Zend Framework 2 Foundations

By Matthew Setter

Zend Framework 2 Foundations

Learn the Foundations of Zend Framework 2

Matthew Setter

This book is for sale at http://leanpub.com/zendframework2-for-beginners

This version was published on 2015-01-13

Tweet This Book!

Please help Matthew Setter by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Zend Framework 2 For Beginners and thought you'd like it too.

The suggested hashtag for this book is #zf2forbeginners.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#zf2forbeginners

This book is dedicated to some special people in my life, without who's support, this book never would have seen the light of day. These are: my beautiful wife Melanie and my parents Jane and Barry. Thank you so much for your support of my ideas and dreams.

Contents

Getting Started

How is it Structured?

The guide will teach you, from a hands-on perspective, the fundamentals of creating an application using Zend Framework 2. When I say fundamentals, I mean just that; **the fundamental knowledge and sections of the framework which you should know to use it in any meaningful way**.

There's quite a lot to take in with ZF2 and for a beginner it can be quite overwhelming. A lot of the time when learning, knowledge goes in one ear and out the other, because there's nothing to relate it to.

So in this book, I take the approach of a freelance developer, tasked with the requirement to build an application which will allow a parent to *record*, *search* and *display* records of when their child was fed. With this information a parent can keep track of when the child was fed and how much they ate.

The Proper Tools

But before we get into the book proper, I want to make sure that you have everything you need to follow along. So in this chapter, we cover the following key areas:

- Required Software
- Additional Tools
- Composer Support

You don't need to use all of the software covered here. But these tools have helped me out immensely. So I recommend them to you, in the hopes they'll help you out as well.

The Required Software

To follow this book, you don't need a lot, except for the following requirements:

- PHP 5.4 / PHP 5.5
- SQLite 3[1]

[1]http://www.sqlite.org/download.html

- Composer[2]
- PHPUnit[3]
- OPCache[4] or APC[5]

Version Control

The book's code repository[6] is stored on Github. If you've not used Git[7] before, or aren't too familiar with it, please don't be concerned.

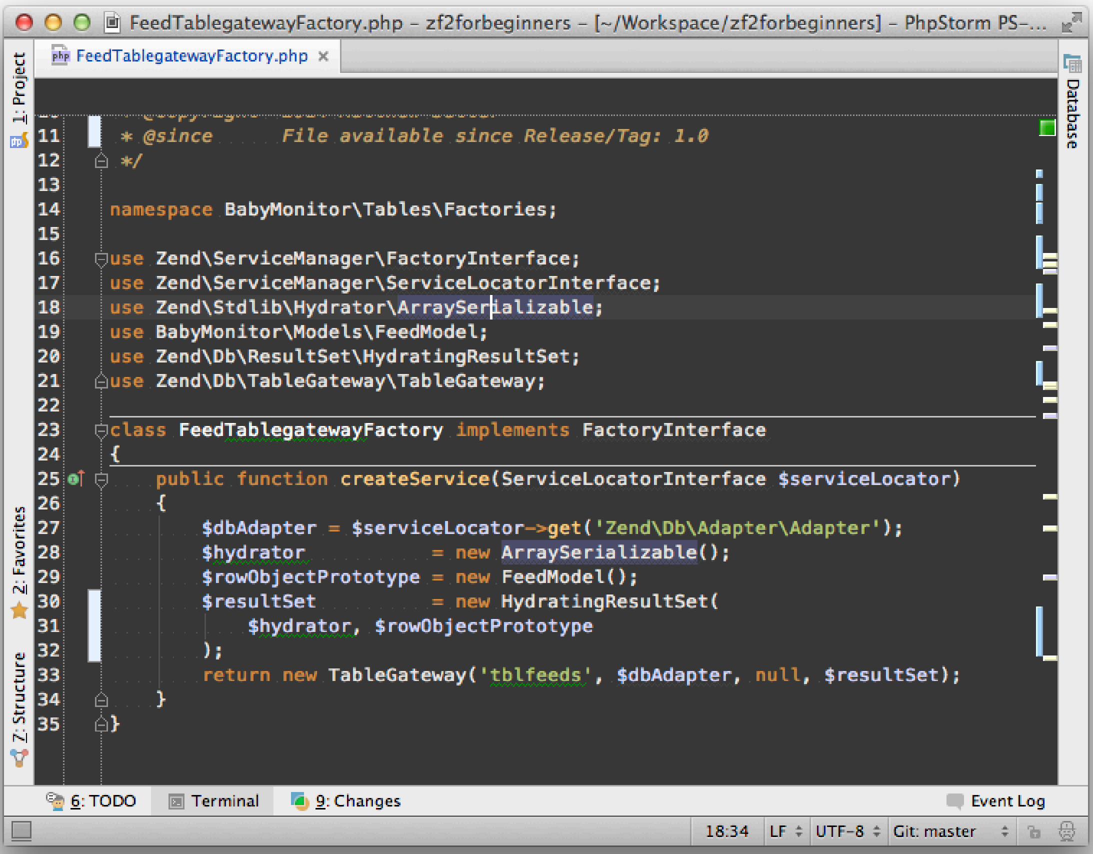
The only commands you'll need to use are `git clone` to get a copy of the repository. However *even this* isn't a must, as you can also get a copy via direct download[8].

Additional Tools

As well as the required software, I use a variety of tools which help me in my daily work[9]. If you're looking for something new or different to help you out, I hope these can be of assistance.

[2]https://getcomposer.org
[3]http://phpunit.de
[4]http://php.net/manual/en/book.opcache.php
[5]http://php.net/manual/en/book.apc.php
[6]https://github.com/settermjd/zf2forbeginners
[7]http://git-scm.com
[8]https://github.com/settermjd/zf2forbeginners
[9]http://www.matthewsetter.com/services/

PHPStorm


FeedTablegatewayFactory.php – zf2forbeginners – [~/Workspace/zf2forbeginners] – PhpStorm PS-...
FeedTablegatewayFactory.php
1: Project
2: Favorites
Z: Structure
Database
6: TODO
Terminal
9: Changes
Event Log
18:34
LF
UTF-8
Git: master

```php
 * @since      File available since Release/Tag: 1.0
 */

namespace BabyMonitor\Tables\Factories;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;
use Zend\Stdlib\Hydrator\ArraySerializable;
use BabyMonitor\Models\FeedModel;
use Zend\Db\ResultSet\HydratingResultSet;
use Zend\Db\TableGateway\TableGateway;

class FeedTablegatewayFactory implements FactoryInterface
{
    public function createService(ServiceLocatorInterface $serviceLocator)
    {
        $dbAdapter = $serviceLocator->get('Zend\Db\Adapter\Adapter');
        $hydrator           = new ArraySerializable();
        $rowObjectPrototype = new FeedModel();
        $resultSet          = new HydratingResultSet(
            $hydrator, $rowObjectPrototype
        );
        return new TableGateway('tblfeeds', $dbAdapter, null, $resultSet);
    }
}
```

