Chapter version 0.1

1 Building modular projects -- Zend Framework 2.0

1.1 Introduction

ZF2 Beta 2 arrived with the dawn of 2012.

Key MVC components were either restructured or created anew. Now, after a hectic meeting two days ago, the push for Beta 3 has begun. Due about the end of February, Beta 3 will bring revitalised Db, Cloud, Config, Console, and View.

Unlike its predecessor, Zend Framework 2.0 uses pluggable modules. The framework will change to cope with the modular development structure.

Figure 1 shows a typical basic project hierarchy: with only the "front" Application module. From there we can build modules into the project, or plug third-party modules in as required.

ZF2 encourages tested third-party module development. As a result, the development community at large can have confidence that modules work efficiently.

As public module repositories grow, speed and convenience of robust RAD projects will grow too. In turn, demand for knowledgeable ZF2 project producers will increase. Given that, it seemed logical to provide developers a head-start whilst the inner sanctum of framework improvements progresses.

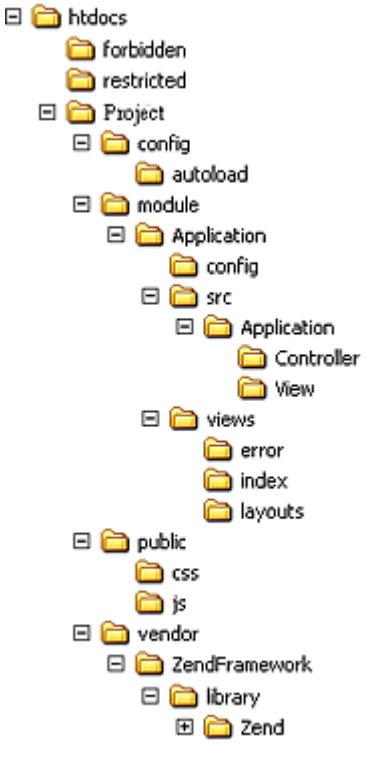

Figure 1: Basic project hierarchy

1.2 Purpose

Until later in its development cycle, Zend Framework 1.xx suffered from obfuscation. Although in its later incarnation extensive documentation and examples were provided, more than a few developers found some aspects of the framework confusing.

To overcome confusion, and reduce the learning curve, this publication provides a guide to building modular projects with Zend Framework 2.0. It will hopefully allow developers fast entry to the revised framework, provide an occasional reference, and answer a few of the more common questions.

1.3 Parts

Phase 1 of "***Building modular projects***" comprises several parts. Together they provide a core application template with login and authorisation.

1.3.1 Part 1: Set-up to follow this guide

Git, project core and other useful bits so developers can follow this guide.

1.3.2 Part 2: A foundation project core

This this part reviews the most basic ZF2 application. Some pointers and considerations are set out as a foundation upon which developers can build. It can serve as the core of new projects, and to test new modules.

By using this project core, developers can satisfy themselves that every new module, and those obtained from public repositories, works alone as intended prior to plugging into larger productions.

1.3.3 Part 3: Adding Mmalrc module

This part provides an example module, MmaIrc (which converts chat logs into a digest).

The process sets out how to plug it into a project, something of DI, the module config file, and Module.php.

1.3.4 Part 4: Adding a third module

Part 3 rehearses adding further modules.

1.3.5 Part 5: Adding ZfcUser module for login

Login authentication and security,courtesy of Evan Coury's exceptional talents and his ZfcUser. This module takes the hard work out of this crucial part of application development.

1.3.6 Part 6: Adding authorisation and ACL

Having logged in, the authorisation and ACL module defines who can access what, where, and how. Includes ACL compliant bread-crumb navigation trails.

1.3.7 Part 7: Adding a module with database access

This part not only employs database access techniques in a simple application, but in conjunction with ZfcUser, authorisation and ACL, how to limit access for users without privileges.

1.3.8 Part 8: Pushing results to new View mechanisms

This part takes a look at how the new "View Model" operates and communicates with Controllers.

1.4 Summary

Version 2.0 introduces a modular approach to Zend Framework developers. Although not ready for release, re-engineering to date enables development of foundation projects and modules.

This guide will hopefully assist developers become familiar with the revised framework.

2 Part 1: Installation

2.1 Set up a development server

Although developers should have no problem using other ZF compliant systems, this guide employs XAMPP.[1] It's an easy-to-install and consistent development server tool-set across differing operating systems.

XAMPP contains everything most developers will need to follow along.

Download it for Windows,[2] Linux[3] or Mac.[4] Download pages contain distribution contents. My preferred folder layout sets the version number, as in Figure 2.

Figure 2: XAMPP hierarchy

2.2 Foundation project installation

Fork or download ***foundation project core*** code from https://github.com/mike-a/foundation-core.

2.2.1 Download method

If downloading, copy foundation core code onto a local development server's /htdocs folder. Check regularly for updates to the core and revise files in the folder accordingly. Remember, though, that as use of different modules increases, so does the need to update.

2.2.2 Git method

Developers should avoid downloading then copying into folders, instead adopting the ZF approach to development.

Become familiar with github.com. Obtain a suitable Git client too. Then it's easy to keep projects up-to-date and synchronised with framework and module updates. Set up an account at GitHub[5] – it's free for open-source projects, including Zend Framework.

1 http://www.apachefriends.org/en/faq-xampp.html
2 http://www.apachefriends.org/en/xampp-windows.html
3 http://www.apachefriends.org/en/xampp-linux.html
4 http://www.apachefriends.org/en/xampp-macosx.html
5 https://github.com/signup/free

Git commands and operations are described throughout this guide (to boost familiarity).

2.2.3 Set up Git

First, go to Git for Windows,[6] Git for Mac,[7] or Git for Linux[8] for installation instructions.

Windows users will find it helpful to visit TortoiseGit download[9] and install, after which right-mouse clicks will provide integrated context menus.

2.2.4 Forking foundation project core (TortoiseGit, Windows)

Right-click in */htdocs* under the XAMPP installation folder. Then select Git Clone as shown in Figure 3.

