Fundamentos HTTP

Qué es HTTP

HTTP es un *protocolo de aplicación* usado en *la web* (www) para comunicaciones entre cliente y servidor. El servidor puede ser un ordenador que aloja una aplicación web, y el cliente web más común que conocemos es el navegador o \*web browser". Pero con el tiempo han salido muchos otros modelos de cliente servidor. De hecho, en la actualidad es muy común que un servidor ejerza de cliente de otros servidores.

HTTP define cómo debe construirse una *petición* y cómo debe devolverse una *respuesta*.

La petición

En una petición se especifica: \* Una línea con el método, el recurso y la versión de HTTP \* Cabeceras opcionales \* La dirección del servidor \* El cuerpo de la petición, separado por una línea en blanco.

PUT /recetas/pollo-al-pil-pil HTTP/1.1

Host: cocinando.com

Content-Type: application/json

{nombre:"Pollo al Pil-Pil", dificultad:2, ingredientes:["1 chorrito de aceite de oliva", "2 pechugas de pollo fileteadas", "Una cabeza de ajos", "Guindillas", "Perejil", "1 limón", "Colorante", "1 vaso de cerveza", "Sal"]}

Métodos

La versión HTTP/1.1 define [ocho posibles métodos](http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html) para una petición. Estos métodos pueden tener algunas propiedades, como *seguro* e *idempotente*.

Métodos seguros

Son métodos que no desencadenan acciones en el servidor, aparte de recoger información. Entre ellos tenemos GET y HEAD. Aunque estos métodos estén definidos como seguros, depende de nosotros, los implementadores, que lo sean efectivamente. Es decir, debemos procurar que las llamadas GET y HEAD desencadenen acciones que alteren el estado del sistema.

Métodos idempotentes

Son métodos que podemos repetir varias veces obteniendo el mismo resultado en el servidor. Entre ellos tenemos GET, HEAD, PUT, DELETE, OPTIONS y TRACE. Por el contrario, el método POST no es idemponente, ya que repitiendo la misma operación alteramos el estado del sistema. Por definición, todos los métodos seguros son idempotentes.

Listado de métodos

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Seguro | Idempotente | Descripción |
| OPTIONS | Sí | Sí | Solicita información sobre las distintas opciones de comunicación disponibles. |
| GET | Sí | Sí | Recupera cualquier información en forma de recurso. |
| HEAD | Sí | Sí | Idéntico a GET, salvo que en la respuesta no se devuelve contenido. Usado para recibir algunos datos del recurso optimizando la transferencia. |
| POST | No | No | Almacena la entidad enviada. El URN representa a otra entidad que se encargará de gestionar ese almacenamiento. |
| PUT | No | Sí | Almacena la entidad enviada. El URN representa a la propia entidad. |
| DELETE | No | Sí | Elimina la entidad representada por el URN. |
| TRACE | Sí | Sí | Utiliza para debug, el servidor devuelve el propio contenido de la petición. |
| CONNECT |  |  | Reservado para establecer conexiones permanentes (tunneling) |
| PATCH (\*) | No | Sí | Realiza modificaciones parciales en entidades existentes. |

(\*) El método PATCH es bastante novedoso y no se incluye en el protocolo, aunque está siendo utilizado cada vez más.

Recursos

Los recursos se identifican por URNs (Uniform Resource Names). Los recursos deben estar identificados de manera unívoca, esto es, sólo debería haber un URN por cada recurso. Ejemplos de recursos válidos son:

* /
* /clientes/23
* /clientes/juan-martinez

Los query parameters enviados no definen recursos. Por ejemplo los siguientes recursos serían equivalentes y no se ajustarían al estándar HTTP. Ambos apuntarían a la colección "clientes":

