twitter: @akrabat

Zend Framework 2

Rob Allen

PHPNW October 2011

The experiment

Agenda

- The state of ZF2
- ZF2 foundations
- A look at a ZF2 MVC application

The state of ZF2

“The primary thrust of ZF 2.0 is to make a more consistent, well-documented product, improving developer productivity and runtime performance.”

Matthew Weier O’Phinney

- PHP 5.3 & namespaces
- Interfaces over abstract classes
- Faster autoloader
- Consistent plugin loading
- DI & service locator
- Event manager
- New MVC system

Pace of development

When?

I *still* have no idea!

ZF2 foundations

Namespaces

Namespaces are a way of encapsulating items

PHP Manual

Namespaces allow us to

- combine libraries with the same class names
- avoid very long class names
- organise our code easily

(They also affect functions and constants)

Defining a namespace

```php
namespace My\Db\Statement;

class Sqlsrv
{
}
```

Namespace constant:

```php
namespace My\Db\Statement;

echo __NAMESPACE__;
```

Namespaces

Within namespace:

```
namespace My\Db\Statement;

function testSqlsrv() {
   $stmt = new Sqlsrv ();
}
```

Fully qualified:

```
$stmt = new \My\Db\Statement\Sqlsrv();
```

Import

Specific class:

```
use My\Db\Statement\Sqlsrv;
$stmt = new Statement\Sqlsrv();
```

Multiple namespaces:

```
use My\Db\Statement;
use My\Db\Adapter;

$stmt = new Statement\Sqlsrv();
$adapter = new Adapter\Sqlsrv();
```

Import

You can’t do this!

```
use Zend\Db\Statement\Sqlsrv;
use Zend\Db\Adapter\Sqlsrv;

$stmt = new Sqlsrv ();
$adapter = new Sqlsrv ();
```

Aliases

Solve with aliases:

```
use My\Db\Statement\Sqlsrv as DbStatement;
use My\Db\Adapter\Sqlsrv as DbAdapter;

$stmt = new DbStatement ();
$adapter = new DbAdapter();
```

Namespace resolution

An unqualified function name is resolved in this order:

- The current namespace is prepended to the function name.
- If the function names doesn't exists in the current namespace, then a global function name is used if it exists.

```
$date = date('Y-m-d');
```

Namespace resolution

An unqualified class name is resolved in this order:

- If there is an import statement that aliases another name to this class name, then the alias is applied.
- Otherwise the current namespace is applied.

```
$dateTime = new \DateTime();
```

Autoloader

Autoloading allows files to be automatically included at the last minute.

zendcoding.com

Zend\Loader\Autoloader

- Class map autoloading, including class map generation
- PSR-0-compliant per-prefix or namespace autoloading
- Fallback PSR-0-compliant include_path autoloading
- Autoloader factory for loading several autoloader strategies at once

PSR-0?

A standard that describes how to name a class so that the autoloader can find it.

Essentially each namespace maps directly to a folder on disk:

i.e.

`\Zend\Config\Ini`

should be found in

`/path/to/Zend/Config/ini.php`

StandardAutoloader

- Inspects classname and finds file on disk
- Load via namespace => directory pairs
- Load via prefix => directory pairs
- Fallback search of include_path

StandardAutoloader

```php
require_once ZF2_PATH.'/Loader/StandardAutoloader.php';

$loader = new Zend\Loader\StandardAutoloader(array(
    'prefixes' => array(
        'MyVendor' => __DIR__ . '/MyVendor',
    ),
    'namespaces' => array(
        'MyNamespace' => __DIR__ . '/MyNamespace',
    ),
    'fallback_autoloader' => true,
));

// register our loader with the SPL autoloader
$loader->register();
```

StandardAutoloader

```php
require_once ZF2_PATH.'/Loader/StandardAutoloader.php';

$loader = new Zend\Loader\StandardAutoloader();

$loader->registerPrefix('MyVendor', __DIR__.'/MyVendor')
       ->registerNamespace('MyNamespace',
           __DIR__.'/MyNamespace')
       ->setFallbackAutoloader(true);

// register our loader with the SPL autoloader
$loader->register();
```

ClassMapAutoloader

- Array of class name to file on disk
- High performance!

```
public function autoload($class)
{
    if (isset($this->map[$class])) {
        include $this->map[$class];
    }
}
```

A class map

autoload_classmap.php:

```
use My\Db\Statement\Sqlsrv;
$stmt = new Statement\Sqlsrv();
```