Observe precise syntax for cloning by visiting https://github.com/mike-a/foundation-core (Figure 4). Note use of "Git Read-Only" to avoid need for a registered security key.[10]

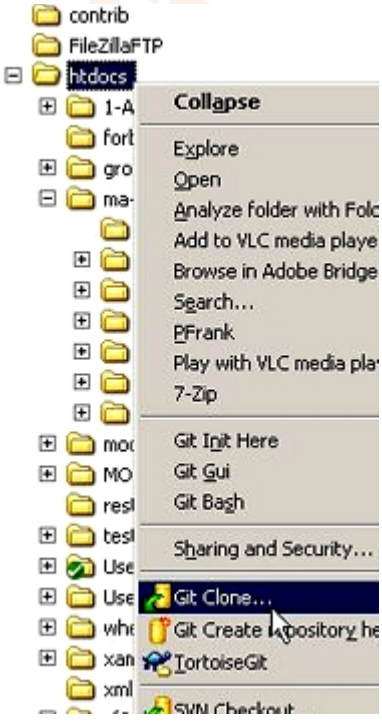

Figure 3: TortoiseGit clone project

Figure 4: Git clone syntax

[6] http://help.github.com/win-set-up-git/
[7] http://help.github.com/mac-set-up-git/
[8] http://help.github.com/linux-set-up-git/
[9] http://code.google.com/p/tortoisegit/
[10] http://jcreamerlive.com/2011/06/01/first-github-repository-tortoise/ provides more

Set the "recursive" check-box. Use the GitHub link to clone (Figure 5).[11]

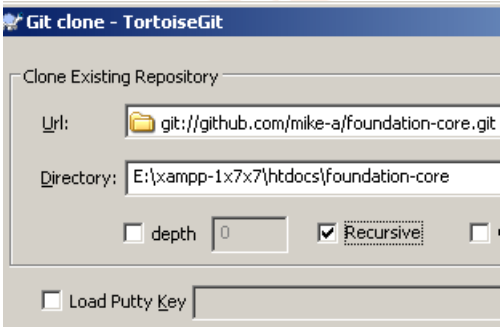

Figure 5: Cloning the core repository

2.2.5 Forking the foundation project core using Git Bash

Using Git Bash, follow clone instructions shown in Figure 6.

Change directory to */htdocs*, as shown in the first line, then *git clone...* Adding *myProject* at the end of the line will install into *myProject* folder. Leave blank to install into the */htdocs* folder (blank is default in this guide).

```
$ cd /E/xampp-1x7x7/htdocs

                    /E/xampp-1x7x7/htdocs
$ git clone -- recursive git://github.com/mike-a/foundation-core.git myProject
Cloning into 'myProject'...
remote: Counting objects:3514, done.
remote: Compressing objects: 100% (1848/1848), done.
remote: Total 3514 (delta 1596), reused 3514 (delta 1596)
Receiving objects: 100% (3514/3514), 3.16 MiB | 13 KiB/s, done.

                    /E/xampp-1x7x7/htdocs

$
```

Figure 6: Cloning from Bash console

2.2.6 Set Hosts file (Windows)

Windows users must also set up the "hosts" file: \WINDOWS \system32\drivers\etc\hosts. Add...

```
127.0.0.1  foundcore.localhost
```

which will match the server name set up in virtual host configuration, next.

2.2.7 Configuring a virtual host

Next, set up a virtual host to point to the project's public/ directory. My system has /apache/conf/extra/httpd-vhosts.conf file. Open it and add...

```
<VirtualHost *:80>
    DocumentRoot "E:/xampp-1x7x7/htdocs/foundation-
core/public"
    ServerName foundcore.localhost
    <Directory "E:/xampp-1x7x7/htdocs/foundation-core/public">
        DirectoryIndex index.php
        AllowOverride All
```

