



---

# SYMPHONY

---

PROGRAMACIÓN WEB



Symfony

15 DE FEBRERO DE 2021

UCA

Ignacio Javier Pérez Gálvez y Aitor Manuel Orellana Romero

## Contenido

1. Symphony.....	3
2. Instalación y Configuración.....	4
2.1 Requisitos del Sistema.....	4
2.2 Instalación de Composer.....	4
2.3 Método de instalación: .....	4
2.4 Actualizando aplicaciones Symfony .....	6
3. Modelo – Vista – Controlador en Symfony .....	6
3.1 Separación en capas más allá del MVC .....	7
3.2 Orientación a objetos.....	7
3.3 La implementación del MVC que realiza Symfony.....	7
4. Primera Página en Symfony. ....	8
4.1 Creando una página: Route y Controller.....	8
4.2 La barra de depuración web.....	10
4.3 Anotaciones.....	10
4.4 Renderizar una plantilla. ....	10
5. Estructura del Proyecto.....	12
6. Rutas en Symfony.....	13
6.1 Creación de RUTAS EN ARCHIVOS YAML,XML,PHP.....	14
6.2 Coincidencia con métodos HTTP.....	15
6.3 Depuración rutas.....	15
6.4 Parámetros en la ruta.....	16
6.5 Parámetros de validación.....	17
6.6 Parámetro por defecto.....	19
6.7 Parámetro de Prioridad.....	20
6.8 Parámetro especial.- .....	21
6.9 Parámetros Extras.- .....	22
6.10 Redirigir ruta con Trailing Slashes .....	22
6.11 Generando URLs.....	22
6.11 Generando URL en controladores.....	23
6.12 URL con Plantillas .....	23
6.13 Errores comunes de las rutas.....	23
7. CONTROLADOR. ....	24
7.1 CONTROLADOR BÁSICO. ....	24
7.2 Gestión de errores y páginas 404.....	26
7.3 Objeto de la solicitud como argumento del controlador.....	27

7.4	Manejo de la Sesión .....	27
7.5	Objeto de solicitud y respuesta.....	28
8.	VISTAS.....	29
8.1	Configuración de Twig.....	30
8.2	Creando Plantillas.....	30
8.3	Nombres de plantillas .....	31
8.4	Ubicación de la plantilla .....	32
8.5	Variables de plantilla.....	32
8.6	Vinculación a páginas .....	32
8.7	Vinculación a CSS, JavaScript y activos de imagen.....	33
8.8	Renderizando una plantilla en controladores.....	34
8.9	Renderizando una plantilla en servicios.....	34
8.10	Renderizando una plantilla directamente desde una ruta .....	35
8.11	Comprobando si existe una plantilla.....	35
8.12	Inspeccionando la información de Twig.....	36
8.13	Incluyendo Plantillas .....	36
8.14	Incorporación de controladores.....	37
9.	Crear una entidad.....	38
9.1	Migraciones: Creación de tablas / esquema de base de datos.....	40
9.2	Persistencia en la base de datos .....	42
9.3	Obteniendo objetos de la base de datos. ....	44
9.4	Consulta Compleja .....	46
10.	Consejos para hacer una aplicación en Symfony .....	46

## 1. Symphony

Symfony se diseñó para que se ajustara a los siguientes requisitos:

- Fácil de instalar y configurar en la mayoría de plataformas.
- Independiente del sistema gestor de bases de datos.
- Sencillo de usar en la mayoría de casos, pero lo suficientemente flexible como para adaptarse a los casos más complejos.
- Basado en la premisa de "convenir en vez de configurar", en la que el desarrollador solo debe configurar aquello que no es convencional.
- Sigue la mayoría de mejores prácticas y patrones de diseño para la web.
- Preparado para aplicaciones empresariales y adaptable a las políticas y arquitecturas propias de cada empresa, además de ser lo suficientemente estable como para desarrollar aplicaciones a largo plazo.
- Código fácil de leer que incluye comentarios de phpDocumentor y que permite un mantenimiento muy sencillo.
- Fácil de extender, lo que permite su integración con librerías desarrolladas por terceros.

## 2. Instalación y Configuración.

### 2.1 Requisitos del Sistema

Para la correcta funcionalidad de Symfony se necesita:

- PHP 7.2.5 o superior con las extensiones ctype, iconv, JSON, Session, SimpleXML y Tokenizer
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

Si instalamos XAMP (PHP 8.0.0) o AppServer (PHP 7.3.10) no necesitamos preocuparnos porque sus últimas versiones nos vienen incorporado todo esto.

<https://www.apachefriends.org/es/index.html> ó <https://www.appserv.org/en/>

- Instalar Visual Studio Code

<https://code.visualstudio.com/>

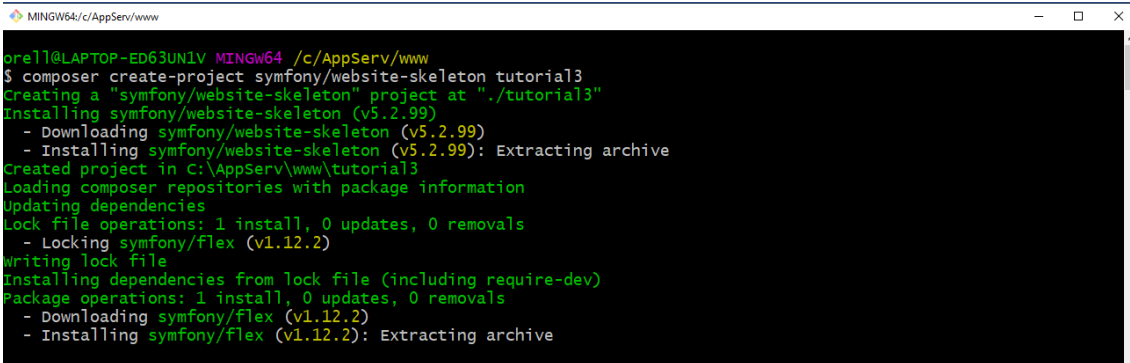
### 2.2 Instalación de Composer.

Actualmente tenemos una versión 2.0.8 de Composer. Es necesaria su instalación puesto que Laravel gestiona sus dependencias de este modo.

<https://getcomposer.org/download/>

### 2.3 Método de instalación:

En la terminal escribiremos el siguiente comando:



```

orell@LAPTOP-ED63UN1V MINGW64 /c/AppServ/www
$ composer create-project symfony/website-skeleton tutorial3
Creating a "symfony/website-skeleton" project at "./tutorial3"
Installing symfony/website-skeleton (v5.2.99)
- Downloading symfony/website-skeleton (v5.2.99)
- Installing symfony/website-skeleton (v5.2.99): Extracting archive
Created project in C:\AppServ\www\tutorial3
Loading composer repositories with package information
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
- Locking symfony/flex (v1.12.2)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
- Downloading symfony/flex (v1.12.2)
- Installing symfony/flex (v1.12.2): Extracting archive

```

Después del proceso de instalación. Symfony nos proveerá de recomendaciones para nuestro proyecto, alguno de ellos puede ser los siguientes:

## SYMPHONY

```
MINGW64/c:/AppServ/www
What's next?

* You're ready to send emails.

* If you want to send emails via a supported email provider, install
  the corresponding bridge.
  For instance, composer require mailgun-mailer for Mailgun.

* If you want to send emails asynchronously:

  1. Install the messenger component by running composer require messenger;
  2. Add 'Symfony\Component\Mailer\Messenger\SendEmailMessage': amqp to the
     config/packages/messenger.yaml file under framework.messenger.routing
     and replace amqp with your transport name of choice.

* Read the documentation at https://symfony.com/doc/master/mailer.html

Database Configuration

* Modify your DATABASE_URL config in .env

* Configure the driver (pgsql) and
  server_version (13) in config/packages/doctrine.yaml
```

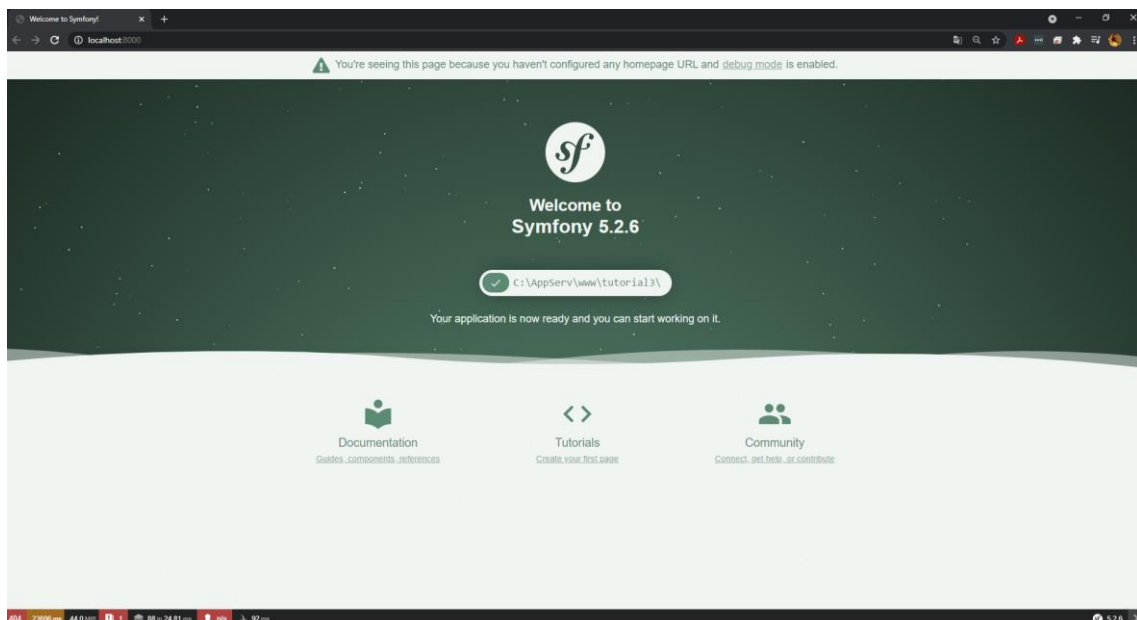
También se puede crear el proyecto con otras versiones de Symfony, pasando un segundo argumento indicando la versión:

```
$ composer create-project symfony/framework-standard-edition my_project_name "2.8.*"
```

Abrimos nuestro proyecto con Visual Studio Code y abrimos una nueva terminal y ejecutamos el comando de arranque :

```
php -S localhost:8000 -t public/
```

Si procedemos a abrir el navegador con la ruta <http://localhost:8000/> nos saldrá lo siguiente:



Si se ve una página en blanco o una página de error en vez de la de bienvenida, esto es debido a la configuración de permisos del directorio. Busca en la documentación de Symfony “Setting up or Fixing File Permissions” para más información y detalles.

Cuando hayamos finalizado de trabajar con la aplicación de Symfony, pararemos el servidor web pulsando Ctrl+C desde el terminal o consola de comandos donde lo hayamos ejecutado.

**Este paso deberá realizarlo siempre** que vaya a realizar el desarrollo de una web utilizando Symfony.

Cuando trabajas con una aplicación symfony la primera vez, puede ser útil el siguiente comando dentro del directorio del proyecto que muestra información sobre el proyecto:

```
$ php bin/console about
```

## 2.4 Actualizando aplicaciones Symfony

En este punto, tiene que haber creado una aplicación Symfony completamente funcional. Cada aplicación depende de numerosas librerías de terceros guardadas en el directorio vendor/ y gestionadas por Composer.

Actualizar algunas de esas librerías de forma frecuente es una buena práctica para prevenir fallos y vulnerabilidades de seguridad.

