



The PHP Company

# Zend Framework 2 Patterns

Matthew Weier O'Phinney  
Project Lead, Zend Framework

# Roadmap for today

---

- Namespaces and Autoloading
- Exceptions
- Configuration
- Plugin systems
- Dispatching
- Inversion of Control

# Format

---

- List *The Problems*
- Detail the *ZF2 Approach*

But first, some history

# Zend Framework 1.X

---

- 1.0 Released in July 2007
- Largely evolutionary development
  - ▶ Inconsistencies in APIs, particularly surrounding plugins and configuration
  - ▶ Best practices have been discovered over time
  - ▶ Many key features for modern applications have been added in recent versions

# Zend Framework 2.0

---

- First new major release
  - ▶ Allowing us to break backwards compatibility
- Focus is on:
  - ▶ Consistency
  - ▶ Performance
  - ▶ Documentation
  - ▶ User productivity

# Namespaces and Autoloading

# The Problems

---

- Lengthy class names
  - ▶ Difficult to refactor
  - ▶ Difficult to retain semantics with shorter names
- Performance issues
  - ▶ Many classes are used JIT, and shouldn't be loaded until needed
- Missing `require_once` statements lead to errors

# ZF2 Approach: Namespaces

---

- Formalize the prefixes used in ZF1
  - ▶ Namespace separator correlates to directory separator
- Help identify dependencies (imports)
  - ▶ Allows refactoring using different implementations easier
  - ▶ Makes packaging easier

# Namespaces

---

```
namespace Zend\EventManager;  
  
use Zend\Stdlib\CallbackHandler;  
  
class EventManager implements EventCollection  
{  
    /* ... */  
}
```

# Namespaces

---

- Interfaces as namespaces
  - ▶ Interface names are adjectives or nouns
  - ▶ Concrete implementations in a sub-namespace named after the interface
  - ▶ Contract-Oriented paradigm

# Interfaces as Namespaces

```
Zend/Session
|-- Storage.php
`-- Storage
    |-- ArrayStorage.php
    `-- SessionStorage.php
```

```
namespace Zend\Session;
interface Storage
{
    /* ... */
}
```

```
namespace Zend\Session\Storage;
use ArrayObject,
    Zend\Session\Storage,
    Zend\Session\Exception;
class ArrayStorage
    extends ArrayObject
    implements Storage
{ /* ... */ }
```

# ZF2 Approach: Autoloading

---

- **No more `require_once` calls!**
- Multiple approaches
  - ▶ ZF1-style `include_path` autoloader
  - ▶ Per-namespace/prefix autoloading
  - ▶ Class-map autoloading

# ZF1-Style Autoloading

---

```
require_once 'Zend/Loader/StandardAutoloader.php';  
$loader =  
    new Zend\Loader\StandardAutoloader(array(  
        'fallback_autoloader' => true,  
    ));  
$loader->register();
```

# ZF2 NS/Prefix Autoloading

---

```
require_once 'Zend/Loader/StandardAutoloader.php';
$loader = new Zend\Loader\StandardAutoloader();
$loader->registerNamespace(
    'My', __DIR__ . '/../library/My')
->registerPrefix(
    'Phly_', __DIR__ . '/../library/Phly');
$loader->register();
```

# ZF2 Class-Map Autoloading

---

```
return array(  
    'My\Foo\Bar' => __DIR__ . '/Foo/Bar.php',  
);
```

```
require_once 'Zend/Loader/ClassMapAutoloader.php';  
$loader = new Zend\Loader\ClassMapAutoloader();  
$loader->registerAutoloadMap(  
    __DIR__ . '/../library/.classmap.php');  
$loader->register();
```

# Exceptions

# The Problems

---

- All exceptions derived from a common class
- No ability to extend more semantic exception types offered in SPL

# ZF2 Approach

---

- Eliminated Zend\_Exception
- Each component defines a marker Exception interface
- Additional exception types are created in an Exception subnamespace
  - ▶ These extend SPL exceptions, and implement the component-level exception interface

# What the solution provides

---

- Catch specific exception types
- Catch SPL exception types
- Catch any component-level exception
- Catch based on global exception type

# Exceptions in use

```
Zend/EventManager  
|-- Exception.php  
`-- Exception  
    |-- InvalidArgument-  
        Exception.php
```

```
namespace Zend\EventManager;  
  
interface Exception {}
```

```
namespace Zend\EventManager\Exception;  
  
use Zend\EventManager\Exception;  
  
class InvalidArgumentException  
    extends \InvalidArgumentException  
    implements Exception  
{}
```