[11] More on '--recursive': http://codicesoftware.blogspot.com/2011/09/merge-recursive-strategy.html

```
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

Note *foundcore* in "ServerName foundcore.localhost". It matches the Windows *Hosts* file, as mentioned above. Start the Apache server via XAMPP (Figure 7).

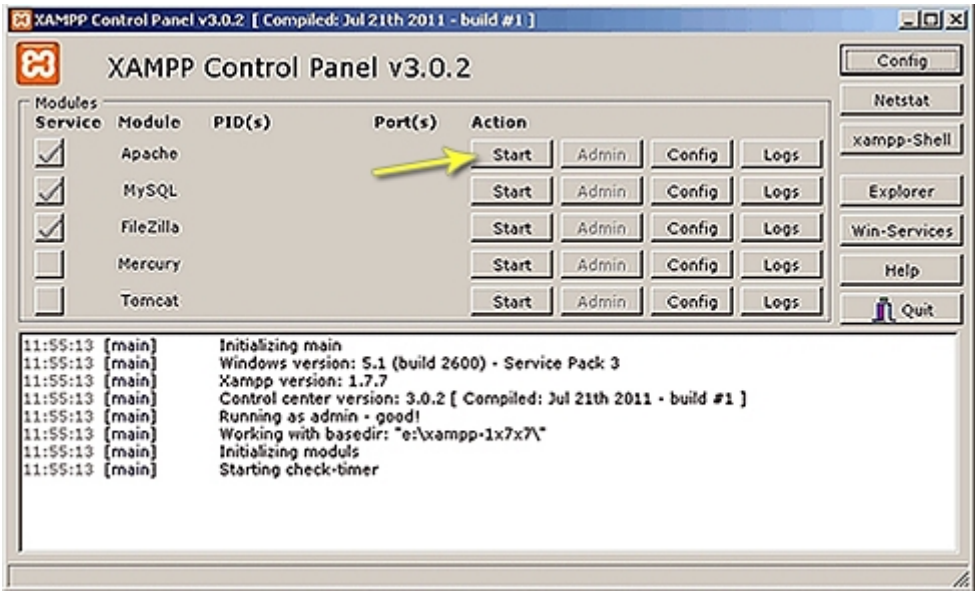

Figure 7: XAMPP control panel

Finally, type the server alias into a browser address bar (Figure 8, next page). That's the foundation core loaded and working!

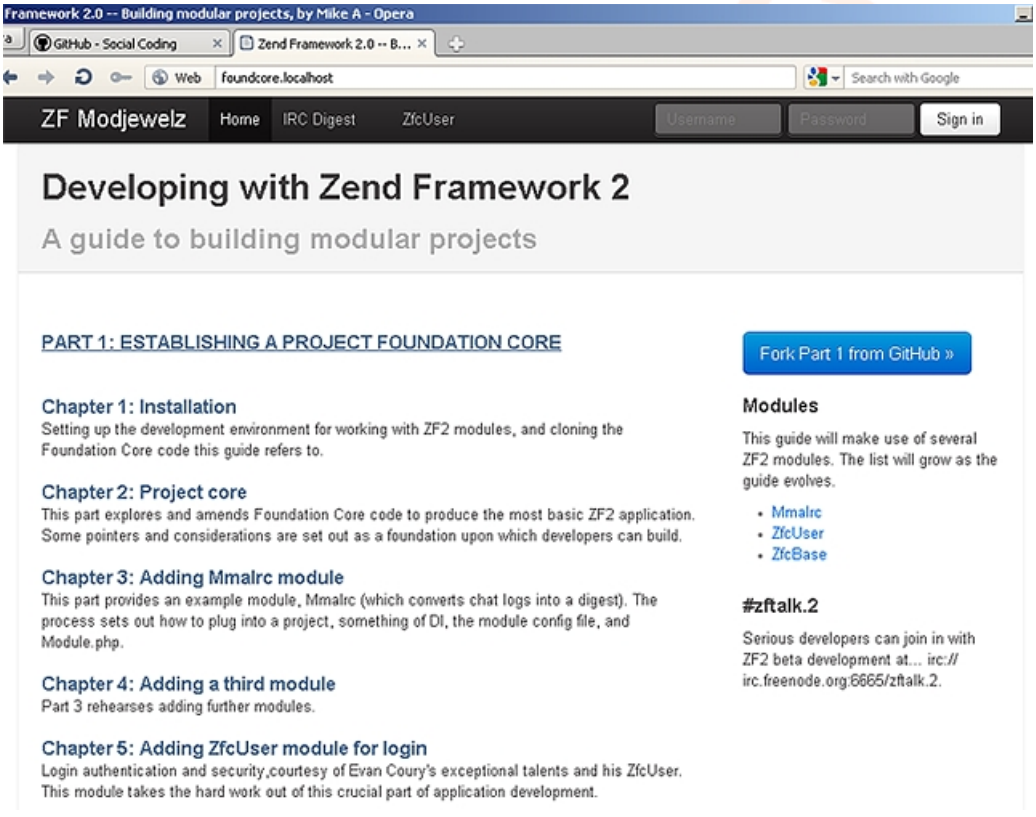

Figure 8: Foundation Core project loaded into browser

2.2.8 A development IDE (NetBeans)

More development tools shall follow as progress into the inner reaches of ZF2 proceeds. Install NetBeans development IDE now though.

Why NetBeans?

NetBeans was a slow development platform. With a combination of development improvements and faster technology, it's come of age. Like the Zend Framework initiative, it comes open-source and free. Given those factors, it seems pointless for high-calibre developers to segment and separate themselves by using different development tools. Conforming to a common development standard makes it possible to quickly and accurately solve coding issues, communicate, and write about projects: all important for modular development.

There's a video at http://netbeans.org/kb/docs/ide/overview-screencast.html.

This guide communicates examples and directives using NetBeans.

2.3 Summary

Users of this guide now have a fully working core to build a project foundation upon. En-route, a development environment was built and basic interaction with Git and GitHub took place.

The next chapter attempts to explain inner workings of the foundation core and the associated parts of Zend Framework, along with a few alterations so readers can play with cause and effect.

3 A FOUNDATION PROJECT CORE

This part reviews the most basic ZF2 application. Some pointers and considerations are set out as a foundation upon which developers can build. It can serve as the core of new projects, and to test new modules.

By using this project core, developers can satisfy themselves that every new module, and those obtained from public repositories, works alone as intended prior to plugging into larger productions.

3.1 Foundation core components

In segments, the foundation core consists of:

- Project resources;
- A bootstrap;
- A global environment coordinator;
- A global environment configuration;
- A module environment coordinator;
- A module environment configuration; and
- An MVC pattern module.

This essentially follows the Zend Framework skeleton application.[12] Here, the application will face some butchering for the sake of explaining various aspects of a core application. To enable that explanation let's wander off into some computer history for a few moments.

3.1.1 Huge old computer systems (AKA mainframes)

During the 1960s and '70s, before microcomputers came along, large organisations employed mainframes for data processing.

The ones I worked with, (ICL 1902/4/6) were lucky if they had 128k of memory. Programmers somehow had to use that processor storage across tape and disk drives, card readers, line printers – often dozens of peripherals, to produce a weekly wage run for perhaps 20,000 to 50,000 employees.

[12] https://github.com/zendframework/ZendSkeletonApplication

Figure 9: ICL 1900 series, early 1970s

There was a smart way to do it though, using COBOL (common business programming language). Starting a system required using switches to physically bootstrap the processor. Then a program could kick in with an "environment division". That's where the processor was told what it had to work with.

Jump forward a few years.

3.1.2 Three decades of wandering

Although using the first microprocessor, the Intel 4004, during the early 1970s, it wasn't until owning a Nascom DIY system, quickly followed by a PET and an Apple I, that my real microcomputer programming began. Then a computer store, and because of it, Pascal, Basic and horrid programming techniques.

The phrase "horrid programming" arises because it was inwardly focussing on individual systems and small networks without any of the old aspects of declaring, considering, respecting and protecting peripheral environments. It wasn't for at least another ten years that decent programmers employed file and record locking, system security, and threading on a regular basis.

Then some smart wotsit, Tim, made a browser for a thing named the Internet.

Jump forward a few more years.

3.1.3 Fast technology, clouds and go-anywhere computing

Peripheral technology, processor speeds, memory size and mobile technology improved phenomenally. By 2009, developers could take into account a vast array of combined processor-peripherals, including mobile computers and

devices accessing wireless Internet connectivity – plus a thing called "cloud computing".

We had come full circle in so far as the need for declaring, considering, respecting and protecting peripheral environments. There is more to go wrong, more security holes, and more places to waste inefficient code upon.

Crucially, project developers need to state from the outset what devices are in use. There are so many freely accessible devices: and readily interchangeable.

3.2 Zend Framework's "Environment division"

Note from Figure 10 that ZF2's present structure has at the beginning an environment division.

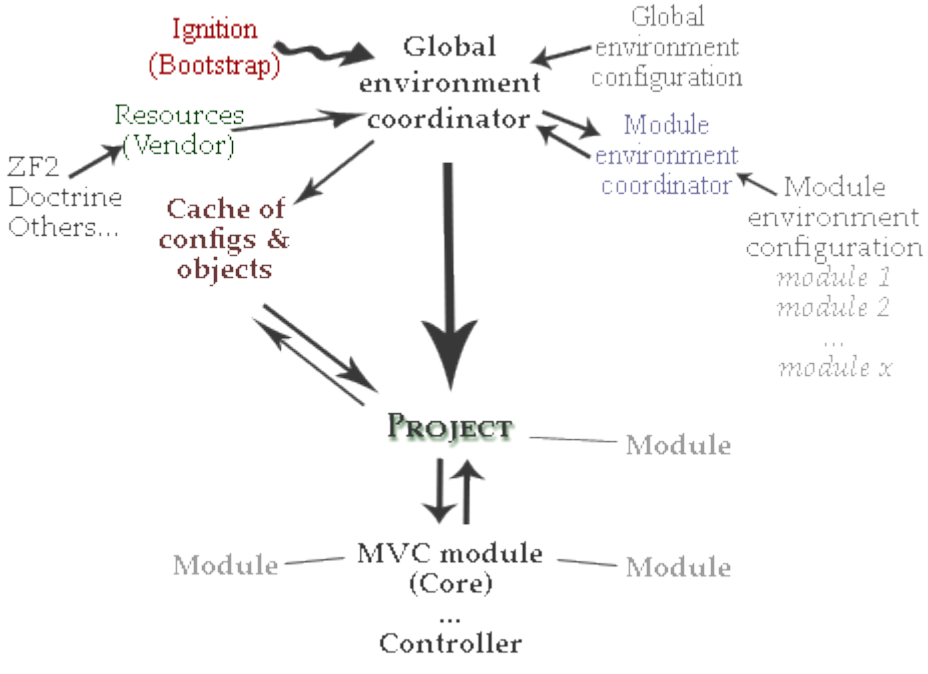

Figure 10: ZF2 foundation core, project flow

This all looks simple enough. Sadly, it's not the way ZF2 developers communicate.

ZF2's environment division consists of objects with names and functionality essentially meaningless to all but the inner sanctum of ZF experts. Hopefully, such complexity will evolve into a much simpler form so that professionals and newcomers alike can share a common basis of psychologically acceptable communication, explanation and tutorial. "Complex" systems of forty years ago

have evolved into something both granny and grandchild can use. Obfuscation and elitism kill markets. As with its predecessor, ZF2 components fail to convey meaning that's quick and simple. There's no need for that dark-age position.

Against this background, we should understand the devotion, passion, knowledge, and genius-level intelligence of the central group of ZF2 developers. Of course they will confine themselves to the labyrinth of server, system and languages where conflict of technologies represents the norm. And we should remember that this core group are at the forefront of communications and computing in human history: providing a backbone for "go anywhere" computing. Practical feedback from real-world users can help the effort – provide boffins with context. Otherwise, as with all products and services, even the best production will fail if people are turned off by it.

Meantime, this guide embarks upon a pseudo-translation of the core, beginning with what the environment division does.

3.3 Bootstrap

Breaking Figure 10 into its environment division provides a view (Figure 11) of what goes on before loading modules.

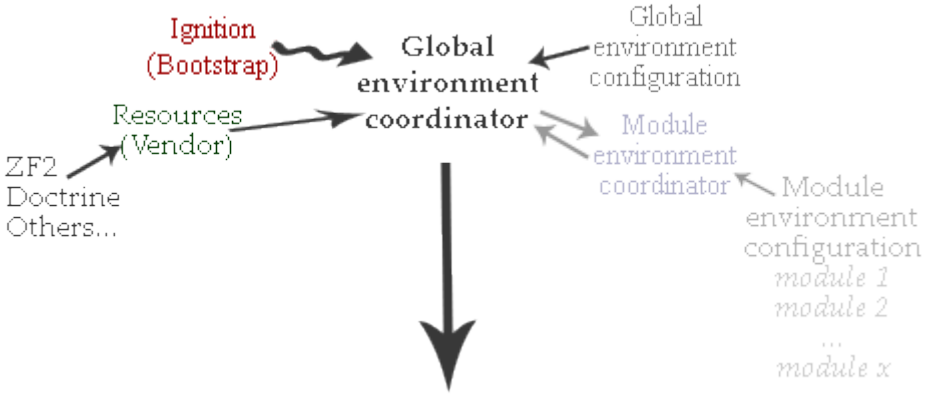

Figure 11: Foundation core environment division

Ignition uses [project root]/index.php. Remember, though, that's not where the project actually begins.

A car needs the right key to fit a lock that allows the key to turn and so fires the process to start an engine, right?

With Zend Framework, that key has come from an address bar in a web browser somewhere, a clicked website link, a direct request, a web crawler, or other server "arrival". A process has already taken place resulting from World Wide Web

networks,[13] using one of its many protocols: HTTP.[14] Having arrived at our server the key relies on the server's set-up to discover what to do next.

Most servers follow a standard configuration resulting in [mySite]/index.* being the default page. Of course, the key, otherwise known as a ***request***, can follow many forms but the first thing the server will examine depends upon server configuration.

It is therefore crucial to ensure servers are configured correctly!

A .htaccess file comes with the cloned foundation core, in the root folder with index.php. Note that it redirects everything to index.php...

[myApp]/.htaccess