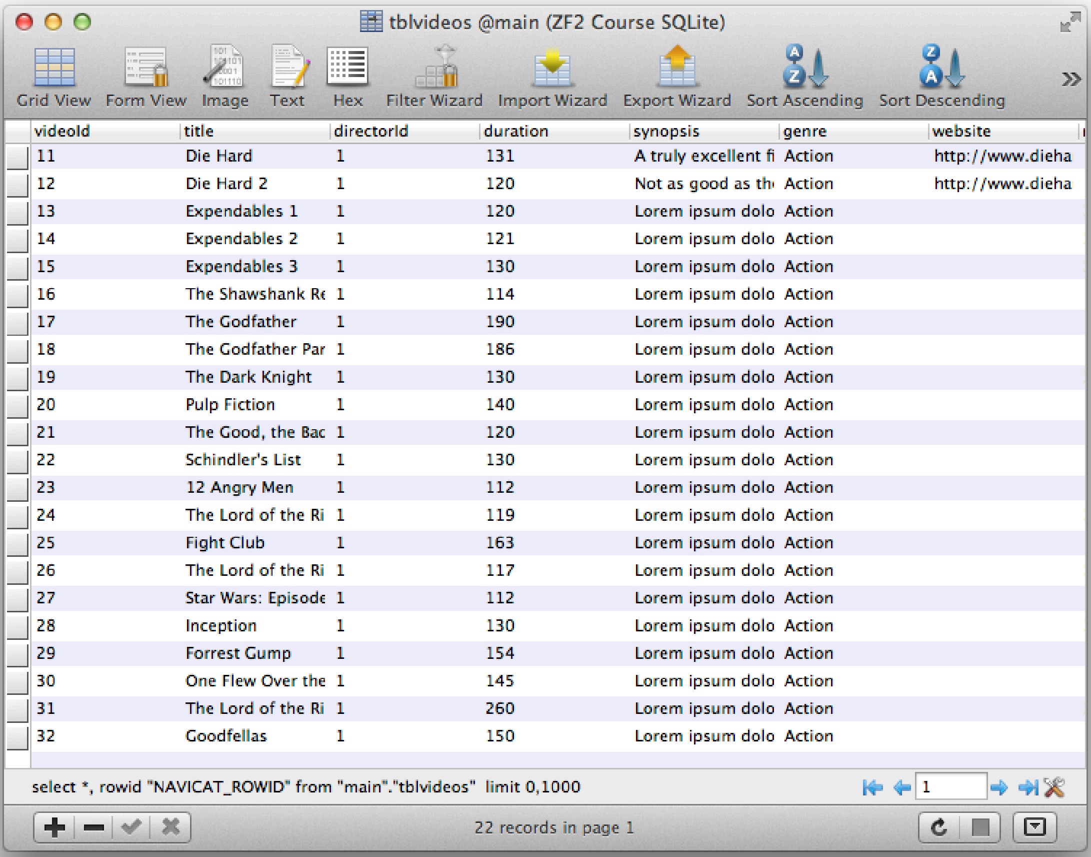
The PHPStorm IDE

If ever there was an IDE that was worthy of the title of **Best PHP IDE of All Time**, PHPStorm is it[10]. It is a comprehensive, cross-platform, plugin-based, IDE. With extensive support for **Vagrant**, **Git** and **Subversion**, **Syntax Highlighting**, and **Remote Debugging** amongst so many other features, PhpStorm is literally a one-stop shop for everything you need for PHP. (no, I'm not paid to say that).

If you've not tried it out yet, I strongly encourage you to do so. However, if you're more comfortable with **VIM**, **Emacs**, **Netbeans** etc, don't feel the need to change.

[10]http://www.jetbrains.com/phpstorm/

Navicat Lite



videoId	title	directorId	duration	synopsis	genre	website
11	Die Hard	1	131	A truly excellent fi	Action	http://www.dieha
12	Die Hard 2	1	120	Not as good as th	Action	http://www.dieha
13	Expendables 1	1	120	Lorem ipsum dolo	Action	
14	Expendables 2	1	121	Lorem ipsum dolo	Action	
15	Expendables 3	1	130	Lorem ipsum dolo	Action	
16	The Shawshank Re	1	114	Lorem ipsum dolo	Action	
17	The Godfather	1	190	Lorem ipsum dolo	Action	
18	The Godfather Par	1	186	Lorem ipsum dolo	Action	
19	The Dark Knight	1	130	Lorem ipsum dolo	Action	
20	Pulp Fiction	1	140	Lorem ipsum dolo	Action	
21	The Good, the Bac	1	120	Lorem ipsum dolo	Action	
22	Schindler's List	1	130	Lorem ipsum dolo	Action	
23	12 Angry Men	1	112	Lorem ipsum dolo	Action	
24	The Lord of the Ri	1	119	Lorem ipsum dolo	Action	
25	Fight Club	1	163	Lorem ipsum dolo	Action	
26	The Lord of the Ri	1	117	Lorem ipsum dolo	Action	
27	Star Wars: Episode	1	112	Lorem ipsum dolo	Action	
28	Inception	1	130	Lorem ipsum dolo	Action	
29	Forrest Gump	1	154	Lorem ipsum dolo	Action	
30	One Flew Over the	1	145	Lorem ipsum dolo	Action	
31	The Lord of the Ri	1	260	Lorem ipsum dolo	Action	
32	Goodfellas	1	150	Lorem ipsum dolo	Action	

```
select *, rowid "NAVICAT_ROWID" from "main"."tblvideos"  limit 0,1000
```

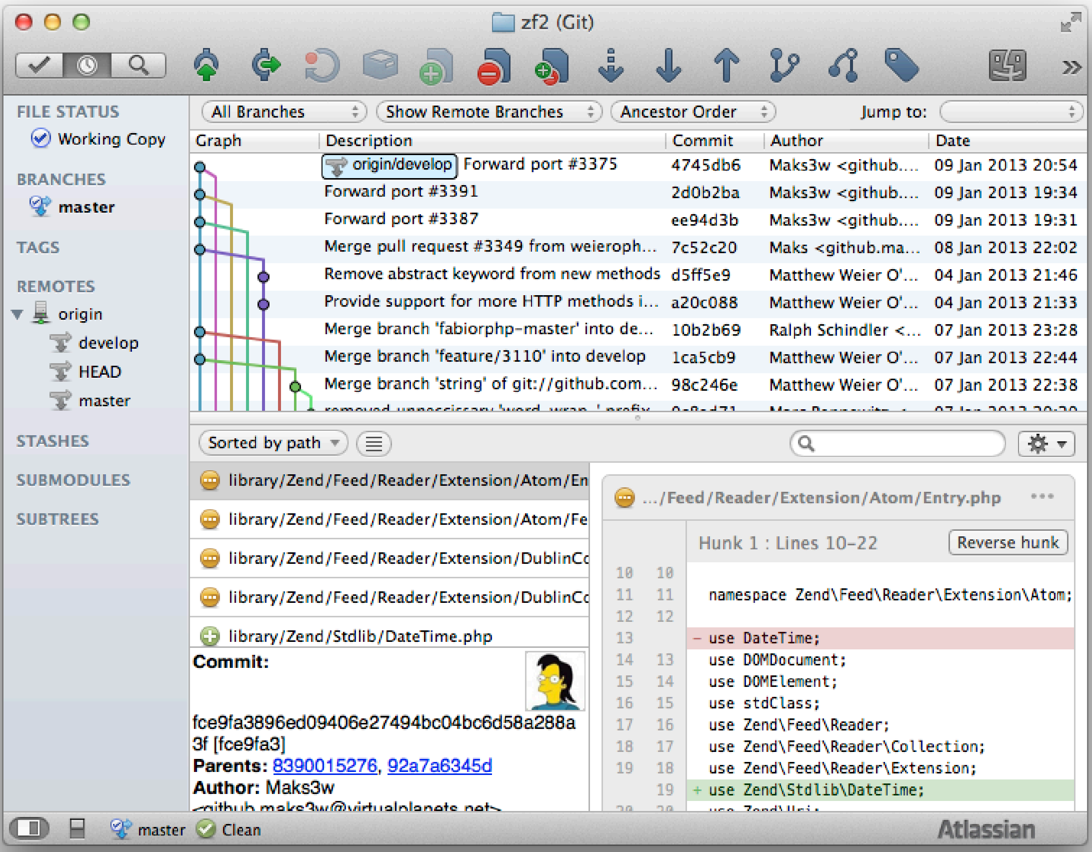
22 records in page 1

Navicat Lite Showing Table Records

There are a lot of database GUIs around. But the one which I've found to be most reliable and feature-rich is Navicat Lite[11] - what's more, *it's free*. Whether you're working with *MySQL*, *PostgreSQL*, *SQLite*, or other database vendors, it has an intelligent and well thought out GUI, which helps increase your productivity.

