xampp\htdocs\ejemplo2>
```

## 6.1 Creación de atributos en cada entidad

Ahora pasaremos a crear las variables que usaremos en cada una de nuestras entidades que creamos con anterioridad. Tendremos las siguientes variables en nuestra entidad:

- Clave primaria: será el identificador que diferenciara a cada una de nuestras entidades. Esta clave es única

```
class Productos{  
    /**  
     * @ORM\Id  
     *  
     * @ORM\Column(type="integer")  
     *  
     * @ORM\GeneratedValue(strategy="AUTO")  
     */  
    protected $idProd;  
}
```

Como se puede comprobar tenemos las siguientes anotaciones para definir el atributo:

- **@ORM\Id** : con esto le decimos que será clave primaria
- **@ORM\Column(type="integer")**: con esto decimos que será tipo entero
- **@ORM\GeneratedValue(strategy="AUTO")**: establecemos una secuencia de manera que el valor se vaya generando automáticamente

## 7. Generando la base de datos desde el CMD y comandos Importantes

Una vez añadidos los atributos a nuestras entidades lo siguiente que haremos será que doctrine cree dichas tablas con sus atributos en la base de datos. Para ello debemos conocer los siguientes comandos:

- **Php bin/console doctrine:schema:drop** :con este comando eliminamos las tablas que tengamos creada de la base de datos
- **Php bin/console doctrine:schema:drop --dump-sql**: mostrará la acción que realizará ese comando
- **Php bin/console**: nos mostrará todos los comandos que podemos usar en symfony
- **Php bin/console doctrine:schema:create** : nos creará las tablas en nuestra base de datos

```
C:\xampp\htdocs\ejemplo2>php bin/console doctrine:schema:create  
ATTENTION: This operation should not be executed in a production environment.  
  
Creating database schema...  
Database schema created successfully!  
  
C:\xampp\htdocs\ejemplo2>
```

- **Php bin/console doctrine:schema:create --dump-sql:** nos mostrara que acciones se realizaran para crear las tablas

```

Unknown column type "int" requested. Any Doctrine type that you use has to
be registered with \Doctrine\DBAL\Types\Type::addType(). You can get a list
of all the known types with \Doctrine\DBAL\Types\Type::getTypesMap(). If t
his error occurs during database introspection then you might have forgot t
o register all database types for a Doctrine Type. Use AbstractPlatform#reg
isterDoctrineTypeMapping() or have your custom types implement Type#getMapp
edDatabaseTypes(). If the type name is empty you might have a problem with
the cache or forgot some mapping information.

doctrine:schema:drop [--dump-sql] [-f|--force] [--full-database] [--em [EM]] [-h|--help] [-q|--quiet] [-v|vv|vvv|--verbose] [-V|--version] [--ansi] [--
ansi] [-n|--no-interaction] [-e|--env ENV] [--no-debug] [--] <command>

C:\xampp\htdocs\ejemplo2>php bin/console doctrine:schema:drop --dump-sql

C:\xampp\htdocs\ejemplo2>php bin/console doctrine:schema:create --dump-sql
CREATE TABLE contactos (id INT AUTO_INCREMENT NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB;
CREATE TABLE productos (id_prod INT AUTO_INCREMENT NOT NULL, PRIMARY KEY(id_prod)) DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB;
CREATE TABLE pujas (id INT AUTO_INCREMENT NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB;
CREATE TABLE usuarios (id_user INT AUTO_INCREMENT NOT NULL, PRIMARY KEY(id_user)) DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB;
CREATE TABLE ventas (id INT AUTO_INCREMENT NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB;

C:\xampp\htdocs\ejemplo2>

```

- **Php bin/console doctrine:schema:update --force :** actualiza las tablas si hemos añadido, eliminado alguna columna o hemos realizado algún cambio en la estructura de alguna de ellas

```

C:\xampp\htdocs\ejemplo2>php bin/console doctrine:schema:update
ATTENTION: This operation should not be executed in a production environment.
            Use the incremental update to detect changes during development and use
            the SQL DDL provided to manually update your database in production.

The Schema-Tool would execute "1" queries to update the database.
Please run the operation by passing one - or both - of the following options:
    doctrine:schema:update --force to execute the command
    doctrine:schema:update --dump-sql to dump the SQL statements to the screen

C:\xampp\htdocs\ejemplo2>php bin/console doctrine:schema:update --force
Updating database schema...
Database schema updated successfully! "1" query was executed

```

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Extra	Acción
1	id_prod	int(11)			No	Ninguna	AUTO_INCREMENT	Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más
2	nombre	varchar(60)	utf8_unicode_ci		No	Ninguna		Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más
3	imagen	varchar(120)	utf8_unicode_ci		No	Ninguna		Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más
4	precio_unitario	decimal(10,0)			No	Ninguna		Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más
5	descripcion	text	utf8_unicode_ci		No	Ninguna		Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más
6	fecha_creacion	datetime			No	Ninguna		Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más

- **Php bin/console doctrine:schema:update --dump-sql :** nos muestra que acción realizara el comando en nuestra base de datos

```

C:\xampp\htdocs\ejemplo2>php bin/console doctrine:schema:update --dump-sql
ALTER TABLE productos ADD nombre VARCHAR(60) NOT NULL, ADD imagen VARCHAR(120) NOT NULL, ADD precio_unitario NUMERIC(10, 0) NOT NULL, ADD descripcion TEXT NOT NULL, ADD fecha_creacion DATETIME NOT NULL;

C:\xampp\htdocs\ejemplo2>

```

## Creando los otros atributos de cada entidad

Para definir otros atributos en nuestras entidades debemos usar anotaciones ya predefinidas por doctrine. Aquí tenemos ejemplos de cómo definir algunos tipos de datos.