* /clientes?nombre=Juan
* /clientes?nombre=Antonio

Cabeceras opcionales

El protocolo HTTP define [una serie de cabeceras](http://www.w3.org/Protocols/HTTP/HTRQ_Headers.html) que pueden enviarse adicionalmente en una petición. Adicionalmente se han extendido otras cabeceras no estándar que muchos servidores aceptan igualmente. Entre las cabeceras más comunes tenemos:

|  |  |  |
| --- | --- | --- |
| Nombre | Descripción | Ejemplo |
| From | Identifica al autor de la petición. | From: my@email.com |
| Accept | Formatos que el cliente acepta | Accept: text/plain, text/html |
| Accept-Encoding | Similar a Accept, especifica los formatos de codificación aceptados. | Accept-Encoding: x-zip |
| Referer | Especifica la dirección desde la que se ha accedido al recurso | Referer : http://misrecetas.com/pollo-al-pil-pil |

La respuesta

Una respuesta HTTP consta de:

* Una linea de status con la versión, el código de respuesta y el nombre del código.
* Cabeceras opcionales
* El cuerpo de la respuesta

HTTP/1.1 200 OK

Date: Mon, 23 May 2005 22:38:34 GMT

Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)

Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT

ETag: "3f80f-1b6-3e1cb03b"

Content-Type: text/html; charset=UTF-8

Content-Length: 131

Connection: close

<html>

<head>

<title>An Example Page</title>

</head>

<body>

Hello World, this is a very simple HTML document.

</body>

</html>

Códigos de respuesta

A continuación se describen algunos códigos de respuesta y su significado. Para una información más completa, consulta la [documentación oficial](http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html).

Código | Nombre | Descripción |

-------|--------|-------------|

100 | Continue | El cliente debe continuar enviando más informacion de la petición. |

200 | OK | La petición ha tenido éxito |

201 | Created | Un nuevo recurso ha sido creado |

202 | Accepted | La petición se ha aceptado y será procesada. |

204 | No Content | La petición se ha llevado a cabo con éxito, pero no se devuelve contenido |

300 | Multiple choices | Indica al cliente distintas opciones donde encontrar el recurso |

301 | Moved permanently | El recurso ya no existe. La nueva ruta debería proporcionarse en el cuerpo de la respuesta |

400 | Bad Request | El servidor no pudo procesar la petición porque esta no estaba bien formada |

403 | Forbidden | La petición ha sido rechazada |

404 | Not Found | El recurso no se ha encontrado |

500 | Internal Server Error | Ha ocurrido un error en el servidor |

501 | Not Implemented | La funcionalidad aún no ha sido implementada |

505 | HTTP Version Not Supported | La versión indicada en la petición es compatible con el servidor |

Request y Response

En el anteriores capítulos hemos visto algunos fundamentos de HTTP y definido Symfony como un *framework HTTP*. Symfony construye una capa de abstracción sobre HTTP, que se sustenta en dos clases importantes del componente HTTPFoundation: Request y Response.

La clase Request

class Request

{

public function \_\_construct(array $query = array(), array $request = array(), array $attributes = array(), array $cookies = array(), array $files = array(), array $server = array(), $content = null)

{

$this->initialize($query, $request, $attributes, $cookies, $files, $server, $content);

}

public function initialize(array $query = array(), array $request = array(), array $attributes = array(), array $cookies = array(), array $files = array(), array $server = array(), $content = null)

{

$this->request = new ParameterBag($request);

$this->query = new ParameterBag($query);

// ...

}

public static function createFromGlobals()

{

$request = new static($\_GET, $\_POST, array(), $\_COOKIE, $\_FILES, $\_SERVER);

// ...

return $request;

}

public static function create($uri, $method = 'GET', $parameters = array(), $cookies = array(), $files = array(), $server = array(), $content = null)

{

// ...

return new static($query, $request, array(), $cookies, $files, $server, $content);

}

public function \_\_toString()

{

return

sprintf('%s %s %s', $this->getMethod(), $this->getRequestUri(), $this->server->get('SERVER\_PROTOCOL'))."\r\n".

$this->headers."\r\n".

$this->getContent();

}

public function get($key, $default = null, $deep = false)

{

// ...

}

public function getSession()

{

return $this->session;

}

public function getContent($asResource = false)

{

// ...

return $this->content;

}

// ...

