Creación de aplicaciones web usando Symfony 2 en Eclipse

Descripción

¿Qué es Symfony?
Symfony es un Framework para desarrollo Web en PHP

¿Qué es un Framework?
Es una abstracción en la que el software proporciona una funcionalidad genérica que el usuario puede cambiar selectivamente mediante código, creando de este modo un software específico de aplicación.
Incluye programas de apoyo, compiladores, bibliotecas de código, una interfaz de programación de aplicaciones (API) y conjuntos de herramientas que reúnen a todos los diferentes componentes para permitir el desarrollo de un proyecto.

Las características principales:

- universal, reutilizable, Inversión de control, comportamiento por defecto, extensibilidad, código no modificable

Descripción

¿Por qué usar Symfony?

No es absolutamente necesario: es "sólo" una de las herramientas que están disponibles para ayudar a desarrollar mejor y más rápido!
- Mejor, se desarrolla una aplicación que cumple unas reglas de negocio y una estructura, y que es a la vez fácil de mantener y actualizable.
- Más rápido, ya que permite ahorrar tiempo mediante la reutilización de módulos genéricos.

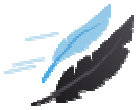
Invertir en la tarea, no en la tecnología
No tener que reinventar la rueda.

Capacidad de actualización y mantenimiento garantizado
A más largo plazo, un framework asegura la longevidad de sus aplicaciones.

Symfony no es una caja negra! En el caso de Symfony, todavía es PHP ...
Las aplicaciones que se desarrollan no se limitan al universo Symfony, y son de forma nativa interoperable con cualquier otra biblioteca de PHP.

Descripción

Beneficios tecnológicos de Symfony


Rápido y poco pesado en memoria


Flexibilidad ilimitada

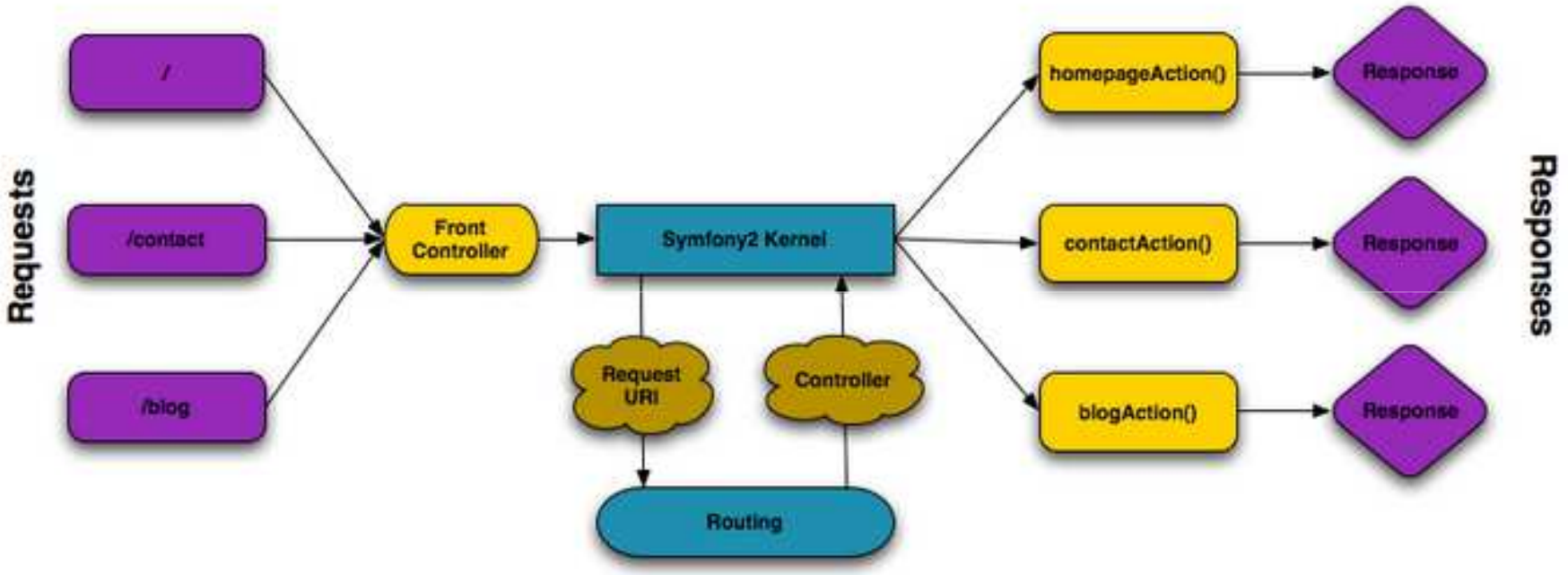

Extensible


Estable y sostenible

Comodidad para el desarrollo


Fácil de usar

Esquema global



Estructura de directorios

Es recomendable usarla, aunque no obligatorio:

- app/: Este directorio contiene la configuración de la aplicación
- src/: Todo el código PHP del proyecto
- vendor/: cualquier librería externa se coloca aquí por conveniencia
- web/: Este es el directorio raiz del web y contiene todos los ficheros accesibles

Estructura de directorios

Disponemos de dos entornos (Front Controllers): desarrollo y producción

/web/app_dev.php/ruta/de/nuestra_aplicacion

/web/app.php/ruta/de/nuestra_aplicacion

Por defecto, Symfony desactiva el entorno de producción

Estructura de un “bundle”

Se definen una estructura común para todos los “bundles” de Symfony2:

- **Controller/** contiene los controladores del “bundle”
- **DependencyInjection/** Contiene ciertas clases que permiten importar configuraciones de servicios, registrar pases del compilador y más (este directorio no es necesario)
- **Resources/config/** Guarda la configuración, incluyendo la configuración de rutas
- **Resources/views/** Contiene las plantillas organizadas por nombre del controlador
- **Resources/public/** Contiene recursos web (imagenes, hojas de estilo, etc) y se copia o se vincula en el directorio web/ mediante el comando de consola “assets:install”
- **Tests/** Contiene todos los tests del “bundle”.

Un “bundle” es tan pequeño o grande como la característica que implementa y sólo contiene los ficheros que son necesarios, nada más.

Symfony en Eclipse

Help -> Install New Software

Add Repository

Framework Symfony

Symfony - http://pulse00.github.com/p2/

Plugin Apatana, que usaremos para configurar un FTP

Aptana - http://download.aptana.com/studio3/plugin/install

Shell para ejecutar comandos de consola

wickedshell - http://www.wickedshell.net/updatesite

Symfony en Eclipse

Es necesario tener instalado “php” para poder ejecutar comandos con la consola de Symfony en local:

php5 - http://windows.php.net/download/#php-5.4

Agregar al PATH de Windows la ruta de instalación.

Dentro del directorio de instalación creamos el fichero *php.ini* copiando *php.ini-development* o *php-ini-production*

Editamos el fichero php.ini descomentando las siguientes líneas y asignando los valores:

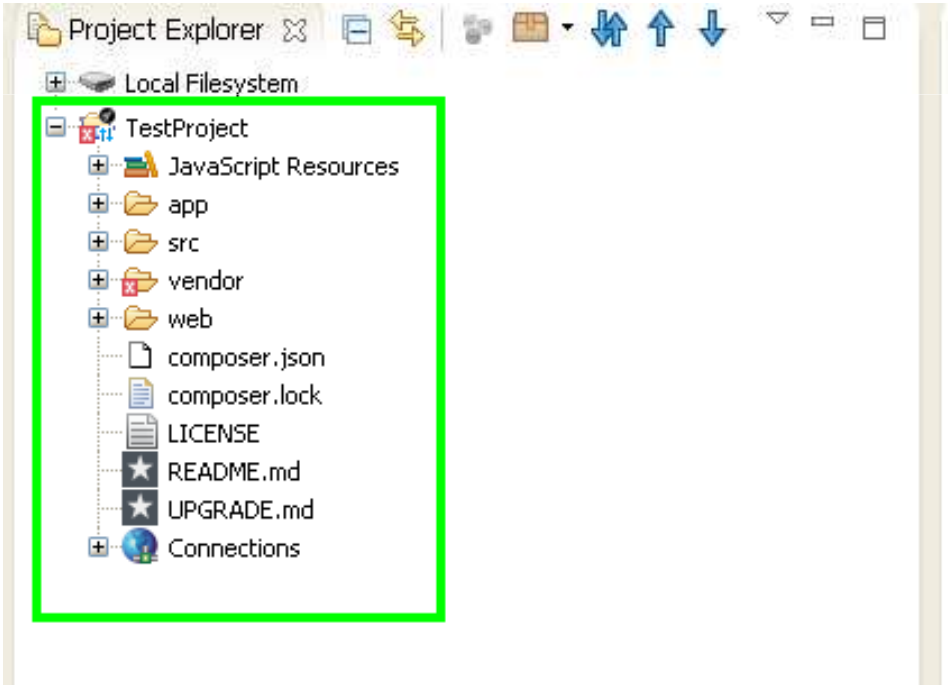
```
...
date.timezone = Europe/Madrid
...
extension=ext/php_pdo_mysql.dll
```

Symfony en Eclipse

Creando un proyecto

File > New > Project.. > Symfony Project

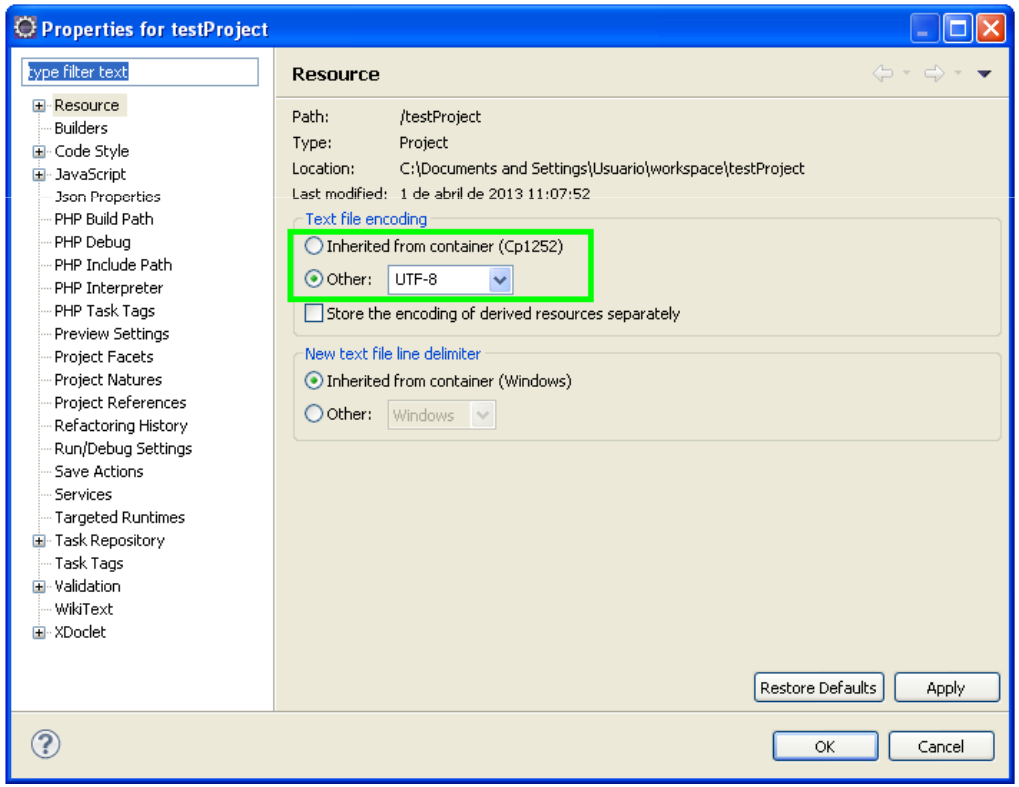
Damos un nombre cualquiera al proyecto y en la parte izquierda nos aparecerá el nuevo proyecto Symfony.



Symfony en Eclipse

Creando un proyecto

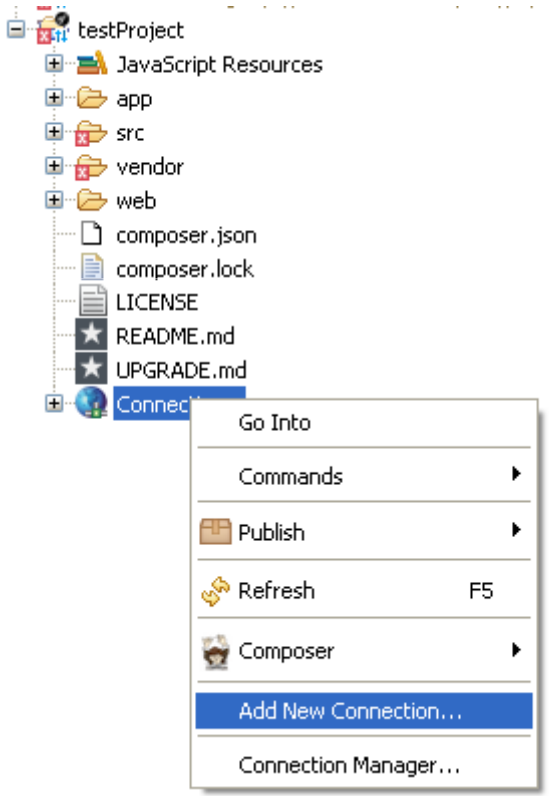
Click con el botón derecho en el proyecto y abrimos las propiedades *"Properties".* Es importante configurar la codificación en UTF-8.



Symfony en Eclipse

Creando un proyecto

Ahora es necesario conectar nuestro proyecto con nuestro sitio FTP.

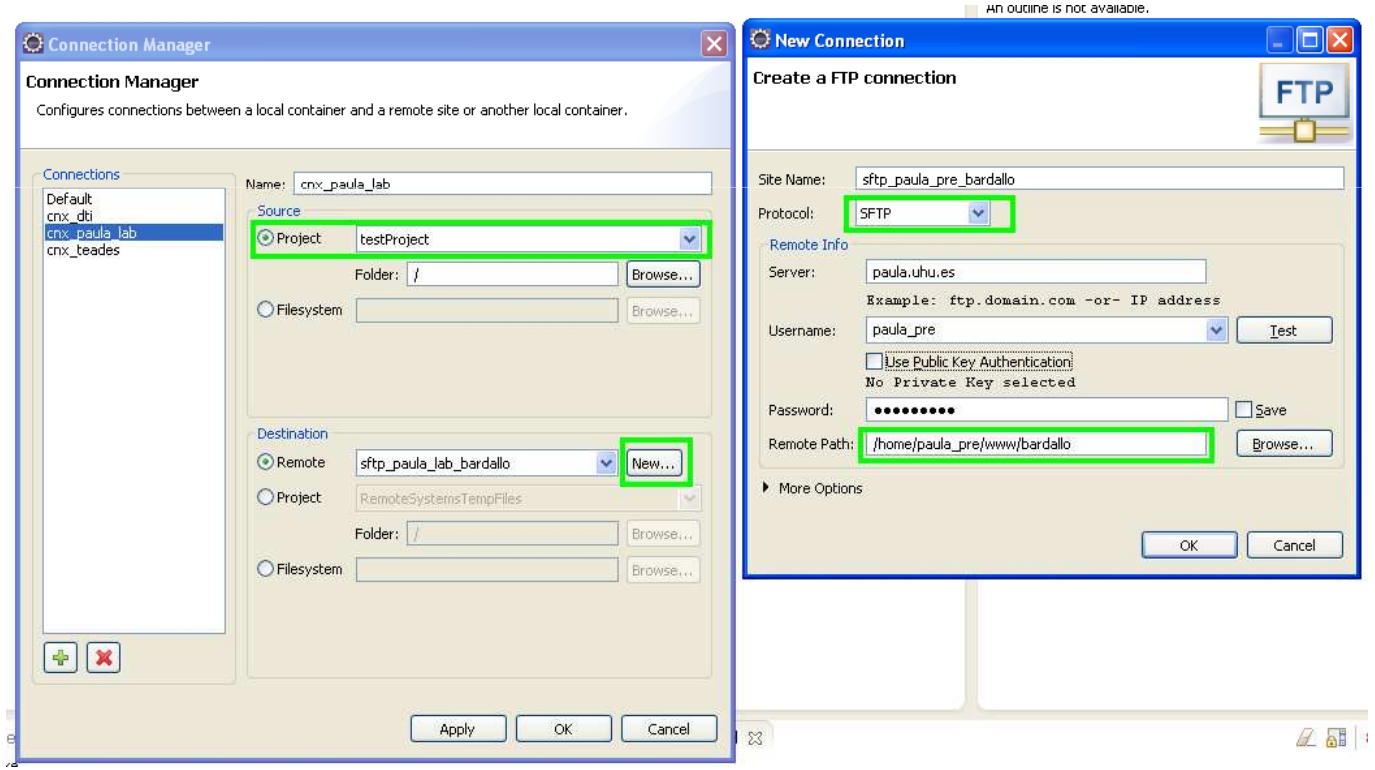


Symfony en Eclipse

Creando un proyecto

Source > “Nombre de nuestro proyecto”

Destination > New.. > Creamos una conexión SFTP



Symfony en Eclipse

Creando un proyecto

Ahora podemos subir, bajar o sincronizar ficheros de nuestro proyecto seleccionandolos en el Explorador de proyectos y usando los iconos:

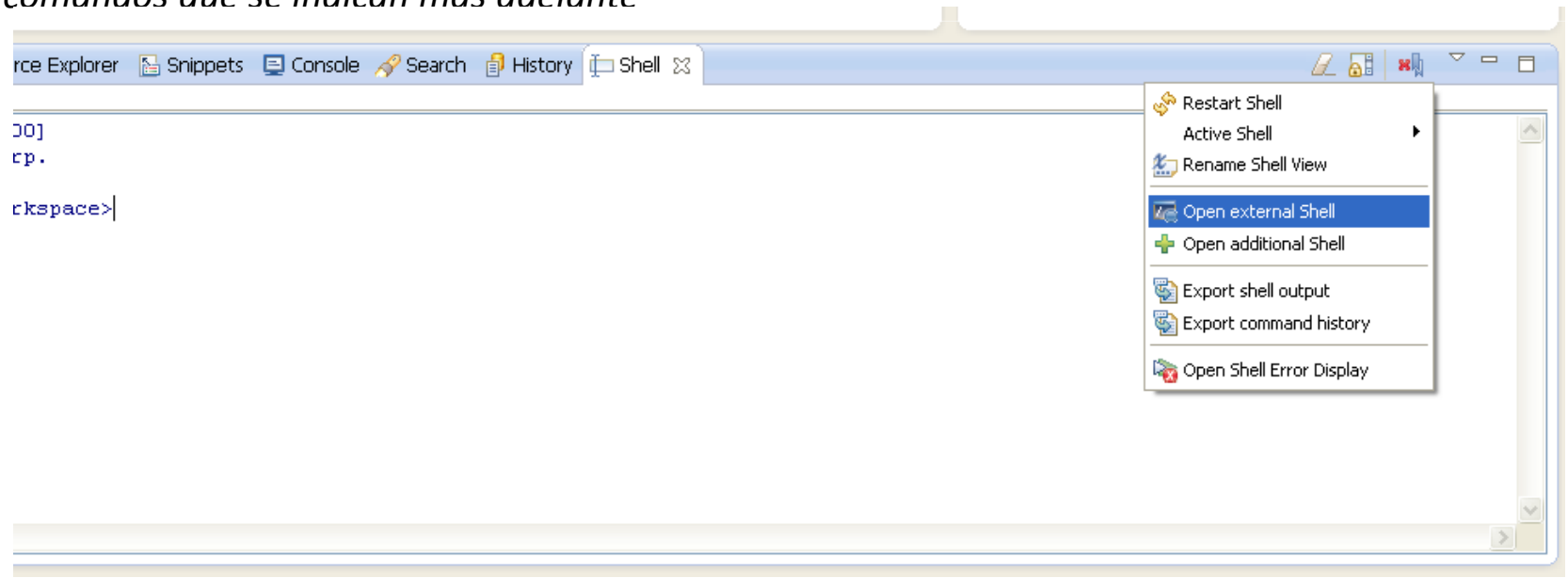
NOTA: Cada vez que hagamos cambios en nuestro proyecto no hay que olvidar subir los ficheros para poder verlos en el navegador

Symfony en Eclipse

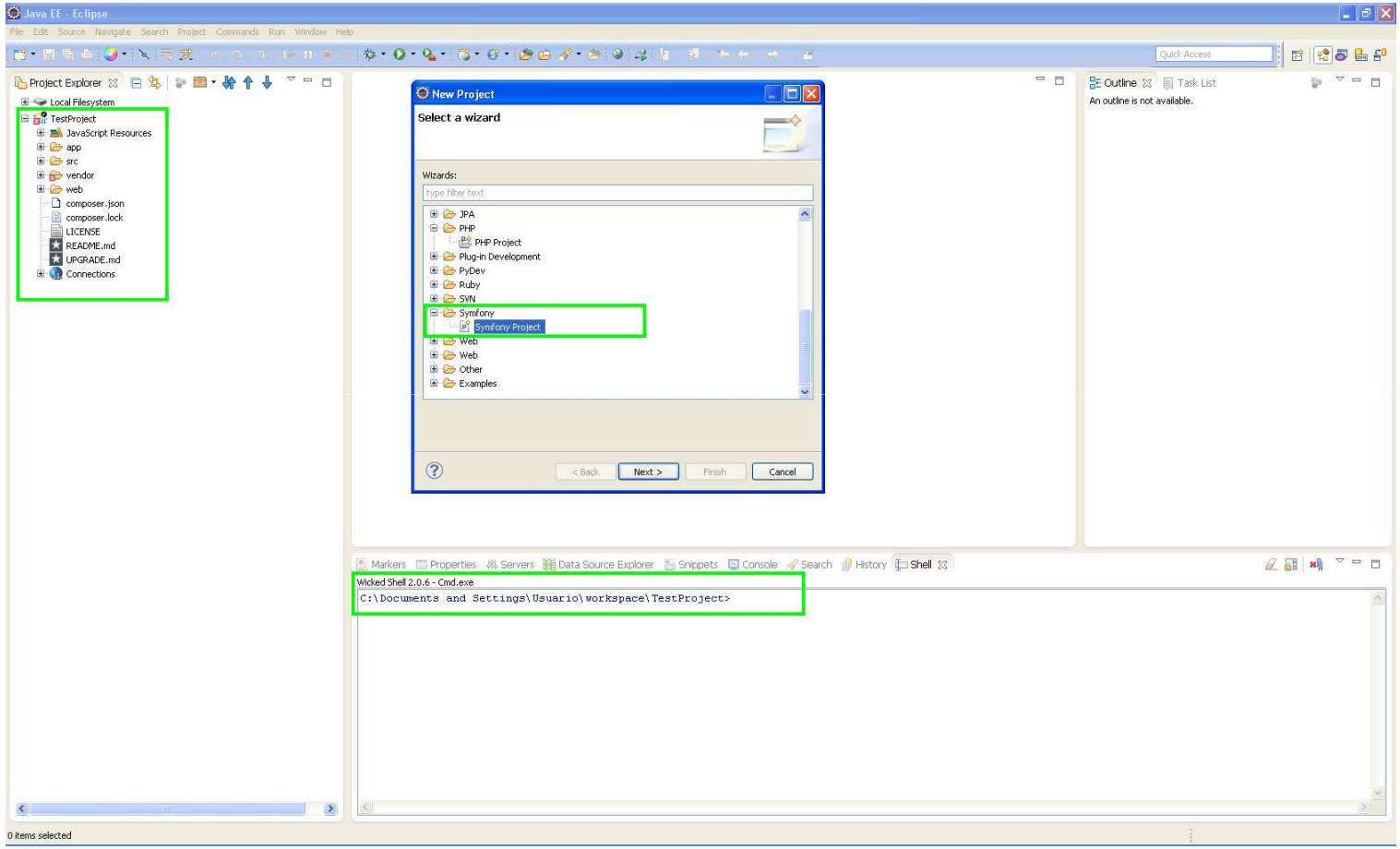
Creando un proyecto

En el entorno hemos instalado el wicked shell, sin embargo, al menos en Windows no está muy conseguido…podemos usar el cmd de windows.

***NOTA**: No olvidar posicionarse en la ruta de nuestro proyecto "cd testProject" para ejecutar los comandos que se indican más adelante*



Symfony en Eclipse



Notas de Instalación

Symfony necesita que tanto el directorio **app/cache** como el directorio **app/logs** sean totalmente accesibles por nuestro usuario.

1. Usando ACL con chmod +a

```
$ sudo chmod +a "www-data allow
    delete,write,append,file_inherit,directory_inherit" app/cache app/logs
$ sudo chmod +a "`whoami` allow
    delete,write,append,file_inherit,directory_inherit" app/cache app/logs
```

2. Usando ACL con setfacl

```
$ sudo setfacl -R -m u:www-data:rwX -m u:`whoami`:rwX app/cache app/logs
$ sudo setfacl -dR -m u:www-data:rwx -m u:`whoami`:rwx app/cache app/logs
```

3. Sin usar ACL (nuestro caso)

   Añadimos una de las instrucciones al principio de los ficheros **app/console**, **web/app.php** y **web/app_dev.php**, dependiendo de los permisos que queramos:

```
umask(0002); // Dará permisos 0775
umask(0000); // Dará permisos 0777
```

Notas de Instalación

Por otro lado, el fichero *web/app_dev.php* no puede usarse fuera de localhost y por tanto no podemos llamarlo desde fuera. Para evitar esto hay que comentar desde la línea 9 a la 21 aproximadamente.

El código es el siguiente:

```
/*
// This check prevents access to debug front controllers that are deployed by
accident to production servers.
// Feel free to remove this, extend it, or make something more sophisticated.
if (isset($_SERVER['HTTP_CLIENT_IP'])
    || isset($_SERVER['HTTP_X_FORWARDED_FOR'])
    || !in_array(@$_SERVER['REMOTE_ADDR'], array(
        '127.0.0.1',
        '::1',
    ))
) {
    header('HTTP/1.0 403 Forbidden');
    exit('You are not allowed to access this file. Check '.basename(__FILE__).'
for more information.');
}
/**/
```

Creando un “bundle”

Ejecutamos desde wickedshell o cmd, donde cada uno esté más cómodo ☺

```
>php app/console generate:bundle --namespace=Adib/TestBundle --format=yml
```

Se crea nuestro “bundle” en src/Adib/TestBundle

Además, en *app/AppKernel.php* se registra el nuevo “bundle”

```
        public function registerBundles() {
                ...
                $bundles = array(
                ...
                new Adib\TestBundle\AdibTestBundle(),
                ...
```

Y se crea una nueva entrada en *app/config/routing.yml*

```
    adib_test:
        resource: "@AdibTestBundle/Resources/config/routing.yml"
        prefix:   /
```

Probando el nuevo “bundle”

¡No hay que olvidar subir los nuevos ficheros creados!

direccion_servidor/nuestro_directorio/web/app_dev.php/hello/Nombre

src/Adib/TestBundle/Resources/config/routing.yml

```
adib_test_homepage:
    pattern:  /hello/{name}
    defaults: { _controller: AdibTestBundle:Default:index }
```

Redirige todas las peticiones con la forma /hello/*nombre* al método indexAction del controlador DefaultController

```
...
public function indexAction($name){
        return $this->render('AdibTestBundle:Default:index.html.twig',
array('name' => $name));
}
...
```

Renderiza la plantilla src/Adib/TestBundle/Resources/views/Default/index.html.twig

NOTA: Para que se vean los cambios hechos en las plantillas hay que borrar siempre el contenido del directorio cache en el entorno de desarrollo

Sistema de rutas

Se leen desde el fichero routing.yml

Pueden escribirse en YML, PHP o XML. Symfony recomienda YML

En el formato YML es importante tener en cuenta las **tabulaciones** a la hora de escribir las diferentes líneas.

Todo aquello que dependa de un nodo anterior debe estar tabulado respecto a éste Ej:

```
Nombre_de_la _ruta:
        parametro1: cualquier_valor
        parametro2:
                parametro1_2: cualquier_valor
```

En el ejemplo, parametro1 y parametro2 dependen de *`Nombre_de_la_ruta`*, por tanto están tabulados respecto a la misma.

Por otro lado, `parametro1_2` depende de `parametro2`, por eso, al igual que antes, está tabulado respecto a `parametro2`.

Sistema de rutas

Los ficheros se leen **secuencialmente** y se usa la **primera ruta** que concuerde con la URL introducida.

Ejemplos:

```
blog:
    path: /blog/{page}
    defaults: { _controller: AcmeBlogBundle:Blog:index, page: 1}
    requirements:
        page: \d+


article_show:
    path: /articles/{culture}/{year}/{title}.{_format}
    defaults: { _controller: AcmeDemoBundle:Article:show, _format: html }
    requirements:
        culture: en|fr
        _format: html|rss
        year: \d+
```

Sistema de rutas

El parámetro **_controller** en una entrada de ruta, especifica el controlador a usar. Para indicarlo se usa un patrón que tiene tres partes separadas por ":"

bundle:controller:action

Por ejemplo, *AcmeBlogBundle:Blog:show* se traduce como:

Bundle: AcmeBlogBundle , ruta: src/Acme/BlogBundle

Controlador: BlogController , ruta: src/Acme/BlogBundle/Controller/BlogController.php

Método: showAction

Sistema de rutas

Los parámetros de una ruta, pueden pasarse como argumentos al método del controlador al que se hace referencia.

Tan sólo hay que llamar a las variables con el mismo nombre

Cambiamos la ruta en nuestro fichero routing.yml

```
    pattern:  /hello/{name}/{surname}
```

Cambiamos nuestra plantilla twig index.html.twig

```
     Hello {{ name }} {{ surname }}, welcome to Adib!
```

Cambiamos el método de nuestro controlador por defecto

```
    public function indexAction($surname,$name){
        return $this->render('AdibTestBundle:Default:index.html.twig',
array('name' => $name, 'surname' => $surname));
    }
```

Sistema de rutas

Además podemos importar rutas de otros ficheros. El ejemplo más claro se encuentra en *app/config/routing.yml*, donde se importa nuestro fichero de rutas

```
adib_test:
    resource: "@AdibTestBundle/Resources/config/routing.yml"
    prefix:   /
```

Cuando importamos rutas, podemos indicar un "prefix" (prefijo) que se añadirá automáticamente a la ruta a la hora de calcularse la correspondencia.

```
adib_test:
    resource: "@AdibTestBundle/Resources/config/routing.yml"
    prefix:   /admin
```

Esto hará que las rutas aceptadas sean del tipo ***/admin****/hello/{name}* en lugar de */hello/{name}*, como vemos, se le añade el prefijo ***/admin*** indicado

Sistema de rutas

Trabajar con rutas

En el código podemos trabajar con las rutas. Tomemos como ejemplo nuestra ruta:

```
adib_test_homepage:
  pattern: /hello/{name}/{surname}
  defaults: { _controller: AdibTestBundle:Default:index }
```

Comprobar rutas:

```
$params = $this->get('router')->match('/hello/John/Doe');
// array(
        // 'name' => ‘John',
        // 'surname' => 'Doe'
        // '_controller' => 'AdibTestBundle:Default:index',
// )
```

Sistema de rutas

Trabajar con rutas

Generar rutas:

Relativas:

```
$uri = $this->get('router')->generate('adib_test_homepage', array('name' =>
'John', 'surname' => 'Doe'));
// /hello/John/Doe
```

Absolutas, tan sólo pasamos un tercer parámetro a *true* al método generate:

```
$uri = $this->get('router')->generate('adib_test_homepage', array('name' =>
'John', 'surname' => 'Doe'),true);
// http://servidor.com/hello/John/Doe
```

Podemos también añadir parámetros extras

```
$uri = $this->get('router')->generate('adib_test_homepage', array('name' =>
'John', 'surname' => 'Doe', 'foo' => 'bar'));
// /hello/John/Doe?foo=bar
```

Sistema de rutas

Como vemos las posibilidades del sistema de rutas son muchas, y aún hay más

Desafortunadamente, aquí no lo veremos ☹

Para saber más:

http://symfony.com/doc/current/book/routing.html

http://symfony.com/doc/current/cookbook/routing/scheme.html

Controlador

Es una función PHP que recibe una petición HTTP, realiza ciertas operaciones y devuelve una respuesta, en este caso un objeto ***Response*** de Symfony2.

El objeto ***Response*** puede ser:

-Página HTML

-Documento XML

-Un array JSON

-Una redirección

-…

Controlador

Symfony cuenta con una clase base ***Controller***, aunque no es necesario extenderla, si es muy recomendable ya que nos permite acceder a muchos métodos ya implementados por Symfony.

```
namespace Acme\HelloBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class HelloController extends Controller {
     public function indexAction($name) {
         return new Response('<html><body>Hello '.$name.'!</body></html>');
     }
}
```

Controlador

Funciones extendidas de Controller

- Redirección
  - `return $this->redirect($this->generateUrl('homepage'));`
  - `return new RedirectResponse($this->generateUrl('homepage'));`
- Reenvío
  - `$url = $this->generateUrl('adib_test_homepage', array('name' => 'John')`
- Generar rutas
  - `$response = $this->forward('AcmeHelloBundle:Hello:fancy', array( 'name' => $name, 'color' => 'green', ));`
- Renderizar plantillas
  - `$content = $this->renderView('AcmeHelloBundle:Hello:index.html.twig', array('name' => $name) ); return new Response($content);`
  - `return $this->render('AcmeHelloBundle:Hello:index.html.twig', array('name' => $name) );`

Controlador

Funciones extendidas de Controller

- Acceso a otros servicios

```
$router = $this->get(‘router’);
$request = $this->getRequest();
$templating = $this->get('templating');
$mailer = $this->get('mailer');
```

- Manejo de errores

```
throw $this->createNotFoundException('The product does not exist');
throw new \Exception('Something went wrong!');
```

- Manejo de la sesión

```
$session = $this->getRequest()->getSession();
$session->set('foo', 'bar');
$foo = $session->get('foo');
$filters = $session->get('filters', array());
```

Controlador

- Objeto ***Response***

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

$response = new Response('Hello '.$name, 200);

$response = new Response(json_encode(array('name' => $name)));
$response->headers->set('Content-Type', 'application/json');
```

- Objeto ***Request***

```php
$request = $this->getRequest();
$request->isXmlHttpRequest(); // is it an Ajax request?
$request->getPreferredLanguage(array('en', 'fr'));
$request->query->get('page'); // get a $_GET parameter
$request->request->get('page'); // get a $_POST parameter
```

Bases de Datos

Trabajar con ORM (Object Relational Mapper)

Nos ayuda a trabajar con bases de datos de una forma más cómoda.

Se encarga de añadir/extraer información a/desde la base de datos

Ayuda a persistir los datos en memoria mapeando las tablas en objetos

Symfony2 viene integrado con dos ORMs: Doctrine y Propel

Ambas herramientas están desacopladas de Symfony y su uso es opcional

Symfony2 usa por defecto Doctrine.

Doctrine

La mejor forma de entender como trabaja es verlo en acción.

- Configurar la base de datos
- Crear un objeto tarea (Task)
- Almacenarlo en la base de datos
- Extraerlo desde la base de datos

Doctrine

Configurar la base de datos

Por conveniencia, los parámetros de conexión con la base de datos se especifican en el fichero *app/config/parameters.yml*

```
parameters:
    database_driver: pdo_mysql
    database_host: paula.uhu.es
    database_name: symfony_1
    database_user: usu_sym
    database_password: symfony13
```

NOTA: El driver pdo_mysql debe estar activado en php

Doctrine puede crear la base de datos (si se cuentan con suficientes privilegios ;-))

