

CAPÍTULO 2

CONTROLADORES

Contenidos

- Mecanismos de ejecución de código en un servidor web
- Generación dinámica de páginas web
- Lenguajes de programación en entorno servidor
- Integración con los lenguajes de marcas
- Lenguajes embebidos en HTML
- Etiquetas para inserción de código
- Mantenimiento del estado. Sesiones
- Mecanismos de separación de la lógica de negocio. Modelo Vista Controlador

Objetivos

- Valorar las ventajas que proporciona la separación en módulos con distintas responsabilidades
- Reconocer los mecanismos de generación de páginas web a partir de lenguajes de marcas con código embebido
- Utilizar etiquetas para la inclusión de código en el lenguaje de marcas
- Identificar los mecanismos disponibles para el mantenimiento de la información que concierne a un cliente web concreto y se han señalado sus ventajas
- Utilizar sesiones para mantener el estado de las aplicaciones web

RESUMEN DEL CAPÍTULO

En este capítulo estudiaremos el Modelo Vista Controlador, su estructura, componentes e implementación en Symfony. Así mismo analizaremos la manera en la que las distintas partes interactúan para crear aplicaciones web mantencibles y escalables.

2.1. INTRODUCCIÓN

Un controlador, en el patrón MVC, tiene como rol comunicar la vista con el modelo, de manera que recibirá las peticiones fruto de la interacción del usuario con la vista, ejecutará la lógica necesaria para resolver la petición, y enviará el modelo de nuevo a la vista con la información que se necesite presentar de vuelta al usuario.

En *Symfony* un controlador es una clase PHP que contendrá acciones, cada una de las cuales podrá ser llamada mediante una URL única.

Ilustremos esto con un ejemplo. A continuación vamos a crear una sencilla aplicación de predicción de tiempo. Para ello vamos a utilizar *Netbeans* (disponible a partir de la versión 8.2) en lugar de la interfaz de comandos de *Composer* para crear la estructura del proyecto. Es necesario descargar el paquete *Symfony Standard Edition* en zip. En el momento de la redacción de este libro, la última versión estable era la 3.1. Para descargarlo puedes dirigirte al siguiente enlace:

http://mirror.nienbo.com/symfony/3.1/Symfony_Standard_3.1.3.zip

Una vez descargado, lo guardamos en:

C:\Program Files\NetBeans 8.2\php

Ahora debemos indicarle a Netbeans dónde hemos almacenado este paquete para que lo pueda utilizar para crear el nuevo proyecto. Para ello debemos ir a Tools > Options > PHP > Frameworks & Options Symfony2/3. Elegimos la opción Sandbox e introducimos la ruta anterior donde guardamos el empaquetado

