Dependency Injection

SensioLabs

Life without Dependency Injection

A common logger class

```
class Logger
{
    public function log($message, $level = 'INFO')
    {
        $formatter = new XmlFormatter();
        $log = $formatter->format($message, $level);
        $this->writeLog($log);
    }
}
```

This code is technically correct, but it introduces a lot of problems not acceptable for professional applications.

The problems of the common logger class

```
class Logger
{
    public function log($message, $level = 'INFO')
    {
        $formatter = new XmlFormatter();
        $log = $formatter->format($message, $level);
        $this->writeLog($log);
    }
}
```

The logger is **very limited**, because it only supports one specific message formatter (XML in this example).

The problems of the common logger class

```
class Logger
{
    public function log($message, $level = 'INFO')
    {
        $formatter = new XmlFormatter();
        $log = $formatter->format($message, $level);
        $this->writeLog($log);
    }
}
```

The logger is **not flexible**, because it only supports this specific XML formatter (other XML formatters won't work).

The problems of the common logger class

```
class Logger
{
    public function log($message, $level = 'INFO')
    {
        $formatter = new XmlFormatter();
        $log = $formatter->format($message, $level);
        $this->writeLog($log);
    }
}
```

The logger is **hard to test**, because it creates its own dependencies. **Rule of thumb**: avoid "**new**" keyword in your code.

First steps towards Dependency Injection

Remove the XMLFormatter dependency (1/2)

```
class Logger
{
  private $formatter;

  function __construct(XmlFormatter $formatter)
  {
    $this->formatter = $formatter;
  }
}
```

The logger no longer creates its **dependencies**, but expects them to be "**injected**" via the **constructor**.

Remove the XMLFormatter dependency (2/2)

```
class Logger
{
  private $formatter;

  public function log($message, $level = 'INFO')
  {
     $log = $this->formatter->format($message, $level);
     $this->writeLog($log);
  }
}
```

The **log()** method now relies on whichever **XMLFormatter** was passed when creating the Logger class.

Using the new logger

```php
$logger = new Logger(new XmlFormatter());
$logger->log('An error to log');
```

OUTPUT

```
<log>
    <message>An error to log</message>
</log>
```

Before creating the logger object, the code must create its **dependencies** and **inject** them into the **constructor**.

Allow to use any formatter in the logger

```
class Logger
{
    private $formatter;

    function __construct(Formatter $formatter)
    {
        $this->formatter = $formatter;
    }
}
```

The constructor argument is now type-hinted with an abstract class or interface. This allows to use any compatible formatter.

Using the new logger

```
$logger = new Logger(new JsonFormatter());
$logger->log('An error to log');
```

```
{ message: "An error to log" }
```

OUTPUT

Logger now accepts any formatter which complies with the **Formatter** interface. This provides flexibility and simplifies testing.

Dependency Injection

What is Dependency Injection?

« Dependency Injection is where components are given their dependencies through their constructors, methods, or directly into fields. »

Source: picocontainer.org

What is Dependency Injection?

«Dependency injection allows a program to follow the **dependency inversion principle**. The client code delegates to external code (the injector) the responsibility of constructing the services and calling the client to inject them.»

Source: wikipedia.org

Dependency Injection types

- **Constructor** injection
  recommended for most Symfony applications
- **Setter** injection
  useful when using optional dependencies
- **Property** injection
  dangerous to use and not really needed

Constructor injection example

```
class Logger
{
    private $formatter;

    function __construct(Formatter $formatter)
    {
        $this->formatter = $formatter;
    }
}
```

Recommended for Symfony applications.

Constructor injection pros and cons

- **Pro:** defines mandatory dependencies.
- **Pro:** dependencies are guaranteed to not change during execution.
- **Con:** optional dependencies are harder to define.
- **Con:** working with class hierarchies is more difficult.

Setter injection example

```
class Logger
{
    private $formatter;

    function setFormatter(Formatter $formatter)
    {
        $this->formatter = $formatter;
    }
}
```

Use it occasionally for setting optional dependencies.

