Zend_Di - Federico Cargnelutti

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

Zend Framework: Zend_Di Component Proposal

Proposed Component Name	Zend_Di
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend_Di
Proposers	Federico Cargnelutti Matthew Weier O'Phinney, Zend Liaison
Revision	0.1 - 21 November 2007: Initial Proposal. (wiki revision: 68)

Table of Contents

1. Overview
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
4. Dependencies on Other Framework Components
5. Theory of Operation
6. Milestones / Tasks
7. Class Index
8. Use Cases
Real-life example
9. Class Skeletons

1. Overview

Zend_Di is a dependency injector component. It minimizes coupling between groups of classes, makes unit testing much simpler, and provides an easy way to re-configure a package to use custom implementations of components. The architecture of the Zend_Di component is based on the following concepts:

- Dependency injection is a technique that consists of passing objects via the constructor or setter methods.
- The Container provides an easy way of re-configuring a package to use custom implementations of components.
- Responsibility for object management is taken over by whatever container is being used to manage those objects.

Benefits of using a DI Container:

- Easy best practice unit testing
- Component reuse
- Centralized configuration
- Clean and declarative architecture
- Maintainability and adaptability

2. References

- Martin Fowler Dependency Injection pattern
- PicoContainer

- NanoContainer

3. Component Requirements, Constraints, and Acceptance Criteria

- This component **will** use Reflection to wire dependencies.
- This component **will** use Configuration to wire dependencies.
    - This component **must** support PHP, XML and INI configuration files.
- This component **must** allow all components to be constructed using Zend_Config instances.

4. Dependencies on Other Framework Components

- Zend_Config (optional)
- Zend_Exception
- Zend_Loader

5. Theory of Operation

Zend_Di provides generic factory classes that instantiate instances of classes. These instances are then configured by the container, allowing construction logic to be reused on a broader level. For example:

```
$components = array(
 'Foo' => array(
  'class'        => 'Zend_Foo',
  'arguments'    => array(
   '__construct' => 'ComponentA',
  ),
 ),
 'ComponentA' => array(
  'class'        => 'Zend_Foo_Component_A',
  'instanceof'   => 'Zend_Foo_Component_Interface',
 ),
 'ComponentB' => array(
  'class'        => 'Zend_Foo_Component_B',
  'instanceof'   => 'Zend_Foo_Component_Interface',
 ),
);

$config = new Zend_Config($components);
// $config = new Zend_Config_Xml('components.xml', 'staging');

$di = new Zend_Di_Container($config);

// Create an instance of Zend_Foo and injects Zend_Foo_Component_A via the constructor method
$foo = $di->loadClass('Foo')->newInstance();
```

Once we separate configuration from use, we can easily test the Car with different Engines. It's just a matter of re-configuring the package and injecting Zend_Car_Parts_Engine_Gas instead of Zend_Car_Parts_Engine_Fuel.

6. Milestones / Tasks

- Milestone 1: [DONE] Design interface
- Milestone 2: [DONE] Write proposal
- Milestone 3: [DONE] Gather feedback and revise design as necessary

- Milestone 4: Review by the Zend team
- Milestone 5: Develop full implementation and unit tests
- Milestone 6: Documentation
- Milestone 7: Future enhancements

7. Class Index

- Zend_Di
- Zend_Di_Container
- Zend_Di_Factory
- Zend_Di_Reflection
- Zend_Di_Parameter
- Zend_Di_Data
- Zend_Di_Registry
- Zend_Di_Storage_Interface
- Zend_Di_Storage_Object
- Zend_Di_Storage_Exception
- Zend_Di_Exception

8. Use Cases

Zend_Di handles injections via the constructor or setters methods. In addition, the component allows the user to map out specifications for components and their dependencies in a configuration file and generate the objects based on that specification.

Assembling Objects Using Reflection

UC-01

Assembling objects using Zend_Di_Reflection

```
class Zend_Foo {
    public function __construct(Zend_Foo_Component_A $componentA) {}
    public function setComponentA(Zend_Foo_Component_Interface $component) {}
    public function setComponentB(Zend_Foo_Component_Interface $component) {}
    public function setComponentC(Zend_Foo_Component_C $componentC) {}
}

$di = new Zend_Di_Reflection();
$di->addComponent('Zend_Foo_Component_A', array('__construct', 'setComponentA'));
$di->addComponent(new Zend_Foo_Component_B(), array('setComponentB'));
$di->addComponent(new Zend_Foo_Component_C());

$di->loadClass('Zend_Foo')->newInstance();
$foo = $di->getComponent('Zend_Foo');
```