Using:

```
require_once ZF2_PATH.'/Loader/ClassMapAutoloader.php';

$autoLoader = new \Zend\Loader\ClassMapAutoloader(
    array(__DIR__ . '/autoload_classmap.php'));

// register with the SPL autoloader
$autoLoader->register();
```

Multiple class maps

```
$loader = new \Zend\Loader\ClassMapAutoloader(array(
    __DIR__ . '/../library/autoload_classmap.php',
    __DIR__ . '/../application/autoload_classmap.php',
));
```

OR:

```
$loader = new \Zend\Loader\ClassMapAutoloader();
$loader->registerAutoloadMap(array(
    __DIR__ . '/../library/autoload_classmap.php',
    __DIR__ . '/../application/autoload_classmap.php',
));
```

Creating classmaps

```
prompt> php path/to/zf2/bin/classmap_generator.php -w
Creating class file map for library in '/var/www/
project/library'...
Wrote classmap file to '/var/www/project/library/
autoload_classmap.php'
```

Combining autoloaders

- Use class maps and the prefixes!
- Very useful in development
- Use ZF2's `AutoloaderFactory` class

```php
require_once ZF2_PATH . '/Loader/AutoloaderFactory.php';

Zend\Loader\AutoloaderFactory::factory(array(
  'Zend\Loader\ClassMapAutoloader' => array(
    __DIR__ . '/../library/Zend/autoload_classmap.php',
  ),
  'Zend\Loader\StandardAutoloader' => array(
    'prefixes' => array(
      'MyVendor' => __DIR__ . '/MyVendor',
    ),
    'namespaces' => array(
      'MyNamespace' => __DIR__ . '/MyNamespace',
    ),
    'fallback_autoloader' => true,
  ),
));
```

Autoloader summary

- Use the class map one in preference as it's *really really* fast!
- Prefix => directory pairs is surprisingly fast though compared to ZF1

Exercise

Update the ZF1 Tutorial to use the Autoloader from ZF2:

1. Using StandardAutoloader
2. Using a classmap for library code and a StandardAutoloader for the application code.

Dependency Injection

Dependency injection means giving an object its instance variables. Really. That's it.

James Shore

Contrived example

```
class Album
{
    protected $artist;

    public function getArtistName()
    {
        return $artist->getName();
    }
}
```

How do we set the `$artist` member variable?

Direct instantiation

```
class Album
{
    protected $artist;

    public function __construct()
    {
        $this->artist = new Artist();
    }

    // etc
}
```

Constructor injection

```
class Album
{
    protected $artist;

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

    // etc
}
```

Calling code:

```
$album = new Album($artist);
```

Constructor injection

```
class Album
{
    protected $artist;

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

    // etc
}
```

Calling code:

```
$album = new Album($artist);
```

Setter injection

```
class Album
{
    protected $artist;

    public function setArtist(Artist $artist)
    {
        $this->artist = $artist;
    }

    // etc
}
```

Calling code:

```
$album = new Album();
$album->setArtist($artist);
```

Advantages

- Can use different objects (e.g. SoloArtist)
- Configuration is natural
- Testing is simplified
- Do not need to change the Album class

Disadvantages

Instantiation and configuration of a class' dependencies are the responsibility of the calling code.

Dependency Injection Container

```
class AlbumContainer
{
    public static $artist;

    public static function createAlbum()
    {
        $album = new Album();
        $album->setArtist(self::$artist);
        // more setXxx() calls here as required

        return $album;
    }
}
```

Calling code:

```
$album = AlbumContainer::createAlbum();
```

A dependency injection container is a component that holds dependency definitions and instantiates them for you.

Zend\Di

- Supports constructor and setter injection
- Configured in code or via config file
- Type hinting makes life easier

Constructor injection

```php
namespace My;

class Artist
{
}

class Album
{
    protected $artist = null;

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

To use an $album:

```
$di = new Zend\Di\DependencyInjector();
$album = $di->get('My\Album');
```

Also:

```
$album2 = $di->newInstance('My\Album');
```

Parameters in dependencies

```
class Artist
{
    protected $name;
    public function __construct($name)
    {
        $this->name = $name;
    }
}
```

Use `setParameters`:

```
$di = new Zend\Di\DependencyInjector();
$di->getInstanceManager()
   ->setParameters('Artist', array('name' => 'Queen'));
```

Parameters in dependencies

Usage:

```
$album = $di->get('My\Album');
```

Set at call time:

```
$album = $di->get('My\Album',
    array('name'=>'Jonathan Coulton'));
```

Setter injection