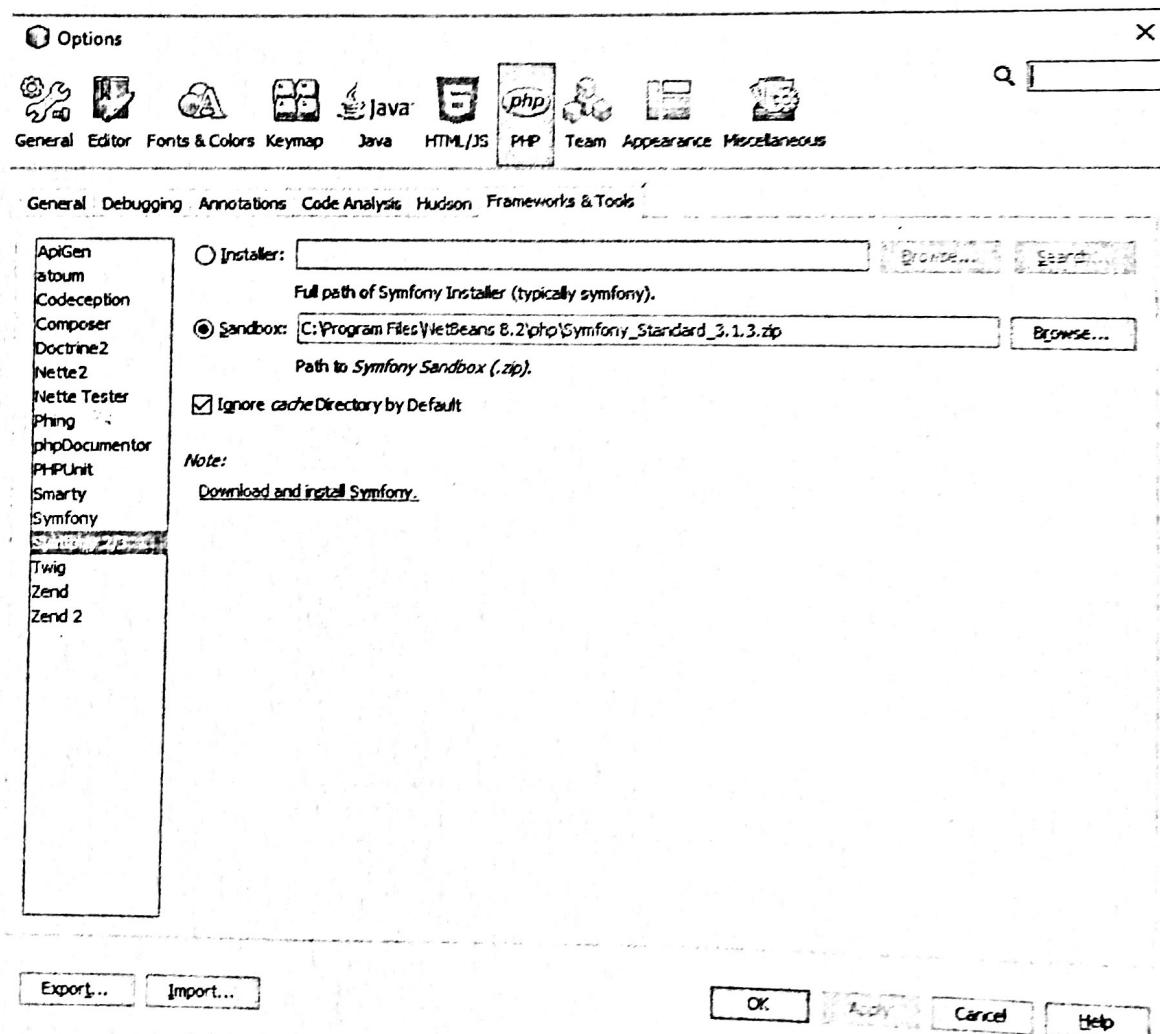


Figura 2.1. Configuración de Symfony3 en Netbeans

Ahora ya podemos crear proyectos con Symfony3 desde nuestro Netbeans. No obstante, ten en cuenta que cuando lo hagas, la carpeta vendor no estará creada, y por tanto, la aplicación no será operativa, pues le faltarán todas las librerías necesarias. Netbeans creará la estructura del proyecto y especificará en el archivo *composer.json* qué librerías con sus versiones utiliza el proyecto. Para que se descarguen debes ejecutar un composer install. Esto también lo puedes hacer desde la propia interfaz de Netbeans haciendo clic con el botón derecho del ratón en el proyecto y accediendo al menú de opciones de Composer:

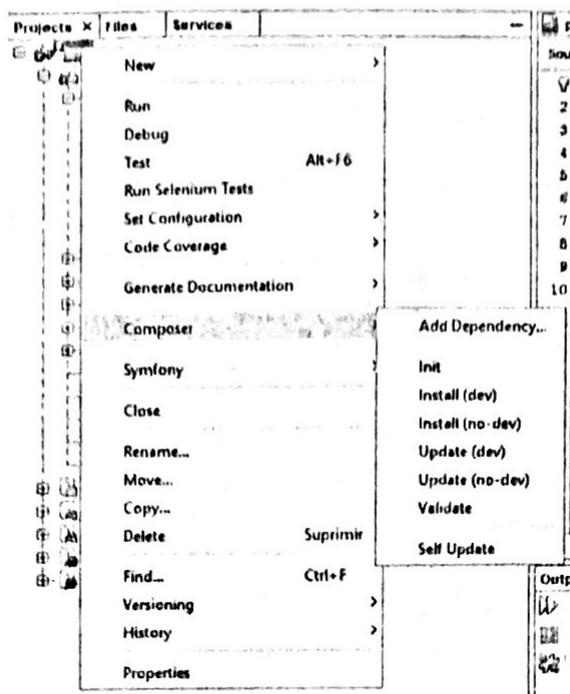


Figura 2.2. Ejecución de composer install desde la interfaz de Netbeans

2.2. ACCIONES SIN PARÁMETROS

Comenzaremos creando un nuevo proyecto llamado *forecast*:

```
composer create-project symfony/framework-standard-edition forecast
```

Seguidamente añadimos una primera versión de un controlador muy sencillo que simplemente devuelva un mensaje a mostrar en el navegador. Para ello, creamos un archivo *ForecastController* en la ruta *app > src > AppBundle > Controller*. El contenido será el siguiente:

```
<?php

namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

/**
 * Simple example controller.
 * @author Eugenia Pérez <info@eugeniaperez.es>
 */
class ForecastController extends Controller {
```

