http://www.robertsallent.com
@robertsallent

v.21.10

SYM13: *Responses*

Fundamentos de *HTTP*: respuestas

Robert Sallent
www.robertsallent.com

SOC Servei d'Ocupació de Catalunya
CIFO Sabadell- Mancomunitat (Vallès)
Generalitat de Catalunya
Unió Europea
Fons social europeu
L'FSE inverteix en el teu futur

Índice

- El protocolo *HTTP*.
- *Responses*
  - Redirecciones
  - Respuestas en *JSON*
  - *JSON – XML – CSV*
  - Anexando *cookies*
  - Sirviendo ficheros

HTTP

- Las aplicaciones web que desarrollamos se encuentran en capa de aplicación (servicio *WWW*) y la información es transmitida mediante el protocolo *HTTP*.

- *HTTP* (*HyperText Transfer Protocol*) permite transferencias de información en la *World Wide Web* y su aparición data de 1999.

- El RFC 2616 especifica la versión 1.1 , definiendo la sintaxis y semántica que utilizan los elementos de software de la arquitectura web para comunicarse.

HTTP

- Es un protocolo **orientado a transacciones** y sigue un esquema de **petición-respuesta** entre un **cliente** y un **servidor**.

- El cliente (*user agent*) realiza una petición (*REQUEST*) al servidor (servidor web) y éste le envía un mensaje de respuesta (*RESPONSE*).

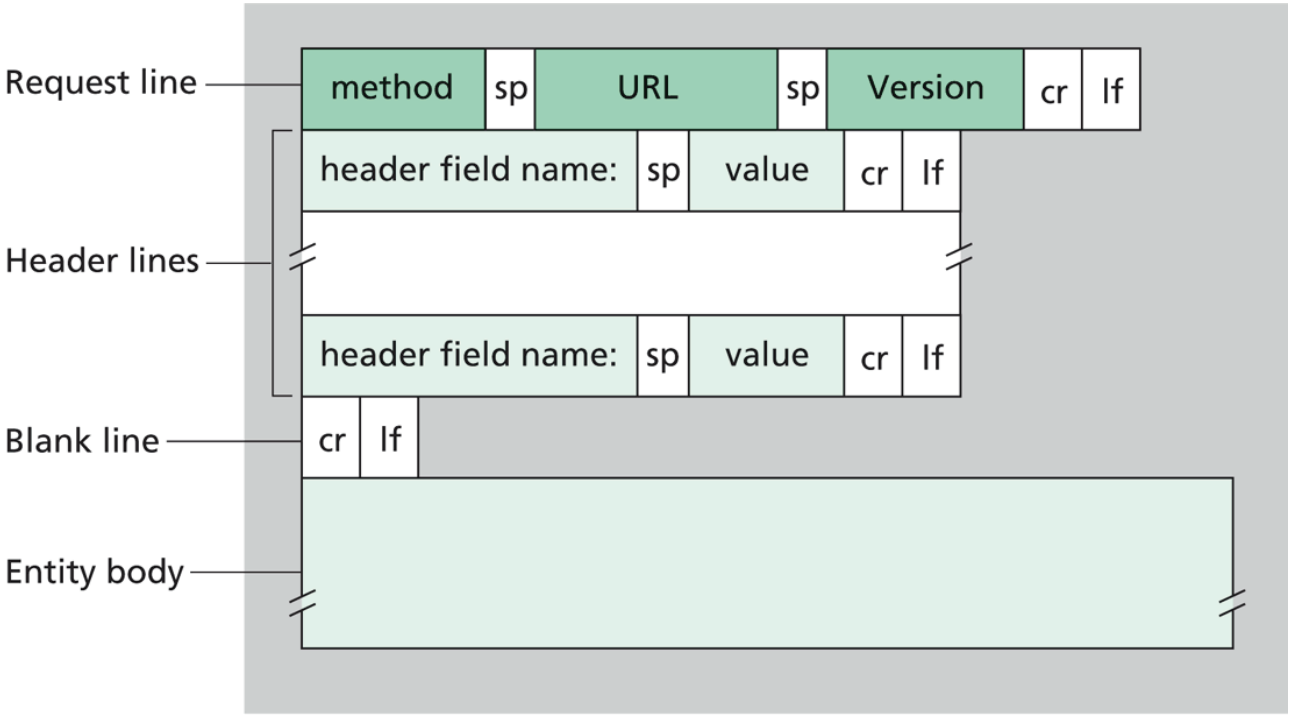
  - Ejemplos de clientes: navegadores web (*Firefox*, *Chrome*, *Opera*, *Safari*…).
  - Ejemplos de servidor web: *Apache* o *lightHTTP*.

HTTP

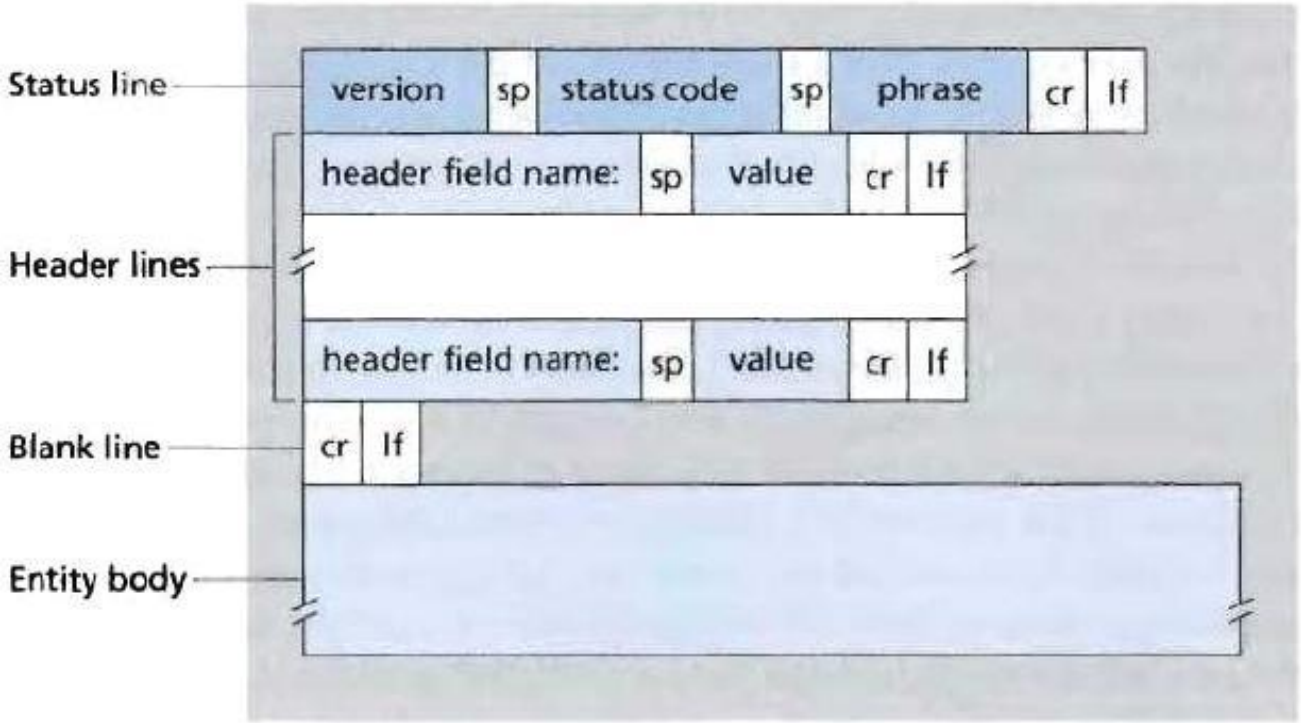
- Los mensajes se transmiten en texto plano, por lo que son más legibles y fáciles de depurar, el inconveniente es que son más largos. Su estructura es:

  - La **línea inicial** (termina con retorno de carro y salto de línea) contiene:
    - Para las peticiones: el método de petición, la *URL* del recurso y la versión *HTTP* del cliente.
    - Para las respuestas: la versión *HTTP* seguida del código de respuesta y la frase asociada a dicho código.
  - Las **cabeceras** (terminan con una línea en blanco) con metadatos.
  - El **cuerpo** del mensaje (opcional), conteniendo los datos que se intercambian entre cliente y servidor.

Ejemplo mensaje *HTTP* (petición)



Ejemplo mensaje *HTTP* (respuesta)



Ejemplo petición - respuesta

Symfony

Petición:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: nombre-cliente
[Línea en blanco]
```

Respuesta:

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 2003 23:59:59 GMT
Content-Type: text/html
Content-Length: 1221

<html>
<body>
<h1>Página principal de tuHost</h1>
(Contenido)
   .
   .
   .
</body>
</html>
```

APACHE HTTP SERVER

Métodos *HTTP*

- En el ejemplo anterior se muestra el formato de un mensaje que utiliza el **método** *GET*, que sirve para solicitar un recurso.

