



reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Copyright &copy; 2008 PÃ¡draic Brady

Go ahead and reopen <http://zfblog/> in a browser. You'll quickly notice that with little effort, it looks oddly pretty and the main content and right column are aligned exactly even.

In the HTML we did some very simple things. We have the standard 3 block solution - Header, Content, Footer. Both are contained in div's with assigned "span" width. Content is further divided into two columns with "span" widths - the total of which add to 1 less than the total of 24 spans. The reason for the missing span is simply to allow room for further styling which in varying browsers can have knock on effects - typically using the full span width across multiple column blocks ends up with a browser breaching the fixed 24 width and leaving a column dropped below the others.

Let's add a few modifications to allow for a more realistic appearance of blog entries with a title and other assembled summary data.

Posted by PÃ¡draic Brady      Lorem Ipsum      Excepteur Sint  
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.      Excepteur Sint  
Tags: Lorem, Ipsum, Excepteur      Excepteur Sint  
Posted by PÃ¡draic Brady      Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.      Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.      Copyright &copy; 2008 PÃ¡draic Brady

What we now have is pretty basic - but even in black and white it looks clean and quite blog-like. If you disable CSS for your browser, well, it still looks quite fine which is always a positive thing.

Step 3: Splitting persistent markup into a Zend\_Layout template Traditionally, PHP templating has rarely addressed a common problem we have using templates. Somewhere in the design above (you can spot this easily) are elements common to almost every future page of our blog application. A stock solution is often to push these elements into separate templates, and include them as needed. This poses the problem of widespread references to these templates. What happens if we decide they only apply to half our site now, and need to replace them for 50% of all existing page specific templates? I call it PAIN. Others call it MIND-NUMBING.

A simple tactic to avoid these symptoms is to segregate common elements in one single location, and make sure individual templates remain blissfully unaware of it's existence (then those 100 templates have nothing to edit if the central common template is replaced for 50 of them!).

In the Zend Framework this tactic was initially proposed over a year ago when attention was slowly turning towards the weak Zend\_View side of the MVC equation. I proposed Zend\_View Enhanced which include, among a lot of things, a Zend\_View based solution to the common layout problem. Ralph Schindler proposed around the same time a Zend\_Layout component for similar purposes. Eventually I lost on that score, and Zend\_Layout was adopted. Such is life - at least the rest of Zend\_View Enhanced with some modification was also adopted, making for a powerful templating solution taking full advantage of the PHP language (and not a limited subset obscured behind a secondary tag language).

Examining our design above we can spot the common bits: header, footer and the righthand column. Zend\_Layout uses a separate template - the Zend Framework manual is extremely obscure about how to actually get the Layout part

working (hopefully fixed soon) but it's very simple.

Create a new file in `/application/views/layouts` called `common.phtml`. What we'll do is copy in the above `index.phtml` template but delete the sections that are likely to change with different page views - namely the entries.

### Lorem Ipsum

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Copyright &copy; 2008 P&Auml;draic Brady

You'll notice I replaced the fake entries with a reference to a `layout()` function. Function calls on the template object instance generally are either called from the template object (an instance of `Zend_View`) or are references to a `View Helper` (a small class encapsulating reusable presentation logic - we'll hear a lot more about these later). In this case, it refers to the `Zend_View_Helper_Layout` class. This class instance has a public variable `$content` which contains all content rendered from all preceding templates (bear in mind `Layouts` are rendered last - after `Action` specific templates).

The process is simple. You visit a `Controller` and `Action`. That `Action`'s template is rendered, and saved to the `Layout View Helper`. The `Layout` is rendered last, and the pre-rendered content inserted as the `Layout` template defines using a call to `$this->layout()->content`.

With our `Layout` template prepared, let's remove all those common items from our `Action` specific template at `/application/views/scripts/index/index.phtml`:

Excepteur SintPosted by P&Auml;draic BradyLorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.Tags: Lorem, Ipsum, ExcepteurExcepteur SintPosted by P&Auml;draic BradyLorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.Tags: Lorem, Ipsum, Excepteur

Now we're getting somewhere interesting. The revised template, only contains the content unique to that `Controller/Action` combination and it's output will be inserted into the common `Layout` we created - after we setup `Zend_Layout`!

**Step 4: Setting up `Zend_Layout` for operation**Getting `Zend_Layout` working is again quite simple. We just need to make a few additions to our `Bootstrap` file. Specifically to the `setupView()` static method.

