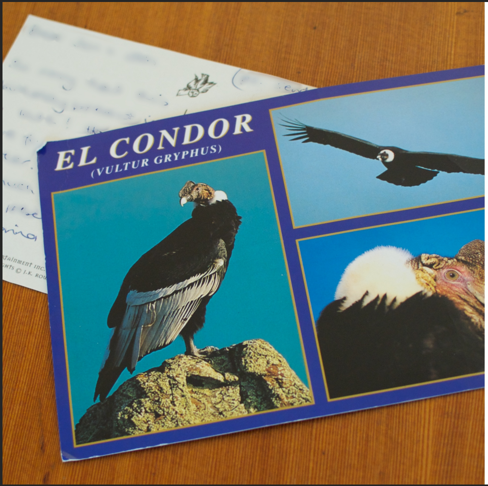
Dependency Injection in ZF2

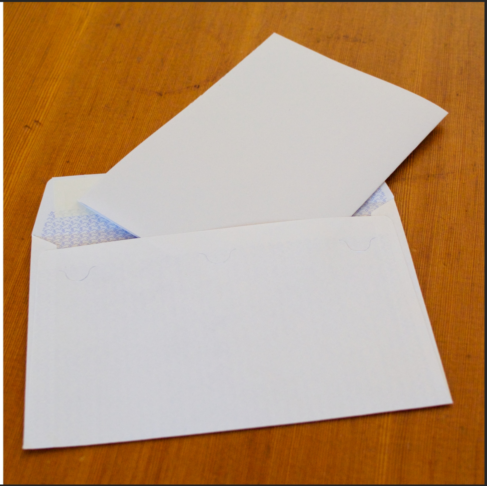
Rob Allen ~ October 2014

Dependency Injection enables loose coupling and loose coupling makes code more maintainable

Mark Seemann

Coupling





Benefits of loose coupling

- *Maintainability* - Classes are more clearly defined
- *Extensibility* - easy to recompose application
- *Testability* - isolate what you’re testing

A worked example

Class A needs class B in order to work.

```
class Letter
{
    protected $paper;

    public function __construct()
    {
        $this->paper = new WritingPaper();
    }
}

// usage:
$letter = new Letter();
$letter->write("Dear John, ...");
```

Pros and cons:

Pros:

- Very simple to use

Cons:

- Cannot test `Letter` in isolation
- Cannot change `$paper`

  This is *tight coupling*

The problem with coupling

- How do we change the paper size?
- How do we change the type of paper?

Method parameters?

```php
class Letter
{
    protected $paper;

    public function __construct($size)
    {
        $this->paper = new WritingPaper($size);
    }
}

// usage:
$letter = new Letter('A4');
$letter->write("Dear John, ...");
```

Use a Registry?

```
class Letter
{
    protected $paper;

    public function __construct()
    {
        $this->paper = Zend_Registry::get('paper');
    }
}

// usage:
Zend_Registry::set('paper', new AirmailPaper('A4'));

$letter = new Letter();
$letter->write("Dear John, ...");
```

Inject the dependency!

Injection

```php
class Letter
{
    protected $paper;

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

// usage:
$letter = new Letter(new WritingPaper('A4'));
$letter->write("Dear John, ...");
```

This is also known as

Inversion of Control

Pros and cons:

Pros:

- Decoupled `$paper` from `Letter`:
    - Can change the type of paper
    - Natural configuration of the `Paper` object
- Can test `Letter` independently

Cons:

- Burden of construction of `$paper` is on the user

Types of injection

Constructor injection:

```
$letter = new Letter($paper);
```

Setter injection:

```
$letter = new Letter();
$letter->setPaper($paper);
```

Interface injection:

```
$letter = new Letter();
if (!$letter instanceof PaperInterface) {
    $letter->setPaper(new WritingPaper())
}
```

Note

Too many constructor parameters is a *code smell*

Two-phase construction is *Bad(TM)*

Rule of thumb

- Constructor injection for required dependencies
- Setter injection for optional dependencies

How about usage?

```
$paper = new AirmailPaper('A4');
$envelope = new Envelope('DL');
$letter = new Letter($paper, $envelope);

$letter->write("Dear John, ...");
```

Setup of dependencies gets *tedious* quickly

Dependency Injection Container

A DIC is an object that handles the creation of objects and their dependencies for you

Dependency resolution can be *automatic* or *configured*

DICs are *optional*

Dependency Injection Container

- Creates objects on demand
- Manages construction of an object’s dependencies
- Separates of configuration from construction
- Can allow for shared objects

That’s all there is to DI

Remember that I said that DICs are *optional*?

Not in ZF2, they’re not!

Zend\ServiceManager

- ZF2’s Dependency Injection Container
- Used *extensively* within ZF2
- Explicit & easy to understand (no magic!)
- Promotes low-coupling & re-usability
- Easy to swap out ZF2 classes with your own

The process

1. Register your services
2. The `Zend\Mvc` operation results in your services being instantiated as required
3. Your app runs and does it’s stuff!