```
>php app/console doctrine:database:create
```

Doctrine

Crear un objeto

• Se pueden crear clases manualmente y luego hacer que Doctrine genere las tablas correspondientes en la base de datos

• Doctrine permite generar las clases automáticamente al mismo tiempo que genera las tablas correspondientes en la base de datos.

Los objetos se crean en la ruta *src/Adib/TestBundle/***Entity**

Veremos las dos formas de crear una clase y hacerla persistente en una base de datos.

Doctrine

Crear un objeto

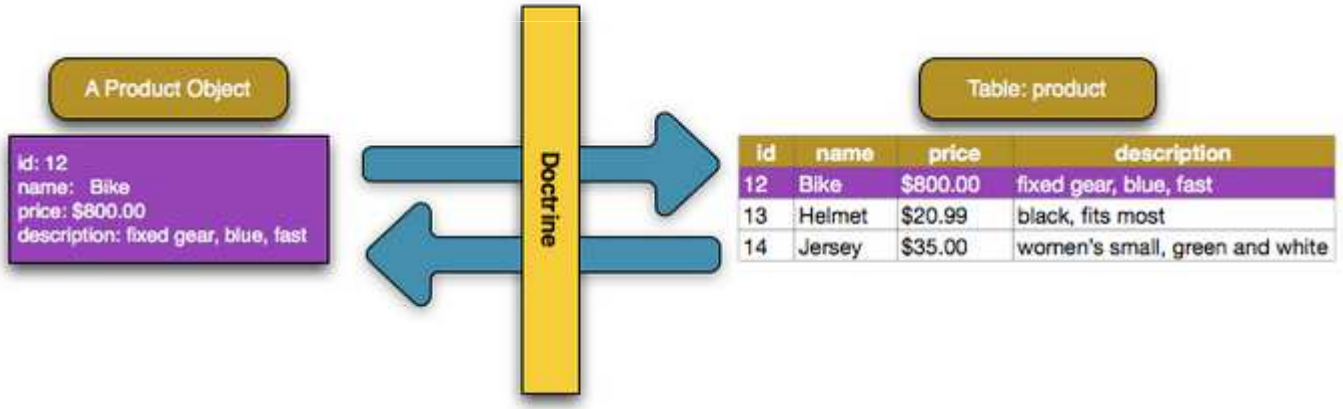
```php
<?php
// src/Adib/TestBundle/Entity/Task.php
namespace Adib\TestBundle\Entity;

class Task {
    protected $title;
    protected $description;
}
```

Doctrine

Crear un objeto

Doctrine es capaz de persistir objetos en bases de datos y extraer información de bases de datos en objetos pero…¿cómo es capaz de hacerlo?

Es necesario añadirle al objeto información de Mapeado



id	name	price	description
12	Bike	$800.00	fixed gear, blue, fast
13	Helmet	$20.99	black, fits most
14	Jersey	$35.00	women's small, green and white

Doctrine

```php
<?php
namespace Adib\TestBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
/**
 * @ORM\Entity
 * @ORM\Table(name="task")
 */
class Task {
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;
    /**
     * @ORM\Column(type="string", length=100)
     */
    protected $title;
    /**
     * @ORM\Column(type="text")
     */
    protected $description;
}
```

Doctrine

Crear un objeto

• Aunque en este caso se han usado anotaciones, Symfony permite usar YAML o XML para especificar los metadatos. En ningún caso se pueden mezclar varios lenguajes entre sí.

• Es necesario añadir un atributo “id” que será el que Doctrine use como primary key

• El nombre de la tabla es opcional, si se omite, Doctrine usará el nombre de la clase

Doctrine

Crear un objeto

Aunque Doctrine conoce ahora como tiene que mapear la información del objeto, aún no podemos acceder a sus propiedades.

Es necesario crear los “getters” y “setters”, afortunadamente, Doctrine lo puede hacer por nosotros :-P

```
>php app/console doctrine:generate:entities Adib/TestBundle/Entity/Task
```

Esta instrucción nos permite:

- generar “getters” and “setters”. Siempre actualiza, nunca borra
- generar clases de repositorio configuradas con la anotación

```
@ORM\Entity(repositoryClass="...")
```

- generar el constructor apropiado para las relaciones 1:n and n:m.

Ademas podemos generar todas las entidades conocidas dentro de diferentes ámbitos

```
>php app/console doctrine:generate:entities AdibTestBundle
>php app/console doctrine:generate:entities Adib
```

Doctrine

Almacenar el objeto en la base de datos

Es necesaria una tabla en nuestra base de datos para almacenar el objeto.

De nuevo Doctrine nos facilita la vida ☺

```
>php app/console doctrine:schema:update --force
```

Este comando también es capaz de actualizar columnas dentro de las tablas

¡Todo Listo!, ahora podemos guardar nuestros objetos en la base de datos y viceversa

Doctrine

Almacenar el objeto en la base de datos

Crear una nueva función en nuestro controlador para crear una tarea

```
    use Adib\TestBundle\Entity\Task;
    use Symfony\Component\HttpFoundation\Response;
    ...
    public function createAction(){
        $task = new Task();
        $task->setTitle("Titulo de la tarea");
        $task->setDescription("Descripcion de la tarea");

        $em = $this->getDoctrine()->getManager();
        /*El método persist() le indica a Doctrine que tiene que gestionar
el objeto Task, pero no realiza ninguna interacción con la base de datos.*/
        $em->persist($task);

/* El método flush() hace que Doctrine recorra todos los objetos marcados
para gestionar y realice las operaciones oportunas. */
        $em->flush();

        return new Response("Creada la tarea con id ".$task->getId());
    }
```

Doctrine

Almacenar el objeto en la base de datos

Tenemos la acción creada en el controlador…¿falta algo?

Hay que crear una ruta para poder ejecutar la acción :-P

Añadimos una ruta al fichero *Resources/config/routing.yml*

```
adib_create_task:
  pattern: /create/task
  defaults: {_controller: AdibTestBundle:Default:create }
```

Doctrine

Extraer el objeto desde la base de datos

Ahora crearemos una acción en nuestro controlador para extraer un objeto de la base de datos y la llamaremos con la ruta: */show/task/{id}*

```
    public function showAction($id)
    {
        $task = $this->getDoctrine()
            ->getRepository('AdibTestBundle:Task')
            ->find($id);

        if (!$task) {
            throw $this->createNotFoundException('No se encontró ninguna
tarea con id '.$id);
        }

        return new Response("Encontrada la tarea con id ".$id."<br>Titulo:
".$task->getTitle()."<br>Descripcion: ".$task->getDescription());
    }
```

Doctrine

Extraer el objeto desde la base de datos

Cuando realizamos consultas sobre un objeto usamos lo que se conoce como su "repositorio".

```
$repository = $this->getDoctrine()->getRepository('AdibTestBundle:Task');
```

Esta clase nos ayuda con una serie de métodos para realizar consultas

```
// Consultar por clave primaria(normalmente "id")
$task = $repository->find($id);
// Nombres de métodos dinámicos para buscar por el valor de una columna
$task = $repository->findOneById($id);
$task = $repository->findOneByTitle('foo');
// buscar *todas* las tareas
$tasks = $repository->findAll();
// Buscar un grupo de tareas basado en el valor de una columna
$tasks = $repository->findByPriority(5);
```

Doctrine

Extraer el objeto desde la base de datos

```
// Consultar una tarea por título y prioridad
$task = $repository->findOneBy(array('title' => 'foo', 'priority' => 5));
// Consultar todas las tareas por título ordenadas por prioridad
$tasks = $repository->findBy(
                        array('title'=>'foo'),
                        array('priority'=>'ASC')
);
```

Doctrine

Extraer el objeto desde la base de datos

En todos los ejemplos hemos usado la clase **Repository** para extraer objetos de la base de datos, la cual ofrece una serie de funcionalidades para manejar dicho objeto.

Las Anotaciones como alternativa

Puede hacerse mediante anotaciones de una forma más cómoda. No lo veremos aquí (de momento…)

http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html

Doctrine

Operaciones con los objetos

Actualización

```
$em = $this->getDoctrine()->getManager();
$task= $em->getRepository('AdibTestBundle:Task')->find($id);
if (!$task) {
    throw $this->createNotFoundException('No existe la tarea '.$id);
}
$task->setTitle('Nuevo título!');
$em->flush();
```

No es necesario llamar a persist(), tan sólo a flush() ya que al llamar a find(), Doctrine ya lo marca para "gestionarlo"

Doctrine

Operaciones con los objetos

Borrado

```
$em = $this->getDoctrine()->getManager();
$task= $em->getRepository('AdibTestBundle:Task')->find($id);
$em->remove($task);
$em->flush();
```

En este caso, el método remove() le indica a Doctrine que marque el objeto para borrado. La consulta DELETE no se ejecuta hasta llamar a flush()

Doctrine

Lenguaje DQL (Doctrine Query Language)

Es muy similar a SQL, solo tenemos que pensar en términos de “objetos” en lugar de filas en una base de datos

```
$em = $this->getDoctrine()->getManager();
$query = $em->createQuery( 'SELECT t FROM AdibTestBundle:Task t WHERE t.priority >
:priority ORDER BY t.priority ASC' )->setParameter('priority', '10');
$tasks = $query->getResult();
```

Podemos asignar más de un parametro en una consulta

```
    ->setParameters(array('priority' => '10', 'name' => 'Foo', ))
```

Doctrine

Lenguaje DQL (Doctrine Query Language)

getResult() devuelve un array de resultados, si solo queremos uno usamos *getSingleResult()* pero debemos asegurar que la consulta sólo devolverá uno, de lo contrario lanzará una excepción que hay que controlar.

```
$query = $em->createQuery('SELECT ...') ->setMaxResults(1);
try {
 $task = $query->getSingleResult();
} catch (\Doctrine\Orm\NoResultException $e) {
 $task = null;
}
```

Para saber más

http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html

Doctrine

Doctrine Query Builder

Es una alternativa a escribir las consultas SQL directamente.

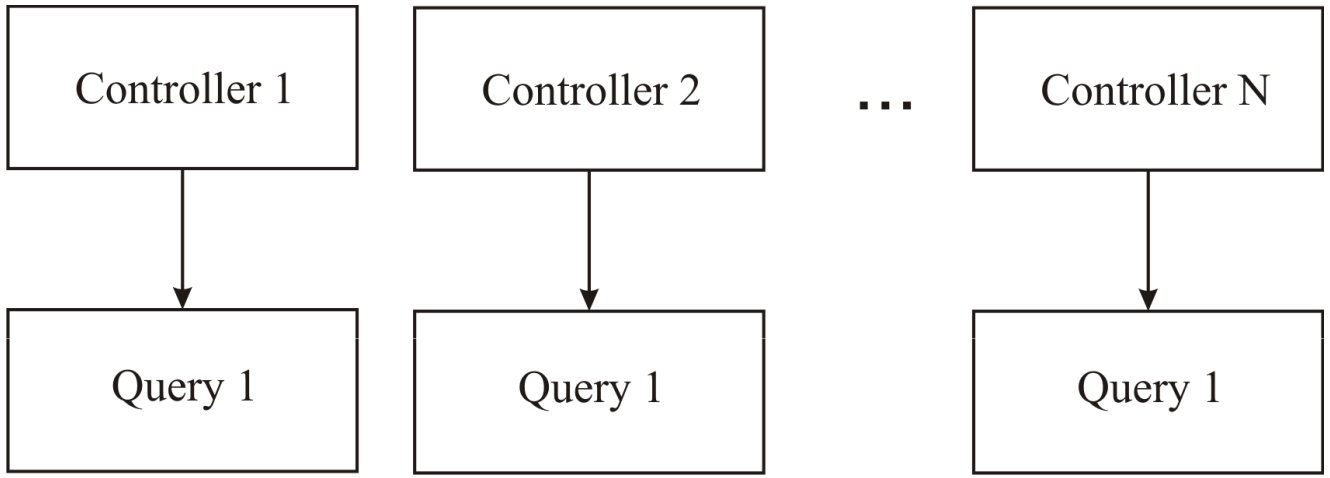
```
$repository = $this->getDoctrine()->getRepository('AdibTestBundle:Task');
$query = $repository->createQueryBuilder('t')
        ->where('t.priority > :priority')
        ->setParameter('priority', '10')
        ->orderBy('t.priority', 'ASC')
        ->getQuery();
$tasks = $query->getResult();
```

Para saber más

http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/query-builder.html

Doctrine

Crear nuestras propias Clases Repositorio



La consulta “Query 1” tendríamos que repetirla N veces, una en cada controlador…

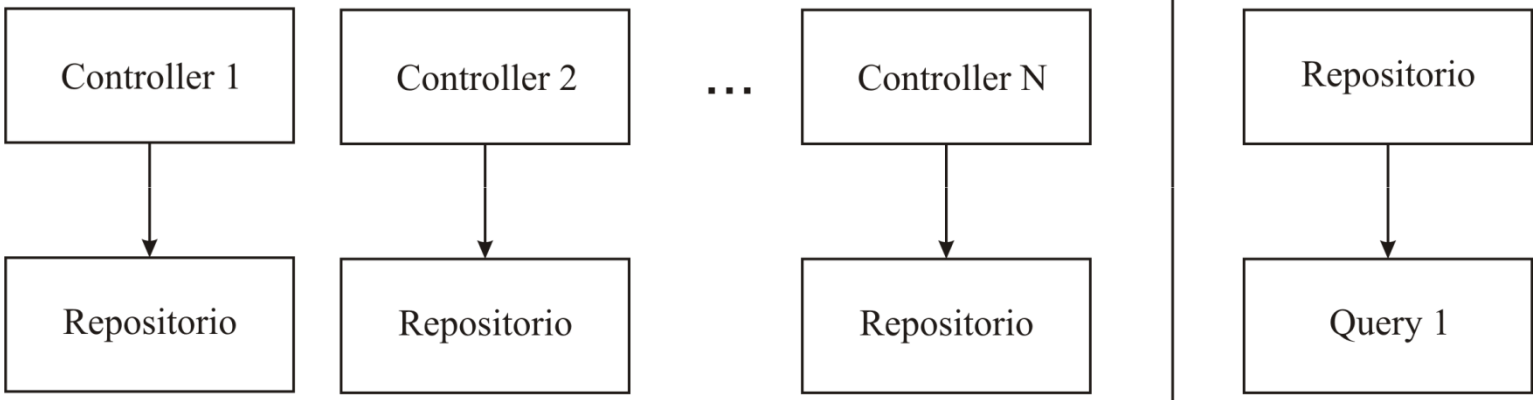
¿Y si hay que aplicar algún cambio a “Query 1”?

Realizamos los cambios N veces acabando con dolor de dedos o …

Doctrine

Crear nuestras propias Clases Repositorio

¡Creamos una clase Repositorio!



Los controladores llaman al repositorio que es quien implementa la consulta.

Si queremos cambiar la consulta sólo se cambia en el repositorio

Doctrine

Crear nuestras propias Clases Repositorio

- Nos ayuda a separar las consultas SQL del código del controlador
- Permite poder reusar las consultas

Añadimos una nueva anotación en la clase Task

```
/**
 * @ORM\Entity(repositoryClass="Adib\TestBundle\Entity\TaskRepository")
 * @ORM\Table(name="task")
 */
class Task {
…
}
```

Volvemos a dejar que Doctrine haga el trabajo:

```
>php app/console doctrine:generate:entities Adib
```

Doctrine

Crear nuestras propias Clases Repositorio

Doctrine ha generado el fichero *src/Adib/TestBundle/Entity/TaskRepository.php*

Todas las consultas relacionadas con Task irán en esta clase.

Extiende de EntityRepository por lo que tendremos disponibles todas las funciones de dicha clase.

Por ejemplo, crear un método findAllOrderedByTitle

```
class TaskRepository extends EntityRepository
{
        public function findAllOrderedByTitle() {
                return $this->getEntityManager()->createQuery('SELECT t FROM
AdibTestBundle:Task t ORDER BY t.title ASC')->getResult();
        }
}
```

Doctrine

Crear nuestras propias Clases Repositorio

Para acceder al repositorio creado desde un controlador

```
$em = $this->getDoctrine()->getManager();
$tasks = $em->getRepository('AdibTestBundle:Task')->findAllOrderedByTitle();
```

Al ser una clase repositorio podemos acceder a todos los métodos predefinidos como *find()* o *findAll()*

Doctrine

Relaciones y asociaciones entre entidades

Nuevo esquema: Una Tarea puede tener una etiqueta (tag), una etiqueta puede estar en varias Tareas

Doctrine puede generar la case *Tag* por nosotros ☺

```
>php app/console doctrine:generate:entity --entity="AdibTestBundle:Tag" --fields="name:string(255)“
```

Mediante este comando, pueden generarse clases simples, podríamos haber generado nuestra clase Task

```
>php app/console doctrine:generate:entity --entity="AdibTestBundle:Task" --fields="title:string(255) description:text priority:integer“
```

NOTA: Es necesario añadir las anotaciones para indicar el nombre de la tabla y la clase repositorio:

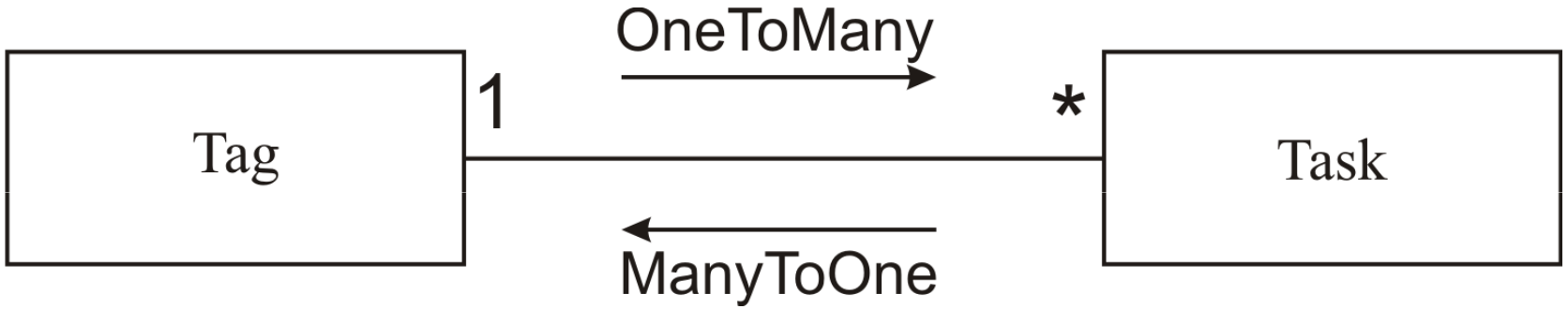
“`@ORM\Table(name="task")`” y

”`@ORM\Entity(repositoryClass="Adib\TestBundle\Entity\TaskRepository")`”

Doctrine

Relaciones y asociaciones entre entidades

Nuevo esquema:



Hay que tenerlo en cuenta para crear el mapeado correctamente

Doctrine

Relaciones y asociaciones entre entidades

Un Tag estará relacionado con varias Tareas, hay que indicarle a Doctrine esta nueva situación de alguna manera.

La clase Tag contendrá un array de Tareas a la que añadiremos información de mapeado.

Es necesario crear el constructor de la clase para inicializar el array tasks ya que Doctrine no lo hace (es bueno, pero no tanto :-P)

Doctrine

Relaciones y asociaciones entre entidades

