

Controller vs Model in Zend Framework

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 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 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!

This article is the part of [Pádraic Brady on blog.astrumfutura.com](http://blog.astrumfutura.com)