```
protected $idProd;
/**
 * @ORM\Column(type="string", length=60, nullable=false)
 */
protected $nombre;
/**
 * @ORM\Column(type="string", length=60, nullable=false)
 */
protected $imagen;
/**
 * @ORM\Column(type="decimal", nullable=false)
 */
protected $precioUnitario;
/**
 * @ORM\Column(type="text", length=400, nullable=false)
 */
protected $descripcion;
/**
 * @ORM\Column(type="date", nullable=false)
 */
protected $fechaCreacion;
}
```

## GETTER Y SETTERS

Para cada uno de los atributos que hemos creados debemos crear sus funciones get y set para poder acceder a los datos y poder utilizarlos como deseemos:

- GET: se usa para que podamos recuperar el valor del atributo que queramos ver de un objeto que tengamos

```
protected function getNombre(){
    return $this->nombre;
}
```

- SET: se usa para modificar el valor del atributo que queramos cambiar de un objeto

```
protected function setNombre($nombre){
    $this->nombre = $nombre;
    return $this;
}
```

Debemos crear estas 2 funciones para cada uno de los atributos que hayamos creado en nuestra entidad como se muestra en el ejemplo de abajo:

```
protected function getIdProd(){
    return $this->idProd;
}

protected function getNombre(){
    return $this->nombre;
}

protected function setNombre($nombre){
    $this->nombre = $nombre;
    return $this;
}

protected function getPrecioUnitario(){
    return $this->precioUnitario;
}

protected function setPrecioUnitario($precioUnitario){
    $this->precioUnitario = $precioUnitario;
    return $this;
}
```

Sin embargo doctrine proporciona un comando con el que podemos autogenerar los métodos get y set sin necesidad de escribirlos nosotros como hicimos con anterioridad. El comando sería el siguiente:

**Php bin/console doctrine:generate:entities nombreBundle\FicheroEntidades\NombreEntidad**

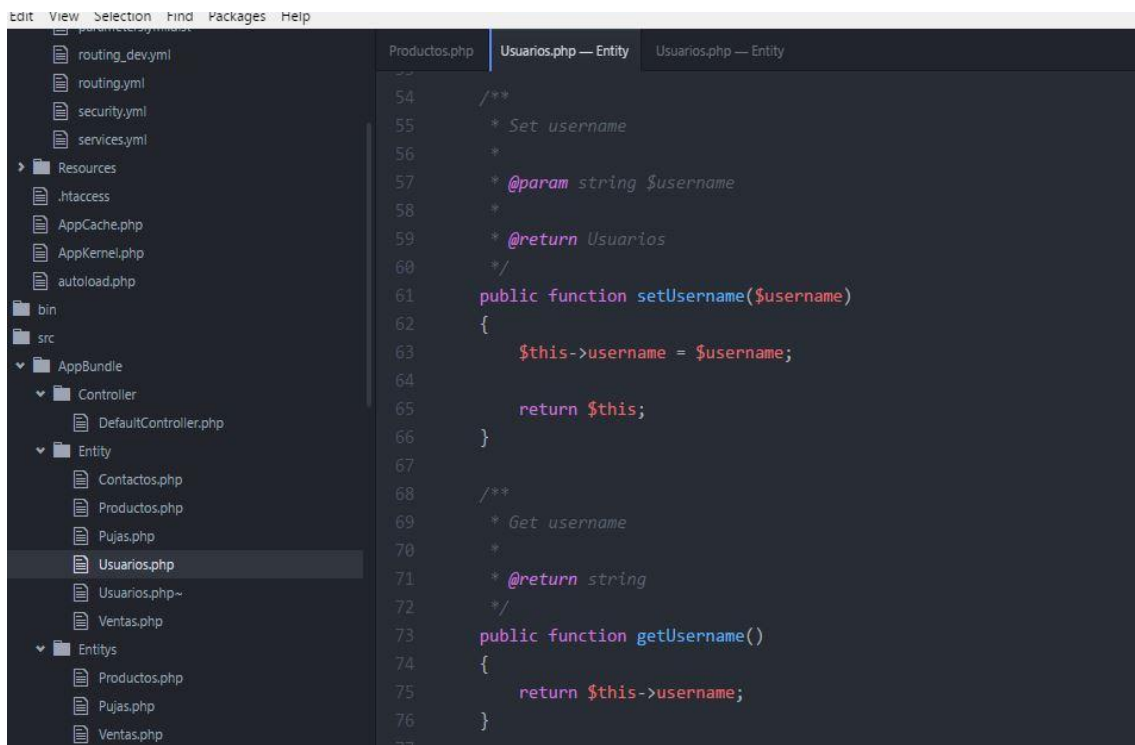
De esta manera quedaría así el comando en el caso de que quiera autogenerar los métodos get y set si tuviéramos un bundle llamado AppBundle y una entidad llamada Usuarios:

```
C:\xampp\htdocs\ejemplo2>php bin/console doctrine:generate:entities AppBundle\Entity\Usuarios
Generating entity "AppBundle\Entity\Usuarios"
> backing up Usuarios.php to Usuarios.php~
> generating AppBundle\Entity\Usuarios

C:\xampp\htdocs\ejemplo2>
```



Una vez realizado el comando tendríamos la entidad Usuarios con los métodos get y set generados por doctrine.



```

54  /**
55   * Set username
56   *
57   * @param string $username
58   *
59   * @return Usuarios
60   */
61  public function setUsername($username)
62  {
63      $this->username = $username;
64
65      return $this;
66  }
67
68  /**
69   * Get username
70   *
71   * @return string
72   */
73  public function getUsername()
74  {
75      return $this->username;
76  }
77  
```

Por si acaso, doctrine realiza una copia de seguridad del archivo antiguo por si queremos recuperar algún cambio que hicimos con anterioridad

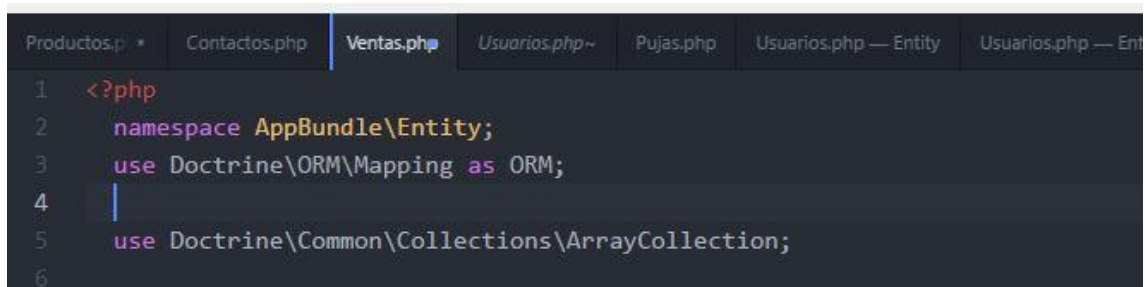
## Relaciones entre tablas

Para las relaciones entre entidades tendremos el siguiente enlace para poder realizar estas operaciones

<http://doctrine-orm.readthedocs.io/projects/doctrine-orm/en/latest/reference/association-mapping.html#one-to-many-bidirectional>

Para hacer el mapeado debemos importar en nuestra entidad la clase ArrayCollection ya que el atributo será una colección o array

**use Doctrine\Common\Collections\ArrayCollection;**



```

1  <?php
2      namespace AppBundle\Entity;
3      use Doctrine\ORM\Mapping as ORM;
4
5      use Doctrine\Common\Collections\ArrayCollection;
6  
```

Relaciones que tenemos:

- 1:N
- N:M;
- N:1;

Por ejemplo si tenemos un caso de una relación OneToMany entre Usuarios y Productos tendríamos que hacer los siguientes pasos:

- Importamos la entidad con la que se va a relacionar y la clase Collection. En este caso como tenemos una entidad intermedia que es la que lo relaciona importamos Ventas.

```
use Doctrine\Common\Collections\ArrayCollection;
use AppBundle\Entity\Ventas;
```

- Creamos la variable que vamos a usar para establecer la relación entre ambas y su constructor

```
protected $productos;
public function __construct(){
    $this->$productos = new ArrayCollection();
}
```

- Insertamos una anotación a \$productos para decirle que la relación va a ser onetomany por ejemplo

```
/**
 * @ORM\OneToMany(targetEntity="Productos", mappedBy="vendedor")
 */
protected $productos;
```

- Volvemos a generar con el comando **doctrine:generate:entities** los métodos de la colección que acabamos de crear en Usuarios

```
C:\xampp\htdocs\ejemplo2>php bin/console doctrine:generate:entities AppBundle\Entity\Usuarios
Generating entity "AppBundle\Entity\Usuarios"
> backing up Usuarios.php to Usuarios.php~
> generating AppBundle\Entity\Usuarios
C:\xampp\htdocs\ejemplo2>
```

Automáticamente symfony nos autogenera los métodos de borrado, añadir y recuperar los datos de dicha colección

```
public function addProducto(\AppBundle\Entity\Productos $producto)
{
    $this->productos[] = $producto;

    return $this;
}

/**
 * Remove producto
 *
 * @param \AppBundle\Entity\Productos $producto
 */
public function removeProducto(\AppBundle\Entity\Productos $producto)
{
    $this->productos->removeElement($producto);
}

/**
 * Get productos
 *
 * @return \Doctrine\Common\Collections\Collection
 */
public function getProductos()
{
    return $this->productos;
}
```

Ahora tenemos que realizar lo mismo en la entidad productos pero en vez de ser OneToMany como en Usuarios ahora es una ManyToOne

- Importamos la entidad ventas que es la que relaciona ambas entidades