Posted by P&Auml;draic Brady in [PHP General](#), [PHP Security](#), [Zend Framework](#) at 11:51

Monday, April 28, 2008

### An Example Zend Framework Blog Application - Part 3: A Simple Hello World Tutorial

It's almost obligatory when introducing a new programming topic, that the author present the simplest possible example. Usually this means getting a programming language or framework to print "Hello World" to the screen. I'm going to be no different. So much for originality...

Previously: Part 2: The MVC Application Architecture

**Step 1: Creating A New Application** Before we jump into programming with the Zend Framework there are a few setup tasks. We need to install the Zend Framework, get a virtual domain running, and have a basic collection of directories and empty files created to hold our source code.

You'll need to download the Zend Framework 1.5 (latest minor release) from <http://framework.zend.com> and put it somewhere on your include path. I try to minimise the number of include paths I end up having so I sometimes pick very odd places (like the PEAR directory which is usually already in the include\_path) or another central library location like PEAR where I might have a dozen libraries all congregated.

A common solution found on blogs and articles is to put the framework into a "library" directory within your application directory tree. Ordinarily I don't recommend this unless your application needs a specific version of the Zend Framework - centralising a Zend Framework location for multiple applications to access can make maintenance a bit easier.

I've settled on using a virtual host mainly because it's easy to setup, and it helps avoid some of those annoying base path issues you get in your HTML when using mod\_rewrite pretty urls - especially in development where, otherwise, the localhost document root becomes a mass of sub-directories upon sub-directories that end up causing base href issues so easily. Usually I open up the virtual-hosts.conf file from Apache's conf directory and add something like:

```
NameVirtualHost :80
```

```
DocumentRoot "/path/to/htdocs"  
ServerName localhost
```

```
ServerName zfblog  
DocumentRoot "/path/to/project/trunk/public"
```

This should result in all HTTP queries for <http://zfblog/> reaching the selected document root (which is the public directory for the project containing index.php, located in our subversion's working copy "trunk" directory). We'll create this directory structure a little later. We usually also need to reference localhost to ensure it's recognised as a host correctly - otherwise any localhost addresses you already have will suddenly divert to zfblog's document root mysteriously .

For those of you on Windows XP or Vista, you also need to add the new virtual domain to your hosts file (usually at C:\WINNT\system32\drivers\etc\hosts). This is tricky in Vista - you will probably need to alter the file's permissions using an Administrator account to grant the local user additional Modify privileges. Make sure to reverse the privileges after! Edit it to look this:

```
# Copyright (c) 1993-1999 Microsoft Corp.  
#  
# This is a sample HOSTS file used by Microsoft TCP/IP for Windows.  
  
127.0.0.1 localhost  
127.0.0.1 zfblog
```

Restart your browser and with a little luck the browser will at least give us a directory listing (once you add the document

root "public" directory as shown below).

The least exciting part is the creation of a tree of directories to contain our source code. Wil Sinclair over at Zend is working on a Zend\_Build component and this will one day hopefully eliminate too much of the repetition in creating these directories and default files. Until then let the monotony ensue (or keep a skeleton setup handy for copy and pasting).

Here's a snapshot of what the Hello World example looks like in the directory view from Eclipse:

Copy this and create the same directory structure in a new directory called something like "zfblog". Putting it all in "trunk" is optional - I've using subversion here which is why that directory exists. If you haven't used Subversion before I highly recommend it - it's simple to learn and use.

Let's examine each of these directories briefly.

The "application" directory is where we place all the components of an application implementing the Model-View-Controller. Inside "application", "controllers", "models" and "views" represent respective locations of controller, model and view source code and templates. Inside "views" there is a subdivision between "filters", "helpers", "layouts" and "scripts". For now, remember we put all templates rendered by the View inside "scripts". In the screenshot, we see an index directory which will contain an "index.phtml" file, which is the view template for indexAction of an IndexController class. The others we'll meet later.

The "modules" directory is also for controllers, models, and views, but in this case they are categorised into multiple divisions of an application. If you think about it, an application can be broken into several parts, for example, the main application and an administration part. The administration section could be partitioned into the Admin Module so it's not intermixed with the main application.

A Bootstrap.php file exists in the "application" directory which represents a class called "Bootstrap". Its purpose is to initialise the Zend Framework, adjust environment settings (for example, timezone and error\_reporting level), and otherwise make application specific tweaks and additions before a HTTP request is processed. Most tutorials take an alternative view and put Bootstrap code into index.php. I strongly suggest you avoid this and use an actual class to organise the Bootstrap code - it makes it a lot easier to read if nothing else!

