



## 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-Controller

The 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 its 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 Model**

The 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  $\hat{a},-500$ . Purchases over  $\hat{a},-500$  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 View**

The 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 Controller**

Controllers 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 Model

No 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 Conclusion

Like 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 22:37

I have been using PHP for less than a year and I am hope that your example will serve to replace the awful blogging system I am running on my site at the moment.

As a trainee programmer, I have only ever taken an MVC approach whilst building Java Desktop Applications where the techniques and benefits are clearer. I wish there were more resources on the net for students of PHP who wish to adopt OO principles and follow best practices.


I look forward to keeping up with the tutorials [I like your writing style]. Yepo. Anonymous on Apr 23 2008, 23:39

Ditto on the writing style 

So, just to clarify, does the ZF lend itself more to a "Fat Model and Thin Controller" style than a "Fat Controller" one? Or doesn't it work like that? I'm very much looking forward to part 3... Anonymous on Apr 24 2008, 17:47

The ZF is largely agnostic to whether you want a Fat Model or a Fat Controller. If you think about it, the business logic is one thing that will vary from app to app, so it's organisation is completely up to the developer.


My own preference is to use a Fat Model - part of the problem with Fat Controllers is that your wishfully neat Controller now expands to hundreds of lines of repetitive code instead of being a simpler 20 line block. In a sense it's like mutating controller actions into the old style Page Controllers PHP is usually famous for, instead of pushing code to other more specialised classes for reuse/maintainance - and most importantly - testing.

One of the common problems with MVC apps is just that - controller become a giant spaghetti mass of code because people forget a framework does not excuse you from maintaining your own application specific class library 

. Anonymous on Apr 24 2008, 19:38

Zend Framework supports both models, but doesn't explicitly encourage one of them. If you use Zend\_Db\_table heavily and create a class for each of your DB tables and put validation code in that class, you will get a Fat Model/Thin Controller architecture. If you are used to other db abstraction libraries like ADODB and don't want a full Object-Relational-Mapping (ORM), you use Zend\_Db, Zend\_Db\_Statement and Zend\_Db\_Select in your controller and you will get a Thin Model/Fat Controller style.

Personally I prefer the Fat Model/Thin Controller style because it can be tested and reused more easily. Anonymous on Apr 24 2008, 19:46

Thanks Pádraic and Gabriel 

It's a lot clearer for me now and, I guess, will become even more so when I see the structure of your model and controller code.

Thanks again for both your replies! Anonymous on Apr 24 2008, 20:03

Actually, I like my models to be as dumb about business logic as possible. That being said, I don't like my controllers to fat either. A way that I've balanced this a little bit is by creating a service layer that handles most of the business logic. That way any controller that needs to can call that service layer and do what it needs to do.

The only thing I like my models doing is validating and persisting their data. The only thing I really like my Controllers doing is

handling flow. I stick anything that I know/think/needs to be re-used by different parts of the application in a service layer. Granted its more lines of code but that way I can rip out controllers, change models around a bit...or even replace that service layer with some other business logic should business processes change, and I don't have to muck much with my model.

Funny to hear be talk about "adding" layers since I **hate** alot of layers. The Java world suffers from an abundance of layers. (At least Grails hides most of them from you). Anonymous on Apr 27 2008, 00:34

"Although GUI applications have turned away from the MVC in recent years"

Have gui apps really turned away from MVC?

When I first started learning about OOP and MVC I came across many articles relating to apple OS X and KDE that seemed to be all about MVC. Anonymous on May 10 2008, 16:38

One of the better introductions to MVC out there. Kudos. Anonymous on Dec 24 2008, 01:26

In reference to part of your article above while talking about the view, you said the following:

"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."

I think it is only fair to mention that Smarty is about as complicated as PHP if not more so. Most designers don't come knowing Smarty or HTML for that matter. Anonymous on Feb 2 2009, 05:04