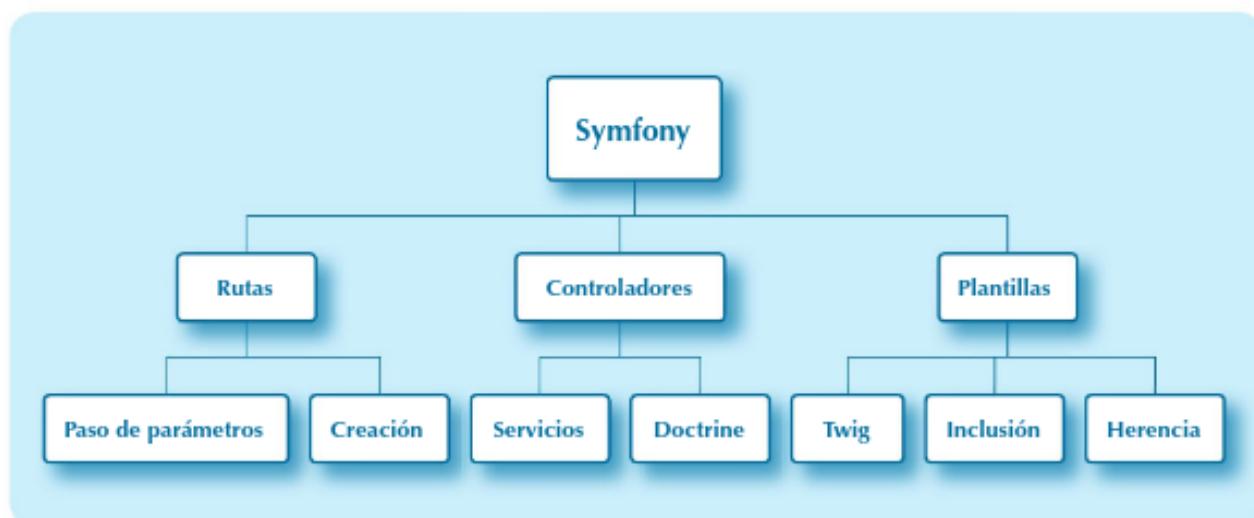


Desarrollo de aplicaciones en Symfony

Objetivos

- ✓ Conocer el patrón MVC y su utilidad para el desarrollo web.
- ✓ Identificar los *frameworks* más importantes que implementan el patrón MVC.
- ✓ Entender la arquitectura de las aplicaciones Symfony.
- ✓ Comprender el formato de rutas de Symfony.
- ✓ Utilizar anotaciones para asociar rutas con controladores.
- ✓ Aprender a utilizar el sistema de plantillas Twig.

Mapa conceptual



Glosario

Controlador. En Symfony, los controladores son las funciones encargadas de procesar la petición del cliente.

Patrón MVC. Patrón de diseño que divide las aplicaciones en tres partes: modelo, vista y controlador.

Plantilla. Fichero en el que se delega la salida. Tiene una parte estática y otra dinámica.

Ruta. Asocian las URL solicitadas por el cliente con el controlador correspondiente.

Servicio. En Symfony, es un objeto que provee alguna funcionalidad útil para el desarrollo, como enviar un correo.

Symfony. Framework para desarrollo web en PHP que sigue el patrón MVC.

Twig. Es el sistema de plantillas por defecto de Symfony.

YAML. Es uno de los formatos disponibles para los metadatos y archivos de configuración, junto con las anotaciones y los ficheros XML.

8.1. El patrón MVC

El patrón MVC divide la aplicación en tres capas: modelo, vista y controlador. Al desacoplar los elementos de la aplicación, se consigue código reusable. Además, permite el desarrollo en paralelo, con equipos independientes para cada capa.

El patrón MVC es uno de los más extendidos para el desarrollo de aplicaciones, no solo en desarrollo web. En la actualidad hay muchos *frameworks* que utilizan este patrón o algunas de las muchas variantes que han surgido.

CUADRO 8.1
Frameworks MVC

Framework	Lenguaje
Spring MVC	Java (JEE)
Symfony	PHP
ASP.NET MVC	ASP
Ruby-on-Rails	Ruby
Angular	JavaScript
TreeFrog	C++

Las tres capas de la aplicación son:

1. *Modelo*: en el modelo está la lógica de negocio. Se encarga de manejar la base de datos de la aplicación.
2. *Vista*: la interfaz gráfica. Una vista muestra una parte del modelo al usuario.
3. *Controlador*: se encarga de recoger las acciones del usuario y, en función de estas, interactúa con el modelo.

Como se puede apreciar, la descripción básica del patrón es muy genérica. Los diferentes *frameworks* difieren en los detalles de implementación.

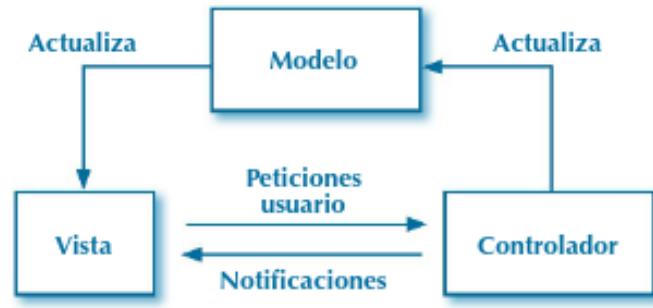


Figura 8.1
Patrón MVC.

8.2. Symfony

Symfony es un *framework* para desarrollo de aplicaciones web en PHP utilizando el patrón MVC. Como todos los *frameworks*, su objetivo es facilitar el desarrollo ofreciendo soluciones para las tareas más habituales.

Por un lado, Symfony plantea las aplicaciones web de una forma determinada a la que el desarrollador debe adaptarse.

Por otro, incluye componentes para muchas de las tareas habituales, como formularios o seguridad. Además, estos componentes son librerías independientes de Symfony y pueden usarse en otros proyectos.

Aunque está impulsado por una empresa, Symfony es un proyecto de código abierto y la comunidad de usuarios ha desarrollado varios componentes interesantes. Muchas aplicaciones conocidas, como Drupal, utilizan Symfony o alguno de sus componentes.

8.2.1. Visión general

La diferencia principal entre las aplicaciones realizadas hasta ahora y Symfony está en cómo se interpretan las URL. Hasta el momento, la interacción entre cliente y servidor se basa en que

el cliente solicita explícitamente al servidor los ficheros que necesita. Por ejemplo, al acceder a localhost/holamundo.php se ejecuta el *script holamundo.php*. De esta manera, los ficheros (y los parámetros que reciben) son la interfaz entre cliente y servidor.

En Symfony, las URL son procesadas por un controlador de *front-end* que analiza las solicitudes para redirigir la petición al controlador adecuado. Un controlador es un método que recibe la petición del cliente, la procesa y genera una respuesta o un reenvío. Los controladores están asociados a una o más rutas: las URL en las que están disponibles.

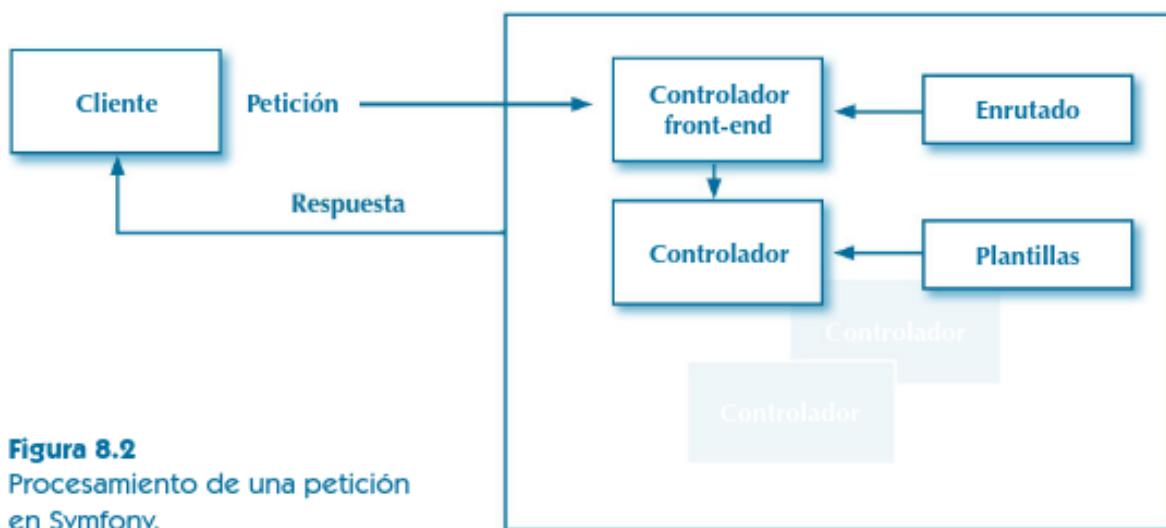


Figura 8.2
Procesamiento de una petición
en Symfony.

Por ejemplo, para definir un controlador que simplemente muestre por pantalla “Hola”, se haría:

```

/**
 * @Route("HolaMundo", name="hola")
 */
public function hola(){
    return new Response('<html><body>Hola</body></html>');
}

```

Como en Doctrine, en Symfony se utilizan anotaciones en bloques de comentarios. En el bloque de comentarios hay una anotación `@route` que sirve para establecer la ruta en la que estará disponible el servidor. Se añade a la ruta base del servidor. Por lo tanto, para acceder al controlador se usaría:

localhost:8000/HolaMundo

De los controladores se espera que devuelvan algo o causen una redirección. En este ejemplo es una cadena de HTML, pero lo normal es utilizar una plantilla.

8.2.2. Instalación

Para crear un nuevo proyecto Symfony se utiliza `composer`:

```
composer create-project symfony/website-skeleton <nOMBRE>
```

Se creará un directorio <nombre> en el que se descargará los componentes de Symfony. La descarga puede llevar varios minutos.

TOMA NOTA

Si en lugar de “website-skeleton” se utiliza “skleton”, se descargan menos componentes. Luego se pueden añadir los que se vayan necesitando.