The "config" directory simply holds the configuration data for the application, for example, database connection details.

The "library" directory can hold a copy of the Zend Framework. Generally I avoid this since it's fairly easy to just add the Zend Framework to the PHP include\_path, but it's still very useful if you want to change Zend Framework versions in use more flexibly. It's also simple to manage if you use it to add an svn::external reference to the Zend Framework subversion repository.

The "public" directory holds all public files accessible by a request to the web server. This includes an index.php file, which handles all application requests by calling on the Bootstrap class, as well as any CSS, HTML, Javascript files, etc.

A note on deployment practice: generally when deploying you'd want to move the "public" directory to your web server to be internet accessible, but keep the non-public directories somewhere below the web root. Since the public index.php ends up only making a simple reference to the Bootstrap file served by including it, this means most of the application files will not be accessible by internet users. It also means that the index.php file will end up deciding where the Bootstrap file is located - so it's the one public file that always needs to be edited by hand for deployment to change that location as needed.

The "tests" directory usually holds any application specific unit, integration and acceptance tests. As I noted previously I'm deliberately taking a test-light approach to stay on topic (I have more than one testing/development article on Zend Devzone if interested in applying TDD or BDD). Main tests I expect to have are for application specific classes: any Model logic for example.

At the end of this step you should have the directory structure in place containing some empty files as follows (create them now, and we'll fill them in later):

```
/application/Bootstrap.php
/application/controller/IndexController.php
```

/application/views/scripts/index/index.phtml  
/public/index.php  
/public/.htaccess

Step 2: Implementing Our Bootstrap File Bootstrapping is when we setup the initial environment, configuration and Zend Framework initialisation for our application. It's not a difficult file to write, but it can grow ever larger and complex as your application gains features. To manage this I strongly suggest implementing it as a class with bitesize methods. Breaking it up does wonders for your sanity. For our Hello World skeleton application, /application/Bootstrap.php contains:

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 17:42

Thursday, April 24, 2008

### **Subversion for the Example Zend Framework Blog Tutorial Series**

You should all see a few commits commencing at the weekend. Probably all in one go since I largely have a standard skeleton I use already.

<http://svn.astrumfutura.org/zfblog/>

As mentioned previously, this is for live development so be patient for the next installments of the tutorial series if you update from the repository one day and find a bunch of unexplainable code .

Posted by Pádraic Brady in [PHP General](#), [PHP Security](#), [Zend Framework](#) at 19:40

Wednesday, April 23, 2008

### An Example Zend Framework Blog Application - Part 2: The MVC Application Architecture

If you are in line waiting for the source code appear, it will in the next entry .

Previously: Part 1: Introductory Planning

After speaking with any number of users about getting started with a framework, I find many do not have an advanced understanding of the corner stone of a current day web application framework: the Model-View-Controller Design Pattern. So let's get over that hill right now, and before we start looking at PHP!

As a bit of background, consider the traditional and still popular approach to writing an application in PHP. Usually you take an approach called the Page Controller. Each HTML page of your application may have it's own dedicated PHP file - often it ends up as many HTML pages per PHP file, but only if those pages are sufficiently similar (e.g. forms and form processing are typical) that the relationship is formed out of the need to re-use source code in the same file. Frequently, these pages will all share a common collection of functions, classes and constants. All pages may need Smarty, for example, or a database connection, or maybe even a standard data collection for Users, ACL, etc.

The problem is that in this scenario it becomes very difficult (but not impossible) to manage growth and change. Every change, and every new feature, requires new code. Where you end up putting that new code becomes a huge concern. Maybe you need to add several changes to several pieces of application logic - but then find the logic is scattered across multiple PHP files. Perhaps you discover your reusable database connection now has SQL statements in 12 files, which now need to refer to a new table field. You can imagine the profusion of small changes in multiple files finally exploding exponentially until you wind up in a situation where the cost of change far exceeds the benefits. This is the point at which many a project has simply stagnated despite the enthusiasm of its developers - and to be clear, I have fallen into that trap previously . Been there, done that, discovered the delight of Object Oriented Programming and reformed my practices!

