Dependency Injection with PHP 5.3... with a bit of PHP 5.4

Fabien Potencier
@fabpot
fabien@symfony.com

Dependency Injection

A real world « web » example

In most web applications, you need to manage the user preferences

- The user language
- Whether the user is authenticated or not
- The user credentials
- ...

This can be done with a User object

- `setLanguage()`, `getLanguage()`
- `setAuthenticated()`, `isAuthenticated()`
- `addCredential()`, `hasCredential()`
- ...

The User information
need to be persisted
between HTTP requests

We use the PHP session for the Storage

```php
class SessionStorage
{
    function __construct($cookieName = 'PHP_SESS_ID')
    {
        session_name($cookieName);
        session_start();
    }

    function set($key, $value)
    {
        $_SESSION[$key] = $value;
    }

    // ...
}
```

```php
class User
{
    protected $storage;

    function __construct()
    {
        $this->storage = new SessionStorage();
    }

    function setLanguage($language)
    {
        $this->storage->set('language', $language);
    }

    // ...
}

$user = new User();
```

Very hard to customize

Very easy to use

```php
class User
{
    protected $storage;

    function __construct($storage)
    {
        $this->storage = $storage;
    }
}

$storage = new SessionStorage();
$user = new User($storage);
```

Very easy to customize

Slightly more difficult to use

That's Dependency Injection

Nothing more

Let's understand why the first example is not a good idea

I want to change the session cookie name

```php
class User
{
    protected $storage;

    function __construct()
    {
        $this->storage = new SessionStorage('SESSION_ID');
    }

    function setLanguage($language)
    {
        $this->storage->set('language', $language);
    }

    // ...
}

$user = new User();
```

Hardcode it in the User class

```php
class User
{
    protected $storage;

    function __construct()
    {
        $this->storage = new SessionStorage(STORAGE_SESSION_NAME);
    }
}

define('STORAGE_SESSION_NAME', 'SESSION_ID');

$user = new User();
```

Add a global configuration?

```php
class User
{
    protected $storage;

    function __construct($sessionName)
    {
        $this->storage = new SessionStorage($sessionName);
    }
}

$user = new User('SESSION_ID');
```

Configure via User?

```php
class User
{
    protected $storage;

    function __construct($storageOptions)
    {
        $this->storage = new
SessionStorage($storageOptions['session_name']);
    }
}

$user = new User(
  array('session_name' => 'SESSION_ID')
);
```

Configure with an array?

I want to change the session storage implementation

Filesystem
MySQL
Memcached
...

```php
class User
{
    protected $storage;

    function __construct()
    {
        $this->storage = Registry::get('session_storage');
    }
}

$storage = new SessionStorage();
Registry::set('session_storage', $storage);
$user = new User();
```

Use a global registry object?

Now, the User depends on the Registry

Instead of harcoding
the Storage dependency
inside the User class constructor

Inject the Storage dependency
in the User object

```php
class User
{
    protected $storage;

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

```php
$storage = new SessionStorage('SESSION_ID');
$user = new User($storage);
```

What are the advantages?

Use different Storage strategies

```php
class User
{
    protected $storage;

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

Use a different Storage engine

```php
$storage = new MySQLSessionStorage('SESSION_ID');
$user = new User($storage);
```

Configuration becomes natural

```php
class User
{
    protected $storage;

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

Configuration is natural

```php
$storage = new MySQLSessionStorage('SESSION_ID');
$user = new User($storage);
```

Wrap third-party classes (Interface / Adapter)

```php
class User
{
    protected $storage;

    function __construct(SessionStorageInterface $storage)
    {
        $this->storage = $storage;
    }
}

interface SessionStorageInterface
{
    function get($key);

    function set($key, $value);
}
```

Add an interface

Mock the Storage object (for testing)

```php
class User
{
    protected $storage;

    function __construct(SessionStorageInterface $storage)
    {
        $this->storage = $storage;
    }
}
```

Mock the Session

```php
class SessionStorageForTests implements SessionStorageInterface
{
    protected $data = array();

    static function set($key, $value)
    {
        self::$data[$key] = $value;
    }
}
```

Use different Storage strategies

Configuration becomes natural

Wrap third-party classes (Interface / Adapter)

Mock the Storage object (for testing)

Easy without changing the User class

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

http://www.picocontainer.org/injection.html