16 Symfony. Desarrollo Web en Entorno Servidor

```
/*
 * Parameterless index action.
 * @return Response
 */
public function indexAction() {
    return new Response('It\'s freezing cold');
}

}
```

Como se puede ver se crea un objeto *Response* con un mensaje, el cual podrás visualizar yendo a la siguiente dirección en tu navegador:

http://localhost/forecast/web/app_dev.php/forecast

Obviamente, para que funcione debes haber configurado la ruta apropiada en el archivo *app > config/routing.yml*:

```
app:
    path: "/forecast"
    defaults: {_controller: AppBundle:Forecast:index}
```

El controlador *ForecastController* anterior tiene una acción *indexAction* que no recibe ningún parámetro. Ten en cuenta que todos los controladores llevarán el sufijo *Controller* y todas las acciones el sufijo *Action*, además de utilizar notación *camel case*.

2.3. ACCIONES CON PARÁMETROS

Veamos ahora una versión similar en el que la acción recibe un parámetro y lo retorna al cliente:

```
/**
 * Action that receives a parameter with the weather info and prints out a
 * message with it.
 * @return Response
 */
public function indexParamAction($weather) {
    return new Response('<html><body>Weather info: It\'s ' . $weather .
        '</body></html>');
}
```

Para que podamos acceder por URL desde el navegador a esta acción, debemos modificar el fichero *routing.yml* de la siguiente manera:

```
forecast_simple:
    path: "/forecast/index"
    defaults: {_controller: AppBundle:Forecast:index}

forecast_param:
    path: "/forecast/{weather}"
    defaults: {_controller: AppBundle:Forecast:indexParam}
```

Como se puede ver, se indican dos rutas distintas. Es necesario que reciban nombres únicos. Además, la segunda especifica la capacidad que tiene para pasar un parámetro *weather* a la acción de destino *indexParam*. Prueba a ejecutar la acción recién creada a través de la siguiente URL:

http://localhost/forecast/web/app_dev.php/forecast/sunny

En este caso, el parámetro *weather* recibirá el valor *sunny*.

Podríamos modificar el archivo de *routing* anterior para especificar un valor por defecto del parámetro *weather*, para que en caso de que el usuario lo omita en su petición, este tome algún valor.

```
forecast_simple:
    path: "/forecast/index"
    defaults: {_controller: AppBundle:Forecast:index}

forecast_param:
    path: "/forecast/{weather}"
    defaults: {_controller: AppBundle:Forecast:indexParam, weather: cloudy}
```

Para que funcione correctamente, debes eliminar la ruta app añadida anteriormente, ya que en caso contrario la petición */forecast* entrará por dicha ruta, y no por la que acabamos de añadir con el parámetro por defecto.

En la siguiente tabla se puede ver qué ruta manejaría cada una de las URLs listadas:

Figura 2.3. Rutas para URLs

URL	Ruta
http://localhost/forecast/web/app_dev.php/forecast	forecast_param
http://localhost/forecast/web/app_dev.php/forecast/index	forecast_simple
http://localhost/forecast/web/app_dev.php/forecast/sunny	forecast_param

ACTIVIDAD 2.1

Crea un proyecto en el que haya un controlador *GreetController* con acción *greetSomethingAction* para que ante una petición */greet/{something}*, el controlador sea capaz de responder con ese *{something}* en cursiva. Impleméntalo primero sin necesidad de añadir una vista, para posteriormente crear una nueva vista asociada a dicho controlador que muestre lo que este le envía.

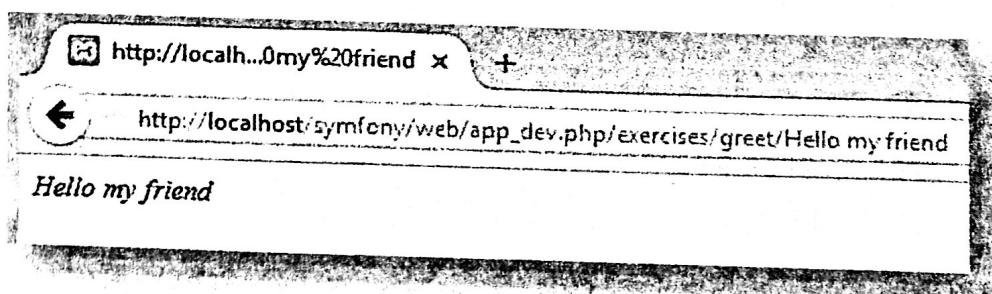


Figura 2.4. Resultado de la actividad

De manera similar al ejemplo anterior, podríamos crear acciones que reciban más de un parámetro. De esta manera, vamos a modificar la ruta anterior para que pueda recibir la predicción del tiempo y la temperatura en grados centígrados.

```
forecast_2_params:
    path: "/forecast/{weather}/{temperature}"
    defaults: {_controller: AppBundle:Forecast:index2Params, weather: cloudy,
```

```
temperature: 15} ;
```