Los proyectos Symfony incluyen su propio servidor web. Se pone en marcha ejecutando este comando desde el directorio del proyecto:

```
php bin/console server:run
```

La salida del comando informa si se ha podido arrancar el servidor o no. Hay que asegurarse de que no haya otro servidor es-
cuchando previamente en el puerto 8.000.

Si todo ha ido bien, al acceder desde el navegador a se debería ver algo parecido a la captura recogida en la figura 8.3.

8.2.3. Estructura de directorios

Al crear un nuevo proyecto se crea una jerarquía de directorios. En Symfony, la ubicación de los ficheros es muy importante. Se espera que cada tipo de componente esté en un directorio concreto. Al principio puede resultar un poco pesado, pero si se utiliza Symfony para varias aplicaciones, se observan las ventajas de estandarizar la organización del proyecto.

La estructura de directorios de un proyecto recién creado puede observarse en la figura 8.4.

Los directorios más importantes son:

- El directorio raíz del proyecto. Aquí está el fichero .env, donde se almacena la configuración de base de datos y correo.
- /config/packages. Está el fichero security.yaml, que contiene la configuración de seguridad.
- /src/Controller. Para los controladores.
- /src/Entity. Se emplea para guardar las entidades.
- /src/Templates. Se utiliza para las plantillas.

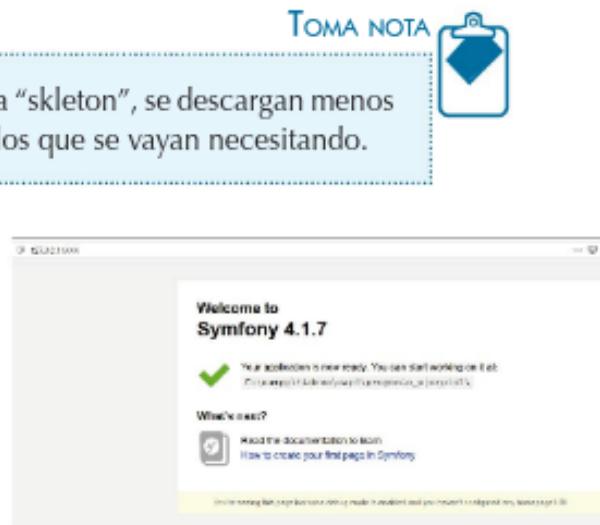


Figura 8.3
Instalación de Symfony.

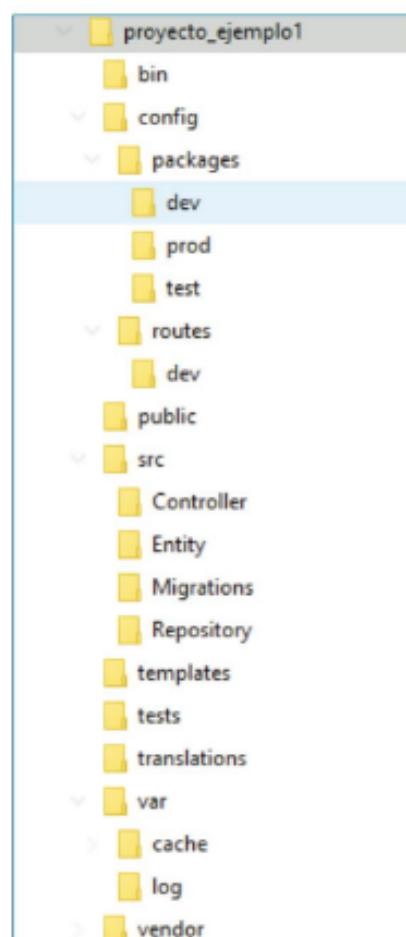


Figura 8.4
Estructura de directorios de un proyecto Symfony.

Actividad propuesta 8.1



Crea un proyecto para probar los ejemplos de este capítulo. Comprueba que el servidor se arranca y revisa la estructura de directorios.

8.3. Controladores

Los controladores son el elemento principal del desarrollo en Symfony. Son métodos que reciben las peticiones del cliente, las procesan y generan una salida o redirección.

Los controladores suelen ser métodos de una clase. Estas clases se denominan *clases controladoras*. Se tienen que guardar en el directorio `/src/Controller` y pueden crearse subdirectorios si es necesario.

Aunque no es obligatorio, pueden heredar de la clase `AbstractController`, que tiene varios métodos prácticos.

El fichero `Ejemplo1.php` contiene una de estas clases, que tiene un método controlador.

```
<?php
// src/Controller/Ejemplo1.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
class Ejemplo1 extends AbstractController{
/**
 * @Route("/hola", name="hola")
 */
public function home_admin(){
    return new Response('<html><body>Hola</body></html>');
}
}
```

Se puede observar que la clase está en el espacio de nombres `App\Controller`. Esto hay que añadirlo en todas las clases controladoras.

En Symfony se recomienda que los controladores sean lo más ligeros posible y contengan poco código. El procesamiento debería delegarse en funciones o clases auxiliares y la salida realizarse a través de plantillas.

8.4. Rutas

Como ya se ha dicho, las aplicaciones Symfony usan un formato de ruta diferente al de las aplicaciones básicas de PHP. Las rutas de los ficheros se sustituyen por rutas que se asocian con un controlador.

8.4.1. Paso de parámetros

El paso de parámetros es más sencillo. Simplemente se añaden a la URL base separados por un carácter '/'. Por ejemplo:

nombre_fichero.php?param1=value1¶m2=value2

se sustituye por:

nombre_ruta/valor1/valor2

Para utilizar esos parámetros hay que incluirlos en la lista de parámetros del controlador, como se hace en este ejemplo, que recibe dos números y muestra su producto.

```
/**
 * @Route("/producto/{num1}/{num2}", name="producto")
 */
public function producto($num1, $num2){
    $producto = $num1 * $num2;
    return new Response("<html><body>" . $producto. "</body></html>");
}
```

Como se puede ver en el ejemplo anterior, la anotación `@Route` puede tener un atributo `name`. Sirve para referirse a esta ruta desde otra parte de la aplicación.



Actividad propuesta 8.2

Escribe un controlador que reciba un número y muestre su factorial. Hay que comprobar que el parámetro sea realmente un número y que no sea negativo.

8.4.2. Valores por defecto

Es posible dar valores opcionales a los argumentos de dos maneras. Una es poniendo el valor por defecto en la lista de argumentos de la función:

```
/**
 * @Route("/defecto1/{num}", name="defecto1")
 */
public function defecto1($num = 3){
    return new Response("<html><body>" . $num. "</body></html>");
}
```

La otra forma es, en la anotación, añadiendo el símbolo '?' y el valor después del argumento.

```
/** * @Route("/defecto2/{num?4}", name="defecto2") */
public function defecto2($num){
    return new Response("<html><body>" . $num. "</body></html>");
}
```

8.4.3. Redirección

Los controladores pueden redirigir a otra ruta en lugar de devolver una página. Si se trata de una ruta sin parámetros, se utiliza simplemente:

```
return $this->redirectToRoute('nombre_ruta');
```

Para indicar la ruta a la que se redirige se utiliza su atributo name. Si la ruta tiene parámetros se pasan en un *array* indicando nombre y valor.

```
return $this->redirectToRoute('nombre_ruta', array('nombrel' => valor1,
    'nombrel' => valor2));
```

En el siguiente ejemplo, se calcula el cuadrado de un número que se pasa como argumento utilizando el controlador del ejemplo anterior. A partir del parámetro num, se redirige a la ruta /producto/num/num.

```
/**
 * @Route("/cuadrado/{num}", name="cuadrado")
 */
public function cuadrado($num){
    return $this->redirectToRoute('producto', array('num1' => $num,
        'num2' => $num));
}
```

8.4.4. Rutas a nivel de clase

La anotación @Route también se puede utilizar sobre las clases, en lugar de sobre los métodos individuales. La ruta indicada se antepone a la de todos los controladores de la clase. En el siguiente ejemplo, para acceder al controlador hola(), habrá que utilizar localhost/base/hola.

```
<?php
// src/Controller/EjemploRutaBase.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;
/**
 * @Route("/base")
 */
class EjemploRutaBase extends AbstractController{
    /**
     * @Route("/hola")
     */
    public function hola(){
        return new Response('<html><body>Hola</body></html>');
    }
}
```

8.4.5. Rutas disponibles

Para ver todas las rutas definidas en la aplicación, se puede utilizar el siguiente comando desde el directorio del proyecto (figura 8.5).

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

Name	Method	Scheme	Host	Path
hola	ANY	ANY	ANY	/hola
producto	ANY	ANY	ANY	/producto/{num1}/{num2}
defecto1	ANY	ANY	ANY	/defecto1/{num}
defecto2	ANY	ANY	ANY	/defecto2/{num}
cuadrado	ANY	ANY	ANY	/cuadrado/{num}
app_empleobasedatos_muestra_equipo	ANY	ANY	ANY	/muestra_equipo
app_empleobasedatos_prueba_correo	ANY	ANY	ANY	/correo
saludo	ANY	ANY	ANY	/saludo/{nombre}
app_empleorutabase_hola	ANY	ANY	ANY	/base/hola
_twig_error_test	ANY	ANY	ANY	/_error/{code}._format
_wdt	ANY	ANY	ANY	/_wdt/{tokens}
_profiler_home	ANY	ANY	ANY	/_profiler/
_profiler_search	ANY	ANY	ANY	/_profiler/search
_profiler_search_bar	ANY	ANY	ANY	/_profiler/search_bar
_profiler_phpinfo	ANY	ANY	ANY	/_profiler/phpInfo
_profiler_search_results	ANY	ANY	ANY	/_profiler/{token}/search/results
_profiler_open_file	ANY	ANY	ANY	/_profiler/open
_profiler	ANY	ANY	ANY	/_profiler/{token}
_profiler_router	ANY	ANY	ANY	/_profiler/{token}/router
_profiler_exception	ANY	ANY	ANY	/_profiler/{token}/exception
_profiler_exception_css	ANY	ANY	ANY	/_profiler/{token}/exception.css

Figura 8.5

Rutas disponibles en la aplicación.