[11]http://www.navicat.com/download

SourceTree



Graph	Description	Commit	Author	Date
	origin/develop Forward port #3375	4745db6	Maks3w <github....	09 Jan 2013 20:54
	Forward port #3391	2d0b2ba	Maks3w <github....	09 Jan 2013 19:34
	Forward port #3387	ee94d3b	Maks3w <github....	09 Jan 2013 19:31
	Merge pull request #3349 from weieroph...	7c52c20	Maks <github.ma...	08 Jan 2013 22:02
	Remove abstract keyword from new methods	d5ff5e9	Matthew Weier O'...	04 Jan 2013 21:46
	Provide support for more HTTP methods i...	a20c088	Matthew Weier O'...	04 Jan 2013 21:33
	Merge branch 'fabiorphp-master' into de...	10b2b69	Ralph Schindler <...	07 Jan 2013 23:28
	Merge branch 'feature/3110' into develop	1ca5cb9	Matthew Weier O'...	07 Jan 2013 22:44
	Merge branch 'string' of git://github.com...	98c246e	Matthew Weier O'...	07 Jan 2013 22:38

library/Zend/Feed/Reader/Extension/Atom/En
library/Zend/Feed/Reader/Extension/Atom/Fe
library/Zend/Feed/Reader/Extension/DublinCo
library/Zend/Feed/Reader/Extension/DublinCo
library/Zend/Stdlib/DateTime.php

Commit:
fce9fa3896ed09406e27494bc04bc6d58a288a3f [fce9fa3]
Parents: 8390015276, 92a7a6345d
Author: Maks3w

.../Feed/Reader/Extension/Atom/Entry.php

Hunk 1 : Lines 10-22

```

 namespace Zend\Feed\Reader\Extension\Atom;

- use DateTime;
 use DOMDocument;
 use DOMElement;
 use stdClass;
 use Zend\Feed\Reader;
 use Zend\Feed\Reader\Collection;
 use Zend\Feed\Reader\Extension;
+ use Zend\Stdlib\DateTime;
```

SourceTree Viewing the ZF2 Codebase

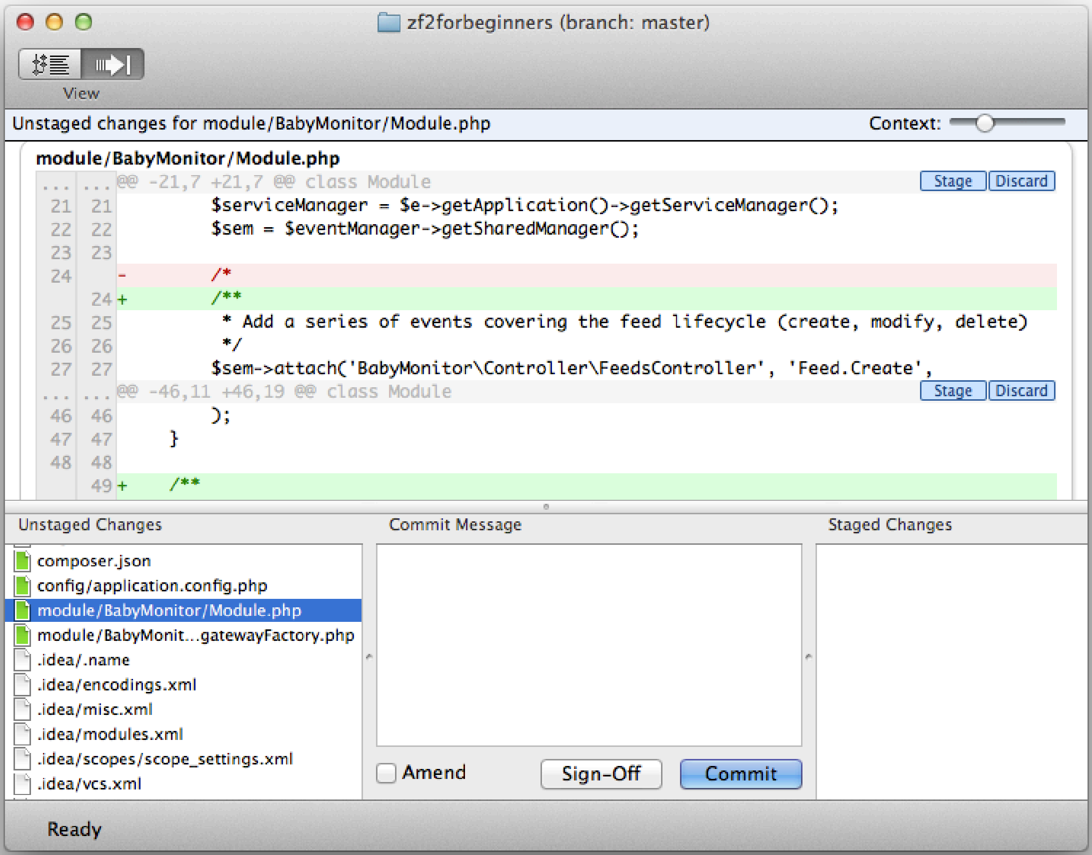
As Martin Fowler[12], author of Patterns of Enterprise Application Architecture (PoEAA)[13], said[14]:

> A shout out to developers of SourceTree - a nice GUI for git and hg. Useful even for a command-line fan like me.

It really is a very well designed and developed front end for working with Git repositories. Unfortunately it's not, yet, available for Linux. Sourcetree[15] puts everything you need at your fingertips, without overwhelming you, like other GUIs can. If you're keen to look back through the repository as it developed, Sourcetree makes it really easy.

[12]http://martinfowler.com
[13]http://martinfowler.com/books/eaa.html
[14]https://twitter.com/martinfowler/status/271001318888460290
[15]http://www.sourcetreeapp.com

GitG/GitK/GitX



Unstaged changes for module/BabyMonitor/Module.php

module/BabyMonitor/Module.php

```
@@ -21,7 +21,7 @@ class Module
         $serviceManager = $e->getApplication()->getServiceManager();
         $sem = $eventManager->getSharedManager();

-        /*
+        /**
          * Add a series of events covering the feed lifecycle (create, modify, delete)
          */
         $sem->attach('BabyMonitor\Controller\FeedsController', 'Feed.Create',
@@ -46,11 +46,19 @@ class Module
         );
     }

+    /**
```

composer.json
config/application.config.php
module/BabyMonitor/Module.php
module/BabyMonit...gatewayFactory.php
.idea/.name
.idea/encodings.xml
.idea/misc.xml
.idea/modules.xml
.idea/scopes/scope_settings.xml
.idea/vcs.xml

The GitX GUI

If you're on Linux, a great Git front-end is GitG[16]. Whilst not as full featured as SourceTree, it's still an excellent tool.

[16]https://wiki.gnome.org/action/show/Apps/Gitg?action=show&redirect=Gitg

Composer Support

The Composer Homepage

Composer is a tool which I'm eternally indebted to, when developing with PHP. If you're not familiar with Composer, the Composer site[17] describes its as follows:

> Composer is a tool for dependency management in PHP. It allows you to declare the dependent libraries your project needs and it will install them in your project for you.

In short, Composer is for PHP, what APT[18] or Yum[19] are for Linux, RubyGems[20] is for Ruby, and PIP[21] is for Python. With a simple `composer.json` file, you can ensure that your project has all the

[17]http://getcomposer.org
[18]http://en.wikipedia.org/wiki/Advanced_Packaging_Tool
[19]http://www.linuxcommand.org/man_pages/yum8.html
[20]https://rubygems.org
[21]http://www.pip-installer.org/en/latest/

dependencies, at the specific versions required.

Gone are the days of writing your application, where other developers experience issues because a library, dependency or third party package either isn't in their environment, or isn't at the minimum version required.

What's more you're able to document your application's dependencies, making the application much more self-documenting; in addition to the file, class and function documentation. With all this said though, there are really only three commands which you need when using composer:

- `composer self-update`: ensures that composer itself is fully up to date with the current release, or a specific release if you prefer
- `composer install`: resolves the packages dependencies adding them to a new vendor directory;
- `composer update`: retrieves the latest versions of the dependencies

There are a wealth of other commands available, which you can find documented on getcomposer.org[22]. If you'd like a simpler way of learning composer, check out the Composer Cheatsheet[23]. This introduction has only touched the tip of the proverbial iceberg. I encourage you to familiarize yourself with Composer, a tool I believe was long overdue.