La ruta anterior haría que URLs como la siguiente fuesen dirigidas a la acción *index2ParamsAction*.

http://localhost/forecast/web/app_dev.php/forecast/sunny/25

```
/**  
 * Action that receives 2 parameters with the weather info and the  
 * temperature  
 * and prints out a message with the information.  
 * @return Response  
 */  
public function index2ParamsAction($weather, $temperature) {  
    return new Response('<html><body>Weather info: It\'s ' . $weather .  
        ' and the current temperature is: ' . $temperature .  
        ' °C</body></html>');  
}
```

El mapeo de los parámetros de la acción se realiza por nombre, por lo que el orden no es importante. Por tanto, todo funcionaría igual si la firma de la acción fuese la siguiente:

```
public function index2ParamsAction($temperature, $weather)
```

En ocasiones, puede suceder que la recepción de parámetros de la manera indicada no sea suficiente, necesitemos leer el valor de parámetros pasados de la manera tradicional:

http://dominio.com?param1=value

Para acceder a este tipo de parámetros podemos utilizar el objeto Request, que nos permite leer parámetros recibidos mediante cualquier método HTTP. Así, completaremos la acción anterior para que reciba el nombre de la ciudad de la que se está imprimiendo la predicción del tiempo en la *query string*:

http://localhost/forecast/web/app_dev.php/forecast/indexRequest/sunny/25?city=Oviedo

Añadimos la siguiente ruta para tratar URLs como la anterior:

```
forecast_request_param:  
    path: "/forecast/indexRequest/{weather}/{temperature}"  
    defaults: {_controller: AppBundle:Forecast:indexRequest, weather: cloudy,  
        temperature: 15}
```

Y la acción correspondiente podría ser:

```
/**  
 * Action that receives 2 parameters with the weather info and the  
 * temperature and prints out a message with the information.  
 * @return Response  
 */  
public function indexRequestAction($weather, $temperature,  
    Request $request) {  
    return new Response('<html><body>Weather info in ' .  
        $request->query->get("city") .  
        ': It\'s ' . $weather . ' and the current temperature is: ' .  
        $temperature . ' °C</body></html>');  
}
```

ACTIVIDAD 2.2

Desarrolla una página que reciba dos parámetros numéricos y muestre la suma de ellos. El *path* de *routing* debe tener este aspecto:

```
/add/{number1}/{number2}
```

Debes definir valores por defecto para los parámetros en la ruta de tal forma que si en la petición no se pasan valores se utilicen los valores por defecto. Desarrolla un controlador llamado *CalculatorController* que recoja los valores y muestre la suma.

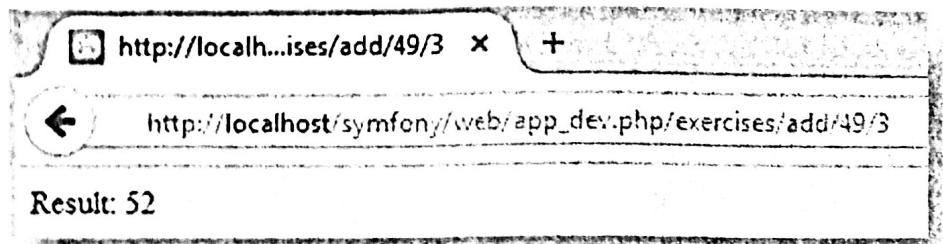


Figura 2.5. Resultado de la actividad

ACTIVIDAD 2.3

Completa el ejercicio anterior añadiendo otras tres rutas para las operaciones:

```
/sub/{number1}/{number2}
/mul/{number1}/{number2}
/div/{number1}/{number2}
```

Utiliza el mismo controlador del Ejercicio 2.