```
use AppBundle\Entity\Ventas;
```

- Creamos el atributo que relaciona a Productos con Usuarios. Como en el mappedBy de usuarios pusimos vendedor el atributo que creamos tiene q llamarse igual para que se produzca la relación

```
protected $vendedor;
```

- Usando anotaciones le decimos que va a ser ManyToOne porque de Productos a Usuarios es N:1.

En el targetEntity pondremos la entidad con la que se relaciona y en inversedBy el campo que creamos en la entidad Usuarios con el que se relacionaría

En JoinColumn como si fuera una clave foránea pondremos en el name el nombre de la columna que tenemos en la entidad Productos que se relacionara con usuarios y en referencedColumnName la columna de Usuarios con la que se relaciona

```
/**
 * @ORM\ManyToOne(targetEntity="Usuarios", inversedBy="productos")
 * @ORM\JoinColumn(name="vendedor", referencedColumnName="id_user")
 */
protected $vendedor;
```



- Por último actualizamos la entidad con el comando `doctrine:generate:entities`

```
C:\xampp\htdocs\ejemplo2>php bin/console doctrine:generate:entities AppBundle\Entity\Productos
Generating entity "AppBundle\Entity\Productos"
> backing up Productos.php to Productos.php~
> generating AppBundle\Entity\Productos
```

```
/**
 * Set vendedor
 *
 * @param AppBundle\Entity\Usuarios $vendedor
 *
 * @return Productos
 */
public function setVendedor(AppBundle\Entity\Usuarios $vendedor = null)
{
    $this->vendedor = $vendedor;

    return $this;
}

/**
 * Get vendedor
 *
 * @return AppBundle\Entity\Usuarios
 */
public function getVendedor()
{
    return $this->vendedor;
}
```

- Y para finalizar del todo actualizamos la base de datos con `doctrine:schema:update` para que se nos creen las relaciones en la base de datos y se nos creara en la base de datos el campo vendedor que relaciona a ambas tablas

```
C:\xampp\htdocs\ejemplo2>php bin/console doctrine:schema:update --force
Updating database schema...
Database schema updated successfully! "3" queries were executed
```

Estructura de tabla

Vista de relaciones

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Extra	Acción
<input type="checkbox"/>	1 id_prod	int(11)			No	Ninguna	AUTO_INCREMENT	 Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más
<input type="checkbox"/>	2 nombre	varchar(60)	utf8_unicode_ci		No	Ninguna		 Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más
<input type="checkbox"/>	3 imagen	varchar(120)	utf8_unicode_ci		No	Ninguna		 Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más
<input type="checkbox"/>	4 precio_unitario	decimal(10,0)			No	Ninguna		 Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más
<input type="checkbox"/>	5 descripcion	text	utf8_unicode_ci		No	Ninguna		 Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más
<input type="checkbox"/>	6 fecha_creacion	datetime			No	Ninguna		 Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más
<input type="checkbox"/>	7 vendedor	int(11)			Si	NULL		 Cambiar  Eliminar  Primaria  Único  Índice  Espacial  Más

## Autogeneración de Entidades teniendo una base de datos ya creada

Esta es una manera sin embargo podemos autogenerar todas las entidades a partir de una base de datos que tengamos creada usando los siguientes comandos:

- 1- Configuramos el fichero parameters.yml estableciendo la configuración de de la base de datos que vamos a usar en el proyecto