```
SetEnv APPLICATION_ENV development

RewriteEngine On
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

Lots of approaches to .htaccess configuration exist.[15] A general rule applies though: **only ever use .htaccess when there's no direct access to server configuration**. If not running a dedicated server, Virtual Machine hosting being cheap and providing configuration access almost always surpasses shared hosting, thus avoiding the need for .htaccess.

After reading on a bit (not yet!), set server configuration in the httpd-vhosts.conf by swapping .htaccess settings into its <Directory> section as follows...

```
<VirtualHost *:80>
    DocumentRoot "E:/xampp-1x7x7/htdocs/foundation-
core/public"
    ServerName foundcore.localhost
    <Directory "E:/xampp-1x7x7/htdocs/foundation-core/public">
        Options Indexes FollowSymLinks MultiViews
        AllowOverride All
        Order allow,deny
        Allow from all
        RewriteEngine On
        RewriteCond %{REQUEST_FILENAME} -s [OR]
        RewriteCond %{REQUEST_FILENAME} -l [OR]
```

[13] Do not confuse with “the Internet”

[14] http://en.wikipedia.org/wiki/Internet#Protocols;
http://en.wikipedia.org/wiki/List_of_HTTP_header_fields

[15] http://www.javascriptkit.com/howto/htaccess.shtml
http://www.freewebmasterhelp.com/tutorials/htaccess

```
            RewriteCond %{REQUEST_FILENAME} -d
            RewriteRule ^.*$ - [NC,L]
            RewriteRule ^.*$ /index.php [NC,L]
            RewriteRule ^.php$ /index.php [NC,L]
        </Directory>
    </VirtualHost>
```

That's not quite the same as described in "Configuring a virtual host", covered in Part 1: Installation[16] – *and for a good reason arrived at in a moment*!

Start he server, open a browser, and enter foundcore.localhost/index.php, whereupon .htaccess will redirect it, and any other request to the project's ignition page, and invoke the foundation core, finally displaying its home page.

Now change /apache/conf/extra/httpd-vhosts.conf to include VirtualHost code as above. Delete or rename .htaccess. Stop and restart the server to invoke changes then again type foundcore.localhost/index.php. Why does it now report an exception?

This provides a clue as to how Zend Framework interacts with servers. Put a "/" at the end of the URI (foundcore.localhost/index.php/) and enter. Your browser should again display the home page. When .htaccess redirected your server it completed the URI with a "/", a fact not so obvious when employing redirects this way.