Ejecuta el comando update de Composer para actualizarlas todas a la vez (esto puede durar varios minutos, dependiendo de la complejidad del proyecto):

```
$ cd tutorial3/
$ composer update
```

Symfony incluye un comando para comprobar si alguna de las dependencias del proyecto contiene alguna vulnerabilidad de seguridad:

```
$ symfony check:security
```

## 3. Modelo – Vista – Controlador en Symfony

Symfony está basado en un patrón clásico del diseño web conocido como arquitectura MVC, que está formado por tres niveles:

- El Modelo representa la información con la que trabaja la aplicación, es decir, su lógica de negocio.

- Se encarga de la abstracción de la lógica relacionada con los datos, haciendo que la vista y las acciones sean independientes de, por ejemplo, el tipo de gestor de bases de datos utilizado por la aplicación.

- La Vista transforma el modelo en una página web que permite al usuario interactuar con ella.

- El Controlador se encarga de procesar las interacciones del usuario y realiza los cambios apropiados en el modelo o en la vista.

- También se encarga de aislar al modelo y a la vista de los detalles del protocolo utilizado para las peticiones (HTTP, consola de comandos, email, etc.).

La arquitectura MVC separa la lógica de negocio (el modelo) y la presentación (la vista) por lo que se consigue un mantenimiento más sencillo de las aplicaciones. Si por ejemplo una misma aplicación debe ejecutarse tanto en un navegador estándar como en un navegador de un dispositivo móvil, solamente es necesario crear una vista nueva para cada dispositivo, manteniendo el controlador y el modelo original.

### 3.1 Separación en capas más allá del MVC

El principio más importante de la arquitectura MVC es la separación del código del programa en tres capas, dependiendo de su naturaleza. La lógica relacionada con los datos se incluye en el modelo, el código de la presentación en la vista y la lógica de la aplicación en el controlador.

La programación se puede simplificar si se utilizan otros patrones de diseño. De esta forma, las capas del modelo, la vista y el controlador se pueden subdividir en más capas.

### 3.2 Orientación a objetos

La orientación a objetos permite a los desarrolladores trabajar con objetos de la vista, objetos del controlador y clases del modelo, transformando las funciones de los ejemplos anteriores en métodos. Se trata de un requisito obligatorio para las arquitecturas de tipo MVC.

### 3.3 La implementación del MVC que realiza Symfony

Piensa por un momento cuántos componentes se necesitan para realizar una página sencilla que muestre un listado de las entradas o artículos de un blog, son necesarios los siguientes componentes:

- La capa del Modelo
  - Abstracción de la base de datos
  - Acceso a los datos
- La capa de la Vista
  - Vista
  - Plantilla
  - Layout
- La capa del Controlador
  - Controlador frontal
  - Acción

En total son siete scripts, lo que parecen muchos archivos para abrir y modificar cada vez que se crea una página. Afortunadamente, Symfony simplifica este proceso.

Symfony toma lo mejor de la arquitectura MVC y la implementa de forma que el desarrollo de aplicaciones sea rápido y sencillo.

En primer lugar, el controlador frontal y el layout son comunes para todas las acciones de la aplicación. Se pueden tener varios controladores y varios layouts, pero solamente es obligatorio tener uno de cada. El controlador frontal es un componente que sólo tiene código relativo al MVC, por lo que no es necesario crear uno, ya que Symfony lo genera de forma automática.

La otra buena noticia es que las clases de la capa del modelo también se generan automáticamente, en función de la estructura de datos de la aplicación.

El ORM se encarga de crear el esqueleto o estructura básica de las clases y genera automáticamente todo el código necesario. Cuando el ORM encuentra restricciones de claves



foráneas (o externas) o cuando encuentra datos de tipo fecha, crea métodos especiales para acceder y modificar esos datos, por lo que la manipulación de datos se convierte en un juego de niños.

La abstracción de la base de datos es completamente transparente para el programador, ya que se realiza de forma nativa mediante PDO PHP Data Objects). Así, si se cambia el sistema gestor de bases de datos en cualquier momento, no se debe reescribir ni una línea de código, ya que tan sólo es necesario modificar un parámetro en un archivo de configuración

Por último, la lógica de la vista se puede transformar en un archivo de configuración sencillo, sin necesidad de programarla.

Por si fuera poco, crear la aplicación con Symfony permite crear páginas XHTML válidas, depurar fácilmente las aplicaciones, crear una configuración sencilla, abstracción de la base de datos utilizada, enrutamiento con URL limpias, varios entornos de desarrollo y muchas otras utilidades para el desarrollo de aplicaciones.

#### 4. Primera Página en Symfony.

Creando una nueva página -que pueden ser páginas HTML o JSON endpoint- tenemos que realizar dos pasos:

1. Crear una route: una route es la URL(Ej. /about) de tu página y puntos del controller.
2. Crear un controller: un controller es un script PHP de funciones que se encargará de construir las páginas. Toma la información de respuesta creando un objeto de Symfony de tipo Response, cada uno puede contener contenido HTML, cadenas JSON o incluso archivos binarios, tanto imágenes como PDF.

##### 4.1 Creando una página: Route y Controller

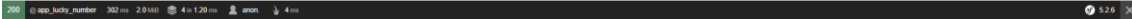
Supongamos que queremos crear la página - /lucky/number - que genera números de la suerte (de forma aleatoria) y los imprime. Para hacer eso, creamos una "Controller class" que la llamaremos LuckyController y un método "controller" que será ejecutado cada vez que vayamos a /lucky/number:

```
LuckyController.php ×
src > Controller > LuckyController.php
1  <?php
2  // src/Controller/LuckyController.php
3  namespace App\Controller;
4
5  use Symfony\Component\HttpFoundation\Response;
6
7  class LuckyController
8  {
9      public function number(): Response
10     {
11         $number = random_int(0, 100);
12
13         return new Response(
14             '<html><body>Lucky number: '.$number.'</body></html>'
15         );
16     }
17 }
18
```

Proseguimos creando la ruta correspondiente:

```
LuckyController.php  ! routes.yaml ×
config > ! routes.yaml
1  #index:
2  #   path: /
3  #   controller: App\Controller\DefaultController::index
4
5  # config/routes.yaml
6
7  # the "app_lucky_number" route name is not important yet
8  app_lucky_number:
9      path: /lucky/number
10     controller: App\Controller\LuckyController::number
11
12
```

Por lo cual si ingresamos la ruta <http://localhost:8000/lucky/number> nos encontraremos con lo siguiente:



## 4.2 La barra de depuración web

Si tu página está funcionando, entonces puedes ver una gran barra a los pies del explorador. Esta se llama *Web Debug Toolbar*. Aprenderás más sobre todo lo que te puede informar a lo largo de tu desarrollo, pero siéntete libre de experimentar: pasa por encima, haz clic en diferentes iconos y obtén información sobre las rutas, estado del sistema, logeo y más.



### 4.3 Anotaciones.

Para usar las rutas con el método de anotación, hace falta empezar con el siguiente comando:

composer require annotations

```
MINGW64: c:/AppServ/www/tutorial3
orell@LAPTOP-ED63UN1V MINGW64 /c/AppServ/www/tutorial3
$ composer require annotations
Using version ^6.1 for sensio/framework-extra-bundle
./composer.json has been updated
Running composer update sensio/framework-extra-bundle
Loading composer repositories with package information
Updating dependencies
Lock file operations: 0 installs, 1 update, 0 removals
  - Upgrading sensio/framework-extra-bundle (v5.6.1 => v6.1.1)
Writing lock file
Installing dependencies from lock file (including require-dev)
Nothing to install, update or remove
Generating optimized autoload files
composer/package-versions-deprecated: Generating version class...
composer/package-versions-deprecated: ...done generating version class
```

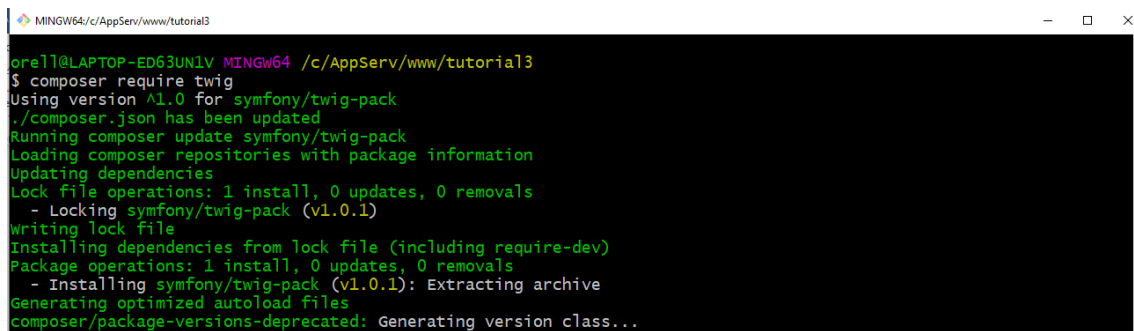
#### 4.4 Renderizar una plantilla.

Si estás volviendo a la página HTML desde el controller, es muy probable que que quieras recargar la plantilla. Afortunadamente, Symfony viene con Twig: un lenguaje de plantillas sencillo, potente y actual.

Primero, asegurate de que “LuckyController” extiende de la clase base de Symfony “AbstractController”.

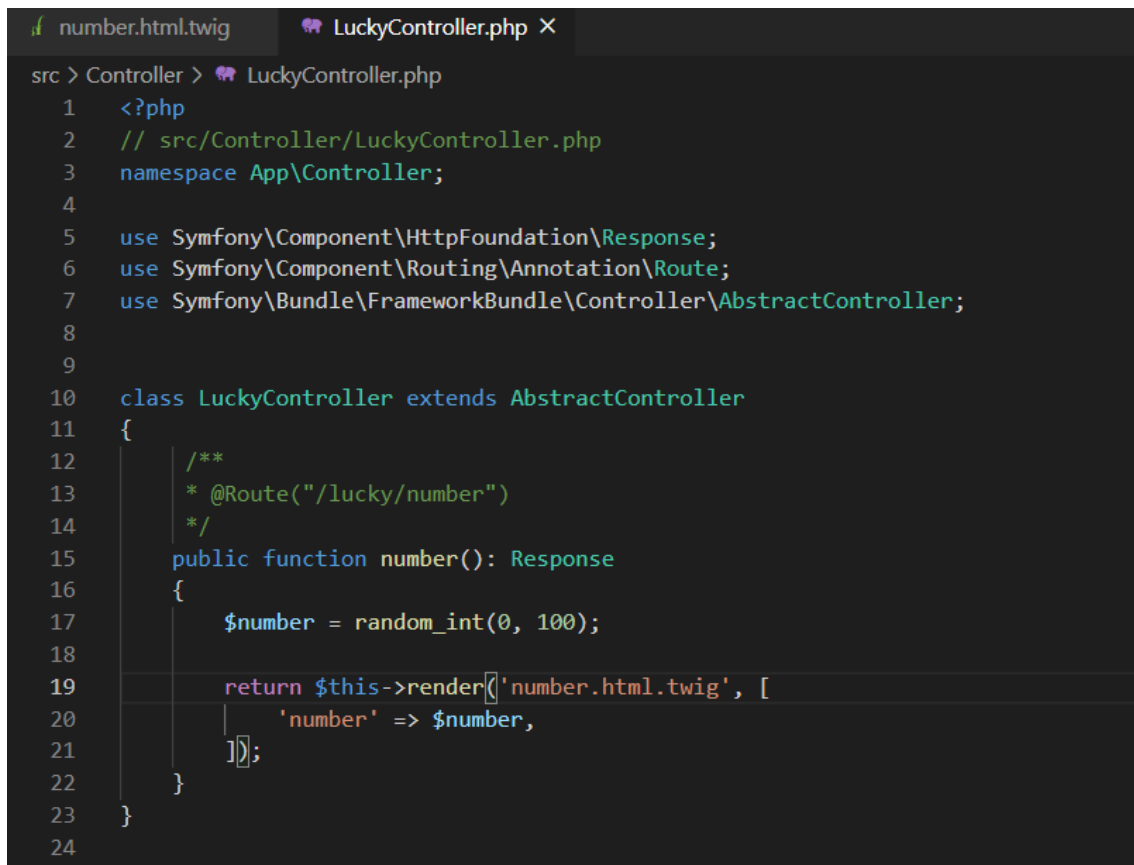
Instalamos el twig package con composer con el siguiente comando:

```
composer require twig
```



```
orell@LAPTOP-ED63UN1V MINGW64 /c/AppServ/www/tutorial3
$ composer require twig
Using version ^1.0 for symfony/twig-pack
./composer.json has been updated
Running composer update symfony/twig-pack
Loading composer repositories with package information
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
- Locking symfony/twig-pack (v1.0.1)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
- Installing symfony/twig-pack (v1.0.1): Extracting archive
Generating optimized autoload files
composer/package-versions-deprecated: Generating version class...
```