Installing on Linux

To install it on Linux, run the following command from your terminal:

```
curl -sS https://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

This will download it, store it under `/usr/local/bin/`, calling it composer, where it is now globally available to all users.

Installing on Windows

To install it on Windows, download and run the Composer Setup application[24].

[22] http://getcomposer.org
[23] http://www.sitepoint.com/composer-cheatsheet/
[24] https://getcomposer.org/Composer-Setup.exe

Testing

From flicking through the table of contents, you may have noticed there's no specific section strictly dedicated to testing. There's a reason for that. To me, testing isn't a separate section from development, but an integral part of the process.

So to separate it out makes no sense. So as we move through the book and develop the respective sections, we'll be doing testing for that section before we develop it.

That way, we can be sure the code we're implementing will be covered before we add it and it stays mentally in context. We'll be using PHPUnit. If you're keen to use other tools, such as PHPSpec[25], feel free to do so. I'd love to see what you come up with.

Required Knowledge

This book assumes you're familiar, though not an expert, with the following technologies:

- PHP (v5.4 min)
- Basic SQL[26] for creating queries with Where/Limit/Order By etc
- Familiarity with using micro or full-stack frameworks. PHP-specific frameworks will be of greater assistance, but aren't specifically necessary

If you're not well versed in any of these, there are some excellent resources and tutorials in the resources section to get you started.

[25]http://www.phpspec.net
[26]http://www.sqlcourse.com/index.html

Key Concepts

NB: The core of this chapter is taken from the beginning ZF2 section on Master Zend Framework[27].

Introduction

Ok, let's get underway learning Zend Framework 2. In this chapter, we get a basic understanding of the key concepts, which you need to know.

These are:

- The Module Manager
- The Event Manager
- The Service Manager

Module Manager

> If I have seen further it is by standing on the shoulders of giants.

Though I and countless others enjoyed the 1.x series, it did leave some things to be desired, **quite a lot actually**. Zend Framework 2 makes up for those misgivings in a number of ways. The first one is the Module Manager, written by Evan Coury. Quoting directly from the manual[28], here is what you can expect from 2.x Module.

> Zend Framework 2.0 introduces a new and powerful approach to modules. This new module system is designed with flexibility, simplicity, and re-usability in mind. A module may contain just about anything: PHP code, including MVC functionality, Library code, view scripts, public assets such as images, CSS, and JavaScript.

Matthew Weier O'Phinney, *Zend Framework 2 Project Lead*, says it most succinctly on his blog[29]:

> In ZF2, a module is simply a namespaced directory, with a single "Module" class under it; no more, and no less, is required.

And here's the description from the Modules author himself, Evan Coury[30]:

> A re-usable piece of functionality that can be used to construct a more complex application.

[27] http://www.masterzendframework.com/tutorial/zend-framework-2-modules-the-applications-heart
[28] http://framework.zend.com/manual/2.0/en/modules/zend.module-manager.intro.html
[29] http://mwop.net/blog/267-Getting-started-writing-ZF2-modules.html
[30] http://evan.pro/zf2-modules-talk.html#slide1

What Are Modules

But what exactly are modules? Modules are composed of three key concepts:

- **The Module Autoloader** - a specialized autoloader responsible for the locating and loading of modules' Module classes, from a variety of sources
- **The Module Manager** - which takes an array of module names and fires a sequence of events for each one
- **ModuleManager Listeners** - event listeners can be attached to the module manager's various events

I'm not going to do you the injustice of parroting the manual; but instead say it as I understand it. In the 1.x series, an application could be seen as a series of partially coupled components.

The 2.x series changes all that. Each module can stand alone, and is designed to work together with other modules or provide the application's entire functionality by itself. This is made possible through the 3 previously mentioned components.

For a great example of this, checkout the excellent and rapidly growing ZF Modules project[31]. It's still a fairly new project, but you can see that as new as it is, there's likely a module to suit your needs.

Here's some good ones:

- SlmMail[32] from Jurian Sluiman[33]
- A static pages module[34] from MWOP (Matthew Weier O'Phinney)[35]
- A ZendDi compiler utility module[36] from Ocramius[37]

Each module that you write can also provide its own *views*, *layouts*, *images*, *css* and *javascript* files, and anything else that it needs. It can even make use of other modules, so that you only need to write **the minimal amount of code that's required**.

Module Configuration Paths

In the `module_listener_options` array key, you'll see:

[31]http://modules.zendframework.com
[32]https://github.com/juriansluiman/SlmMail
[33]https://juriansluiman.nl
[34]https://github.com/phly/PhlySimplePage
[35]http://mwop.net
[36]https://github.com/Ocramius/OcraDiCompiler
[37]http://ocramius.github.io

```
'config_glob_paths'    => array(
  'config/autoload/{,*.}{global,local}.php',
),
```

What this does is to tell the ModuleManager which configuration files to combine together in to the aggregate configuration the application will use. Any file under `config/autoload/` which ends in either `global.php` or `local.php` will be merged together to form the application's configuration settings.

Important Functions

getConfig

The module configuration I've covered so far is the stock one. It's made possible by the following function in `Module.php`:

```
public function getConfig()
{
  return include __DIR__ . '/config/module.config.php';
}
```

It returns an array or Traversable object that contains the environment delineated configuration for your application, i.e., *production*, *staging*, *development*, etc. In it you can include a number of configurations areas, such as **dependency injection**, **routing**, **views** and more.

getAutoloaderConfig

You have to make sure that the Module Autoloader can make best use of your module implement the `getAutoloaderConfig` function. This informs the Module Autoloader where to find the required classes.

The Remaining Module Structure

Now what I've covered so far is a basic module. Let's look at the remainder of the standard module directory structure.

src

src stores all the PHP code files for your application. Similar in style to the 1.x series it can also include directories for:

- **Controller** - All your modules controller files with the accompanying actions
- **Form** - All your form objects
- **Model** - All your database objects

view

Not much different from version 1, in version 2 all of the view templates are located under view using a directory structure which by default adheres to the following naming convention:

```
module/
  controller/
    <action name>.phtml
```

test

This is a very welcome inclusion in 2.x. I felt so often that testing was really an after thought in Zend Framework 1? Despite documentation being available on the Internet, there always seemed to be:

1. Problems using it
2. A lack of consistency in the information provided
3. Gaps in what was written

As I originally followed the testing section of the Zend Framework 2 manual, I was relieved to see it work first time; **no issues and no hassles**. It was a real pleasure to see how easy and integral testing was now considered. Nothing gives more confidence than solid tests when developing applications.

language

Unless you're developing a very localized application, the possibility of not localizing or internationalizing it is very slim. So *localization* is baked in, right from the get go.

If you're getting started with the skeleton application, which I strongly suggest you do, you'll see it contains an Application module that has its language directory pre-populated with **PO** and **MO** files for a wide assortment of languages.

However the book doesn't cover translations, so I'm not going to cover it in further depth. However it's handy to know that it's available.

Other Essentials

autoload_classmap.php

You may be familiar with an option in the 1.x series that allowed you to compile a class list so that it didn't have be searched on every invocation.

An even better setup is integrated in to the 2.x series. Under your module, you can add a file, called `autoload_classmap.php`. This returns an array of classes which is PSR-0 compliant[38]; then use the built in `classmap_generator.php`, script, available in /vendor/bin/, to keep it up to date.

To create a classmap file for a module, run the following command from the project root directory:

```
php ../vendor/bin/classmap_generator.php --library <Your Module> --output <Your \
Module>/autoload_classmap.php --overwrite --sort
```

For a good understanding of the new autoloader, check out the great post by Rob Allen (one of the core ZF2 contributors)[39].

module_paths

As you may have seen in the code snippet above, there was an array key called `module_paths` in the configuration returned from `application.config.php`. This is part of what makes the new module structure so flexible. The directories specified there are the paths that the Module Autoloader searches when loading modules.

If you want to stick with the recommended structure, then keep './module' in there. If you want to go with a different structure, change the name. I like the configuration provided, because you keep your modules under **./modules** and add 3rd party libraries under **./vendor**. That way you know what's yours and what's external.

It's a simple approach, which I believe will pay off big maintenance dividends down the track.

Key Points To Be Aware Of

Though all this change is very exciting, please be aware that you can't just make a simple transition to the 2.x series. The 2.x series takes a lot of the PHP 5.3 advancements in to account.

To make as smooth transition as possible ensure that you're conversant with:

- Namespaces[40]
- Closures / Anonymous Functions[41]

For a good understanding of what's required to migrate a Zend Framework 1 application to Zend Framework 2, read this presentation[42] by Bart McLeod (Zend Framework 2 developer and contributor).

[38]http://www.php-fig.org/psr/psr-0/
[39]http://akrabat.com/zend-framework-2/using-zendloaderautoloader/
[40]http://www.php.net/manual/en/language.namespaces.php
[41]http://de3.php.net/manual/en/functions.anonymous.php
[42]http://spaceweb.nl/presentations/zf1-zf2-pfc.pdf

The EventManager

The Event Manager is the component of the framework which allows you to both setup and hook in to events in your application's lifecycle. These can be events which come with Zend Framework 2[43] or ones which you define yourself.

There are events for modules, controllers and views, covering *routing*, *errors*, *dispatching*, *bootstrapping*, *rendering* and *responses*. Let's look at it in a bit more detail.

It Implements the Observer Pattern

This is the key pattern to be familiar with when understanding the EventManager. The concept, according to Wikipedia[44], is simply this:

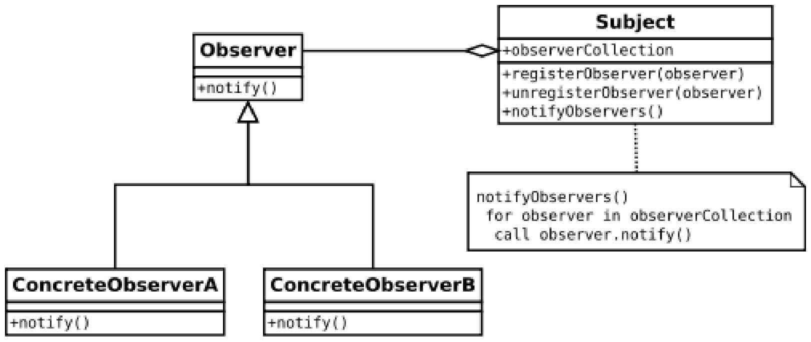
> An application has an object, which has a registered list of observers. It performs its actions as normal and notifies the observers when a specific event has occurred. These observers are then able to take action relevant to only themselves.

On the surface of it, this is a rather simple and effective method of approaching development, as it has several advantages. These are:

- **No Monster Classes**: code can be written with a single purpose in mind, following the Single Responsibility Principle[45]
- **Easier to extend**: So much code starts out simply, then new requests and desires appear. With the Observer pattern it's simple to satisfy both aims
- **Easier to test**: Imagine trying to test a monolithic codebase, or one which has so many different needs and desires? When each part is simple, concise and decoupled, it makes the job of testing far simpler.

[43]http://akrabat.com/zend-framework-2/a-list-of-zf2-events/
[44]http://en.wikipedia.org/wiki/Observer_pattern
[45]http://www.objectmentor.com/resources/articles/srp.pdf

It Implements Aspect-Oriented Programming (AOP)



Aspect Oriented Programming

Wikipedia says this[46] about Aspect-Oriented programming:

> The motivation for aspect-oriented programming approaches stem from the problems caused by code scattering and tangling. The purpose of Aspect-Oriented Software Development is to provide systematic means to modularize crosscutting concerns.
>
> The implementation of a concern is scattered if its code is spread out over multiple modules. The concern affects the implementation of multiple modules. Its implementation is not modular. The implementation of a concern is tangled if its code is intermixed with code that implements other concerns. The module in which tangling occurs is not cohesive.

In short, think about it this way: *Write clean, well organized, code, with one purpose in mind; code which doesn't try to solve more than one problem*. For example, if you write code which searches a datasource for information matching a query, it doesn't attempt to render the information for the user as well.

It has an Event-Driven Architecture

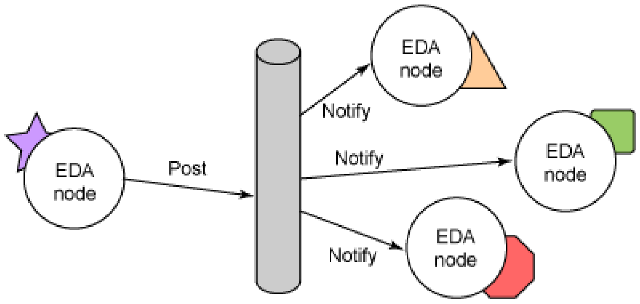
The following two statements from an article on Event-driven architecture show why it's an excellent approach to take:

> Building applications and systems around an event-driven architecture allows these applications and systems to be constructed in a manner that facilitates more responsiveness, because event-driven systems are, by design, more normalized to unpredictable and asynchronous environments.[2]

[46]https://en.wikipedia.org/wiki/Aspect-oriented_programming

> Event-driven architecture can complement service-oriented architecture (SOA) because services can be activated by triggers fired on incoming events.[2][3] This paradigm is particularly useful whenever the sink does not provide any self-contained executive.

To help you out a bit more, here's a simple, diagram courtesy of IBM[47]



EDA Architecture, courtesy of IBM

I hope you can see from this definition how Event-Driven architecture and Aspect-Oriented design serve to compliment each other. By implementing them, Zend Framework 2 allows you to write complex applications in a cohesive, yet decoupled, manner.

Because you can write a series of components (or sub-components) all focusing on doing one task really well. These components are then linked together through listeners and are triggered (or called in to action) as needed. I'd suggest this, in combination with good lazy-loading, makes it even simpler and more effective.

What else can you do?

Here's what else you can do with events:

- **Attach to many events at once**: Need to listen on multiple events, such as logging the occurrence of all actions in a controller being fired?
- **Detach listeners**: Maybe a listener is no longer required, or not available due to a service outage; then remove it from the list.
- **Short-circuit execution:** This helps you stop other listeners executing if, for example, there's no further work to be done. It's also handy for performance by flattening out the call stack.
- **Set priorities on events**: This is handy if different listeners need to fire off earlier or later in the event process. A trivial example is form validation where you want to log information before and afterwards. The higher the number, the higher its priority — and the earlier it will fire.

[47]https://www.ibm.com/developerworks/library/ws-soa-eda-esb/#N100A3

Why Events Are Good

Events, when used properly, are a good thing as they let your code listen for and respond to events throughout the application's lifecycle. Say for example, your application, a simple one like what we'll be making here, has registered events for when records are *created*, *updated* and *deleted*.

Your boss walks in and says that it now needs to send a tweet when a new record's created. Who knows why, but it's now *a must-have feature*. Traditionally, you *might* have extended the database class to add the tweet functionality.

Or you may have instantiated a Twitter object in your controller, after the original save work was completed, sending the tweet. The first way is just all-round wrong, the second way, not too bad. But both ways, especially the first, can make the code much less maintainable.

Since you have events registered which listen for the record creation event, you can hook in there to send a tweet, retrieving or instantiating the relevant class, which can retrieve relevant information from the event which's just fired and not need to know anything else about the state of the application.

This allows you to add the required functionality, but in a much more maintainable way.

Service Manager

The Zend Framework 2 Service Manager simplifies web application development in a number of ways, primarily by making configuration a breeze. In this chapter we'll step through what the ServiceManager is, how to use it and how it simplifies application development time.

The **ServiceManager** component is a highly critical, but potentially misunderstood, aspect of the framework; one that when understood makes the rest of the framework a breeze, well sort of. It's often believed that the ServiceManager over-complicates the entire framework. Some people say it's an attempt to *out Java Java*.

I completely disagree!

I believe that it's this fundamental concept that **makes the framework simpler**, that makes developing web applications with it easier, less stressful and more maintainable.

What is the ServiceManager

Put in simplest terms, the ZF ServiceManager component is an implementation of the Service Locator pattern[48]; in short a simple application registry that provides objects (*in a lazy-loaded fashion*) within application as needed — but with some nice additions.

[48]http://en.wikipedia.org/wiki/Service_locator_pattern

It allows us to perform Inversion of Control (IoC)[49], which allows us to remove dependencies and tight coupling from our applications. The result of this is **applications that are simpler to build, test and maintain**.

You can try to think of it in more complex ways if you'd like to, but really there's no need. Where it does get a little complicated however, at least initially, is how it's all configured. It does take a bit of getting your head around — but not too much.

How do you Configure it?

The Service Manager is able to be configured in a number of ways, the key ones are:

- **factories**: A fully qualified class name, PHP callable object, or a class implementing `Zend\ServiceManager\Fa`
- **invokables**: A string, which is a fully qualified class name, able to be instantiated later
- **initializers**: A fully qualified class name, PHP callable object, or a class implementing `Zend\ServiceManager\InitializerInterface`. If listed, will be applied to objects retrieved from the Service Manager to perform additional initialization
- **configuration classes**: These are classes implementing `Zend\ServiceManager\ConfigInterface`. Classes implementing this interface are able to configure the Service Manager, performing the initialization that is covered in these five points
- **aliases**: An associative array of aliases to services (or aliases to aliases). It may not seem like a good idea, but from a readability perspective, I believe this is an excellent technique to have available. What's more, they help alleviate namspace clashes, where one class has the same name as another. Aliasing one of the conflicting keys to another can resolve the clash.

Quick Note

There are a few others, but they're outside the scope of the book.

Dependency Injection

This is likely one of the most controversial choices in the Framework, basing so much of it around the use of a Dependency Injection (DI)[50] implementation. You may not, consciously, have a lot of experience with dependency injection and dependency injection containers. But you'll likely have used them at one time or another, without realizing it.

What's more, DI containers aren't new; Symfony has one, there's a more common, framework-agnostic one, PHP-DI[51], Aura Di[52] and Pimple[53]. The majority of the critique of them comes from two fronts:

[49]http://en.wikipedia.org/wiki/Inversion_of_control
[50]http://en.wikipedia.org/wiki/Dependency_injection
[51]http://php-di.org
[52]http://auraphp.com/packages/Aura.Di/
[53]http://pimple.sensiolabs.org

1. That by using them, the configuration of the application isn't as transparent
2. It makes maintaining applications quite complex

I'll discount the often cited argument:

> but what if you leave and our other developers don't have those skills, so we can't maintain it

This isn't valid and smacks of laziness as well as an unwillingness to learn and try new things. But before I get in to how to use it in Zend Framework 2, I want to debunk both of the prior stated objections.

When done correctly, you'll have a central location where all of your objects are configured. When you need to make a change, you only need look in the repository for the relevant configuration and change it there.

After the change is applied every reference to objects retrieved from the repository reflect the change. You don't need to search for usages throughout the application and refactor them. You don't need to update multiple tests. The changes benefit the entire application.

What Has This Shown Us?

This process of stepping through the configuration and usage of the ServiceManager has shown us 5 things:

1. **Configuration Is Straight-Forward**: Whilst a bit involved (because of all the locations where configurations can be located) it follows a clearly defined set of conventions, in a standard set of locations.
2. **It's Easy to Debug**: By having standard conventions and locations, we can track configurations in the application logically and systematically.
3. **It Works across Modules**: In Zend Framework 1, we had to jump through hoops to work with multiple modules. In ZF2 it's all baked in. Just follow the conventions and it works.
4. **Applications are More Dependable**: This is a bit of a reiteration of the previous points. But by being able to configure applications in a readily debuggable manner, you need less time and effort; effort which can be spent more meaningfully in other aspects of development.
5. **Easier to Maintain**: When applications as more *consistent*, *predictable*, *debuggable*, *logical* (in my mind if nowhere else), they're easier to optimize. Why? Because you're working with a known quantity.

Possibly a little boring, but following convention makes adding new components quicker and more predictable. Being a maintainer, you know how applications hang together, you know where to look, the options which can be used, how to override them and so on.

Chapter Recap

Please don't feel cheated with this introduction. To be honest, there's a hell of a lot that can be involved in making full use of the new Modules system, despite what I said at the start. My aim here was to give you a simple and effective introduction that we can build on progressively.

Routing

Now that we have a, basic, working application, let's start to dive deeper in to how it's configured and constructed, starting with routing. In Zend Framework 2 routing is quite similar to version 1, supporting a wide variety of possibilities[54] for composing routes.

Given that, I want to show you the simplest, least energy intensive, ways to configure it. To do that let's review the routing table, how it's constructed, and how it works. When it's completed it's going to support routes for the following actions:

- Adding Feeds
- Editing Feeds
- Deleting Feeds
- Searching Feeds
- Viewing All Feeds

All of the routes will have the prefix `/baby-monitor`, as they're child routes of the original parent and will inherit all of the configuration and settings of the parent. Here's what they'll look like and what they'll do.

Route Name	**Route**	**Description**
Managing Feeds	`/baby-monitor/feeds/manage[/:id]`	Supports creating new feeds and update existing feeds. A feed is located based on the value of `id` route parameter.
Deleting Feeds	`/baby-monitor/feeds/delete[/:id]`	Supports deleting an existing feed based on the value of `:id`. A feed is located based on the value of `id` route parameter.
Viewing Feeds	`/baby-monitor/feeds/view[/:id]`	Supports displaying a feed based on the value of `:id`. A feed is located based on the value of `id` route parameter.
Feeds Index	`/baby-monitor`	Display all feeds
Search Feeds	`/baby-monitor/feeds/search[/:startDate][/:endDate]`	Supports searching feeds based on the values supplied in the `:startDate` and `:endDate` route parameters.

[54] http://framework.zend.com/manual/2.3/en/modules/zend.mvc.routing.html

Deleting Feeds

Starting with the delete action, add the following route in `config/module.config.php`, after the `action` element, in the `child_routes` element.