```php
namespace My;
class Artist
{
    protected $name;
    public function __construct($name)
    {
        $this->name = $name;
    }
}

class Album
{
    protected $artist = null;
    public function setArtist(Artist $artist)
    {
        $this->artist = $artist;
    }
}
```

To use an $album:

Enable:

```
$di->getDefinition()->getIntrospectionRuleset()
    ->addSetterRule('paramCanBeOptional', false);
```

Use:

```
$album = $di->get('My\Album', array('name'=>'Train'));
```

Definitions

- RuntimeDefinition
  - Resolves at runtime (reflection-based)
- BuilderDefinition
  - Programmatic creation via `Builder\PhpClass`
- ArrayDefinition
  - Dependencies defined in an array
- AggregateDefinition
  - Combine different definitions

BuilderDefinition

```
use Zend\Di\DependencyInjector, Zend\Di\Definition,
    Zend\Di\Definition\Builder;

// Builder definition for My\Album
$method = new Builder\InjectionMethod();
$method->setName('setArtist');
$method->addParameter('artist', 'My\Artist');

$class = new Builder\PhpClass();
$class->setName('My\Album');
$class->addInjectionMethod($method);

$builderDef = new Definition\BuilderDefinition();
$builderDef->addClass($class);

$di = new DependencyInjector();
$di->setDefinition($builderDef);
```

Compilation to ArrayDefinition

```
use Zend\Di,
   Zend\Code\Scanner\ScannerDirectory;

$compiler = new Di\Definition\Compiler();
$compiler->addCodeScannerDirectory(
    new ScannerDirectory('path/to/library/My/')
);
$definition = $compiler->compile();

$di = new Di\DependencyInjector();
$di->setDefinition($definition);
$album = $di->get('My\Album', array('Oasis'));
```

Persistance

```
file_put_contents(
   __DIR__ . '/di-definition.php',
   '<?php return ' .
     var_export($definition->toArray(), true) . ';'
);
```

Load:

```
$definition = new Zend\Di\Definition\ArrayDefinition(
    include __DIR__ . '/di-definition.php'
);
```

Aliases

- Makes it easier to specify dependencies
- Can retrieve from the injector by alias too.

```
$di = new Zend\Di\DependencyInjector();
$im = $di->getInstanceManager();

$im->addAlias('artist', 'My\Artist');
$im->addAlias('album', 'My\Album');

$im->setParameters('artist', array('name' => 'Blur'));

$artist = $di->get("artist");
$album = $di->get("album", array("name" => "Queen"));
```

Configuration

Config file:

```
[production]
di.instance.alias.album = 'My\Album'
di.instance.alias.artist = 'My\Artist'

di.instance.artist.parameters.name = 'Marillion'
```

PHP file:

```php
use Zend\Config, Zend\Di;
$config = new Config\Ini('application.ini', 'dev');
$diConfig = new Di\Configuration($config->di);

$di = new Di\DependencyInjector($diConfig);
$artist = $di->get("artist");
$album = $di->get("album", array("name" => "Queen"));
```

Aggregation

```php
use Zend\Di\Definition;

$arrayDef = new Definition\ArrayDefinition(
    include __DIR__ . '/di-definition.php'
);

$runtimeDef = new Definition\RuntimeDefinition();

// Aggregate the two definitions
$aggregateDef = new Definition\AggregateDefinition();
$aggregateDef->addDefinition($arrayDef);
$aggregateDef->addDefinition($runtimeDef);

$di = new \Zend\Di\DependencyInjector();
$di->setDefinition($aggregateDef);
```

DI summary

- DI allows defining dependencies independent of the calling code
- You still need to wire things up - it's just all in one place with DIC.
- This is used extensively in the MVC system

Event Manager

Event Manager allows a class to publish events which other components can listen for and act when that event occurs.

Me!

Terminology

- An **Event Manager** is an object that aggregates listeners for one or more named events, and which triggers events.
- A **Listener** is a callback that can react to an event.
- An **Event** is an action.
- A **Target** is an object that creates events

(Shameless stolen from Matthew Weier O'Phinney!)

Simple example

```
use Zend\EventManager\EventManager,
    Zend\EventManager\Event;

$callback = function($event) {
    echo "An event has happened!\n";
    var_dump($event->getName());
    var_dump($event->getParams());
};

$eventManager = new EventManager();
$eventManager->attach('eventName', $callback);

echo "\nRaise an event\n";
$eventManager->trigger('eventName', null,
    array('one'=>1, 'two'=>2));
```

Target