}

La clase Request contiene todo lo necesario para realizar una petición web, incluídas las cookies que permitirán gestionar las sesiones, archivos transferidos, etcétera. Podríamos crear una request utilizando las viariables globales PHP:

$request = Request::createFromGlobals();

Así es como se procesa una petición en web/app.php:

$kernel = new AppKernel('prod');

$request = Request::createFromGlobals();

$response = $kernel->handle($request);

Pero también es posible crear una petición programáticamente sin necesidad de realizar una request real, pasarla a nuestra aplicación Symfony y procesar la respuesta:

$kernel = new AppKernel('prod');

$request = Request::create('/recetas/pollo-al-pil-pil', 'GET');

$response = $kernel->handle($request);

La clase Response

class Response

{

public function \_\_construct($content = '', $status = 200, $headers = array())

{

// ...

}

public static function create($content = '', $status = 200, $headers = array())

{

return new static($content, $status, $headers);

}

public function \_\_toString()

{

return

sprintf('HTTP/%s %s %s', $this->version, $this->statusCode, $this->statusText)."\r\n".

$this->headers."\r\n".

$this->getContent();

}

public function sendHeaders()

{

header(sprintf('HTTP/%s %s %s', $this->version, $this->statusCode, $this->statusText));

// ...

return $this;

}

public function sendContent()

{

echo $this->content;

return $this;

}

public function send()

{

$this->sendHeaders();

$this->sendContent();

// ...

return $this;

}

// ...

}

Response encapsula una respuesta HTTP. Toda petición a AppKernel debe devolver un objeto response. Como se observa en web/app.php, una vez recuperado el objeto Response se invoca al método send() encargado de mostrar la respuesta:

$response = $kernel->handle($request);

$response->send();

Tipos especiales de respuesta

Un caso específico de respuesta es la clase [JsonResponse](http://api.symfony.com/2.2/Symfony/Component/HttpFoundation/JsonResponse.html) para respuestas cuyo contenido se devuelve en [JSON](http://en.wikipedia.org/wiki/JSON).

Otro ejemplo lo tenemos en la clase [RedirectResponse](http://api.symfony.com/master/Symfony/Component/HttpFoundation/RedirectResponse.html) que indica a los clientes la dirección a la que deben dirigirse.

La clase [StreamedResponse](http://api.symfony.com/master/Symfony/Component/HttpFoundation/StreamedResponse.html) ofrece respuestas en streaming y es utilizada fundamentalmente en medios pesados como vídeo o grandes volúmenes de datos.

Eventos en el ciclo de una petición

A lo largo del ciclo de vida de una petición, Symfony dispara distintos eventos que permiten a sus componentes reaccionar para modificar la respuesta o realizar acciones diversas. Cualquier bundle de nuestra aplicación puede asímismo adherirse a estos eventos.

Antes de procesar la petición, en el método handle() de la clase HttpKernel, se dispara un evento KernelEvents::REQUEST:

$event = new GetResponseEvent($this, $request, $type);

$this->dispatcher->dispatch(KernelEvents::REQUEST, $event);

Para obtener el controlador que atenderá la petición se utiliza un evento KernelEvents::CONTROLLER. El componente de routing recibe este evento, examina el objeto Request y la configuración de enrutado y devuelve el controlador correspondiente.

$event = new FilterControllerEvent($this, $controller, $request, $type);

$this->dispatcher->dispatch(KernelEvents::CONTROLLER, $event);

$controller = $event->getController();

En algunas ocasiones el controlador no devuelve una instancia de Response. Por ejemplo, la etiqueta [Template()](http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/view.html) permite aligerar las acciones de un controlador devolviendo un array y siguiendo cierto convenio. En estos casos, AppKernel dispara un evento KernelEvents::VIEW, y serán otros componentes los que se encarguen de construir el objeto Response.

Cualquier excepción que ocurra durante el procesamiento de una petición tendrá como consecuencia que se dispare un evento KernelEvents::EXCEPTION. Symfony utiliza este evento para enmascarar la excepción en una vista más amigable y ofrecer herramientas de debugging.

Una vez enviada la respuesta, en web/app.php se dispara un evento KernelEvents::TERMINATE.

$kernel->terminate($request, $response);

Routing

Para una información más completa y actualizada, consultad la [documentación oficial](http://symfony.com/doc/current/book/routing.html).