Setter injection pros and cons

- **Pro:** works well with optional dependencies.
- **Pro:** the setter can be called repeatedly to set a collection of dependencies.
- **Con:** dependencies can vary during runtime (and produce hard to debug side-effects).
- **Con:** there is no guarantee that the setter is called (dependency may not exist).

Property injection example

```
class Logger
{
    public $formatter;
}
```

Making the property **public** allows any other part of the application to set its value (i.e. to **inject** the **dependency**).

Don't use this type of injection. It introduces a lot of drawbacks.

Property injection pros and cons

- **Pro:** works well with optional dependencies.
- **Con:** same as the setter injection.
- **Con:** PHP properties cannot define types, so dependencies aren't type-hinted.

The Service Container

Dependency Injection creates new problems

```
$logger = new Logger(new XmlFormatter());
$logger->log('...');
```

Creating objects becomes more **complex** because you need to know the objects and properties on which each project depends.

You need to store and maintain the "**instructions**" to build each object. And you must solve issues such as circular dependencies.

What is a Service Container?

«The **Service Container** is simply a PHP object that manages the instantiation of services. »

« A **Service** is any PHP object that performs a "global" task (e.g. logging messages or delivering emails) »

The Service Container in Symfony

- All Symfony **core classes** use the service container.
- The service container emphasizes an architecture that promotes **reusable** and **decoupled** code.
- The service container is the biggest contributor to the **speed** and **extensibility** of Symfony.
- You can't **master Symfony** without mastering first the service container.

Full details: symfony.com/doc/current/service_container.html

Using the Service Container in Symfony

```
// getting a service instance
$logger = $container->get('logger');

// getting a configuration parameter
$host = $container->getParameter(
  'database_host'
);
```

Symfony builds the Service Container automatically and provides lots of ready-to-use services and configuration parameters.

An example of the generated services

```php
// var/cache/dev/appDevDebugProjectContainer.php
class appDevDebugProjectContainer extends Container
{
    $this->methodMap = array(
        // ...
        'logger' => 'getLoggerService',
    );

    protected function getLoggerService()
    {
        $instance = new \Symfony\Bridge\Monolog\Logger('app');
        $this->services['logger'] = $instance;

        $instance->pushHandler($this->get('monolog.handler.console'));
        $instance->pushHandler($this->get('monolog.handler.main'));
        $instance->pushHandler($this->get('monolog.handler.debug'));

        return $instance;
    }
}
```

Instantiation

Initialization

Built-in services

Services provided by Symfony

- Symfony already provides services for lots of common web tasks (sending emails, encoding passwords, generating URLs, etc.)
- You just need to define your own services (which can use the built-in services if needed)

Display the list of services available in your app

```
$ php bin/console debug:container

Service ID                  Class name
annotation_reader           Doctrine\Common\Annotations\FileCacheReader
assetic.asset_manager       Assetic\Factory\LazyAssetManager
assetic.controller          Symfony\Bundle\AsseticBundle\Controller\AsseticController
assetic.filter.cssrewrite   Assetic\Filter\CssRewriteFilter
assetic.filter.jsqueeze     Assetic\Filter\JSqueezeFilter
assetic.filter.scssphp      Assetic\Filter\ScssphpFilter
assetic.filter_manager      Symfony\Bundle\AsseticBundle\FilterManager
assetic.request_listener    Symfony\Bundle\AsseticBundle\EventListener\RequestListener
cache_clearer               Symfony\Component\HttpKernel\CacheClearer\ChainCacheClearer
cache_warmer                Symfony\Component\HttpKernel\CacheWarmer\CacheWarmerAggregate

...

validator                   Symfony\Component\Validator\ValidatorInterface
var_dumper.cloner           Symfony\Component\VarDumper\Cloner\VarCloner
```

Most commonly used built-in services

- doctrine
- filesystem
- form.factory
- form.registry
- logger
- mailer
- profiler
- property_accessor
- request_stack
- router
- security.authentication_utils
- security.authorization_checker
- security.csrf_provider
- security.password_encoder
- security.token_storage
- service_container
- templating
- translator
- twig
- validator

Built-in service names are concise and self-explanatory.