En los ejemplos anteriores los controladores devuelven un objeto

Responde que se crea con una cadena de HTML. En general, la salida de los controladores en Symfony se realiza a través de plantillas, ya sea HTML, XML o cualquier otro formato.

Estas plantillas contienen la parte estática de la página y también la lógica de presentación. De esta manera, se desacopla la lógica de negocio de la de presentación. Pueden recibir argumentos, que son los datos que hay que mostrar.

Por ejemplo, para mostrar una tabla de empleados se puede utilizar una plantilla que reciba como argumento un *array* de empleados. La plantilla se encarga de generar las etiquetas correspondientes a la tabla.

8.5.1. Introducción a Twig

Se pueden utilizar plantillas en PHP o utilizar la librería Twig, que es la opción por defecto. El siguiente ejemplo es una plantilla HTML con Twig.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Saludo</title>
    </head>
    <body>
        Hola {{ nombre }}
    </body>
</html>
```

La única parte dinámica es `{{ nombre }}`. Con esta notación se hace referencia a un parámetro de la plantilla. Al mostrar la plantilla, esa parte se sustituirá por el valor del parámetro nombre.

RECUERDA

- ✓ La extensión de las plantillas es `<formato_generado>.twig`. Por ejemplo, `xml.twig` o `html.twig`.

El controlador `saludo()` recibe como parámetro un nombre y devuelve una página web con el texto “Hola <nombre>”, utilizando la plantilla anterior.

```
<?php
// src/Controller/EjemploPlantillas.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
class EjemploPlantillas extends AbstractController{
/**
 * @Route("/saludo/{nombre}", name="saludo")
 */
public function saludo($nombre){
    return $this->render('saludo.html.twig', array ('nombre' =>
    $nombre));
}
}
```

El controlador pasa el parámetro a la plantilla, que genera la página completa. De esta manera, si se accede a `localhost/saludo/Paco`, el navegador mostrará “Hola Paco”.

Para mostrar una plantilla desde un controlador se utiliza el método `render` (se hereda de `AbstractController`). Este método recibe la ruta de la plantilla y, opcionalmente, un `array` con parámetros. Las claves del `array` tienen que coincidir con los nombres de los parámetros que se usan en la plantilla.

En Twig hay tres tipos de etiquetas:

- `{{ ... }}`. Para introducir el valor de un parámetro o expresión.
- `{% ... %}`. Para introducir lógica o definir secciones.
- `{# ... #}`. Para comentarios.

En las plantillas puede haber estructuras condicionales y bucles. Por ejemplo, la siguiente plantilla recibe como parámetro un número y muestra un mensaje según sea positivo o no.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Saludo</title>
    </head>
    <body>
        {% if numero > 0 %}
            Positivo
        {% else %}
            Menor o igual que 0
        {%endif%}
    </body>
</html>
```

Para probarla, se puede utilizar este controlador.

```
/**
 * @Route("/positivo/{num}", name="positivo")
 */
public function positivo($num){
    return $this->render('if.html.twig', array('numero'=> $num));
}
```



Actividad propuesta 8.3

Escribe un controlador que reciba un número y muestre su factorial utilizando una plantilla. La plantilla recibirá el resultado y un parámetro llamado error. Si error es TRUE, en lugar del resultado hay que mostrar un mensaje apropiado.

El siguiente ejemplo es más complicado. Utiliza un bucle para crear las filas de una tabla. Recibe el argumento filas, que se espera que sea un *array* o similar (iterable). Para cada elemento del *array* introduce una fila usando los campos de ese elemento. Estos elementos tienen que tener campos *codigo* y *nombre*.

```
!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Saludo</title>
    </head>
    <body>
        <table>
            <tr><th>Código</th><th>Nombre</th></tr>
            {%for fila in filas %}
                <tr><td>{{ fila.codigo }}</td>
                    <td>{{ fila.nombre }}</td></tr>
            {%endfor%}
        </table>
    </body>
</html>
El controlador tabla usa la plantilla con un array.
/** 
 * @Route("/tabla/", name="tabla")
 */
public function tabla(){
    $filas = array(array('codigo'=> '1', 'nombre' =>'Sevilla' ),
    array('codigo'=> '2', 'nombre' =>'Madrid' ));
    return $this->render('tabla.html.twig', array ( 'filas' => $filas));
}
```

8.5.2. Rutas en plantillas

Para introducir una ruta dentro de la plantilla, por ejemplo, en el atributo `href` de un vínculo, se usa la función `path()`. Si no tiene parámetros, es simplemente:

```
{% path('nombre_ruta') %}
```

Si hay parámetros se añaden en un array.

```
{% path('nombre_ruta', {'param1': valor1, 'param2': valor2}) %}
```

También existe la función `url()`, que devuelve una ruta absoluta y se usa de la misma forma.

8.5.3. Inclusión y herencia

Es posible combinar varias plantillas mediante inclusión y herencia. La inclusión consiste simplemente en insertar una plantilla dentro de otra. Por ejemplo, si varias páginas tienen una cabecera común, se puede hacer una plantilla cabecera:

```
<a href="{{ path('categorias') }}>Home</a>
<a href="{{ path('carrito') }}>Carrito</a>
<a href="{{ path('logout') }}>Cerrar sesión</a>
```

e incluirla en las demás plantillas con:

```
{% include('cabecera.html.twig') %}
```

La herencia entre plantillas es parecida a la herencia entre objetos. La plantilla que hereda puede modificar o ampliar la plantilla base. Una plantilla base podría ser así:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>{% block title %}Aplicación de pedidos{% endblock %}</title>
  </head>
  <body>
    <header>
      {% include('cabecera.html.twig') %}
    </header>
    <hr>
    {% block body %}{% endblock %}
  </body>
</html>
```

Esta plantilla:

- Incluye la plantilla cabecera.
- Define dos bloques con las etiquetas `{% block ... %}` y `{% endblock ... %}` para el título y el cuerpo.

Las plantillas que hereden de esta podrán sustituir los bloques, algo similar a la sobrescripción de métodos. Por ejemplo, una plantilla que utilice:

```
{% extends 'base.html.twig %}
{% block title %}Lista de categorías{% endblock %}
```

sería como la plantilla base pero cambiaría el título.

RECUERDA

En las plantillas se pueden usar las siguientes variables:

- `app.user`. El usuario actual de la aplicación. Disponible si usa el sistema de usuarios de Symfony (ver más adelante).
- `app.session`. Las variables de sesión.
- `app.request`. El objeto Request al que se está respondiendo.

8.6. Servicios

En Symfony hay una serie de objetos llamados *servicios*, que proveen funcionalidad útil para el desarrollo, como registro o envío de correos. Para utilizarlos dentro de un controlador, hay que añadir un parámetro con la clase adecuada (*type-hinting*).

Uno de estos objetos es `Request`, que tiene información sobre la petición que recibe el servidor. Se utiliza en el siguiente ejemplo:

```
/**
 * @Route("/testRequest", name = "testRequest")
 */
public function testRequest(Request $request){
    $ip = $request->getClientIp();
    return new Response(
        '<html><body>IP: '.$ip.'</body></html>');
}
```

Otro servicio importante es `SessionInterface`, que sirve para manejar las variables de sesión, como se puede ver en este ejemplo que utiliza dos controladores. El primer controlador crea la variable de sesión y redirige al segundo, que la muestra.

```
/**
 * @Route("/sesion1", name = "sesion1")
 */
public function sesion1(SessionInterface $session){
    $session->set("variable", 100);
    return $this->redirect(Route('sesion2'));
}
```

```

    /**
 * @Route("/sesion2", name = "sesion2")
 */
public function sesion2(SessionInterface $session){
    $var = $session->get("variable");
    return new Response('<html><body>' . $var . '</body></html>');
}

```

- Hay que incluir un parámetro `SessionInterface` en los dos controladores. Representa la sesión actual.
- Con el método `set()`, se asigna valor a una variable de sesión. El primer argumento es el nombre de la variable y el segundo, su valor.
- El método `get()` devuelve el valor de la variable cuyo nombre se pasa como argumento.

Se puede ver la lista de los servicios disponibles ejecutando el comando:

```
php bin/console debug:autowiring
```

desde el directorio del proyecto. Con la instalación *website-skeleton* se instalan muchos servicios, así que la salida del comando es muy larga.

8.7. Bases de datos

Aunque se pueden usar otros, Doctrine es el ORM por defecto de Symfony. La configuración de la base de datos se introduce en el fichero `.env`. Para utilizar la base de datos del capítulo anterior (Doctrine), se utilizaría:

```
DATABASE_URL=mysql://root:@127.0.0.1:3306/doctrine
```

Hay que colocar las entidades de la aplicación en el directorio `/scr/Entity`. Se pueden utilizar subcarpetas para organizarlas.

Para usar Doctrine en un controlador, hay que conseguir el `$entityManager` usando:

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

Este es el mismo `$entityManager` del capítulo anterior y se maneja de la misma manera, como se puede comprobar en el siguiente controlador, que utiliza Doctrine para cargar el equipo con código 1. Utiliza la entidad del capítulo anterior.

```

    /**
 * @Route("/mostrar_equipo")
 */
public function mostrar_equipo(){
    $entityManager = $this->getDoctrine()->getManager();
    $eq = $entityManager->find(Equipo::class, 1);
    $nombre = $eq->getNombre();
    return new Response('<html><body>' . $nombre . '</body></html>');
}

```



Actividad propuesta 8.4

Escribe un controlador que reciba el código de un equipo y muestre sus datos utilizando una plantilla. Si el equipo no existe hay que mostrar un mensaje de error.

8.8. Formularios

Los formularios son una de las tareas más habituales en el desarrollo web. Symfony tiene un componente específico. Al principio puede resultar complejo, pero se puede utilizar también en proyectos que no usen Symfony.

Para crear un formulario:

- Se obtiene un objeto `FormBuilder` a partir del método `CreateFormBuilder()`, de `AbstractController`.
- Los controles del formulario se añaden a este objeto utilizando el método `add()`.
- Una vez añadidos todos los campos del formulario, se obtiene el formulario con el método `getForm()`.

Por defecto, los formularios se envían a su propia ruta. Para diferenciar entre cuándo se solicitan los datos y cuándo se envían, se puede utilizar `$form->isSubmitted()`.

El controlador `formuHola` muestra un formulario para que el usuario introduzca nombre y apellido. Al enviarlo, muestra un saludo con esos datos.