- No es el único método de *HTTP*, el protocolo define los siguientes métodos:

  - **GET**: obtener el documento especificado.
  - **HEAD**: obtener la cabecera de un documento (usado para saber si el documento existe).
  - **POST**: envía información al servidor junto con un documento. Se usa en formularios.

Métodos *HTTP*

- **PUT**: sube el documento al servidor.
- **DELETE:** elimina el documento del servidor, siempre y cuando se tenga permiso.
- **TRACE**: el servidor retorna una copia de la petición.

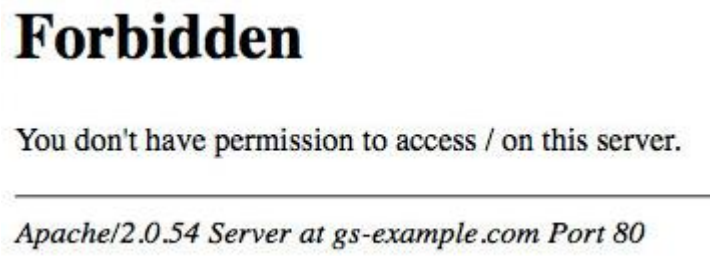
  Se usa para monitorear las transformaciones que ha sufrido el mensaje inicial al pasar pasarelas o servidores intermedios.
- **OPTIONS**: el servidor responde indicando qué métodos acepta.

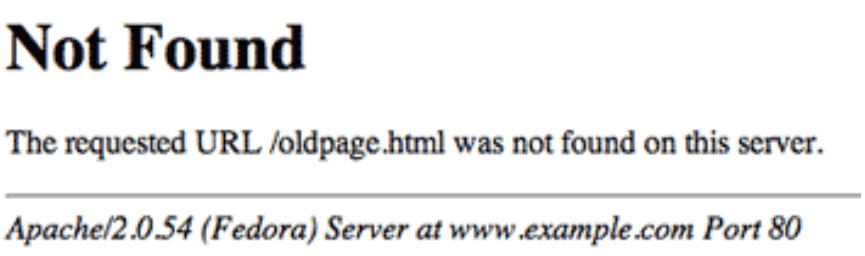
- En las *APIS RESTFUL*, los métodos *GET*, *POST*, *PUT* y *DELETE* se corresponden con las operaciones del *CRUD*: recuperar, crear, actualizar y borrar respectivamente.

Códigos de respuesta *HTTP*

- Los mensajes de respuesta contienen un código y una frase asociada al mensaje.

- Cuando se realiza la petición, pueden pasar muchas cosas: todo va bien (*200*), no se encontró el documento solicitado (*404*), acceso prohibido (*403*), el documento ya no está disponible (*410*), error interno del servidor (*500*)…





Códigos de respuesta *HTTP*

- Cada situación se asocia a un código:
  - **1xx**: información genérica
  - **2xx**: éxito
  - **3xx**: redirección
  - **4xx**: error del cliente
  - **5xx**: error en el servidor.
- Podéis encontrar el listado de mensajes en multitud de sitios, como por ejemplo en:
  https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

En *Symfony*

- Esta conversación petición-respuesta es el proceso fundamental para todas las comunicaciones en la web.

- El objetivo final de nuestras aplicaciones, es **comprender siempre cada una de las peticiones recibidas y crear y retornar la respuesta adecuada**.

- *Symfony* nos provee de una forma para trabajar tanto con peticiones como con respuestas mediante dos clases (*Request* y *Response*) que forman el componente *http-foundation*.

En *Symfony*

- Este componente define una capa orientada a objeto para las especificaciones de *HTTP*.

- En *PHP* llano:

  - El análisis de las peticiones se implementa haciendo uso de las variables superglobales: *$_GET, $_POST, $_FILES, $_COOKIE, $_SESSION*…
  - La generación de respuestas usando *echo, header(), setCookie()*…

- *HttpFoundation* reemplaza todo esto por clases con métodos que nos permitirán hacer todas las operaciones de forma más sencilla.

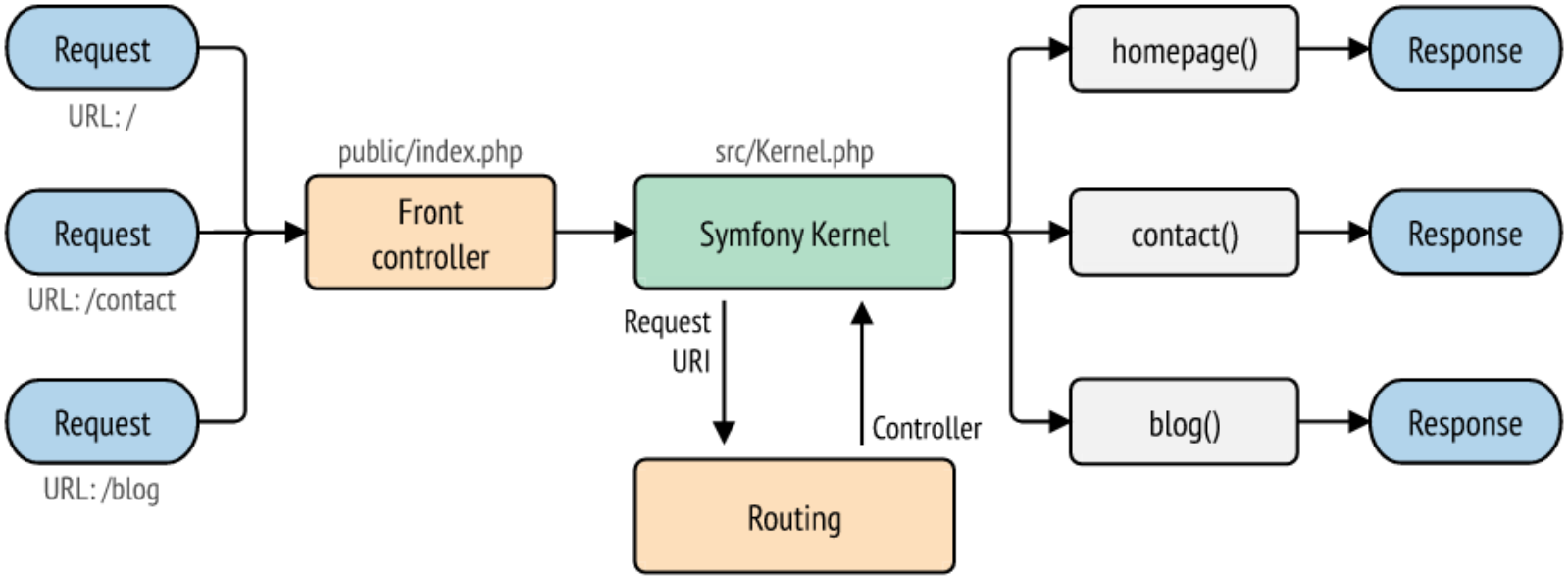
En *Symfony*

- Nuestra tarea será implementar la lógica que interprete las peticiones y que genere las respuestas adecuadas.

- Para ello, también contaremos con la ayuda de un **controlador frontal** (*FrontController*).
  - Cualquier petición a la aplicación será gestionada por este controlador.
  - El módulo de reescritura de direcciones del servidor web (*mod_rewrite*) será el responsable de permitirnos hacer uso de *URLs* amigables.

- Y de un ***router*** para dirigir correctamente las peticiones hacia los métodos adecuados.

Ejemplo en *Symfony*



En *Symfony*

- Resumiendo:

1. Un cliente realiza una petición.
2. La petición llega al controlador frontal.
3. El controlador frontal arranca *Symfony* y pasa la información de la petición.
4. Internamente, *Symfony* usa rutas y controladores para crear la respuesta.
5. *Symfony* convierte el objeto respuesta en los *headers* y contenido que será enviado al cliente.

Responses

Respuestas en *Symfony*

Robert Sallent
www.robertsallent.com

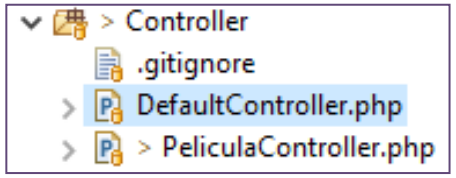
SOC Servei d'Ocupació de Catalunya
CIFO Sabadell- Mancomunitat (Vallès)
Generalitat de Catalunya
Unió Europea
Fons social europeu
L'FSE inverteix en el teu futur

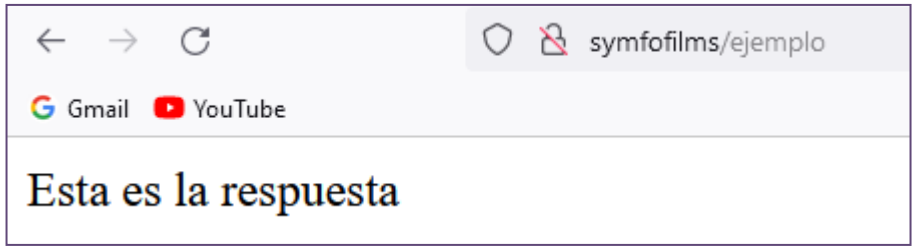
Response

- Las respuestas que debe generar nuestra aplicación serán objetos de la clase *Symfony\Component\HttpFoundation\Response*.
  - En los ejemplos ya hemos visto que podemos generar respuestas a partir de una simple cadena de texto o una página web completa.
- Este objeto almacena toda la información que se debe enviar al cliente para una petición determinada.
- El constructor recibe tres argumentos: el contenido de la respuesta, el código de estado y un *array* de encabezados *HTTP*.