```
// clase Tag
use Doctrine\Common\Collections\ArrayCollection;
class Tag {
    // ...
    /**
     * @ORM\OneToMany(targetEntity="Task", mappedBy="tag")
     */
    protected $tasks;

    public function __construct() {
        $this->tasks = new ArrayCollection();
    }
}
```

En targetEntity hemos escrito Task ya que Tag pertenece al mismo namespace, si perteneciese a otro namespace o incluso otro bundle, habría que indicar la ruta completa.

Doctrine

Relaciones y asociaciones entre entidades

En nuestra clase Task también tenemos que añadir información de mapeado

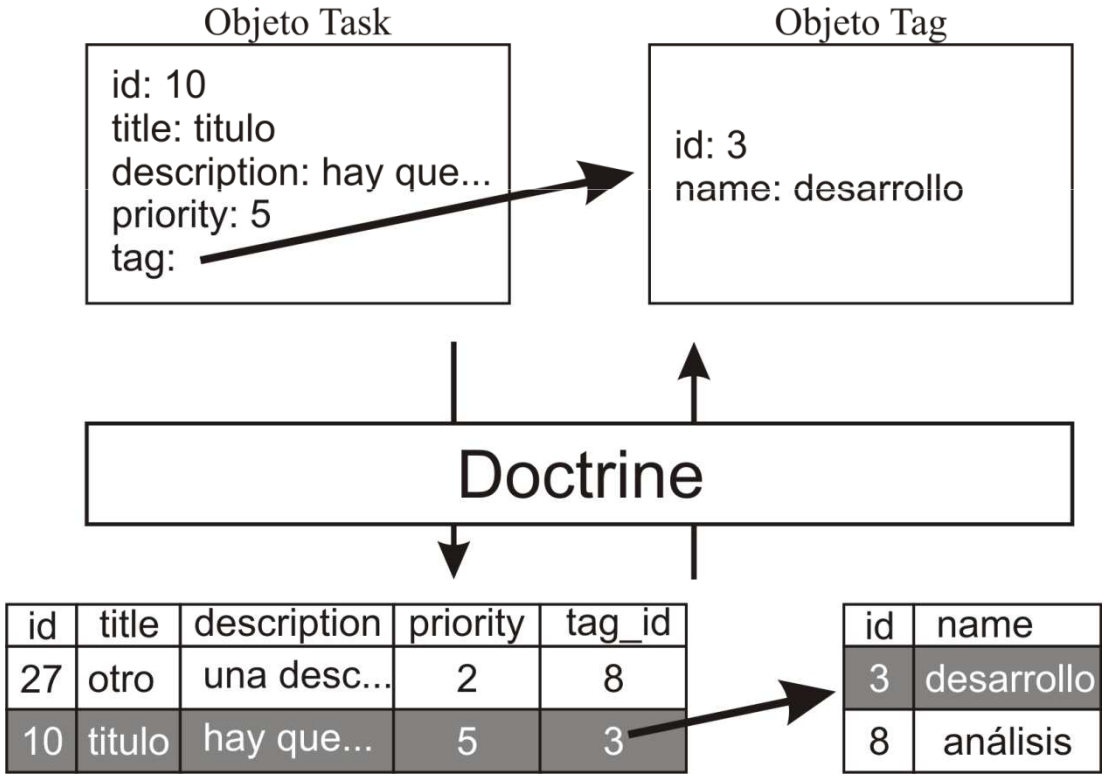
```
class Task {
        // ...
        /**
         * @ORM\ManyToOne(targetEntity="Tag", inversedBy="tasks")
         * @ORM\JoinColumn(name="tag_id", referencedColumnName="id")
         */
        protected $tag;

        // ...
}
```

Indicamos a Doctrine que la clase relacionada es *Tag*, de la que vamos a guardar el id en un campo llamado *tag_id* que estará en la tabla *Task*

Doctrine

Relaciones y asociaciones entre entidades

A nivel de objeto *$tag* será un objeto *Tag*, pero a nivel de tablas Doctrine guardará el *id* del *Tag* en la columna *tag_id* de la tabla *Task*



Doctrine

Relaciones y asociaciones entre entidades

Ahora que está todo claro…generamos los getters y setters nuevos y actualizamos el esquema de la base de datos. (nosotros no, ¡Doctrine! ;-))

```
>php app/console doctrine:generate:entities Adib

>php app/console doctrine:schema:update --force
```

Doctrine

Relaciones y asociaciones entre entidades. Operaciones con los objetos

Guardar objetos relacionados.

```
public function createAction(){
        $tag = new Tag();
        $tag->setName("desarrollo");

        $task = new Task();
        $task->setTitle("Implementar clase symfony");
        $task->setDescription("Descripcion de la tarea");
        $task->setPriority(5);
        // Relacionamos la tarea con la etiqueta
        $task->setTag($tag);

        $em = $this->getDoctrine()->getManager();
        $em->persist($tag);
        $em->persist($task);
        $em->flush();

        return new Response("Creada la tarea con id ".$task->getId()." con el tag id
".$tag->getId());
    }
```

Doctrine

Relaciones y asociaciones entre entidades. Operaciones con los objetos

Extraer objetos relacionados.

Al igual que antes obtenemos el objeto *Task* usando su repositorio, una vez tenemos el objeto podemos acceder al objeto *Tag* mediante el getter

```
public function showAction($id){
    $task = $this->getDoctrine()->getRepository('AdibTestBundle:Task')->find($id);
    $tag = $task->getTag();
    $tagName = ($tag == null)?"No tag":$task->getTag()->getName();
    // ...
    return new Response("Encontrada la tarea con id ".$id."<br>Titulo: ".$task-
>getTitle()."<br>Descripcion: ".$task->getDescription()."<br>Tag: ".$tagName);
}
```

El objeto *Tag* no es cargado hasta que se “pregunta” por él. Se carga “perezosamente”

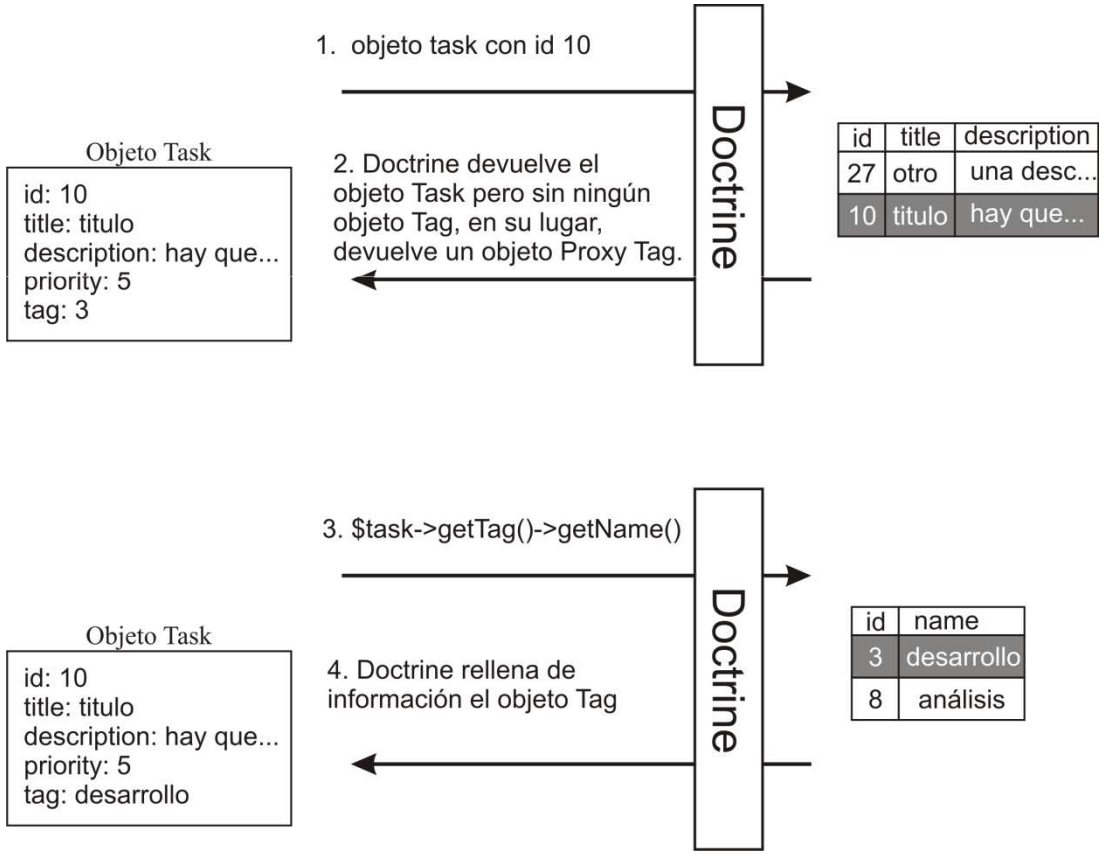
(lazily loaded).

Esta operación es transparente a nosotros. Como siempre, Doctrine se encarga ☺

Doctrine

Relaciones y asociaciones entre entidades. Operaciones con los objetos

Extraer objetos relacionados. “Carga perezosa”



Doctrine

Relaciones y asociaciones entre entidades. Operaciones con los objetos

Extraer objetos relacionados. “Carga perezosa”

- Un objeto “cargado perezosamente”, realmente es un objeto Proxy.
- Es exactamente igual que el objeto real al que representa.
- Mientras no tengamos que acceder a ninguna de sus propiedades el objeto real no será cargado.
- Las clases proxy son creadas por Doctrine para no tener que usar consultas innecesarias.
- Estas clases son creadas en el directorio cache

Doctrine

Relaciones y asociaciones entre entidades. Operaciones con los objetos

Extraer objetos relacionados.

Tambien podemos extraer un objeto *Tag* con todos sus objetos *Task*

```
public function showTagAction($id){
        $tag = $this->getDoctrine()->getRepository('AdibTestBundle:Tag')->find($id);

        if (!$tag) {
            throw $this->createNotFoundException('No se encontró ninguna tag con id
'.$id);
        }

        $tasks = $tag->getTasks();
        $str = "";
        foreach ($tasks as $task) {
            $str .= "Titulo: ".$task->getTitle()."<br>Descripcion: ".$task-
>getDescription()."<br>Prioridad: ".$task->getPriority()."<br>--<br>";
        }
        return new Response("Encontrada tag id ".$id."<br />Nombre: ".$tag-
>getName()."<br>Tareas asociadas: <br><br>".$str);
}
```

Doctrine

Relaciones y asociaciones entre entidades. Operaciones con los objetos

Uniendo objetos relacionados.

Al traernos objetos relacionados, si vamos a necesitar ambos objetos, los pasos anteriores ejecutarían dos consultas.

Podemos hacer una única consulta y traernos ambos objetos.

Haremos uso del repositorio del objeto *Task (Adib/TestBundle/Entity/TaskRepository.php)*

```
public function findOneByIdJoinedToTag($id) {
     $query = $this->getEntityManager()->createQuery('SELECT t, tg FROM
     AdibTestBundle:Task t JOIN t.tag tg WHERE t.id = :id')->setParameter('id', $id);
     try {
          return $query->getSingleResult();
     } catch ( \Doctrine\ORM\NoResultException $e) {
          return null;
     }
}
```

Doctrine

Relaciones y asociaciones entre entidades. Operaciones con los objetos

Uniendo objetos relacionados.

Ahora podemos usar la función para cargar desde la base de datos un objeto *Task* con su objeto *Tag* relacionado.

…

```
$task = $this->getDoctrine()->getRepository('AdibTestBundle:Task')-
>findOneByIdJoinedToTag($id);
$tag = $task->getTag();
```

…

Doctrine

Relaciones y asociaciones entre entidades.

Hemos visto las asociaciones OneToMany y ManyToOne

Existen otros tipos como las OneToOne o ManyToMany.

Para más información sobre las Asociaciones

http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html

NOTA: Al usar anotaciones, siempre tenemos que usar ORM delante de los parámetros (ej. ORM\OneToMany), ya que en la documentación de Doctrine no aparece. No olvidar usar la instrucción "`use Doctrine\ORM\Mapping as ORM;`*"*

Doctrine

Lifecycle Callbacks (Llamadas durante el ciclo de vida)

Acciones que queremos ejecutar antes o despues de una inserción, actualización o eliminación de una entidad de la base de datos.

Al usar anotaciones, debemos activar las “lifecycle callbacks”. No sería necesario si usamos YAML o XML

```
/**
 * @ORM\Entity()
 * @ORM\HasLifecycleCallbacks()
 */
class Task {
    // ...
}
```

Doctrine

Lifecycle Callbacks (Llamadas durante el ciclo de vida)

Ahora podemos hacer que Doctrine ejecute un método cuando ocurra alguno de los eventos que hay disponibles.

Vamos a añadir dos atributos nuevos a nuestra clase *Task* que recojan la fecha de creación y de finalización de la tarea.

Serán *dueDate* y *created*, ambos de tipo “datetime”.

Haremos que el valor de *created* se asigne de forma automática

Doctrine

Lifecycle Callbacks (Llamadas durante el ciclo de vida)

Podemos definir un método que asigne la fecha de creación antes de guardar el objeto en la base de datos la primera vez.

```
/**
 * @ORM\PrePersist
 */
public function setCreatedValue() {
    $this->created = new \DateTime();
}
```

Los métodos deben ser simples, de ahí que no lleven argumentos, para métodos más complicados existen otras alternativas:

http://symfony.com/doc/current/cookbook/doctrine/event_listeners_subscribers.html

Doctrine

Lifecycle Callbacks (Llamadas durante el ciclo de vida)

Existen otros eventos que pueden llamarse:

- *preRemove*
- *postRemove*
- *prePersist*
- *postPersist*
- *preUpdate*
- *postUpdate*
- *postLoad*
- *LoadClassMetadata*

Más información:

http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/events.html#lifecycle-events

Doctrine

¿Aún con ganas de más?

Existen varias extensiones para facilitar aun más el trabajo con Doctrine:

http://symfony.com/doc/current/cookbook/doctrine/common_extensions.html

Además para conocer con más detalle todos los tipos de campos disponibles

http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#doctrine-mapping-types

Cada campo tiene un conjunto de opciones que pueden aplicarse. Además de *type*, podemos especificar *name, length, unique* y *nullable*.

```
/**
 * @ORM\Column(name="email_address", unique=true, length=150)
 */
protected $email;
```

http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#property-mapping

Formularios

Es una de las tareas más comunes en desarrollo web

Symfony integra el componente *Form* el cual nos ayuda a trabajar con formularios de una forma fácil.

Este componente es una librería externa y puede usarse de forma independiente de Symfony.

Al igual que con Doctrine, la mejor forma de entenderlo es verlo en acción ☺

Formularios

Vamos a crear un formulario para poder crear las tareas. Creamos un método *newAction* en nuestro *DefaultController.* (No olvidemos crear la ruta)

```
use Symfony\Component\HttpFoundation\Request;
...
public function newAction(Request $request) {
        // Creamos una Tarea y le damos algunos valores
        $task = new Task();
        $task->setTitle('Escribir un post');
        $task->setDescription('Tengo que escribir un post en mi blog');
        $task->setPriority(5);
        $task->setDueDate(new \DateTime('tomorrow'));
        $form = $this->createFormBuilder($task)->add('title', 'text')-
>add('dueDate', 'date')->getForm();
        return $this->render('AdibTestBundle:Default:new.html.twig', array( 'form'
=> $form->createView(),));
}
```

Formularios

En el ejemplo, hemos creado un formulario que sólo renderiza dos campos del objeto *Task*

Para cada campo hemos indicado el tipo, de forma que el *FormBuilder* sepa que elemento html debe mostrar.

Para finalizar, es necesario crear la página que será mostrada al finalizar la acción, en este caso: *src/Adib/TestBundle/Resources/views/Default/new.html.twig*

```
<form action="{{ path('adib_new_task') }}" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}
    <input type="submit" />
</form>
```

En este caso la ruta creada es “adib_new_task” con el pattern */new/task*

Formularios

Para mostrar el formulario tan sólo tenemos que llamar a form_widget() .

Además los datos insertados en el objeto *Task* se muestran en los campos del formulario.

El sistema de formularios accede de forma automática a los métodos *setters* y *getters* de nuestro objeto.

Para un atributo de tipo boolean podemos usar métodos *issers (Ej. isPublished())*
A partir de la versión 2.1 de Symfony, también puede acceder a los métodos *hassers (Ej. hasReminder())*

Todos los métodos son necesarios ya que los atributos de nuestro objeto los hemos marcado como “protected” en lugar de “public” y por tanto sólo pueden ser accedidos mediante métodos.

Formularios

Manejo de los datos enviados

La segunda tarea de un formulario es recoger los datos introducidos por el usuario y asignarlos al objeto en cuestión.

Hay que realizar una correspondencia (*binding*) entre los datos del formulario y el objeto.

Emplearemos el método *bind()* (sustituye a *bindRequest ()* a partir de Symfony 2.1) para dicho propósito.

El método *bind()* asigna los datos del formulario al objeto **independientemente de que sean válidos o no**.

Formularios

Manejo de los datos enviados

