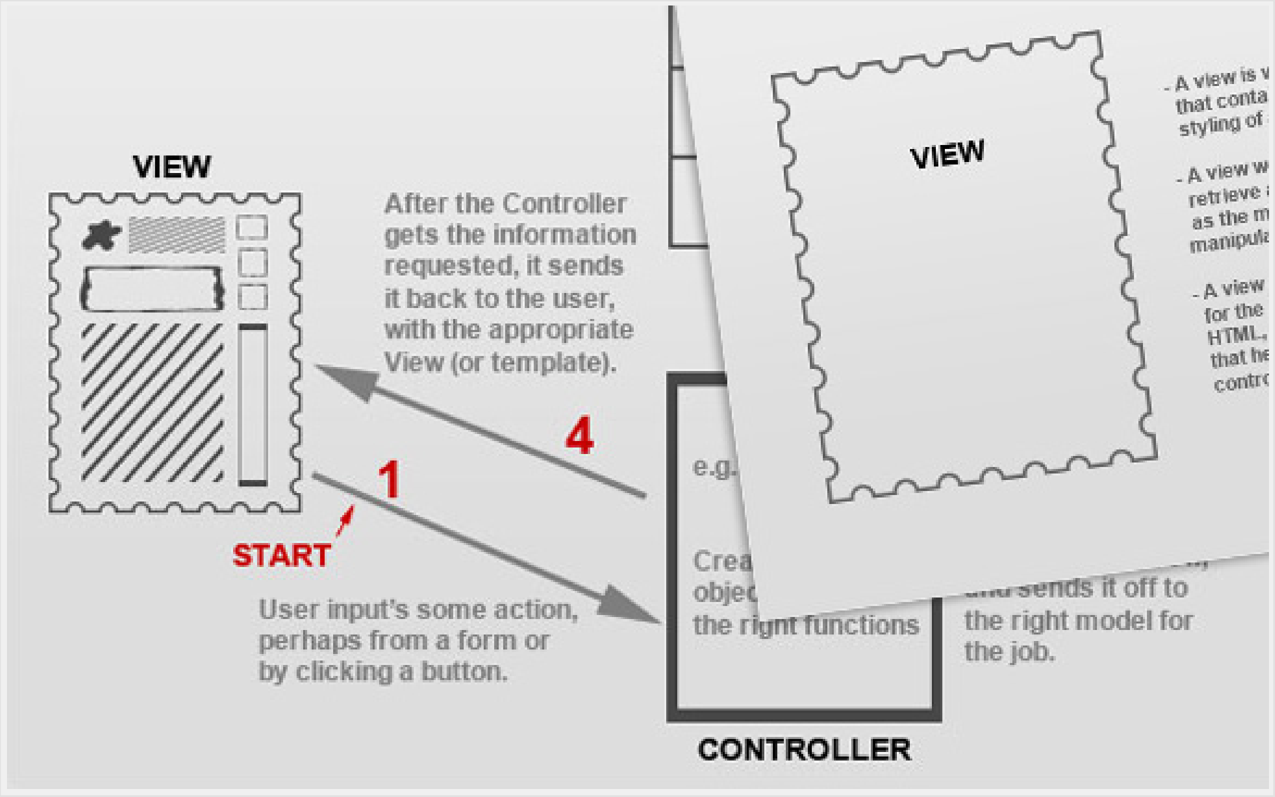
onextrapixel.com

A Detailed Overview of the Model-View-Controller (MVC) Coding Structure

March 14, 2012 • 4 min read • original

The **model-view-controller or MVC** is software architecture commonly used for creating web applications or software. In other words, it's a structure for web applications to follow in order to ensure efficiency and consistency. Many of the most popular frameworks use the MVC architecture, including ASP.NET, CodeIgniter, Zend, Django, and Ruby on Rails. At the same time, there are many web developers who don't use a coding framework yet still set up their applications to follow the MVC structure.


VIEW
After the Controller gets the information requested, it sends it back to the user, with the appropriate View (or template).
4
1
START
User input's some action, perhaps from a form or by clicking a button.
e.g.
Crea
objec
the right functions
and sends it off to the right model for the job.
CONTROLLER
VIEW
- A view is v
that conta
styling of
- A view w
retrieve
as the m
manipula
- A view
for the
HTML,
that he
contro

In this article we'll look in more depth at what the MVC architecture is in detail, and what each part is, and why you should be using it.