The Model-View-ControllerThe Model-View-Controller Pattern (or MVC as it's usually abbreviated) is a general solution to the question of how to separate responsibilities in an application in a highly structured manner. The pattern's name mentions all three separations: Model, View and Controller. Although MVC may seem to be one of those esoteric concepts in programming, it's actually quite a simple concept. You pick some nugget of functionality, determine it's purpose, and assign to one of the three separations, and then to a specific class. Once everything is split correctly, you end up with small pieces which are reusable, centralised, accessible, and fit together into building blocks with an abstract API - working now with abstracted APIs makes incremental changes extremely simple (if done correctly of course ). With everything tidily organised into objects with specific responsibilities the cost of change is normally reduced significantly (which is really the whole point - we want change to be cheap, easy and horror free).

Obviously I'm not covering the entire field of Object Oriented Programming here. But hopefully the message is sufficiently clear. Also the OO nature of the Zend Framework is largely why it contains so many components. We don't simply have Zend\_Controller - we have Zend\_Controller\_Action, Zend\_Controller\_Router, Zend\_Controller\_Front, etc. Each specific role or responsibility is covered by it's own class. This certainly results in a profusion of focused classes to such an extent it can become difficult to see how all the pieces work - but honestly you only need the outer abstracted API and can ignore the rest unless you really really want to customise something.

To be clear, the MVC is common as dirt. It is widely used in the Zend Framework, Solar, Symfony, Ruby on Rails, merb, Django, Spring, and countless other frameworks. It is an unescapable concept when adopting a framework for web applications in almost any language.

The ModelThe Model is responsible for maintaining state between HTTP requests in a PHP web application. Any data which must be preserved between HTTP requests is destined for the Model segment of your application. This goes for user session data as much as rows in an external database. It also incorporates the rules and restraints governing that data which is referred to as the "business logic". For example, if you wrote business logic for an Order Model in an inventory management application, company internal controls could dictate that purchase orders be subject to a single purchase cash limit of \$500. Purchases over \$500 would need to be considered illegal actions by your Order Model (unless perhaps authorised by someone with elevated authority). Models are therefore the logical location for data access but may also act as a central location for examining, verifying and making final manipulations on that data before

it's stored, and even after it's retrieved.

The ViewThe View is responsible for generating a user interface for your application. In PHP, this is often narrowly defined as where to put all your presentational HTML. While this is true, it's also the place where you can create a system of dynamically generating HTML, RSS, XML, JSON or indeed anything at all being sent to the client browser or application.

The View is ordinarily organised into template files but it can also simply be echoed from or manipulated by the Controller prior to output. It's essential to remember that the View is not just a file format - it also encompasses any PHP code or parsed tags used to organise, filter, decorate and manipulate the format based on data retrieved from one or more Models (or as is often the case, passed from the Model to the View by the Controller).

On a side note, this Blog will not use Smarty. Smarty has a respected history in PHP, but it does have serious failings once you start thinking of the View as a jigsaw puzzle of potentially dozens of reusable pieces pulled together in a single overarching layout. In effect, this method View management is so closely related to OOP as a concept that using PHP itself as a templating language becomes almost inevitable. That's not without a cost (do all designers know PHP?) but it is a manageable cost.

The ControllerControllers are almost deceptively simple in comparison. The primary function of the Controller is to control and delegate. In a typical PHP request to an MVC architecture, the Controller will retrieve user input, supervise the filtering, validation and processing of that input, manage the Model, and finally delegate output generation to the View (optionally passing it one or more Models required to process the current template). The Controller also has a unique difference from other forms of PHP architectural forms since it only requires a single point of entry into the application - almost inevitably `index.php`.

Controller vs ModelNo quick tour of MVC would be complete without a brief mention of at least one extremely common variance in MVC apps. Borrowing a term, it's the idea of a Fat Model and Thin Controller.

A Fat Model, is a Model which takes on as much business logic and data manipulation as you can fit into it. The result is a large body of reusable logic accessible from any Controller. This, in theory, results in a Thin Controller - when all this logic is bundled into the Model behind some suitable APIs, the Controller's average size should be reduced. Less Controller code, less obscuring crud hiding exactly what a Controller is doing.

The opposite is a Thin Model/Fat Controller - business logic is dumped into the Controller which obviously increases its size, and secondly means that code is not reusable (unless you decide to reuse the Controller from another Controller - which is rarely a good idea efficiency wise).

In concert these three segments of an application implement a Model-View-Controller architecture. It's become a widely recognised solution suitable for web applications and it's evident in the majority of the current generation of framework for many programming languages.