UC-02

Assembling objects using Zend_Di_Container

```
class Zend_Foo {
    public function __construct(Zend_Foo_Component_A $componentA) {}
    public function setComponentA(Zend_Foo_Component_Interface $component) {}
    public function setComponentB(Zend_Foo_Component_Interface $component) {}
    public function setComponentC(Zend_Foo_Component_C $componentC) {}
}

$di = new Zend_Di_Container();

$di->loadClass('Zend_Foo')
 ->addComponent('Zend_Foo_Component_A')
 ->selectMethod('setComponentA')
  ->addComponent('Zend_Foo_Component_A')
 ->selectMethod('setComponentB')
  ->addComponent('Zend_Foo_Component_B')
 ->selectMethod('setComponentC')
  ->addComponent('Zend_Foo_Component_C')
 ->newInstance();
```

Assembling Objects Using Configuration

The configuration is typically set up in a different file. Each package can have its own configuration file: PHP, INI or XML file. The configuration file holds the components specifications and package dependencies.

You can pass an instance of Zend_Config via the constructor, or set a configuration array using the setConfigArray() method.

The cases below assume that the following classes have been defined:

```
class Zend_Foo {
 public function __construct(
  Zend_Foo_Component_Interface $componentA = null,
  Zend_Foo_Component_Interface $componentB = null,
  $arg3 = null,
  $arg4 = null) {
 }

 public function setComponentA(Zend_Foo_Component_Interface $component, $arg2 = null) {
 }
}

interface Zend_Foo_Component_Interface {
}
class Zend_Foo_Component_A implements Zend_Foo_Component_Interface {
}
class Zend_Foo_Component_B implements Zend_Foo_Component_Interface {
}
```

UC-01

```
$components = array(
 'Foo' => array(
  'class'        => 'Zend_Foo',
  'arguments'    => array(
   '__construct' => 'ComponentA',
  ),
 ),
 'ComponentA' => array(
  'class'        => 'Zend_Foo_Component_A',
  'instanceof'   => 'Zend_Foo_Component_Interface',
 ),
 'ComponentB' => array(
  'class'        => 'Zend_Foo_Component_B',
  'instanceof'   => 'Zend_Foo_Component_Interface',
 ),
);

$config = new Zend_Config($components);
// $config = new Zend_Config_Xml('components.xml', 'staging');

$di = new Zend_Di_Container($config);

// Create an instance of Zend_Foo and injects Zend_Foo_Component_A via the constructor method
$foo = $di->loadClass('Foo')->newInstance();
```

The two major flavors of Dependency Injection are Setter Injection (injection via setter methods) and Constructor Injection (injection via constructor arguments). Zend_Di provides support for both, and even allows you to mix the two when configuring the one object.

Constructor dependency injection

When a class is loaded, the constructor method is selected by default.

UC-02

```
// Inject a dependency
$di->loadClass('Foo')
 ->addComponent('ComponentA')
 ->newInstance();
```

UC-03

```
// Inject multiple dependencies
$di->loadClass('Foo')
 ->addComponent('ComponentA')
 ->newInstance();
```

UC-04

```
// Inject dependencies and pass arguments
$di->loadClass('Foo')
 ->addComponent('ComponentA')
 ->addComponent('ComponentB')
 ->addValue('arg3')
 ->addValue('arg4')
 ->newInstance();

// Or...
$di->loadClass('Foo')
 ->addComponent('ComponentA', 'ComponentB')
 ->addValue('arg3', 'arg4')
 ->newInstance();
```

Users can map out specifications for components and their dependencies. So whenever a class is loaded, Zend_Di will inject the dependencies automatically. For example:

UC-05

```
$config = array(
 'Foo' => array(
  'class'        => 'Zend_Foo',
  'instanceof'   => 'Zend_Foo',
  'arguments'    => array(
   '__construct'   => 'ComponentA, ComponentB, :param',
   'setComponentA' => 'ComponentA'
  ),
 ...

$di = new Zend_Di_Container();
$di->setConfigArray($config);

// Bind parameter and create an instance of the Zend_Foo class
$di->loadClass('Foo')
 ->bindParam(':param', 'Parameter 1')
 ->newInstance();
```

Setter dependency injection

UC-06

```
// Pass dependencies through the setComponentA() method
$di->loadClass('Foo')
 ->selectMethod('setComponentA')
  ->addComponent('ComponentA')
 ->newInstance();
```

Zend_Di injects dependencies using the top-down fashion, starting with the constructor and ending with the setter methods.