```
use Zend\EventManager\EventManager,
    Zend\EventManager\Event;
class MyTarget
{
  public $eventManager;
  public function __construct()
  {
    $this->eventManager = new EventManager();
  }

  public function doIt()
  {
    $event = new Event();
    $event->setTarget($this);
    $event->setParam('one', 1);
    $event->setParam('two', 2);
    $this->eventManager->trigger('doIt.pre', $event);
  }
}
```

Target

```
$callback = function ($event) {
    echo "Responding to doIt.pre!\n";
    var_dump(get_class($event->getTarget()));
    var_dump($event->getName());
    var_dump($event->getParams());
};
```

```
$target = new MyTarget();
$target->eventManager->attach('doIt.pre', $callback);
$target->doIt();
```

StaticEventManager

```php
class MyTarget
{
  public $eventManager;
  public function __construct()
  {
    $this->eventManager = new EventManager(__CLASS__);
  }
  // etc
}

// attach a listener
use Zend\EventManager\StaticEventManager;
$events = StaticEventManager::getInstance();
$events->attach('MyTarget', 'doIt.pre', $callback);

// cause event to trigger
$target->doIt();
```

StaticEventManager

- Statically attached listeners are called after the directly attached ones.
- The identifier can be an array:

```
$this->eventManager = new EventManager(
    array(__CLASS__, get_called_class()));
```

  Allows for attaching to generic parent class or specific child

- Disable statically attached listeners:

```
$this->eventManager->setStaticConnections(null);
```

- Re-enable:

```
$this->eventManager->setStaticConnections(
    StaticEventManager::getInstance());
```

Listener aggregates

- `HandlerAggregate` interface provides for attaching and detaching event listeners
- Two methods:
  - `attach(EventCollection $events)`
  - `detach(EventCollection $events)`
- (`EventCollection` is the interface that `EventManager` implements)

Using HandlerAggregate

```
class MyListener implements HandlerAggregate
{
    protected $eventHandlers = array();
    public function attach(EventCollection $events)
    {
        $this->handlers[] = $events->attach('event1',
            array($this, 'doSomething'));
        $this->handlers[] = $events->attach('event2',
            array($this, 'doSomethingElse'));
    }
    public function detach(EventCollection $events)
    {
        foreach ($this->handlers as $key => $handler) {
            $events->detach($handler);
            unset($this->handlers[$key]);
        }
        $this->handlers = array();
    }
```

Using HandlerAggregate

```php
$target = new MyTarget();

// Attach the target's event manager to the listener
$listener = new MyListener();
$listener->attach($target->getEventManager());

// Now we can trigger some events
$target->doSomething();
$target->doSomethingElse();

// Detach
$listener->detach($target->getEventManager());

// this trigger will not fire the listener's
$target->doSomething();
```

Returned values from listeners

```php
public function triggerEvent1()
{
  $results = $this->eventManager->trigger('event1', $this);
  foreach ($results as $result) {
    var_dump($result);
  }
}
```

- `$results` is a `ResponseCollection`
- `$results` are in reverse order (latest triggered event is first)

Short-circuiting

```php
public function triggerEvent1()
{
  $results = $this->eventManager->trigger('event1', $this);
  foreach ($results as $result) {
    var_dump($result);
  }
}
```

- `$results` is a `ResponseCollection`
- `$results` are in reverse order (latest triggered event is first)

Short-circuiting

```
public function doIt()
{
    $params = array('id' => 1);
    $results = $this->eventManager->trigger('doIt.pre',
        $this, $params,
        function ($result) {
            if($result == 'done') {
                return true;
            }
            return false;
        }
    );
    if ($results->stopped()) {
        // We ended early
    }
}
```

Priority

- Control the order of execution of listeners
- `$priority` is last parameter to `attach()`