In the Zend Framework, these three separations are represented by the `Zend_Db`, `Zend_View` and `Zend_Controller` components. You'll be hearing a lot about these three and the classes they are composed of in the chapters to come! Together these form the backbone of the Zend Framework's MVC architecture and underpin a lot of it's best practices.

The Model-View-Controller was originally used to promote the "separation of concerns" in desktop GUI applications. By separating each concern into an independent layer of the application, it led to a decrease in coupling which in turn made applications easier to design, write, test and maintain. Although GUI applications have turned away from the MVC in recent years, it's proven to be highly effective when applied to web applications.

In the framework this adds a lot of predictable structure since each segment of MVC for any one supported request is segregated into its own group of files. The Controller is represented by `Zend_Controller_Front` and `Zend_Controller_Action` subclasses, the Model by subclasses of `Zend_Db_Table`, and the View by `.phtml` template files and View Helpers. The Zend Framework manages how each is orchestrated in the big picture, leaving you free to focus on just those groupings without worrying about all the code combining them together.

In a sense it's like building a house where the foundations, walls and internal wiring and plumbing are already in place, and all that's left is the internal decoration and a roof. It may take some time to learn how to decorate and roof the prepared sections but once you have learned how, later houses are finished a lot faster!

In ConclusionLike I said, this is upfront assault on understanding MVC. It's not an exhaustive description, so do feel free

to run a few searches and Google and read up on the topic. For web application developers, MVC reading is never a waste. There is a huge body of thought existing covering topics from MVC Testing to which MVC style works best for different situations.

Next up: We investigate a Hello World example which is not quite the simplest possible example since it illustrates a few housekeeping rules to keep even this simple example as tidy and malleable as possible.

Yes. It will have actual PHP code!

Continue to: [Part 3: A Simple Hello World Tutorial](#)

Posted by Pádraic Brady in [PHP General](#), [PHP Security](#), [Zend Framework](#) at 15:37

### **HTMLPurifer 3.1.0 Release Candidate Available**

I am shocked. Here I was the other day gloomily staring at the result of a subversion update to trunk from the HTMLPurifer respository and I got weird fatal errors. Two days later and all is revealed!!!

<http://htmlpurifier.org/news/2008/3.1.0rc1-released.html>

Major change is to implementing Autoloading which is probably why my svn update failed so miserably (I feel bad falling so far behind updates - the head scratching moments are almost amusing at this stage!). Tags are now properly autoclosed. Lots of other stuff marked as new features.

Linked to page has another link to the NEWS summary of changes since 3.0.0.

HTMLPurifer is possibly the most understated underpublicised quality library in PHP today. I consider it a fundamental standard library that is automatically included in every PHP web application I start these days. I mentioned it earlier for the ZF Blog series opener.

Posted by Pádraic Brady in [PHP General](#), [PHP Security](#) at 00:08

Tuesday, April 22, 2008

### An Example Zend Framework Blog Application - Part 1: Introductory Planning

The writing habit has kicked in after a few forum posts and lengthy emails. This inevitable result means I need something long and preferably technical to write about for a while. Since I've just come from the Devnetwork Forums where I was reading a query as to whether any standalone applications using the Zend Framework are available in open source (not many apparently), it was as good an excuse as any.

So here is Part 1 of another lengthy PHP tutorial series on writing a blog application using the Zend Framework. To make things interesting, this will not be an artificial example - when development has ceased, I will deploy and put into production my marvelous new blog, and then proceed to modify it to the point of death, bending it to my will. Muahahahaha.

Be afraid. Be very [deleted cliché].

Starting any new application is like walking into a shop and being dazzled by the displays. You want everything but finally realise you only have so much resources to spend. So you isolate the specifics you must have, and focus on those. That's why I bought a Classic iPod, and not the much flashier iPod (crap all storage anyway).

If we drill down the typical blog application we get a very short list of must-haves.

1. Authentication for Authors
2. Authorisation to create/edit/delete/read entries
3. Methods for adding new entries, and modifying them
4. Publishing entries as RSS and Atom
5. Permalinks unique for each entry (SEO friendly of course)
6. Commenting system
7. Spam detection for new comments
8. Perhaps, trackback detection

These 8 points will form the basis for Maugrim's Marvellous Blog 1.0. Serendipity and Wordpress watch your backs!