Avoiding this and similar HTTP request message[17] issues requires knowing from the outset what the server does and can do when an incoming request arrives – and becoming familiar with the various facets of HTTP.[18] Apart from avoiding time-consuming mistakes, developers knowledgeable about HTTP can reduce Internet noise, speed up page delivery, and benefit from resulting viewer retention. Avoiding .htaccess files improves efficiency too because a server only needs to call redirect settings once, at ignition time, instead of every time the Web accesses a project.

To re-emphasise – **only use the .htaccess file with shared servers**!

3.4 Ignition: index.php

A server receives an HTTP message, then redirects to an igniter: index.php.

..project/public/index.php

```
<?php
chdir(dirname(__DIR__));
require_once (getenv('ZF2_PATH') ?: 'vendor/ZendFramework/library')
   . '/Zend/Loader/AutoloaderFactory.php';
Zend\Loader\AutoloaderFactory::factory(array(
```

[16] http://httpd.apache.org/docs/1.3/misc/howto.html
http://httpd.apache.org/docs/2.2/urlmapping.html
[17] http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_message
[18] http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

```
        'Zend\Loader\StandardAutoloader' => array()
    ));

$appConfig = include 'config/application.config.php';

$listenerOptions  =
    new Zend\Module\Listener\ListenerOptions(
        $appConfig['module_listener_options']
        );
$defaultListeners =
    new Zend\Module\Listener\DefaultListenerAggregate($listenerOptions);
$defaultListeners
    ->getConfigListener()
    ->addConfigGlobPath('config/autoload/*.config.php');

$moduleManager =
    new Zend\Module\Manager($appConfig['modules']);
$moduleManager->events()->attachAggregate($defaultListeners);
$moduleManager->loadModules();

$bootstrap  =
    new Zend\Mvc\Bootstrap($defaultListeners
        ->getConfigListener()
        ->getMergedConfig()
        );
$application = new Zend\Mvc\Application;
$bootstrap->bootstrap($application);
$application->run()->send();
```

This file loads essential components from the framework, global project configuration, then runs it.

Take warning: *Zend Framework ignition components have not evolved nor matured sufficiently into cleanly-named, understandable, objects*. Accordingly, they are not capable of polite vivid description.

3.4.1 Configuration objects

To begin requires setting ZF2 library folder path and "loader" objects. Look at the Loader folder (Figure 12).[19]

Two objects load from this folder: AutoLoaderFactory and StandardAutoloader. What, though, are all those other loaders and stuff? Why does ignition of the project environment sit in this folder?

[19] http://packages.zendframework.com/docs/latest/manual/en/zend.loader.html

Developers should understand that as a framework, ZF2 is a toolbox with various compartments. Ever seen one of those plumbers with a disorganised tool-kit and a habit of displaying his (or her) butt-crack above the belt line? Meet the systems development equivalent! It gets worse – as readers will discover, there are autoloader objects sprinkled throughout framework folders. As a toolbox, the development team have embarked on a naming architecture designed to cause confusion from the outset. However, whilst it has become confused, confusing and oddly managed when compared to real-life business developments, Zend Framework surpasses other frameworks. Accordingly, from the outset, understand that *when using ZF2 objects we do so on a pick-and-mix basis, not on the basis of structured compartmentalisation*.

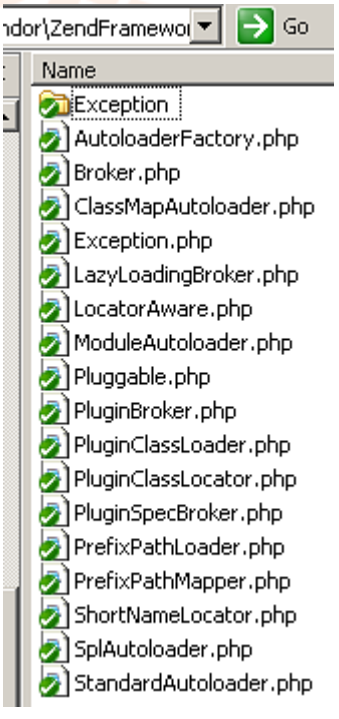

Figure 12: Loader Folder

Continuing read-through of index.php code, ignition moves on to configuration, explicitly calling AutoLoaderFactory and StandardAutoloader.

3.4.2 Load project configuration

In essence, project-wide configuration loads into $moduleManager, a Zend\Module\Manager object (Figure 13).

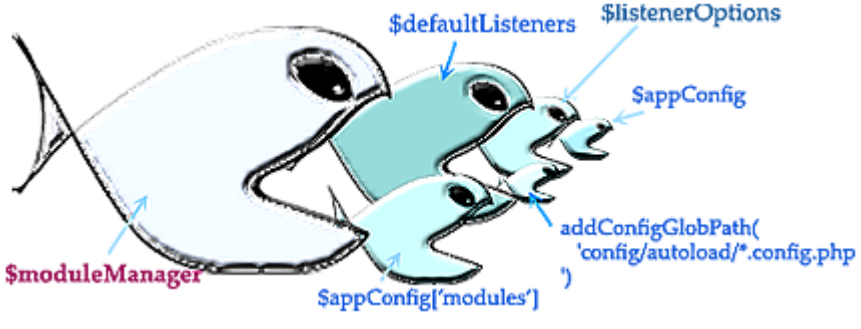

Figure 13: Project configuration

$modulemanager consumes five separate unintuitive areas of configuration. Project environment bootstrapping should not need this level of disorganised complexity!

First, project-wide configuration sets into variable $appconfig with...

```
$appConfig = include 'config/application.config.php';
```

Next, the file invokes a project-wide default listeners object by first consuming a listener options object with:

```
$listenerOptions  =
    new Zend\Module\Listener\ListenerOptions(
            $appConfig['module_listener_options']
            );
```

then

```
$defaultListeners =
    new Zend\Module\Listener\DefaultListenerAggregate(
             $listenerOptions
             );
```

and then adds a path for autoloading separate configuration files (should a developer care to use them in the project)...

```
$defaultListeners
    ->getConfigListener()
    ->addConfigGlobPath('config/autoload/*.config.php');
```

That's three steps to set what seems like a straightforward set of configuration parameters. Why not set these in one go? Because **two** objects then consume $defaultListeners before project booting, that's why.

To provide on-the-fly module loading by reference to the 'modules' array set in application.config.php...

```
$moduleManager =
    new Zend\Module\Manager($appConfig['modules']);
$moduleManager->events()->attachAggregate($defaultListeners);
$moduleManager->loadModules();
```

and then again in the bootstrap object...

```
$bootstrap   =
    new Zend\Mvc\Bootstrap($defaultListeners
            ->getConfigListener()
            ->getMergedConfig()
            );
```

Notice that this technique employs two concepts of modern programming, ***listeners*** and ***events*** (associated with the *observer* pattern[20]). By setting up objects with an environment capable of dynamic change, ZF2 has done away with the need for fixed configuration. **What at first appears a long-winded chain of ignition has instead provided a profound level of flexibility**.