```php
$storage = new SessionStorage();

// constructor injection
$user = new User($storage);

// setter injection
$user = new User();
$user->setStorage($storage);

// property injection
$user = new User();
$user->storage = $storage;
```

A slightly more complex web example

```php
$request = new Request();
$response = new Response();

$storage = new FileSessionStorage('SESSION_ID');
$user = new User($storage);
```

```php
$cache = new FileCache(
    array('dir' => __DIR__.'/cache')
);
$routing = new Routing($cache);
```

```php
class Application
{
    function __construct()
    {
        $this->request = new WebRequest();
        $this->response = new WebResponse();

        $storage = new FileSessionStorage('SESSION_ID');
        $this->user = new User($storage);

        $cache = new FileCache(
            array('dir' => dirname(__FILE__).'/cache')
        );
        $this->routing = new Routing($cache);
    }
}

$application = new Application();
```

Back to square 1

```php
class Application
{
    function __construct()
    {
        $this->request = new WebRequest();
        $this->response = new WebResponse();

        $storage = new FileSessionStorage('SESSION_ID');
        $this->user = new User($storage);

        $cache = new FileCache(
            array('dir' => dirname(__FILE__).'/cache')
        );
        $this->routing = new Routing($cache);
    }
}

$application = new Application();
```

We need a Container

Describes objects
and their dependencies

Instantiates and configures
objects on-demand

A container
SHOULD be able to manage
ANY PHP object (POPO)

The objects MUST not know
that they are managed
by a container

- Parameters
  - The SessionStorageInterface implementation we want to use (the class name)
  - The session name
- Objects
  - SessionStorage
  - User
- Dependencies
  - User depends on a SessionStorageInterface implementation

Let's build a simple container with PHP 5.3

DI Container

Managing parameters

```php
class Container
{
    protected $parameters = array();

    public function setParameter($key, $value)
    {
        $this->parameters[$key] = $value;
    }

    public function getParameter($key)
    {
        return $this->parameters[$key];
    }
}
```

Decoupling

```php
$container = new Container();
$container->setParameter('session_name', 'SESSION_ID');
$container->setParameter('storage_class', 'SessionStorage');
```

Customization

```php
$class = $container->getParameter('storage_class');
$sessionStorage = new $class($container-
>getParameter('session_name'));
$user = new User($sessionStorage);
```

Objects creation

Using PHP magic methods

```php
class Container
{
    protected $parameters = array();

    public function __set($key, $value)
    {
        $this->parameters[$key] = $value;
    }

    public function __get($key)
    {
        return $this->parameters[$key];
    }
}
```

Interface is cleaner

```
$container = new Container();
$container->session_name = 'SESSION_ID';
$container->storage_class = 'SessionStorage';
```

```
$sessionStorage = new $container->storage_class($container-
>session_name);
$user = new User($sessionStorage);
```

DI Container

Managing objects

We need a way to describe how to create objects, without actually instantiating anything!

Anonymous functions to the rescue!

A lambda is a function
defined on the fly
with no name

```php
function () { echo 'Hello world!'; };
```

A lambda can be stored in a variable

```php
$hello = function () { echo 'Hello world!'; };
```

And then it can be used
as any other PHP callable

```
$hello();
```

```
call_user_func($hello);
```

You can also pass a lambda
as an argument to a function or method

```
function foo(Closure $func) {
    $func();
}

foo($hello);
```

```php
$hello = function ($name) { echo 'Hello '.$name; };

$hello('Fabien');

call_user_func($hello, 'Fabien');
```

```php
function foo(Closure $func, $name) {
    $func($name);
}

foo($hello, 'Fabien');
```

DI Container

Managing objects

```php
class Container
{
    protected $parameters = array();
    protected $objects = array();

    public function __set($key, $value)
    {
        $this->parameters[$key] = $value;
    }

    public function __get($key)
    {
        return $this->parameters[$key];
    }

    public function setService($key, Closure $service)
    {
        $this->objects[$key] = $serv
    }

    public function getService($key)
    {
        return $this->objects[$key]($this);
    }
```

Store a lambda able to create the object on-demand

Ask the closure to create the object and pass the current Container

Description