El sistema de enrutado relaciona los recursos indicados en las peticiones con los controladores encargados de gestionarlos.

Symfony carga todas las rutas del archivo de enrutado de la aplicación, normalmente en app/config/routing.yml. En este archivo podemos incluir referencias a otras fuentes junto con otras opciones de configuración.

app/config/routing.yml my\_recipes: resource: "@MyRecipesBundle/Resources/config/routing.yml" prefix: /

Los archivos de configuración pueden escribirse en yml, xml o php. En este material didáctico veremos todos los ejemplos en yml.

Crear una ruta

Es una buena práctica almacenar las rutas proporcionadas por nuestros bundles en sus propios archivos routing.yml. De esta manera garantizaremos su portabilidad y mantendremos el enrutado más organizado. Una ruta está formada por dos atributos básicos: el patrón de la ruta o path y un diccionario que define el controlador a utilizar:

recipes\_list:

path: /recipes/

defaults: { \_controller: MyRecipesBundle:Recipe:list }

recipes\_show:

path: /recipes/{recipe\_name}

defaults: { \_controller: MyRecipesBundle:Recipe:show }

Estas dos acciones están capturando las peticiones a los recursos especificados en path y enviándolas a sendos controladores. Los valores entre llaves simbolizan parámetros que son transferidos al controlador.

La clase controladora concreta y el método a utilizar se especifican según el siguiente convenio:

Nombre de bundle : Nombre del controlador : Nombre de la acción

De acuerdo con los anteriores ejemplos, se invocaría finalmente a los siguientes métodos:

My\RecipesBundle\Controller\RecipeController::list();

My\RecipesBundle\Controller\RecipeController::show($recipe\_name);

Parámetros opcionales

Si tenemos rutas que comparten una misma acción del controlador, podemos reorganizarlas utilizando parámetros por defecto. Por ejemplo, en la siguiente configuración:

recipes\_list:

path: /recipes/

defaults: { \_controller: MyRecipesBundle:Recipe:list }

recipes\_list\_page:

path: /recipes/{page}

defaults: { \_controller: MyRecipesBundle:Recipe:list }

Ambas dirigen la petición al controlador Acme\BlogBundle\Controller\BlogController::indexAction(), con la diferencia de que en la ruta inferior se está especificando el número de página. Estas dos rutas podrían refactorizarse en una única ruta con parámetro por defecto:

recipes\_list:

path: /recipes/{page}

defaults: { \_controller: MyRecipesBundle:Recipe:list, page: 1 }

Requisitos

Validación de parámetros

Si volvemos a los ejemplos anteriores, el resultado en nuestro fichero de enrutado sería el siguiente:

recipes\_list:

path: /recipes/{page}

defaults: { \_controller: MyRecipesBundle:Recipe:list, page: 1 }

recipes\_show:

path: /recipes/{recipe\_name}

defaults: { \_controller: MyRecipesBundle:Recipe:show }

¿Cómo debería reaccionar Symfony ante la petición /recipes/5? ¿Debería responder con la página 5 del listado de recetas? ¿O, al contrario, devería responder con la receta de nombre "5"?

En este caso Symfony respondería con la primera opción, ya que *las rutas que se definen primero tienen prioridad*. Pero la segunda ruta se vería por tanto enmascarada por la primera.

En Symfony es posible asegurar que las rutas cumplen algunos requisitos mediante la clave requirements. El conflicto anterior podría resolverse añadiéndose la siguiente validación:

recipes\_list:

path: /recipes/{page}

defaults: { \_controller: MyRecipesBundle:Recipe:list, page: 1 }

requirements:

page: \d+

recipes\_show:

path: /recipes/{recipe\_name}

defaults: { \_controller: MyRecipesBundle:Recipe:show }

Nótese la adición del parámetro requirements en la ruta recipes\_list, donde se especifica que el parámetro page debe ser un número entero (uno o más dígitos). Ante una petición al recurso /recipes/pollo-al-pil-pil, el componente de enrutado comprobaría en primer lugar que se cumplieran todos los requisitos de la ruta recipes\_list. Al no cumplirse el requisito, repetiría la operación con recipes\_show, vinculando la ruta a la acción My\RecipesBundle\Controller\RecipeController::show().

