Inyección de dependencias

La inyección de dependencias es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree el objeto. También se conoce como IoC (Inversion of Control).

Veámoslo en un ejemplo. Pongamos que queremos utilizar la clase Symfony\Component\Ldap\Ldap en nuestro controlador para realizar una conexión manualmente a LDAP y realizar las operaciones que nos apetezca.

Método tradicional

El método tradicional es obtener el objeto Ldap a través del constructor de la clase con el operador new:

```
use Symfony\Component\Ldap\Ldap;

public function indexAction()
{
    $ldap = new Ldap();


    // ...
}
```

Pero el constructor de la clase Ldap, necesita un parámetro de entrada:

```
final class Ldap implements LdapInterface
{
    private $adapter;

    public function __construct(AdapterInterface $adapter)
    {
        $this->adapter = $adapter;
    }
```

Por lo que para hacer new Ldap, necesitamos primero un objeto Adapter:

```
use Symfony\Component\Ldap\Ldap;
use Symfony\Component\Ldap\Adapter\ExtLdap\Adapter;

public function indexAction()
{
    $ldapAdapter = new Adapter();
    $ldap = new Ldap($ldapAdapter);

    // ...
}
```

Pero si ahora nos fijamos en el constructor de Adapter, necesitamos también un parámetro de entrada que es un array.

```
class Adapter implements AdapterInterface
{
    private $config;
    private $connection;
    private $entryManager;

    public function __construct(array $config = array())
    {
        if (!extension_loaded('ldap')) {
            throw new LdapException('The LDAP PHP extensio
n is not enabled.');
        }

        $this->config = $config;
    }
```

Y lo peor es que simplemente mirando el constructor, no sabemos qué contenido debe de tener el array $config.

Después de mirar la documentación de la clase Adapter, averiguamos la información que nos hace falta y ya podemos instanciar la clase

```
use Symfony\Component\Ldap\Ldap;
use Symfony\Component\Ldap\Adapter\ExtLdap\Adapter;

public function indexAction()
{
```

```
    $config = array('host' => '138.100.191.229',
                    'port' => 636,
                    'encryption' => 'ssl',
                    'options' => array(
                        'protocol_version' => 3,
                        'referrals' => false
                    ));

    $ldapAdapter = new Adapter($config);

    $ldap = new Ldap($ldapAdapter);

    // ...
}
```

Y ya por fin, obtenemos nuestro servicio ldap listo para ser utilizado.

Problemas de esto:

- Hacen falta demasiadas líneas de programación simplemente para instanciar un único objeto.
- Hace falta conocimiento de las clases y de los constructores para poder instanciarlas todas.
- Si modifico el constructor de un servicio (para añadir otro parámetro, o para cambiarlo por otro...) tengo que revisar TODA la aplicación para cambiar todos los new que haya de dicho servicio.

Con inyección de dependencias

Si se utiliza el patrón de inyección de dependencias, es el sistema el que se encarga de suministrar los objetos correspondientes a cada clase.

Simplemente tipando en la acción de un controlador el tipo de cada servicio, Symfony se encargará de instanciar todas clases que sean necesarias:

```
public function indexAction(Psr\Log\LoggerInterface $logge
r, Doctrine\ORM\EntityManagerInterface $em, Symfony\Compon
ent\Ldap\Ldap $ldap)
{


}
```

Con este patrón, desparecen todos los problemas mencionados anteriormente.

También funciona la inyección de dependencias en los constructores de otros servicios. Es decir, un servicio puede pedir al contenedor de servicios que le inyecte otros servicios simplemente tipándolos en los argumentos del constructor.

El contenedor de Servicios

Una aplicación está llena de objetos útiles: Un objeto “Mailer” es útil para enviar correos, el “EntityManager” para hacer operaciones con las entidades de Doctrine…

En Symfony, estos “objetos útiles” se llaman servicios, y viven dentro de un objeto especial llamado contenedor de servicios. El contenedor nos permite centralizar el modo en el que los objetos son construidos. Simplifica el desarrollo, ayuda a construir una arquitectura robusta y es muy rápido.