Continuando, modificamos el controlador de la siguiente manera para que pueda hacer uso de la plantilla:

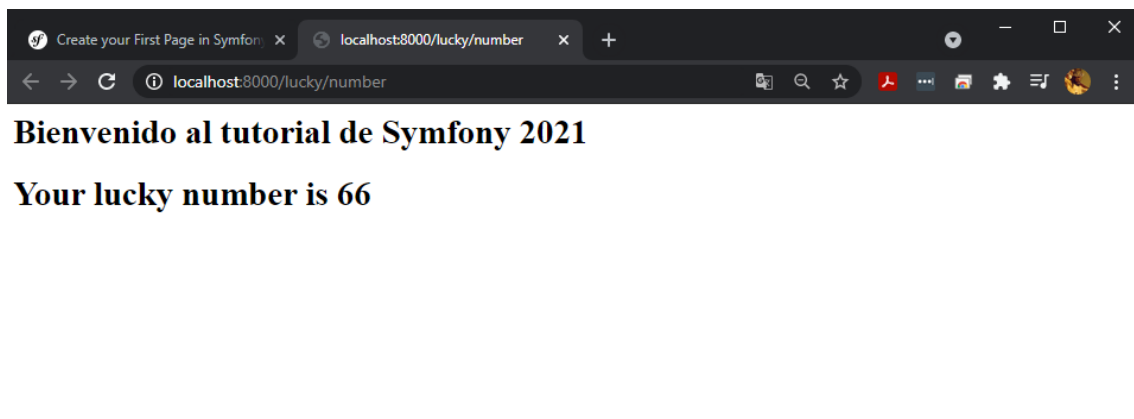


```
number.html.twig  LuckyController.php X
src > Controller > LuckyController.php
1  <?php
2  // src/Controller/LuckyController.php
3  namespace App\Controller;
4
5  use Symfony\Component\HttpFoundation\Response;
6  use Symfony\Component\Routing\Annotation\Route;
7  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
8
9
10 class LuckyController extends AbstractController
11 {
12     /**
13      * @Route("/lucky/number")
14      */
15     public function number(): Response
16     {
17         $number = random_int(0, 100);
18
19         return $this->render('number.html.twig', [
20             'number' => $number,
21         ]);
22     }
23 }
24
```

Los archivos de plantillas se encuentran en el directorio templates/. Crea una nueva vista en el directorio templates/lucky con un archivo con el siguiente nombre number.html.twig y el código que se muestra a continuación:

```
number.html.twig X
templates > number.html.twig
1  {# templates/lucky/number.html.twig #}
2  <h1> Bienvenido al tutorial de Symfony 2021 </h1>
3  <h1>Your lucky number is {{ number }}</h1>
4
```

Una vez obtenida la plantilla, nos basta con Volver a la página y observar lo que nos aparece:



## 5. Estructura del Proyecto

Ya hemos estado trabajando en dos de las carpetas más importantes de nuestro proyecto, ahora veremos la importancia y uso de las demás dentro de la estructura que nos ofrece Symfony:

- config/
  - Contains... configuration!. You will configure routes, services and packages.
- src/
  - All your PHP code lives here.
- templates/
  - All your Twig templates live here.

Most of the time, you'll be working in src/, templates/ or config/. As you keep reading, you'll learn what can be done inside each of these. So what about the other directories in the project?

- bin/
  - The famous bin/console file lives here (and other, less important executable files).
- var/
  - This is where automatically-created files are stored, like cache files (var/cache/) and logs (var/log/).
- vendor/
  - Third-party (i.e. "vendor") libraries live here! These are downloaded via the Composer package manager.
- public/

- This is the document root for your project: you put any publicly accessible files here.

And when you install new packages, new directories will be created automatically when needed.

## 6. Rutas en Symfony

<https://symfony.com/doc/current/routing.html>

El uso de rutas elegantes es algo que va ligado a una aplicación web seria. Esto significa que no podemos tener URLs de este tipo `index.php?article_id=57` sino más bien del tipo `/read/intro-to-symfony`.

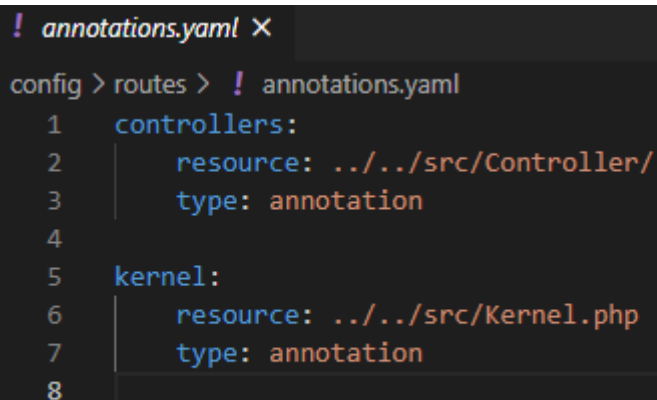
Las rutas se pueden configurar en YAML, XML, PHP o usando atributos o anotaciones. Todos los formatos ofrecen las mismas características y rendimiento, así que elija su favorito. Symfony recomienda atributos porque es conveniente colocar la ruta y el controlador en el mismo lugar.

En este aspecto también hemos de considerar la flexibilidad, puesto que si quisiéramos cambiar una ruta de `/blog` a `/news` ¿cuántos cambios tendríamos que hacer?. Con las posibilidades que nos ofrece Symfony muy pocos.

En PHP 8, puede usar atributos nativos para configurar rutas de inmediato. En PHP 7, donde los atributos no están disponibles, puede usar anotaciones en su lugar, proporcionadas por la biblioteca de anotaciones de Doctrine.

En caso de que desee utilizar anotaciones en lugar de atributos, ejecute este comando una vez en su aplicación para habilitarlas:

```
composer require doctrine/annotations
```

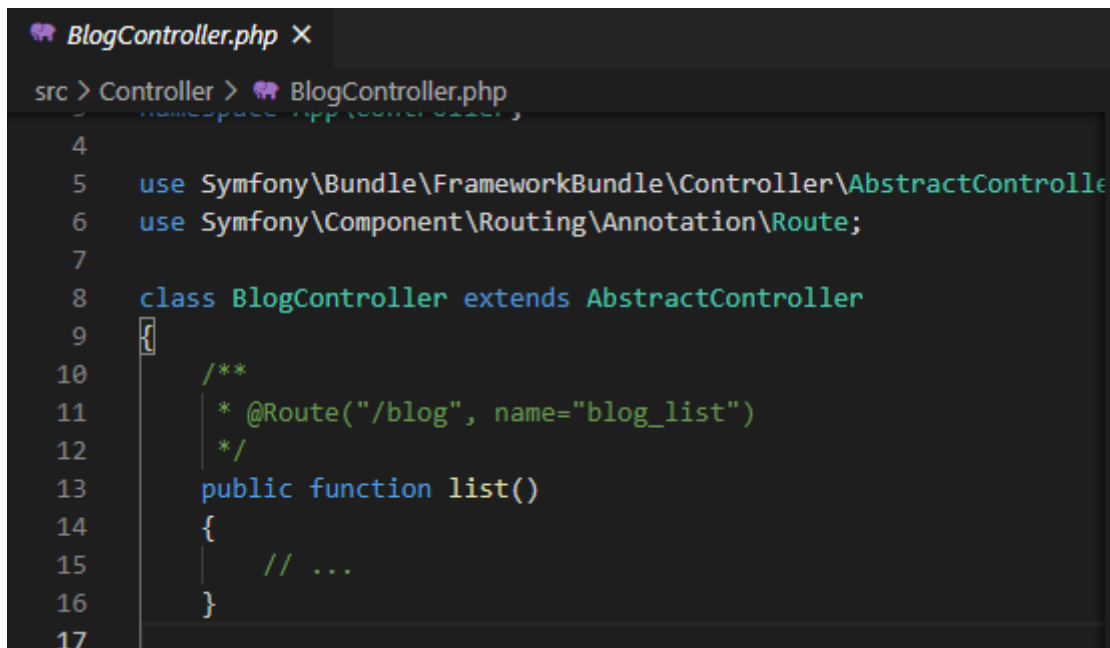


```
! annotations.yaml X
config > routes > ! annotations.yaml
1 controllers:
2   resource: ../../src/Controller/
3   type: annotation
4
5 kernel:
6   resource: ../../src/Kernel.php
7   type: annotation
8
```

Esta configuración le dice a Symfony que busque rutas definidas como anotaciones en cualquier clase PHP almacenada en el `src/Controller/` directorio.

Suponga que desea definir una ruta para la `/blogURL` en su aplicación. Para hacerlo, cree una

clase de controlador como la siguiente:



```

1  namespace App\Controller;
2
3
4
5  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6  use Symfony\Component\Routing\Annotation\Route;
7
8  class BlogController extends AbstractController
9  {
10
11      /**
12       * @Route("/blog", name="blog_list")
13       */
14      public function list()
15      {
16          // ...
17      }
18  }

```

Esta configuración define una ruta llamada `blog_list` que coincide cuando el usuario solicita la `/blogURL`. Cuando ocurre la coincidencia, la aplicación ejecuta el `list()` método de la `BlogController` clase.

*Nota: Si define varias clases de PHP en el mismo archivo, Symfony solo carga las rutas de la primera clase, ignorando todas las demás rutas.*

## 6.1 Creación de RUTAS EN ARCHIVOS YAML,XML,PHP.

En lugar de definir rutas en las clases de controlador, puede definir las en un archivo YAML, XML o PHP separado. La principal ventaja es que no requieren ninguna dependencia adicional. El principal inconveniente es que tiene que trabajar con varios archivos al verificar el enrutamiento de alguna acción del controlador.