Why You Should Use It

Without a good reason to use a new structure, framework, technology, or trend, many developers may have a hard time getting motivated to learn a new topic. So, to begin, we'll first introduce why the model-view-controller architecture is so important, and why you should begin adopting its practices in your next web application.

With the separation of the use of code between the three file types, you obtain a separation in web application logic. You obtain an almost complete separation in programming logic and the interface code *(markup and styles)*, and further separation in the type of logical code for the application.

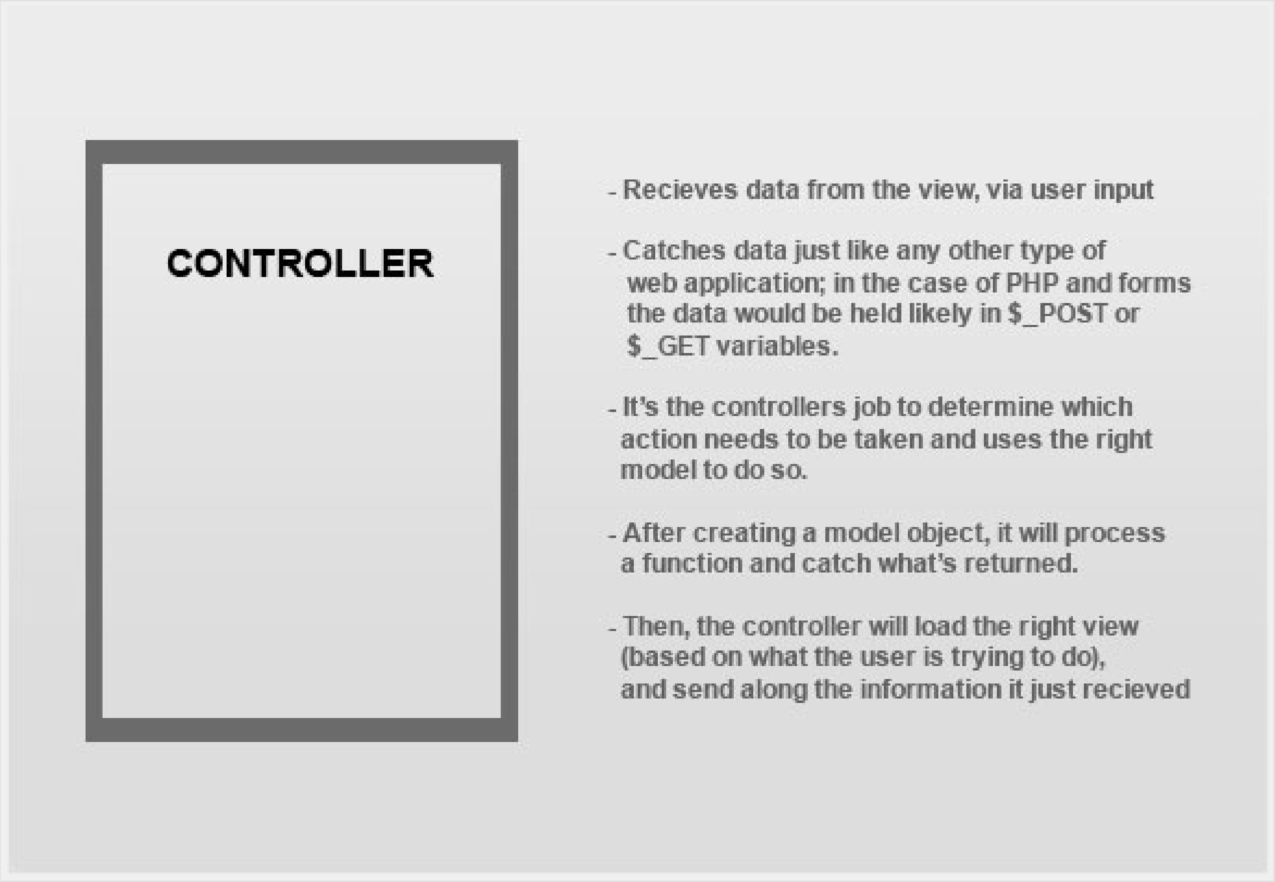
Basically, a web application or piece of software that follows the MVC structure separates the three main types of functionality into three types of files: models, views, and controllers *(as you may have guessed)*. This allows for each portion to be designed, implemented, and tested independently from any other one, keeping code organized. Keeping the code organized means being able to find what is needed quickly, test features, correct or alter them quicker, and add new functionality with ease. It also means more efficient code, and a better way to re-use code for faster applications.