```
class EjemploFormularios extends AbstractController{
    /**
     * @Route("/formuHola", name = "formuHola")
     */
    public function formuHola(Request $request) {
        $form = $this->createFormBuilder()
            ->add('nombre', TextType::class)
            ->add('apellido', TextType::class)
            ->add('Enviar', SubmitType::class, array('label'=>'Sumar'))
            ->getForm();
        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid()) {
            $datos = $form->getData();
            $nombre = $datos['nombre'];
            $apellido = $datos['apellido'];
            return new Response('<html><body>Hola '. $nombre.' '. $apellido
                .'

```

La plantilla para mostrar un formulario es simplemente:

```
 {{ form_start(form) }}  
 {{ form_widget(form) }}  
 {{ form_end(form) }}
```

Symfony se encarga de generar el HTML a partir del objeto \$form.

TOMA NOTA

Por supuesto, también se pueden utilizar formularios HTML directamente en las plantillas TWIG, como cualquier elemento de HTML.



8.9. Envío de correo

Symfony utiliza la librería SwiftMailer para enviar correos. Está incluida en la instalación con opción website-skeleton. El servidor de correo se configura con la variable MAILER_URL, que está en fichero .env. Para usar una cuenta de Gmail se utiliza:

```
MAILER_URL=gmail://<usuario>:<contraseña>@localhost
```

RECUERDA

- ✓ Hay que permitir el acceso a "Aplicaciones menos seguras" en la configuración de cuenta de Gmail.

Los correos se crean utilizando la clase SwiftMessage. Enviar un correo es muy fácil, como se puede ver en este ejemplo:

```
/**  
 * @Route("/correo")  
 */  
public function prueba_correo(\Swift_Mailer $mailer){  
    $message = new \Swift_Message();  
    $message->setFrom('pruebas@consymfony.com');  
    $message->setTo('direccion@consymfony.com');  
    $message->setBody("Pruebas");  
    $mailer->send($message);  
    return new Response('<html><body>Enviado</body></html>');  
}
```

Para el cuerpo del correo también se puede utilizar una plantilla.



Actividad propuesta 8.5

Escribe un controlador para enviar un correo electrónico que reciba como parámetros (al menos) la dirección de destino y el cuerpo del correo.

8.10. Seguridad. Usuarios y roles

Symfony cuenta con un componente de seguridad basado en la idea de usuarios y roles. Aunque no es obligatorio, resulta muy útil que las aplicaciones lo utilicen.

El componente de seguridad se configura a través del fichero `/config/packages/security.yaml`. Inicialmente tiene esta estructura:

```
security:
    providers:
        firewalls:
            dev:
                pattern: ^/(_profiler|wdt)|css|images|js/
                security: false
            main:
                anonymous: true
    access_control:
```

TOMA NOTA

En los ficheros YAML la tabulación es parte de la sintaxis.



- Dentro de la sección `providers` se define el *proveedor de usuarios*, es decir, de dónde obtiene Symfony los usuarios. Hay varias opciones, pero la más habitual es que este proveedor sea una base de datos.
- La sección `firewalls` es para la autenticación de usuarios. El `firewall dev` es para uso interno de Symfony. Con `anonymous: true`, se permite el acceso a usuarios que no hayan hecho *login*.
- La sección `access_control` permite limitar el acceso a determinadas rutas según el rol del usuario.

Para relacionar el sistema de usuarios de Symfony con una tabla de usuarios hay que indicar a Symfony qué entidad los representa. Veamos cómo configurar el fichero para utilizar la tabla de restaurantes de la aplicación de pedidos del capítulo 4.

```
security:
    providers:
        pedidos:
            entity:
                class: App\Entity\Restaurante
                property: correo
```

```
encoders:
    App\Entity\Restaurante:
        algorithm: plaintext
```

Se define un proveedor con nombre *pedidos*. Se asocia con la entidad Restaurante (que representa la tabla de restaurantes). En *property* se indica el atributo que se usa como nombre de usuario para hacer login. La sección *encoders* se usa para especificar cómo se encripta la clave; en este caso, texto plano.

La entidad que representa a los usuarios debe implementar las interfaces:

1. UserInterface, con los métodos:

- getUsername()*. Devuelve el nombre de usuario, que será uno de los atributos de la clase.
- getPassword()*. Devuelve la clave del usuario, que será otro atributo.
- getRoles()*. Devuelve un *array* con los roles del usuario.
- getSalt()*. La sal es un parámetro que usan algunos algoritmos de encriptación. Si se usó, hay que indicarlo.
- eraseCredentials()*. Para borrar datos privados de la entidad antes de serializarla.

2. Serializable:

- serialize()*. Tiene que serializar al menos el nombre de usuario, la clave y, si se usó, la sal.
- unserialize()*. Realiza la operación inversa.

La entidad Restaurante se crea como se vio en el capítulo anterior y se le añaden los métodos de las interfaces. Los métodos *getUsername()* y *getPassword()* tienen que devolver el correo y la clave del restaurante, respectivamente.

RECUERDA

- ✓ En el capítulo 9 se utiliza el componente de seguridad de Symfony para rehacer la aplicación de pedidos.

```
<?php
// src/Entity/Restaurante.php
namespace App\Entity;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;
/**
 * @ORM\Entity @ORM\Table(name="restaurantes")
 */
class Restaurante implements UserInterface, \Serializable{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
```

```
* @ORM\Column(type="integer", name="CodRes")
*/
private $codRes;
/**
* @ORM\Column(type="string", name = "Correo")
*/
private $correo;
/**
* @ORM\Column(type="string", name = "Clave")
*/
private $clave;
/**
* @ORM\Column(type="string", name = "Pais")
*/
private $pais;
/**
* @ORM\Column(type="string", name = "CP")
*/
private $CP;
/**
* @ORM\Column(type="string", name = "Ciudad")
*/
private $ciudad;
/**
* @ORM\Column(type="string", name = "Direccion")
*/
private $direccion;
/**
* @ORM\Column(type="integer", name = "rol")
*/
private $rol;
public function getRol(){
    return $this->rol;
}
/**
* @param mixed $rol
*/
public function setRol($rol){
    $this->rol = $rol;
}
public function getCodRes(){
    return $this->codRes;
}
public function getCorreo(){
    return $this->correo;
}
public function setCorreo($correo){
    $this->correo = $correo;
}
public function getClave(){
    return $this->clave;
}
public function setClave($clave){
    $this->clave = $clave;
```

```
    }
    public function getPais(){
        return $this->pais;
    }
    public function setPais($pais){
        $this->pais = $pais;
    }
    public function getCP(){
        return $this->CP;
    }
    public function setCP($CP) {
        $this->CP = $CP;
    }
    public function getCiudad(){
        return $this->ciudad;
    }
    public function setCiudad($ciudad){
        $this->ciudad = $ciudad;
    }
    public function getDireccion(){
        return $this->direccion;
    }
    public function serialize(){
        return serialize(array(
            $this->codRes,
            $this->correo,
            $this->clave,
        ));
    }
    public function unserialize($serialized){
        list (
            $this->codRes,
            $this->correo,
            $this->clave,
        ) = unserialize($serialized);
    }
    public function setDireccion($direccion){
        $this->direccion = $direccion;
    }
    public function getRoles(){
        return array('ROLE_USER');
    }
    public function getPassword(){
        return $this->getClave();
    }
    public function getSalt(){
        return;
    }
    public function getUsername(){
        return $this->getCorreo();
    }
    public function eraseCredentials(){
```

```

        return;
    }
}

```

Como se puede ver, hay un método `getRoles()`. Este método devuelve un *array* con los roles asignados al usuario. Si la aplicación no define roles, conviene utilizar el rol predefinido `ROLE_USER`. El programador puede definir más siempre que empiecen por `ROLE_`.

8.10.1. Control de acceso

La sección `access_control` del fichero `security.yaml` permite limitar el acceso a determinadas rutas según el rol del usuario. La siguiente configuración restringiría el acceso a todas las rutas que empiecen por “admin”.

- { path: ^/admin, roles: ROLE_ADMIN }

Solo podrían acceder a las mismas los usuarios con rol `ROLE_ADMIN`.

Otra posibilidad es controlar el acceso mediante código en los controladores. Por ejemplo, para asegurarse de que el usuario haya hecho *login*.

```
$this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');
```

Si no ha hecho *login*, se le redirige al formulario de *login*.

También se puede usar para comprobar roles:

```
$this->denyAccessUnlessGranted(ROLE_ADMIN');
```

La última opción es usar una anotación. Puede ponerse sobre una clase o sobre un método. La siguiente anotación hace que el controlador solo esté disponible para los usuarios con `ROLE_USER`. Es el rol por defecto, así que sirve para que solo puedan acceder quienes hayan abierto sesión.

```

/**
 * @Security("has_role('ROLE_USER')")
 */

```

8.10.2. Abrir sesión

Para asociar un formulario de *login* con el sistema de usuarios de Symfony, se añade un elemento `form_login` al `firewall main`.

```

security:
  providers:
    pedidos:
      entity:

```

```

        class: App\Entity\Restaurante
        property: correo
encoders:
    App\Entity\Restaurante:
        algorithm: plaintext
firewalls:
    dev:
        pattern: ^/(_profiler|wdt)|css|images|js/
        security: false
    main:
        anonymous: true
        form_login:
            login_path: login
            check_path: login
            default_target_path: ruta_defecto
        provider: pedidos

```

Hay que indicar:

- *login_path*: la ruta a la que se envía a un usuario que no ha hecho *login* si intenta acceder a una ruta que lo requiera.
- *check_path*: la ruta a la que se envía el formulario de *login*.
- *default_target_path*: ruta por defecto a la que enviar tras hacer *login* con éxito. Opcional.
- *provider*: el proveedor de usuarios. Symfony se encargará de validar usuario y contraseña usando la tabla asociada.

Con la configuración del ejemplo, hay que crear un controlador en la ruta *login* que se encargue de mostrar el formulario. No tiene que consultar a la base de datos para ver si los datos son correctos, lo hace Symfony. Puede ser tan sencillo como:

```

class PedidosLogin extends AbstractController{
    /**
     * @Route("/login", name="login")
     */
    public function login(){
        return $this->render('login.html.twig');
    }
}

```

La plantilla con el formulario también es muy sencilla, solo hay que ocuparse de que **action** apunte a la ruta correcta.

