

Symfony y Doctrine

Doctrine es el ORM por defecto en la edición estándar de Symfony, y permite tratar con bases de datos Mysql, PostgreSQL o MicrosoftSQL

Una de las tareas más comunes y difíciles en cualquier aplicación es el tratamiento de información con una **base de datos**. La versión **Symfony Standard Edition** integra el **ORM Doctrine** por defecto, librería cuyo objetivo es proporcionar herramientas potentes para el tratamiento de la información.

Doctrine permite mapear objetos a una base de datos relacional, como **MySQL**, **PostgreSQL** o **MicrosoftSQL**, aunque también se puede hacer con **MongoDB** con la librería **Doctrine ODM** y el bundle **DoctrineMongoDBBundle**.

Vamos a ver un ejemplo de un objeto Product. Primero configuraremos la base de datos, después crearemos el objeto Product, lo incluiremos en la base de datos (**persist**) y finalmente lo extraeremos (**fetch**).

Configurar la base de datos

Primero se ha de configurar correctamente la conexión a la base de datos. Por convención esta información se configura normalmente en un archivo **app/config/parameters.yml**:

```
# app/config/parameters.yml
parameters:
    database_driver: pdo_mysql
    database_host: localhost
    database_name: tienda
    database_user: root
```

Se emplea parameters.yml para esta información porque así puedes mantener diferentes versiones del archivo para cada servidor. Puedes también guardar información de la configuración de la base de datos fuera del proyecto, como dentro de la configuración de Apache, por ejemplo ([enlace](#)).

Ahora que doctrine conoce la base de datos, puedes hacer que la cree:

```
php bin/console doctrine:database:create
```

Crear una clase Entity

Se supone que creamos una aplicación donde los productos se mostrarán. Sin pensar en Doctrine o en bases de datos, ya sabemos que necesitamos un objeto Product para representar estos productos. Creamos la clase dentro del directorio Entity en el AppBundle:

```
// src/AppBundle/Entity/Product.php
namespace AppBundle\Entity;

class Product
{
    protected $name;
    protected $price;
    protected $description;
}
```

La clase, normalmente llamada "entity", que significa una clase básica que guarda datos, es simple y ayuda a la administración de los productos en la compañía. Esta clase no se puede persistir en la base de datos todavía.

Se puede emplear un comando que te guiará paso a paso con la creación de una entidad:

```
php bin/console doctrine:generate:entity
```

Añadir información de mapeo

Doctrine permite trabajar con bases de datos de una forma más especial que simplemente pasando filas de una tabla a un array. En su lugar, Doctrine permite persistir objetos enteros en la base de datos y extraer objetos enteros. Se enlaza una **clase PHP con una tabla de base de datos**, y las **propiedades de esa clase PHP con las columnas de la tabla**:



propiedades han de enlazarse a la base de datos. Estos metadatos pueden especificarse con diferentes formatos como YAML, XML, o directamente en la clase Product con anotaciones:

```
// src/AppBundle/Entity/Product.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Class Product
 * @ORM\Entity
 * @ORM\Table(name="product")
 */
class Product
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string", length=100)
     */
    protected $name;

    /**
     * @ORM\Column(type="decimal", scale=2)
     */
    protected $price;

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

Un bundle puede aceptar **sólo un tipo de formato de información de metadatos**. No es posible mezclar metadatos provenientes de **YAML** y de **PHP**, por ejemplo.

El **nombre de la tabla es opcional**, y si se omite, se determinará automáticamente basándose en el nombre de la clase entity.

Doctrine permite elegir entre una gran variedad de campos, cada uno con sus propias opciones. Puedes ver su documentación sobre el mapeo de datos [aquí](#).

reservadas [aquí](#).

Generar getters y setters

Incluso cuando **Doctrine** sabe cómo persistir un **objeto Product en la base de datos**, la clase en sí no es realmente útil todavía. Ya que **Product** es simplemente una clase PHP normal, necesitas crear métodos **getter** y **setter** (como `getName()` o `setName()`) para acceder a las propiedades (ya que las propiedades son **protected**). Puedes hacerlo automáticamente con el siguiente comando:

```
php bin/console doctrine:generate:entities AppBundle\Entity\Product
```

Este comando asegura que se generan todos los getters y setters para la clase **Product** (si ya está generado, no lo repite).

Con el comando `doctrine:generate:entities` se puede:

- Generar getters y setters
- Gerarar clases repositorio configuradas con la anotación `@ORM\Entity`
- Generar el constructor apropiado para relaciones 1:n y n:m

El comando genera un backup de **Product** por si acaso, llamado **Product.php~**. En algunos casos este archivo puede causar un error "**Cannot redeclare class**". Puedes borrarlo o utilizar la opción **--no-backup** para prevenir la generación de estos **archivos backup**.

También puedes generar todas las entidades de vez de un bundle o incluso de un namespace:

```
// Genera todas las entidades en AppBundle
php bin/console doctrine:generate:entities AppBundle
// Genera todas las entidades de los bundles en el namespace Acme
php bin/console doctrine:generate:entities Acme
```

Crear las tablas de la base de datos

Ahora tienes una clase usable **Product** con información de mapeo de forma que **Doctrine** sabe como persistir en la tabla correspondiente. De momento no tenemos la tabla **product** en la base de datos. **Doctrine** puede crear automáticamente las tablas para cualquier entidad conocida en tu aplicación:

```
php bin/console doctrine:schema:update --force
```

SQL necesarias para actualizar la base de datos y formarla adecuadamente. Si añades una nueva propiedad en `Product` y ejecutas de nuevo este comando, generará la sentencia `ALTER TABLE` necesaria para añadir la nueva columna en la tabla existente `product`.

Una forma incluso mejor para beneficiarse de esta funcionalidad es a través de **migraciones**, que permiten generar estas sentencias SQL y guardarlas en clases **migration** que pueden ejecutarse sistemáticamente en tu servidor de producción para migrar tu esquema de base de datos de forma segura.

Persistir objetos en la base de datos

Ahora que ya hemos enlazado la entidad **Product** con la tabla correspondiente `product`, podemos persistir datos en la base de datos. Desde un controlador es bastante sencillo. Vamos a añadir un método para crear un producto en `ProductController`:

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

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use AppBundle\Entity\Product;
use Symfony\Component\HttpFoundation\Response;

class ProductController extends Controller
{
    /**
     * @Route("/create", name="crearProducto")
     */
    public function createAction()
    {
        $product = new Product();
        $product->setName('Algo');
        $product->setPrice('20');
        $product->setDescription('Holalala');

        $em = $this->getDoctrine()->getManager();

        $em->persistent($product);
        $em->flush();

        return new Response('Se ha creado el producto con id: '.$product->getId());
    }
}
```

Este ejemplo utiliza el método `getDoctrine()` que hereda de la **clase base Controller**. Este método es un shortcut para obtener el **service doctrine**. Puedes

El método `persist()` le dice a **Doctrine** que administre el objeto `$product`. Esto todavía no ejecuta ninguna sentencia en la base de datos.

Cuando se llama al método `flush()`, Doctrine mira entre los objetos que administra si hay alguno que también requiera ser persistido. En este ejemplo, el objeto `$product` no se ha persistido todavía, por lo que el entity manager ejecuta una sentencia `INSERT` y se crea una fila en la tabla `product`.

Ya que Doctrine está pendiente de todas las entidades administradas, cuando llamas al método `flush()` ejecuta las sentencias en el orden correcto. Utiliza **sentencias preparadas** cacheadas para mejorar el rendimiento. Por ejemplo, si persistes un total de 100 objetos `Product` y llamas a `flush()`, Doctrine ejecutará 100 sentencias `INSERT` en una sola sentencia preparada.

Cuando se crean o actualizan objetos, el workflow es siempre el mismo. Si a Doctrine se le dice que cree un nuevo registro que ya existe, sabe actualizarlo, mediante una sentencia `UPDATE` en lugar de `INSERT`.

Doctrine proporciona una librería que permite cargar datos de testing programáticamente en el proyecto, con el **DoctrineFixturesBundle**.