|  |
| --- |
| > Nota del Autor: Aunque este caso se expone en la documentación oficial, en mi opinión es una mala práctica resolver estos conflictos mediante la etiqueta requirements. Los recursos deberían ser unívocos, no dar lugar a la ambigüedad. Imagínese el caso de la novela "1984", que en una aplicación para bibliotecas podría responder al recurso /books/1984. ¡Seguríamos teniendo colisiones con la página 1984 del listado de libros!. Una solución mejor para resolver el problema de la paginación es utilizar query arguments: /books?page=1984. |

Un mejor uso de la etiqueta requirements es validar las rutas anticipadamente, antes de que las peticiones lleguen al controlador. Si validamos en el enrutado que un parámetro numérico {id} es efectivamente numérico, evitaremos realizar futuras comprobaciones o consultas innecesarias a la base de datos.

recipes\_show:

path: /recipes/{recipe\_id}

defaults: { \_controller: MyRecipesBundle:Recipe:show }

requirements:

recipe\_id: \d+

Validación de método

En Symfony es posible enviar recursos a controladores distintos en función del método utilizado.

recipes\_list:

path: /recipes/

defaults: { \_controller: MyRecipesBundle:Recipe:list }

methods: [GET]

recipes\_add:

path: /recipes/

defaults: { \_controller: MyRecipesBundle:Recipe:create }

methods: [POST]

Según esta configuración, se utilizará la acción RecipeController::list() para las peticiones a '/recipes/' con el verbo 'GET', y RecipeController::create() para la misma petición con el verbo 'POST'.

Validación de host

Symfony permite validar la dirección del host al que se envía la petición. Puede ser útil, por ejemplo, cuando queremos separar la web de un API REST utilizando subdominios:

recipes\_api\_list:

path: /recipes/

host: api.recipes.com

defaults: { \_controller: MyRecipesBundle:API:list }

recipes\_list:

path: /recipes/

defaults: { \_controller: MyRecipesBundle:Recipe:list }

Validación de formato

Con el filtro \_format podemos cribar las peticiones según la cabecera HTTP Content-Type.

recipes\_list:

path: /recipes/

defaults: { \_controller: MyRecipesBundle:Recipe:list }

requirements:

\_format: html|rss

recipes\_list\_json:

path: /recipes/

defaults: { \_controller: MyRecipesBundle:API:list }

requirements:

\_format: json

De este modo podemos obtener representaciones distintas del mismo recurso según el formato recibido.

Generar rutas con el componente de enrutado

Cualquier clase con acceso al contenedor de inyección de dependencias de Symfony puede generar URIs.

$router = $this->get('router');

$uri = $router->generate('recipes\_show', array('recipe\_id' => 55));

// $uri == '/recipes/55';

A continación se describen otros modos de uso:

// Rutas absolutas: http://www.misrecetas.com/recipes/55

$router = $this->get('router');

$uri = $router->generate('recipes\_show', array('recipe\_id' => 55), true);

// Query strings: /recipes/55?param1=foo

$router = $this->get('router');

$uri = $router->generate('recipes\_show', array('recipe\_id' => 55, 'param1' => 'foo'), true);

Los controladores disponen de un método auxiliar generateUrl().

$uri = $this->generateUrl('recipes\_show', array('recipe\_id' => 55));

// $uri == '/recipes/55';

Controlador

Para una información más completa y actualizada, consultad la [documentación oficial](http://symfony.com/doc/current/book/controller.html).

Un controlador es una función o método que recoge la información de una petición HTTP y devuelve una respuesta HTTP. Tal y como se ha descrito en el capítulo de enrutado, el enrutador es el encargado de buscar qué controlador se utiliza en cada petición y proporcionarle los parámetros necesarios.

recipes\_list:

path: /recipes/

defaults: { \_controller: MyRecipesBundle:Recipe:list }

recipes\_show:

path: /recipes/{recipe\_id}

defaults: { \_controller: MyRecipesBundle:Recipe:show }

// src/My/RecipesBundle/Controller/RecipeController.php

namespace My\RecipesBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class RecipeController extends Controller

{

public function listAction()

{

return new Response('<html><body><p>No hay recetas</p></body></html>');

}

public function showAction($recipe\_id)

{

return new Response('...');

}

}

Paso de parámetros

Además de los parámetros proporcionados por el enrutador, en Symfony podemos inyectar el propio objeto Request.

public function showAction($recipe\_id, Request $request)

{

return new Response('...');

}

Gracias al uso de la [reflexión](http://en.wikipedia.org/wiki/Reflection_computer_programming) no importa el orden en el que especifiquemos estos parámetros.