```
parameters:
    database_host: 127.0.0.1
    database_port: null
    database_name: phonejapan
    database_user: japon
    database_password: 1234
    mailer_transport: smtp
    mailer_host: 127.0.0.1
    mailer_user: null
    mailer_password: null
    secret: ceab6d69d6542d1b0dd2e05d0e7c645f9c60fdd1
```



The screenshot shows the phpMyAdmin interface with a table list for a database named 'phonejapan'. The table list includes columns for 'Tabla', 'Acción', 'Filas', 'Tipo', 'Cotejamiento', 'Tamaño', and 'Residuo a depurar'. The tables listed are: características, cesta, lineadepedido, opinion, pedido, producto, proveedor, suministro, and usuario.

Tabla	Acción	Filas	Tipo	Cotejamiento	Tamaño	Residuo a depurar
caracteristicas	Examinar Estructura Buscar Insertar Vaciar Eliminar	15	InnoDB	latin1_swedish_ci	32 KB	-
cesta	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	latin1_swedish_ci	32 KB	-
lineadepedido	Examinar Estructura Buscar Insertar Vaciar Eliminar	9	InnoDB	latin1_swedish_ci	48 KB	-
opinion	Examinar Estructura Buscar Insertar Vaciar Eliminar	14	InnoDB	latin1_swedish_ci	48 KB	-
pedido	Examinar Estructura Buscar Insertar Vaciar Eliminar	5	InnoDB	latin1_swedish_ci	32 KB	-
producto	Examinar Estructura Buscar Insertar Vaciar Eliminar	15	InnoDB	latin1_swedish_ci	16 KB	-
proveedor	Examinar Estructura Buscar Insertar Vaciar Eliminar	8	InnoDB	latin1_swedish_ci	16 KB	-
suministro	Examinar Estructura Buscar Insertar Vaciar Eliminar	16	InnoDB	latin1_swedish_ci	48 KB	-
usuario	Examinar Estructura Buscar Insertar Vaciar Eliminar	14	InnoDB	latin1_swedish_ci	32 KB	-

- 2- Generamos un fichero xml de la base de datos que tengamos creada de la siguiente manera

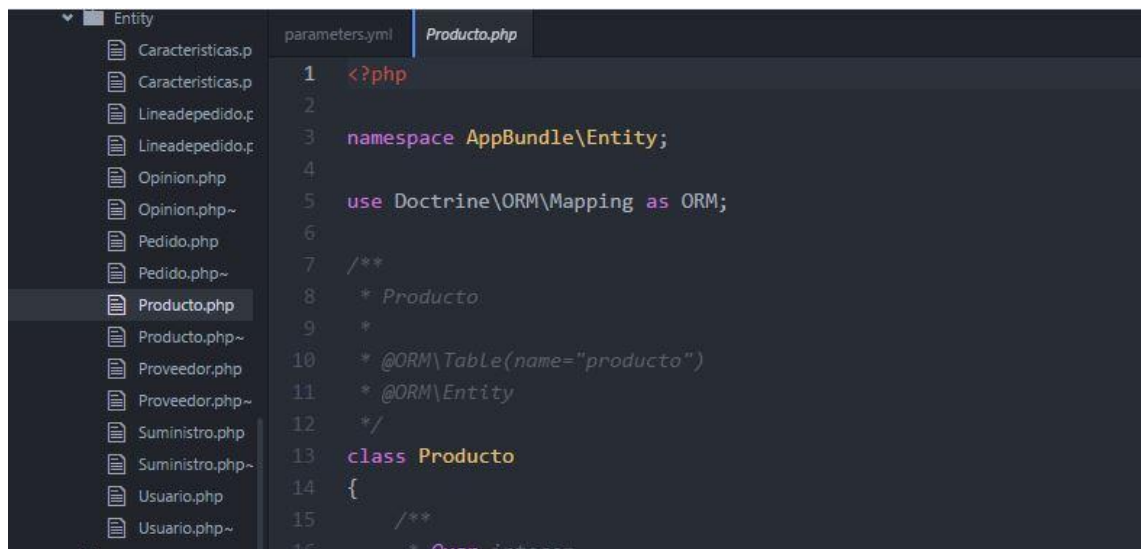
```
php bin/console doctrine:mapping:convert xml ./src/AppBundle/
/Resources/config/doctrine/metadata/orm --from-database --force
```

- 3- Generamos las entidades a partir del fichero xml que se creo utilizando el siguiente comando

```
php bin/console doctrine:mapping:import AppBundle annotation
```

- 4- Generamos los getters y setters de las entidades y los métodos de los atributos de la relación

```
php bin/console doctrine:generate:entities AppBundle
```



## 8. Operaciones en Controlador

### 8.1 Insertando datos en la base de datos desde controlador

Vamos a ver cómo podemos crear un nuevo objeto. Para ello, desde cualquier acción de cualquier controlador de nuestra aplicación, podemos hacer algo tan simple como:

```
$producto = new Producto();
```

Por supuesto, tendremos que haber incluido la ruta de esta entidad en la parte superior de este controlador de la siguiente manera:

```
use AppBundle\Entity\Producto;
```

Ahora, ya tenemos un objeto Producto vacío. Podemos hacer uso de los métodos get y set que creamos previamente para rellenarlo:

```
$producto->setNombre("Coche");  
$producto->setPrecio("10000");  
$producto->setDescripcion("Coche azul nuevo");
```

Bien, ahora mismo ya tenemos un objeto que nos gustaría que estuviese almacenado en la base de datos, pero ahora mismo el objeto sólo existe en este momento, si actualizamos la página perderíamos estos datos, por lo que tenemos que persistirlos en la base de datos. Para ello lo primero que vamos a hacer es inyectar el servicio de Doctrine. Ya veremos más adelante qué significa esto, pero por ahora nos quedamos con que vamos a traernos a nuestro controlador todas las funciones de Doctrine:

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

Ahora, tenemos el objeto `$em` que es el manager de Doctrine desde el que podemos utilizar muchas funcionalidades. Si queremos persistir este objeto en base de datos tan sólo necesitamos hacer:

```
$em->persist($producto);  
$em->flush();
```

Y nuestro objeto ya estará almacenado en la base de datos. De forma general nos quedaria de la siguiente forma:

```
use AppBundle\Entity\Producto;  
...  
class DefaultController extends Controller  
{  
    public function insertAction(Request $request)  
    {  
        $producto = new Producto();  
        $producto->setNombre("Coche");  
        $producto->setPrecio("10000");  
        $producto->setDescripcion("Coche azul nuevo");  
        $em = $this->getDoctrine()->getManager();  
        $em->persist($producto);  
        $em->flush();  
        return $this->render('AppBundle:default:index.html.twig', array( "texto" => "el  
Producto fue insertado"));  
    }  
}
```

## 8.2 Buscando datos de la BD desde el controlador

Tenemos que realizarlo a partir de los siguientes pasos:

- 1- Importamos la entidad de Productos o obtenemos el repositorio de la entidad productos