Ejemplo creando una respuesta con el constructor

```php
/**
 * @Route("/ejemplo")
 */
public function ejemplo():Response{

    // crea una respuesta
    $response = new Response(
            'Esta es la respuesta',          // contenido
            Response::HTTP_OK,               // 200 (OK)
            ['content-type' => 'text/html']  // encabezados
        );

    return $response;
}
```





Response

- Podemos incorporar o actualizar la información a posteriori, usando diversos métodos *setter* o las propiedades que iremos viendo a continuación, por ejemplo:

```
$response->headers->set('content-type', 'application/json');
```

- El método `setCharset()` nos permitirá indicar el conjunto de caracteres utilizado en la respuesta. Por defecto *Symfony* asumirá que la respuesta debe ser *UTF-8*.

```
$response->setCharset('ISO-8859-1');
```

Enviando la respuesta

- Desde un método de controlador, debemos retornar una respuesta, así que tan solo tendríamos que hacer el *return* del objeto.

- Existe un método `send()` nos permite enviar la respuesta desde cualquier punto de nuestra aplicación.

```
$response->send();
```

- Antes de enviar la respuesta, opcionalmente podemos llamar el método `prepare()` para evitar que exista alguna incompatibilidad con la especificación *HTTP* indicada en la petición.

```
$response->prepare($request);
```

Redirects

Redireccionando peticiones

Robert Sallent
www.robertsallent.com

SOC Servei d'Ocupació de Catalunya
CIFO Sabadell- Mancomunitat (Vallès)
Generalitat de Catalunya
Unió Europea
Fons social europeu
L'FSE inverteix en el teu futur

Redirecciones

- Es habitual que, tras realizar una determinada operación, la aplicación nos deba redireccionar hacia otro lugar (otra ruta distinta).

- Por ejemplo:

  - Tras actualizar una película, se puede redireccionar hacia la página de detalles de la misma (que además nos debería mostrar un mensaje de confirmación de que la operación ha sido realizada correctamente.
  - Si no aceptamos los términos, se nos puede redireccionar hacia un sitio externo (Google, por ejemplo).

Redirecciones

- **Las redirecciones son un tipo de *Response***.

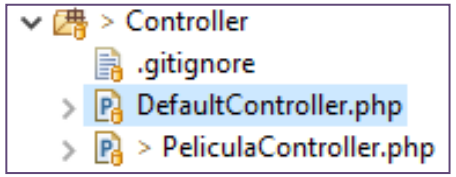
- Para hacer redirecciones externas, podemos crear un objeto de tipo *RedirectResponse* de la siguiente forma:

```
$response = new RedirectResponse('https://juegayestudia.com');
```

- Y para las redirecciones internas, tan solo tenemos que indicar la ruta de la operación a realizar.

```
$response = new RedirectResponse('/pelicula');
```

Ejemplo redirección a un lugar externo

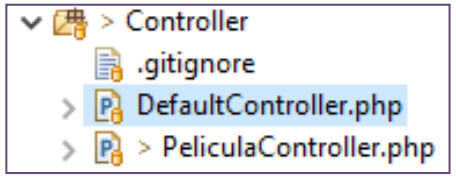


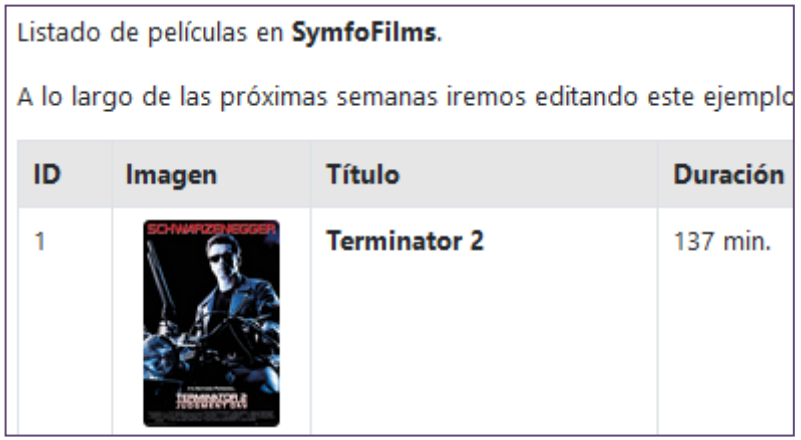

```php
/**
 * @Route("/out")
 */
public function out():Response{

    $response = new RedirectResponse('https://juegayestudia.com');

    return $response;
}
```

Ejemplo resultado

Ejemplo redirección interna

```php
/**
 * @Route("/redireccion")
 */
public function redireccion():Response{
    return new RedirectResponse('/peliculas');
}
```





Redirecciones

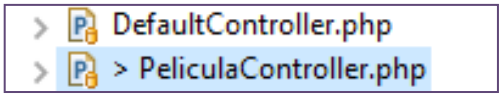
- Para redirecciones más avanzadas, podemos usar los métodos `redirect()` o `redirectToRoute()` de la clase *AbstractContoller*.

- Como estos métodos son heredados por nuestros controladores, los podemos usar así:

```
$this->redirect('https://juegayestudia.com');
```

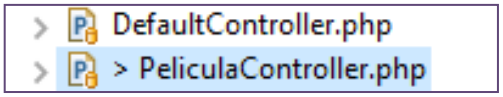
- La gracia del método `redirectToRoute()` es que nos permitirá añadir los parámetros de la redirección a modo de *array* asociativo.

Ejemplo redirección tras editar la peli



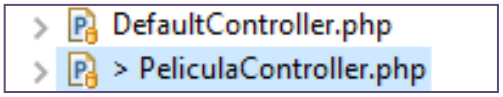

```php
/**
 * @Route("/pelicula/edit/{id}", name="pelicula_edit")
 */
public function edit(Pelicula $peli, Request $request):Response{
    // crea el formulario
    $formulario = $this->createForm(PeliculaFormType::class, $peli, ['method' => 'POST']);
    $formulario->handleRequest($request);

    // si el formulario fue enviado y es válido...
    if ($formulario->isSubmitted() && $formulario->isValid()) {

        // guarda los cambios en la BDD
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($peli);     // indica a doctrine que queremos guardar la peli
        $entityManager->flush();            // ejecuta las consultas

        // redirige a "ver detalles de la peli"
        // eliminaremos el mensaje de confirmación, pero no pasa nada ya que lo volveremos
        // a poner más adelante, cuando hablemos de sesiones y flasheo
        return $this->redirectToRoute('pelicula_show',['id' => $peli->getId()]);
    }

    return $this->render("pelicula/edit.html.twig", [     // carga la vista con el formulario
           "formulario"=>$formulario->createView(),
           "pelicula" => $peli
    ]);
}
```

Ejemplo redirección tras crear la peli




```php
/**
 * @Route("/pelicula/create", name="pelicula_create")
 */
public function create(Request $request):Response{
    $peli = new Pelicula(); // crea el objeto de tipo Pelicula

    // crea el formulario
    $formulario = $this->createForm(PeliculaFormType::class, $peli, ['method' => 'POST']);
    $formulario->handleRequest($request);

    // si el formulario fue enviado y es válido...
    if ($formulario->isSubmitted() && $formulario->isValid()) {

        // almacenar los datos de la peli en la BDD
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($peli);     // indica a doctrine que queremos guardar la peli
        $entityManager->flush();            // ejecuta las consultas

        // redirige a la vista de detalles
        return $this->redirectToRoute('pelicula_show', ['id' => $peli->getId()]);
    }

    // carga la vista con el formulario
    return $this->render("pelicula/create.html.twig", ["formulario"=>$formulario->createView()]);
}
```

Ejemplo redirección tras eliminar la peli




```php
/**
 * @Route("/pelicula/delete/{id}", name="pelicula_delete")
 */
public function delete(Pelicula $peli, Request $request):Response{
    // creación del formulario
    $formulario = $this->createForm(PeliculaDeleteFormType::class, $peli, ['method' => 'POST']);
    $formulario->handleRequest($request);

    // si el formulario llega y es válido...
    if ($formulario->isSubmitted() && $formulario->isValid()) {

        // borra la película de la BDD
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->remove($peli);  // indica que hay que borrar la película
        $entityManager->flush();        // aplicamos los cambios

        return $this->redirectToRoute('pelicula_list'); // redirigimos a la lista de películas
    }

    // muestra el formulario de confirmación de borrado
    return $this->render("pelicula/delete.html.twig", [
        "formulario"=>$formulario->createView(),
        "pelicula" => $peli
    ]);
}
```

JSON

Respondiendo en *JSON*

Robert Sallent
www.robertsallent.com

SOC Servei d'Ocupació de Catalunya
CIFO Sabadell- Mancomunitat (Vallès)
Generalitat de Catalunya
Unió Europea
Fons social europeu
L'FSE inverteix en el teu futur

Respondiendo en *JSON*

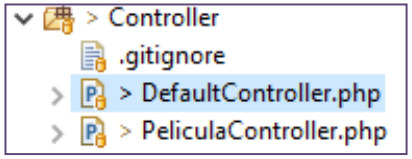
- Cuando creamos servicios o *APIs* completas, uno de los formatos más extendidos para las respuestas es el ***Javascript Object Notation*** o ***JSON***.