El controlador base

Symfony proporciona una clase base Controller que podemos extender en nuestros controladores y que proporciona algunos métodos útiles.:

Redirigir

Crea una respuesta de tipo RedirectResponse.

public function showAction()

{

return $this->redirect($this->generateUrl('redirect\_path'));

}

Reenviar

En lugar de redirigir al cliente indicándole una nueva ruta, pasa la petición a otro controlador.

public function showAction($id)

{

return $this->forward('MyForwardingBundle:Forwarded:show', array(

'id' => $id,

));

}

Internamente se realiza una [sub-request](http://symfony.com/doc/current/components/http_kernel/introduction.html" \l "http-kernel-sub-requests). El objeto Request se clona y se resuelve como si se tratase de una nueva petición.

Renderizar

La mayoría de aplicaciones web hacen uso de plantillas para renderizar contenido HTML y devolverlo en la respuesta. El controlador base de Symfony dispone de un método para ello:

public function showAction($id)

{

// ...

return $this->render('MyRecipesBundle:Recipe:show.html.twig', array(

'recipe' => $recipe,

));

}

Acceder a servicios

El controlador base de Symfony implementa la interfaz ContainerAwareInterface, y por tanto el framework automáticamente proporciona el contenedor de inyección de dependencias a cualquier clase que lo extienda.

public function showAction($id)

{

$templating = $this->get('templating');

$router = $this->get('router');

$mailer = $this->get('mailer');

}

Errores y excepciones

Para generar una respuesta con el código 404 basta con levantar una excepción de tipo NotFoundException.

public function showAction($id)

{

throw $this->createNotFoundException('...');

throw \Exception(...);

}

Cualquier excepción será capturada por Symfony, devolviendo un código de error 500.

|  |
| --- |
| ¡Ojo! En entornos de desarrollo, las excepciones son capturadas y se muestra una página de debugging, pero el código de respuesta es 200. Debe tenerse este factor en cuenta cuando se realicen pruebas manuales o automáticas sobre los códigos de respuesta, para evitar posibles falsos negativos/positivos. |

La sesión

Por defecto, Symfony 2 utiliza cookies para almacenar los datos de la sesión del cliente. Es posible manipular el contenido de estas cookies mediante el objeto sesión contenido en la petición:

public function showAction($id, Request $request)

{

$session = $request->getSession();

$session->set('clave', 'valor');

$session->get('clave');

}

Mensajes Flash

Los mensajes Flash se almacenan en la sesión y su objetivo es mostrar mensajes del sistema al cliente. Pueden definirse varios niveles de mensaje flash para representar la urgencia o importancia de cada uno de ellos.

public function createAction(Request $request)

{

$session = $request->getSession();

$session->getFlashBag()->add(

'notice',

'Has publicado una nueva receta'

);

}

Es nuestra labor renderizar estos mensajes en la plantilla correspondiente. Veremos más sobre esto en el capítulo de Twig.

{% for flashMessage in app.session.flashbag.get('notice') %}

<div class="flash-notice">

{{ flashMessage }}

</div>

{% endfor %}

SensioFrameworkExtraBundle

El Bundle SensioFrameworkExtraBundle proporciona algunas funcionalidades interesantes que podemos añadir a nuestros controladores en forma de anotaciones. Consulta [la documentación oficial](http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/index.html) para obtener una información más completa y actualizada.

@Route y @Method

Utiliza la anotación @Route para indicar la ruta de la que el controlador es responsable sin necesidad de ficheros de enrutamiento específicos. `@Method

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

/\*\*

\* @Route('/recipes')

\*/

class RecipeController extends Controller

/\*\*

\* @Route('/{id}', name="recipe\_show", requirements={"id" = "\d+"})

\* @Method({"GET"})

\*/

public function showAction($id)

{

// ...