El siguiente ejemplo muestra cómo definir en YAML / XML / PHP una ruta llamada `blog_list` que asocia la `/blogURL` con la `list()` acción de `BlogController`:

```

! routes.yaml X
config > ! routes.yaml
1  #index:
2  #   path: /
3  #   controller: App\Controller\DefaultController::index
4
5  # config/routes.yaml
6
7  # the "app_lucky_number" route name is not important yet
8  app_lucky_number:
9      path: /lucky/number
10     controller: App\Controller\LuckyController::number
11
12  blog_list:
13      ...path: /blog
14      ...# the controller value has the format 'controller_class::method_name'
15      ...controller: App\Controller\BlogController::list
16
17      # if the action is implemented as the __invoke() method of the
18      # controller class, you can skip the '::method_name' part:
19      # controller: App\Controller\BlogController
20
21

```

## 6.2 Coincidencia con métodos HTTP

Por defecto, se ajustan a cualquier rutas HTTP verbo ( GET, POST, PUT, etc.) Use la `methods` opción de restringir los verbos cada ruta debe responder a:SS

```

1  # config/routes.yaml
2  api_post_show:
3      path:      /api/posts/{id}
4      controller: App\Controller\BlogApiController::show
5      methods:   GET|HEAD
6
7  api_post_edit:
8      path:      /api/posts/{id}
9      controller: App\Controller\BlogApiController::edit
10     methods:   PUT

```

## 6.3 Depuración rutas.

A medida que su aplicación crezca, eventualmente tendrá muchas rutas. Symfony incluye algunos comandos para ayudarlo a depurar problemas de enrutamiento. Primero, el `debug:router` comando enumera todas las rutas de su aplicación en el mismo orden en que Symfony las evalúa:



```
PS C:\AppServ\www\tutorial3> php bin/console debug:router
```

Name	Method	Scheme	Host	Path
_preview_error	ANY	ANY	ANY	/_error/{code}.{_format}
_wdt	ANY	ANY	ANY	/_wdt/{token}
_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
blog_show	ANY	ANY	ANY	/blog/{slug}
app_lucky_number	ANY	ANY	ANY	/lucky/number
blog_list	ANY	ANY	ANY	/blog

#### 6.4 Parámetros en la ruta

Los ejemplos anteriores definieron rutas donde la URL nunca cambia (por ejemplo /blog). Sin embargo, es común definir rutas donde algunas partes son variables. Por ejemplo, la URL para mostrar alguna publicación de blog probablemente incluirá el título o slug (por ejemplo, /blog/my-first-posto /blog/all-about-symfony).

En las rutas de Symfony, las partes variables están envueltas y deben tener un nombre único. Por ejemplo, la ruta para mostrar el contenido de la publicación del blog se define como `{ ... }/blog/{slug}`

```

! routes.yaml X
config > ! routes.yaml
1  #index:
2  #   path: /
3  #   controller: App\Controller\DefaultController::index
4
5  # config/routes.yaml
6
7  # the "app_lucky_number" route name is not important yet
8  ~ app_lucky_number:
9      path: /lucky/number
10     controller: App\Controller\LuckyController::number
11
12  ~ blog_list:
13      path: /blog
14      # the controller value has the format 'controller_class::method_name'
15      controller: App\Controller\BlogController::list
16
17      # if the action is implemented as the __invoke() method of the
18      # controller class, you can skip the '::method_name' part:
19      # controller: App\Controller\BlogController
20
21  ~ blog_show:
22      ... path: /blog/{slug}
23      ... controller: App\Controller\BlogController::show
24
25
26

```

El nombre de la parte variable ( {slug} en este ejemplo) se usa para crear una variable PHP donde el contenido de la ruta se almacena y se pasa al controlador. Si un usuario visita la /blog/my-first-postURL, Symfony ejecuta el show() método en la BlogController clase y pasa un argumento al método. \$slug = 'my-first-post' show()

Las rutas pueden definir cualquier número de parámetros, pero cada uno de ellos solo se puede utilizar una vez en cada ruta (p /blog/posts-about-{category}/page/{pageNumber}. Ej .).

## 6.5 Parámetros de validación

Imagine que su aplicación tiene una blog\_show ruta (URL:) /blog/{slug} y una blog\_list ruta (URL:) /blog/{page}. Dado que los parámetros de ruta aceptan cualquier valor, no hay forma de diferenciar ambas rutas.

Si el usuario lo solicita /blog/my-first-post, ambas rutas coincidirán y Symfony usará la ruta que se definió primero. Para solucionar esto, agregue algo de validación al {page} parámetro usando la requirements opción:

```

1  // src/Controller/BlogController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\HttpFoundation\Response;
6  use Symfony\Component\Routing\Annotation\Route;
7
8  class BlogController extends AbstractController
9  {
10     /**
11      * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
12      */
13     public function list(int $page): Response
14     {
15         // ...
16     }
17
18     /**
19      * @Route("/blog/{slug}", name="blog_show")
20      */
21     public function show(string $slug): Response
22     {
23         // ...
24     }
25 }

```

URL	Ruta	Parámetros
/blog/2	blog_list	\$page = 2
/blog/my-first-post	blog_show	\$slug = my-first-post

Si lo prefiere, los requisitos se pueden incluir en cada parámetro utilizando la sintaxis {parameter\_name<requirements>}. Esta función hace que la configuración sea más concisa, pero puede reducir la legibilidad de la ruta cuando los requisitos son complejos

```

1  // src/Controller/BlogController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\HttpFoundation\Response;
6  use Symfony\Component\Routing\Annotation\Route;
7
8  class BlogController extends AbstractController
9  {
10     /**
11      * @Route("/blog/{page<\d+>}", name="blog_list")
12      */
13     public function list(int $page): Response
14     {
15         // ...
16     }
17 }

```

#### 6.6 Parámetro por defecto

En el ejemplo anterior, la URL de `blog_list` es `/blog/{page}`. Si los usuarios visitan `/blog/1`, coincidirá. Pero si lo visitan `/blog`, no coincidirá. Tan pronto como agregue un parámetro a una ruta, debe tener un valor.

Puede volver a hacer `blog_list` coincidir cuando el usuario visita `/blog` agregando un valor predeterminado para el `{page}` parámetro. Cuando se utilizan anotaciones, los valores predeterminados se definen en los argumentos de la acción del controlador. En el resto de formatos de configuración se definen con la `defaults` opción:

```

1  // src/Controller/BlogController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\HttpFoundation\Response;
6  use Symfony\Component\Routing\Annotation\Route;
7
8  class BlogController extends AbstractController
9  {
10     /**
11      * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
12      */
13     public function list(int $page = 1): Response
14     {
15         // ...
16     }
17 }

```

```

1  # config/routes.yaml
2  blog_list:
3      path:      /blog/{page}
4      controller: App\Controller\BlogController::list
5      defaults:
6          page: 1
7      requirements:
8          page: '\d+'
9

```

Ahora, cuando el usuario visite /blog, la blog\_listruta coincidirá y \$pagetendrá un valor predeterminado de 1.

### 6.7 Parámetro de Prioridad

Al definir un patrón codicioso que coincide con muchas rutas, esto puede estar al comienzo de su colección de enrutamiento y evita que cualquier ruta definida después coincida. Un priority está disponible para permitirle elegir el orden de sus rutas, y solo está disponible cuando se utilizan anotaciones.

```

1  // src/Controller/BlogController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\Routing\Annotation\Route;
6
7  class BlogController extends AbstractController
8  {
9      /**
10       * This route has a greedy pattern and is defined first.
11       *
12       * @Route("/blog/{slug}", name="blog_show")
13       */
14       public function show(string $slug)
15       {
16           // ...
17       }
18
19       /**
20       * This route could not be matched without defining a higher priority than 0.
21       *
22       * @Route("/blog/list", name="blog_list", priority=2)
23       */
24       public function list()
25       {
26           // ...
27       }
28  }

```

## 6.8 Parámetro especial.-

Además de sus propios parámetros, las rutas pueden incluir cualquiera de los siguientes parámetros especiales creados por Symfony:

- **\_controller**
  - Este parámetro se utiliza para determinar qué controlador y acción se ejecuta cuando la ruta coincide.
- **\_format**
  - El valor coincidente se utiliza para establecer el "formato de solicitud" del Request objeto. Esto se usa para cosas tales como establecer el Content-Type de la respuesta (por ejemplo, un json formato se traduce en un Content-Type de application/json).
- **\_fragment**
  - Se usa para establecer el identificador de fragmento, que es la última parte opcional de una URL que comienza con un #carácter y se usa para identificar una parte de un documento.
- **\_locale**
  - Se utiliza para establecer la configuración regional en la solicitud.

Puede incluir estos atributos (excepto `_fragment`) tanto en rutas individuales como en importaciones de rutas. Symfony define algunos atributos especiales con el mismo nombre (excepto el subrayado inicial) para que puedas definirlos más fácilmente:

```

1  // src/Controller/ArticleController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\HttpFoundation\Response;
6  use Symfony\Component\Routing\Annotation\Route;
7
8  class ArticleController extends AbstractController
9  {
10     /**
11      * @Route(
12      *     "/articles/{_locale}/search.{_format}",
13      *     locale="en",
14      *     format="html",
15      *     requirements={
16      *         "_locale": "en|fr",
17      *         "_format": "html|xml",
18      *     }
19      * )
20      */
21     public function search(): Response
22     {
23     }
24 }
```

### 6.9 Parámetros Extras.-

En la defaults opción de una ruta, opcionalmente puede definir parámetros no incluidos en la configuración de la ruta. Esto es útil para pasar argumentos adicionales a los controladores de las rutas:

```

1  // src/Controller/BlogController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\HttpFoundation\Response;
6  use Symfony\Component\Routing\Annotation\Route;
7
8  class BlogController extends AbstractController
9  {
10     /**
11      * @Route("/blog/{page}", name="blog_index", defaults={"page": 1, "title": "Hello world"
12      */
13     public function index(int $page, string $title): Response
14     {
15         // ...
16     }
17 }

```

### 6.10 Redirigir ruta con Trailing Slashes

Históricamente, las URL han seguido la convención de UNIX de agregar barras al final de los directorios (por ejemplo <https://example.com/foo/>) y eliminarlas para hacer referencia a los archivos ( <https://example.com/foo>). Aunque está bien ofrecer contenido diferente para ambas URL, hoy en día es común tratar ambas URL como la misma URL y redirigir entre ellas.

Symfony sigue esta lógica para redirigir entre URL con y sin barras diagonales finales (pero solo para solicitudes GET y HEAD):

URL de ruta	Si la URL solicitada es /foo	Si la URL solicitada es /foo/
/foo	Coincide ( 200 respuesta de estado)	Hace una 301 redirección a /foo
/foo/	Hace una 301 redirección a /foo/	Coincide ( 200 respuesta de estado)

### 6.11 Generando URLs

Los sistemas de enrutamiento son bidireccionales: 1) asocian URL con controladores (como se explicó en las secciones anteriores); 2) generan URL para una ruta determinada. La generación de URL a partir de rutas le permite no escribir los valores manualmente en sus plantillas HTML. Además, si la URL de alguna ruta cambia, solo tienes que actualizar la configuración de la ruta y se actualizarán todos los enlaces. <a href="...">

Para generar una URL, debe especificar el nombre de la ruta (p blog\_show. Ej. ) Y los valores de los parámetros definidos por la ruta (p . Ej .).slug = my-blog-post

Por eso cada ruta tiene un nombre interno que debe ser único en la aplicación. Si no establece el nombre de la ruta explícitamente con la name opción, Symfony genera un nombre automático basado en el controlador y la acción.