UC-07

```
// Constructor and setter dependency injection
$di->loadClass('Foo')
 ->addComponent('ComponentA', 'ComponentB')
 ->addValue('arg3', 'arg4')
 ->selectMethod('setComponentA')
  ->addComponent('ComponentA')
  ->addValue('arg2')
 ->newInstance();
```

Users can map out specifications for a component:

UC-08

```
$config = array(
 'Foo' => array(
  'class'        => 'Zend_Foo',
  'instanceof'   => 'Zend_Foo',
  'arguments'    => array(
   'setComponentA' => 'ComponentA',
  ),
 ...

$di = new Zend_Di_Container();
$di->setConfigArray($config);

$foo = $di->loadClass('Foo')->newInstance();
```

Storage Containers

You can tell Zend_Di what components to manage by adding them to a container (the order of registration has no significance). Containers are stored are retrieved using the Zend_Di_Registry class. The Zend_Di_Registry::getContainer() method returns an instance of Zend_Di_Storage_Interface.

UC-09

```
$di = new Zend_Di_Container();
$di->setConfigArray($config);
$foo = $di->loadClass('Foo')->newInstance();

$registry = $di->getRegistry();
$registry->open('FooPackage');
$registry->add('Foo');
$registry->close();

// Get an instance of the container FooPackage
$fooPackage = $registry->getContainer('FooPackage');

while ($obj = $fooPackage->current()) {
 echo $fooPackage->getClassName();
 $fooPackage->next();
}

// Get a single instance of the Foo class
$foo = $registry->getSingleton('Foo');
```

You can register your own container as long as you pass an instance of Zend_Di_Storage_Interface. New containers can be register using the Zend_Di_Registry::setStorage() method.

UC-10

```
class Zend_Di_Storage_Cache implements Zend_Di_Storage_Interface {
}

$di = new Zend_Di_Container();
$di->getRegistry()->setStorage(new Zend_Di_Storage_Cache());
```

Real-life example

Please visit the following page: http://framework.zend.com/wiki/display/ZFPROP/Zend_Di+Example

9. Class Skeletons

- Zend_Di_Container

```
class Zend_Di_Container
{
 public function __construct(Zend_Config $config = null)
 public function setConfigArray(array $config)
 public function loadClass($componentName)
 public function selectMethod($methodName)
 public function setMethod($methodName = '__construct')
 public function addComponent()
 public function addValue()
 public function bindParam($identifier, $variable, $dataType = null)
 public function newInstance()
 public function getSingleton($componentName)
 public function getContainer($containerName)
 public function buildConstructorArgs()
 public function buildSetterArgs()
 public function getParameter()
 public function setParameter(Zend_Di_Component_Parameter $parameter = null)
 public function getRegistry()
 public function setRegistry(Zend_Di_Registry $registry = null)
 public function getFactory()
 public function setFactory(Zend_Di_Factory $factory = null)
 protected function _getInstanceOf($className)
}
```

- Zend_Di_Factory

```
class Zend_Di_Factory
{
 public function create(Zend_Di_Parameter $parameter, array $componentSpec)
 public function loadClass($className, $componentName = null)
 public function isClassDefined($className)
}
```

- Zend_Di_Parameter

```
class Zend_Di_Parameter
{
 public function setMethodArgs(array $args, $component, $method)
 public function setParameter($data, $methodName, $dataType)
 public function bindParam($identifier, $variable, $dataType = null)
 public function getParamValue($parameter)
 public function setConstructorArgs(array $args)
 public function getConstructorArgs()
 public function hasConstructorArgs()
 public function setSetterArgs(array $args)
 public function getSetterArgs()
 public function hasSetterArgs()
}
```

- Zend_Di_Registry

```
class Zend_Di_Registry
{
 public function __construct(array $identityMap = null)
 public function open($containerName)
 public function getContainer($containerName)
 public function setContainer($containerName)
 public function isContainer($containerName)
 public function add($name)
 public function close()
 public function isSelected()
 public function setStorage(Zend_Di_Storage_Interface $obj = null)
 public function getStorage()
 public function setSingleton($componentName, $obj)
 public function getSingleton($componentName)
 public function isSingleton($componentName)
}
```

- Zend_Di_Storage_Object

```
class Zend_Di_Storage_Object implements Zend_Di_Storage_Interface, Iterator
{
 public function newInstance()
 public function get($className)
 public function set($className, $instance)
 public function isStored($className)
 public function getClassName()
 public function next()
 public function rewind()
 public function current()
 public function key()
 public function valid()}
}
```

]]></ac:plain-text-body></ac:macro>
]]></ac:plain-text-body></ac:macro>