```php
$container = new Container();
$container->session_name = 'SESSION_ID';
$container->storage_class = 'SessionStorage';
$container->setService('user', function ($c) {
    return new User($c->getService('storage'));
});
$container->setService('storage', function ($c) {
    return new $c->storage_class($c->session_name);
});
```

Creating the User
is now as easy as before

```php
$user = $container->getService('user');
```

Order does not matter

```php
$container = new Container();

$container->setService('storage', function ($c) {
    return new $c->storage_class($c->session_name);
});

$container->setService('user', function ($c) {
    return new User($c->getService('storage'));
});

$container->session_name = 'SESSION_ID';

$container->storage_class = 'SessionStorage';
```

```php
class Container
{
    protected $values = array();

    function __set($id, $value)
    {
        $this->values[$id] = $value;
    }

    function __get($id)
    {
        if (is_callable($this->values[$id])) {
            return $this->values[$id]($this);
        } else {
            return $this->values[$id];
        }
    }
}
```

Simplify the code

Unified interface

```php
$container = new Container();
$container->session_name = 'SESSION_ID';
$container->storage_class = 'SessionStorage';
$container->user = function ($c) {
    return new User($c->storage);
};
$container->storage = function ($c) {
    return new $c->storage_class($c->session_name);
};
```

```php
$user = $container->user;
```

DI Container

Scope

For some objects, like the user, the container must always return the same instance

```
spl_object_hash($container->user)
```

```
!==
```

```
spl_object_hash($container->user)
```

```php
$container->user = function ($c) {
    static $user;

    if (is_null($user)) {
        $user = new User($c->storage);
    }

    return $user;
};
```

```
spl_object_hash($container->user)
```

```
spl_object_hash($container->user)
```

```php
$container->user = $container->asShared(function ($c) {
    return new User($c->storage);
});
```

A closure is a lambda
that remembers the context
of its creation . . .

```php
class Article
{
    public function __construct($title)
    {
        $this->title = $title;
    }

    public function getTitle()
    {
        return $this->title;
    }
}

$articles = array(
    new Article('Title 1'),
    new Article('Title 2'),
);
```

```php
$mapper = function ($article) {
    return $article->getTitle();
};
```

```php
$titles = array_map($mapper, $articles);
```

```php
$method = 'getTitle';

$mapper = function ($article) use($method) {
    return $article->$method();
};
```

```php
$method = 'getAuthor';

$titles = array_map($mapper, $articles);
```

```php
$mapper = function ($method) {
    return function ($article) use($method) {
        return $article->$method();
    };
};
```

```php
$titles = array_map($mapper('getTitle'), $articles);
```

```php
$authors = array_map($mapper('getAuthor'), $articles);
```

```php
$container->user = $container->asShared(function ($c) {
    return new User($c->storage);
});
```

```php
function asShared(Closure $lambda)
{
    return function ($container) use ($lambda)
    {
        static $object;

        if (is_null($object)) {
            $object = $lambda($container);
        }

        return $object;
    };
}
```

```php
class Container
{
    protected $values = array();

    function __set($id, $value)
    {
        $this->values[$id] = $value;
    }

    function __get($id)
    {
        if (!isset($this->values[$id])) {
            throw new InvalidArgumentException(sprintf('Value
"%s" is not defined.', $id));
        }

        if (is_callable($this->values[$id])) {
            return $this->values[$id]($this);
        } else {
            return $this->values[$id];
        }
    }
}
```

Error management

```php
class Container
{
    protected $values = array();

    function __set($id, $value)
    {
        $this->values[$id] = $value;
    }

    function __get($id)
    {
        if (!isset($this->values[$id])) {
            throw new InvalidArgumentException(sprintf('Value "%s" is not defined.', $id));
        }

        if (is_callable($this->values[$id])) {
            return $this->values[$id]($this);
        } else {
            return $this->values[$id];
        }
    }

    function asShared($callable)
    {
        return function ($c) use ($callable) {
            static $object;

            if (is_null($object)) {
                $object = $callable($c);
            }

            return $object;
        };
    }
}
```

40 LOC for a fully-featured container

```php
$container = new Container();
$container->session_name = 'SESSION_ID';
$container->storage_class = 'SessionStorage';
$container->user = $container->asShared(function ($c) {
    return new User($c->storage);
});
$container->storage = $container->asShared(function ($c) {
    return new $c->storage_class($c->session_name);
});
```

github.com/fabpot/pimple

...and with PHP 5.4?