## 6.11 Generando URL en controladores.

Si su controlador se extiende desde `AbstractController`, use el `generateUrl()` asistente:

```
// src/Controller/BlogController.php
namespace App\Controller;

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

class BlogController extends AbstractController
{
    /**
     * @Route("/blog", name="blog_list")
     */
    public function list(): Response
    {
        // generate a URL with no route arguments
        $signUpPage = $this->generateUrl('sign_up');

        // generate a URL with route arguments
        $userProfilePage = $this->generateUrl('user_profile', [
            'username' => $user->getUsername(),
        ]);

        // generated URLs are "absolute paths" by default. Pass a third optional
        // argument to generate different URLs (e.g. an "absolute URL")
        $signUpPage = $this->generateUrl('sign_up', [], UrlGeneratorInterface::ABSOLUTE_URL);

        // when a route is localized, Symfony uses by default the current request locale
        // pass a different '_locale' value if you want to set the locale explicitly
        $signUpPageInDutch = $this->generateUrl('sign_up', ['_locale' => 'nl']);

        // ...
    }
}
```

## 6.12 URL con Plantillas

```
1 <a href="{{ path('blog_index') }}">Homepage</a>
2
3 {# ... #}
4
5 {% for post in blog_posts %}
6     <h1>
7         <a href="{{ path('blog_post', {slug: post.slug}) }}">{{ post.title }}</a>
8     </h1>
9
10    <p>{{ post.excerpt }}</p>
11 {% endfor %}
```

## 6.13 Errores comunes de las rutas.

- Controller `App\Controller\BlogController::show()` requires that you provide a value for the `“$slug”` argument.
  - This happens when your controller method has an argument (e.g. `$slug`)



```
public function show(string $slug): Response
{
    // ...
}
```

- 
- Some mandatory parameters are missing (“slug”) to generate a URL for route “blog\_show”.
  - This means that you’re trying to generate a URL to the blog\_show route but you are not passing a slug value (which is required, because it has a {slug} parameter in the route path). To fix this, pass a slug value when generating the route:

```
$this->generateUrl('blog_show', ['slug' => 'slug-value']);
```

- O en un Twig

```
1 {{ path('blog_show', {slug: 'slug-value'}) }}
```

## 7. CONTROLADOR.

<https://symfony.com/doc/current/controller.html>

En Symfony, la capa del controlador, que contiene el código que liga la lógica de negocio con la presentación, está dividida en varios componentes que se utilizan para diversos propósitos:

- El controlador frontal es el único punto de entrada a la aplicación. Carga la configuración y determina la acción a ejecutarse.
- Las acciones contienen la lógica de la aplicación. Verifican la integridad de las peticiones y preparan los datos requeridos por la capa de presentación.
- Los objetos request, response y session dan acceso a los parámetros de la petición, las cabeceras de las respuestas y a los datos persistentes del usuario. Se utilizan muy a menudo en la capa del controlador.
- Los filtros son trozos de código ejecutados para cada petición, antes o después de una acción. Por ejemplo, los filtros de seguridad y validación son comúnmente utilizados en aplicaciones web. Puedes extender el framework creando tus propios filtros.

Este capítulo describe todos estos componentes, pero no te abrumes porque sean muchos componentes. Para una página básica, es probable que solo necesites escribir algunas líneas de código en la clase de la acción, y eso es todo. Los otros componentes del controlador solamente se utilizan en situaciones específicas.

### 7.1 CONTROLADOR BÁSICO.

Si bien un controlador puede ser cualquier PHP invocable (función, método en un objeto o a Closure), un controlador suele ser un método dentro de una clase de controlador:

```
PS C:\AppServ\www\tutorial3> php bin/console make:controller PruebaController
```

```
created: src/Controller/PruebaController.php
created: templates/prueba/index.html.twig
```

Success!

Next: Open your new controller class and add some pages!

```
PS C:\AppServ\www\tutorial3>
```

Por lo cual si nos vamos a la ruta del controlador nos encontraremos un archivo como el siguiente

```
PruebaController.php X
src > Controller > PruebaController.php
1  <?php
2
3  namespace App\Controller;
4
5  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6  use Symfony\Component\HttpFoundation\Response;
7  use Symfony\Component\Routing\Annotation\Route;
8
9  class PruebaController extends AbstractController
10 {
11     /**
12      * @Route("/prueba", name="prueba")
13      */
14     public function index(): Response
15     {
16         return $this->render('prueba/index.html.twig', [
17             'controller_name' => 'PruebaController',
18         ]);
19     }
20 }
21
```

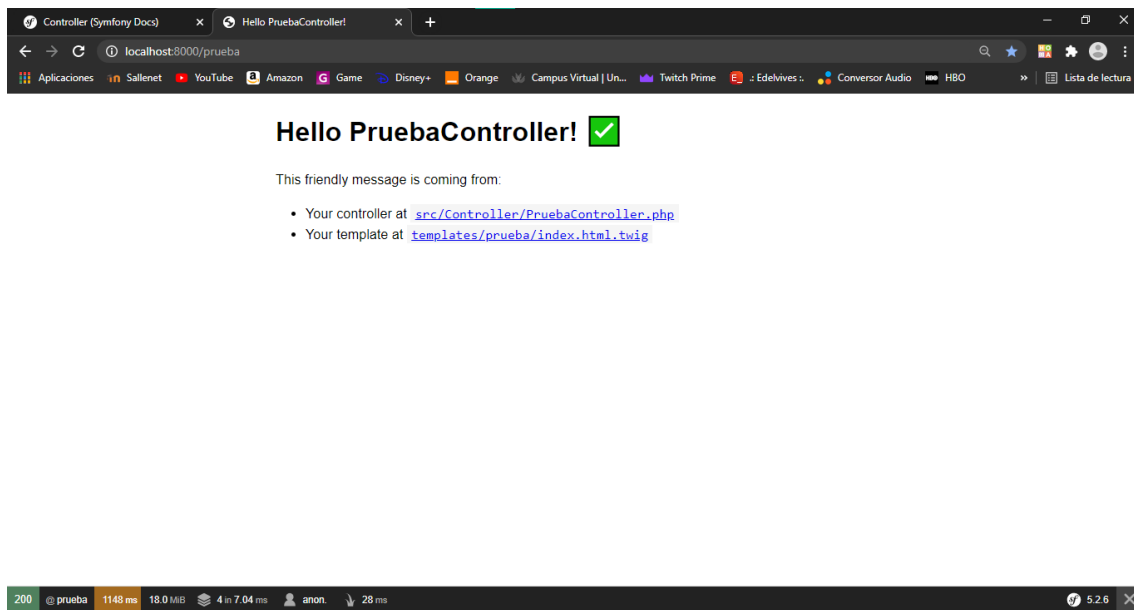
Y en su mismo caso con la ruta de la plantilla :

```

1 index.html.twig X
2 templates > prueba > index.html.twig
3 {% extends 'base.html.twig' %}
4
5 {% block title %}Hello PruebaController!{% endblock %}
6
7 {% block body %}
8     <style>
9         .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5 sans-serif; }
10        .example-wrapper code { background: #f5f5f5; padding: 2px 6px; }
11    </style>
12
13    <div class="example-wrapper">
14        <h1>Hello {{ controller_name }}! ✓</h1>
15
16        This friendly message is coming from:
17        <ul>
18            <li>Your controller at <code><a href="{{ 'C:/AppServ/www/tutorial3/src/Controller/PruebaController.php'|file_link(0) }}">src/Controller/PruebaController.php</a></code></li>
19            <li>Your template at <code><a href="{{ 'C:/AppServ/www/tutorial3/templates/prueba/index.html.twig'|file_link(0) }}">templates/prueba/index.html.twig</a></code></li>
20        </ul>
21    </div>
22 {% endblock %}

```

Por lo cual si nos vamos a la ruta <http://localhost:8000/prueba> nos mostraría lo siguiente:



Para la generación de un CRUD haría falta ejecutar el siguiente comando (previamente habiendo creado la entidad Doctrine):

```

> php bin/console make:crud Product

created: src/Controller/ProductController.php
created: src/Form/ProductType.php
created: templates/product/_delete_form.html.twig
created: templates/product/_form.html.twig
created: templates/product/edit.html.twig
created: templates/product/index.html.twig
created: templates/product/new.html.twig
created: templates/product/show.html.twig

```

## 7.2 Gestión de errores y páginas 404.

Cuando no se encuentran cosas, debe devolver una respuesta 404. Para hacer esto, lanza un tipo especial de excepción:

```

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

// ...
public function index(): Response
{
    // retrieve the object from database
    $product = ...;
    if (!$product) {
        throw $this->createNotFoundException('The product does not exist');

        // the above is just a shortcut for:
        // throw new NotFoundHttpException('The product does not exist');
    }

    return $this->render(...);
}

```

El `createNotFoundException()` método es solo un atajo para crear un `Symfony\Component\HttpFoundation\Exception\NotFoundHttpException` objeto especial, que finalmente desencadena una respuesta HTTP 404 dentro de Symfony.

Si lanza una excepción que se extiende o es una instancia de `Symfony\Component\HttpFoundation\Exception\HttpException`, Symfony usará el código de estado HTTP apropiado. De lo contrario, la respuesta tendrá un código de estado HTTP 500:

```

// this exception ultimately generates a 500 status error
throw new \Exception('Something went wrong!');

```

### 7.3 Objeto de la solicitud como argumento del controlador.

<https://symfony.com/doc/current/controller.html#request-object-info>

```

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
// ...

public function index(Request $request, string $firstName, string $lastName): Response
{
    $page = $request->query->get('page', 1);

    // ...
}

```

### 7.4 Manejo de la Sesión.

Symfony proporciona un servicio de sesión que puede utilizar para almacenar información sobre el usuario entre solicitudes. La sesión está habilitada de forma predeterminada, pero solo se iniciará si lee o escribe desde ella.

El almacenamiento de sesiones y otras configuraciones se pueden controlar bajo la configuración `framework.session` en `config/packages/framework.yaml`.

Para obtener la sesión, agregue un argumento y escriba una sugerencia con `Symfony\Component\HttpFoundation\Session\SessionInterface`:

```

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Session\SessionInterface;
// ...

public function index(SessionInterface $session): Response
{
    // stores an attribute for reuse during a Later user request
    $session->set('foo', 'bar');

    // gets the attribute set by another controller in another request
    $foobar = $session->get('foobar');

    // uses a default value if the attribute doesn't exist
    $filters = $session->get('filters', []);

    // ...
}

```

Los atributos almacenados permanecen en la sesión durante el resto de la sesión de ese usuario.

Para más información, mirar el siguiente enlace: <https://symfony.com/doc/current/session.html>

## 7.5 Objeto de solicitud y respuesta

Como se mencionó anteriormente, Symfony pasará el Request objeto a cualquier argumento de controlador que esté insinuado con la Request clase:

```

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

public function index(Request $request): Response
{
    $request->isXmlHttpRequest(); // is it an Ajax request?

    $request->getPreferredLanguage(['en', 'fr']);

    // retrieves GET and POST variables respectively
    $request->query->get('page');
    $request->request->get('page');

    // retrieves SERVER variables
    $request->server->get('HTTP_HOST');

    // retrieves an instance of UploadedFile identified by foo
    $request->files->get('foo');

    // retrieves a COOKIE value
    $request->cookies->get('PHPSESSID');

    // retrieves an HTTP request header, with normalized, Lowercase keys
    $request->headers->get('host');
    $request->headers->get('content-type');
}

```

La Request clase tiene varias propiedades y métodos públicos que devuelven cualquier información que necesite sobre la solicitud.

Al igual que el Request, el Response objeto tiene una headers propiedad pública . Este objeto es del tipo `Symfony\Component\HttpFoundation\ResponseHeaderBag` y proporciona métodos para obtener y configurar encabezados de respuesta. Los nombres de los encabezados están normalizados. Como resultado, el nombre Content-Type es equivalente al nombre content-type o content\_type.