```
use Symfony\Component\HttpFoundation\Request;

public function newAction(Request $request) {
     // Creamos un objeto Task limpio
     $task = new Task();
     // Creamos el formulario
     $form = $this->createFormBuilder($task)->add('title', 'text')->add('dueDate',
     'date')->getForm();

     if ($request->isMethod('POST')) {
          $form->bind($request);
          if ($form->isValid()) {
                    // realizar alguna acción (Ej. Guardar en la base de datos)
                    $em = $this->getDoctrine()->getManager();
                    $em->persist($task);
                    $em->flush();
                    return $this->redirect($this->generateUrl('task_success'));
          }
     }
     // ...
}
```

Vamos a probar nuestro formulario…¿Qué ocurre?

Formularios

Manejo de los datos enviados

El formulario creado no asigna todos los valores a nuestro objeto Task, ya que solo renderizamos el título y la fecha de realización de la tarea. Al intentar guardarlo en la base de datos no se permiten valores *null* para description y priority.

Solución:

1. Renderizamos todos los campos necesarios para asignar a nuestro objeto

```
$form = $this->createFormBuilder($task)->add('title', 'text')->add('description',
'textarea')->add('priority', 'integer')->add('dueDate', 'date')->getForm();
```

2. Podemos indicar a Doctrine que la descripción y la prioridad pueden ser null

```
/**
 * @Column(type="text",  nullable=true)
 */
protected $description;
```

Probemos la primera solución… ¿Todavía falla ☹?. Aún sigue faltando algo…

Formularios

Manejo de los datos enviados

¡La ruta!. No hemos creado ninguna ruta llamada *task_success*

```
return $this->redirect($this->generateUrl('task_success'));
```

Solucion:

1. Crear la ruta… :-P
2. Redirigir la página a una ruta que ya tenemos

```
$this->redirect($this->generateUrl('adib_show_task',array("id"=> $task-
>getId())));
```

En este caso se mostraría la tarea recien creada

Formularios

Manejo de los datos enviados

```
public function newAction(Request $request) {
    $response = null;
    $task = new Task(); // Creamos una Tarea limpia, sin atributos

    // Se crea el formulario con todos los campos necesarios
    $form = $this->createFormBuilder($task)->add('title', 'text')->add('description',
'textarea')->add('priority', 'integer')->add('dueDate', 'date')->getForm();

    if($request->isMethod("POST")){
          $form->bind($request); // Se asignan los valores
          if($form->isValid()){
              // Realizamos alguna acción, Ej. Guardar en base de datos
              $em = $this->getDoctrine()->getManager();
              $em->persist($task);
              $em->flush();
              $response = $this->redirect($this->generateUrl('adib_show_task',
array("id"=> $task->getId())));
           }
     }
      if($response == null){ {
        $response = $this->render('AdibTestBundle:Default:new.html.twig', array('form' =>
        $form->createView(),));
    }
    return $response;
}
```

Formularios

Manejo de los datos enviados

Cuando un controlador maneja formularios, pueden darse tres casos:

• Al cargar la página inicialmente, la petición es de tipo GET y el formulario se crea y se visualiza.

• Cuando el usuario envía el formulario (a través de POST) con **datos no válidos**, **se realiza la correspondencia** (bind) entre el formulario y el objeto y se vuelve a renderizar mostrando los errores de validación.

• Cuando el usuario envía el formulario con **datos correctos, se realiza el *bind*** del formulario con el objeto, que queda a nuestra disposición para realizar cualquier acción (Ej. guardarlo en la base de datos) y finalmente redirigir al usuario a otra página.

Formularios

Validación

La validación se aplica a nivel de objeto

El método *$form->isValid()* es una forma directa de “preguntar” al objeto si sus datos son válidos una vez hecho el *bind()* con el formulario.

La validación se realiza añadiendo una serie de reglas (restricciones) a una clase.

Formularios

Validación

Vamos a añadir algunas restricciones a nuestra clase Task:

Creamos el fichero *Adib/TestBundle/Resources/config/validation.yml*

```
Adib\TestBundle\Entity\Task:
    properties:
            title:
                    - NotBlank: ~
            dueDate:
                    - NotBlank: ~
                    - Type: \DateTime
            priority:
                    - NotBlank: ~
                    - Type: integer
```

NOTA: Algunos navegadores con soporte para HTML5 realizan validación automática de campos en blanco, para evitarlo debemos añadir el parámetro "novalidate" al formulario en el fichero de plantilla "twig":

```
<form novalidate  …>
```

Formularios

Validación

En el código, podemos acceder al servicio de validación (aunque no es lo habitual):

```
$task = new Task();
// ... Añadir datos al objeto Task
$validator = $this->get('validator');
$errors = $validator->validate($task);
if (count($errors) > 0) {
        return new Response(print_r($errors, true));
} else {
        return new Response('¡La tarea es válida!');
}
```

Formularios

Validación

El validador “valida” un objeto usando ***constraints*** ,“*restricciones”* en español, que no son más que reglas que deben cumplirse.

Mapea una o varias reglas con la clase correspondiente y comprueba si se cumplen o no dichas reglas.

Una restricción es realmente un objeto PHP que realiza sentencias afirmativas para comprobar que una regla se cumple.

Dado un valor, una restricción nos indicará si cumple alguna de las reglas que contiene dicha restricción.

Formularios

Validación

Symfony2 contiene un gran número de las restricciones que más se suelen usar:

Restricciones básicas

`NotBlank, Blank, NotNull ,Null, True, False, Type`

Restricciones para String

`Email, Length, Url, Regex, Ip`

Restricciones para Numeros

`Range`

Restricciones de Fechas

`Date, DateTime, Time`

Restricciones para colecciones (Collection)

`Choice, Collection, Count, UniqueEntity, Language, Locale, Country`

Restricciones para Ficheros

`File, Image`

Restricciones Financieras

`CardScheme, Luhn`

Otras restricciones

`Callback, All, UserPassword, Valid`

Además podemos definir nuestras propias restricciones, pero eso…es otra historia ☺

http://symfony.com/doc/current/cookbook/validation/custom_constraint.html

Formularios

Validación

Algunas restricciones pueden ser mas complejas, por ejemplo, “Choice”:

```
properties:
    priority:
        - Choice: {choices: [1,2,3,4,5], message: Choose a valid priority [1-5].}
```

En general, todas las restricciones aceptan parámetros en forma de array, sin embargo, también permiten pasar el valor de una opción por defecto en lugar del array

```
properties:
    priority:
        - Choice: [1,2,3,4,5]
```

Esto hace la configuración de las restricciones más sencilla y rápida.

Si no estamos seguros de cómo configurar una restricción con opción por defecto, siempre podemos recurrir a la documentación de la API.

http://symfony.com/doc/current/reference/constraints.html

Formularios

Validación

Las restricciones pueden aplicarse a ***propiedades (atributos)*** de nuestra clase o a cualquier método ***getter*** público que tengamos , también existen algunas que pueden aplicarse a la ***clase*** completa.

Propiedades

Es el más común y fácil de usar. Podemos validar propiedades públicas, privadas o protegidas (para las privadas y protegidas son necesarios métodos para acceder a ellas)

```
properties:
   title:
      - NotBlank: ~
```

Formularios

Validación

Getters

Nos permite usar reglas de validación más complejas.

Podemos añadir validación a cualquier método público que comience por “get” o “is”, los llamados “getters”.

Por ejemplo, evitar que una tarea tenga el título y la descripción iguales.

```
getters:
    titleEqualdescription:
      - "False": {message: "The title and description should be different!!"}
```

Debemos crear el método “*isTitleEqualDescription*” en nuestra clase *Task*

```
public function isTitleEqualdescription(){
    return strcmp($this->title, $this->description) == 0;
}
```

Formularios

Validación

Getters

Como puede verse, al mapear la restricción con el método correspondiente, no se ha indicado el nombre del método completo, se ha omitido “get” o “is”.

Con esto conseguimos poder reutilizar nuestra restricción en cualquier otro método que se llame igual o cualquier propiedad sin necesidad de cambiar nuestra lógica de validación.

Clases

Existen algunas restricciones que se aplican a nivel de *clase*, por ejemplo *Callback*. Cuando es validada se ejecutan los métodos especificados en dicha restricción, por lo que nos da mucha flexibilidad para poder implementar nuestra propia validación.

Formularios

Validación

Hasta ahora hemos visto como añadir validación a una clase y comprobar si pasa o no todas las restricciones definidas…¿y si sólo queremos validar algunas restricciones y no todas?

Para eso existen los Grupos de Validación (Validation Groups)

- Organizamos las restricciones por grupos
- Ejecutamos en cada caso el grupo que nos interese

Veamos como funciona a través de un ejemplo

Formularios

Validación

Ahora para crear una tarea, basta con indicar el título y fecha de finalización, pero no es necesario indicar ni la prioridad ni la descripción. Ambas pueden editarse posteriormente.

```
Adib\TestBundle\Entity\Task:
  properties:
    title:
      - NotBlank: {groups: [creation, edition]}
    dueDate:
      - NotBlank: {groups: [creation, edition]}
      - Type: {type: \DateTime, groups: [creation, edition]}
    priority:
      - NotBlank: {groups: [edition]}
      - Type: {type: integer, groups: [edition]}
      - Choice: {choices: [1,2,3,4,5], message: Choose a valid priority [1-
5]., groups: [edition]}
  getters:
    titleEqualdescription:
      - "False": {message: "The title and description should be different!!",
groups: [edition]}
```

Formularios

Validación

Se han creado dos grupos de restricciones, “creation” y “edition”, correspondientes a la creación y a la edición de una Tarea respectivamente.

Aquellas restricciones a las que no se les asigna ningún grupo , pertenecen de forma automática al grupo *Default*

Ahora hay que llamar al validador correspondiente en nuestro controlador:

```
$form = $this->createFormBuilder($task, array('validation_groups'=>
array('creation'),))->add('title', 'text‘)->add('dueDate', 'date‘)->getForm();
```

Si lo probamos falla…¿Por qué?

Formularios

Validación

Aunque hemos quitado los campos de descripción y prioridad del formulario, cuando se realiza el *bind* se les asigna null , ¿Qué hay que hacer para permitir valores null en la base de datos?. Ya lo hemos visto anteriormente

Si recordamos, tenemos que asignar la propiedad *nullable=true* a la columna deseada dentro de la clase *Task*, en este caso *description* y *priority*

```
/**
 * @ORM\Column(type="text", nullable=true)
 */
protected $description;
```

Una vez configurados description y priority debemos actualizar las tablas de la base de datos para que permitan valores null

```
>php app/console doctrine:schema:update --force
```

Formularios

Validación

Las Tareas ahora se guardan sin descripción ni prioridad…Vamos a crear un formulario para editar tareas.

- Creamos una nueva ruta que nos lleve a la acción de editar una Tarea

```
adib_edit_task:
  pattern: /edit/task/{id}
  defaults: {_controller: AdibTestBundle:Default:edit }
```

- Además tenemos que crear una nueva vista para el formulario de edición

```
<form novalidate action="{{ path('adib_edit_task',{"id":id}) }}" method="post" {{
form_enctype(form) }}>
    {{ form_widget(form) }}
    <input type="submit" />
</form>
```

Formularios

Validación

- Creamos una nueva acción en nuestro controlador por defecto

```
public function editAction(Request $request, $id) {
     $response = null;
     $task = $this->getDoctrine()->getRepository('AdibTestBundle:Task')->find($id);
     if (!$task) {
          throw $this->createNotFoundException('No se encontró ninguna tarea con id '.$id);
     }
     $form = $this->createFormBuilder($task, array('validation_groups'=>array('edition'),))->add('title',
     'text')->add('description', 'textarea')->add('priority', 'integer')->add('dueDate', 'date')->getForm();
     // Si el formulario ha sido enviado por el usuario
     if($request->isMethod("POST")){
           // Se asignan los valores
           $form->bind($request);
           if($form->isValid()){
                 // Guardamos la tarea editada
                 $em = $this->getDoctrine()->getManager();
                 $em->flush();
                 $response = $this->redirect($this->generateUrl('adib_show_task',array("id"=> $task-
                 >getId())));
          }
     }
     if($response == null){
          $response = $this->render('AdibTestBundle:Default:edit.html.twig', array('form' => $form-
     >createView(),'id' => $task->getId(),));
     }
     return $response;
}
```

Formularios

Validación

Si necesitamos añadir algún tipo de lógica para elegir el grupo de validación que se activa, podemos realizar una llamada a una función externa en dos vías (*Los ejemplos se implementan en el método setDefaultOptions de la clase Formulario que veremos más adelante*):

1. Llamamos a una función externa de una clase

```
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver) {
    $resolver->setDefaults(array('validation_groups'=> array('Acme\\AcmeBundle\\Entity\\Client',
'determineValidationGroups'), ));
}
```

2. Implementar la función directamente

```
use Symfony\Component\Form\FormInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver) {
      $resolver->setDefaults(array('validation_groups' => function(FormInterface $form) {
                  $data = $form->getData();
                  if (Entity\Client::TYPE_PERSON == $data->getType()) {
                              return array('person');
                  } else {
                              return array('company');
                  }
      },));
}
```

Formularios

Validación

Existe también la posibilidad de validar campos individuales usando los objetos constraint de symfony.

```
use Symfony\Component\Validator\Constraints\Email;
...
$email  = “unemail@depruebas.com”;
$emailConstraint = new Email();
// Las opciones de cada restriccion pueden asignarse como propiedades del objeto
$emailConstraint->message = 'Invalid email address';
// Usamos el validador para comprobar el valor de la variable $email
$errorList = $this->get('validator')->validateValue($email, $emailConstraint);
// Comprobamos si hubo errores...
if (count($errorList) != 0) {
    $errorMessage = $errorList[0]->getMessage();
    // ... Lo que sea con el error...
}
```

Formularios

Tipos de Campos

Symfony implementa una serie de campos de formularios predefinidos

Campos de Texto
`text, textarea, email, integer, money, number, password, percent, search, url`

Campos de selección
`choice, entity, country, language, locale, timezone`

Capos de Fecha y Tiempo
`date, datetime, time, birthday`

Otros campos
`checkbox, file, radio`

Campos de Grupo
`collection, repeated`

Campos ocultos
`hidden, csrf,`

Campos Base
`field, form`

Podemos crear nuestros propios campos de formulario:

http://symfony.com/doc/current/cookbook/form/create_custom_field_type.html

Formularios

Tipos de Campos

Cada tipo de campo tiene una serie de opciones para poder configurarlo. Estas opciones se le pasan en forma de ***array.*** En nuestro ejemplo:

Renderiza un campo especial fecha formado por tres select.

```
->add('dueDate', 'date')
```

Podemos hacer que sea un campo de texto

```
->add('dueDate', 'date', array('widget' => 'single_text'))
```

El array puede contener varios parámetros que dependerán del campo en cuestión.

```
->add('dueDate', 'date', array('widget' => 'single_text', 'label' => 'Fecha fin',))
```

Renderiza el campo fecha con el label "*Fecha fin*" en lugar de "*Duedate*"

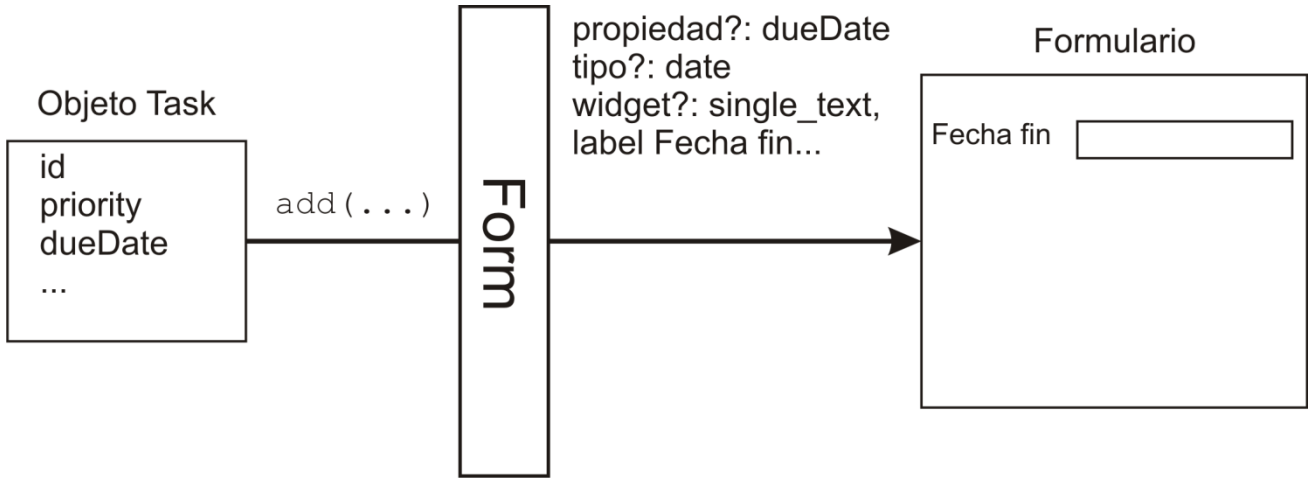
Formularios

Tipos de Campos

Como vemos el método add tiene tres parámetros de entrada correspondientes a:

1. nombre de la **propiedad** de nuestro **objeto**,
2. Tipo de de la **propiedad** de nuestro **objeto**
3. Opciones del **widget** que se mostrará en el **formulario**

```
->add('dueDate', 'date', array('widget' => 'single_text', 'label' => 'Fecha fin',))
```



Formularios

Tipos de Campos

La opción ***required***

Esta opción puede aplicarse a cualquier tipo de campo.

Por defecto es “true” y se emplea para que los exploradores con soporte HTML5 realicen la validación de si el campo está en blanco.

¡Cuidado!, esta opción hace que la **validación** se realice en el **cliente**, **NO** en el **servidor**, por eso si el navegador no soporta HTML5 un valor en blanco puede ser dado por válido a no ser que en el servidor comprobemos el valor con una restricción *NotBlank* o *NotNull*.

En resumen, **¡siempre hay que usar validación en el servidor!**

Formularios

Tipos de Campos

Symfony es capaz de “adivinar” el tipo de campos de nuestro formulario.

Tan sólo debemos omitir el segundo parámetro del método *add* o pasar *null* como argumento.