# Exceptions in use

---

```
namespace Zend\EventManager\Exception;
use Zend\EventManager\Exception;
try {
    $events->trigger('foo.bar', $object);
} catch (InvalidArgumentException $e) {
} catch (Exception $e) {
} catch (\InvalidArgumentException $e) {
} catch (\Exception $e) {
}
```

# Configuration

# The Problems

---

- Case-SENSitivity
- Varying APIs
  - ▶ `setOptions()`
  - ▶ `setConfig()`
  - ▶ `__construct()`
  - ▶ explicit setters

# ZF2 Approach

---

- Option names will be lowercase\_underscore\_separated\_words
- Standard solution across components
  - ▶ `setOptions()` style, proxying to setters, or
  - ▶ per-component configuration objects

# setOptions() style

```
class Foo
{
    public function setOptions($options)
    {
        if (!is_array($options)
            && !($options instanceof \Traversable)
        ) {
            throw new \InvalidArgumentException();
        }

        foreach ($options as $key => $value) {
            $method = normalize($key);
            if (method_exists($this, $method)) {
                $this->$method($value);
            }
        }
    }
}
```

# Options object style

```
class FooOptions extends Options
{
    public $bar;
    public $baz;

    public function __construct($options = null)
    {
        if (!is_array($options)
            && !($options instanceof \Traversable)
        ) {
            throw new \InvalidArgumentException();
        }

        foreach ($options as $key => $value) {
            $prop = normalize($key);
            $this->$prop = $value;
        }
    }
}
```

# Options object style

---

```
class Foo
{
    public function __construct(Options $options = null)
    {
        if (null !== $options) {
            foreach ($options as $key => $value) {
                $this->$key = $value;
            }
        }
    }
}
```

# Plugin Architectures

# Terminology

---

- For our purposes, a “plugin” is any class that is determined at runtime.
  - ▶ Action and view helpers
  - ▶ Adapters
  - ▶ Filters and validators

# The Problems

---

- Varying approaches to dynamically discovering plugin classes
  - ▶ Prefix-path stacks (*most common*)
  - ▶ Paths relative to the calling class
  - ▶ Setters to indicate classes
- Most common approach is terrible
  - ▶ Bad performance
  - ▶ Hard to debug
  - ▶ No caching of discovered plugins

# ZF2 Approach: Plugin Broker

---

- Separate Plugin Location interface
  - ▶ Allows varying implementation of plugin lookup
- Separate Plugin Broker interface
  - ▶ Composes a Plugin Locator

# ZF2 Approach: Plugin Broker

---

- Standard implementation across components
  - ▶ Subclassing standard implementation allows type-hinting, caching discovered plugins, etc.
  - ▶ while allowing you to substitute your own implementations
- Class-map location by default
- 2-5x performance gains!
- Easier to debug

# Plugin class location

```
namespace Zend\Loader;

interface ShortNameLocater
{
    public function isLoaded($name);
    public function getClassName($name);
    public function load($name);
}
```

```
namespace Zend\View;
use Zend\Loader\PluginClassLoader;
class HelperLoader extends PluginClassLoader
{
    protected $plugins = array(
        'action' => 'Zend\View\Helper\Action',
        'baseurl' => 'Zend\View\Helper\BaseUrl',
        /* ... */
    );
}
```

# Plugin broker

---

```
namespace Zend\Loader;

interface Broker
{
    public function load($plugin, array $options = null);
    public function getPlugins();
    public function isLoaded($name);
    public function register($name, $plugin);
    public function unregister($name);
    public function setClassLoader(
        ShortNameLocator $loader);
    public function getClassLoader();
}
```

# Plugin broker

```
class HelperBroker extends PluginBroker {
    protected $defaultClassLoader = 'Zend\View\HelperLoader';
    protected $view;
    public function setView(Renderer $view) {}
    public function getView() {}
    public function load($plugin, array $options = null) {
        $helper = parent::load($plugin, $options);
        if (null !== ($view = $this->getView())) {
            $helper->setView($view);
        }
        return $helper;
    }
    protected function validatePlugin($plugin)
    {
        if (!$plugin instanceof Helper) {
            throw new InvalidHelperException();
        }
        return true;
    }
}
```

# ZF2 Approach: Events

---

- Trigger events at interesting points in your application
  - ▶ Use as basic subject/observer pattern
  - ▶ Or as intercepting filters
  - ▶ Or a full-fledged Aspect-Oriented Programming system

# ZF2 Approach: Events

---

- Compose an EventManager to a class
- Attach handlers to events
  - ▶ Handlers receive an Event
    - event name
    - target (calling) object
    - parameters passed
  - ▶ Handlers can also be attached statically

# Triggering an event

---

```
public function doSomething(  
    $with, $params = array()  
) {  
    $this->events()->trigger(  
        __FUNCTION__, compact('with', 'params')  
    );  
    /* ... */  
}
```