Service configuration

Where to define custom services

`app/config/services.[yml|xml|php]`

Recommended by the Symfony Best Practices. Mostly used for small to medium applications that follow the practice of defining just one bundle called **AppBundle**.

`<your-bundle>/Resources/config/services.[yml|xml|php]`

Recommended for applications that split their code into several bundles and also for reusable third-party bundles.

Defining services in the application config

```yaml
# app/config/services.yml
services:
    service_1:
        # ...
    services_2:
        # ...
    # ...
```

Services defined in the global **services.yml** configuration file are automatically loaded by Symfony.

Defining services in the bundle config (1/2)

```yaml
# src/AcmeBlogBundle/Resources/config/services.yml
services:
    service_1:
        # ...
    services_2:
        # ...
    # ...
```

Services defined in the **bundles'** configuration files are **not** automatically loaded by Symfony.

Defining services in the bundle config (2/2)

```php
namespace Acme\BlogBundle\DependencyInjection;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\Config\FileLocator;

class AcmeBlogExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $loader = new YamlFileLoader(
            $container, new FileLocator(__DIR__.'/../Resources/config')
        );

        $loader->load('services.yml');
    }

    // ...
}
```

This "**Dependency Injection extension**" looks for and loads the configuration files defined in **Resources/config/**

Basic service definition

A simple class with no arguments (XML)

```
<!-- app/config/services.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services
        http://symfony.com/schema/dic/services/services-1.0.xsd">
  <services>
    <service id="app.markdown"
             class="AppBundle\Service\MarkdownParser" />
  </services>
</container>
```

```
$parser = new \AppBundle\Service\MarkdownParser();
```

A simple class with no arguments (YML)

```yaml
# app/config/services.yml
services:
    app.markdown:
        class: AppBundle\Service\MarkdownParser
```

```php
$parser = new \AppBundle\Service\MarkdownParser();
```

Service definition (id, class)

```yaml
app.markdown:
    class: AppBundle\Service\MarkdownParser
```

```xml
<service id="app.markdown"
         class="AppBundle\Service\MarkdownParser" />
```

The **id** parameter is the service name. **Best Practice:** use one or two word names preffixed with **app.**

The **class** parameter is the fully qualified class name to instantiate.

A simple class with arguments (XML)

```
<!-- app/config/services.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services
        http://symfony.com/schema/dic/services/services-1.0.xsd">
  <services>
    <service id="app.slugger"
             class="AppBundle\Service\Slugger">
        <argument>_</argument>
    </service>
  </services>
</container>
```

The argument is just an **underscore** because this argument refers to the character used to separate words in the slug.

```
$slugger = new \AppBundle\Service\Slugger('_');
```

A simple class with arguments (YML)

```yaml
# app/config/services.yml
services:
    app.markdown:
        class: AppBundle\Service\Slugger
        arguments: ['_']
```

```php
$slugger = new \AppBundle\Service\Slugger('_');
```

A complex service (XML)

```xml
<!-- app/config/services.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services/services-1.0.xsd">
  <services>
    <service id="app.publisher"
             class="AppBundle\Service\Publisher">
      <argument type="service" id="app.markdown" />
      <argument type="service" id="app.slugger" />
    </service>
  </services>
</container>
```

```php
$publisher = new \AppBundle\Service\Publisher(
    new \AppBundle\Service\Markdown(), new \AppBundle\Service\Slugger('_')
);
```

A complex service (YML)

```yaml
# app/config/services.yml
services:
    # ...
    app.publisher:
        class: AppBundle\Service\Publisher
        arguments: ['@app.markdown', '@app.slugger']
```