Probably one of the greatest benefits however is that many developers understand and use the MVC structure for creating web applications. If developers use any of the popular web development frameworks, then they understand and use the MVC structure as well. Because of this consistency, managing a project between several developers can be easier as well.

Now you know why it′s important and basically what the structure′s purpose is, so let′s now look into each portion:

Controllers

The controller can be considered the ″middle man″ of the application. It works with the user, taking in data, and then working with the model to get the appropriate data or calculation, and then working with the view to show the response to the user.



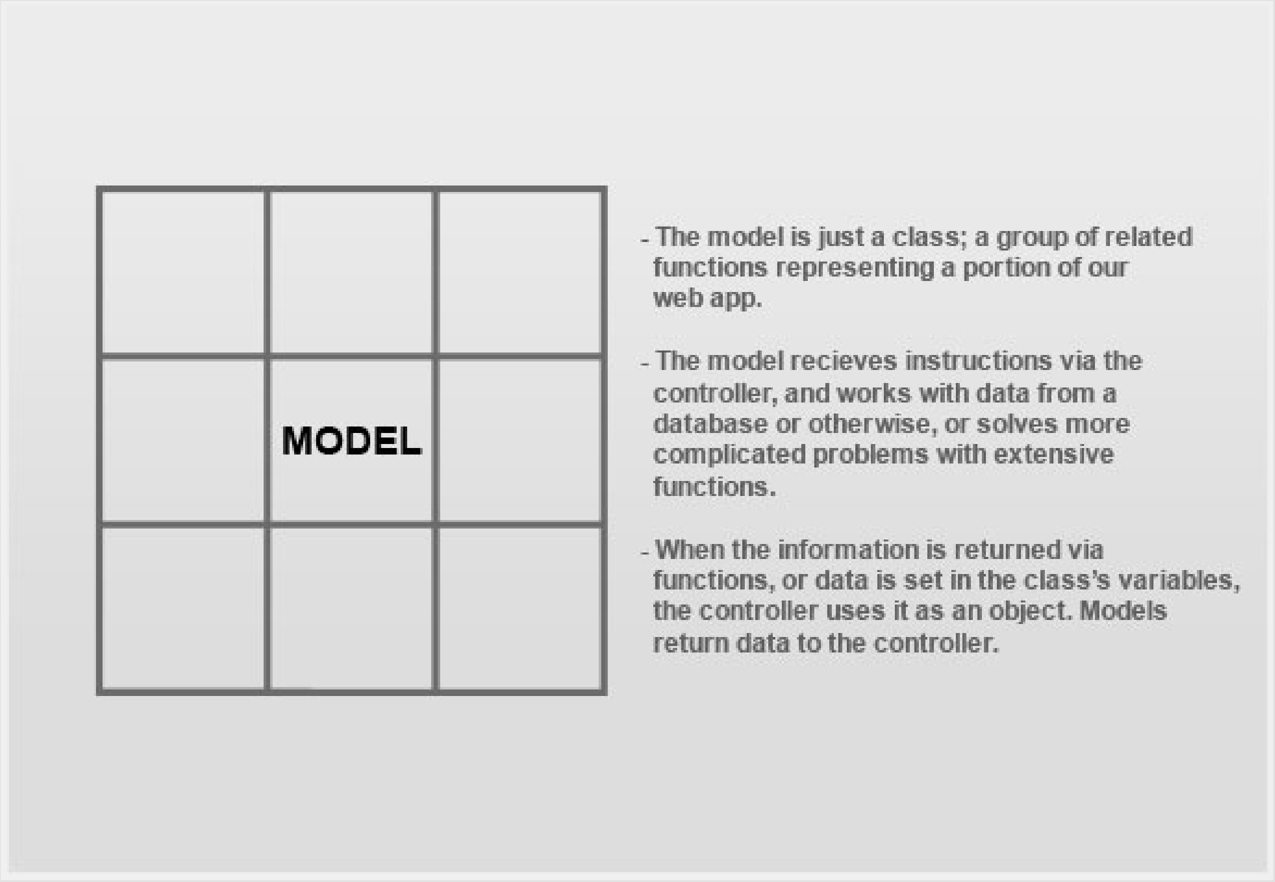
- Recieves data from the view, via user input
- Catches data just like any other type of web application; in the case of PHP and forms the data would be held likely in $_POST or $_GET variables.
- It's the controllers job to determine which action needs to be taken and uses the right model to do so.
- After creating a model object, it will process a function and catch what's returned.
- Then, the controller will load the right view (based on what the user is trying to do), and send along the information it just recieved