- Como ya sabemos, podemos usar la función *json_encode()* de *PHP* para generar código *JSON* a partir de objetos *PHP*.

  - En este caso, debemos añadir el contenido *JSON* a la respuesta e indicar en el encabezado que el *Content-Type* es *application/json*.

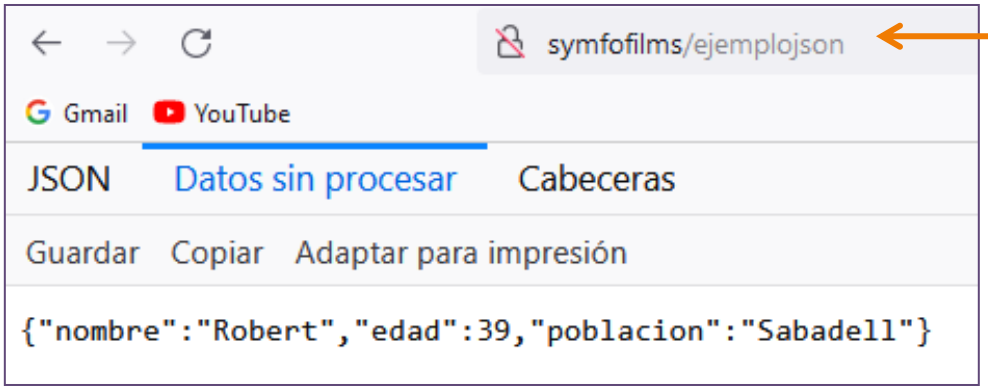
- Pero también podemos generar directamente respuestas *JSON* usando la clase *JsonResponse*, que hereda de *Response*.

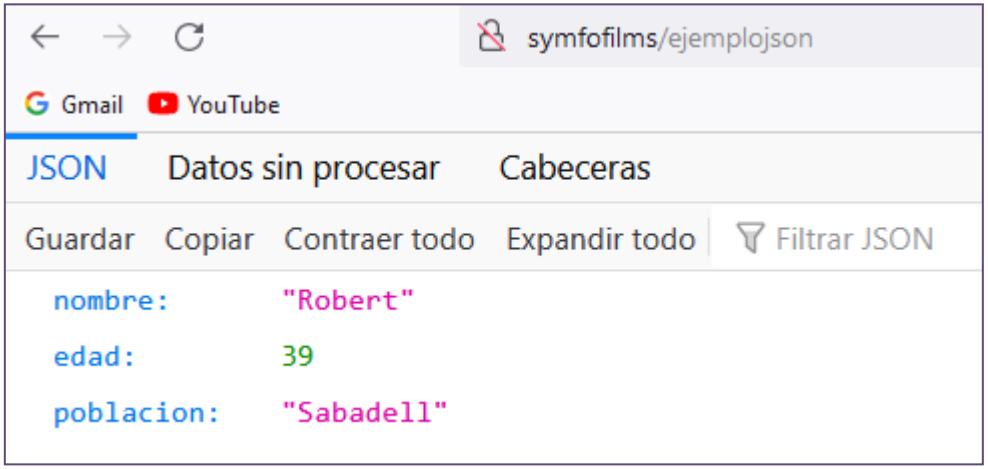
Ejemplo generando *JSON* con *json_encode()*



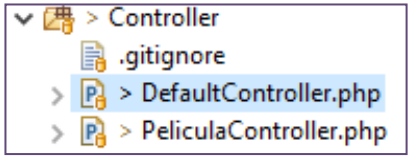

```php
/**
 * @Route("/ejemplojson")
 */
public function ejemploJson():Response{

    // creo un objeto stdClass y le coloco algunos datos
    $persona = new \stdClass();  // la contrabarra es por el namespace
    $persona->nombre = "Robert";
    $persona->edad = 39;
    $persona->poblacion = "Sabadell";

    // preparo la respuesta, con el contenido en JSON y el header adecuado
    $response = new Response();
    $response->setContent(json_encode($persona));
    $response->headers->set('Content-Type', 'application/json');

    // retorno la respuesta
    return $response;
}
```

Ejemplo resultado





Ejemplo generando *JSON* con *JsonResponse*




```php
/**
 * @Route("/ejemplojson")
 */
public function ejemploJson():Response{

    // creo un objeto stdClass y le coloco algunos datos
    $persona = new \stdClass();  // la contrabarra es por el namespace
    $persona->nombre = "Robert";
    $persona->edad = 39;
    $persona->poblacion = "Sabadell";

    // retorna los datos en json
    return new JsonResponse($persona);
}
```

JSON y Doctrine ORM

- Cuando trabajamos con *Doctrine ORM*, la cosa es un poco más compleja, ya que debemos normalizar y serializar a *JSON* los objetos del tipo entidad.

- A modo de ejemplo, voy a mostrar cómo podemos hacer un método que retorne todas las películas de nuestro sitio en *JSON*.

- El primer paso será instalar el paquete *serializer* haciendo:

```
composer require symfony/serializer
```

Ejemplo instalando *serializer*

```
c:\xampp\htdocs\symfofilms>composer require symfony/serializer
./composer.json has been updated
Running composer update symfony/serializer
Loading composer repositories with package information
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
  - Locking symfony/serializer (v5.3.8)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Downloading symfony/serializer (v5.3.8)
  - Installing symfony/serializer (v5.3.8): Extracting archive
Generating optimized autoload files
composer/package-versions-deprecated: Generating version class...
composer/package-versions-deprecated: ...done generating version class
66 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
```

JSON y Doctrine ORM

- Para hacer la codificación a *JSON* necesitaremos al menos un codificador (*encoder*) y un normalizador (*normalizer*).
  - El **codificador** se encarga de convertir al formato especificado.
  - El **normalizador** se encarga de mapear los datos a objeto.
- Por defecto dispondremos de codificadores para *JSON*, *XML*, *CSV*, *YAML*, de los que nos interesa el de *JSON*.
- Respecto a los normalizadores, nos quedaremos con el *ObjectNormalizer*, que sirve para manejar objetos de datos.

JSON y Doctrine ORM

- Debemos recordar añadir los *namespaces* adecuados para usar correctamente el *serializador*.

```
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;
use Symfony\Component\Serializer\Serializer;
```

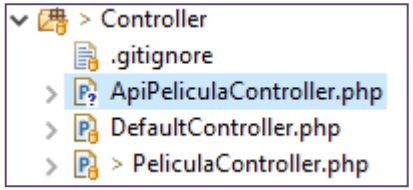
- Aunque, como hemos ido viendo, si los olvidamos *Symfony* nos lo sugerirá con un estupendo mensaje de error.

Attempted to load class "Serializer" from namespace "App\Controller".
Did you forget a "use" statement for "Symfony\Component\Serializer\Serializer"?