En Symfony, se requiere un controlador para devolver un Response objeto:

```
use Symfony\Component\HttpFoundation\Response;

// creates a simple Response with a 200 status code (the default)
$response = new Response('Hello '.$name, Response::HTTP_OK);

// creates a CSS-response with a 200 status code
$response = new Response('<style> ... </style>');
$response->headers->set('Content-Type', 'text/css');
```

## 8. VISTAS

El lenguaje de plantillas Twig le permite escribir plantillas concisas y legibles que son más amigables para los diseñadores web y, en varios sentidos, más poderosas que las plantillas PHP. Eche un vistazo al siguiente ejemplo de plantilla de Twig. Incluso si es la primera vez que ve a Twig, probablemente entienda la mayor parte:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Welcome to Symfony!</title>
5   </head>
6   <body>
7     <h1>{{ page_title }}</h1>
8
9     {% if user.isLoggedIn %}
10      Hello {{ user.name }}!
11    {% endif %}
12
13    {# ... #}
14  </body>
15 </html>
```

La sintaxis de Twig se basa en estas tres construcciones:

- `{{ ... }}`, utilizado para mostrar el contenido de una variable o el resultado de evaluar una expresión;
- `{% ... %}`, utilizado para ejecutar alguna lógica, como un condicional o un bucle;
- `{# ... #}`, utilizado para agregar comentarios a la plantilla (a diferencia de los comentarios HTML, estos comentarios no se incluyen en la página renderizada).

No puede ejecutar código PHP dentro de las plantillas Twig, pero Twig proporciona utilidades para ejecutar algo de lógica en las plantillas. Por ejemplo, los filtros modifican el contenido antes de ser renderizado, como el upper filtro para contenido en mayúsculas:

```
1 {{ title|upper }}
```

Twig viene con una larga lista de etiquetas , filtros y funciones que están disponibles de forma predeterminada. En las aplicaciones Symfony también puede utilizar estos filtros y funciones Twig definidos por Symfony y puede crear sus propios filtros y funciones Twig .

Twig es rápido en el prod entorno (porque las plantillas se compilan en PHP y se almacenan en caché automáticamente), pero es conveniente de usar en el dev entorno (porque las plantillas se vuelven a compilar automáticamente cuando las cambia).

### 8.1 Configuración de Twig

Twig tiene varias opciones de configuración para definir cosas como el formato utilizado para mostrar números y fechas, el almacenamiento en caché de la plantilla, etc. Lea la referencia de configuración de Twig para obtener más información.

### 8.2 Creando Plantillas

Antes de explicar en detalle cómo crear y representar plantillas, mire el siguiente ejemplo para obtener una descripción general rápida de todo el proceso. Primero, debe crear un nuevo archivo en el templates/directorio para almacenar el contenido de la plantilla:

```
index.html.twig X
templates > prueba > index.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Hello PruebaController!{% endblock %}
4
5  {% block body %}
6  <style>
7      .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font:
8      .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
9  </style>
10
11  <div class="example-wrapper">
12      <h1>Hello {{ controller_name }}! ✓</h1>
13      <h2> {{ titulo }} ddel año {{ anno }} </h2>
14      This friendly message is coming from:
15      <ul>
16          <li>Your controller at <code><a href="{{ 'C:/AppServ/www/tutorial3/s
17          <li>Your template at <code><a href="{{ 'C:/AppServ/www/tutorial3/tem
18      </ul>
19  </div>
20  {% endblock %}
21
```

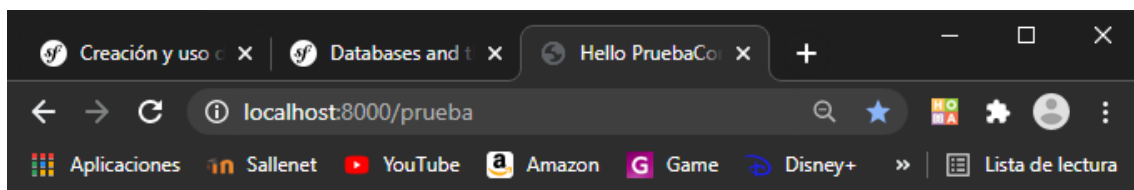
Luego, cree un controlador que renderice esta plantilla y le pase las variables necesarias:

```

src > Controller > PruebaController.php
1  <?php
2
3  namespace App\Controller;
4
5  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6  use Symfony\Component\HttpFoundation\Response;
7  use Symfony\Component\Routing\Annotation\Route;
8
9  class PruebaController extends AbstractController
10 {
11     /**
12      * @Route("/prueba", name="prueba")
13      */
14     public function index(): Response
15     {
16         return $this->render('prueba/index.html.twig', [
17             'controller_name' => 'PruebaController',
18             'titulo' => 'Tutorial Symfony',
19             'anno' => '2021'
20         ]);
21     }
22 }

```

De tal modo se verá las variables de esta forma:



# Hello PruebaController!

## Tutorial Symfony ddel año 2021

This friendly message is coming from:

- Your controller at [src/Controller/PruebaController.php](#)
- Your template at [templates/prueba/index.html.twig](#)

### 8.3 Nombres de plantillas

Symfony recomienda lo siguiente para los nombres de las plantillas:

Uso caso serpiente para nombres de archivos y directorios (por ejemplo blog\_posts.html.twig, admin/default\_theme/blog/index.html.twig, etc.);



Definir dos extensiones para nombres de archivo (por ejemplo, index.html.twig o blog\_posts.xml.twig) siendo la primera extensión ( html, xml, etc.) el formato final de que la plantilla generará.

Aunque las plantillas generalmente generan contenido HTML, pueden generar cualquier formato basado en texto. Es por eso que la convención de dos extensiones simplifica la forma en que se crean y renderizan las plantillas para múltiples formatos.

#### 8.4 Ubicación de la plantilla

Las plantillas se almacenan de forma predeterminada en el templates/directorio. Cuando un servicio o controlador procesa la product/index.html.twig plantilla, en realidad se refieren al <your-project>/templates/product/index.html.twig archivo.

El directorio de plantillas predeterminado se puede configurar con la opción twig.default\_path y puede agregar más directorios de plantillas como se explica más adelante en este artículo.

#### 8.5 Variables de plantilla

Una necesidad común de las plantillas es imprimir los valores almacenados en las plantillas pasadas desde el controlador o el servicio. Las variables suelen almacenar objetos y matrices en lugar de cadenas, números y valores booleanos. Es por eso que Twig proporciona acceso rápido a variables PHP complejas. Considere la siguiente plantilla:

```
1 <p>{{ user.name }} added this comment on {{ comment.publishedAt|date }}</p>
```

La user.name notación significa que desea mostrar cierta información (name) almacenada en una variable (user). ¿Es user una matriz o un objeto? ¿Es name una propiedad o un método? En Twig esto no importa.

Al usar la foo.bar notación, Twig intenta obtener el valor de la variable en el siguiente orden:

1. \$foo['bar'] (matriz y elemento);
2. \$foo->bar (objeto y propiedad pública);
3. \$foo->bar() (objeto y método público);
4. \$foo->getBar() (método de objeto y captador );
5. \$foo->isBar() (método objeto e issuer );
6. \$foo->hasBar() (método de objeto y hasser );
7. Si no existe ninguno de los anteriores, use null (o lance una Twig\Error\RuntimeException excepción si la opción strict\_variables está habilitada).

Esto permite evolucionar el código de su aplicación sin tener que cambiar el código de la plantilla (puede comenzar con variables de matriz para la prueba de concepto de la aplicación, luego pasar a objetos con métodos, etc.)

#### 8.6 Vinculación a páginas

En lugar de escribir las URL de los enlaces a mano, utilice la path() función para generar URL basadas en la configuración de enrutamiento .

Más tarde, si desea modificar la URL de una página en particular, todo lo que tendrá que hacer es cambiar la configuración de enrutamiento: las plantillas generarán automáticamente la nueva URL.

Considere la siguiente configuración de enrutamiento:

```

Anotaciones  YAML  XML  PHP
1  # config/routes.yaml
2  blog_index:
3      path:      /
4      controller: App\Controller\BlogController::index
5
6  blog_post:
7      path:      /article/{slug}
8      controller: App\Controller\BlogController::show

```

Utilice la `path()` función Twig para enlazar a estas páginas y pase el nombre de la ruta como primer argumento y los parámetros de la ruta como segundo argumento opcional:

```

1  <a href="{{ path('blog_index') }}">Homepage</a>
2
3  {# ... #}
4
5  {% for post in blog_posts %}
6      <h1>
7          <a href="{{ path('blog_post', {slug: post.slug}) }}">{{ post.title }}</a>
8      </h1>
9
10     <p>{{ post.excerpt }}</p>
11 {% endfor %}

```

La `path()` función genera URL relativas. Si necesita generar URL absolutas (por ejemplo, al representar plantillas para correos electrónicos o fuentes RSS), use la `url()` función, que toma los mismos argumentos que `path()` (por ejemplo) `<a href="{{ url('blog_index') }}"> ... </a>`

## 8.7 Vinculación a CSS, JavaScript y activos de imagen

Si una plantilla necesita vincularse a un activo estático (por ejemplo, una imagen), Symfony proporciona una `asset()` función Twig para ayudar a generar esa URL. Primero, instale el `asset` paquete:

```
> composer require symfony/asset
```

Ahora puede usar la `asset()` función:

```

1  {# the image lives at "public/images/Logo.png" #}
2  
3
4  {# the CSS file lives at "public/css/blog.css" #}
5  <link href="{{ asset('css/blog.css') }}" rel="stylesheet" />
6
7  {# the JS file lives at "public/bundles/acme/js/loader.js" #}
8  <script src="{{ asset('bundles/acme/js/loader.js') }}"></script>

```

El `asset()` objetivo principal de la función es hacer que su aplicación sea más portátil. Si su aplicación vive en la raíz de su host (por ejemplo `https://example.com`), entonces la ruta renderizada debería ser `/images/logo.png`. Pero si su aplicación vive en un subdirectorio (p `https://example.com/my_app`. Ej. ), Cada ruta de activo debe representarse con el subdirectorio (p `/my_app/images/logo.png`. Ej. ). La `asset()`función se encarga de esto determinando cómo se está utilizando su aplicación y generando las rutas correctas en consecuencia.

Si necesita URL absolutas para los activos, use la `absolute_url()` función Twig de la siguiente manera:

```
1 
2
3 <link rel="shortcut icon" href="{{ absolute_url('favicon.png') }}">
```

## 8.8 Renderizando una plantilla en controladores

Si su controlador se extiende desde `AbstractController` , use el `render()` asistente:

```
// src/Controller/ProductController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;

class ProductController extends AbstractController
{
    public function index(): Response
    {
        // ...

        // the 'render()' method returns a 'Response' object with the
        // contents created by the template
        return $this->render('product/index.html.twig', [
            'category' => '...',
            'promotions' => ['...', '...'],
        ]);

        // the 'renderView()' method only returns the contents created by the
        // template, so you can use those contents later in a 'Response' object
        $contents = $this->renderView('product/index.html.twig', [
            'category' => '...',
            'promotions' => ['...', '...'],
        ]);

        return new Response($contents);
    }
}
```

Si su controlador no se extiende desde `AbstractController`, deberá buscar servicios en su controlador y usar el `render()`método del twig servicio.

## 8.9 Renderizando una plantilla en servicios

Injecta el twig servicio Symfony en tus propios servicios y usa su `render()`método. Al usar el cableado automático del servicio , solo necesita agregar un argumento en el constructor del servicio y escribirlo con la `Twig\Environment` clase:

```
// src/Service/SomeService.php
namespace App\Service;

use Twig\Environment;

class SomeService
{
    private $twig;

    public function __construct(Environment $twig)
    {
        $this->twig = $twig;
    }

    public function someMethod()
    {
        // ...