```php
$container = new Container();
$container->session_name = 'SESSION_ID';
$container->storage_class = 'SessionStorage';
$container->user = $container->asShared(function () {
    return new User($this->storage);
});
$container->storage = $container->asShared(function () {
    return new $this->storage_class($this->session_name);
});
```

```php
$title = function () {
    echo $this->title;
};

print get_class($title);
// \Closure
```

```php
class Article
{
    public $title;

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

```php
$a = new Article('Title 1');

$title = function () {
    echo $this->title;
};

$getTitle = $title->bindTo($a);
$getTitle();
```

```php
class Article
{
    private $title;

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

```php
$a = new Article('Title 1');

$title = function () {
    echo $this->title;
};

$getTitle = $title->bindTo($a, $a);
$getTitle();
```

wiki.php.net/rfc/closures/object-extension

```php
if (is_callable($this->values[$id])) {
    $value = $value->bindTo($this);

    return $this->values[$id]();
} else {
    return $this->values[$id];
}
```

```php
function asShared($callable)
{
    $callable = $callable->bindTo($this);

    return function () use ($callable) {
        static $object;

        if (is_null($object)) {
            $object = $callable();
        }

        return $object;
    };
}
```

```php
$container = new Container();
$container->session_name = 'SESSION_ID';
$container->storage_class = 'SessionStorage';
$container->user = $container->asShared(function () {
    return new User($this->storage);
});
$container->storage = $container->asShared(function () {
    return new $this->storage_class($this->session_name);
});
```

A DI Container

does NOT manage

ALL your objects

A DI Container / Service Container manages your SERVICES

Good rule of thumb:

It manages "Global" objects

Objects with only one instance (!= Singletons)

LIKE

a User, a Request,
a database Connection, a Logger, . . .

UNLIKE

Model objects (a Product, a blog Post, . . .)

Performance

The Symfony2 Dependency Injection Component

Rock-solid implementation of a DIC in PHP 5.3

	Lines		
Total		88.11%	1274 / 1446
Dumper		98.81%	500 / 506
Loader		71.22%	391 / 549
Builder.php		97.35%	147 / 151
BuilderConfiguration.php		100.00%	63 / 63
Container.php		97.18%	69 / 71
ContainerInterface.php		100.00%	1 / 1
Definition.php		100.00%	34 / 34
FileResource.php		100.00%	8 / 8
Parameter.php		100.00%	4 / 4
Reference.php		100.00%	6 / 6
ResourceInterface.php		100.00%	1 / 1
SimpleXMLElement.php		96.15%	50 / 52

At the core of the Symfony2 framework

```php
namespace Symfony\Component\DependencyInjection;

interface ContainerInterface
{
    function set($id, $service, $scope = self::SCOPE_CONTAINER);
    function get($id, $invalidBehavior =
self::EXCEPTION_ON_INVALID_REFERENCE);
    function has($id);
    function getParameter($name);
    function hasParameter($name);
    function setParameter($name, $value);

    function enterScope($name);
    function leaveScope($name);
    function addScope(ScopeInterface $scope);
    function hasScope($name);
    function isScopeActive($name);
}
```

Very flexible

Configuration in PHP, XML, YAML, or INI

```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Reference;

$sc = new ContainerBuilder();
$sc->setParameter('session_name', 'SESSION_ID');
$sc->setParameter('storage_class', 'SessionStorage');
$sc
    ->register('user', 'User')
    ->addArgument(new Reference('storage'))
;
$sc->register('storage', '%storage_class%')
    ->addArgument('%session_name%')
;

$sc->get('user');
```

```
parameters:
    session_name:  SESSION_ID
    storage_class: SessionStorage

services:
  user:
    class: User
    arguments: [@storage]
storage:
    class: %storage_class%
    arguments: [%session_name%]
```

```php
use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;
use Symfony\Component\Config\FileLocator;

$sc = new ContainerBuilder();

$loader = new YamlFileLoader($sc, new FileLocator(__DIR__));
$loader->load('services.yml');
```

```xml
<container xmlns="http://symfony.com/schema/dic/services">
    <parameters>
        <parameter key="session_name">SESSION_ID</parameter>
        <parameter key="storage_class">SessionStorage</parameter>
    </parameters>

    <services>
        <service id="user" class="User">
            <argument type="service" id="storage" />
        </service>