```
$form = $this->createFormBuilder($task)
->add('title')
->add('dueDate', null, array('widget' => 'single_text'))
->getForm();
```

Del mismo modo, Symfony también puede intentar asignar los valores a las opciones del campo:

- required: La añade basandose en el valor de las restricciones como *NotBlank* o *NotNull* o en los metadatos de Doctrine (*nullable*).
- max_length: Si el campo es de texto, la opción max_length puede “adivinarla” a partir de las restricciones *Length* o *Range* o desde los metadatos de Doctrine (*length*).

Formularios

Tipos de Campos

Si quisiéramos cambiar cualquier opción de las que Symfony ha “adivinado” de forma automática tan sólo tendríamos que sobreescribirla dándole el valor que queramos. (Symfony no siempre tiene que tener razón ☺)

```
->add('title', null, array('max_length' => 4))
```

Formularios

Renderizar formularios en una Plantilla

Hasta ahora hemos visto como renderizar un formulario con una línea de código.

Habitualmente necesitamos más flexibilidad por lo que Symfony nos permite renderizar el formulario en una plantilla twig.

Vamos a cambiar nuestra plantilla *new.html.twig* con el siguiente código:

```
<form novalidate action="{{ path('adib_new_task') }}" method="post" {{
form_enctype(form) }}>
        {{ form_errors(form) }}
        {{ form_row(form.title) }}
        {{ form_row(form.dueDate) }}
        {{ form_rest(form) }}
        <input type="submit" />
</form>
```

Formularios

Renderizar formularios en una Plantilla

Veamos el código por partes:

- `form_enctype(form)` – en caso de subir ficheros, mediante esta función se renderiza la opción enctype="multipart/form-data"
- `form_errors(form)` – Renderiza cualquier error del formulario (los errores específicos de cada campo se muestran al lado)
- `form_row(form.title)` – Renderiza la etiqueta, los errores y el campo HTML.
- `form_rest(form)` – Renderiza cualquier campo que no haya sido renderizado. Es una buena práctica colocarlo al final del formulario (en caso de olvidar renderizar algún campo o no querer renderizar manualmente campos ocultos). Además nos ayuda a protegernos de los ataques CSRF (Cross-site request forgery). Ver descripción en http://symfony.com/doc/current/book/forms.html#forms-csrf

Formularios

Renderizar formularios en una Plantilla

Aunque form_row nos facilita la vida, algunas veces podemos necesitar renderizar cada parte por separado

```
{{ form_errors(form) }}
<div>
        {{ form_label(form.title) }}
        {{ form_errors(form.title) }}
        {{ form_widget(form.title) }}
</div>
<div>
        {{ form_label(form.dueDate) }}
        {{ form_errors(form.dueDate) }}
        {{ form_widget(form.dueDate) }}
</div>
{{ form_rest(form) }}
```

Formularios

Renderizar formularios en una Plantilla

Además podemos especificar la etiqueta si no queremos que la autogenere

```
{{ form_label(form.title, 'Título de la tarea') }}
```

Contamos también con opciones adicionales dependiendo del tipo de cada campo. Por ejemplo la opción *attr* es muy usada para modificar atributos de un elemento:

```
{{ form_widget(form.title, { 'attr': {'class': 'task_field'} }) }}
```

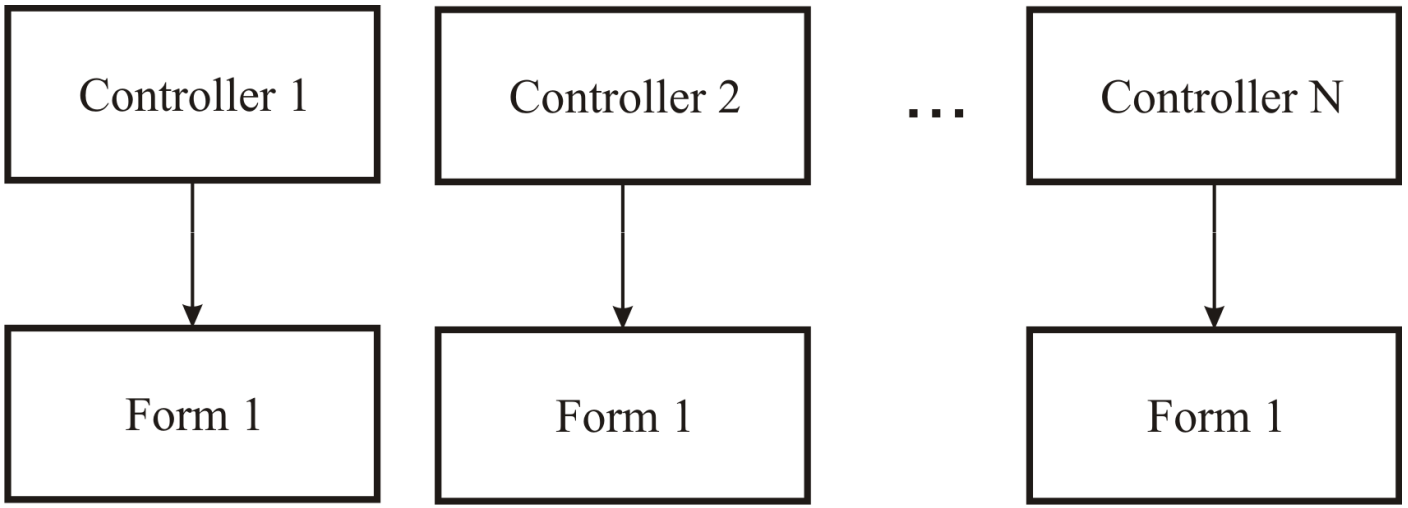
En caso de necesitar renderizar campos a mano, podemos también acceder a valores de los campos. como id, name y label

```
{{ form.title.vars.id }}
```

Formularios

Crear Clases Formulario

Imaginemos un caso parecido al que teníamos con las clases repositorio en Doctrine…

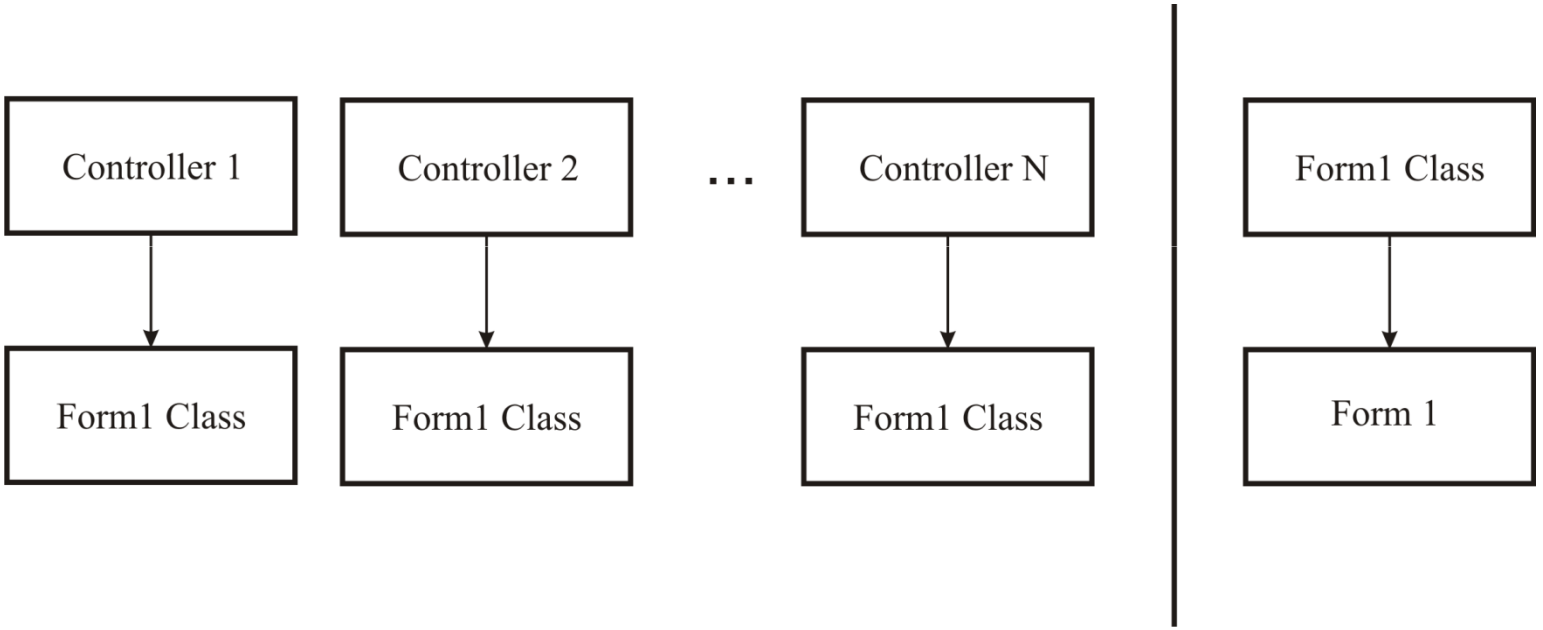


¿No hay ningún remedio para esto en Symfony?

Formularios

Crear Clases Formulario

¡¡Pues claro!!, existen las *Clases Formulario* (Form Classes)



No nos vamos a ir sin probarlo… ☺

Formularios

Crear Clases Formulario

Creamos un nuevo fichero en la ruta *src/Adib/TestBundle/Form/Type/TaskType.php*

```
namespace Adib\TestBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class TaskType extends AbstractType{
    public function buildForm(FormBuilderInterface $builder, array $options){
        $builder->add('title');
        $builder->add('dueDate', null, array('widget' => 'single_text'));
        if($options["validation_groups"][0] == "edition"){
                $builder->add('description','textarea');
                $builder->add('priority');
        }
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver){
        $resolver->setDefaults(array('data_class' => 'Adib\TestBundle\Entity\Task',));
    }

    public function getName(){
        return 'task';
    }
}
```

Formularios

Crear Clases Formulario

Como vemos hay que extender de *AbstractType* e implementar **obligatoriamente** las funciones **buildForm** y **getName** (el **nombre** que devuelve la función **getName** debe ser **único**).

Aunque no es obligatoria, sí es recomendable implementar la función *setDefaultOptions*, en la que asignamos la variable `data_class` la cual le indicará a la clase Form el objeto que va a contener los datos del formulario (en nuestro caso, *Task*).

Observar que a la hora de construir el formulario se comprueba el grupo de validación que se emplea, de esta forma elegimos qué campos se van a renderizar:

```
if($options["validation_groups"][0] == "edition")
```

Formularios

Crear Clases Formulario

Ahora podemos llamar a nuestra clase formulario desde el controlador para crear un objeto *Form* de una forma sencilla y rápida

```
// Hay que añadir una nueva directiva use para usar nuestra clase TaskType
use Adib\TestBundle\Form\Type\TaskType;

public function newAction() {
        $task = ...;
        $form = $this->createForm(new TaskType(),$task,
array('validation_groups' => 'creation'));
        // …
}
```

Como vemos, hay que asignar el grupo de validación que se usará pasándolo como tercer parámetro al método *createForm*, de esta forma se renderizarán sólo los campos necesarios

Formularios

Crear Clases Formulario

Al mapear formularios a objetos, **todos** los campos del formulario son mapeados.

En caso de que el formulario tenga un campo que no existe en nuestro objeto se lanzará una excepción…

¿Y si queremos añadir un campo o no queremos mapear uno de los atributos de nuestro objeto?

Para eso symfony nos ofrece la propiedad "`mapped`", que podemos usar para que un campo no sea mapeado.

Podemos usarlo de dos formas diferentes:

1. Para no mapear campos de nuestro objeto
2. Para renderizar campos en un formulario que no existen en nuestro objeto

Formularios

Crear Clases Formulario

1. Para no mapear campos de nuestro objeto

```
$builder->add('dueDate', null, array('mapped' => false));
```

En este caso al atributo *dueDate* de nuestro objeto se le asignaría **null**

2. Para renderizar campos en un formulario que no existen en nuestro objeto

```
$builder->add('agree', "checkbox", array('mapped' => false));
```

En cualquier caso siempre podemos acceder al campo del formulario

```
$form->get('dueDate')->getData();
$form->get('agree')->getData();
```

Formularios

Formularios embebidos

Un formulario puede contener campos que correspondan a varios objetos.

¿Qué ocurre si queremos asignar nuestro objeto *Tag* al objeto *Task* mediante formulario?

Afortunadamente esto es normal para Symfony ☺

Los pasos a seguir serán:

1. Crear nuestra *Clase Formulario* para el objeto *Tag*
2. Añadir un campo nuevo en nuestra clase *TaskType* para que renderize el formulario creado para *Tag*
3. Añadir la opción `cascade_validation` en *TaskType* para que valide también los campos de *Tag*

¡Nada de agobios!...que ahora lo vemos paso a paso…;-)

Formularios

Formularios embebidos

1. Crear nuestra *Clase Formulario* para el objeto *Tag*

Creamos el fichero *TagType.php* de la misma forma que creamos *TaskType.php*

```
class CategoryType extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options) {
        $builder->add('name');
    }
    ...
}
```

Faltan las funciones `setDefaultOptions` y `getName` que son análogas a las de la clase *TaskType* creada anteriormente. En `setDefaultOptions` hay que indicar el parámetro `data_class` y para la función `getName` recordar usar un identificador único (ej. `"tag"`).

Formularios

Formularios embebidos

2. Añadir un campo nuevo en nuestra clase *TaskType* para que renderize el formulario creado para *Tag*

```
public function buildForm(FormBuilderInterface $builder, array $options) {
    // ...
    $builder->add('tag', new TagType());
    if($options["validation_groups"][0] == "edition"){
         //...
}
```

En *TaskType* diferenciamos entre dos tipos de formularios, el de inserción y el de edición, el nuevo campo *tag* podemos ponerlo donde queramos (dentro o fuera del **if**) dependiendo de donde queremos que se asigne dicho valor.

¿Que pasaría si ponemos el campo dentro del **if**?...no lo probaremos aquí, pero cada uno es libre de probarlo ☺

Formularios

Formularios embebidos

1. Añadir la opción `cascade_validation` en *TaskType* para que valide también los campos de *Tag*

```
public function setDefaultOptions(OptionsResolverInterface $resolver){
    $resolver->setDefaults(array(
        'data_class' => 'Adib\TestBundle\Entity\Task',
        'cascade_validation' => true,));
}
```

Con esto indicamos a nuestro formulario para *Task*, que también tiene que validar los "subobjetos" que contenga, en este caso el objeto *Tag* que hemos añadido al formulario.

Pues ya podemos probar nuestro nuevo formulario…¿Que ocurre?

Formularios

Formularios embebidos

¡Un fallo espantoso!...¿Que esto?...está controlado ☺

Simplemente al intentar guardar en la base de datos nuestro objeto Task, Doctrine comprueba que existe un objeto Tag que no ha sido marcado para persistir y no sabe que hacer…este Doctrine…hay que decirselo todo…;-)

Tenemos dos opciones:

1. Llamar a `$em->persist($task->getTag());` antes de llamar a *flush()*
2. Añadir una anotación en el atributo Tag de nuestra clase Task para indicarle que las operaciones de persist las haga en cascada.

```
@ORM\ManyToOne(targetEntity="Tag", inversedBy="tasks",cascade={"persist"})
```

Puede usarse una u otra, o ambas a la vez sin problemas.

Con esto ya no falla, a no ser que intentemos dejar en blanco el campo Tag. ¿Qué falta?

Formularios

Formularios embebidos

Pues algo importante, ¡las reglas de validación para Tag!

Añadimos una nueva entrada en nuestro fichero de validación *“validation.yml”:*

```
Adib\TestBundle\Entity\Tag:
  properties:
    name:
      - NotBlank: {groups: [creation]}
```

Ahora si intentamos añadir un Tag en blanco nos aparecerá el error correspondiente.

Formularios

Formularios embebidos

Como podemos comprobar por cada texto que introducimos en el campo *Tag*, se crea una nueva entrada en la tabla *Tag*…¿y si quisiéramos elegir una de las Tags existentes?

Nada más fácil gracias a Symfony ☺

En nuestra clase TaskType cambiamos la definición del campo Tag por esta:

```
$builder->add('tag','entity',array('class'=>'Adib\TestBundle\Entity\Tag',
'property' => 'name'));
```

Ahora se nos presenta un campo select para elegir entre las Tags que ya existen en la base de datos…¿fácil verdad?

Formularios

Formularios embebidos

Ahora vamos a ver el caso del formulario para una *Tag*

Para crear una Tag sólo necesitaríamos su nombre.

¿Y si queremos editar una *Tag* existente con todas sus *Tasks*?.

En symfony podemos embeber una colección de formularios dentro de un mismo formulario.

Creamos dos nuevas rutas que apunten a dos nuevas funciones que crearemos en nuestro controlador. La primera para añadir nuevas Tags y la segunda para editarlas.