```
    'delete' => array(
      'type' => 'segment',
      'options' => array(
        'route' => '/feeds/delete[/:id]',
        'constraints' => array(
          'id' => '[0-9]+',
        ),
        'defaults' => array(
          'action' => 'delete',
        )
      ),
      'may_terminate' => true,
    ),
```

This defines the route `/baby-monitor/feeds/delete`. Notice the route definition contains `[/:id]`? This defines an optional, named, parameter called `id`. If it's supplied in the request, the **constraints** element limits the value that can be supplied to only integers. This is done by supplying the regex (regular expression): '[0-9]+''.

New to Regular Expressions?

If regexes are new to you, I suggest getting to know the basics[55]. Not only do regexes come in very handy, but they save you a lot of time and effort, even when the basics are mastered.

We've also defined a **defaults** element; this defines the action which will handle the request when it's matched, specifically the **delete** action. We didn't need to specify the controller, as it was defined in the parent route, which this route inherits.

Managing Feeds (Adding & Editing)

[55] http://www.regular-expressions.info

```
'manage' => array(
  'type' => 'segment',
  'options' => array(
    'route' => '/feeds/manage[/:id]',
    'constraints' => array(
      'id' => '[0-9]+',
    ),
    'defaults' => array(
      'action' => 'manage',
    )
  ),
  'may_terminate' => true,
),
```

Next we've defined the `/baby-monitor/feeds/manage` route, which is almost identical to the delete route. I'll skip going over it as the only thing things which have changed are the name, base of the route and the default action which handles the request, which is **manage**.

Viewing Feeds

```
'manage' => array(
  'type' => 'segment',
  'options' => array(
    'route' => '/feeds/view[/:id]',
    'constraints' => array(
      'id' => '[0-9]+',
    ),
    'defaults' => array(
      'action' => 'view',
    )
  ),
  'may_terminate' => true,
),
```

Next we've defined the `/baby-monitor/feeds/view` route, which again is almost identical to the delete and manage routes, and handled by the **view** action.

The Search Route

Finally, let's look at the search route.

```
'search' => array(
  'type'    => 'segment',
  'options' => array(
    'route'    => '/feeds/search[/][:start][/:end]',
    'constraints' => array(
      'start' => '[0-9]{4}(-[0-9]{4}){2}',
      'end'   => '[0-9]{4}(-[0-9]{4}){2}',
    ),
    'defaults' => array(
      'action' => 'Search'
    ),
  ),
),
```

As in the first example, we've specified options for *route name*, *route*, *constraints* and *defaults*. Based on the first example, you can see that the route's called `search` and we've specified two optional constraints, **start** and **end**. These will, later, support two elements: `startDate` and `endDate`.

We've then specified a regex constraint for both, which requires the dates supplied to be in the format `yyyy-mm-dd`. Finally, we've specified an extra default, `action`, which will route all matching requests to the Search action.

Why That Date Format?

Just a quick note on the date format; I've chosen this format, because *I'm Australian* and it makes more sense to me than the U.S. format of `yyyy-dd-mm`. Ok, this regex wouldn't care whether you specified it in US or standard English format. I'm mentioning it to avoid any confusion later, because the way the code will process it is the standard English format.

Try Dispatching to Each Route

Now that all the routes are setup, take a moment to route to all of them, to see what they're like and make sure they work.

Using Route with ViewHelpers

With the routes setup, let's look at how we can make use of the routes, and demonstrate their flexibility, in view templates. One of the best and most straight-forward examples is by using the URL ViewHelper in a view template.

This will show how to create a url href, just by supplying the route name, with some optional parameters. For example, to generate the route `/baby-monitor`, we'd invoke the url view helper, in any view template, like this:

```
<?php print $this->url('feeds', array(
        'controller' => 'feeds',
        'action' => 'index'
 )); ?>
```

This will generate a route based on the configuration in the route named `feeds`, and supply the parameters in the associative array supplied as the second parameter. This works for both parent and child routes.

Referencing Child Routes

It may not be clear at first how to reference a child route, so let's look at how it's done in the following example, where we'll reference the `delete` and `manage` child routes.