Registering services

Configure your services:

1. in an array in a config file
2. in a method within a *Module* class
3. direct method call

in config

```php
// Application/config/module.config.php:
return [
  'service_manager' => [
    'invokables' => [
        'session' => 'Zend\Session\Storage',
    ],
    'factories' => [
        'db' => 'My\DBAL\DriverManagerFactory',
    ],
  ]
];
```

in a Module class

```php
// Application::Module
public function getServiceConfig()
{
    return [
        'factories' => [
            'UserMapper' =>  function ($sm) {
                $db = $sm->get('db');
                return new UserMapper($db);
            },
        ],
    ];
}
```

Types of services

Instances	`services`
Constructor-less classes	`invokables`
Objects with dependencies	`factories`
Aliased services	`aliases`
Automated initialization	`initializers`
Multiple related objects	`abstract_factories`

Instances

```
// programmatically
$sm->setService('foo', $fooInstance);

// configuration
'services' => [
    'foo' => new Foo(),
]
```

Invokables

```
// programmatically
$sm->setInvokableClass('foo', 'Bar\Foo');

// configuration
'invokables' => [
    'foo' => 'Bar\Foo',
]
```

Factories

```
// programmatically
$sm->setFactory('foo', function($sm) {
        $dependency = $sm->get('Dependency')
        return new Foo($dependency);
    });

// configuration
'factories' => [
  'foo' => function($sm) { //.. },
  'bar' => 'Some\Static::method',
  'baz' => 'Class\Implementing\FactoryInterface',
  'bat' => 'Class\Implementing\Invoke',
]
```

Aliases

```
// programmatically
$sm->setAlias('foo_db', 'db_adapter');

// configuration
'aliases' => [
    'foo_db', 'db_adapter', // alias of a service
    'bar_db', 'foo_db',     // alias of an alias
]

// All the same instance
$db = $sm->get('db_adapter');
$db = $sm->get('foo_db');
$db = $sm->get('bar_db');
```

Initializers

```
// programmatically
$sm->addInitializer($callback);

// configuration
'initializers' => [
  $instance,
  $callback,
  'Class\Implementing\InitializerInterface',
  'Class\That\Implements\__invoke',
]
```

An initializer

```
function($instance, $sm) {
    if ($instance instanceof FooAwareInterface) {
        return;
    }
    $instance->setFoo($sm->get('foo'));
},
```

Abstract factories

```
array(
    'abstract_factories' => [
        'Class\Implementing\AbstractFactoryInterface'
        $someAbstractFactoryInstance,
    ]
);
```

An abstract factory

```
class MyClassLoader implements AbstractFactoryInterface
{
  public function canCreateServiceWithName(
    ServiceLocatorInterface $services, $name,
    $requestedName
  ) {
    // return true or false
  }

  public function createServiceWithName(/* same sig */)
  {
    // return instance required
  }
}
```

Real-world configuration

```
'service_manager' => [
  'invokables' => [
      'Comment\CommentMapper' => 'Comment\CommentMapper',
  ],
  'factories' => [
    'Zend\Db\Adapter\Adapter' =>
        'Zend\Db\Adapter\AdapterServiceFactory',
    'site_navigation' =>
        'Application\NavigationSiteNavigationFactory',
  ],
],
```

Application\Module

```
public function getServiceConfig()
{
  return [
    'factories' => [
      'LogWriter' => function ($sm) {
        $file = 'log_' . date('F') . '.txt';
        return new LogWriterStream("var/log/$file");
      },
      'Zend\Log' => function ($sm) {
        $log = new Logger();
        $log->addWriter($sm->get('LogWriter');
        return $log;
      },
    ],
  ];
}
```

User\Module

```
public function getServiceConfig()
{
  return [
    'initializers' => [
      function ($instance, $sm) {
        if ($instance instanceof UserAwareInterface) {
          $authService = $sm->get('zfcuser_auth_service');
          $user        = $authService->getIdentity();
          $instance->setUser($user);
        }
      }
    ],
  ];
}
```

Using our services in a controller

Controller configuration

```
// module.config.php
'controllers' => [
  'invokables' => [
    'Application\Controller\Index' =>
        'Application\Controller\IndexController',
    'Application\Controller\Blog' =>
        'Application\Controller\BlogController',
  ],
],
```

Controller set-up is simply another *service manager*!

Inject into your controller

```
public function getControllerConfig()
{
  return [
    'factories' => [
      'Application\Controller\Blog' => function ($csm) {
        $sm = $csm->getServiceLocator();
        $blogs = $sm->get('Application\BlogMapper');
        $comments = $sm->get('Comment\Mapper');

        return new BlogController($blogs, $comments);
      },
    ],
  ];
}
```

Everything is a ServiceManager!

Top 10

Name	Config key
`ServiceManager`	`service_manager`
`ControllerManager`	`controllers`
`ControllerPluginManager`	`controller_plugins`
`ViewHelperManager`	`view_helpers`
`FormElementManager`	`form_elements`
`InputFilterManager`	`input_filters`
`FilterManager`	`filters`
`ValidatorManager`	`validators`
`RoutePluginManager`	`route_manager`
`HydratorPluginManager`	`hydrators`

There are 42 plugin managers in ZF2

View helpers

```
'view_helpers' => array(
    'invokables' => array(
        'formRow' => 'Application\View\Helper\FormRow',
    ),
    'factories' => array(
        'lastComment' => 'Comment\View\Helper\LastComment',
    ),
),
```

View helpers

```php
class LastComment implements FactoryInterface
{
    public function createService(
        ServiceLocatorInterface $serviceLocator)
    {
        $locator = $sm->getServiceLocator();
        $mapper  = $locator->get('Comment\CommentMapper');
        return new LastComment($mapper);
    }
}
```

Learn once,
reuse everywhere!

A note on Service Location

```
class BlogController extends AbstractActionController
{
    protected $mapper;

    public function __construct()
    {
        $sm = $this->getServiceLocator();
        $this->mapper  = $sm->get('Blog\BlogMapper');
    }
}
```

Service Location

- Application pulls its dependencies in when it needs them
- Still decouples concrete implementations

Don’t do this!

Recap

Dependency injection promotes:

- loose coupling
- easier testing
- separation of configuration from usage

Recap

Zend\ServiceManager is used *everywhere*

Six ways to configure:

- `invokables`
- `factories`
- `aliases`
- `initializers`
- `services`
- `abstract_factories`

Thank you!

https://m.joind.in/talk/66651