2.4. LA CLASE BASE CONTROLLER

Symfony pone a disposición de los desarrolladores una clase *Controller* de la cual podemos heredar al crear nuestros propios controladores. Al hacerlo, esto nos dará acceso a una serie de *helpers* útiles en diversos escenarios, evitando la repetición de código. Algunas de las funcionalidades a las que nos darán acceso la clase *Controller* son:

Redirección

A través del método *redirectToRoute()* podemos redirigir al usuario a otra página de nuestro sitio. Probemos a implementar un nuevo controlador, *RedirectController* con una acción que implementa tal redirección:

```
class RedirectController extends Controller {
    /**
     * Internal redirection example. By default a 302 (temporary) redirection
     * is performed.
     * @return Response
     */
    public function internalRedirectAction() {
        return $this->redirectToRoute("redirect_index");
    }
}
```

La acción anterior está haciendo una redirección temporal (código HTTP 302) a la ruta `redirect_index`. Este nombre de ruta debe existir en nuestro fichero de rutas.

```
redirect_index:
    path: "/redirect/index"
    defaults: {_controller: AppBundle:Redirect:index}
```

En caso de que quisiésemos implementar una redirección permanente (301), podemos indicar los patrones necesarios al método `redirectToRoute()`.

```
/**
 * Internal redirection example generating a 301 (permanent) redirection.
 * @return Response
 */
public function internalRedirectPermanentAction() {
    return $this->redirectToRoute("redirect_index", array(), 301);
}
```

En caso de querer redireccionar a una página externa, podemos utilizar el método `redirect()`.

```
/**
 * External redirection example.
 * @return Response
 */
public function externalRedirectAction() {
    return $this->redirect("http://eugeniaperez.es");
}
```

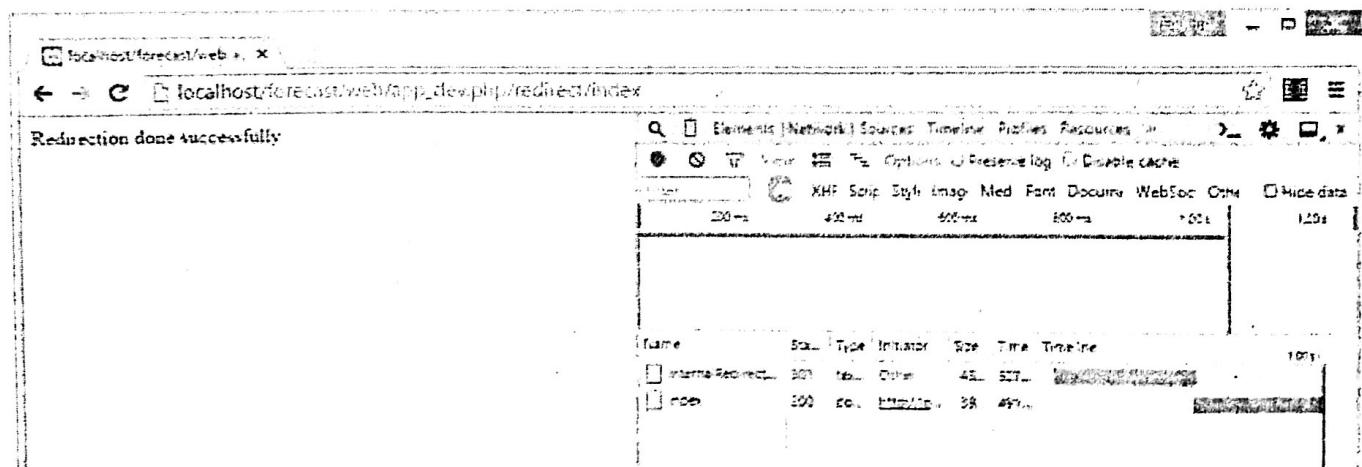


Figura 2.6. Redirección permanente (301)

2.5. MANEJO DE ERRORES

Cuando algo anómalo sucede dentro de una acción de un controlador se suelen utilizar excepciones para gestionar los errores y actuar en consecuencia. Cuando se lanza una excepción, por lo general se traduce una respuesta al cliente con un código HTTP 500. Un tipo de error particular se produce cuando el cliente pide un recurso que no se encuentra. Esto se suele tratar como un 404 *Not Found*. Para generar este tipo de error desde una acción podemos hacer uso de la clase `NotFoundHttpException`.

```
/**
 * Throws a 404 exception if the $id does not match the criteria.
 * @return Response
*/
```

```

    */
public function getAction($id) {
    if ($id == null || $id != 1) {
        throw $this->createNotFoundException('The register requested does not
exist');
    } else {
        return new Response("Register " . $id);
    }
}

```

La ruta necesaria para que esta acción esté accesible sería:

```
get_temp:
path: "/temperature/{id}"
defaults: {_controller: AppBundle:Temperature:get}
```

Si ahora intentásemos acceder por ejemplo a la siguiente URL, obtendríamos una respuesta de error 404:

http://localhost/forecast/web/app_dev.php/temperature/5

ACTIVIDAD 2.4

Crea una nueva vista a la que se acceda a través de una acción que reciba un parámetro denominado *username*. Si este es distinto de admin la acción devolverá una excepción de Symfony de tipo *AccessDeniedHttpException*.