        $htmlContents = $this->twig->render('product/index.html.twig', [
            'category' => '...',
            'promotions' => ['...', '...'],
        ]);
    }
}
```

#### 8.10 Renderizando una plantilla directamente desde una ruta

Aunque las plantillas generalmente se representan en controladores y servicios, puede representar páginas estáticas que no necesitan ninguna variable directamente desde la definición de la ruta. Utilice el especial `Symfony\Bundle\FrameworkBundle\Controller\TemplateController` proporcionado por Symfony:

```
YAML  XML  PHP

1  # config/routes.yaml
2  acme_privacy:
3      path:          /privacy
4      controller:    Symfony\Bundle\FrameworkBundle\Controller\TemplateController
5      defaults:
6          # the path of the template to render
7          template:   'static/privacy.html.twig'
8
9          # special options defined by Symfony to set the page cache
10         maxAge:      86400
11         sharedAge:   86400
12
13         # whether or not caching should apply for client caches only
14         private:     true
15
16         # optionally you can define some arguments passed to the template
17         context:
18             site_name: 'ACME'
19             theme: 'dark'
```

#### 8.11 Comprobando si existe una plantilla

Las plantillas se cargan en la aplicación mediante un cargador de plantillas Twig, que también proporciona un método para verificar la existencia de la plantilla. Primero, obtenga el cargador:

```
use Twig\Environment;

// this code assumes that your service uses autowiring to inject dependencies
// otherwise, inject the service called 'twig' manually
public function __construct(Environment $twig)
{
    $loader = $twig->getLoader();
}
```

Luego, pase la ruta de la plantilla Twig al exists() método del cargador:

### 8.12 Inspeccionando la información de Twig

El debug:twig comando enumera toda la información disponible sobre Twig (funciones, filtros, variables globales, etc.). Es útil verificar si sus extensiones Twig personalizadas están funcionando correctamente y también verificar las características de Twig agregadas al instalar paquetes:

```
# List general information
> php bin/console debug:twig

# filter output by any keyword
> php bin/console debug:twig --filter=date

# pass a template path to show the physical file which will be loaded
> php bin/console debug:twig @Twig/Exception/error.html.twig
```

### 8.13 Incluyendo Plantillas

Si cierto código Twig se repite en varias plantillas, puede extraerlo en un único "fragmento de plantilla" e incluirlo en otras plantillas. Imagine que el siguiente código para mostrar la información del usuario se repite en varios lugares:

```
1  {# templates/blog/index.html.twig #}
2
3  {# ... #}
4  <div class="user-profile">
5      
6      <p>{{ user.fullName }} - {{ user.email }}</p>
7  </div>
```

Primero, cree una nueva plantilla Twig llamada blog/\_user\_profile.html.twig (el \_prefijo es opcional, pero es una convención que se usa para diferenciar mejor entre plantillas completas y fragmentos de plantillas).

Luego, elimine ese contenido de la blog/index.html.twig plantilla original y agregue lo siguiente para incluir el fragmento de la plantilla:

```
1  {# templates/blog/index.html.twig #}
2
3  {# ... #}
4  {{ include('blog/_user_profile.html.twig') }}
```

La `include()` función Twig toma como argumento la ruta de la plantilla a incluir. La plantilla incluida tiene acceso a todas las variables de la plantilla que la incluye (use la opción `with_context` para controlar esto).

También puede pasar variables a la plantilla incluida. Esto es útil, por ejemplo, para cambiar el nombre de las variables. Imagine que su plantilla almacena la información del usuario en una variable llamada `blog_post.author` lugar de la `user` variable que espera el fragmento de plantilla. Utilice lo siguiente para cambiar el nombre de la variable:

```
1  {% templates/blog/index.html.twig %}
2
3  {% ... %}
4  {{ include('blog/_user_profile.html.twig', {user: blog_post.author}) }}
```

#### 8.14 Incorporación de controladores

La inclusión de fragmentos de plantilla es útil para reutilizar el mismo contenido en varias páginas. Sin embargo, esta técnica no es la mejor solución en algunos casos.

Imagine que el fragmento de plantilla muestra los tres artículos de blog más recientes. Para hacer eso, necesita realizar una consulta en la base de datos para obtener esos artículos. Cuando utilice la `include()` función, deberá realizar la misma consulta de base de datos en cada página que incluya el fragmento. Esto no es muy conveniente.

Una mejor alternativa es incrustar el resultado de ejecutar algún controlador con las funciones `render()` y `controller()` Twig.

Primero, cree el controlador que muestre una cierta cantidad de artículos recientes:

```
// src/Controller/BlogController.php
namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
// ...

class BlogController extends AbstractController
{
    public function recentArticles(int $max = 3): Response
    {
        // get the recent articles somehow (e.g. making a database query)
        $articles = ['...', '...', '...'];

        return $this->render('blog/_recent_articles.html.twig', [
            'articles' => $articles
        ]);
    }
}
```

Luego, cree el `blog/_recent_articles.html.twig` fragmento de plantilla (el `_` prefijo en el nombre de la plantilla es opcional, pero es una convención que se usa para diferenciar mejor entre plantillas completas y fragmentos de plantilla):

```

1  {% templates/blog/_recent_articles.html.twig %}
2  {% for article in articles %}
3      <a href="{ path('blog_show', {slug: article.slug}) }}" >
4          {{ article.title }}
5      </a>
6  {% endfor %}

```

Ahora puede llamar a este controlador desde cualquier plantilla para incrustar su resultado:

```

1  {% templates/base.html.twig %}
2
3  {% ... %}
4  <div id="sidebar">
5      {% if the controller is associated with a route, use the path() or url() functions %}
6      {{ render(path('latest_articles', {max: 3})) }}
7      {{ render(url('latest_articles', {max: 3})) }}
8
9      {% if you don't want to expose the controller with a public URL,
10         use the controller() function to define the controller to execute %}
11      {{ render(controller(
12          'App\Controller\BlogController::recentArticles', {max: 3}
13      )) }}
14  </div>

```

Cuando se utiliza la `controller()` función, no se accede a los controladores mediante una ruta Symfony normal, sino a través de una URL especial que se utiliza exclusivamente para servir esos fragmentos de plantilla. Configure esa URL especial en la `fragments` opción:

```

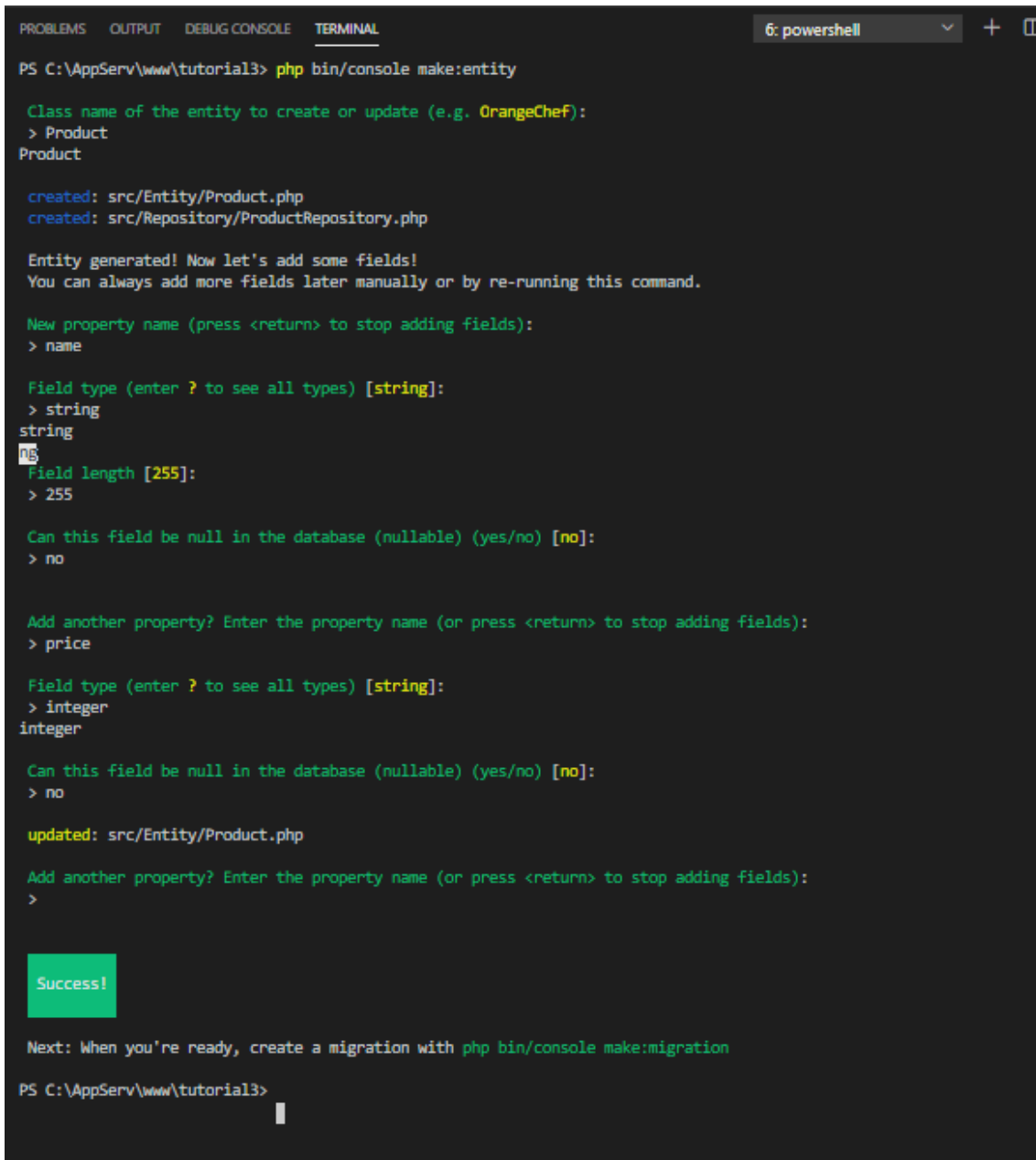
YAML  XML  PHP
1  # config/packages/framework.yaml
2  framework:
3      # ...
4      fragments: { path: /_fragment }

```

## 9. Crear una entidad

Suponga que está creando una aplicación en la que deben mostrarse productos. Sin siquiera pensar en Doctrine o bases de datos, ya sabe que necesita un `Product` objeto para representar esos productos.

Puede usar el `make:entity` comando para crear esta clase y cualquier campo que necesite. El comando le hará algunas preguntas; respóndalas como se hace a continuación:



```
PS C:\AppServ\www\tutorial3> php bin/console make:entity

Class name of the entity to create or update (e.g. OrangeChef):
> Product
Product

created: src/Entity/Product.php
created: src/Repository/ProductRepository.php

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

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

Field type (enter ? to see all types) [string]:
> string
string
Field length [255]:
> 255

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

Add another property? Enter the property name (or press <return> to stop adding fields):
> price

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

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

updated: src/Entity/Product.php

Add another property? Enter the property name (or press <return> to stop adding fields):
>

Success!

Next: When you're ready, create a migration with php bin/console make:migration

PS C:\AppServ\www\tutorial3>
```

Como apreciamos en la captura, podemos manejar con dicho comando la construcción de los atributos de nuestra entidad Product y nos genera un archivo Product.php como consecuencia de nuestra entidad.

El make:entity comando es una herramienta para hacer la vida más fácil. Pero este es su código: agregue / elimine campos, agregue / elimine métodos o actualice la configuración.

En la siguiente captura verás el diseño que tiene una entidad la cual podrá manipular a su gusto.