Extraer objetos de la base de datos (`fetch`)

Extraer un objeto de la base de datos es muy sencillo. En el mismo controller `ProductController` creamos el método `showAction()`:

```
/**
 * @Route("/product/{id}", name="showproduct")
 */
public function showAction($id)
{
    $product = $this->getDoctrine()->getRepository('AppBundle:Product')->find($id);

    if(!$product) {
        throw $this->createNotFoundException(
            'No se ha encontrado el producto con id: '.$id
        );
    }

    return $this->render('product.html.twig', array(
        'product' => $product
    ));
}
```

Cuando solicitas un tipo particular de objeto, siempre empleas lo que es conocido como repositorio. Un repositorio es una clase PHP cuyo único trabajo es ayudar a

```
$repository = $this->getDoctrine()->getRepository('AppBundle:Product');
```

El string AppBundle:Product es un shortcut que se puede emplear en cualquier parte en Doctrine en lugar de usar la clase entera de la entidad (AppBundle\Entity\Product). Siempre que tu entidad esté en el namespace Entity de tu bundle.

Una vez que tienes el repositorio, tienes acceso a varios métodos:

```
// Consulta por key primaria (normalmente "id")
$product = $repository->find($id);

// Nombres de métodos para encontrar en función del valor de una columna
$product = $repository->findOneById($id);
$product = $repository->findOneByName('foo');

// Encontrar todos los productos
$products = $repository->findAll();

// Encontrar un grupo de productos basándose en un valor de columna
$products = $repository->findByPrice(19.99);
```

Con los métodos `findBy()` y `findOneBy()` puedes extraer objetos bajo múltiples condiciones:

```
// Consulta un producto que coincida con nombre y precio
$product = $repository->findOneBy(
    array('name' => 'foo', 'price' => 19.99)
);

// Consulta todos los productos que coinciden con el nombre, ordenados por precio
$products = $repository->findBy(
    array('name' => 'foo'),
    array('price' => 'ASC')
);
```

Cuando renderizas cualquier página, puedes ver cuantas queries se han realizado en la barra debug. Si haces click en el ícono que muestra las queries, se abrirá el profiler, mostrando las consultas exactas que se hayan hecho.

El ícono se mostrará amarillo si hubo más de 50 consultas en la página. Esto podría indicar que algo no va bien.

Actualizar un objeto

```

/**
 * @Route("/update/{id}", name="updateproduct")
 */
public function updateAction($id)
{
    $em = $this->getDoctrine()->getManager();
    $product = $em->getRepository('AppBundle:Product')->find($id);

    if(!$product) {
        throw $this->createNotFoundException(
            'No product found for id' . $id
        );
    }

    $product->setName('Alcachofas');
    $em->flush();

    return $this->redirectToRoute('homepage');
}

```

Actualizar un objeto se hace en tres pasos:

1. Extraer el objeto de **Doctrine**
2. Modificar el objeto
3. Llamar a **flush()** con el entity manager

Llamar a **\$em->persist(\$product)** no es necesario. Este método simplemente le dice a Doctrine que administre o que vigile el objeto **\$product**. En este caso, como ya trajimos el objeto **\$product** de Doctrine, ya está administrado.

Borrar un Objeto

Borrar un objeto es muy similar, pero requiere una llamada al método **remove()** del entity manager.

```

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

```

El método **remove()** notifica a Doctrine que quieres remover el objeto de la base de datos pero la consulta **DELETE** no es ejecutada hasta que se llama al método **flush()**.

Consultar objetos

Ya has visto como el objeto repositorio permite has consultas básicas sin hacer nada.

Pero Doctrine también permite realizar consultas más completas utilizando el Doctrine Query Language (DQL). DQL es similar a SQL pero tienes que imaginar que estás haciendo una consulta para uno o más objetos de una clase entity (como Product) en lugar de consultar filas de una tabla.

Cuando se hacen consultas se tienen dos opciones: [Doctrine queries](#) o [Doctrine Query Builder](#).

