## Introduction

This post introduces the Model-View-Controller architectural style. We’ll use this style for all our web sites. The MVC style was originally introduced by Sun.

You can find a lecture on MVC here: <https://ist.berkeley.edu/as-ag/pub/pdf/mvc-seminar.pdf>.

The Wikipedia article is here: <http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>.

Apple’s take on the style can be found here:

<https://developer.apple.com/library/ios/documentation/general/conceptual/devpedia-cocoacore/MVC.html>

For the tl;dr people among you, the short version is described in this document.

## Objectives

The MVC style builds on our ideas of ***separation of concerns*** and ***SOLID*** values. As we start to build larger websites, we need to impose some order on the classes we use and how they interact. We’ll organise our classes into three families: models, views and controllers. Note that there will be many of each in a typical web application.

## Model

A model takes responsibility for the integrity and storage of data. It is the heart of our application and defines the business rules that characterise the app. For example, let’s say we are writing an eCommerce application that will manage products. We’ll have a class for a product that will be the product model. This model will know how to validate product data so we only store valid data in the database and it will know how to store the data in the database. Whenever we want to work with product data, we’ll ask the product model to do it for us. Of course, we’ll have many other models for orders, customers etc., but they’ll all fall into the model family; they’ll do the same sort of thing, just on different data.

A model should know nothing about how it is used. This guarantees that we can use the model *anywhere* we need it. In particular, it won’t be concerned about how the data are displayed (that’s the job of views, discussed below) and it won’t be concerned with details about how the user interacts with the data (that’s the job of a controller, discussed below).

If we’re smart about how we set things up, we can write and test models independently of any other components. More on this later.

## View

The job of a view is to display information to a user. In web, we’ll use a lot of html to do this. Basically, we’ll give the view a model so it can get the data it needs and say go create the html we need. Classes in the view family should be the only ones that create html. Also, a view is only concerned with presentation – it should not update anything or work directly with the database.

## Controller

A controller is responsible for managing user interaction. There is normally a separate controller for each system feature or process. The controller will normally manage user interaction, create the model(s) needed for the feature, and create views to display feedback to the user.

## Implementation

To make things concrete, I’ll illustrate some of these ideas in code. Rather than interfaces, I’m using abstract classes this time. I’ve created an AbstractModel, an AbstractView and an Abstract Controller. The view works with a template so we can separate the html from the code. This is separation of concerns and at this point, I could hand over the UI design work to the front-end designer team. I’ve created a master page and some css as a starting point.

Remember that, because we’re following SOLID princples, I can extend these classes easily in the future. Also, we’re taking an ***agile*** approach to architecture. Rather than creating all the architecture up front, we’ll create just the bits we need, just in time.

### AbstractModel

The abstract model has almost nothing in in (yet!) just three methods:

* getDB()
* hasChanges();
* save();

The dependency on an IDatabase is injected in the constructor.

### AbstractView

The abstract view is a bit more interesting. There is a getter and a setter for a model and three methods relating to template handling:

* setTemplate($template)
* setTemplateField($name,$value){
* setTemplateFields($fields)

The template is basically html with some replaceable parameters. There are two other methods:

* prepare ()
* render()

Prepare will normally be overridden in the subclass. It gives the subclass a chance to do its work. Render will not normally be overridden. This method will actually write out the html. Basically, it’ll be the only place in the whole site that uses a print or echo statement.

I’ve used a setter for the model and parameter-less constructor and prepare and render methods. I could leave the setter out and just inject it in the constructor. Alternatively, I could pass it as a parameter on the prepare method. What do you all think? What are the strengths and weaknesses of each approach?

### AbstractController

The AbstractController has a few protected helper methods as a convenience for subclasses:

* getDB()
* getURI()
* getConfig()

These give access to the parts of the context injected in the constructor. The heavy lifting is carried out by the main process method. This validates the http verb used in the request and manages the logic of display or redirect and post-back identification. The main method called is getView which is expected to be overridden by subclasses. Redirection is not yet implemented and throws a ‘not yet implemented’ exception.

### Sample implementation

To illustrate some of the ideas I’ve implemented some parts of a person list application. I’ve written two model classes: a person class managing a single person, and a people class that wraps a list of person objects. I’ve written a PeopleView class that handled the display of the list with editing links and a PeopleController to tie it all together.

To exercise this, I’ve written a PersonList app (just 5 lines of code). This just creates a context, creates a controller, and asks it to do its work.

### Evaluation

I’ve written this as a standalone script. This is a bit clumsy so I’ll refactor some of this after my next post which will introduce the concept of clean URIs and a *front-end controller*. You’ll also notice that I’ve left out the menubar. That’s also because we’ve not yet made the mode to clean URIs.

Meanwhile, have a play with the supplied code, ask questions, challenge the design, have fun.

Hopefully, you’ll all agree that we’re finally getting to have something that looks like a web site.