```

Product.php X
src > Entity > Product.php
1  <?php
2
3  namespace App\Entity;
4
5  use App\Repository\ProductRepository;
6  use Doctrine\ORM\Mapping as ORM;
7
8  /**
9   * @ORM\Entity(repositoryClass=ProductRepository::class)
10  */
11  class Product
12  {
13      /**
14       * @ORM\Id
15       * @ORM\GeneratedValue
16       * @ORM\Column(type="integer")
17       */
18      private $id;
19
20      /**
21       * @ORM\Column(type="string", length=255)
22       */
23      private $name;
24
25      /**
26       * @ORM\Column(type="integer")
27       */
28      private $price;
29
30      public function getId(): ?int
31      {
32          return $this->id;
33      }
34
35      public function getName(): ?string
36      {
37          return $this->name;
38      }
39
40      public function setName(string $name): self
41      {
42          $this->name = $name;
43
44          return $this;
45      }
46
47      public function getPrice(): ?int
48      {
49          return $this->price;

```

### 9.1 Migraciones: Creación de tablas / esquema de base de datos.

La Product clase está completamente configurada y lista para guardar en una producto tabla. Si acaba de definir esta clase, su base de datos todavía no tiene la product tabla. Para agregarlo, puede aprovechar DoctrineMigrationsBundle , que ya está instalado:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
7: powershell

Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\AppServ\www\tutorial13> php bin/console make:migration
[2021-04-14T22:23:05.726682+00:00] doctrine.DEBUG: SHOW FULL TABLES WHERE Table_type = 'BASE TABLE' [] []
[2021-04-14T22:23:05.868779+00:00] doctrine.DEBUG: SHOW FULL TABLES WHERE Table_type = 'BASE TABLE' [] []
[2021-04-14T22:23:06.032371+00:00] doctrine.DEBUG: SHOW FULL TABLES WHERE Table_type = 'BASE TABLE' [] []

Success!

Next: Review the new migration "migrations/Version20210414222305.php"
Then: Run the migration with php bin/console doctrine:migrations:migrate
See https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html
PS C:\AppServ\www\tutorial13>

```

Si vemos el archivo que nos generó nos muestra lo siguiente:

```

migrations > Version20210414222305.php
1  <?php
2
3  declare(strict_types=1);
4
5  namespace DoctrineMigrations;
6
7  use Doctrine\DBAL\Schema\Schema;
8  use Doctrine\Migrations\AbstractMigration;
9
10 /**
11  * Auto-generated Migration: Please modify to your needs!
12  */
13 final class Version20210414222305 extends AbstractMigration
14 {
15     public function getDescription() : string
16     {
17         return '';
18     }
19
20     public function up(Schema $schema) : void
21     {
22         // this up() migration is auto-generated, please modify it to your needs
23         $this->addSql('CREATE TABLE product (id INT AUTO_INCREMENT NOT NULL, name VARCHAR(255)
24     )
25
26     public function down(Schema $schema) : void
27     {
28         // this down() migration is auto-generated, please modify it to your needs
29         $this->addSql('DROP TABLE product');
30     }
31 }
32

```

Este comando ejecuta todos los archivos de migración que aún no se han ejecutado en su base de datos. Debe ejecutar este comando en producción cuando implemente para mantener actualizada su base de datos de producción.

*php bin/console doctrine:migrations:migrate*

```

PS C:\AppServ\www\tutorial3> php bin/console doctrine:migrations:migrate

WARNING! You are about to execute a migration in database "tutorial" that could result in schema changes and data loss. Are you
sure you wish to continue? (yes/no) [yes]:
> yes

[2021-04-14T22:26:23.040141+00:00] doctrine.DEBUG: SHOW FULL TABLES WHERE Table_type = 'BASE TABLE' [] []
[2021-04-14T22:26:23.092408+00:00] doctrine.DEBUG: CREATE TABLE doctrine_migration_versions (version VARCHAR(191) NOT NULL, exe
uted_at DATETIME DEFAULT NULL, execution_time INT DEFAULT NULL, PRIMARY KEY(version)) DEFAULT CHARACTER SET utf8 COLLATE `utf8_u
nicode_ci` ENGINE = InnoDB [] []
[2021-04-14T22:26:23.385134+00:00] doctrine.DEBUG: SHOW FULL TABLES WHERE Table_type = 'BASE TABLE' [] []
[2021-04-14T22:26:23.390444+00:00] doctrine.DEBUG: SHOW FULL TABLES WHERE Table_type = 'BASE TABLE' [] []
[2021-04-14T22:26:23.398252+00:00] doctrine.DEBUG: SELECT COLUMN_NAME AS Field, COLUMN_TYPE AS Type, IS_NULLABLE AS `Null`, COL
UMN_KEY AS `Key`, COLUMN_DEFAULT AS `Default`, EXTRA AS Extra, COLUMN_COMMENT AS Comment, CHARACTER_SET_NAME AS CharacterSet, COL
LATION_NAME AS Collation FROM information_schema.COLUMNS WHERE TABLE_SCHEMA = 'tutorial' AND TABLE_NAME = 'doctrine_migration_ve
rsions' ORDER BY ORDINAL_POSITION ASC [] []
[2021-04-14T22:26:23.403704+00:00] doctrine.DEBUG: SELECT DISTINCT `c`.`CONSTRAINT_NAME` AS `c`, `c`.`COLUMN_NAME` AS `c`, `r`.`REFERENCED_TABLE_NAM

```

Y su correspondiente vista en la base de datos

The screenshot shows a MySQL database management interface. At the top, the breadcrumb navigation indicates the current location: **Servidor: localhost » Base de datos: tutorial » Tabla: product**. Below this, there are several tabs: **Examinar** (selected), **Estructura**, **SQL**, **Buscar**, and **Insertar**. A green message box states: **MySQL ha devuelto un conjunto de valores vacío (es decir: cero columnas).** (La consulta devolvió un conjunto de valores vacío). Below this, the SQL query `SELECT * FROM `product`` is displayed. A checkbox for **Perfilando** is present, along with links for **[Editar en línea]**, **[Editar]**, and **[Explicar SQL]**. The query result is shown as a table with three columns: **id**, **name**, and **price**. Below the table, there is a section titled **Operaciones sobre los resultados de la consulta** with a button for **Crear vista**.

## 9.2 Persistencia en la base de datos

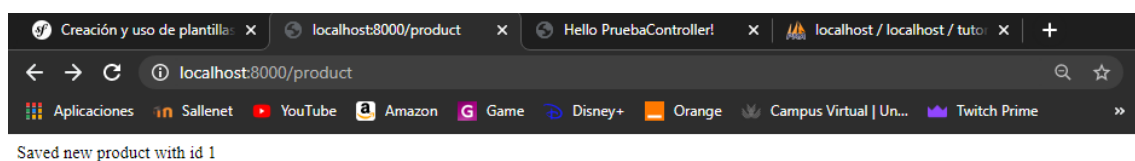
En el controlador que vamos a crear para la entidad producto, vamos a meter el siguiente código:

```

ProductController.php x index.html.twig
src > Controller > ProductController.php
1  <?php
2
3  namespace App\Controller;
4
5  use App\Entity\Product;
6  use Doctrine\ORM\EntityManagerInterface;
7
8  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
9  use Symfony\Component\HttpFoundation\Response;
10 use Symfony\Component\Routing\Annotation\Route;
11
12 class ProductController extends AbstractController
13 {
14
15
16     /**
17      * @Route("/product", name="create_product")
18      */
19     public function createProduct(): Response
20     {
21         // you can fetch the EntityManager via $this->getDoctrine()
22         // or you can add an argument to the action: createProduct(EntityManagerInterface $entityManager)
23         $entityManager = $this->getDoctrine()->getManager();
24
25         $product = new Product();
26         $product->setName('Keyboard');
27         $product->setPrice(1999);
28
29         // tell Doctrine you want to (eventually) save the Product (no queries yet)
30         $entityManager->persist($product);
31
32         // actually executes the queries (i.e. the INSERT query)
33         $entityManager->flush();
34
35         return new Response('Saved new product with id '.$product->getId());
36     }
37
38

```

Crearé una nueva instancia de producto, guardará dicha entidad y devolverá el mensaje de guardado pertinente.



Y su muestra de guardado en la base de datos:

✓ Mostrando filas 0 - 0 (total de 1, La consulta tardó 0,0008 segundos.)

SELECT \* FROM `product`

☐ Perfilando [Editar en línea] [Editar] [Explicar]

☐ Mostrar todo | Número de filas: 25 | Filtrar filas:

+ Opciones

	id	name	price
<input type="checkbox"/> Editar <input type="checkbox"/> Copiar <input type="checkbox"/> Borrar	1	Keyboard	1999

### 9.3 Obteniendo objetos de la base de datos.

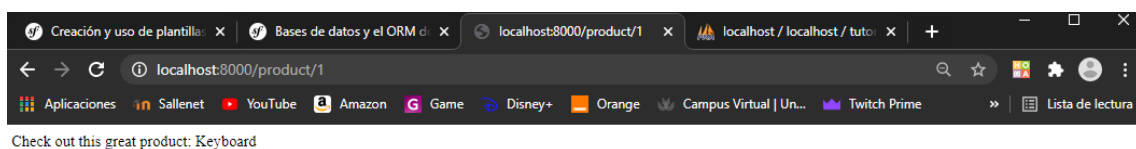
Recuperar un objeto de la base de datos es aún más fácil. Suponga que desea poder ir /product/1 a ver su nuevo producto, para ello, creamos una nueva función con su ruta en el controlador:

```

37
38  /**
39   * @Route("/product/{id}", name="product_show")
40   */
41  public function show(int $id): Response
42  {
43      $product = $this->getDoctrine()
44                  ->getRepository(Product::class)
45                  ->find($id);
46
47      if (!$product) {
48          throw $this->createNotFoundException(
49              'No product found for id '.$id
50          );
51      }
52
53      return new Response('Check out this great product: '.$product->getName());
54
55      // or render a template
56      // in the template, print things with {{ product.name }}
57      // return $this->render('product/show.html.twig', ['product' => $product]);
58  }
59
60

```

Y posteriormente hacemos la llamada:



Para actualizar un producto bastaría con la siguiente función:

```

60  ▾  /**
61      * @Route("/product/edit/{id}")
62      */
63      public function update(int $id): Response
64      {
65          $entityManager = $this->getDoctrine()->getManager();
66          $product = $entityManager->getRepository(Product::class)->find($id);
67
68          if (!$product) {
69              throw $this->createNotFoundException(
70                  'No product found for id '.$id
71              );
72          }
73
74          $product->setName('New product name!');
75          $entityManager->flush();
76
77          return $this->redirectToRoute('product_show', [
78              'id' => $product->getId()
79          ]);
80      }
81  }
82

```

Y para hacer su eliminación se usaría el método `remove` (Como era de esperar, el `remove()` método notifica a Doctrine que le gustaría eliminar el objeto dado de la base de datos. La DELETE consulta no se ejecuta realmente hasta `flush()` que se llama al método.)

```

$entityManager->remove($product);
$entityManager->flush();

```

## 9.4 Consulta Compleja

```
// src/Repository/ProductRepository.php

// ...
class ProductRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Product::class);
    }

    /**
     * @return Product[]
     */
    public function findAllGreaterThanPrice(int $price): array
    {
        $entityManager = $this->getEntityManager();

        $query = $entityManager->createQuery(
            'SELECT p
            FROM App\Entity\Product p
            WHERE p.price > :price
            ORDER BY p.price ASC'
        )->setParameter('price', $price);

        // returns an array of Product objects
        return $query->getResult();
    }
}
```

La cadena pasada a `createQuery()` puede parecerse a SQL, pero es Doctrine Query Language. Esto le permite escribir consultas utilizando un lenguaje de consulta comúnmente conocido, pero haciendo referencia a objetos PHP en su lugar (es decir, en la FROM declaración).

Ahora, puede llamar a este método en el repositorio:

```
// from inside a controller
$minPrice = 1000;

$products = $this->getDoctrine()
    ->getRepository(Product::class)
    ->findAllGreaterThanPrice($minPrice);

// ...
```

## 10. Consejos para hacer una aplicación en Symfony

Habiendo repasado los conceptos MVC y características de este manual y si hiciera falta un vistazo a la documentación que proporciona el sitio web oficial, se puede hacer una web de artículos creando la entidad artículos con su controlador y sus plantillas y definiendo correctamente las rutas.

<https://symfony.com/doc/current/index.html>