```
adib_new_tag:
  pattern: /new/tag
  defaults: {_controller: AdibTestBundle:Default:newTag }

adib_edit_tag:
  pattern: /edit/tag/{id}
  defaults: {_controller: AdibTestBundle:Default:editTag }
```

Formularios

Formularios embebidos

Creamos las funciones en cuestión:

```
public function newTagAction(Request $request)
public function editTagAction(Request $request, $id)
```

Estas funciones son análogas a las que tenemos para Task pero usando el objeto Tag en su lugar y llamando a nuestra clase formulario *TagType*.

Además debemos crear dos nuevas plantillas twig a las que llamaremos en cada función.

Estas plantillas serán: *newTag.html.twig* y *editTag.html.twig*

Formularios

Formularios embebidos

En nuestra clase *TagType* es necesario renderizar el campo `tasks`, para ello añadimos la siguiente línea dentro del método `buildForm`:

```
...
$builder->add('name', null);
$builder->add('tasks','collection',array('type' => new TaskType()));
...
```

En este caso le indicamos que vamos a renderizar la propiedad `tasks` del objeto `Tag`. Además le indicamos que es de tipo `collection`, es decir, un array de objetos `Tasks`.

Por último es necesario indicarle también de qué tipo son los objetos de dicha `collection`…en nuestro caso de tipo formulario de Tarea, es decir, `TaskType`

Formularios

Formularios embebidos

Para el caso de *newTag.html.twig* podemos usar la función *form_widget()* para renderizar el formulario directamente.

Para el caso de *editTag.html.twig* usamos el siguiente codigo:

```
<form novalidate action="{{ path('adib_edit_tag',{"id":id}) }}" method="post"
{{ form_enctype(form) }}>
    {{ form_row(form.name) }}
    <h3>Tasks</h3>
        {# itera sobre cada objeto task y renderiza el formulario asociado #}
        {% for task in form.tasks %}
        {{ form_widget(task) }}
        ----------------------------------------------
        <br />
        {% endfor %}

    {{ form_rest(form) }}
    <input type="submit" />
</form>
```

Formularios

Formularios sin un objeto

Hasta ahora hemos enlazado nuestro objeto con un formulario

El FormBuilder enlaza nuestro objeto con los campos del formulario de modo que puede obtener y asignar los valores de dicho objeto.

¿Y si queremos construir un formulario sin usar un objeto?

De hecho, por defecto la clase *Form* siempre asume que estamos trabajando con array de datos y somos nosotros los que indicamos que queremos trabajar con objetos.

1. Pasando el objeto al crear el formulario (como primer argumento de *createFormBuilder* o como segundo argumento de *createForm*)
2. Declarando el parámetro `data_class` en nuestro formulario.

Sin ninguna de estas opciones, el formulario trabaja sin enlazarse con objetos ☺

Formularios

Formularios sin un objeto

Ejemplo. Crear un formulario de contacto. (¡No olvidar crear la plantilla twig y la ruta!)

```
public function contactAction(Request $request) {
     $response = null;
     $defaultData = array('message' => 'Type your message here');
     $form = $this->createFormBuilder($defaultData)->add('name', 'text')
          ->add('email', 'email')
          ->add('message', 'textarea')
          ->getForm();
     if($request->isMethod('POST')) {
          $form->bind($request);
          if($form->isValid()){
              // data es un array con las claves "name", "email" y "message“
              $data = $form->getData();
              $response = new Response("Enviada información de ".$data['email']);
          }
     }
     if($response == null){
          $response = $this->render('AdibTestBundle:Default:contact.html.twig',
          array('form' => $form->createView(),));
     }
     return $response;
}
```

Formularios

Formularios sin un objeto

¿Qué ocurre si dejamos los campos en blanco?

¡Falta la validación!, en este caso debemos añadirla a mano usando los objetos

“constraint” que nos ofrece Symfony.

```
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\Email;
// ...
$form = $this->createFormBuilder($defaultData)->add('name', 'text',array(
'constraints' => array(
                    new NotBlank(),
                    new Length(array('min'=> 3))
)))->add('email', 'email', array('constraints' => new Email()))
          ->add('message', 'textarea')
          ->getForm();
```

Tambien podemos usar grupos de validación

```
new NotBlank(array('groups' => array('create', 'update'))
```

Formularios

¿Deseosos de más?

Subir ficheros con doctrine

http://symfony.com/doc/current/cookbook/doctrine/file_uploads.html

Información sobre formularios embebidos:

http://symfony.com/doc/current/cookbook/form/form_collections.html

Información sobre los Tipos de campos de Form:

http://symfony.com/doc/current/reference/forms/types.html

Modificar formularios dinámicamente usando eventos:

http://symfony.com/doc/current/cookbook/form/dynamic_form_modification.html

Capítulo del libro sobre validación

http://symfony.com/doc/current/book/validation.html

Capítulo del libro sobre formularios

http://symfony.com/doc/current/book/forms.html

Internacionalización

Proceso de abstraer los mensajes de nuestra aplicación a una capa que se encargue de traducirlos al “locale” (región) del usuario

“locale “ se refiere no sólo al lenguaje, también al país. Se recomienda su representación según el código ISO639-1 del lenguaje, un guión bajo “_” y el código ISO3166 Alpha-2 del país.

Que no de miedo…para nosotros se resume en “es_ES” (español_España)

Ejemplo de uso:

```
// Este código siempre se mostrará en inglés
echo 'Hello World';
// Este otro se traducirá al lenguaje del usuario o por defecto al inglés
echo $translator->trans('Hello World');
```

Internacionalización

El proceso se divide en cuatro pasos:

1. Activar y configurar el componente "*Translation*" de Symfony
2. Abstraer los "strings" mediante llamadas al "*Translator"*
3. Crear las traducciones de los idiomas que queramos
4. Determinar, asignar y gestionar la configuración regional del usuario para una determinada petición (Request) o para toda la sesión.

Internacionalización

1. Activar y configurar el componente "*Translation*" de Symfony

Fichero *app/config/config.yml*

```
framework:
        translator: { fallback: %locale% }
```

La opción fallback le indica a Symfony la region por defecto en caso de **NO** encontrar ninguna traducción para el idioma actual.

Si dejamos %locale% usará el que esté configurado en el fichero *app/config/parameters.yml,* pero podemos forzar el que queramos (Ej. "`fallback: en`" o "`fallback: es`"…)

Internacionalización

2. Abstraer los “strings” mediante llamadas al “*Translator*”

Para traducir mensajes usamos el servicio “*Translator*” (recordemos que los servicios pueden llamarse con “`$this->`” gracias a extender de controller).

Hay que llamar al método “*trans()*” de dicho servicio.

```
$t = $this->get('translator')->trans('Symfony2 is great');
```

Symfony2 intentará traducir el mensaje basándose en la configuración reginal “locale” del usuario.

¿Por arte de magia?... Nooo, hay que proporcionarle un “diccionario” o “catálogo” ☺

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Los ficheros de traducciones deben llamarse "*<domain>.<idioma>.<loader>*".

Por **defecto**, Symfony buscará ficheros llamados **messages**.<idioma>.<loader> (donde *idioma* puede ser es, en, fr, etc…y *loader* yml, php o xliff).

Como vemos, los ficheros pueden escribirse en yml, php o xliff (XML Localization Interchange File Format). Symfony2 recomienda el uso de xliff

En nuestro caso, usaremos el formato yml (en mi opinión es más entendible y más cómodo).

Para los escépticos, vamos a ver un ejemplo en formato yml y en xliff...

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Ejemplo en yml:

```
Symfony2 is great: Symfony2 es genial!
```

Ejemplo en xliff:

```
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
      <file source-language="en" datatype="plaintext" original="file.ext">
            <body>
                  <trans-unit id="1">
                        <source>Symfony2 is great</source>
                        <target>Symfony2 es genial!</target>
                  </trans-unit>
            </body>
      </file>
</xliff>
```

¿Convencidos ya por el yml ? ☺

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Symfony buscará por una traducción en tres niveles diferentes:

1. app/Resources/translations
2. app/Resources/NombreDeNuestroBundle/translations
3. Resources/translations de nuestro bundle

Los ficheros están ordenados de mayor a menor prioridad, es decir, las traducciones de *app/Resources/translations* son de mayor prioridad (sustituyen) a todas las de niveles inferiores. Las de *app/Resources/NombreDeNuestroBundle/translations*, a las de nuestro bundle. (No hace falta que siga… ☺*)*

Internacionalización

3. Crear las traducciones de los idiomas que queramos

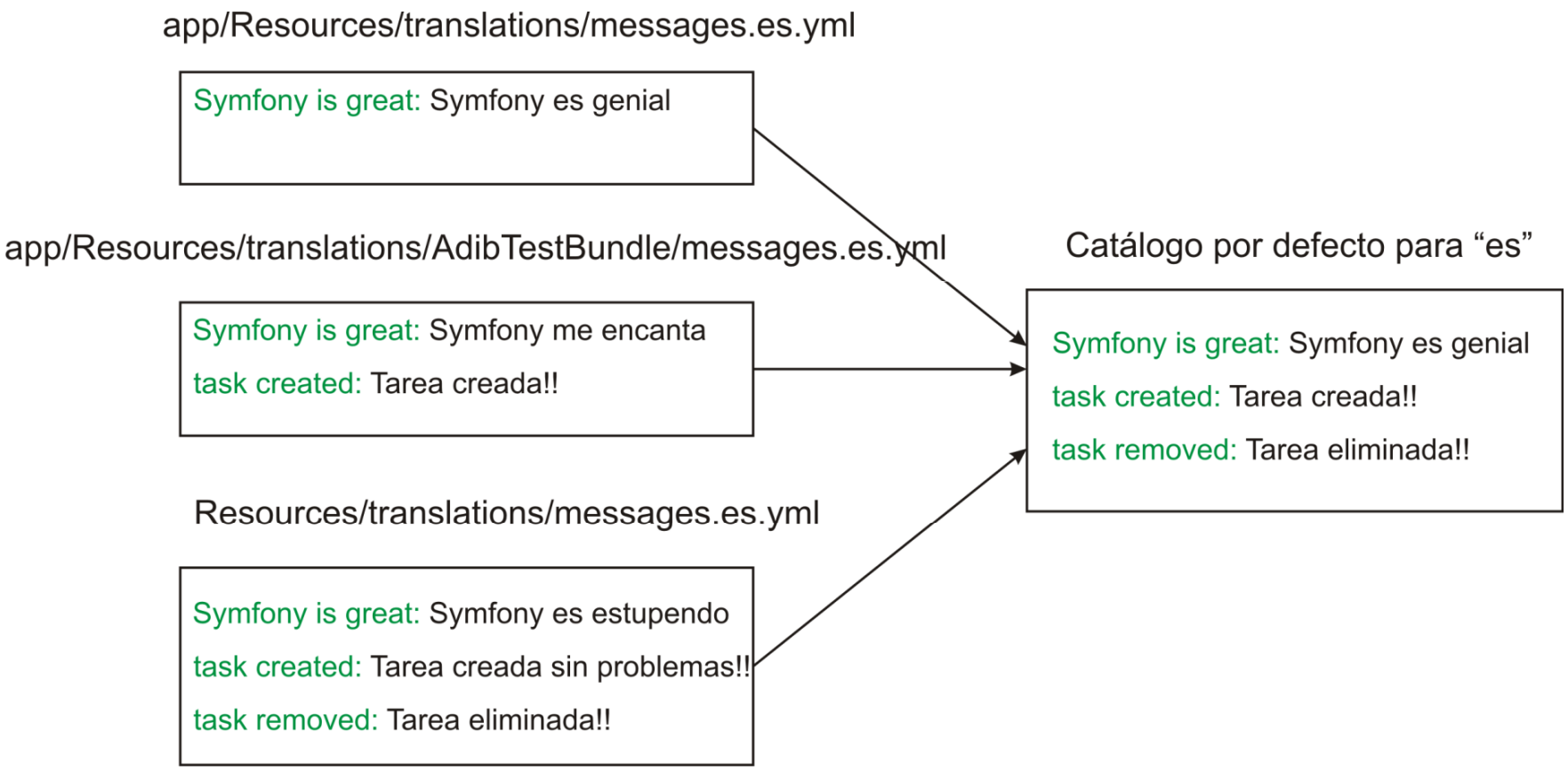
Realmente Symfony construye un gran catálogo de traducciones para cada idioma y para cada dominio (*domain*) recorriendo todos los ficheros de traducciones que se hayan creado**.**

Para cada traducción encontrada, siempre elige aquella del nivel más prioritario. En caso de no encontrar ninguna traducción, devolvería el texto original.

A continuación se explica con una imagen (dicen que vale más que 1000 palabras :-P)

Internacionalización

3. Crear las traducciones de los idiomas que queramos



Internacionalización

3. Crear las traducciones de los idiomas que queramos

Como vemos, el dominio de todos los ficheros es ***messages***, que es el **dominio por defecto**

Podemos usar otros dominios tan sólo creando el fichero con el nombre deseado y luego pasándolo como parámetro al *Translator*.

Ej. Imaginemos que tenemos el fichero *errors.es.yml* con el siguiente mensaje de error.

```
No task found: No se encontró la tarea
```

Para usar el traductor habría que indicarle el dominio “errors”, ya que es el nombre del fichero.

```
$this->get('translator')->trans('No task found', array(), 'errors');
```

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Existe otro dominio especial que es el de los ficheros de validación, para traducir estos mensajes es necesario crear el fichero "***validators***.<idioma>.<loader>" .

Si en nuestro fichero "*validation.yml*" tenemos:

```
Adib\TestBundle\Entity\Task:
  properties:
    title:
      - NotBlank: {groups: [creation, edition], message: task.title.not_blank}
```

En el fichero "*validators.es.yml*" tendríamos:

```
task.title.not_blank: Por favor, indique un título para la tarea
```

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Podemos usar variables en las traducciones (*placeholders*)

```
$t = $this->get('translator')->trans('Hello %name%', array('%name%'=> $name));
```

En el fichero de traduccion tan sólo hemos de indicar el "*placeholder*":

```
Hello %name%: Hola %name%
```

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Symfony permite usar mensajes reales o *keywords* para las traducciones:

```
$t = $translator->trans('Symfony2 is great');
$t = $translator->trans('symfony2.great');
```

En el primer método los mensajes se escriben en el idioma por defecto que se quiera (inglés en este caso), y luego se usa como "keyword" en los ficheros de traducción.

Ventaja:

- No es necesario crear un fichero con el idioma por defecto

Inconveniente:

- Si cambiamos la frase (ej. "Symfony2 is really great"), habría que cambiarlo en todos los ficheros de traducción

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Symfony permite usar mensajes reales o *keywords* para las traducciones:

```
$t = $translator->trans('Symfony2 is great');
$t = $translator->trans('symfony2.great');
```

En el segundo se usa una “keyword” que se aproxime a lo que el mensaje quiere decir.

Ventaja:

- La “keyword” no cambia nunca en ningún fichero

Inconveniente:

- Hay que crear el fichero para el idioma por defecto.

Cada uno es libre de usar el método que quiera, pero Symfony recomienda este último ☺

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Los ficheros de traducción en formato yml o php soportan ids anidadas para no tener que repetir texto en las “keywords”:

```
symfony2:
    is:
        great: Symfony2 is great
        amazing: Symfony2 is amazing
    has:
        bundles: Symfony2 has bundles
user:
    login: Login
```

Los múltiples niveles podríamos ponerlos en una línea separando cada nivel con un (.):

```
symfony2.is.great: Symfony2 is great
symfony2.is.amazing: Symfony2 is amazing
symfony2.has.bundles: Symfony2 has bundles
user.login: Login
```

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Pluralización de los mensajes

Si tenemos dos traducciones distintas debido a los plurales, se separaran ambas por "**|**"

```
apples: 'Hay una manzana|Hay %count% manzanas'
```

Para usar los plurales usamos el método `transChoice()` en lugar de `trans()`:

```
$this->get('translator')->transChoice("apples",1)
$this->get('translator')->transChoice("apples",10, array("%count%" => 10))
```

El segundo parámetro siempre es obligatorio y le indicará al traductor si debe usar el plural o el singular.

El tercer parámetro es un array con los valores que debe sustituir en nuestro texto al traducirlo

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Pluralización de los mensajes

El traductor tiene una serie de reglas por defecto para cada idioma.

En general para el español y el inglés:

- usará la primera parte de nuestro texto para el singular cuando el segundo argumento sea 1 (*Hay una manzana*)
- usará la segunda parte de nuestro texto para el plural cuando dicho argumento sea distinto de 1 (0,2,3…) (*Hay 0 manzanas*)

Cada idioma tiene sus reglas por defecto para los plurales (algunos llegan a tener ¡6 reglas para plurales!...afortunadamente no es nuestro caso ☺)

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Pluralización de los mensajes

Decir “Hay 0 manzanas” no es muy lógico ¿verdad?

¿Tendrá algo symfony2 para remediar esto…?,

¡Pues claro!, podemos especificar nuestros propios intervalos en las traducciones.

Numeros exactos:

```
{1,2,3,4}
```

Intervalos abiertos o cerrados usando “[” o “]” dependiendo del caso

```
]-1,2[
[2,19]
```

Incluso podemos usar infinito para dichos intervalos

```
[1,+Inf[
]-Inf,0]
```

Internacionalización

3. Crear las traducciones de los idiomas que queramos

Pluralización de los mensajes

El traductor comprobará si el número indicado encaja en algún intervalo y en caso positivo, lo usará, sino, podemos hacer que use las reglas por defecto del idioma actual.

En este caso están cubiertos todos los intervalos desde 0 hasta infinito:

```
apples: '{0} No hay manzanas|{1} Hay una manzana|]1,19] Hay %count%
manzanas|[20,Inf] Hay un viaje de manzanas'
```

En este otro caso al usar 1 o cualquier número entre 2 y 19, el traductor usaría los mensajes por defecto. Si no indicamos estos mensajes no traduciría nada…

```
apples: '{0} No hay manzanas|[20,Inf] Hay un viaje de manzanas|Hay una
manzana|Hay %count% manzanas'
```

Internacionalización

4. Determinar, asignar y gestionar la configuración regional del usuario para una determinada petición (Request) o para toda la sesión.

La configuración regional (locale) del usuario se guarda en la petición (request).

A través del objeto Request podemos manejar el *locale* del usuario

```
$request = $this->getRequest();
$locale = $request->getLocale();
$request->setLocale('es_ES');
```

También es posible usar la sesion para guardar el *locale* mientras dure la misma

```
$this->get('session')->set('_locale', 'es_ES')
```

Si el *locale* no ha sido asignado en la sesión se selecciona el configurado como *fallback.*

Aun así es posible garantizar que siempre haya un locale por defecto mediante el parámetro `default_locale` que se configura en el fichero *app/config/config.yml*

```
framework:
    default_locale: es
```

Internacionalización

4. Determinar, asignar y gestionar la configuración regional del usuario para una determinada petición (Request) o para toda la sesión.

Locale en la URL

Alguien podría verse “tentado” a usar el *locale* guardado en la sesión para mostrar una misma página en varios idiomas (ej. http://www.example.com/contact).

¡Esto **NO** debe hacerse!, uno de los fundamentos de la web es que una URL siempre devuelve el mismo recurso.

Para estos casos podemos usar como ya vimos, el sistema de rutas, usando el parámetro especial “_locale”:

```
contact:
    path: /{_locale}/contact
    defaults: { _controller: AcmeDemoBundle:Contact:index, _locale: en }
    requirements:
         _locale: en|fr|de
```

Internacionalización

¿Ávidos de más conocimientos?

Existen opciones para usar las traducciones interactuando con bases de datos…pero eso no lo veremos ☹

Guardar traducciones en bases de datos:

http://api.symfony.com/2.2/Symfony/Component/Translation/Loader/LoaderInterface.html

Traducir el contenido de las bases de datos:

https://github.com/l3pp4rd/DoctrineExtensions

Traducciones en las plantillas:

http://symfony.com/doc/2.0/book/translation.html#translations-in-templates

Capítulo del libro referente a las traducciones:

http://symfony.com/doc/current/book/translation.html

Plantillas

¿Que es una plantilla?

Un fichero de texto que puede generar ficheros de diferentes formatos (HTML, XML, CSV, LaTeX ...). La más conocida es la plantilla *PHP.*

Symfony sin embargo, incorpora un lenguaje más potente para generar plantillas de forma más concisa, legible y amigables para los diseñadores web…¡Twig! (http://twig.sensiolabs.org/)

```
<!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>
```

Plantillas

Twig define tres tipos de sintaxis:

- `{{ ... }}`: “Dice algo”, muestra una variable o el resultado de una expresión.
- `{% ... %}`: “Hace algo”, ejecuta instrucciones como por ejemplo bucles “`for`”.
- `{# ... #}`: Permite escribir comentarios que no serán interpretados por Twig.

Twig contiene filtros que modifican el contenido mostrado .

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

Además por defecto implementa una larga lista de tags y filtros a los que podemos añadir los nuestros propios.

Otra de las ventajas de Twig es que soporta funciones e incluso podemos añadir nuevas.

Plantillas

Herencia de plantillas y disposición (Layout)

Generalmente las páginas comparten elementos comunes como la cabecera, el pie, barras de opciones…

Twig soluciona esto facilmente mediante la herencia.

Se construye una plantilla con la disposición base (layout) y se definen los elementos comunes como “bloques”

Cualquier plantilla que herede de la principal contará con estos bloques que podrá sobreescribir si se requiere.

Plantillas

Herencia de plantillas y disposición (Layout)

Construyamos un layout base:

Generalemente para una aplicación se crea en "*app/Resources/views/base.html.twig*"

Otra opción sería crear un layout base sólo para nuestro *bundle en* "*src/Resources/views*"

En cualquier caso, al crear una plantilla "hija" tendríamos que importar la plantilla creada.

Crearemos tres bloques (title, sidebar y body).

Plantillas

Herencia de plantillas y disposición (Layout)

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>{% block title %}Test Application{% endblock %}</title>
</head>
<body>
    <div id="sidebar">
        {% block sidebar %}
            <ul>
                <li><a href="{{ path("adib_new_task") }}">Nueva tarea</a></li>
            </ul>
        {% endblock %}
    </div>
    <div id="content">
        {% block body %}{% endblock %}
    </div>
</body>
</html>
```

Plantillas

Herencia de plantillas y disposición (Layout)

Ahora vamos a crear una plantilla hija que herede de la plantilla base y que muestre la lista de tareas. (En este caso omitiremos tanto la creación del Action en nuestro controller como la ruta).

```
{% extends '::base.html.twig' %}
{% block title %}Mi lista de tareas{% endblock %}
{% block body %}
        {% for task in tasklist %}
                <h2>{{ task.title }}</h2>
                <p>{{ task.description }}</p>
        {% endfor %}
{% endblock %}
```

Mediante "**extends**" indicamos que esta plantilla hereda de la plantilla base.

Al no usar el bloque "**sidebar**", se usará el de plantilla padre

Plantillas

Herencia de plantillas y disposición (Layout)

Cosas a tener en cuenta para trabajar con herencia:

• La tag "`{% extends %}`" siempre debe ser la primera en la plantilla hija

• Cuanto más bloques "`{% block %}`" tenga nuestra plantilla base mejor, esto nos da mayor flexibilidad ya que las plantillas hijas no tienen por qué definir todos los bloques.

• Si tenemos contenido repetido en varias plantillas debemos moverlo a un "`{% block %}`" en la plantilla base. En otros casos podemos hacer uso de "`include`" (ya lo veremos)

• Si necesitamos contenido de la plantilla base podemos usarlo llamando a la función `{{ parent() }}`. Es muy útil cuando no queremos sobreescribir por completo un bloque:

```
{% block sidebar %}
   <h3>Table of Contents</h3>
   {# Lo que queramos añadir... #}
   {{ parent() }}
{% endblock %}
```

Plantillas

Nombres y ubicación de las plantillas

Por defecto se crean en dos ubicaciones:

- *app/Resources/views/*: Son plantillas a nivel de toda la aplicación
- *nuestroBundle/Resources/views/*: Son plantillas a nivel de nuestro bundle

Symfony2 usa la sintaxis "***bundle:controller:template***" para las plantillas:

- ***AdibTestBundle:Default:new.html.twig****:* La plantilla estaría en Resources/views/Default dentro de nuestro bundle
- ***AdibTestBundle::new.html.twig****:* Al no indicar ningún controller,l a plantilla estaría en Resources/views/ dentro de nuestro bundle
- ***::new.html.twig****:* Al no indicar ni bundle ni controller, la plantilla estaría en app/Resources/views/

NOTA: *A igualdad de nombres, la plantillas en app sobreescriben a las de nuestro bundle*

Plantillas

Nombres y ubicación de las plantillas

Los nombres de los archivos de plantilla deben especificar tanto el formato de salida como el motor de plantillas a usar (recordemos que puede ser PHP o Twig)

- `index.html.twig` formato HTML, motor Twig
- `index.html.php` formato HTML, motor PHP
- `index.css.twig` formato CSS, motor Twig

La extensión que especifica el **motor** es **obligatoria**.

La extensión que especifica el fichero de **salida** aunque no es obligatoria sí es **recomendable**.

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Symfony2 incluye muchas etiquetas Twig y funciones que facilitan la tarea de diseño.

Algunos ejemplos que ya hemos visto son `{% block %}` y `{% extends %}`

Vamos a ver algunas de las tareas más comunes en el uso de plantillas:

- Incluir otras plantillas
- Enlazar con otras páginas
- Incluir imágenes

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Incluir otras plantillas

Se suele usar cuando tenemos un fragmento de plantilla que va a repetirse en varias partes de nuestro código (Regla básica de programación).

Ejemplo:

Vamos a crear una plantilla para mostrar una tarea. Creamos el archivo

`Resources/views/Default/task.html.twig`

```
<h2>{{ task.title }}</h2>
<h4>Tag: {{ task.tag.name }}</h4>
<div>creada: {{ task.created |date('Y-m-d-H:i') }}</div>
<div>finaliza: {{ task.dueDate |date('Y-m-d-H:i') }}</div>
<h3>Descripción:</h3>
<p>
{{ task.description }}
</p>
<h4>prioridad: {{ task.priority }}</h4>
```

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Incluir otras plantillas

Podemos hacer uso de la plantilla en nuestro controlador en la función showAction.

Modificaremos la plantilla creada anteriormente para mostrar la lista de tareas:

```
{% extends '::base.html.twig' %}
{% block title %}Mi lista de tareas{% endblock %}
{% block body %}
        {% for task in tasklist %}
          {% include 'AdibTestBundle:Default:task.html.twig' with {'task': task} %}
        {% endfor %}
{% endblock %}
```

Hemos sustituido el interior del bucle **`for`**, por un **`include`** de la plantilla que muestra los detalles de una tarea.

En Symfony 2.2 (Eclipse incluye la 2.1), el tag **`{% include %}`** ha pasado a ser una función:

```
{{ include('AdibTestBundle:Default:task.html.twig',{'task': task}) }}
```

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Incluir llamadas a Controladores

En las plantillas, podemos usar llamadas a controladores para obtener una determinada salida.
Ejemplo. Tenemos una función en Controller que lista las `$max` tareas mas recientes…

```
public function recentTasksAction($max = 3){
// ...
// Extrae de la base de datos las $max tareas mas recientes
$tasks = ...;
// Renderiza una plantilla para mostrarlas
return $this->render('AdibTestBundle:Default:recentList.html.twig', array('tasks' =>
$tasks) );
```

La función en taskRepository podría ser algo parecido a esta:

```
$this->getEntityManager()->createQuery("SELECT t FROM AdibTestBundle:Task t WHERE
t.title like '%' ORDER BY t.created ASC")->setMaxResults($max)->getResult();
```

La plantilla podría ser algo parecido a esto:

```
{% for task in tasks %}
    <a href="{{ path("adib_show_task",{"id":task.id}) }}">{{ task.title }}</a>
{% endfor %}
```

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Incluir llamadas a Controladores

Ahora por ejemplo desde la plantilla base podemos hacer que nuestra “sidebar” muestre la lista de las tres tareas más recientes:

```
<div id="sidebar">
    {% render 'AdibTestBundle:Default:recentTasks' with {'max':3} %}
</div>
```

NOTA: En Symfony 2.2 se han creado las funciones `render()` y `controller()`

```
{{ render(controller('AdibTestBundle:Default:recentTasks', {'max':3})) }}
```

- A la función controller podemos pasarle parámetros, en este caso max = 3
- Las llamadas a controladores es una buena práctica desde el punto de vista de la organización y reutilización del código.
- Además permite a las plantillas acceder a información de la que no tiene por que ser conocedora

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Contenido asíncrono con “*hinclude.js*”

Tanto el contenido de los controladores como cualquier url pueden ser cargadas asíncronamente en nuestras plantillas usando la librería javascript “hinclude.js”.

(http://mnot.github.io/hinclude/)

Nos descargamos la librería y la copiamos en “*web/js*”.

Añadimos la siguiente cabecera en todas aquellas páginas que vayan a hacer uso, en nuestro caso lo añadimos en nuestra plantilla base.

```
<html xmlns:hx="http://purl.org/NET/hinclude">
<head>
    ...
    <script src="{{ asset('js/hinclude.js') }}" type="text/javascript"></script>
</head>
```

¡Con esto ya podemos cargar contenido asíncronamente! ☺

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Contenido asíncrono con “*hinclude.js*”

En nuestro caso, al usar Symfony 2.1, tendríamos dos formas de realizar llamadas:

- Carga de una url:

```
{% render url('...') with {}, {'standalone': 'js'} %}
```

- Carga de un Controlador:

```
{% render path("adib_recent_tasks",{'max':3}) with {},{'standalone':'js'} %}
```

Tanto para urls como para controladores siempre debemos indicar `{'standalone': 'js'}`

En el caso de los controladores usamos la función `path()`pasandole como argumentos una de nuestras rutas y los argumentos que hagan falta.

Prueba. Si añadimos un sleep(5) en la función `recentTasksAction()`...¿Qué ocurre?

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Contenido asíncrono con “*hinclude.js*”

Podemos asignar un contenido por defecto mientras se cargan las páginas:

```
framework:
    # ...
    templating: { engines: ['twig'], hinclude_default_template:
AdibTestBundle:Default:hincludeDefault.html.twig }
```

En Symfony 2.2 se ha cambiado la forma de usar “hinclude.js” para llamadas a los

controladores :

Tendríamos que modificar el fichero *app/config/config.yml*

```
framework:
          # ...
          fragments: { path: /_fragment }
```

Esto nos permite usar las llamadas a controladores usando la función `render_hinclude()`

```
{{ render_hinclude(controller('...'), {'default':
'AdibTestBundle:Default:hincludeDefault.html.twig'}) }}

{{ render_hinclude(controller('...'), {'default': 'Loading...'}) }}
```

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Generar vínculos

Usamos las funciones *path()* o *url()* para referirnos a las rutas guardadas en *routing.yml*.

De esta forma cualquier cambio en una ruta sólo se hace en dicho fichero y no en el código.

path() genera rutas relativas y *url()* rutas absolutas, por lo demás funcionan exactamente igual.

```
<a href="{{ path('adib_new_task') }}">Nueva tarea</a>

<a href="{{ url('adib_new_task') }}">Nueva tarea</a>
```

Tambien podemos pasar parámetros si son necesarios en nuestras rutas

```
<a href="{{ path('adib_show_task',{'id':23}) }}">Ver tarea 23</a>

<a href="{{ url('adib_show_task',{'id':23}) }}">Ver tarea 23</a>
```

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Enlazar recursos (assets)

Para enlazar imágenes, css o javascripts en nuestras plantillas, disponemos de la función *asset()*.

Esta función hace nuestra aplicación más portable ya que genera las rutas de forma relativa a donde se encuentre nuestra aplicación.

```
<img src="{{ asset('images/logo.png') }}" alt="Symfony!" />
<link href="{{ asset('css/blog.css') }}" rel="stylesheet" type="text/css" />
```

Si nuestra aplicación estuviese en *paula.uhu.es* esta función devolvería `paula.uhu.es/images/logo.png`, sin embargo si estuviese en *paula.uhu.es/my_app* entonces devolvería `paula.uhu.es/my_app/images/logo.png`

¡Nuestras páginas serán más portables! ☺

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Enlazar recursos (assets)

La función asset nos permite cargar también hojas de estilo css y ficheros javascript para poder usarlas en nuestras páginas.

Añadimos en nuestra layout base dos bloques, uno para css y otro para javascript

```
<html>
<head>
    {# ... #}
    {% block stylesheets %}
          <link href="{{ asset('/css/main.css') }}" type="text/css" rel="stylesheet" />
    {% endblock %}
</head>
<body>
     {# ... #}
     {% block javascripts %}
          <script src="{{ asset('/js/main.js') }}" type="text/javascript"></script>
     {% endblock %}
</body>
</html>
```

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Enlazar recursos (assets)

Ahora desde cualquier página “hija” que herede, dispondrá de los bloques definidos e incluso podría añadir nuevas páginas css o javascript:

```
{% extends '::base.html.twig' %}
{% block stylesheets %}
{{ parent() }}
<link href="{{ asset('/css/contact.css') }}" type="text/css" rel="stylesheet" />
{% endblock %}
{# ... #}
```

Como vemos, hacemos uso de la función `parent()` para no sobreescribir el bloque `stylesheets` por completo sino añadir nuestra nueva hoja css.

Plantillas

Etiquetas y “Ayudantes” (Helpers)

Variables globales

Durante cada petición, Symfony2 carga la variable global para las plantillas `app`, la cual podemos usar para acceder a algunas propiedades que pueden ser útiles:

- `app.security – El contexto de seguridad.`
- `app.user – El objeto usuario actual.`
- `app.request – El objeto request.`
- `app.session – El objeto sesión.`
- `app.environment – El entorno actual (dev, prod, etc).`
- `app.debug - True si estamos en modo debug. False en otro caso.`

Como siempre en Symfony2, podemos definir nuestras propias variables Globales:

http://symfony.com/doc/current/cookbook/templating/global_variables.html

Plantillas

Sobreescribiendo plantillas

Symfony2 nos permite sobreescribir cualquier plantilla de cualquier bundle, esto es útil por ejemplo si queremos dar una imagen corporativa a un bundle externo a nosotros.

Ejemplo:

```
public function indexAction() {
    // some logic to retrieve the blogs
    $blogs = ...;
    $this->render('AcmeBlogBundle:Blog:index.html.twig', array('blogs' => $blogs));
}
```

Para renderizar `index.html.twig`, Symfony2 busca en dos sitios en el siguiente orden:

1. **app**/Resources/AcmeBlogBundle/views/Blog/index.html.twig
2. **src**/Acme/BlogBundle/Resources/views/Blog/index.html.twig

Como vemos, copiando `index.html.twig` al directorio app podríamos dar nuestro propio estilo al BlogBundle sin modificar ningún fichero dentro del mismo.

Plantillas

Herencia de tres niveles

Es una forma de llevar a cabo la herencia entre plantillas de manera que sea lo más flexible posible.

No es obligatorio seguir estos principios…pero como siempre, es lo que recomienda Symfony2 ☺

Plantillas

Herencia de tres niveles

- Nivel 1. Plantilla base en `app/Resources/views/base.html.twig`
- Nivel 2. Una plantilla por cada "sección" de nuestro sitio web extendiendo de la plantilla base. Ejemplo, el fichero `src/Adib/TestBundle/Resources/views/layout.html.twig`

```
{% extends '::base.html.twig' %}
{% block body %}<h1>Test Application</h1>
{% block content %}
{% endblock %}
{% endblock %}
```

- Nivel 3. Una plantilla por cada página extendiendo de la plantilla de sección correspondiente. Por ejemplo, el indexAction de DefaultController `src/Adib/TestBundle/Resources/views/Default/index.html.twig`

```
{% extends 'AdibTestBundle::layout.html.twig' %}
{% block content %}
{% for task in tasks %}
    <h2>{{ task.title }}</h2>
    <p>{{ task.description}}</p>
{% endfor %}
{% endblock %}
```

Plantillas

¿Os pica el gusanillo de Twig?

Existe mucho más sobre Twig que puede consultarse:

Añadir nuestras propias extensiones:

http://twig.sensiolabs.org/doc/advanced.html#creating-an-extension

El paquete Assetic para trabajar con assets:

http://symfony.com/doc/current/cookbook/assetic/asset_management.html

Sobreescribir partes de un Bundle mediante herencia de plantillas:

http://symfony.com/doc/current/cookbook/bundles/inheritance.html

Caracteres de escape

http://symfony.com/doc/current/book/templating.html#output-escaping

Formato de los ficheros de salida

http://symfony.com/doc/current/book/templating.html#id2

Capítulo del libro referente a las plantillas:

http://symfony.com/doc/current/book/templating.html

The End

Llegó el fin…Ooohhhh!! ☹

“Un camino de mil millas comienza con un paso.”

(Benjamin Franklin)

Hasta aquí el primer paso…el resto del camino os toca recorrerlo a vosotros ☺

¡Gracias por vuestra atención!

Juan Manuel Bardallo González

Servicio de Informática y Comunicaciones. Universidad de Huelva