}

}

@ParamConverter

Permite realizar algunas transformaciones sobre los parámetros de entrada del controlador. Útil, por ejemplo, en servicios REST donde los identificadores de los recursos están enmascarados.

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

/\*\*

\* @Route('/{id}', name="recipe\_show", requirements={"id" = "\d+"})

\* @ParamConverter("id", class="My\ReceiptBundle\MaskedId")

\*/

public function showAction(MaskedResource $id)

{

$public\_id = $id->getPublic();

$private\_id = $id->getPrivate();

// ...

}

El bundle proporciona dos conversores base. El de Doctrine permite cargar automáticamente entidades de la base de datos a partir de ids.

use My\RecipeBundle\Entity\Recipe;

/\*\*

\* @Route('/{id}', name="recipe\_show", requirements={"id" = "\d+"})

\*/

public function showAction(Recipe $recipe)

{

// ...

}

El DateTimeConverter transforma automáticamente fechas en objetos DateTime

use My\RecipeBundle\Entity\Recipe;

/\*\*

\* @Route('/{start}/{end}', name="recipe\_show")

\* @ParamConverter("start", options={"format": "Y-m-d"})

\* @ParamConverter("end", options={"format": "Y-m-d"})

\*/

public function listByDatesAction(\DateTime $start, \DateTime $end)

{

// ...

}

@Template

Con Template() podemos escribir controladores de un modo más elegante utilizando convenios en la organización de plantillas. Los métodos a continuación serían equivalentes:

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

public function showAction($id)

{

// ...

return $this->render('MyRecipesBundle:Recipe:show.html.twig', array(

'recipe' => $recipe,

));

}

/\*\*

\* @Template()

\*/

public function showAction($id)

{

// ...

return array(

'recipe' => $recipe,

);

}

La vista

En capítulos anteriores hemos visto que el enrutado se encarga de distribuir las peticiones entre los distintos controladores. Los controladores, a su vez, extraen la información necesaria de la petición y construyen con ellos una respuesta. Aunque sería posible devolver HTML directamente desde el controlador, es recomendable delegar esta función al motor de plantillas.

El motor por defecto en Symfony 2 es *twig*.

<!DOCTYPE html>

<html>

<head>

<title>Welcome to Symfony!</title>

</head>

<body>

<h1>{{ page\_title }}</h1>

<ul id="navigation">

{% for item in navigation %}

<li><a href="{{ item.href }}">{{ item.caption }}</a></li>

{% endfor %}

</ul>

</body>

</html>

En twig, las llaves dobles {{ ... }} se utilizan para mostrar una variable, mientras que la combinación de llave y paréntesis {% ... %} simboliza el uso de una expresión.

Tal y como vimos en el capítulo de controladores, para renderizar una plantilla desde un controlador utilizamos el método render().

public function showAction($id)

{

// ...

return $this->render('MyRecipesBundle:Recipe:show.html.twig', array(

'recipe' => $recipe,

));

}

Symfony buscará la plantilla show.html.twig indicada en la ruta src\My\RecipesBundle\Resources\views\Recipe\show.html.twig.

Baste con esta pequeña introducción a las vistas por ahora, más adelante dedicaremos un tema completo a Twig y el renderizado de plantillas.

El Modelo

El modelo de nuestra aplicación es todo aquello que representa a las necesidades de negocio. La mayoría de las aplicaciones disponen de una capa de persistencia para almacenar las distintas entidades en la base de datos. En una aplicación típica de Symfony 2, la interacción con la base de datos se abstrae utilizando [Object Relational Mappers](http://en.wikipedia.org/wiki/Object-relational_mapping) como [Doctrine 2](http://www.doctrine-project.org/). Estas herramientas, así como cualquier otra con la que interactúe la aplicación, son de libre elección.

Por lo tanto Symfony no ofrece guías respecto al modelado del negocio, la organización de las clases que la conforman y las decisiones en cuanto a la arquitectura y diseño del negocio.