When the user carries out any action, it first goes to the controller. It will take in any data, for example `$_GET` and `$_POST` variables in PHP, and determine what should be done with the data. In short, models deal with handling data and extended functionality. Therefore, the controller's job at this point is to determine which model should be called, and then call the appropriate function within that model. After calling the function, it will catch the result *(usually in a variable)*.

After it gets whatever information it needs from the model, it will determine what view to send that feedback to, and send the user there.

Models

A model is simply a representation of something we need to deal within our application. It is a "model" for something we must represent in code, such as a book, user, bank account, or whatever. The model is responsible for holding the functions and variables that are involved with whatever it's representing. You can think of a model's logic as the core concept to object oriented programming — models are just our "classes". However, don't let this confuse you as controllers are technically structured as classes as well.



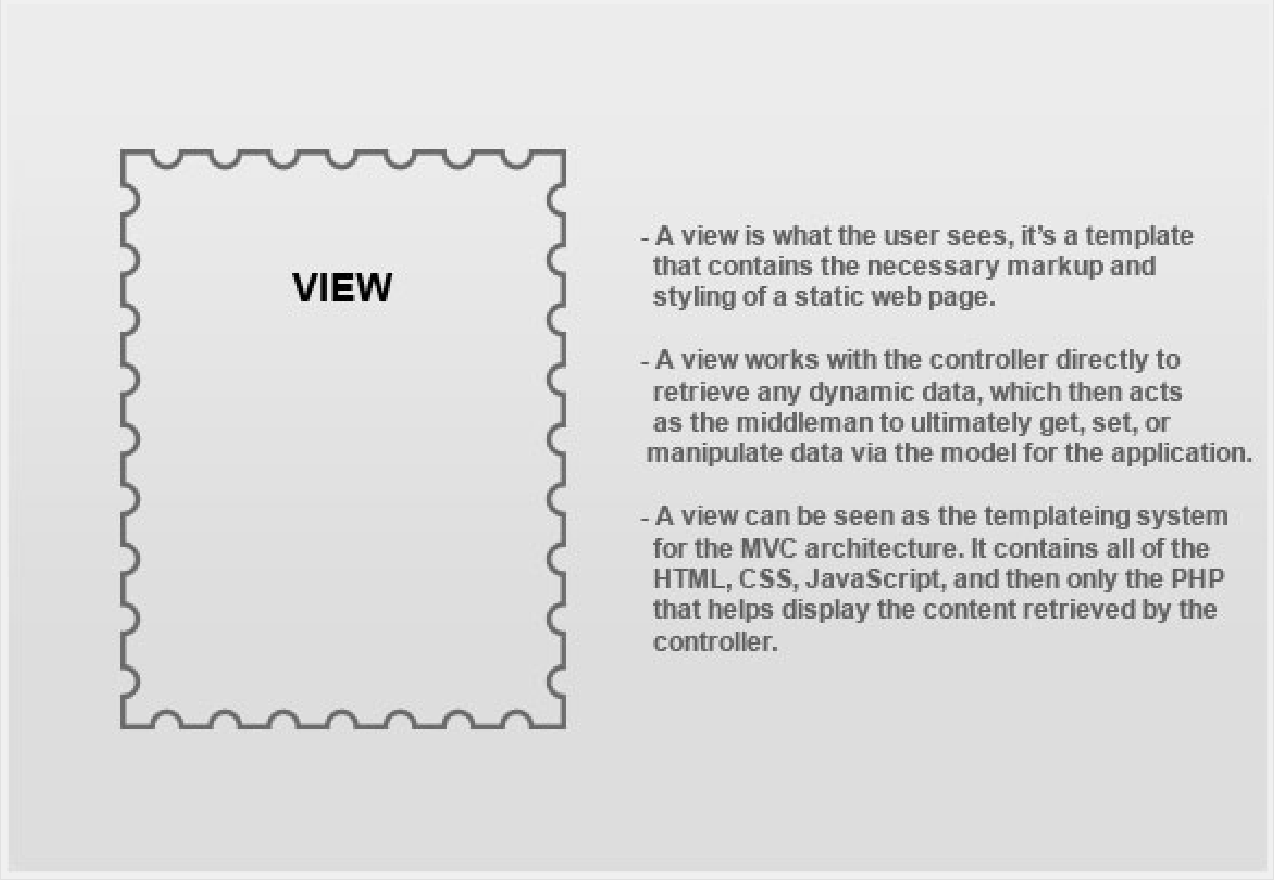
- The model is just a class; a group of related functions representing a portion of our web app.
- The model recieves instructions via the controller, and works with data from a database or otherwise, or solves more complicated problems with extensive functions.
- When the information is returned via functions, or data is set in the class's variables, the controller uses it as an object. Models return data to the controller.