Ejemplo creando un nuevo controlador

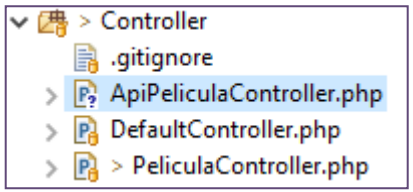

```
c:\xampp\htdocs\symfofilms>php bin/console make:controller

 Choose a name for your controller class (e.g. VictoriousGnomeController):
 > ApiPeliculaController

 created: src/Controller/ApiPeliculaController.php
 created: templates/api_pelicula/index.html.twig

  Success!

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

Ejemplo de entidades *Doctrine* a *JSON* (1/2)



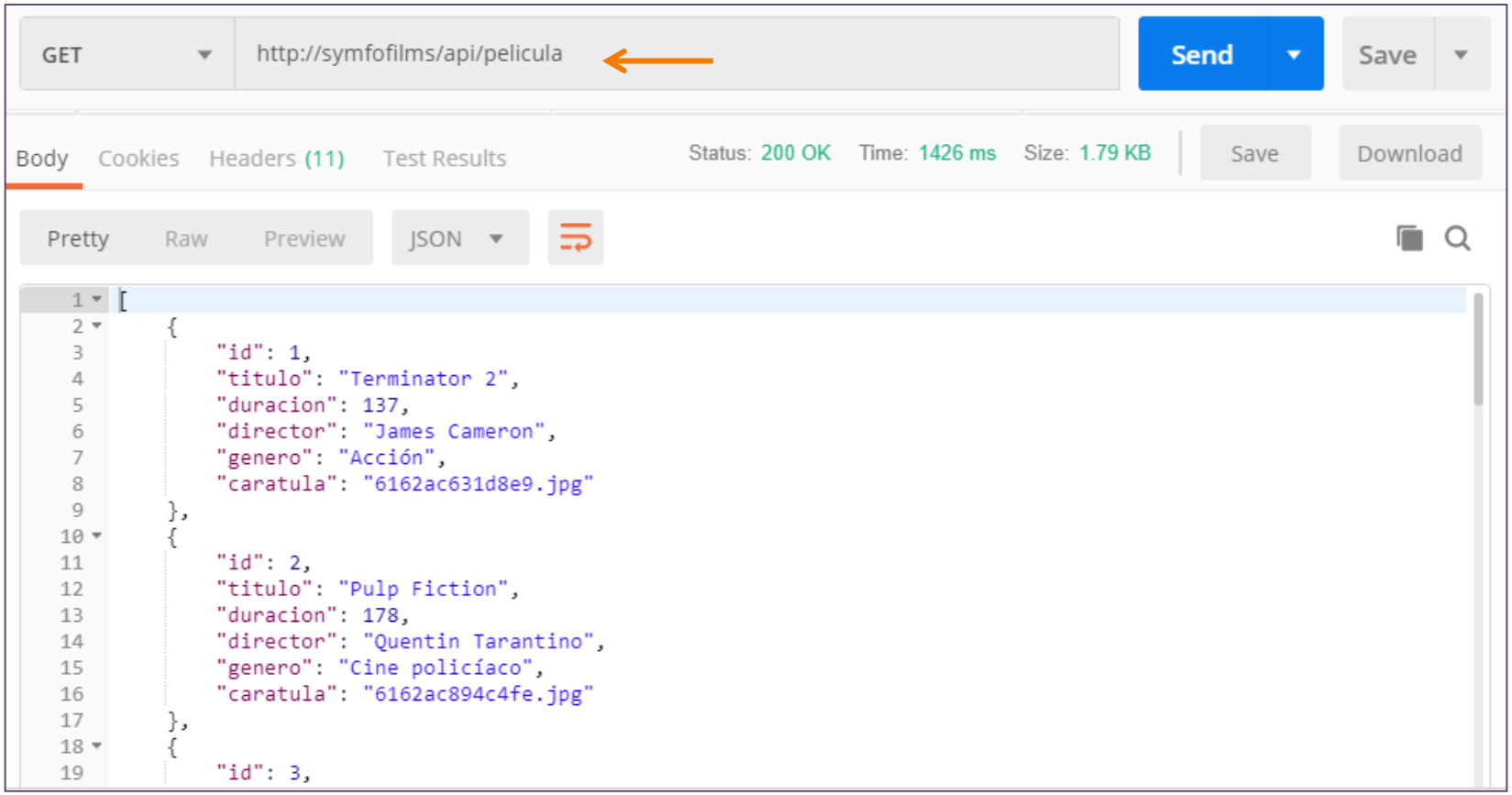

```php
namespace App\Controller;

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

use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;
use Symfony\Component\Serializer\Serializer;

use App\Entity\Pelicula;
```

Ejemplo de entidades *Doctrine* a *JSON* (2/2)




```php
class ApiPeliculaController extends AbstractController{

    /**
     * @Route("/api/pelicula", name="peliculas_json")
     */
    public function pelisJson():Response{
        // recuperar las pelis
        $pelis = $this->getDoctrine()->getRepository(Pelicula::class)->findAll();

        // preparar el serializador
        $serializer = new Serializer([new ObjectNormalizer()], [new JsonEncoder()]);

        // normalizar y serializar la respuesta en JSON
        $contenido = $serializer->serialize($pelis, 'json');

        // crear la respuesta y establecer el Content-Type
        $response = new Response($contenido);
        $response->headers->set('Content-Type', 'application/json');

        // retorna la respuesta en JSON con los resultados
        return $response;
    }
}
```

Ejemplo resultado



JSON - XML - CSV

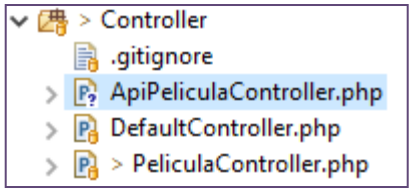
- Puede parecer un poco complicado esto de tener que usar el serializador y el nomalizador, pero tiene sus ventajas.

- Imaginad que quiero retornar la misma información en *JSON* o *XML* o *CSV*, dependiendo de lo que necesite el cliente.

  - Podemos mejorar el método anterior para que reciba por parámetro el formato y genere la respuesta de manera dinámica. Recordad los “use”:

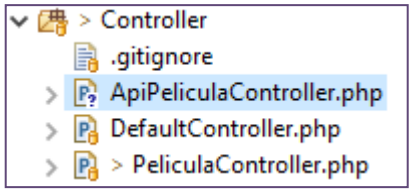
```
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Encoder\XmlEncoder;
use Symfony\Component\Serializer\Encoder\CsvEncoder;

use Symfony\Component\Serializer\Exception\NotEncodableValueException;
```

Ejemplo *JSON – XML – CSV (1/2)*



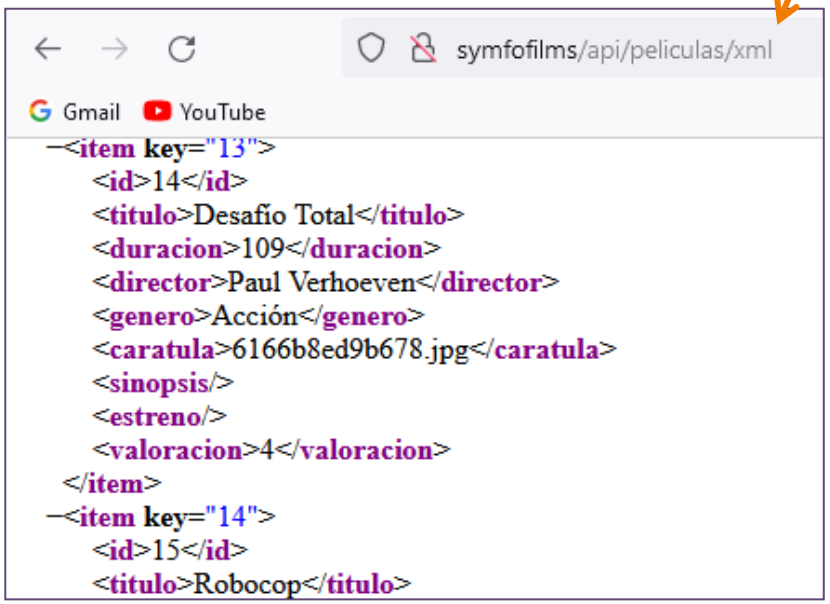

```php
/**
 * @Route("/api/peliculas/{formato}", name="api_peliculas")
 */
public function pelis(string $formato = 'json'):Response{
    // recuperar las pelis
    $pelis = $this->getDoctrine()->getRepository(Pelicula::class)->findAll();

    // preparar el serializador
    $serializer = new Serializer([new ObjectNormalizer()],
        [new JsonEncoder(), new XmlEncoder(), new CsvEncoder()]);

    // normalizar y serializar la respuesta
    $formato = strtolower($formato); // paso a minúsculas el formato

    try{
        $contenido = $serializer->serialize($pelis, $formato);
    }catch(NotEncodableValueException $e){
        return new Response('Formato no válido');
    }
```

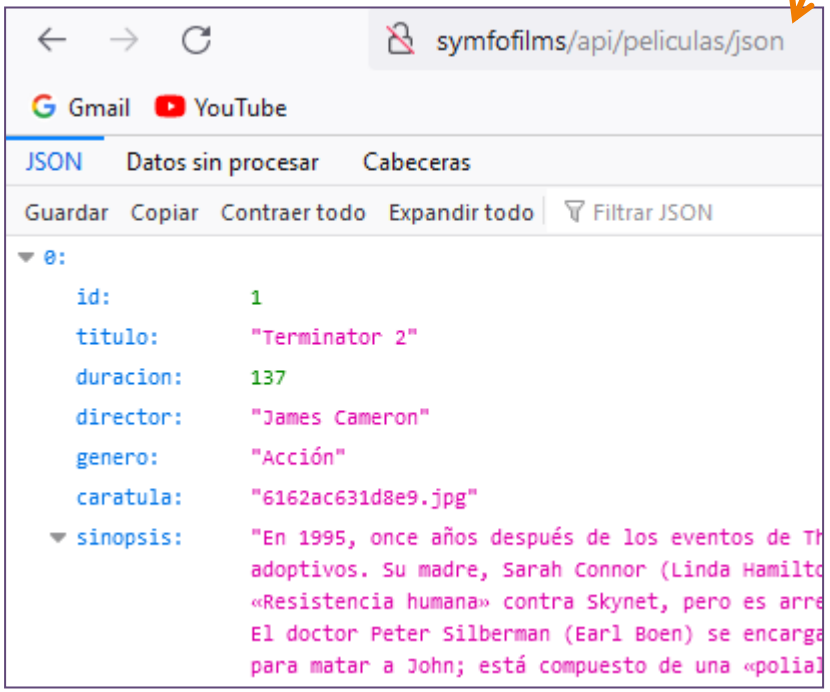
Ejemplo *JSON – XML – CSV (2/2)*



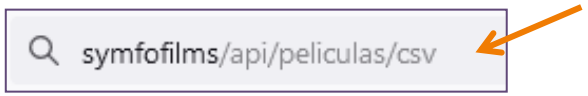