```

{# templates/login.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Formulario de login</title>
    </head>
    <body>
        <form action="{{ path('login') }}" method="post">
            Usuario

```

```
<input type="text" id="username" name="_username" />
Clave
<input type="password" id="password" name="_password" />
<button type="submit">login</button>
</form>
</body>
</html>
```

8.10.3. Cerrar sesión

Con el sistema de usuarios de Symfony no hace falta escribir un controlador para cerrar sesión, se puede configurar en el fichero de seguridad. Dentro del *firewall* se añade un elemento *logout*.

```
security:
...
firewalls:
    dev:
        ...
    main:
        ...
        logout:
            path: /logout
            target: /login
```

De esta manera, si se redirige al usuario a la ruta indicada en *path* (en este caso, /logout), Symfony cierra la sesión y le redirige a la ruta indicada en *target* (en este caso, /login).

Aunque no haga falta escribir un controlador, la ruta tiene que existir. Para definir una ruta sin controlador se usa el fichero **/config/routes.yaml**. Hay que añadir:

```
logout:
    path: /logout
```

Con esta configuración, para cerrar sesión desde cualquier punto de la aplicación basta con redirigir a *logout*.

Resumen

- El patrón MVC reparte la lógica de la aplicación en tres capas: modelo, vista y controlador.
- La arquitectura de las aplicaciones en Symfony está basada en este patrón.
- Los proyectos de Symfony tienen una estructura de directorios marcada. Cada tipo de componente tiene un directorio asignado.
- La configuración de los servidores de base de datos y correo se hace en el fichero *.env*.

- Los controladores son el componente central de Symfony. Se encargan de procesar las peticiones del cliente y devolver una respuesta.
- La configuración de rutas asocia las URL que solicita el cliente con el controlador correspondiente. Se puede introducir mediante anotaciones.
- La salida se realiza habitualmente mediante una plantilla. Las plantillas contienen la lógica de presentación.
- El ORM por defecto en Symfony es Doctrine, pero se pueden usar otros.
- El componente de seguridad de Symfony se puede integrar con una base de datos de usuarios.
- La configuración de seguridad se guarda en el fichero security.yaml.



Ejercicios propuestos

Para estos ejercicios utiliza la base de datos Doctrine, del capítulo anterior.

1. Escribe un controlador que reciba dos números y muestre el resto al dividir el primero por el segundo. El segundo parámetro es opcional, con valor por defecto 2.
2. Escribe un controlador que muestre una tabla con los datos de todos los jugadores.
3. Escribe un controlador que reciba el código de un equipo y muestre una tabla con los datos de sus jugadores.
4. Escribe un controlador con un formulario para el ejercicio anterior.
5. Escribe un controlador con un formulario para enviar correo.
6. Escribe un controlador que devuelva una lista con vínculos a los controladores de los ejercicios propuestos 4 y 5.

ACTIVIDADES DE AUTOEVALUACIÓN

1. Las rutas de la aplicación se definen:
 - a) Con anotaciones.
 - b) En el fichero .env.
 - c) En el directorio de los controladores.
2. Twig permite:
 - a) Inclusión de plantillas.
 - b) Herencia de plantillas.
 - c) Ambas.

3. El rol por defecto de los usuarios en Symfony es:
 a) ROLE_BASIC
 b) ROLE_USER
 c) ROLE_DEFAULT
4. Las plantillas se encargan de:
 a) La salida.
 b) El enrutado.
 c) Manejar la base de datos.
5. Los controladores son:
 a) Objetos que controlan el acceso no autorizado.
 b) Métodos asociados a una ruta.
 c) Métodos que validan la información introducida por el usuario.
6. Para redireccionar se utiliza el método:
 a) toURL().
 b) header("Location:").
 c) redirectToRoute().
7. Para incluir una ruta en una plantilla se utiliza:
 a) path().
 b) url().
 c) Ambas.
8. En el fichero .env se configura:
 a) La base de datos.
 b) La seguridad de la aplicación.
 c) El directorio de las plantillas.
9. El ORM por defecto de Symfony es:
 a) Propel.
 b) Doctrine.
 c) Hibernate.
10. Las entidades se guardan en el directorio:
 a) /src/Entity.
 b) /src/Controller.
 c) Cualquiera, se especifica en .env.

SOLUCIONES:

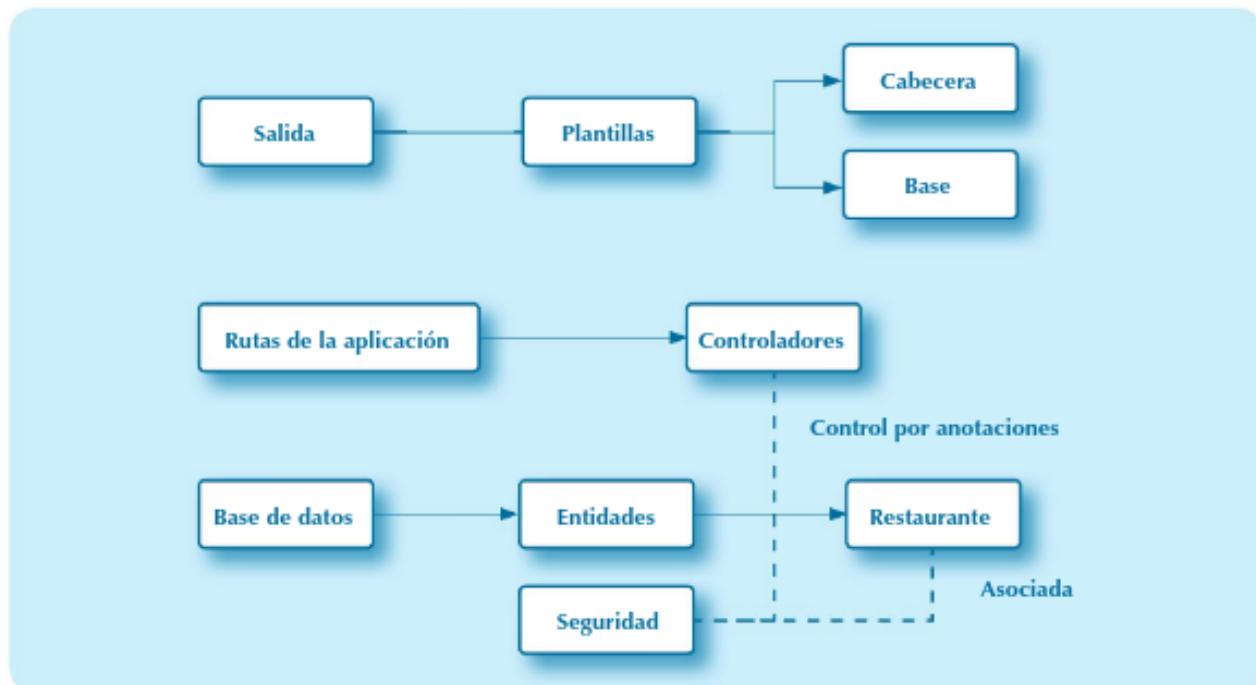
- | | | |
|---|--|--|
| 1. a <input type="checkbox"/> b <input type="checkbox"/> c | 5. <input type="checkbox"/> a b <input type="checkbox"/> c | 9. <input type="checkbox"/> a b <input type="checkbox"/> c |
| 2. <input type="checkbox"/> a <input type="checkbox"/> b c | 6. <input type="checkbox"/> a <input type="checkbox"/> b <input type="checkbox"/> c | 10. <input type="checkbox"/> a <input type="checkbox"/> b c |
| 3. <input type="checkbox"/> a b <input type="checkbox"/> c | 7. <input type="checkbox"/> a <input type="checkbox"/> b c | |
| 4. a <input type="checkbox"/> b <input type="checkbox"/> c | 8. a <input type="checkbox"/> b <input type="checkbox"/> c | |

Aplicación de pedidos en Symfony

Objetivos

- ✓ Rediseñar la aplicación de pedidos con Symfony.
- ✓ Utilizar el sistema de plantillas Twig.
- ✓ Integrar el sistema de usuarios de Symfony.
- ✓ Manejar la base de datos de pedidos utilizando Doctrine.
- ✓ Valorar las ventajas de este enfoque.

Mapa conceptual



Glosario

@Security("has_role()"). Anotación para restringir el acceso a una ruta a los usuarios con un rol determinado.

app.user. Si se usa el sistema de usuarios de Symfony, esta variable está disponible en las plantillas. Tiene información sobre el usuario.

Proveedor de usuarios. El origen de los datos de los usuarios para el sistema de seguridad Symfony.

Rol. Asignando roles a los usuarios se puede restringir el acceso a determinadas partes de la aplicación.

UserInterface. En el sistema de usuarios de Symfony, la entidad que representa a los usuarios tiene que implementar esta interfaz.

9.1. Diseño de la aplicación

En este capítulo se rediseña la aplicación de pedidos del capítulo 4 como una aplicación de una sola página utilizando Symfony. La nueva aplicación tendrá el mismo aspecto y funcionalidad que la anterior. Como se podrá apreciar, con un buen diseño y conociendo el *framework*, el desarrollo con Symfony es verdaderamente rápido.

Gran parte del diseño de la aplicación se mantiene sin cambios:

- La base de datos. Utiliza la misma que las otras aplicaciones.
- El mapa de pantallas.
- La variable para el carrito.

Los cambios son:

- a) La salida se hace con plantillas Twig.
- b) La base de datos se maneja con Doctrine.
- c) Los ficheros de la aplicación original son sustituidos por métodos controladores.

RECUERDA

✓ Si no recuerdas bien la aplicación, revisa el apartado 4.3.

9.1.1. Plantillas

Como se recordará, después del *login* todas las páginas de la aplicación tienen una estructura común. Una cabecera en la parte superior y una sección principal cuyo contenido varía. Para representarla se utilizará esta jerarquía de plantillas.

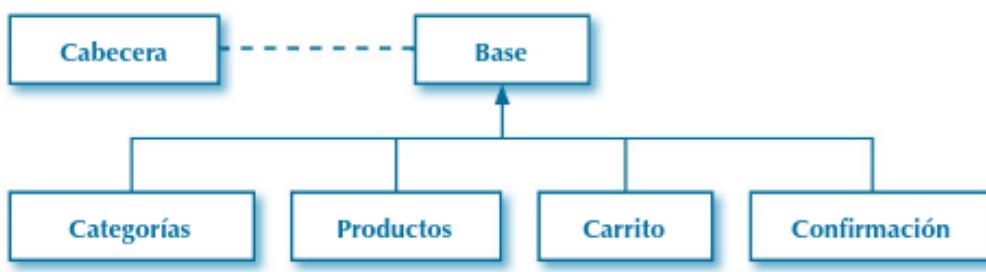


Figura 9.1
Jerarquía de plantillas.

- Una plantilla contiene la cabecera de la página.
- La plantilla base incluye la cabecera y define la estructura de la página.
- El resto de las plantillas heredan de la plantilla base y la sobrescriben para incluir su contenido específico.

Además, hay plantillas para:

- Formulario de *login*.
- El correo de confirmación de pedido.

9.1.2. Entidades

Para poder utilizar Doctrine hay que crear las entidades de la tabla de base de datos pedidos como se vio en el capítulo 7. Habrá que hacer una entidad por cada tabla. Es importante definir las asociaciones:

- Entre Categorías y Productos habrá una asociación bidireccional. Es decir, cada producto contendrá una referencia a su categoría y cada categoría tendrá un atributo con los productos de esa categoría. Así, se podrán obtener los productos de la categoría sin tener que hacer consultas.
- Entre Pedidos y Restaurantes habrá una asociación unidireccional. Es decir, cada pedido contendrá una referencia al restaurante que lo realiza, pero los restaurantes no tendrán un atributo con sus pedidos.
- Para la tabla de PedidosProductos, que se relaciona con Pedidos y Productos, usaremos dos asociaciones unidireccionales. En PedidosProductos habrá referencias a Pedidos y Productos, pero al contrario no.

La entidad Restaurantes es especial, ya que representa a los usuarios de la aplicación y se va a integrar en el componente de seguridad de Symfony. Como se comentó en el capítulo anterior, el sistema de usuarios de Symfony utiliza el concepto de *roles*, pero en esta aplicación no se diferencia entre tipos de usuarios. En la tabla de restaurantes hay un campo rol, pero está presente para futuras ampliaciones.

Por lo tanto, el único rol que se utiliza es el rol por defecto, ROLE_USER.

9.1.3. Rutas de la aplicación

Hay que transformar los ficheros de la aplicación del capítulo 4 en controladores. Para cada fichero (solicitado directamente por el cliente) habrá que hacer un método controlador equivalente:

- Hay que escoger una ruta para cada controlador; si recibe parámetros, hay que incluirlos en la ruta.
- Si la salida es HTML, se realiza mediante una plantilla.
- Si la salida es una redirección, hay que ajustar la ruta.

CUADRO 9.1

Rutas de la aplicación

Ruta	Descripción	Parámetros	Plantilla/redirección
/login name = 'login'	Formulario de <i>login</i>	\$_POST['usuario'] \$_POST['clave']	login.html.twig /categorias
/logout name = 'logout'	Cierra la sesión		/login
/categorias name = 'categorias'	Muestra la lista de categorías con vínculos a productos/{id}		categorias.html.twig
/productos/{id} name = 'productos'	Muestra los productos de la categoría, permite añadir al carro de la compra	El código de la categoría	productos.html.twig
			[.../...]

CUADRO 9.1 (CONT.)

/carrito name = 'carrito'	Muestra el carro de la compra, permite quitar productos y confirmar el pedido		carrito.html.twig
/anadir name = 'anadir'	Añade productos al carro	\$_POST['cod'] \$_POST['unidades']	/carrito
/eliminar name = 'eliminar'	Elimina productos del carro	\$_POST['cod'] \$_POST['unidades']	/carrito
/realizarPedido name = 'realizarPedido'	Inserta el pedido en la base de datos, envía correos de confirmación y muestra mensajes de error o éxito		pedido.html.twig correo.html.twig

TOMA NOTA



Las rutas "login" y "logout" estarán asociadas con el sistema de usuarios de Symfony. La ruta "logout" no está asociada a ningún controlador. Existe para el sistema de usuarios de Symfony. Al redirigir a un usuario a esa ruta, Symfony cerrará la sesión.

9.2. Implementación

Con todos los elementos de la aplicación bien definidos, se puede proceder a la implementación.

El primer paso es crear un nuevo proyecto:

```
composer create-project symfony/website-skeleton PedidosSymfony
```

Lucgo hay que modificar el fichero `.env` para incluir la configuración de la base de datos y el servidor de correo.

```
DATABASE_URL=mysql://root:@127.0.0.1:3306/pedidos
MAILER_URL=gmail://<usuario>:<clave>@localhost
```



Actividad propuesta 9.1

Configura el fichero `.env` para utilizar una cuenta de Gmail y poder probar el correo de confirmación de pedidos de la aplicación.

En los siguientes apartados se explicará cómo implementar las plantillas, las entidades, los controladores y la seguridad.

9.3. Plantillas

Para comenzar hay que hacer las plantillas, así se pueden ir probando los controladores.

9.3.1. Login

Un formulario sencillo que apunta a la ruta *login*.

```
{# templates/login.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Aplicación de pedidos</title>
    </head>
    <body>
        <form action="{{ path('login') }}" method="post">
            Usuario
            <input type="text" id="username" name="_username" />
            Clave
            <input type="password" id="password" name="_password" />
            <button type="submit">login</button>
        </form>
    </body>
</html>
```

9.3.2. Plantilla base

Una vez abierta sesión, todas las páginas comparten una estructura base definida por esta plantilla.

```
1  {# templates/base.html.twig #}
2  <!DOCTYPE html>
3  <html>
4      <head>
5          <meta charset="UTF-8">
6          <title>{% block title %}Aplicación de pedidos{% endblock %}</title>
7      </head>
8      <body>
9          {% block body %}
10         <header>
11             {% block header %}
12                 {{ include('cabecera.html.twig') }}
13                 <hr>
14             {% endblock %}
15         </header>
16         {% block contenido %}
```

```

17      {% endblock %}
18      {% endblock %} 
19  </body>
20 </html>
```

Se definen varios bloques que serán sobrescritos por las otras plantillas:

- Línea 6: `title`, para el título de la página.
- Líneas 11-14: `header`, que incluye la cabecera.
- Líneas 16-17: `contenido`. Aquí es donde se sitúa el contenido específico de cada sección.

9.3.3. Cabecera

La cabecera está incluida en la plantilla base.

```

Usuario: {{ app.user.correo }}
<a href="{{ path('categorias') }}>Home</a>
<a href="{{ path('carrito') }}>Carrito</a>
<a href="{{ path('logout') }}>Cerrar sesión</a>
```

Incluye:

- El correo del usuario a través de `app.user`.
- Los vínculos para la lista de categorías, el carrito y cerrar sesión. Utiliza la función `path()` con el nombre de la ruta correspondiente.

9.3.4. Lista de categorías

Extiende la plantilla base y modifica el bloque `contenido` para mostrar la lista de categorías.

```

{# templates/categorias.html.twig #}
{% extends 'base.html.twig' %}
{% block title %}Lista de categorías{% endblock %}
{% block contenido %}
<ul>
{% for cat in categorias %}
    <li>
        <a href="{{path('productos',{'id':cat.CodCat})}}>
            {{cat.Nombre}}</a>
    </li>
    {% endfor %}
</ul>
{% endblock %}
```

Esta plantilla recibe como parámetro un *array* con las categorías. Como en las versiones anteriores, los elementos de la lista son vínculos cuyo texto es el nombre de la categoría y que apuntan a la tabla de productos de esa categoría; en este caso, a la ruta `productos/{id}`.

9.3.5. Tabla de productos

Muestra la tabla de productos de una categoría, incluyendo los formularios para añadir. Son como los del capítulo 4, solo cambia el valor de `action`. Ahora, en lugar de `anadir.php` se utiliza el nombre de la ruta correspondiente, `anadir`. Recibe como argumento un `array` de productos.

```

{# templates/productos.html.twig #}
{% extends 'base.html.twig' %}

{% block title %}Tabla de productos{% endblock %}
{% block contenido %}



| Nombre            | Descripción            | Stock            | Peso            |                                                                                                                                                                                                                                                                         |
|-------------------|------------------------|------------------|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Comprar           |                        |                  |                 |                                                                                                                                                                                                                                                                         |
| {{ prod.nombre }} | {{ prod.descripcion }} | {{ prod.stock }} | {{ prod.peso }} | <form action = "{{path('anadir')}}" method = 'POST'>             <input name='unidades' type='number' min ='1' value='1'>             <input type = 'submit' value='Comprar'>             <input name = 'cod' type= 'hidden' value = "{{prod.codProd}}>         </form> |
|                   |                        |                  |                 |                                                                                                                                                                                                                                                                         |



{% endblock %}

```

9.3.6. El carrito de la compra

Muestra una tabla con los datos del carrito de la compra. Recibe un *array* con los datos de los productos y las unidades pedidas. Incluye los formularios para eliminar productos, que también son como los del capítulo 4. En este caso la ruta es **eliminar**.