        <service id="storage" class="%storage_class%">
            <argument>%session_name%</argument>
        </service>
    </services>
</container>
```

```xml
<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">
    <parameters>
        <parameter key="session_name">SESSION_ID</parameter>
        <parameter key="storage_class">SessionStorage</parameter>
    </parameters>
    <services>
        <service id="user" class="User">
            <argument type="service" id="storage" />
        </service>

        <service id="storage" class="%storage_class%">
            <argument>%session_name%</argument>
        </service>
    </services>
</container>
```

```xml
<container xmlns="http://symfony-project.org/schema/dic/services">
  <import resource="parameters.yml" />
  <import resource="parameters.ini" />
  <import resource="services.xml" />
</container>
```

```yaml
imports:
  - { resource: parameters.yml }
  - { resource: parameters.ini }
  - { resource: services.xml }
```

As fast as it can be

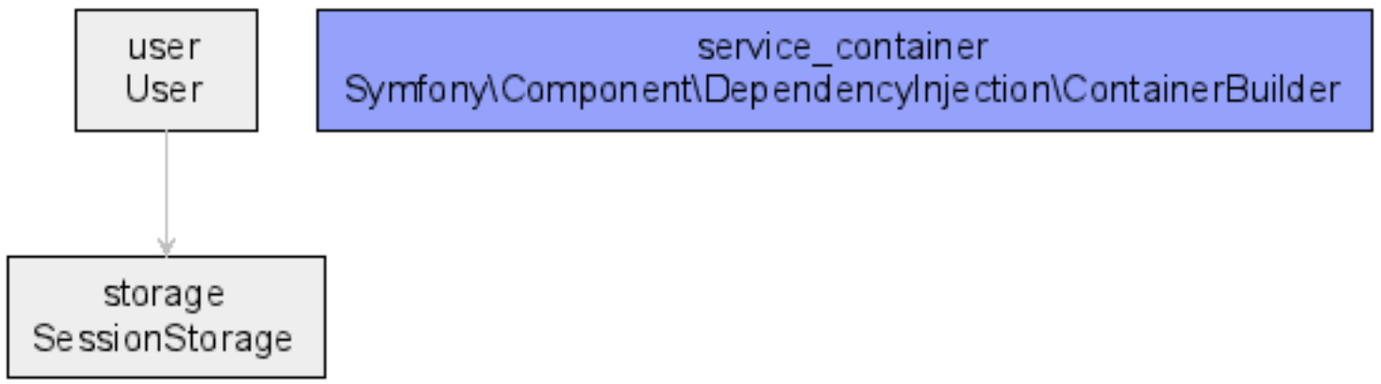
The container can be
"compiled" down
to plain PHP code

```php
use Symfony\Component\DependencyInjection\Dumper\PhpDumper;

$dumper = new PhpDumper($sc);
echo $dumper->dump();
```

```php
class ProjectServiceContainer extends Container
{
    public function __construct()
    {
        parent::__construct(new ParameterBag($this->getDefaultParameters()));
    }

    protected function getStorageService()
    {
        $class = $this->getParameter('storage_class');
        return $this->services['storage'] = new $class($this->getParameter('session_name'));
    }

    protected function getUserService()
    {
        return $this->services['user'] = new \User($this->get('storage'));
    }

    protected function getDefaultParameters()
    {
        return array(
            'session_name' => 'SESSION_ID',
            'storage_class' => 'SessionStorage',
        );
    }
}
```

```php
$dumper = new GraphvizDumper($sc);
echo $dumper->dump();
```

```
$ doc -Tpng -o sc.png sc.dot
```



There is more...

Semantic configuration
Scope management
Compiler passes (optimizers)
Aliases
Abstract definitions
...

Remember, most of the time, you don't need a Container to use Dependency Injection

You can start to use and benefit from Dependency Injection today

by implementing it
in your projects

by using externals libraries
that already use DI
without the need of a container

Symfony
Zend Framework
Doctrine
Swift Mailer
...

Questions?

Sensio S.A.
92-98, boulevard Victor Hugo
92 115 Clichy Cedex
FRANCE
Tél. : +33 1 40 99 80 80

Contact

Fabien Potencier
fabien.potencier at sensio.com

http://sensiolabs.com/

http://symfony.com/

http://fabien.potencier.org/