```
Use AppBundle\Entity\Producto  
$em = $this->getDoctrine()->getRepository("AppBundle:Producto");
```

- 2- Realizamos la consulta usando el entity manager

```
$consulta = $em->createQueryBuilder('p') → Alias  
    ->where('p.modelo= :id' ) → where de la consulta  
    ->setParameter('id', $id ) → parametro puesto en el where con : y su valor  
    ->getQuery();
```

```
$productos = $em->findAll(); → devuelve todas las filas de la tabla descrita en el em  
$productos = $em->findBy(array('marca' => 'Samsung')); → búsqueda por un campo
```

<http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html> → si queremos realizar consultas propias

- 3- Recuperamos el resultado de la consulta realizada con anterioridad (devuelto en forma de array)

```
$producto = $consulta->getResult();
```

Realizados los pasos anteriores nos quedaría una cosa similar a la que tenemos debajo:

### Use AppBundle\Entity\Productos

class DefaultController extends Controller

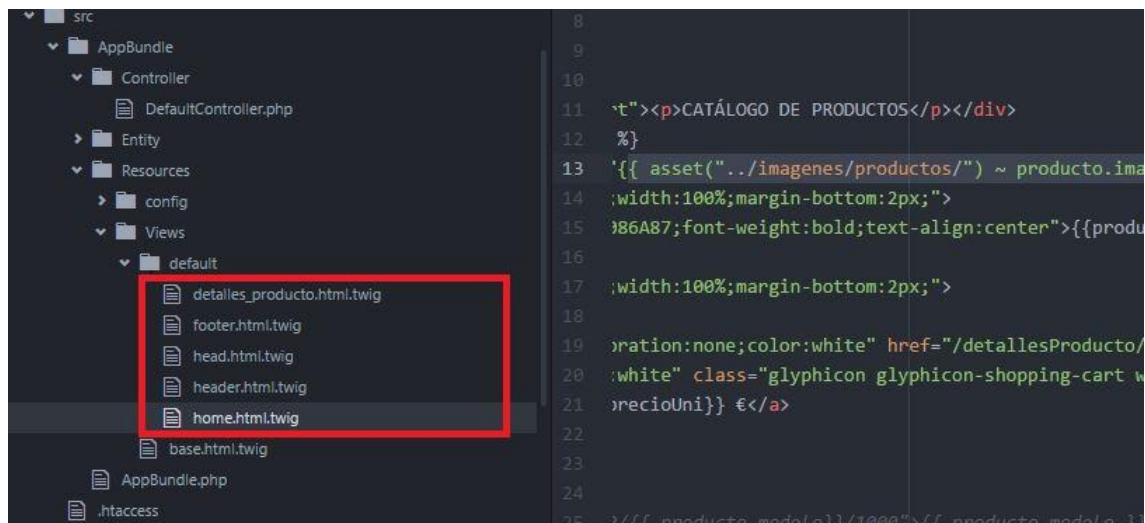
```
{
    public function pagina2Action($id,$num,Request $request){
        $em = $this->getDoctrine()->getRepository("AppBundle:Producto");
        $consulta = $em->createQueryBuilder('p')
            ->where('p.modelo= :id' )
            ->setParameter('id', $id )
            ->getQuery();
        $producto = $consulta->getResult();
        $item = $producto[0];
        return $this->render('AppBundle:default:pagina2.html.twig', array( 'producto' => $item,
'num'=>$num));
    }
}
```