```
{# templates/carrito.html.twig #}
{% extends 'base.html.twig' %}

{% block title %}Carrito de la compra{% endblock %}

{% block contenido %}

{% if productos is empty %}

<p> El carrito está vacío </p>

{% else %}

<table>
    <tr>
        <th>Nombre</th><th>Descripción</th>
        <th>Stock</th><th>Peso</th><th>Unidades</th><th>Eliminar</th>
    </tr>
    {% for prod in productos %}
```

```

<tr>
    <td>{{ prod.nombre }}</td>
    <td>{{ prod.descripcion }}</td>
    <td>{{ prod.stock }}</td>
    <td>{{ prod.peso }}</td>
    <td>{{ prod.unidades }}</td>
    <td>
        <form action={{path('eliminar')}} method = 'POST'>
            <input name='unidades' type='number' min ='1' value= '1'>
            <input type = 'submit' value='Eliminar'>
            <input name = 'cod' type='hidden'
            value = {{prod.codProd}}>
        </form>
    </td>
</tr>
{% endfor %}
</table>
<a href = {{ path('realizarPedido') }}>Realizar Pedido</a>
{% endif %}
{% endblock %}

```

9.3.7. Confirmación del pedido

Este controlador es muy sencillo, muestra mensajes de error o confirmación. Recibe un código de error y el código del pedido.

```

{% template('pedido.html.twig') %}
{% extends 'base.html.twig' %}
{% block title %}Confirmación de pedido{% endblock %}
{% block contenido %}
    {% if error == 1 %}
        No hay productos en el carrito
    {% elseif error == 2%}
        Error de la base de datos al procesar el pedido,
        consulte con el administrador
    {% else %}
        Pedido {{ id }} realizado con éxito
    {% endif %}
{% endblock %}

```

9.3.8. Correo

Esta plantilla se utiliza para crear el correo. Es una tabla con los datos de los productos del pedido y las unidades. Es como la del carrito de la compra, pero sin formularios.

```

{% template('correo.html.twig') %}
Pedido {{ id }} realizado con éxito

```

```
<table>
    <tr><th>Nombre</th><th>Descripción</th>
        <th>Peso</th><th>Unidades</th>
    </tr>
    {% for prod in productos %}
    <tr>
        <td>{{ prod.nombre }}</td>
        <td>{{ prod.descripcion }}</td>
        <td>{{ prod.peso }}</td>
        <td>{{ prod.unidades }}</td>
    </tr>
    {% endfor %}
</table>
```

Actividad propuesta 9.2



Las plantillas de la tabla de productos, el carrito y el correo tienen muchos elementos en común. ¿Crees que sería buena idea reescribirlas usando herencia e inclusión? Analiza las ventajas e inconvenientes.

9.4. Entidades

Habrá una entidad por cada tabla. La entidad Restaurante representa a los usuarios de la aplicación. Se utiliza la entidad desarrollada en el capítulo anterior.

Se emplea la entidad Producto para la tabla de productos, asociada con Categoría.

```
<?php
// src/Entity/Producto.php
namespace App\Entity;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;
/**
 * @ORM\Entity @ORM\Table(name="productos")
 */
class Producto {
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer", name="CodProd")
     */
    private $codProd;
    /**
     * @ORM\Column(type="string")
     */
}
```

```
private $nombre;
/**
 * @ORM\Column(type="string")
 */
private $descripcion;
/**
 * @ORM\Column(type="float")
 */
private $peso;
/**
 * @ORM\Column(type="integer")
 */
private $stock;
/**
 * @ORM\ManyToOne(targetEntity="Categoria", inversedBy = "productos")
 * @ORM\JoinColumn(name="Categoria", referencedColumnName="CodCat")
 */
private $categoria;
public function getCodProd() {
    return $this->codProd;
}
public function setCodProd($codProd) {
    $this->codProd = $codProd;
}
public function getNombre() {
    return $this->nombre;
}
public function setNombre($nombre) {
    $this->nombre = $nombre;
}
public function getDescripcion() {
    return $this->descripcion;
}
public function setDescripcion($descripcion) {
    $this->descripcion = $descripcion;
}
public function getPeso() {
    return $this->peso;
}
public function setPeso($peso) {
    $this->peso = $peso;
}
public function getStock(){
    return $this->stock;
}
public function setStock($stock) {
    $this->stock = $stock;
}
```

```

    public function getCategoría() {
        return $this->categoría;
    }
    public function setCategoría($categoría) {
        $this->categoría = $categoría;
    }
}

```

La asociación con Categoría es bidireccional. La clase Categoría tiene un campo productos en el que se cargan los productos de la categoría. Esto lo hace Doctrine cuando se accede al campo.

```

<?php
// src/Entity/Categoría.php
namespace App\Entity;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;
/**
 * @ORM\Entity @ORM\Table(name="categorias")
 */
class Categoría {
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer", name="CodCat")
     */
    private $codCat;
    /**
     * @ORM\Column(type="string")
     */
    private $nombre;
    /**
     * @ORM\Column(type="string")
     */
    private $descripción;
    /**
     * Bidireccional
     * @ORM\OneToMany(targetEntity="Producto", mappedBy="categoría")
     */
    private $productos;
    public function getCodCat() {
        return $this->codCat;
    }
    public function getProductos() {
        return $this->productos;
    }
    public function getNombre() {
        return $this->nombre;
    }
}

```

```
public function setNombre($nombre) {
    $this->nombre = $nombre;
}
public function getDescripcion() {
    return $this->descripcion;
}
public function setDescripcion($descripcion) {
    $this->descripcion = $descripcion;
}
}
```

Se emplea la entidad Pedido para la tabla de pedidos, asociada con Restaurante.

```
<?php
// src/Entity/Pedido.php
namespace App\Entity;
use Doctrine\ORM\Mapping as ORM;
/**
 * @ORM\Entity @ORM\Table(name="pedidos")
 */
class Pedido
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer", name="CodPed")
     */
    private $codPed;
    /**
     * @ORM\Column(type="datetime", name = "Fecha")
     */
    private $fecha;
    /**
     * @ORM\Column(type="integer", name = "Enviado")
     */
    private $enviado;
    /**
     * @ORM\ManyToOne(targetEntity="Restaurante")
     * @ORM\JoinColumn(name="Restaurante", referencedColumnName="CodRes")
     */
    private $restaurante;
    public function getCodPed() {
        return $this->codPed;
    }
    public function getFecha() {
        return $this->fecha;
    }
    public function setFecha($fecha) {
```

```

        $this->fecha = $fecha;
    }
    public function getEnviado() {
        return $this->enviado;
    }
    public function setEnviado($enviado) {
        $this->enviado = $enviado;
    }
    public function getRestaurante() {
        return $this->restaurante;
    }
    public function setRestaurante($restaurante) {
        $this->restaurante = $restaurante;
    }
}

```

Y finalmente, se utiliza la entidad PedidosProducto, que tiene dos asociaciones: con Pedido y con Productos.

```

<?php
// src/Entity/PedidoProducto.php
namespace App\Entity;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;
/**
 * @ORM\Entity @ORM\Table(name="pedidosproductos")
 */
class PedidoProducto{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer", name="CodPedProd")
     */
    private $codPedProd;
    /**
     * @ORM\ManyToOne(targetEntity="Pedido")
     * @ORM\JoinColumn(name="Pedido", referencedColumnName="CodPed")
     */
    private $codPed;
    /**
     * @ORM\ManyToOne(targetEntity="Producto")
     * @ORM\JoinColumn(name="Producto", referencedColumnName="CodProd")
     */
    private $codProd;
    /**
     * @ORM\Column(type="integer", name = "unidades")
     */
    private $unidades;
}

```

```

public function getCodPedProd() {
    return $this->codPedProd;
}
public function getCodPed() {
    return $this->codPed;
}
public function setCodPed($codPed) {
    $this->codPed = $codPed;
}
public function getCodProd() {
    return $this->codProd;
}
public function setCodProd($codProd) {
    $this->codProd = $codProd;
}
public function getUnidades() {
    return $this->unidades;
}
public function setUnidades($unidades) {
    $this->unidades = $unidades;
}
}

```

9.5. Controladores

El controlador de *login* está en la clase `PedidosLogin`; el resto, en `PedidosBase`. Esta clase tendrá una anotación de seguridad para que solo puedan acceder a sus controladores los usuarios que hayan abierto sesión.

9.5.1. Abrir sesión

Este controlador estará asociado al sistema de usuarios de Symfony. Por tanto, no tiene que ocuparse de las comprobaciones, solo de mostrar el formulario.

```

class PedidosLogin extends AbstractController{
    /**
     * @Route("/login", name="login")
     */
    public function login(){
        return $this->render('login.html.twig');
    }
}

```

9.5.2. Lista de categorías

Carga los datos de la tabla de categorías con el método `findAll()` y pasa el resultado a la plantilla.

```
/**
 * @Route("/categorias", name="categorias")
 */
public function mostrarCategorias() {
    $categorias = $this->getDoctrine()
        ->getRepository(Categoría::class)
        ->findAll();
    return $this->render("categorias.html.twig",
        array('categorias'=>$categorias));
}
```

9.5.3. Tabla de productos

Este controlador es el que devuelve la tabla de productos. Recibe el código de la categoría como parámetro. Busca la categoría con el método `find()` y accede a sus productos con el método `getProductos()`, que utiliza la asociación bidireccional. Pasa los productos a la plantilla.

```
/**
 * @Route("/productos/{id}", name="productos")
 */
public function mostrarProductos($id) {
    $productos = $this->getDoctrine()
        ->getRepository(Categoría::class)
        ->find($id)
        ->getProductos();
    if (!$productos) {
        throw $this->createNotFoundException('Categoría no encontrada');
    }
    return $this->render("productos.html.twig", array('productos'=>
        $productos));
}
```

9.5.4. Carrito

A partir de la variable de sesión de carrito, monta un *array* con los datos que hay que mostrar y se los pasa a la plantilla. Para cada producto, busca con `find()` todos sus datos y añade las unidades pedidas.