For example, a "User" model would be a class that represents our user. *(It models our user.)* It holds variables that pertain to the user's information, and functions that get the user's info, set it, update it, get their profile picture, update their status, or do anything else that would require interaction with the database, or otherwise an extensive function *(like making a calculation based on user input)*.

Just like a class in traditional object oriented programming, a model simply contains a collection of similar functions, or functions that deal with whatever the model is representing.

Views

Finally, after the controller requests information from the model it sends it to a view. A view is just like the application's templating system — there might be a view for a certain type of page layout *(profile page)*, a mobile view, or a view for a particular theme/skin. A view will contain all of the markup, CSS, and etc. that you traditionally use with creating a static web page.

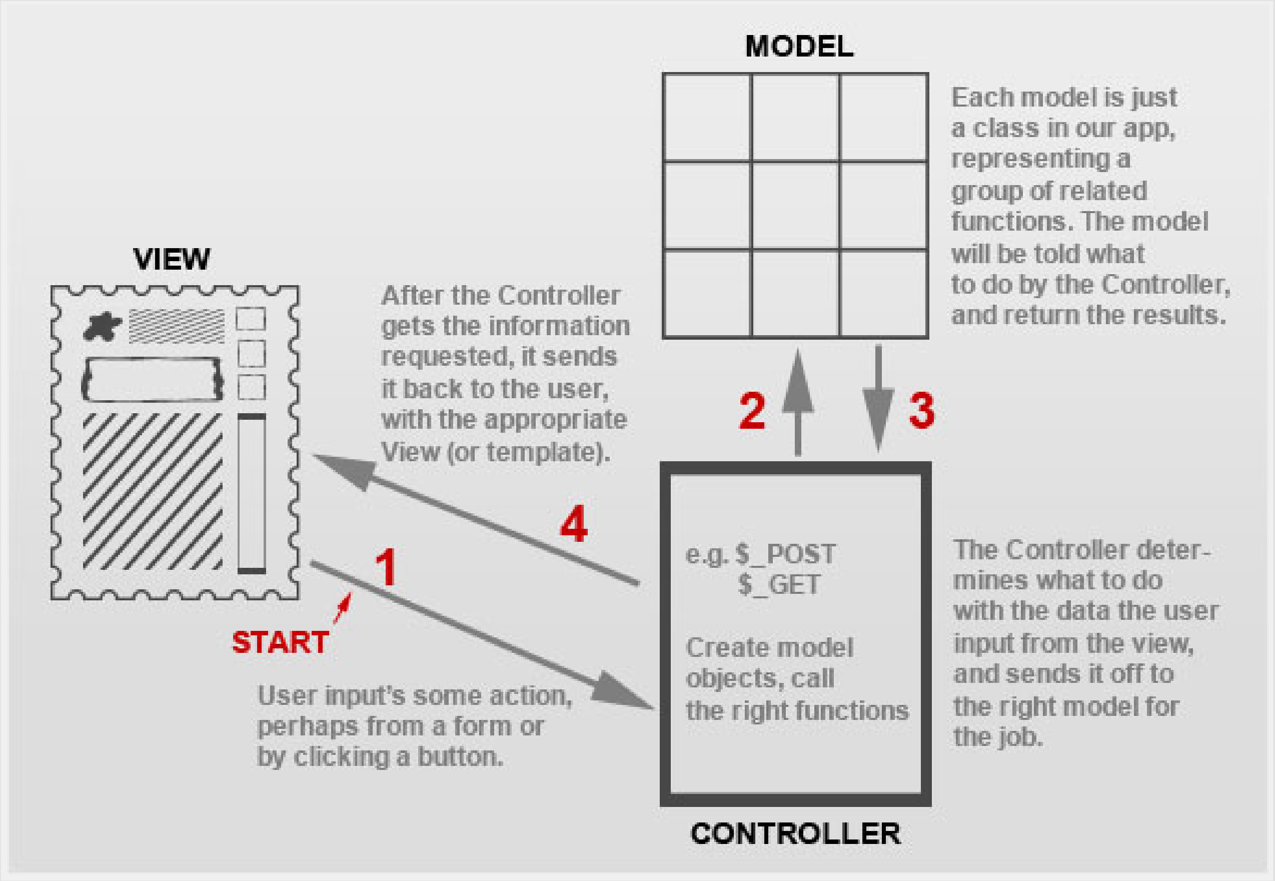


- A view is what the user sees, it's a template that contains the necessary markup and styling of a static web page.
- A view works with the controller directly to retrieve any dynamic data, which then acts as the middleman to ultimately get, set, or manipulate data via the model for the application.
- A view can be seen as the templateing system for the MVC architecture. It contains all of the HTML, CSS, JavaScript, and then only the PHP that helps display the content retrieved by the controller.

It's what the user sees, and what the controller turns to the user. The controller simply forwards the user to the correct view, based on what they're trying to do, but after they've received data from the model and forwarded this information to the view. The view then displays the information it's given, in the format it's structured with.

MVC Diagram