```
    // crear la respuesta y establecer el Content-Type
    $response = new Response($contenido);

    switch($formato){
        case 'json' : $formato = 'application/json'; break;
        case 'xml' : $formato = 'text/xml'; break;
        case 'csv' : $formato = 'text/csv'; break;
        default: $formato = 'text/plain';
    }

    $response->headers->set('Content-Type', $formato);

    // retorna la respuesta en con los resultados en el formato deseado
    return $response;
}
```

Ejemplo resultado



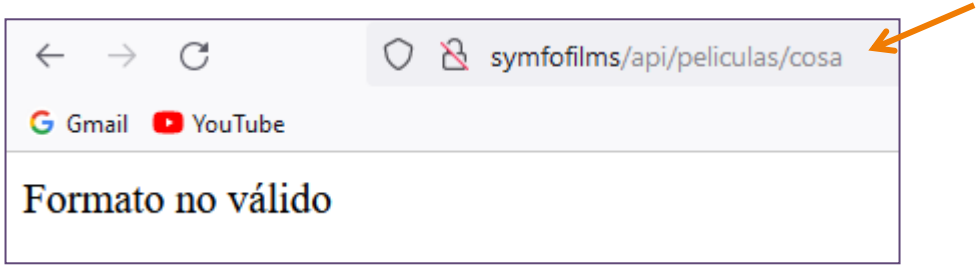


Ejemplo resultado




```
id,titulo,duracion,director,genero,caratula,sinopsis,estreno,valoracion
1,"Terminator 2",137,"James Cameron",Acción,6162ac631d8e9.jpg,"En 1995, once años después
Aunque no puede imitar máquinas complejas como pistolas o bombas, es capaz de moldear par
El T-1000 asume la identidad de un oficial de policía y persigue a John. Mientras tanto,
2,"Pulp Fiction",178,"Quentin Tarantino","Cine policíaco",6162ac894c4fe.jpg,,,
3,"Jurassick Park",127,"Steven Spielberg",Acción,6162ac9829f29.jpg,,,
4,"Amores perros",154,"Alejandro González Iñárritu",Drama,6162ad066ab14.jpg,,,
5,"El Hoyo",94,"Galder Gaztelu-Urrutia",Suspense,6162ad1476e79.jpg,,,
6,Avatar,162,"James Cameron",Acción,6162ad23c97ef.jpg,,,
7,"Regreso al Futuro",116,"Robert Zemeckis",Comedia,6162ad365e298.jpg,,,
8,"Blade Runner",117,"Ridley Scott",Acción,6162ad4676557.jpg,,,
9,Batman,126,"Tim Burton",Acción,6162ad55e74f7.jpg,,,
10,"El juego del calamar",1,"Hwang Dong-hyuk",Acción,6162be9b71321.jpg,,,
11,Cube,90,"Vincenzo Natali",Terror,6162acf36d84b.jpg,,,
12,Titanic,167,"James Cameron",Drama,6164681717dc0.jpg,,,
13,"Los Goonies",114,"Richard Donner",Aventuras,6166bc8a0936a.jpg,,,
14,"Desafío Total",109,"Paul Verhoeven",Acción,6166b8ed9b678.jpg,,,4
15,Robocop,102,"Paul Verhoeven",Acción,616950ad682f6.jpg,"En un futuro cercano distópico,
```

Ejemplo resultado



Cookies

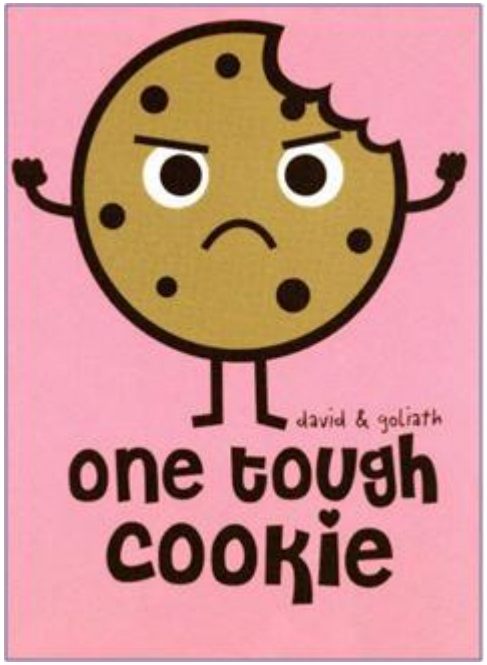
Añadiendo *cookies* a la respuesta

Robert Sallent
www.robertsallent.com

Cookies

- Podemos **anexar *cookies*** a la respuesta, de la misma forma que podremos recuperar información de las *cookies* que vienen con la petición.

- En este punto, sería interesante repasar los apuntes sobre *cookies* y su manipulación con *PHP* que os comparto en el *Drive*.



Cookies

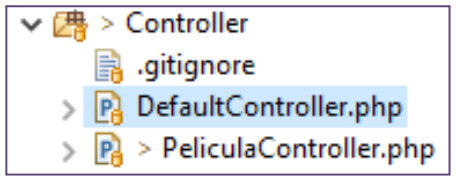
- Las *cookies* se pueden manipular en *Symfony* mediante la propiedad *headers* del objeto *Response*.

```
$response->headers->setCookie(
    Cookie::create('autor', 'Robert')
);
```

- El método `setCookie()` recibe un objeto de la clase *Symfony\Component\HttpFoundation\Cookie* como parámetro.

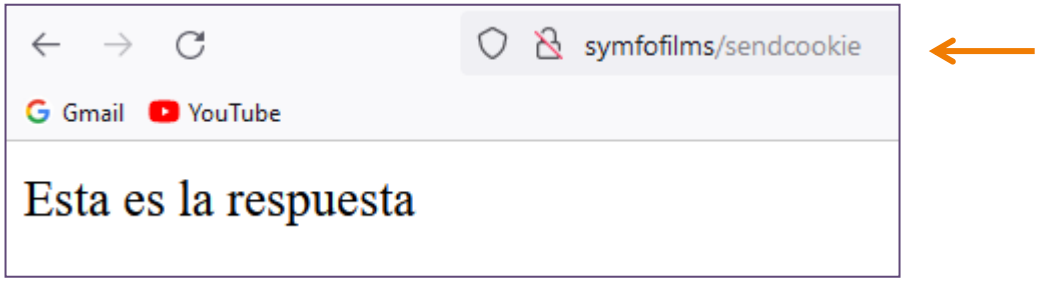
- El método estático `Cookie::create()` nos permite crear la cookie con las características deseadas.

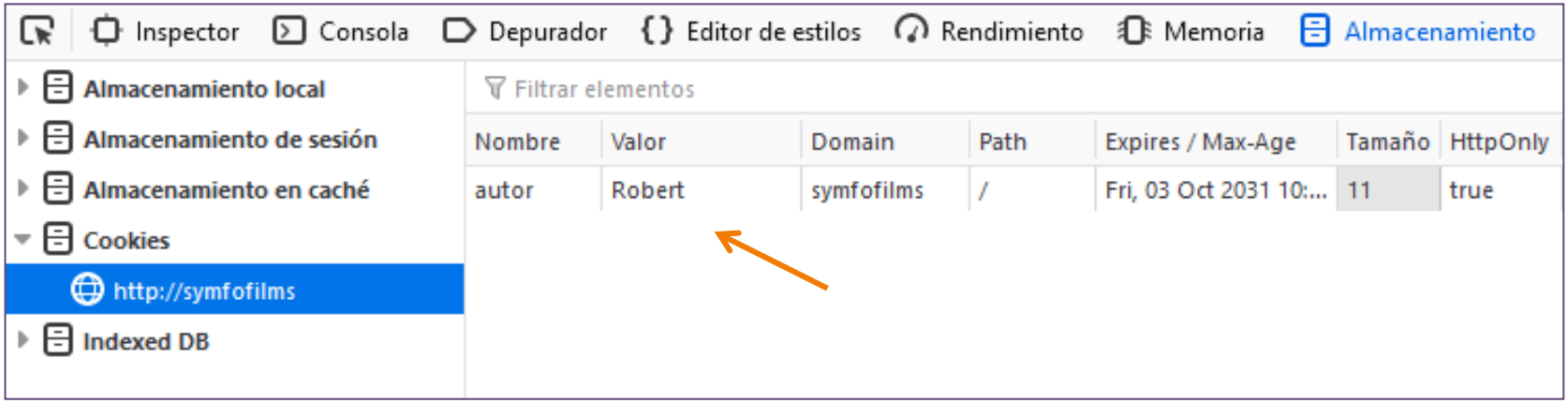
Ejemplo anexando cookies