```
public function pagina2Action($id,$num,Request $request){
    //realizamos la busqueda sobre una entidad, en este caso productos
    $em = $this->getDoctrine()->getRepository("AppBundle:Producto");
    //query a la base de datos
    $consulta = $em->createQueryBuilder('p')
        ->where('p.modelo= :id' )
        ->setParameter('id', $id )
        ->getQuery();
    //devuelve el resultado de la query(en este caso un array)
    $producto = $consulta->getResult();
    $item = $producto[0];
    return $this->render('AppBundle:default:pagina2.html.twig', array( 'producto' => $item, 'num'=>$num));
}
```

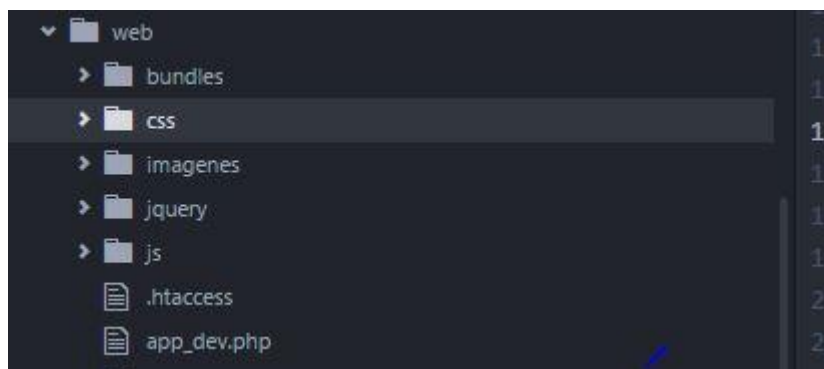
## 8. Vistas (Archivos HTML.twig)

Las vistas es la parte visible de la aplicación web para el usuario. A través de ellas el usuario podrá navegar por nuestra web. Los archivos HTML pasaran a tener una extensión **html.twig** ya que en symfony se trabaja con esta herramienta. Debemos tener en cuenta que los archivos estarán organizados de la siguiente manera:

- Archivos HTML.twig: son las vistas de nuestra aplicación y van alojados en la carpeta **src** de nuestro **Bundle Principal (AppBundle)** que es donde tenemos nuestros controladores y entidades dentro de la siguiente ruta (**src/AppBundle/Resources/Views**). No hace falta que este dentro de la carpeta defaults que se crea por defecto. Puede estar fuera de dicha carpeta los archivos. **No se usa PHP**



- Archivos CSS, JQuery, JS, Imagenes...: Se crean en la carpeta web de nuestra aplicación.





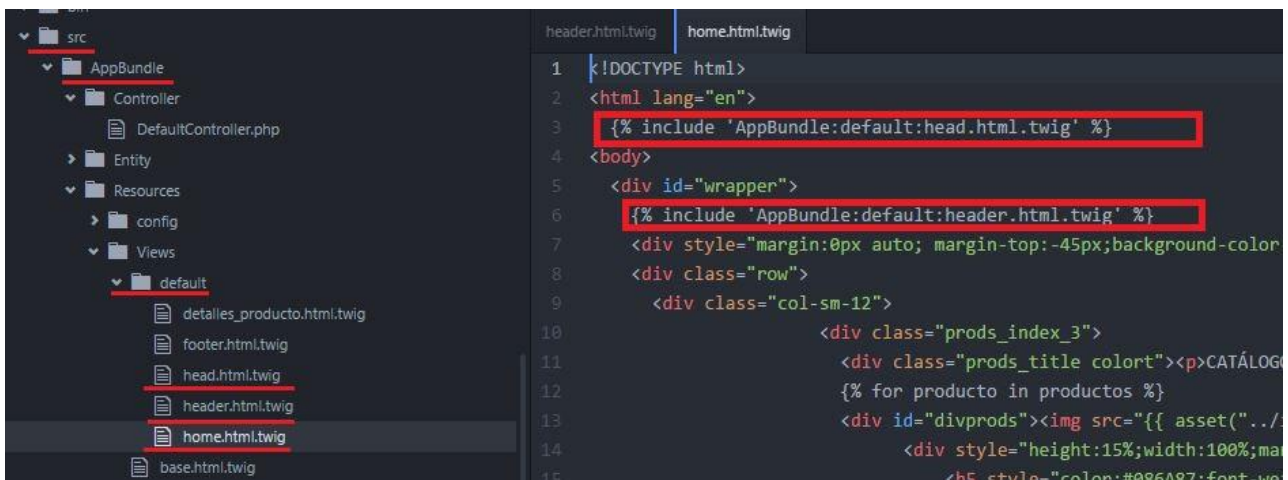
## 8.1. Incluir archivos a nuestras vistas (css, js, jquery, imagenes)

Para incluir en una vista (pagina.html.twig) que se encuentra en el directorio **src** a un archivo que se encuentra en el directorio **web**: css, imágenes, jquery, js lo haremos de la siguiente manera:

- Incluir código html de otra pagina.html.twig a nuestra pagina.html.twig principal: se utiliza el comando:

```
{% include Bundle_principal_vistas:carpeta_default:nombre_pagina.html.twig %}
```

Ejemplo:



- Incluir archivos internos alojados en la carpeta **web**: (css, js, jquery): usamos el comando **asset** dentro del **href** de nuestro link o script que se usa de la siguiente forma:

`{{ nombre_parametro }}` --> mostrar parámetro devuelto por 1 controlador

`{{ objeto.nombre_parametro }}` --> si un controlador devuelve un objeto

```
{{ asset('directorio_css/archivo_css.css') }}
```

```
{{ asset('directorio_js/archivo_js.js') }}
```

```
{{ asset('directorio_jquery/archivo_jquery.js') }}
```

```

```

Ejemplo:

```
<link rel="stylesheet" href="{{asset('css/bootstrap.min.css')}}">
```

```
<link rel="stylesheet" href="{{asset('css/bootstrap.min.css')}}">
```

```

```

`` : en este caso estamos haciendo que las urls imágenes sean dinámica añadiendo a la url la imagen del producto cogida de la base de datos con el `" ~ "`.

## 8.2 Incluir formularios en Vistas y trabajar con sus datos: GET Y POST

Con respecto al uso de formularios en symfony existen 2 formas de introducir formularios:

- En la misma vista
- Creando una clase donde creamos el formulario y lo incluimos en la vista

El método que voy a explicar es el que se utiliza en la misma vista. Lo primero es situarse en el enrutamiento de nuestra aplicación. Para ello comentamos con anterioridad que las rutas teníamos que ponerlas en el archivo **routing.yml (/config)** en el creamos la ruta de nuestra página index donde tendremos el formulario