Consultar objetos con DQL

Queremos consultar productos, pero mostrar sólo aquellos que cuestan más de 19.99 , ordenados del más barato al más caro. Podemos utilizar DQL:

```
$em = $this->getDoctrine()->getManager();
$query = $em->createQuery(
    'SELECT p
     FROM AppBundle:Product p
     WHERE p.price > :price
     ORDER BY p.price ASC'
)->setParameter('price', '19.99');

$products = $query->getResult();
// para obtener sólo un resultado:
// $product = $query->setMaxResults(1)->getOneOrNullResult();
```

Si tienes costumbre de usar SQL, [DQL](#) resultará muy sencillo. La diferencia más grande es que tendrás que pensar en términos de [objetos](#) en lugar de [filas](#) en una base de datos. Seleccionamos desde el objeto [AppBundle:Product](#) y le pones un alias [p](#).

Observa el método [setParameter\(\)](#). Cuando se trabaja con Doctrine siempre es una buena idea establecer valores externos como [placeholders](#) ([:price](#) en el ejemplo anterior) ya que previene ataques [SQL injection](#).

El método [getResult\(\)](#) devuelve un array de resultados. Para obtener sólo uno se puede usar [getOneOrNullResult\(\)](#):

```
$product = $query->setMaxResults(1)->getOneOrNullResult();
```

DQL es muy potente, permitiéndote también unir entidades, agrupar, etc. Puedes ver más en la [documentación oficial de DQL](#).

Consultar objetos con Doctrine Query Builder

cuando tu código empieza a complicarse con DQL y empiezas a concatenar strings:

```
$repository = $this->getDoctrine()
    ->getRepository('AppBundle:Product');

// createQueryBuilder automáticamente selecciona FROM AppBundle:Product
// y le pone un alias "p"
$query = $repository->createQueryBuilder('p')
    ->where('p.price > :price')
    ->setParameter('price', '19.99')
    ->orderBy('p.price', 'ASC')
    ->getQuery();

$products = $query->getResult();
// para obtener sólo un resultado:
// $product = $query->setMaxResults(1)->getOneOrNullResult();
```

El objeto **QueryBuilder** contiene cada método necesario para construir la consulta. Llamando al método **getQuery()** el query builder devuelve un objeto normal Query, que puede usarse para obtener el resultado del query. Puedes leer la [documentación del QueryBuilder](#).

Clases repositorio personalizadas

En las secciones anteriores hemos contruído y utilizado consultas más complicadas desde un controller. Para aislarlas, testear y reusar estas consultas, es una buena práctica crear una clase repositorio para la entidad y añadir métodos con la lógica de las consultas ahí.

Para hacerlo, añade el nombre de la clase repository a la definición del mapeo:

```
// src/AppBundle/Entity/Product.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="ProductRepository")
 */
class Product
{
    //...
}
```

Doctrine puede generar la clase repository ejecutando el comando de antes:

```
php bin/console doctrine:generate:entities AppBundle
```

```
// src/AppBundle/Entity/ProductRepository.php
namespace AppBundle\Entity;

use Doctrine\ORM\EntityRepository;

class ProductRepository extends EntityRepository
{
    public function findAllOrderedByName()
    {
        return $this->getEntityManager()
            ->createQuery(
                'SELECT p FROM AppBundle:Product p ORDER BY p.name ASC'
            )
            ->getResult();
    }
}
```

Puede accederse al entity manager desde el repositorio con `$this->getEntityManager()`.

Ahora podemos emplear ese método como los métodos por defecto del repositorio:

```
$em = $this->getDoctrine()->getManager();
$products = $em->getRepository('AppBundle:Product')
    ->findAllOrderedByName();
```

Relaciones/Asociaciones entre entidades

Suponemos que los productos de la aplicación pertenecen todos a una categoría. En este caso necesitaremos un objeto Category y una forma de relacionar un objeto Product con un objeto Category. Primero creamos la entidad Category. Podemos hacerlo con el siguiente comando:

```
php bin/console doctrine:generate:entity --no-interaction --entity="AppBundle:Category"
```

Ahora ya tenemos una entidad **Category**, con un campo **id**, un campo **name** y los métodos **getter** y **setter** asociados.