Sometimes seeing something makes it easier to understand. Below is a visual diagram of how the model-view-controller architecture works, starting from the user's view (which would be a template, or view in MVC), sending the requested action through the architecture, and ending back at the view with the completed request.


MODEL
Each model is just a class in our app, representing a group of related functions. The model will be told what to do by the Controller, and return the results.
VIEW
After the Controller gets the information requested, it sends it back to the user, with the appropriate View (or template).
2
3
4
1
START
e.g. $_POST $_GET
Create model objects, call the right functions
The Controller determines what to do with the data the user input from the view, and sends it off to the right model for the job.
User input's some action, perhaps from a form or by clicking a button.
CONTROLLER

File Structure

The file structure for using MVC in the standard way is quite simple — there are simply folders for views, models, and controllers, and they all link to each other via one directory up. Of course, with any web application you'll also have other folders and files, such as an index file (shown below), a folder for images, includes, and etc.

Below is a simple MVC directory structure with some example files. These follow my general naming conventions, and while each developer may have their own, it's smart to have some standard naming convention of some sort. The files in the example structure below are merely examples, and developers may choose to structure or name their files slightly differently.

```
controllers/
        updateUserController.php
        showUserController.php
        manageContentController.php
        manageBlogController.php
models/
        User.php
        Blog.php
        Content.php
views/
        userProfile.php
        userEditProfile.php
        contentPage.php
        postPage.php
index.php
```