```
/**
 * @Route("/carrito", name="carrito")
 */
public function mostrarCarrito(SessionInterface $session){
    /* para cada elemento del carrito se consulta la base de datos y se
     recuperan sus datos*/
    $productos = [];
    $carrito = $session->get('carrito');
```

```

/* si el carrito no existe se crea como un array vacío*/
if(is_null($carrito)){
    $carrito = array();
    $session->set('carrito', $carrito);
}
/* se crea array con todos los datos de los productos y la cantidad*/
foreach ($carrito as $codigo => $cantidad){
    $producto = $this->getDoctrine()
        ->getRepository(Producto::class)
        ->find((int)$codigo);
    $elem = [];
    $elem['codProd'] = $producto->getCodProd();
    $elem['nombre'] = $producto->getNombre();
    $elem['peso'] = $producto->getPeso();
    $elem['stock'] = $producto->getStock();
    $elem['descripcion'] = $producto->getDescripcion();
    $elem['unidades'] = implode($cantidad);
    $productos[] = $elem;
}
return $this->render("carrito.html.twig",
array('productos'=>$productos));
}

```

9.5.5. Añadir y eliminar

Estos controladores son muy parecidos a los de la aplicación original. Solo cambia que acceden a la sesión con el objeto de Symfony. Modifican la variable de sesión y redirigen a carrito.

```

/**
 * @Route("/anadir", name="anadir")
 */
public function anadir(SessionInterface $session) {
    $id = $_POST['cod'];
    $unidades= $_POST['unidades'];
    $carrito = $session->get('carrito');
    if(is_null($carrito)){
        $carrito = array();
    }
    if(isset($carrito[$id])){
        $carrito[$id]['unidades'] += intval($unidades);
    }else{
        $carrito[$id]['unidades'] = intval($unidades);
    }
    $session->set('carrito', $carrito);
    return $this->redirectToRoute('carrito');
}
/**
```

```

* @Route("/eliminar", name="eliminar")
*/
public function eliminar(SessionInterface $session){
    $id = $_POST['cod'];
    $unidades= $_POST['unidades'];
    $carrito = $session->get('carrito');
    if(is_null($carrito)){
        $carrito = array();
    }
    if(isset($carrito[$id])){
        $carrito[$id]['unidades'] -= intval($unidades);
        if($carrito[$id]['unidades'] <= 0) {
            unset($carrito[$id]);
        }
    }
    $session->set('carrito', $carrito);
    return $this->redirectToRoute('carrito');
}

```

Actividad propuesta 9.3



¿Qué cambios habría que hacer para que el controlador de añadir verifique que no se incluyan más unidades de las disponibles para ningún producto?

9.5.6. Realizar pedido

Este es el controlador más complicado. Modifica la base de datos y envía los correos. Para enviar los correos utiliza la plantilla `correo.twig.html`; para mostrar si ha habido error o confirmar el pedido, `pedido.html.twig`.

```

/**
 * @Route("/realizarPedido", name="realizarPedido")
*/
public function realizarPedido(SessionInterface $session, \Swift_Mailer
$mailer) {
    $entityManager = $this->getDoctrine()->getManager();
    $carrito = $session->get('carrito');
    /* si el carrito no existe, o está vacío*/
    if(is_null($carrito) || count($carrito)==0){
        return $this->render("pedido.html.twig",
        array('error'=>1));
    }else{
        #crear un nuevo pedido
}

```

```
$pedido = new Pedido();
$pedido->setFecha(new \DateTime());
$pedido->setEnviado(0);
$pedido->setRestaurante($this->getUser());
$entityManager->persist($pedido);
#recorrer carrito creando nuevos pedidoproducto
foreach ($carrito as $codigo => $cantidad){
    $producto = $this->getDoctrine()
        ->getRepository(Producto::class)
        ->find($codigo);
    $fila = new PedidoProducto();
    $fila->setCodProd($producto);
    $fila->setUnidades( implode($cantidad));
    $fila->setCodPed($pedido);
    //actualizar el stock
    $cantidad = implode($cantidad);
    $query = $entityManager->createQuery(
        "UPDATE App\Entity\Producto p
         SET p.stock = p.stock - $cantidad
        WHERE p.codProd = $codigo");
    $resul = $query->getResult();
    $entityManager->persist($fila);
}
/*si hay error con la BD,
Muestra plantilla con el código adecuado*/
try{
    $entityManager->flush();
} catch (Exception $e) {
    return $this->render("pedido.html.twig",
        array('error'=>2));
}
/*prepara el array de productos para la plantilla*/
foreach ($carrito as $codigo => $cantidad){
    $producto = $this->getDoctrine()
        ->getRepository(Producto::class)
        ->find((int)$codigo);
    $elem = [];
    $elem['codProd'] = $producto->getCodProd();
    $elem['nombre'] = $producto->getNombre();
    $elem['peso'] = $producto->getPeso();
    $elem['stock'] = $producto->getStock();
    $elem['descripcion'] = $producto->getDescripcion();
    $elem['unidades'] = implode($cantidad);
    $productos[] = $elem;
}
//vaciar el carrito
$session->set('carrito', array());
```

```

/* mandar el correo */
$message = (new \Swift_Message())
    ->setFrom('noreply@empresafalsa.com', 'Sistema de pedidos')
    ->setTo($this->getUser()->getCorreo())
    ->setSubject("Pedido ". $pedido->getCodPed(). "confirmado")
    ->setBody($this->renderView('correo.html.twig',
        array('id'=>$pedido->getCodPed(),
            'productos'=> $productos),
        'text/html'));
$mailer->send($message);
return $this->render("pedido.html.twig",
    array('error'=>0, 'id'=>$pedido->getCodPed(),
        'productos'=> $productos));
}

```

Actividad propuesta 9.4



¿Qué cambios habría que hacer para que el controlador de realizar pedido verifique que no se incluyan más unidades de las disponibles para ningún producto? ¿Crees que es necesario hacerlo si ya se controló al añadir los productos al carrito?

9.6. Seguridad

Hay que modificar el fichero `/config/packages/security.yaml` para asociar el sistema de usuarios de Symfony con la entidad Restaurante y especificar las rutas de `login` y `logout`.

```

security:
    providers:
        pedidos:
            entity:
                class: App\Entity\Restaurante
                property: correo
    encoders:
        App\Entity\Restaurante:
            algorithm: plaintext
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true
            form_login:
                login_path: login
                check_path: login
                default_target_path: categorias

```

```

    provider: pedidos
logout:
    path: /logout
    target: /login

```

También hay que crear la ruta para “logout” en el fichero `/config/routes.yaml`:

```

logout:
    path: /logout

```

En lugar de utilizar la sección `access_control`, la clase `PedidosBase` está protegida con la anotación:

```

/**
 * @Security("has_role('ROLE_USER')")
 */
class PedidosBase extends AbstractController{

```

De esta manera, todos los controladores de la clase están restringidos a los usuarios que hayan abierto sesión.



Actividad propuesta 9.5

¿Cómo se utilizaría la sección `access_control` para controlar el acceso?

Resumen

- El diseño de la base de datos y la estructura para el carrito de la compra se mantienen igual que en las versiones anteriores.
- Las funciones de acceso a la base de datos y envío de correo cambian para adaptarse al *framework*.
- La base de datos se maneja a través de Doctrine. Hay una entidad por cada tabla.
- Para la salida, los controladores utilizan plantillas Twig.
- La estructura de la página se implementa a través de una jerarquía de plantillas.
- La aplicación utiliza el sistema de usuarios de Symfony. La entidad Restaurante tiene que implementar UserInterface.
- La asociación entre Productos y Categorías es bidireccional, así no hace falta hacer una consulta para encontrar los productos de una categoría.
- El resto de las asociaciones son unidireccionales.
- Los ficheros de la aplicación original se sustituyen por controladores.
- El control de acceso se realiza con una anotación en la clase `PedidosBase`, que incluye todos los controladores menos los de `login` y `logout`.



Ejercicios propuestos

- Al añadir un producto, la aplicación redirige al carrito. Modifícalo para que redirija a la tabla de productos, con la misma categoría.
- Modifica la tabla de productos para que no muestre los productos sin stock.
- Añade formularios para que los administradores (usuarios con rol 1) puedan insertar nuevas categorías y productos. Protege el acceso a ellos.
- Modifica la cabecera para que muestre a los administradores vínculos a los formularios del ejercicio 3.
- Modifica las tablas de productos y carrito para utilizar el componente de formularios de Symfony.
- En este capítulo se utilizan plantillas HTML, pero también podría haberse utilizado JSON, como en el capítulo 6. ¿Qué modificaciones habría que hacer para utilizar AJAX? Rediseña la aplicación intentando reutilizar el código del cliente del capítulo 6.

ACTIVIDADES DE AUTOEVALUACIÓN

- La asociación entre Producto y Categoría es:
 a) No hay.
 b) Unidireccional.
 c) Bidireccional.
- Sobre la plantilla cabecera.html.twig se puede decir que:
 a) Hereda de base.html.twig.
 b) base.html.twig hereda de ella.
 c) Ninguna de las anteriores.
- La entidad que representa a los usuarios de la aplicación es:
 a) Usuario.
 b) Restaurante.
 c) Pedido.
- El control de acceso se realiza con:
 a) La sección access_control del fichero security.yaml.
 b) Una anotación en la clase PedidosBase.
 c) Una anotación en la clase PedidosLogin.
- La configuración para la base de datos se ha introducido en:
 a) El fichero security.yaml.
 b) El fichero .env.
 c) No hay base de datos.

6. La ruta *eliminar* redirige a:

- a) carrito.
- b) categorías.
- c) cerrar sesión.

7. Para el envío de correo:

- a) Se reutilizan las funciones de la aplicación original.
- b) Se utiliza el componente de Symfony.
- c) No está configurado.

8. Para manejar la base de datos:

- a) Se utiliza Doctrine.
- b) Se reutilizan las funciones de la aplicación original.
- c) Se reutilizan las funciones de la aplicación original con pequeñas modificaciones.

9. Los formularios de añadir y eliminar productos:

- a) Se envían con JavaScript.
- b) Usan el componente de Symfony.
- c) Son formularios HTML normales.

10. En la aplicación se utilizan los roles:

- a) ROLE_USER y ROLE_ADMIN.
- b) ROLE_USER.
- c) No se utilizan roles.

SOLUCIONES:

1. **a** **b** **c**

2. **a** **b** **c**

3. **a** **b** **c**

4. **a** **b** **c**

5. **a** **b** **c**

6. **a** **b** **c**

7. **a** **b** **c**

8. **a** **b** **c**

9. **a** **b** **c**

10. **a** **b** **c**