But seriously, we will not be challenging S9Y or WP. The application will not have a user friendly installer. You better pray you remember (or I tell you) how virtual hosts work for Apache 2.2. Please ensure you at least have PHP 5.2 installed! And no, there will be no plugin system (well, maybe a simple View Helper driven one integrated into our overall Application Layout). We'll dispense with heavy acceptance testing, and rely only on some light unit testing with PHPUnit where we decide to write some custom classes (if any).

To every extent possible, we'll rely solely on Zend Framework components (possibly a few in-proposal components as needed).

To make my life easier, here are the tools/libraries you can expect to see mentioned. The list is not exhaustive, but pretty much lays out what I consider standard libraries besides the Zend Framework itself. These extend to areas the ZF simply does not cover. It's pretty short actually...Zend Framework 1.5

PHPUnit

HTMLPurifier

Blueprint CSS Framework

jQueryIf you intend following this series and adding to my Google Pagerank, which always is appreciated for the day I launch a massive campaign of Google Text ads here, I will setup a public subversion repository containing the live code, i.e. the stuff I will edit week on week.

Finally, I am not responsible if the final application resulting from this tutorial crashes servers and results in the meltdown of the Internet. Such events are obviously Acts Of God, and I am a mere mortal who cannot be held liable.

Objective of the "An Example Zend Framework Blog Application" series:

Like I said earlier, I have fallen out of love with Serendipity. After a few years worth of upgrading, many of my posts have

lost cohesion riddled with minor formatting errors from various Geshi, rich text editor and plugin filter order changes. I've also decided the main problem with an out of the box solution is that you need to learn a whole new API and development approach before you can tinker with the source code. Did I mention I am completely willing to pay money not to edit procedural code?

The series will culminate with a complete blogging solution for my personal needs. I'll cook it, bake it, beat it into submission, and then deploy it to replace this existing Serendipity installation. Along the way I'll revisit numerous Zend Framework components, and probably praise or dissect them for problems or shortcomings. Along with my own mistakes, no doubt!

Reading the series in order, should hopefully piece together a reasonable approach to getting a Zend Framework based application off the ground. I will keep the more Agile/XP mentions to a minimum (not the focus of this series) but they will be in the background if anyone has questions.

Next up: A quick introduction to the Zend Framework of only fifteen pages of text. Installing the Framework. Setting up the simplest possible Zend Framework application whose sole utility is to say Hello.

Continue to: [Part 2: The MVC Application Architecture](#)

Posted by Pádraic Brady in [PHP General](#), [PHP Security](#), [Zend Framework](#) at 16:15

Sunday, April 20, 2008

### **He creepeth back into PHP**

A number of folk have displayed some concern over my absence from all things PHP for the last month or more, something exacerbated by my lax attention to incoming emails.

To fill in the why, I recently lost a close friend to cancer after he endured a four year battle with the disease. Following this loss I haven't had much in the way of enthusiasm for open source or writing hobbies and the path to renewed vigour for plugging you all with articles, critical posts, and my usual rambling on random topics has been a long one. I've continued to work away on Eve Online which has been one of the few sources of distracting fun these days...and not much else online outside the 9-5 job.

C'est la vie.

My friend always had a few things going for him. He was incredibly courageous in facing the inevitable. He remained intensely faithful to God, an all too rare display for young people these days when I count myself as one of the few under-thirties to attend local church in my community, and one of the fewer who consider it not just another boring pointless practice. And I have never once witnessed him act out of anger or unkindness.

He was one of those bright lights in a person's life that leaves a horribly empty void behind when they are no longer with you.

Rest in peace my friend.

In my absence I've turned away from a few opportunities in PHP I would ordinarily have been overjoyed to have. Chances are I'll look into how salvagable some of those are but I'm not too anxious about them to be honest. I'm looking towards the blank slate of the next few months if they're not retrievable and wondering what to fill them with. PHPSpec/PHPMock and the mailing list (I'll catch up on the last few emails soon) goes without saying. The book I was considering is more than likely a lost cause unless the author position remains unfilled - no recent emails there so that appears possible. Well, more time to freelance a few articles. Maybe I'll poke around the ZF mailing list to see if I missed anything interesting and get some of those Microformat proposals moving.

For now I'm going to nip into Eve and figure out whether I really want to train Heavy Missiles V. Once that's done I'm going to hit my usual PHP train of activity and pick up the pieces from the last month. All the usual haunts will once again see Maugrim The Reaper putting in an appearance and finding someone to argue over something with.

Posted by Pádraic Brady in Irishisms, PHP Security at 11:34