Para relacionar las entidades Category y Product, creamos una propiedad **products** en la clase Category:

```
// src/AppBundle/Entity/Category.php
// ...
use Doctrine\Common\Collections\ArrayCollection;
```

```
// ...

/**
 * @ORM\OneToMany(targetEntity="Product", mappedBy="category")
 */
protected $products;

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

Ya que el objeto Category va a referenciar a varios objetos Product, una propiedad array de products se añade para almacenar esos objetos Product.

El código en el método __construct es importante porque Doctrine requiere que la propiedad \$products sea un objeto ArrayCollection. Esto objeto parece y actúa casi igual que un array, pero añade alguna flexibilidad.

El valor **targetEntity** puede referenciar a una entidad con un namespace válido, no sólo entidades definidas en el mismo namespace.

Ya que cada objeto **Product** puede hacer referencia a sólo una categoría, añadimos la propiedad **\$category** en la clase Product:

```
/**
 * @ORM\ManyToOne(targetEntity="Category", inversedBy="products")
 * @ORM\JoinColumn(name="category_id", referencedColumnName="id")
 */
protected $category;
```

Ahora que ya hemos añadido una nueva propiedad tanto en la clase **Category** como en la clase **Product**, le decimos a Doctrine que genere los métodos gettes y setter:

```
php bin/console doctrine:generate:entities AppBundle
```

Ahora tenemos dos clases, Category y Product con una relación natural one-to-many. La clase Category almacena un array de objetos Product y el objeto Product puede almacenar un objeto Category. En otras palabras, has construído las clases de forma que tiene sentido para tus requisitos. El hecho de que los datos han de ser persistidos en la base de datos es siempre secundario.

Si miramos a los metadatos sobre la propiedad **\$category** en la clase **Product**, esa información le dice a **Doctrine** que la clase relacionada es **Category** y que debería almacenar el id del registro de la categoría en un campo **_categoryid** en la tabla

el valor id de la categoría en una columna `_categoryid` de la tabla product.

Relación entre tablas Doctrine

Los metadatos sobre la propiedad `$products` del objeto `Category` es menos importante, y simplemente le dice a Doctrine que mire en la propiedad `Product.category` para averiguar cómo está mapeada la relación.

Antes de continuar hay que asegurarse de que doctrine añade la nueva tabla `category`, y la columna `product.category_id` y una nueva foreign key:

```
php bin/console doctrine:schema:update --force
```

Este comando sólo debería usarse durante el desarrollo. Para un método más robusto de actualización sistemática en una base de datos en producción, es mejor emplear [migraciones](#).

Guardar entidades relacionadas

Ahora podemos crear una categoría, un producto, y asignar el producto a una categoría:

```
use AppBundle\Entity\Product;
use AppBundle\Entity\Category;
use Symfony\Component\HttpFoundation\Response;

class ProductController extends Controller
{
    /**
     * @Route("/create", name="crearProducto")
     */
    public function createAction()
    {
        $category = new Category();
        $category->setName('Comida');

        $product = new Product();
        $product->setName('Manzana');
        $product->setPrice('1.20');
        $product->setDescription('Manzana roja');

        $product->setCategory($category);

        $em = $this->getDoctrine()->getManager();

        $em->persist($category);
        $em->persist($product);
        $em->flush();
```

```
y tu de categoria. $category->getCategory() ,  
}
```

Extraer objetos relacionados

Cuando necesitamos extraer objetos asociados, el workflow es igual que antes. Primero extraemos un objeto `$product` y después accedemos a su categoría relacionada:

```
public function showAction($id)  
{  
    $product = $this->getDoctrine()  
        ->getRepository('AppBundle:Product')  
        ->find($id);  
  
    $categoryName = $product->getCategory()->getName();  
  
    // ...  
}
```

En este ejemplo primero consultamos un objeto `Product` basándonos en el `id` del producto. Esto emite una consulta sólo para los datos del producto y rellena el objeto `$product` con esos datos. Después, cuando llamamos a `$product->getCategory()->getName()`, `Doctrine` hace una segunda consulta para encontrar la categoría que está relacionada con este producto. Prepara al objeto `$category` y lo devueleve.

Relaciones entre tablas Doctrine

Lo que es importante es el hecho de que tienes fácil acceso a la categoría relacionada con el producto, pero los datos de la categoría no son extraídos hasta que no pides la categoría (esto se llama **lazy loading**).

También podemos consultar al revés:

```
public function showProductsAction($id)  
{  
    $category = $this->getDoctrine()  
        ->getRepository('AppBundle:Category')  
        ->find($id);  
  