```
// Generate the delete feed route
<?php print $this->url('feeds/delete', array('id' => 1)); ?>

// Generate the manage feed route
<?php print $this->url('feeds/manage', array('id' => 1)); ?>
```

In both of these examples, we've started with the parent route name, `feeds`, then added the child route name on to the end, separated by a slash, supplying the route parameters as and where required. Each route supports an optional parameter, id, which has been supplied in the second parameter.

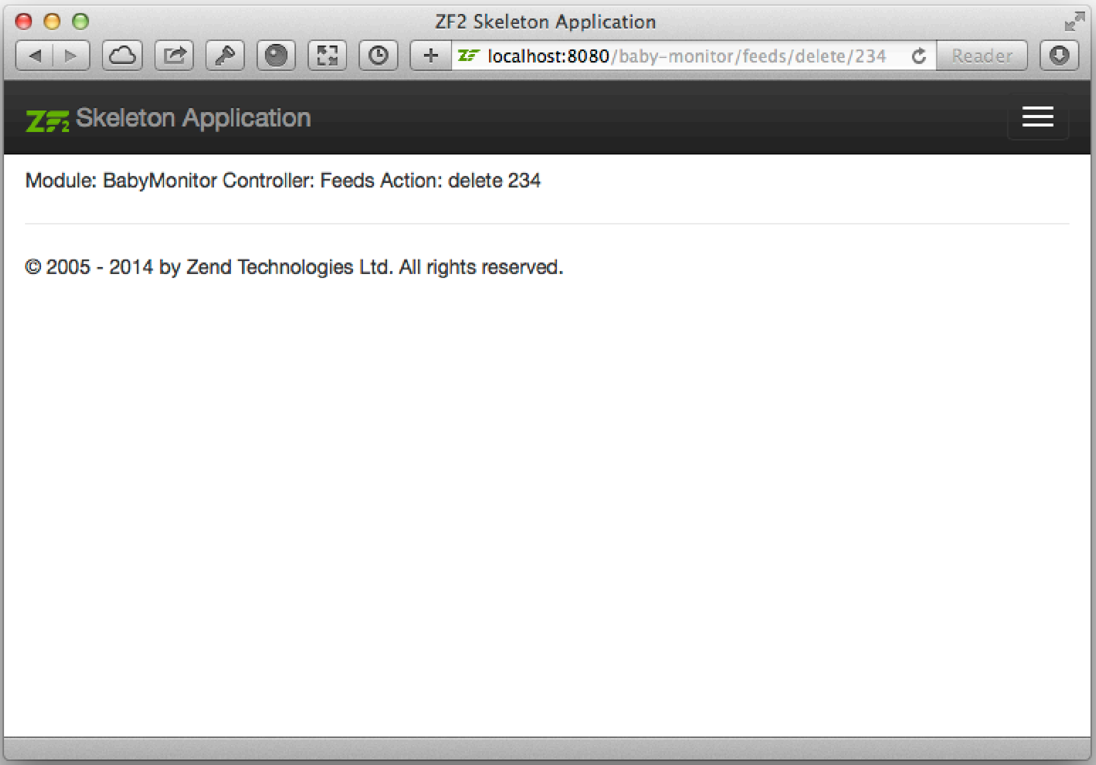
Accessing the Parameters in Controller Actions

As we saw in the last example both the delete and manage both accept one parameter, id, let's look at how to get access to it, in a controller action, if it's supplied in a request. Assuming that you've dispatched to `http://localhost:8080/baby-monitor/feeds/delete/234`, let's see how we retrieve 234 in the delete action.

```
public function deleteAction()
{
  return new ViewModel(array(
    'id' => $this->params()->fromRoute('id', 0)
  ));
}
```

Here, I've initialized a view template variable `id` by calling `fromRoute()` on the `params` controller helper. This retrieves the route parameter named `id`. Notice `0` as the second parameter to `fromRoute`? This supplies a default value, in case `id` is not available.

To complete the example, I've added `<?php print $this->id; ?>` to **delete.phtml**. Now, when you route to `http://localhost:8080/baby-monitor/feeds/delete`, if you supply nothing, `0` will be displayed, along with the other template information. If, on the other hand, you dispatched to `http://localhost:8080/baby-monitor/feeds/delete/23`, `23` would be displayed.



Module: BabyMonitor Controller: Feeds Action: delete 234

© 2005 - 2014 by Zend Technologies Ltd. All rights reserved.

View route parameter

Chapter Recap

And that's the basics of setting up routing in Zend Framework 2, covering simple, literal routes, to slightly more complicated segment routes. I encourage you to explore and experiment with the other route types available.

3rd Party Modules

One of the best things about Zend Framework 2 is its modular structure. One of the key advantages of Zend Framework 2 over version 1 is that modules are now first class citizens. If you're familiar with version 1, you'll know that modules were really only modules in name only.

As well as making this much needed change, some of the contributors also started modules.zendframework.com[56]. I encourage you to check it out and consider searching the 510+ available modules (at the time of writing) before thinking of writing your own.

You may be re-implementing the wheel.

Like Packagist[57] is for the wider PHP community, ZF2 Modules is the place where the best modules can be found; helping us all save time in our development efforts.

To demonstrate just how flexible 3rd party modules make development with Zend Framework 2, we'll now integrate a third party module to implement email support in the application, which we'll use in the chapter on Events shortly.

Specifically, we're going to use the excellent SlmMail package[58] by Jurian Sluiman[59] and Michaël Gallego[60]. SlmMail provides a range of transport layers for `\Zend\Mail`, which will make it easy to integrate with the excellent Mandrill email service from MailChimp[61]. To complete this chapter, you'll need to have an account with Mandrill[62].

When you have one (or if you already have one), navigate to the SMTP and API Credentials page[63], and click **+New API Key** at the bottom of the page, add in a description and click **Create API Key**. You'll then be redirected to the page again, where you can copy the key in to `/config/autoload/slm_mail.mandrill.local.php`. Uncomment the `key`element and paste the key in.

Here's what we'll do:

1. Add the library as a dependency in `composer.json`
2. Create a class, `\BabyMonitor\Notify\Feed\EmailNotifier.php` which will use the SlmMail when creating and sending emails
3. The class will implement the `FactoryInterface` to have it's dependencies automatically supplied when it's instantiated by the ServiceManager

[56]http://modules.zendframework.com
[57]https://packagist.org
[58]http://modules.zendframework.com/juriansluiman/SlmMail
[59]https://juriansluiman.nl
[60]http://www.michaelgallego.fr
[61]http://mandrill.com
[62]https://www.mandrill.com/signup/
[63]https://mandrillapp.com/settings/index

Integrating the Library

First we need to add the dependency in `composer.json` as previously mentioned. You can either add `"slm/mail": "~1.4",` in to the `"require"` section by hand. Or from the command line, in the root of your project directory, run the following command:

```
composer require slm/mail "dev-master"
```

If you added the entry by hand to composer.json, run `composer update` and wait for that to complete. When it's done, you'll see a new directory under vendor called `slm/mail`. Feel free to explore as you have time.

Enabling the Module

With the module source now in vendor, we need to activate the module. In `config/application.config.php` add `'SlmMail',` to `'modules'` array. It doesn't matter where in the list it goes, so long as it's there.

Finally, copy `./vendor/slm/mail/config/slm_mail.mandrill.local.php.dist` to `config/autoload`, removing the `.dist` extension. With these steps carried out, you're ready to go. So let's move on and create a class which uses the functionality.

Implementing the NotifyInterface Interface

Now we'll create an interface, called `NotifyInterface`, under `module/BabyMonitor/src/BabyMonitor/Notify/Feed` which our email class, called `EmailNotifier.php`, also located in that directory, will implement. Ok an interface isn't strictly necessary. But I wanted to do it to maintain good habits. Copy the code below in to the new interface class.

```php
namespace BabyMonitor\Notify\Feed;

use BabyMonitor\Model\FeedModel;

interface NotifyInterface
{
  public function notify(
                FeedModel $feed, $notificationType
  );
}
```

You can see it declares one method, `notify()`, which takes two parameters:

- A `FeedModel` object, which provides data for the email notification
- A string, which specifies the notification type

Creating the Core Class - EmailNotifier.php

Now let's step through the notification class, EmailNotifier.php, and see how it makes use of `Slm\Mail`.

```php
namespace BabyMonitor\Notify\Feed;

use BabyMonitor\Model\FeedModel;
use Zend\Mail\Message;
use Zend\Mail\Transport\TransportInterface;
```

Firstly we specify the namespace and bring in the classes which we'll need in the class.

```php
class EmailNotifier implements NotifyInterface
{
  const DEFAULT_SUBJECT = 'Baby Monitor Feed Notification';
  const NOTIFY_CREATE = 'create';
  const NOTIFY_UPDATE = 'update';
  const NOTIFY_DELETE = 'delete';

  protected $_config;
  protected $_mailTransport;
```

Next, we define a set of constants, used for the notification states and default email subject, as well as two variables, `$_config` and `$_mailTransport`. `$_config` stores the config we'll need and `$_mailTransport` stores the Email transport. Both of these two are provided in the constructor, below.