2.6. MANEJO DE LA SESIÓN

Dado que *Symfony* es un *framework* construido sobre PHP, y este permite a los desarrolladores acceder al objeto *Session* para guardar y recuperar información, *Symfony* también permite el acceso a dicho recurso. El objeto sesión almacena la información en el servidor y se crea una instancia por cada usuario que visita el sitio Web, por lo que no conviene saturarlo.

Para almacenar y recuperar información de la sesión basta con utilizar los métodos *set()* y *get()*. Por ejemplo, a continuación vamos a crear una acción que cada vez que sea accedida guardará en sesión un log de la fecha a la que ha sido ejecutada.

```
/**
 * Creates a log every time is called.
 * @return Response
 */
public function checkAction(Request $request) {
    $session = $request->getSession();
    $log = $session->get('log', array());

    array_push($log, 'Temperature checked at ' .
        date('l jS \of F Y h:i:s A'));
    $session->set('log', $log);

    return new Response(
        'Temperature checked and log persisted successfully');
}
```

22 Symfony. Desarrollo Web en Entorno Servidor

Como se puede ver, se recupera de sesión el objeto con clave *log*. En caso de que no existiera, se retorna un *array* vacío. Después, con el método *array_push()* de PHP se inserta un nuevo registro de *log*.

Nota cómo a la hora de definir la ruta, podría darse una ambigüedad con respecto a la anterior ruta que obtenía una temperatura para un id en concreto:

```
check_temp:  
    path: "/temperature/check"  
    defaults: {_controller: AppBundle:Temperature:check}
```

Es por ello que deberíamos obligar a que todo id que enviamos como parámetro en la ruta anterior tenga que ser necesariamente numérico. Este requerimiento lo podemos especificar de la siguiente manera:

```
requirements:  
    id: \d+
```

Una vez haya varios registros almacenados, podemos visualizarlos mediante otra acción que acceda a sesión e imprima el contenido del objeto *log*.

```
/**  
 * Returns a list of all the logs stored in the Session.  
 * @return Response  
 */  
public function getAllAction(Request $request) {  
    $session = $request->getSession();  
    $log = $session->get('log', array());  
    $result = '';  
  
    foreach ($log as $item) {  
        $result .= $item . '<br />';  
    }  
  
    return new Response($result);  
}
```

El resultado de la ejecución podría ser el siguiente.

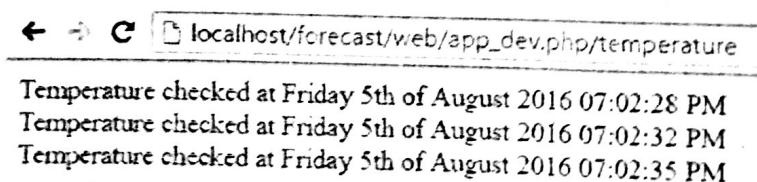


Figura 2.7. Resultado de consultar el listado de logs de la sesión

2.6.1. Mensajes entre pantallas

Tradicionalmente, la sesión se utilizaba para, entre otras cosas, pasar información de una página a otra que era necesaria en la página destino. Esto tiene el inconveniente de que, a fin de no sobrecargar la memoria del servidor, había que borrar los objetos de sesión a medida que ya no fueran necesarios, generando código repetitivo y obligando al desarrollador a estar pendiente de realizar esta tarea. No obstante, en muchas ocasiones hay datos que solo queremos pasar de una página a otra, pero que no queremos que se persista de manera indefinida. Para ello existen los mensajes *flash*, cuyo tiempo de vida es de solo una petición.

Veamos un ejemplo en el que se pasa el típico mensaje de confirmación de una página a otra. El mensaje será generado desde una acción de un controlador y posteriormente leído desde una vista.

```

/**
 * Adds a flash message to be rendered in a view.
 */
public function indexAction() {
    $this->addFlash('location', 'You are located in Oviedo, Spain');

    return $this->render('location/index.html.twig');
}

```

A continuación mostrarlo sería así de sencillo:

```

{% extends 'base.html.twig' %}

{% block body %}
    {% for flashMessage in app.session.flashbag.get('location') %}
        <div class="flash-notice">
            {{ flashMessage }}
        </div>
    {% endfor %}
{% endblock %}

```

2.7. REQUEST Y RESPONSE

Toda acción de un controlador debe retornar un objeto de tipo *Response*. Esta clase es una abstracción de la respuesta HTTP que el servidor envía a los clientes. La respuesta de un mensaje HTTP contiene una serie de cabeceras con información acerca de las características de esta. Estas cabeceras pueden ser alteradas fácilmente, ya que *Symfony* facilita un objeto *HeaderBag* para su manejo.