Developers can set up ignition in various ways for a huge number of project environments – not only for websites. Doing so requires some understanding of listeners and events (and dependency injection, a subject turned to in a few moments). Of course, developers need only use ignition "templates" like the *foundation core* one here, but it's not a huge step to reach the same level of understanding of boffins, geeks and geniuses within the worldwide ZF2 development community. Apart from its application to ZF2 projects, knowing about listeners and events enables a quantum leap into power-development for the future – the type of approach essential for coping with interchangeable devices present and to come.

3.4.3 Ignite

With configuration objects instantiated, configuration parameters loaded, and the whole lot set into a bootstrap object, it's time to ignite the application by instantiating a new Application object.

```
$application = new Zend\Mvc\Application;
$bootstrap->bootstrap($application);
$application->run()->send();
```

Open Zend Framework's /Mvc/Application file. Observe, the *Application object* sets an application's principle objects: events; locator; request; response and router. *Understanding the Application object and its associated processes provides a foundation for ZF2 MVC expertise*.

$bootstrap->bootstrap($application) then simply sets up the locator, router and events before chaining application->run() (NetBeans right-click, Figure 14), and the send method of Zend\Http\PhpEnvironment (instantiated by the use statement at the top of Zend\Mvc\Application, Figure 15).

[20] http://en.wikipedia.org/wiki/Observer_pattern
http://en.wikipedia.org/wiki/DOM_events
http://stackoverflow.com/questions/615749/does-this-match-your-definition-of-a-listener-object

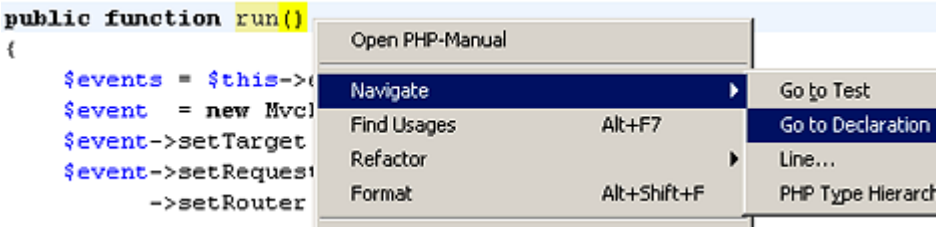

Figure 14: NetBeans Go to Declaration

```
<?php

namespace Zend\Mvc;

use ArrayObject,
    Zend\Di\Exception\ClassNotFoundException,
    Zend\Di\Locator,
    Zend\EventManager\EventCollection,
    Zend\EventManager\EventManager,
    Zend\Http\Header\Cookie,
    Zend\Http\PhpEnvironment\Request as PhpHttpRequest,
    Zend\Http\PhpEnvironment\Response as PhpHttpResponse,
    Zend\Uri\Http as HttpUri,
    Zend\Stdlib\Dispatchable,
    Zend\Stdlib\IsAssocArray,
    Zend\Stdlib\Parameters,
    Zend\Stdlib\RequestDescription as Request,
    Zend\Stdlib\ResponseDescription as Response;
```

Figure 15: Zend/Mvc/Application.php use statement

3.4.4 Why "Ignite"?

Why refer to "ignite" instead of "bootstrap" when a bootstrap object was instantiated? *Because bootstrap processing has not yet finished*!

To clarify, taking ZF2's use of the term "bootstrap" here, when applied to aviation, would imply that once engines have started an aircraft will immediately column-rotate and take off. Ignition then air-born, immediately followed by passengers' expletive-ridden exclamations expressing profound concern. Disastrous, of course – and an extreme discouragement from flying. ZF1 unnecessarily discouraged many developers due to similar ambiguous architectural and naming conventions.

To overcome this daft position until the framework becomes more "obvious", become familiar with what's going on during ignition: ***it's not difficult***.

3.4.5 Dependency injection

Whilst on the subject of becoming familiar with the framework, ZF2 developers largely based environment configuration upon dependency injection ("DI").

It will quickly become apparent to anyone using ZF2 that familiarisation with DI is essential. It may take a day or two to familiarise with ignition and bootstrap procedures, but that effort will come to waste without at least a fundamental knowledge of DI. So what is it?

In brief, DI provides a way to turn input to a DI container into an object. Put another way, it allows on-the-fly creation of objects or change in an object's state.

Why use DI?

Of course, every programmer knows of constants, global variables and configuration arrays – static settings. Programmers also know how unwieldy large static setting files and containers become, and how difficult they are to adjust without restarting an application. What, though, if a mechanism could put settings into objects? That would allow setting and getting an object's property values as required, inheriting and extending the object, and most other things a class does or allows. Thus environment settings become dynamic. We can then set values *and methods* into a dynamic settings object, pull that object and its settings into another object containing settings pertinent to another part of the project, and so on until building the whole project environment.

Look again at index.php.

```
$appConfig = include 'config/application.config.php';
$listenerOptions  =
    new Zend\Module\Listener\ListenerOptions(
            $appConfig['module_listener_options']
            );
$defaultListeners =
    new Zend\Module\Listener\DefaultListenerAggregate(
            $listenerOptions
            );
$defaultListeners
    ->getConfigListener()
    ->addConfigGlobPath('config/autoload/*.config.php');
$moduleManager =
    new Zend\Module\Manager($appConfig['modules']);
$moduleManager->events()->attachAggregate($defaultListeners);
$moduleManager->loadModules();

$bootstrap   =
    new Zend\Mvc\Bootstrap($defaultListeners
```

```
                ->getConfigListener()
                ->getMergedConfig()
                );

$application = new Zend\Mvc\Application;
$bootstrap->bootstrap($application);
```

At ignition stage, setting files are gathered and converted into objects which other objects in turn consume. This process results in a $bootstrap object, already containing application configuration, then consuming the $application object. Now there's a project-wide environment container offering settings and methods for general consumption *and change*. No complex static configuration files or arrays – it was all done on-the-fly using DI. That's powerful – and worth the braincell-twisting effort of DI learning resulting in that "Eureka!" moment.

Those input values required for DI containers to regurgitate as objects, mentioned above, appear in two places: *in the configuration file* and *in the setter method of the object instantiating the configuration*. DI does the rest.

Recall that ZF2 was primarily developed as a modular framework and MVC system. Modules arrive as independent devices with their own features, set-up and operation. With DI, project developers can plug in modules knowing that ZF2 DI will take care of the rest – dynamically.