```php
/**
 * @Route("/sendcookie")
 */
public function sendCookie():Response{

    $cookie = Cookie::create('autor')
        ->withValue('Robert')
        ->withExpires(strtotime('Fri, 01-Oct-2031 10:00:00 GMT'))
        ->withSecure(false);

    // crea una respuesta
    $response = new Response(
            'Esta es la respuesta',          // contenido
            Response::HTTP_OK,               // 200 (OK)
            ['content-type' => 'text/html']  // encabezados
        );

    $response->headers->setCookie($cookie);

    return $response;
}
```

Ejemplo resultado





Sirviendo ficheros

Añadiendo ficheros a la respuesta

Robert Sallent
www.robertsallent.com

SOC Servei d'Ocupació de Catalunya
CIFO Sabadell- Mancomunitat (Vallès)
Generalitat de Catalunya
Unió Europea
Fons social europeu
L'FSE inverteix en el teu futur

Sirviendo ficheros

- A veces, la respuesta a una petición resultará en el envío de un fichero.

- No hay ningún problema en calcular la ruta o el nombre del fichero de forma dinámica en función de la forma de la petición. También existe la posibilidad de eliminar el fichero del servidor una vez enviado.

- Si queremos hacer uso de la determinación automática de tipos mime (con lo que no nos hará falta indicarlo nosotros, debemos instalar el paquete *mime* con:

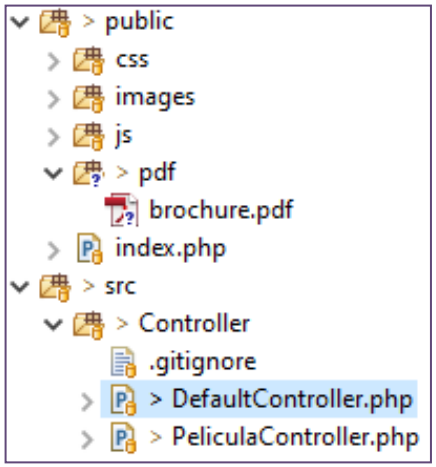
```
composer require symfony/mime
```

Ejemplo sirviendo ficheros

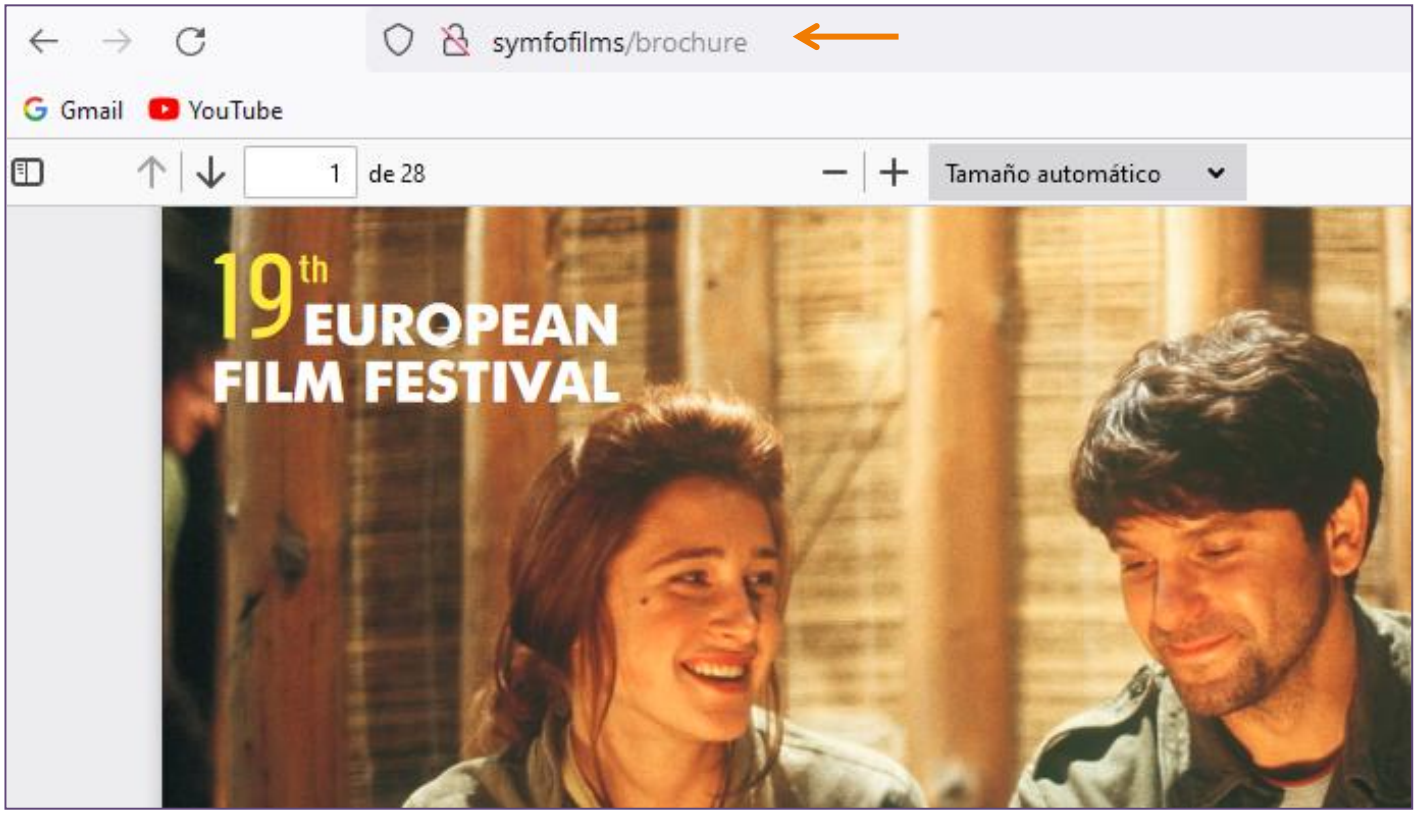

```
c:\xampp\htdocs\symfofilms>composer require symfony/mime
./composer.json has been updated
Running composer update symfony/mime
Loading composer repositories with package information
Updating dependencies
Lock file operations: 2 installs, 0 updates, 0 removals
  - Locking symfony/mime (v5.3.8)
  - Locking symfony/polyfill-intl-idn (v1.23.0)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Downloading symfony/mime (v5.3.8)
  - Installing symfony/polyfill-intl-idn (v1.23.0): Extracting archive
```

Ejemplo sirviendo ficheros

```php
use Symfony\Component\HttpFoundation\BinaryFileResponse;


class DefaultController extends AbstractController{
     * Método index...
    public function index(): Response{
        return $this->render("portada.html.twig");
    }