Por ejemplo, la siguiente acción crea una respuesta y le cambia el *content-type* a JSON. Este formato es muy útil cuando se quieren retornar datos al cliente a través de una API o servicio *REST* para ser consumido por el cliente desde *Javascript* (por ejemplo).

```

/**
 * Returns a JSON response.
 */
public function indexJsonAction() {
    $response = new Response(json_encode(array('location' =>
        'You are located in Oviedo, Spain')));
    $response->headers->set('Content-type', 'application/json');

    return $response;
}

```

Como se puede ver, al invocar esta acción desde el navegador se mostrará un *array* en *Javascript* con la información insertada desde la acción.

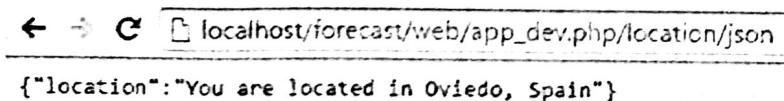


Figura 2.8. Respuesta de una acción en formato JSON

2.8. MOSTRAR UNA VISTA NO ASOCIADA A UNA ACCIÓN

En ocasiones puede ser útil crear vistas que no estén asociadas a ninguna acción de un controlador, ya que la información a mostrar es estática y no necesita de ningún procesamiento de datos previo. En ese caso, se

puede utilizar el controlador de *SymfonyFrameworkBundle:Template:template*. Para ello, basta con crear una vista dentro de la carpeta *app > Resources > views* de nuestro proyecto como hacemos normalmente. En este ejemplo crearemos una nueva carpeta dentro de *views* llamada *standalone* para guardar nuestra vista exenta de controlador.

Una vez hecho esto sólo falta declarar la ruta. Presta atención al valor de la clave *_controller*.

standalone:

```
path: "/standalone"
defaults:
    _controller: FrameworkBundle:Template:template
    template: standalone/index.html.twig
    maxAge: 86400
    sharedAge: 86400
```

Como el contenido de esta página va a ser siempre el mismo, suele ser habitual cachearlas para acelerar su carga en el cliente una vez que ya la ha visitado una primera vez. Para ello se utilizan los parámetros *maxAge* y *sharedAge*. El valor se expresa en segundos, por lo que estamos ajustando la duración de la cache a 24 horas.

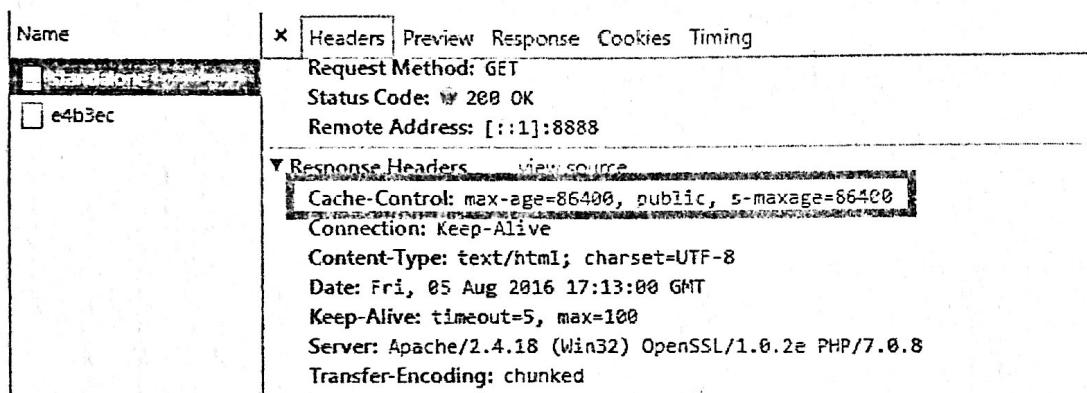


Figura 2.9. Cabeceras de respuesta del servidor

2.9. REDIRECCIONES A OTRAS ACCIONES

Aunque el flujo habitual suele ser que tras ejecutarse una acción de un controlador se renderice una vista, a veces podemos necesitar invocar otras acciones. Esto es posible con el método *forward()*.

```
/**
 * Action that will redirect the flow to another action.
 * @return Response
 */
public function indexAction() {
    $response = $this->forward('AppBundle:Forward:finish', array(
        'test' => true
    ));

    return $response;
}

/**
 * Action to be redirected to.
 * @return Response
 */
public function finishAction($test) {
    if ($test) {
```

```

        :
        return new Response(
            'Redirection done receiving the parameter successfully');
    } else {
        return new Response('Parameter was not received successfully');
    }
}
}

```

Como se puede ver, la primera acción `indexAction()` redirige la aplicación a `finishAction()` pasándole como parámetro la variable booleana `test`. La acción destino simplemente recibirá el flujo de ejecución e imprimirá un mensaje, entrando por el primer `if` si todo funciona correctamente.