```
$eventManager->attach('doIt.pre', $callback, $priority);
```

  - Default is 1
    - Larger increases priority
    - Smaller decreases priority

EventManager summary

- Decouples objects from one another
- Increases flexibility
- Expect to use in MVC system

Exercise

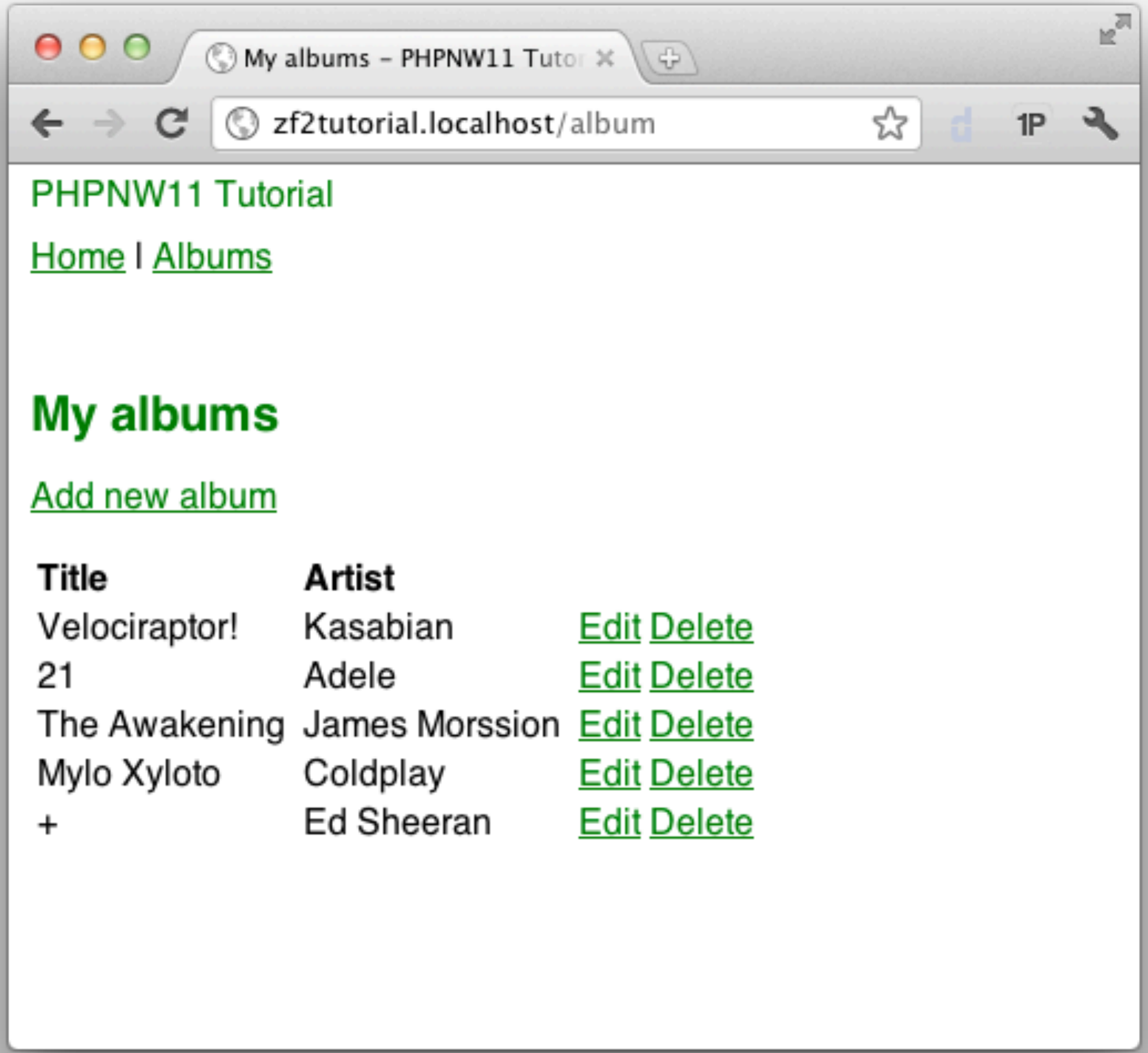
Using the ZF2 EventManager, add logging to `Application_Models_DbTable_Albums`

(I have created `Application_Model_Logger` as a starting point for you...)

A ZF2 Application

This is very much

PROTOTYPE code


My albums - PHPNW11 Tuto
zf2tutorial.localhost/album
PHPNW11 Tutorial
Home | Albums
My albums
Add new album
Title
Artist
Velociraptor!
Kasabian
Edit Delete
21
Adele
Edit Delete
The Awakening
James Morssion
Edit Delete
Mylo Xyloto
Coldplay
Edit Delete
+
Ed Sheeran
Edit Delete

Exercise

Use the ZF2 Skeleton Application and Skeleton Module to create a website that displays a random quote on the home page.

Resources

- http://framework.zend.com/zf2
- https://github.com/akrabat/zf2-tutorial
- https://github.com/zendframework/ZendSkeletonApplication
- https://github.com/zendframework/ZendSkeletonModule
- https://github.com/weierophinney/zf-quickstart/tree/features/zf2-mvc-module

Thank you

feedback: http://joind.in/3459
email: rob@akrabat.com
twitter: @akrabat