    /**
     * @Route("/brochure")
     */
    public function brochure():Response{
        return new BinaryFileResponse('pdf/brochure.pdf');
    }
```



Ejemplo resultado



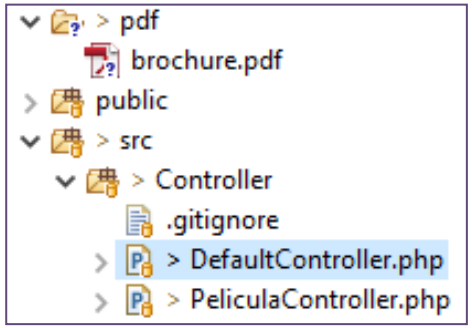
Sirviendo ficheros

- En el ejemplo anterior, hemos colocado el fichero en una carpeta pública, con lo que sería perfectamente accesible mediante la *URL*.
- A veces querremos hacer que nuestros ficheros no sean accesibles por *URL*. Para ello, los debemos colocar fuera de la carpeta *public*.
- En el siguiente ejemplo, he movido el directorio *pdf* a la carpeta raíz y he variado la forma de recuperar el fichero.
  - De este modo, no se podrá acceder directamente al fichero mediante una *URL*, comprobadlo, a ver si podéis ;)

Ejemplo sirviendo ficheros no en carpeta *public*

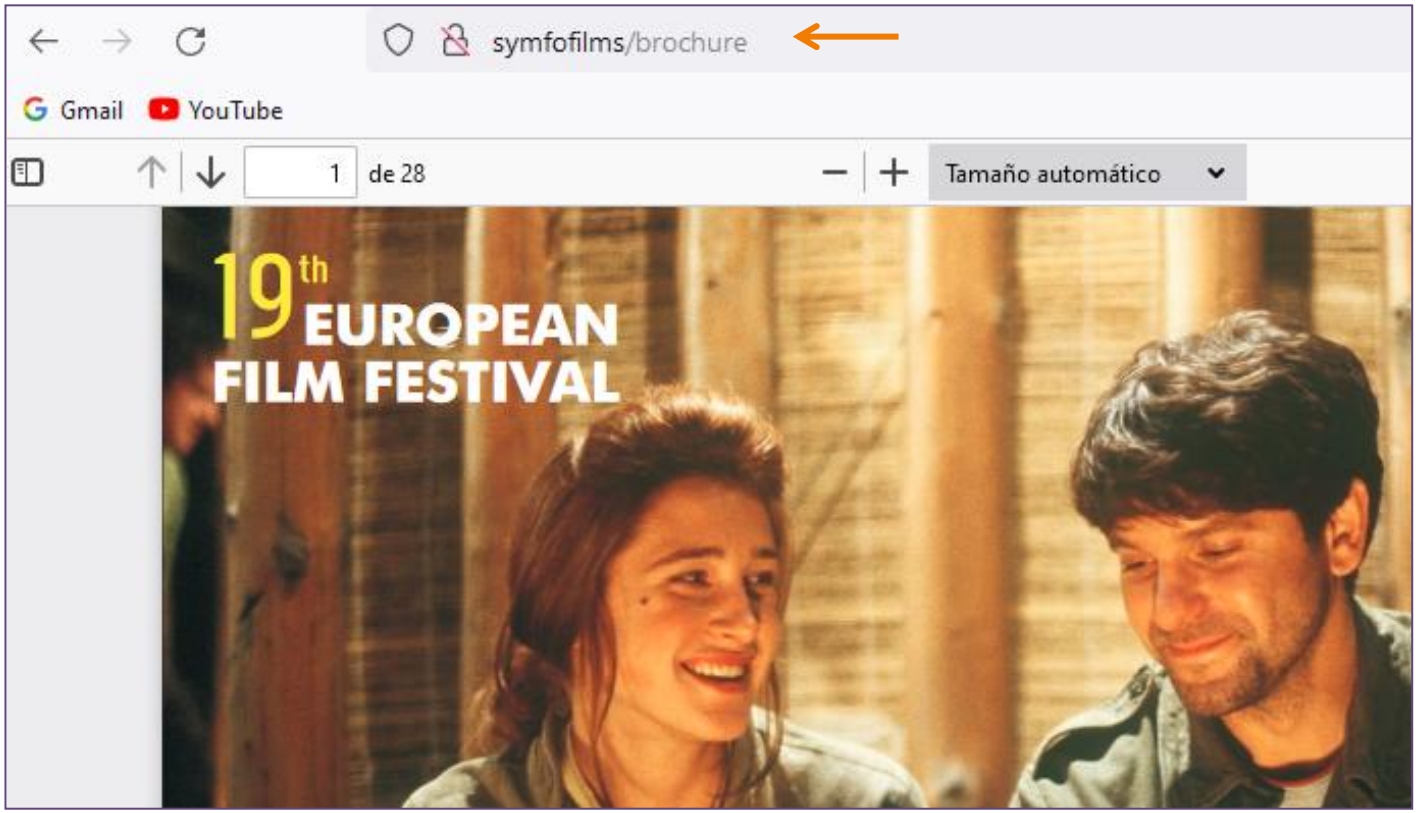
```php
/**
 * @Route("/brochure")
 */
public function brochure():Response{
    return new BinaryFileResponse(__DIR__.'/../../pdf/brochure.pdf');
}
```

O bien…




```php
/**
 * @Route("/brochure")
 */
public function brochure(Kernel $kernel):Response{

    // recupera la raíz del proyecto
    $raiz = $kernel->getProjectDir();

    // calcula la ruta a partir de la raíz
    return new BinaryFileResponse($raiz.'/pdf/brochure.pdf');
}
```

Ejemplo resultado



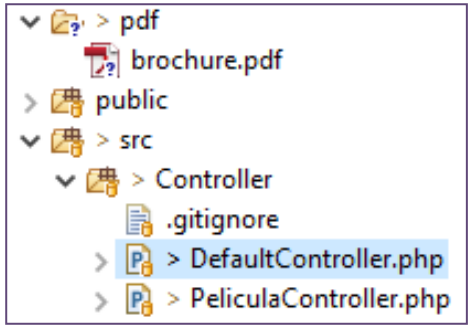
Sirviendo ficheros

- Como hemos hecho uso del paquete *mime*, no hemos tenido que indicar el tipo del fichero.

- En el caso de no haberlo usado, deberíamos indicar el tipo *mime* en los encabezados de la respuesta.

- Si el documento servido es un fichero que el navegador es capaz de abrir, lo abrirá. En caso contrario mostrará el cuadro de diálogo para descargarlo.

Ejemplo indicando el tipo mime




```php
/**
 * @Route("/brochure")
 */
public function brochure(Kernel $kernel):Response{

    // recupera la raíz del proyecto
    $raiz = $kernel->getProjectDir();

    // calcula la ruta a partir de la raíz y prepara la respuesta
    $response =  new BinaryFileResponse($raiz.'/pdf/brochure.pdf');

    // establece el tipo mime a application/pdf
    $response->headers->set('Content-Type', 'application/pdf');

    // envía la respuesta
    return $response;
}
```

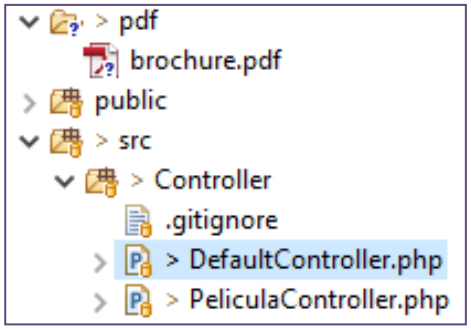
Sirviendo ficheros

- Si a la hora de enviar el fichero, queremos obligar a su descarga, en lugar de que se muestre en el navegador, podemos cambiar el *Content-Disposition*.

- Los posibles valores para el *Content-Disposition* son: ***inline*** (valor predeterminado, indicando que puede ser mostrado dentro de una página web, o como la página web) o ***attachment*** (indicando que será descargado mostrando un diálogo *'Guardar como'*).

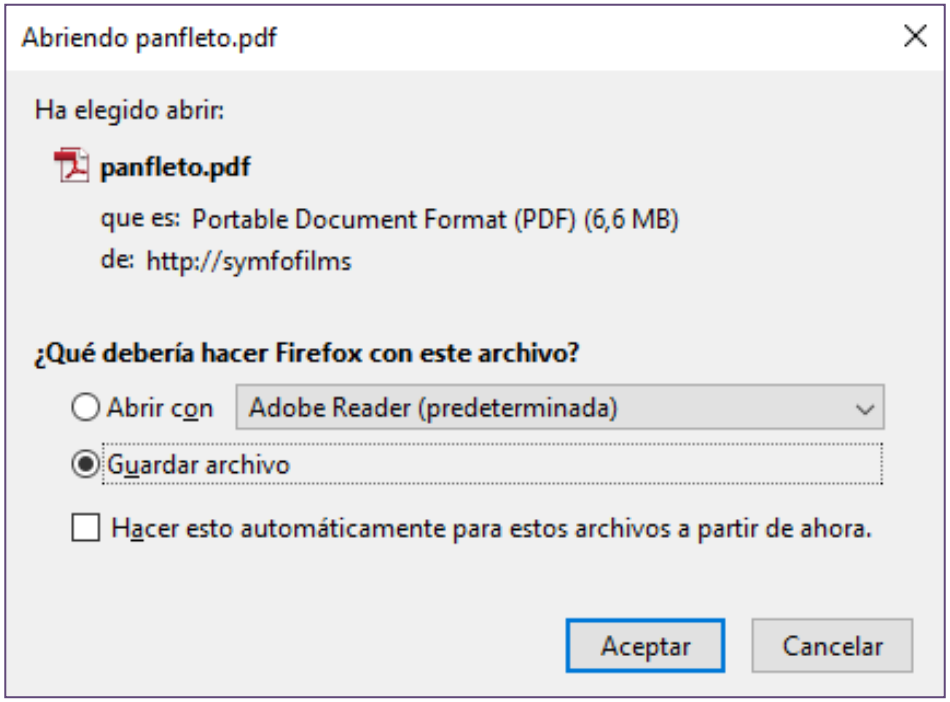
- Podemos cambiar el nombre al fichero al descargarlo.

Ejemplo obligando a la descarga del fichero




```php
/**
 * @Route("/brochure")
 */
public function brochure(Kernel $kernel):Response{

    // recupera la raíz del proyecto
    $raiz = $kernel->getProjectDir();

    // calcula la ruta a partir de la raíz y prepara la respuesta
    $response =  new BinaryFileResponse($raiz.'/pdf/brochure.pdf');

    // establece el tipo mime a application/pdf
    $response->headers->set('Content-Type', 'application/pdf');

    // hará que el fichero se descarge, en lugar de mostrarse
    $response->setContentDisposition(
        ResponseHeaderBag::DISPOSITION_ATTACHMENT, 'panfleto.pdf');

    // envía la respuesta
    return $response;
}
```

Ejemplo resultado



Borrar tras servir

- Si deseamos que el fichero se borre del servidor una vez descargado, cosa que viene bien para ficheros que se han generado de forma automatizada y que no queremos que se mantengan en el servidor, podemos usar el método *deleteFileAfterSend()*.

```
// borrará el fichero tras ser enviado
$response->deleteFileAfterSend();
```

- En nuestro caso, si lo hacéis veréis que el folleto se borrará y la siguiente vez que lo intentéis nos dará un error de fichero no encontrado.