# Listening to an event

```
use Zend\EventManager\EventManager as Events;

$events = new Events();
$events->attach('doSomething', function($e) use ($log) {
    $event = $e->getName();
    $target = get_class($e->getTarget());
    $params = json_encode($e->getParams());
    $message = sprintf('%s (%s): %s', $event, $target,
$params);
    $log->info($message);
});
```

# Attaching statically to an event

---

```
use Zend\EventManager\StaticEventManager as AllEvents;  
  
$events = AllEvents::getInstance();  
  
// Specify the class composing an EventManager  
// as first arg  
$events->attach('Foo', 'doSomething', function($e) {});
```

# Dispatchers

# The Problems

---

- Not terribly performant
- Hard to customize
- Hard to inject controllers with dependencies
- Forces pre-initialization of resources if you want them configured by `Zend_Application`

# ZF2 Approach

---

- Discrete Request, Response, and Dispatchable interfaces
  - ▶ Request encompasses request environment
  - ▶ Response aggregates response returned
  - ▶ Dispatchable objects formalize a Strategy pattern

# ZF2 Approach

---

- Anything Dispatchable can be attached to the MVC
  - ▶ Server components (XML-RPC, JSON-RPC, etc.)
- Allows building your own MVC approach
  - ▶ Do you want action methods to receive explicit arguments?
  - ▶ Do you want to select a different action method based on request headers?

# MVC Interfaces

```
interface Message
{
    public function setMetadata($spec, $value = null);
    public function getMetadata($key = null);
    public function setContent($content);
    public function getContent();
}
```

```
interface Request
    extends Message
{
    public function
        __toString();
    public function
        fromString($string);
}
```

```
interface Response
    extends Message
{
    public function
        __toString();
    public function
        fromString($string);
    public function send();
}
```

# MVC Interfaces

---

```
interface Dispatchable
{
    public function dispatch(
        Request $request,
        Response $response = null);
}
```

# Inversion of Control

# The Problems

---

- How do objects get dependencies?
  - ▶ In particular, how do Controllers get dependencies?

# ZF2 Approach

---

- Service Locator
  - ▶ Basic pattern:
    - `set($name, $service)`
    - `get($name)`
  - ▶ Formalization of application services (mailer, logger, profiler, etc.)
  - ▶ Good interface for typehinting

# Service Locator

```
use Zend\Di\ServiceLocator,  
    Zend\EventManager\EventManager;  
  
class MyLocator extends ServiceLocator  
{  
    protected $events;  
    protected $map = array('events' => 'getEvents');  
  
    public function getEvents()  
    {  
        if (null !== $this->events) {  
            return $this->events;  
        }  
        $this->events = new EventManager();  
        return $this->events;  
    }  
}
```

# ZF2 Approach

---

- Dependency Injection Container
  - ▶ Scaffolding for constructor and setter injection
  - ▶ Use programmatically, or from configuration
  - ▶ Typically used to seed a service locator

# Dependency Injection

```
$db = new Definition( 'My\Db\Adapter\Sqlite' );
$db->setParam( 'name', __DIR__ . '/../data/db/users.db' );

$mapper = new Definition( 'My\Mapper\Db' );
$mapper->addMethodCall(
    'setAdapter', array( new Reference( 'db' ) ) );

$service = new Definition( 'My\Resource\Users' );
$service->setParam( 'mapper', new Reference( 'mapper' ) );

$di = new DependencyInjector;
$di->setDefinitions( array(
    'db'      => $db,
    'mapper'  => $mapper,
    'users'   => $service,
) );

$users = $di->get( 'users' ); // My\Resource\Users
```

# Controllers as services

---

- Solves issue of controller dependencies
- Each request only instantiates what's needed for that request
- Better testability of controllers

# Controllers as services

---

```
$UserController = new Definition('Site\Controller\User');
$UserController->setParam(
    'service', new Reference('users'));
$di->setDefinition($UserController, 'controller-user');

// Inside dispatcher:
$controller = $di->get($controllerName);
$result = $controller->dispatch($request, $response);
```

More to come!

# Zend Framework 2.0

---

- Schedule:
  - ▶ MVC milestone by end of May
  - ▶ Preview Release following MVC milestone
  - ▶ Beta release during summer
  - ▶ Stable by end-of-year

# Resources

- 
- ZF2 Wiki: <http://bit.ly/zf2wiki>
  - ZF2 Git information: <http://bit.ly/zf2gitguide>
  - ZF2 MVC sandbox:  
<git://git.mwop.net/zf2sandbox.git>
  - ZF2 DI prototype: <http://bit.ly/gBBnDS>

# Thank you!

- <http://framework.zend.com/>
- <http://twitter.com/weierophinney>