By using DI containers to turn parameters into objects, developers can now instigate unit testing.[21] In turn, unit testing opens up an opportunity for TDD.[22] These extensive subjects are for much later in this guide.

In summary, ZF2 development rotates around DI for environment handling. Knowing the basics of DI and understanding modern environment handling forms the basis for ZF2's top-quality approach to project development.

Some good DI resources...

- http://www.slideshare.net/fabpot/dependency-injection-in-php-5354
- http://mwop.net/blog/260-Dependency-Injection-An-analogy
- http://www.phpconference.co.uk/talk/advanced-oo-patterns
- http://miller.limethinking.co.uk/2011/05/19/when-dependency-injection-goes-wrong/
- http://en.wikipedia.org/wiki/Dependency_injection

[21] http://en.wikipedia.org/wiki/Unit_testing
http://devzone.zend.com/1115/an-introduction-to-the-art-of-unit-testing-in-php/

[22] http://en.wikipedia.org/wiki/Test-driven_development
http://stackoverflow.com/questions/46276/test-driven-development-in-php
http://net.tutsplus.com/tutorials/php/the-newbies-guide-to-test-driven-development/

- http://blog.astrumfutura.com/2011/10/zend-framework-2-0-dependency-injection-part-1/
- http://jamesshore.com/Blog/Dependency-Injection-Demystified.html
- http://www.martinfowler.com/articles/injection.html
- http://www.essex.ac.uk/ccfea/research/WorkingPapers/2012/CCFEA-WP056-12.pdf
- http://msdn.microsoft.com/en-us/magazine/cc163739.aspx

3.5 Ignition->Bootstrap: [myApp]/module/Application/Module.php

Zend Framework began life as a component library and MVC pattern[23] application framework. ZF2 being modular, many ZF1 library components may become modules, each available online instead of in an enormous framework library.

There are many advantages to this approach.

Projects would consist of less unused, redundant, library files. Developers will have an improved understanding of the framework because choosing to include building blocks requires knowledge of how those blocks work together. Developer communities can improve modules, subclass them and combine module sets into larger modules with enhanced functionality.

With all this power comes potential complexity if developers do not understand how to employ modules, or fail to consider interaction and conflict between them.

To circumvent potential issues and provide flexibility to developers, each module comes with its own environment setting class: Module.php.

A project needs to add module-specific environments to the global configuration, perhaps for many modules. All modules must contain a Module class. It merges via DI into one global Module environment setting.

Foundation core has a single module so far. "Application" module sits under /module folder. All other modules will go here too.

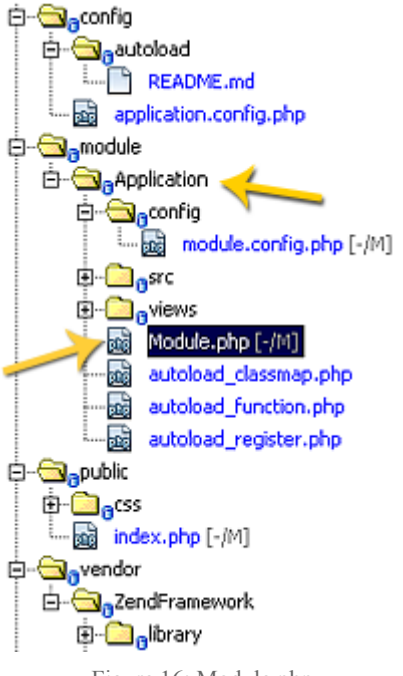

Figure 16: Module.php

[23] http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Application module serves as a default, or home, module. Accordingly, it needs to serve default Model, View and Controller functionality. Being a default module, it should also serve basic features like exception handling. Further modules may have different configurations (covered in later chapters), but a default module must serve a View otherwise the project would have no output.

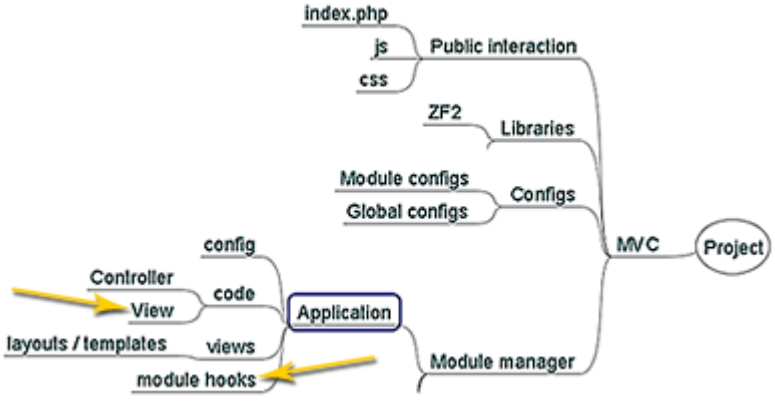

Figure 17: Application Module.php hierarchy and environment

Figure 17 displays Application module's relationship in project hierarchy, its need to employ View object (there being no other), and where it hooks into the project (by using Module.php, within Zend\Module\Manager chain).

So far: ignition begins an MVC application and ZF2 being modular, each module's environment also needs bootstrapping.

Observe code for Module.php...

./module/Application/Module.php

```
<?php

namespace Application;

use Zend\Module\Manager,
    Zend\EventManager\StaticEventManager,
    Zend\Module\Consumer\AutoloaderProvider;

class Module implements AutoloaderProvider
{
    protected $view;
    protected $viewListener;

    public function init(Manager $moduleManager)
    {
        $events = StaticEventManager::getInstance();
        $events->attach('bootstrap', 'bootstrap', array(
            $this,
            'initializeView'),
```

```
        100);
    }

    public function getAutoloaderConfig()
    {
        return array(
            'Zend\Loader\ClassMapAutoloader' => array(
                __DIR__ . '/autoload_classmap.php',
            ),
            'Zend\Loader\StandardAutoloader' => array(
                'namespaces' => array(
                    __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
                ),
            ),
        );
    }

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

    public function initializeView($e)
    {
        $app          = $e->getParam('application');
        $locator      = $app->getLocator();
        $config       = $e->getParam('config');
        $view         = $this->getView($app);
        $viewListener = $this->getViewListener($view, $config);
        $app->events()->attachAggregate($viewListener);
        $events       = StaticEventManager::getInstance();
        $viewListener->registerStaticListeners($events, $locator);
    }

    protected function getViewListener($view, $config)
    {
        if ($this->viewListener instanceof View\Listener) {
            return $this->viewListener;
        }

        $viewListener       = new View\Listener($view, $config->layout);
        $viewListener->setDisplayExceptionsFlag($config->display_exceptions);

        $this->viewListener = $viewListener;
        return $viewListener;
```

```
    }

    protected function getView($app)
    {
        if ($this->view) {
            return $this->view;
        }

        $di   = $app->getLocator();
        $view = $di->get('view');
        $url  = $view->plugin('url');
        $url->setRouter($app->getRouter());
        $view->plugin('doctype')->setDoctype('HTML5');
        $view->plugin('headTitle')->setSeparator(' - ')
                                  ->setAutoEscape(false)
                                  ->append('Zend Framework 2.0 -- EdpUser Module');
        $view->plugin('headLink')->appendStylesheet('/css/bootstrap.min.css');
        $view->plugin('headScript')->prependFile(
            'http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js'
            );
        $view->plugin('headScript')->appendFile('/js/bootstrap-alerts.js');
        $this->view = $view;
        return $view;
    }
}
```

A brief clinical assessment indicates two predominant features of this module environment set-up.