The "@" character means that the string must be interpreted as the name of a service.

```php
$publisher = new \AppBundle\Service\Publisher(
    new \AppBundle\Service\Markdown(), new \AppBundle\Service\Slugger('_')
);
```

Service definition (arguments, XML)

```
<argument>ROLE_USER</argument>
<argument type="string">ROLE_USER</argument>
```

Strings

```
<argument type="constant">PDO::FETCH_NUM</argument>
```

Constants

```
<argument type="service" id="logger" />
```

Services

```
<argument type="collection">
    <argument key="cache_dir">../cache</argument>
    <argument key="debug" type="constant">true</argument>
</argument>
```

Arrays

Service definition (arguments, YML)

```yaml
arguments: ['ROLE_USER']
```

Strings

```
(Not supported)
```

Constants

```yaml
arguments: ['@logger']
```

Services

```yaml
arguments:
    options:
        cache_dir: '../cache'
        debug: true
```

Arrays

Optional services (XML)

```
<!-- app/config/services.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services/services-1.0.xsd">
  <services>
    <service id="app.mailer"
             class="AppBundle\Service\Mailer">
      <argument type="service" id="app.logger"
                on-invalid="ignore" />
    </service>
  </services>
</container>
```

If this service doesn't exist, the application won't trigger an exception.

```
$mailer = new \AppBundle\Service\Mailer(new \AppBundle\Service\Logger());
$mailer = new \AppBundle\Service\Mailer();
```

Optional services (YML)

```yaml
# app/config/services.yml
services:
    # ...
    app.mailer:
        class: AppBundle\Service\Mailer
        arguments: ['@?app.logger']
```

The "@?" syntax means that the string must interpreted as the name of an optional service. If the service doesn't exist, the application won't trigger an exception.

```php
$mailer = new \AppBundle\Service\Mailer(new \AppBundle\Service\Logger());
$mailer = new \AppBundle\Service\Mailer();
```

Optional services and constructors

```
namespace AppBundle\Service;
use AppBundle\Service\Logger;

class Mailer
{
    public function __construct(Logger $logger = null)
    {
        // ...
    }

    // ...
}
```

When using optional services, the **constructors** must be prepared to accept **null** arguments.

Service aliasing (XML)

```xml
<!-- app/config/services.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services/services-1.0.xsd">
  <services>
    <!-- https://github.com/erusev/parsedown -->
    <service id="app.markdown.parsedown" class="..." />
    <!-- https://github.com/thephpleague/commonmark -->
    <service id="app.markdown.commonmark" class="..." />
    <!-- https://github.com/michelf/php-markdown -->
    <service id="app.markdown.phpmarkdown" class="..." />

    <service id="app.markdown" alias="app.markdown.parsedown" />
 </services>
</container>
```

```php
$parser = $this->get('app.markdown');
```

Service aliasing (YML)

```yml
# app/config/services.yml
app.markdown.parsedown:
    class: ...
app.markdown.commonmark:
    class: ...
app.markdown.phpmarkdown:
    class: ...

app.markdown:
    alias: app.markdown.parsedown
```

```php
$parser = $this->get('app.markdown');
```

Private services (XML)

```
<!-- app/config/services.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services/services-1.0.xsd">
  <services>
    <service id="app.logger"
             class="AppBundle\Service\Logger"
             public="false" />
  </services>
</container>
```

Private services are meant to be used as arguments for other services. You shouldn't use private services in your own code.

Private services (YML)

```yaml
# app/config/services.yml
services:
    # ...
    app.logger:
        class: AppBundle\Service\Logger
        public: false
```

Private services are meant to be used as arguments for other services. You shouldn't use private services in your own code.