El contenedor de servicios actúa mediante el patrón de inyección de dependencias cuando tipamos la clase en un parámetro de entrada de un controlador o de un constructor.

```
public function index(Doctrine\ORM\EntityManagerInterface
$em)
{

}
```

El siguiente comando nos da una lista de los servicios que tenemos disponibles:

> *bin/console debug:autowiring*

Se puede ejecutar el comando para buscar algo específico:

> *bin/console debug:autowiring cache*

Para obtener la lista completa con más detalles, tenemos otro comando:

> *bin/console debug:container*

NOTA: El contenedor de dependencias utiliza la técnica de lazy-loading: no instancia un servicio hasta que se pide dicho servicio. Si no se pide, no se instancia.

NOTA: Un servicio se crea una única vez. Si en varias partes de la aplicación se le pide a Symfony un mismo servicio, Symfony devolverá siempre la misma instancia del servicio.

Creación y configuración de servicios

El fichero services.yml

```
# app/config/services.yml
services:
    # default configuration for services in *this* file
    _defaults:
        # Habilita el tipado de argumentos en los métodos constructores de los servicios
        autowire: true
        # Con autoconfigure true no es necesario poner tags a los servicos. Symfony las averigua por las interfaces que implementan.
        autoconfigure: true
        # Solamente se pueden obtener servicios con $container->get() si son públicos
        public: false

    # makes classes in src/AppBundle available to be used as services
    AppBundle\:
        resource: '../../src/*'
        # you can exclude directories or files
        # but if a service is unused, it's removed anyway
```

```
        exclude: '../../src/{Entity{Entity,Migrations,Test
s,Kernel.php}'
```

autowire

Habilita el tipado de argumentos en los métodos constructores de los servicios

arguments

Cuando un servicio necesita argumentos que no son instancias de clases sino que son valores (como un host, un username un password, etc) no queda más remedio que declarar el servicio y establecer los valores de los argumentos

```
    Symfony\Component\Ldap\Ldap:
        arguments: ['@Symfony\Component\Ldap\Adapter\ExtLd
ap\Adapter']

    Symfony\Component\Ldap\Adapter\ExtLdap\Adapter:
        arguments:
            -   host: 138.100.191.229
                port: 636
                encryption: ssl
                options:
                    protocol_version: 3
                    referrals: false
```

public

Solamente se pueden obtener servicios con $container->get() si dichos servicios son públicos.

tags

A algunos servicios hay que etiquetarlos para que symfony sepa donde van a ser utilizados dentro del framework.

Por ejemplo: para crear una extensión de Twig, necesitamos crear una clase, registrarla como servicio y etiquetarla con *twig.extension*.

Otro ejemplo: para crear un voter, hay que crear una clase, registrarla como servicio y etiquetarla con security.voter.

```
    App\Twig\MyTwigExtension:
        tags: [twig.extension]
    app.post_voter:
        class: App\Security\EditarEventoVoter
        tags:
            - { name: security.voter }
        public: false
```

autoconfigure

Con autoconfigure true no es necesario poner tags a los servic ios.

Symfony las averigua por las interfaces que implementan.

En los ejemplos anteriores, Symfony sabra que el servicio MyTwigExtension es una extensión de Twig porque la clase implementa Twig_ExtensionInterface y que el servicio app.post_voter es un voter porque la clase implementa VoterInterface.

resource y exclude

La clave resource se utiliza para registrar de forma rápida como servicios todas las clases dentro de un directorio. El id de cada servicio es su fully-qualified class name.

La clave exclude se utiliza para excluir directorios.

```
    App\:
        resource: '../src/*'
        exclude: '../src/{Entity,Migrations,Tests,Kernel.p
hp}'
```

Registrar varios servicios con la misma clase

Es posible registrar varios servicios distintos que utilicen la misma clase. Basta con ponerles identificadores distintos.

```
services:
```

```
    site_update_manager.superadmin:
        class: AppBundle\Updates\SiteUpdateManager
        # you CAN still use autowiring: we just want to sh
ow what it looks like without
        autowire: false
        # manually wire all arguments
        arguments:
            - '@AppBundle\Service\MessageGenerator'
            - '@mailer'
            - 'superadmin@example.com'

    site_update_manager.normal_users:
        class: AppBundle\Updates\SiteUpdateManager
        autowire: false
        arguments:
            - '@AppBundle\Service\MessageGenerator'
            - '@mailer'
            - 'contact@example.com'

    # Create an alias, so that - by default - if you type-
hint SiteUpdateManager,
    # the site_update_manager.superadmin will be used
    AppBundle\Updates\SiteUpdateManager: '@site_update_man
ager.superadmin'
```