```
index.html.twig | routing.yml — config | routing.yml — config | welcome.html.twig | DefaultControl
```

```
inicio:
  path: "/"
  defaults: { _controller: AppBundle:Default:inicio}
```

Este path ("localhost:8000/") llama al método **inicioAction** del controlador **DefaultController** que se encuentra en el Bundle **AppBundle** ( estos datos se pueden ver en el atributo defaults)

En el método redireccionaremos al usuario a la vista (index.html.twig) para que vea el formulario. En el return le decimos la vista a la que lo vamos a devolver. La vista se encuentra en **AppBundle:default:index.html.twig** siendo:

- AppBundle: bundle donde está la vista almacenada
- Default: carpeta donde están las vistas (html.twig) (subrayado en azul)
- Index.html.twig: nombre de la vista que queremos mostrar. (subrayado en rojo)

```
7 use Symfony\Component\HttpFoundation\Request;
8
9 class DefaultController extends Controller
10 {
11     public function inicioAction(Request $request){
12         return $this->render('AppBundle:default:index.html.twig');
13     }
14
15     public function welcomeAction(Request $request){
16         if ('POST' === $request->getMethod()){
17             //POST
18             $name = $request->request->get('nombre');
```



```
index.html.twig  routing.yml — config  routing.yml — config  welcome.html.twig  DefaultController.php
1  <form action="{{ path('bienvenido') }}" method="POST">
2      <p>
3          <label for="nombre">Nombre: </label>
4          <input type="text" name="nombre" id="nombre">
5          <br/> <br/>
6          <label for="apellido">Apellido: </label>
7          <input type="text" name="apellido" id="apellido">
8          <br/> <br/>
9          <input type="submit" value="Enviar">
10         <input type="reset">
11     </p>
12 </form>
13
```

← → ↻ 🏠 📄 localhost:8000

Nombre:

Apellido:

En el action el path “bienvenido” llama a un método del controlador principal que usaremos para comprobar el formulario que es el método welcomeAction:

```
index.html.twig  routing.yml — config  routing.yml — config  welcome.html.twig  DefaultController.php
inicio:
  path: "/"
  defaults: { _controller: AppBundle:Default:inicio}

bienvenido:
  path: "/welcome"
  defaults: { _controller: AppBundle:Default:welcome}
```

```
public function welcomeAction(Request $request){
    if ('POST' === $request->getMethod()){
        //POST
        $name = $request->request->get('nombre');
        $surname = $request->request->get('apellido');
        //GET
        //$name = $request->query->get('nombre'); // GET
        //$surname = $request->query->get('apellido'); // GET

        return $this->render('AppBundle:default:welcome.html.twig', array( "dato1" => $name, "dato2" => $surname));
    }
}
```

Ahora vamos a explicar por encima el método `welcomeAction` que esta encima paso a paso:

**\$request:** todo lo que mandemos en un formulario, el modo y demás se guardan en este parámetro.

- 1- Debemos comprobar el método que estamos usando para enviar el formulario:

```
if ('POST' === $request->getMethod()){ --> se comprueba si el envio es por POST
    ---Operaciones---
}
```

- 2- Dentro de la condición recogemos los datos que enviamos con anterioridad en el formulario:

```
//POST(usamos request abajo)
$name = $request->request->get('nombre');
$surname = $request->request->get('apellido');
```

```
//GET (usamos query abajo)
$name = $request->query->get('nombre');
$surname = $request->query->get('apellido');
```

- 3- Devolvemos al usuario a la vista y enviamos los datos que hemos recuperado del formulario en el return:

```
return $this->render('AppBundle:default:welcome.html.twig', array( "dato1" =>
$name, "dato2" => $surname));
```

De esta manera nos quedaria de la siguiente forma:

```
public function welcomeAction(Request $request){
    if ('POST' === $request->getMethod()){
        //POST
        $name = $request->request->get('nombre');
        $surname = $request->request->get('apellido');
        //GET
        //$name = $request->query->get('nombre'); // GET
        //$surname = $request->query->get('apellido'); // GET

        return $this->render('AppBundle:default:welcome.html.twig', array( "dato1" => $name, "dato2" => $surname));
    }
}
```

Como en el return estamos devolviendo al usuario a la vista **welcome.html.twig** en ella haciendo uso de `{{ nombre_parametro }}` recuperamos los datos que enviamos en el return a la vista

```
public function welcomeAction(Request $request){  
    if ('POST' === $request->getMethod()){  
        //POST  
        $name = $request->request->get('nombre');  
        $surname = $request->request->get('apellido');  
        //GET  
        //$name = $request->query->get('nombre'); // GET  
        //$surname = $request->query->get('apellido'); // GET  
  
        return $this->render('AppBundle:default:welcome.html.twig', array( "dato1" => $name, "dato2" => $surname));  
    }  
}
```

```
1 hola {{dato1}} {{dato2}}  
2
```

← → ↻ 🏠 📄 localhost:8000/welcome

hola JUAN JAPON