Display the list of public and private services

```
$ php app/console debug:container --show-private

Service ID                  Class name
annotation_reader           Doctrine\Common\Annotations\FileCacheReader
assetic.asset_manager       Assetic\Factory\LazyAssetManager
assetic.controller          Symfony\Bundle\AsseticBundle\Controller\AsseticController
assetic.filter.cssrewrite   Assetic\Filter\CssRewriteFilter
assetic.filter.jsqueeze     Assetic\Filter\JSqueezeFilter
assetic.filter.scssphp      Assetic\Filter\ScssphpFilter
assetic.filter_manager      Symfony\Bundle\AsseticBundle\FilterManager
assetic.request_listener    Symfony\Bundle\AsseticBundle\EventListener\RequestListener
cache_clearer               Symfony\Component\HttpKernel\CacheClearer\ChainCacheClearer
cache_warmer                Symfony\Component\HttpKernel\CacheWarmer\CacheWarmerAggregate

...

validator                   Symfony\Component\Validator\ValidatorInterface
var_dumper.cloner           Symfony\Component\VarDumper\Cloner\VarCloner
```

Private services can be inlined

- If a private service is **used once**, Symfony **inlines** it automatically to improve performance.
- Inlined services won't appear in **container:debug** command output.
- Inlined services cannot be accessed through the service container (**$this->get('service')**)

Full details:
symfony.com/doc/current/components/dependency_injection/advanced.html#inlined-private-services

Advanced service definition

Calling methods when creating services (XML)

```
<!-- app/config/services.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services/services-1.0.xsd">
  <services>
   <service id="app.mailer" class="AppBundle\Service\Mailer">
      <call method="initialize" />
      <call method="setLogger">
         <argument type="service" id="app.logger" />
      </call>
   </service>
  </services>
</container>
```

This is the way to use setter injection

```
$mailer = new \AppBundle\Service\Mailer();
$mailer->initialize();
$mailer->setLogger(new \AppBundle\Service\Logger());
```

Calling methods when creating services (YML)

```yaml
# app/config/services.yml
services:
    app.mailer:
        class: AppBundle\Service\Mailer
        calls:
            - [initialize]
            - [setLogger, ["@app.logger"]]
```

This is the way to use setter injection

```php
$mailer = new \AppBundle\Service\Mailer();
$mailer->initialize();
$mailer->setLogger(new \AppBundle\Service\Logger());
```

Using factories to create services (XML)

```
<!-- app/config/services.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<container ...>
  <services>
    <service id="app.mailer" class="AppBundle\Service\Mailer">
      <factory class="AppBundle\Service\MailerFactory"
               method="createMailer" />
    </service>
    <service id="app.mailer" class="AppBundle\Service\Mailer">
      <factory service="app.mailer_factory"
               method="createMailer" />
    </service>
  </services>
</container>
```

Using factories to create services (YML)

```yaml
# app/config/services.yml
services:
    app.mailer:
        class: AppBundle\Service\Mailer
        factory: [AppBundle\Service\MailerFactory, createMailer]

     app.mailer:
        class: AppBundle\Service\Mailer
        factory: ['@app.mailer_factory', createMailer]
```

Using parent services (XML)

```xml
<!-- app/config/services.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<container ...>
 <services>
   <service id="app.mailer" abstract="true">
      <call method="setLogger">
         <argument type="service" id="app.logger" />
      </call>

      <call method="setTemplating">
         <argument type="service" id="app.templating" />
      </call>
   </service>

   <service id="app.newsletter" class="AppBundle\Service\Mailer\Newsletter"
            parent="app.mailer" />

   <service id="app.notifier" class="AppBundle\Service\Mailer\Notifier"
            parent="app.mailer" />
 </services>
</container>
```

Using parent services (YML)

```yaml
# app/config/services.yml
services:
    # ...
    app.mailer:
        abstract:  true
        calls:
            - [setLogger, ["@app.logger"]]
            - [setTemplating, ["@app.templating"]]

    app.newsletter:
        class:  AppBundle\Service\Mailer\Newsletter
        parent: app.mailer

    app.notifier:
        class:  AppBundle\Service\Mailer\Notifier
        parent: app.mailer
```

Parent services are declared as **abstract**. They are not retrieved or used by other services and they don't define the **class** option

Children services which define the **parent** option inherit the arguments and method calls from that parent service.

SensioLabs