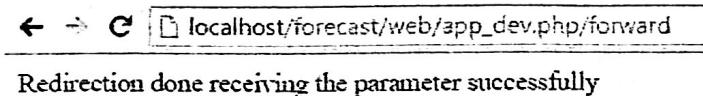


Figura 2.10. Resultado de realizar una redirección entre dos acciones

COMPRUEBA TU APRENDIZAJE

1º) Desarrolla una página que reciba dos parámetros, `user` y `password`. El `path` del `routing` debe tener el siguiente aspecto:

```
/login/{user}/{password}
```

Esto debe ir a un controlador llamado `SessionController` que debe decidir lo siguiente:

- Si el usuario y contraseña es correcto (por ejemplo si son `admin` y `1234` respectivamente), se almacena en sesión el usuario y el controlador debe hacer un `forward*` a una ruta que apunta a una acción en la que se indica que es correcto y se muestra el usuario almacenado en sesión.
- Si la contraseña empieza por letra mayúscula, entonces manejamos el error `createNotFoundException` y con un mensaje como “La contraseña debe empezar por minúscula”.
- Si el usuario y contraseña es incorrecto, es decir, no cumple con las condiciones anteriores, entonces, el controlador debe hacer un `redirect` a una ruta en la que un controlador cargue una plantilla `twig` para decir que el `login` es incorrecto.

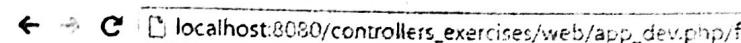
2º) Crea un controlador llamado `NetworkController`. En él se define las siguientes rutas:

- `/signin/{name}`: llamará a la acción `signinAction` del controlador enviándole el parámetro `name` (en caso de no recibir nada tomará el valor de `anónimo`). A continuación se guarda el nombre en sesión, y se crea un mensaje `flash` con este contenido: `"Hola ". $name`. Finalmente se “renderiza” a una página `index.html.twig`, donde se muestra el contenido del mensaje `flash`.

Ten en cuenta que esta acción solo debe permitir nombres formados por letras minúsculas o mayúsculas (*requirements*).

- `/{locale}`: dirige a la acción `languageAction`, que recibe el parámetro `locale`. Lo primero de todo es guardarla en sesión. Este parámetro tendrá el valor de “es” por defecto, y como requerimiento solo podrá tener el valor de `es`, `en` o `fr` (el o lógico se pondría así |).

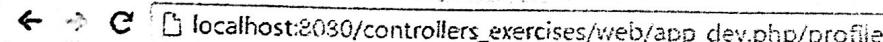
En la acción comprobaremos si el parámetro es igual a "es", en cuyo caso se "renderiza" una página con un saludo en español. En caso de que sea "en", "renderizamos" una página con el mensaje en inglés y si es "fr", en francés.



localhost:8080/controllers_exercises/web/app_dev.php/fr

Bonjour!

- `/profile`: conduce a la acción `profileAction`, en la que se retorna un mensaje con el nombre del usuario que actualmente existe en sesión. Algo que puedes hacer es, en caso de que exista un `locale` en sesión, devolver un mensaje con el nombre del usuario, en el idioma oportuno.



localhost:2030/controllers_exercises/web/app_dev.php/profile

Le profil de l'utilisateur: Eugenia

- `/info`: conduce a una acción `infoAction` que hace una redirección externa a una página web que consideres.
- `/logout`: en la acción `logoutAction` debemos establecer a cadena vacía las variables de sesión `nombre de usuario` y del `locale`. A continuación retornamos un mensaje de despedida.

ACTIVIDADES DE AMPLIACIÓN

En este capítulo hemos visto cómo mantener el estado de la aplicación entre peticiones a través de la sesión. Investiga qué otras maneras existen para persistir datos entre peticiones tanto en cliente como en servidor.

Además, modifica el ejemplo en el que se hacía uso de la sesión para que funcione con otro de los sistemas de persistencia que hayas encontrado.