First, it begins with namespace Application. ZF2 employs PHP 5.3+ namespace convention. Accordingly, ALL ZF2 objects begin with a namespace, without which an exception will occur. This forces coherent architecture and project hierarchy thus making projects easier to understand, especially large projects.

Second, this module primarily instigates View. Logically, it's obvious for a module at the front of a whole project to set the primary environment for output. Some modules may not need a View object, some may need to alter the view object, most modules will need to establish other environmental constructs *essential only to the particular module*.

As a rule, ***if an environment configuration concerns whole-module behaviour, it belongs in Module.php and the module's associated configuration files: if not then it doesn't belong in Module.php***.

Wait a moment though: from a systems analyst's perspective, why does the Application module's Module.php contain View configuration? Surely that belongs elsewhere?

It does! At the time of writing, though, ZF2 View remained in development. In that circumstance it was easier to establish View in this module's Module.php. There's a warning though. Developers may become accustomed to using early examples of basic projects as templates, thus forming bad habits. Alternative approaches will appear later in this guide. Fortunately, adopting View setting here provides an opportunity for readers to observe one use of Module.php in an environment bootstrapping context.

For now, follow Application/Module object...

3.5.1 Consumed objects

See from the use statement that Module.php consumes

- Module\Manager,
- EventManager\StaticEventManager, and
- Module\Consumer\AutoloaderProvider

3.5.2 Extending the project environment via Module.php

Notice Module as a class within Application namespace. In ZF2, file names follow object name and PHP namespace. This is mandatory, not advisory. Project hierarchy immediately understands that Application module and its configuring Module object resides at [myProject]/module/Application/Module.php.

Module object implements AutoloaderProvider interface (go see with NetBeans right-click on AutoloaderProvider > Navigate > Go to Declaration). Interface method signatures must occur in implementing objects,[24] meaning that Module must contain a getAutoloaderConfig() method (see below, Establishing View).

Module object initialises by reference to Module/Manager's $moduleManager object, instantiated in index.php. Take a look at what that does at this early stage by outputting with print_r($moduleManager) to observe EventManager container in a browser.

```
public function init(Manager $moduleManager)
{
 echo "<pre>";
 print_r($moduleManager);
 echo "</pre>";

  $events = StaticEventManager::getInstance();
  $events->attach('bootstrap', 'bootstrap', array(
      $this,
      'initializeView'),
      100);
}
```

[24] http://php.net/manual/en/language.oop5.interfaces.php

This module will need a View. First, though, the project environment must know a View event exists. Setting an instance of StaticEventManager object to initialise by attaching the View to an array of events accomplishes this -- in this instance within a bootstrap event with a key of 'bootstrap'.

Now there's a View and a ViewListener registered in the project's Bootstrap container. Take another look at EventManager with print_r($events); at the end of the init(Manager $moduleManager) method.

The module now has access to events!

It's so easy to extend a project's environment this way. Take an environment container, like the bootstrap one above, it's key=>value pair ('bootstrap', 'bootstrap'), then set a callback ($this, 'initializeView') into it.

Consider what an event will do before placing it. Events, and associated listeners, work at a project level. Consider how the module will work upon the event rather than dump huge numbers of events and listeners into the global environment – don't become a litterbug.

3.5.3 Configuring Module.php (getAutoloaderConfig)

AutoloaderProvider interface causes use of getAutoloaderConfig().

Use getAutoloaderConfig() to return an array of Module's configuration paths. Think about it – the method has mandatory occurrence because a project will always need to know the module's structural hierarchy. Perhaps not an important issue in this simple module, but this area of operation has far more importance to developers wanting to plug module's into modules to create module groups.

Next comes the getConfig() method.

At first it doesn't seem to belong to anything. Until commenting out, that is. Try it and see the result in a browser! Exception – an important one because it exposes Module object as part of ignition procedure when index.php processes.

Everything done in Module.php, for all modules, happens as the result of the penultimate line in index.php: $bootstrap->bootstrap($application).

The getConfig() method simply sets a path to module.config.php for every module's dependency injection processing, in this case at /config/module.config.php.

Uncomment getConfig() to restore. It's possible that the server may need restarting (and in NetBeans right-click on Module.php and commit, as shown in Figure 18).

3.5.4 Establishing View

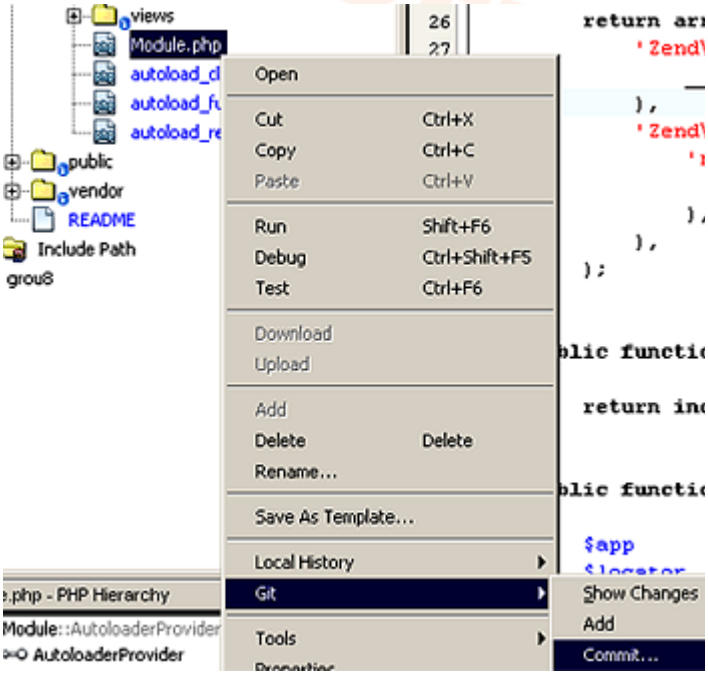

Figure 18: Git commit