More Resources

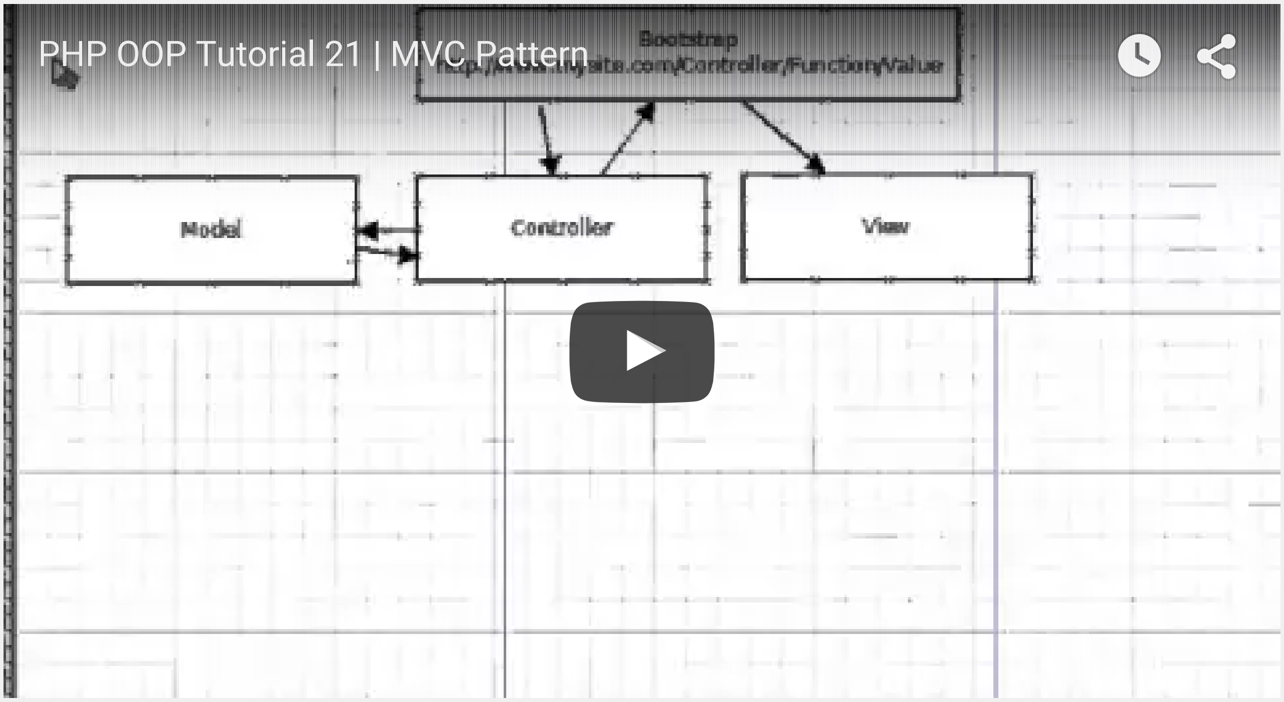
This tutorial has just been a primer to the MVC architecture, but of course there are many more tutorials on the subject. Whether you want to learn more, or are looking for more clarity on the subject, below are some more tutorials, videos, and resources for undertanding this software architecture.

Written Tutorials

Video Tutorials

Build your First Tiny PHP MVC Framework

PHP OOP Tutorial 21 | MVC Pattern


PHP OOP Tutorial 21 | MVC Pattern
Bootstrap
http://www.mysite.com/Controller/Function/Value
Model
Controller
View

PHP Login System Using OOP and MVC - Introduction

Log In

Username: * admin

Password: * ••••••

* required fields

Submit

PHP Login System Using OOP and MVC - File and Database Setup

PHP Login System Using OOP and MVC - File and Database Setup

Model

- A model is an object representing data or even activity, e.g. a database table or even some plant floor production-machine process.
- The model manages the behavior and data of the application domain, responds to requests for information about its state and responds to instructions to change state.
- The model represents enterprise data and the business rules that govern access to and updates of this data. Often the model serves as a software approximation to a real-world process, so simple real-world modeling techniques apply when defining the model.
- The model is the piece that represents the state and low-level behavior of the component. It manages the state and conducts all transformations on that state. The model has no specific knowledge of either its controllers or its views. The view is the piece that manages the visual display of the state represented by the model. A model can have more than one view.

Note that the model may not necessarily have a persistent data store (database), but if it does it may access it through a separate Data Access Object (DAO).

View

- A view is some form of visualisation of the state of the model.
- The view manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application. Instead of a bitmapped display the view may generate HTML or PDF output.
- The view renders the contents of a model. It accesses enterprise data through the model and specifies how that data should be presented.
- The view is responsible for mapping graphics onto a device. A view typically has a one to one correspondence with a display surface and knows how to render to it. A view attaches to a model and renders its contents to the display surface.

Controller

- A controller offers facilities to change the state of the model. The controller interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.
- A controller is the means by which the user interacts with the application. A controller accepts input from the user and instructs the model and view to perform actions based on that input. In effect, the controller is responsible for mapping end-user action to application response.
- The controller translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application they appear as HTTP GET and POST requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

ASP.NET from Scratch: MVC

Conclusion

The model-view-controller architecture is a software structure that any developer should learn. I've unfortunately seen myself how some developers will use coding frameworks that use MVC, such as CodeIgniter or CakePHP and not use the MVC concept correctly. These frameworks use this architecture for a reason, and it is meant to keep logic and design separate, while maintaining efficiency. Even without frameworks, developers should apply this architecture to an extent, and understand it for larger projects in the least.

Do you have any other great resources for learning the MVC architecture? Any tips on understanding it, or tips for using it?

Original URL:

http://www.onextrapixel.com/2012/03/14/a-detailed-overview-of-the-model-view-controller-mvc-coding-structure/