    $products = $category->getProducts();  
  
    // ...  
}
```

relacionados, pero sólo una vez. La variable `$products` es un array de todos los objetos `Product` que están enlazados al objeto `Category` dado a través del valor `_categoryid`.

Relaciones y clases Proxy

El **lazy loading** es posible porque, cuando es necesario, Doctrine devuelve un objeto proxy en lugar del objeto real. En el ejemplo anterior, si hacemos `dump` para obtener la clase de `$category`:

```
dump(get_class($category));
// Obtenemos: "Proxies\AppBundle\Entity\CategoryProxy"
```

Este objeto proxy extiende el objeto `Category`, y parece y actúa como él. La diferencia es que usando un objeto proxy, Doctrine puede atrasar la consulta de los datos `Category` reales hasta que realmente necesites esos datos (por ejemplo hasta que llames a `$category->getName()`).

Las clases proxy son generadas por Doctrine y se guardan en el directorio `cache`. Y aunque no te des cuenta de que un objeto `$category` es realmente un objeto proxy, es importante tenerlo en cuenta.

En la siguiente sección, cuando extraigamos los datos de producto y categoría de vez con un `join`, Doctrine devolverá el objeto `Category` de verdad, ya que nada ha de ser cargado de forma `lazy`.

Unir registros relacionados

En los ejemplos anteriores se han hecho dos consultas, una para el objeto original (por ejemplo, `Category`) y otra para el objeto relacionado (los objetos `Product`).

Si de primeras ya sabes que necesitarás acceder a ambos objetos, puedes evitar la segunda consulta con un `join` en la consulta original mediante un método:

```
// src/AppBundle/Entity/ProductRepository.php
public function findOneByIdJoinedToCategory($id)
{
    $query = $this->getEntityManager()
        ->createQuery(
            'SELECT p, c FROM AppBundle:Product p
             JOIN p.category c
             WHERE p.id = :id'
        )->setParameter('id', $id);

    try {
        return $query->getSingleResult();
    }
}
```

{}

Ahora podemos usar este método en el controller para consultar para un objeto **Product** y su categoría relacionada con una sola consulta:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AppBundle:Product')
        ->findOneByIdJoinedToCategory($id);

    $category = $product->getCategory();

    // ...
}
```

Para más información sobre asociaciones puedes leer la [Association Mapping Documentation](#).

Lifecycle Callbacks

A veces necesitamos hacer una acción justo antes o después de que una entidad es insertada, actualizada o eliminada. Estos tipos de acciones son conocidos como lifecycle callbacks, ya que son métodos callback que necesitas ejecutar durante los diferentes estados del ciclo de vida de una entidad.

Si usamos anotaciones para los metadatos, debemos activar los lifecycle callbacks así:

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

Ahora podemos decirle a Doctrine que ejecute un método en cualquiera de los eventos lifecycle disponibles. Por ejemplo, queremos crear una columna `createdAt` con la fecha actual, sólo cuando la entidad es persistida (insertada) por primera vez (se supone que ya hemos creado y mapeado la propiedad `createdAt`):

```
// src/AppBundle/Entity/Product.php

/**
 * @ORM\PrePersist
```

```
$this->createdAt = new \DateTime();  
}
```

Ahora, justo antes de que la entidad sea persistida por primera vez, Doctrine llamará automáticamente a este método y el campo `createdAt` se establecerá en la fecha actual.

Puedes ver más eventos lifecycle en la [Lifecycle Events documentation](#).

Se puede ver que el método `setCreatedAtValue()` no recibe ningún argumento. Este es siempre el caso para los lifecycle callbacks y es intencional: los lifecycle callbacks deberían ser simples métodos que tienen en cuenta la transformación interna de datos en la entidad (por ejemplo establecer un campo `created/updated`, generar un valor `slug`, etc).

Si lo que se quiere es hacer algo más pesado, como [logging](#) o [enviar un email](#), deberías registrar una clase externa como event listener o subscriber y darle acceso a los recursos que sean necesarios.

Fuentes: symfony.com



Copyright © Diego Lázaro 2018

Sitio construido con [Symfony & Semantic–UI](#)