```php
  public function __construct(
              $emailConfig,
              TransportInterface $mailTransport
        ) {
    if (empty($emailConfig)) {
      throw new EmailNotifierException(
        'Missing notifier configuration data'
      );
    }

    $this->_config = $emailConfig;
    $this->_mailTransport = $mailTransport;
  }
```

If `$emailConfig` is empty, an `EmailNotifierException` is thrown so that we know what happened. I've not included the code for that class, as it only extends the base PHP `\Exception` class. Now let's look at the `notify()` method.

```php
public function notify(FeedModel $feed, $notificationType)
{
  if (empty($this->_config['subject'])) {
    $subject = self::DEFAULT_SUBJECT;
  } else {
    $subject = $this->_config['subject'];
  }

  $message = new Message();
  $message->setBody(
              $this->getNotificationBody(
                      $feed, $notificationType
              )
      )
    ->setSubject($subject)
    ->addFrom($this->_config['address']['from'])
    ->addTo($this->_config['address']['to']);

  return $this->_mailTransport->send($message);
}
```

Here we have the core of the class, `notify()`. Firstly, we try and extract the subject from `$_config`. If it's not available, we use `DEFAULT_SUBJECT` instead. Then we create a new `\Zend\Mail\Message` object, forming the basis of the email we'll be sending.

We create the body using the `getNotificationBody()` method we'll see next, set the *subject*, *from* and *to* email addresses; and finish up by returning the result of calling the `send()` method of `$_-mailTransport`.

```php
  public function getNotificationBody(FeedModel $feed, $notificationType)
  {
    switch ($notificationType) {
      case (self::NOTIFY_UPDATE):
        $message = sprintf(
                                "Feed %d has been updated", $feed->feedId
                        );
                break;

      case (self::NOTIFY_DELETE):
        $message = sprintf(
                                        "Feed %d has been deleted", $feed->feedId
                                );
        break;

      case (self::NOTIFY_CREATE):
      default:
        $message = sprintf(
                                        "Feed has been created. Id is: %d", $feed->feedId
                                );
        break;
    }

    return $message;
  }
}
```

Right, let's look at how we construct the message body. Using a `FeedModel` object and notification type, we use `sprintf` to construct a simple message, informing the user what has happened, whether that's that a feed has been *updated*, *deleted* or *created*.

Whilst we only need the `feedId` property from the FeedModel object, I passed in the entire object, as this was the first draft of the function and I'm thinking about fleshing out the function as time goes on.

Finishing up, create a new class, EmailNotifierException, in the Notify/Feed directory. This class isn't special, as it only extends PHP's core Exception class. It's there so that we have a Notify-specific exception class, in case things go wrong. In it, add the following code:

```
namespace BabyMonitor\Notify\Feed;

class EmailNotifierException extends \Exception {

}
```

Instantiating the Class

Let's wind this up by looking at how the class is instantiated. In the final new class, EmailNotifier-Factory.php, located in `Notify/Feed/Factory`.

```php
namespace BabyMonitor\Notify\Feed\Factory;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;
use Zend\ServiceManager\Exception\ServiceNotCreatedException;
use Zend\Cache\Exception\ExtensionNotLoadedException;
use BabyMonitor\Notify\Feed\EmailNotifier;
```

As always, we start off with the namespace declaration and use statements which we'll need in the class.

```php
class EmailNotifierFactory implements FactoryInterface
{
  public function createService(ServiceLocatorInterface $serviceLocator)
  {
    $config = $serviceLocator->get('Config');
    $emailConfig = '';

    if (array_key_exists('notification', $config)) {
      $emailConfig = $config['notification'];
    }

    $mailTransport = $serviceLocator->get(
      'SlmMail\Mail\Transport\MandrillTransport'
    );

    $notifier = new EmailNotifier(
                        $emailConfig, $mailTransport
                 );

    return $notifier;
  }
}
```

Then we implement the one function, `createService()`, mandated by the `FactoryInterface` interface which we're implementing. We retrieve the application config from the ServiceLocator object, which we'll see shortly passes in the `email` element.

If it has an element called `notification`, we use it to initialize `$emailConfig`, just making it a bit easier to reference. After this, we initialize a new variable, `$mailTransport` by retrieving the `SlmMail\Mail\Transport\MandrillTransport` service which is made available in `Slm\Mail`.

This makes it really simple to use Mandrill[64] as the underlying transport for our emails. We finish up by instantiating and returning a new `EmailNotifier` object, passing in the `$emailConfig` and `$mailTransport` parameters which it needs.

Making the EmailNotifier Available to Our App

One final step's still required, defining the service in ServiceManager. As we're implementing `FactoryInterface`, then we need to add a reference (below) to the class in the `factories` element of the array returned from `getServiceConfig`.

```
'BabyMonitor\Notify\Feed\EmailNotifier' => 'BabyMonitor\Notify\Feed\Factory\Emai\
lNotifierFactory',
```

In the snippet above, I've just included the relevant code section. Here, you can see that when `BabyMonitor\Notify\Feed\EmailNotifier` is retrieved from the ServiceManager in the Events chapter, `EmailNotifierFactory` will be instantiated and the result of calling `createService()` will be returned.

We really didn't need to go to all this effort just to send an email, which contained a simple body and subject, right? No, we didn't. But it was a good example of using an external module, which is actively developed, tested and maintained, which avoids us having to commit to doing so both now and over the longer term lifetime of our application.

The Notification Configuration

We now need to provide the configuration which the notification classes will use. In `config/autoload`, create a new class, `notify.global.php` and in there, add a to and from email address as appropriate.

```php
<?php

return array(
  'notification' => array(
    'address' => array(
      'to' => '',
      'from' => ''
    ),
    'subject' => 'Baby Monitor Feed Notification'
  )
);
```

[64] https://mandrill.com

Wiring it up with Events

Now let's trigger an email to be automatically sent when a new record is created, by creating a custom event. I appreciate that I've not touched much on events in the book, so I'll keep this simple. But they're a good thing to get to know as you learn Zend Framework 2.

In Module.php, firstly add in the following use statements:

```php
use Zend\Mvc\ModuleRouteListener;
use Zend\Mvc\MvcEvent;
use BabyMonitor\Notify\Feed\EmailNotifier;
```

Then, add in the function below, then we'll work through it.

```php
public function onBootstrap(MvcEvent $e)
{
  $eventManager = $e->getApplication()->getEventManager();
  $moduleRouteListener = new ModuleRouteListener();
  $moduleRouteListener->attach($eventManager);
  $serviceManager = $e->getApplication()->getServiceManager();
  $sem = $eventManager->getSharedManager();

  $sem->attach(
                'BabyMonitor\Controller\FeedsController',
                'Feed.Create',
    function($e) use($serviceManager)
                {
                        $notifier = $serviceManager->get(
                                'BabyMonitor\Notify\Feed\EmailNotifier'
                        );
      $notifier->notify(
                                $e->getParams()['feedData'],
                                EmailNotifier::NOTIFY_CREATE
      );
    }
  );
}
```

Firstly we get access to the EventManager and by calling its `attach()` method, create a custom event, `Feed.Create`, on the FeedsController, which will fire off the closure when it's triggered. The closure will retrieve the EmailNotifier service from the ServiceManager and call its `notify()` method, passing in the event information, which we'll see in a moment, and the notification type.

The event information provides state and related information for the event to work with. In this case, a FeedModel object. Now let's update the FeedsController manage action to trigger the event after a record's been created. In the manageAction, after the call to `save()`on `$this->_feedTable`, add in the following code.

```
$this->getEventManager()->trigger('Feed.Modify', $this, array(
  'feedData' => $feed
));
```

This gets access to the EventManager and triggers the `Feed.Modify` event, by passing in the event to trigger and the object to trigger it on, followed by the event data, an associative array containing the FeedModel object. To run it, create a new feed record and check your email inbox. You should see an email with the body `Feed has been created. Id is: 1.`

Further Information

One thing I should add, before we finish up, is give you further information to work with on the module. So be sure to check out the project site[65] where you'll find copious information on the module, as well as all of the providers which are available with it.

As these are all commercial email providers, Jurian and Michaël have created a pricing table for each provider[66], so you know what the cost may be if you use them.

Chapter Recap

And that's how to integrate external modules in to your application. In case you're overly enthusiastic, please don't just grab any module and add it to your application. Make sure that you're comfortable with the developer(s) and with the code itself, that it doesn't contain anything malicious. I'm not suggesting that you should suspect everyone, but please keep a healthy respect for security as you bring in external code in to your application.

[65]http://modules.zendframework.com/juriansluiman/SlmMail
[66]http://modules.zendframework.com/docs/Pricing.md