A simple MVC framework

Radu Potop

Masterat TI An I

Table of Contents

Introduction....3
Concepts....3
Existing solutions....4
Architecture....5
Framework design....5
    Component design....5
    Class design....6
Developing the framework....9
Example application....10
Conclusions....10
References....11

Introduction

In this project we will follow the creation of a framework for web applications. It's called "*A simple MVC framework"* because it focuses mostly on implementing the Model–View–Controller pattern, and not on including all possible features like bigger / production frameworks do. However, it does have a set of libraries which help us create web applications more easily.

Concepts

Neither frameworks nor MVC are new concepts. The first ones are (usually) massive sets of libraries, sometimes paired with IDEs, that try to ease our work with developing applications of all kinds. The latter was invented by Trygve Reenskaug in 1979 at Xerox PARC [1], as a method to separate the user interface (view) from business logic (controller) and from data storage (model). In the figure below we have the relations between these components:

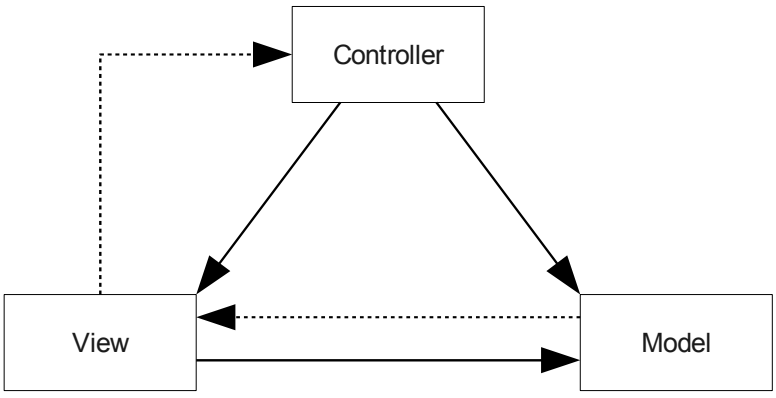

Fig 1: MVC

In a nutshell, the controller interrogates the model for data, does processing on it, then passes it to the view, where it is displayed to the user. When the user alters data in the view,

the view emits events that are listened to by the controller.
The controller then fetches this data, processes it, and submits it back to the model for storage. Optionally the View can *talk* to the Model, but we won't use this feature in our framework.

The solid lines represent a direct association, while the dashed lines represent an indirect association via an observer [2][3].

Existing solutions

Framework driven web application development flourished after Ruby-on-Rails (RoR) was introduced in 2004 [4]. Even though frameworks for desktop applications were common-place, with web development the situation was much more different. Most web applications were made using in-house tools that varied in quality, often lacking any design or good code practices. *Spaghetti code* was used to describe this situation, where HTML code was mixed with programming language code and with SQL queries.

After Ruby-on-Rails, other frameworks surfaced such as Django for Python or Zend Framework for PHP. Because Zend Framework is quite a heavy-weight application, a lot of lighter frameworks emerged for the PHP language, some of the most popular being CodeIgniter and CakePHP. All these frameworks implement MVC as a fundamental part.

Our framework is somewhat inspired by CakePHP, but without sharing any code. Despite being considered light, CakePHP still has a hefty 160000 LOC. Our framework is much smaller than that at about 500 LOC. Even though it's written in PHP which is notorious for procedural coding and the bad practices it induced in its programmers, our framework uses entirely an object-oriented design, and tries to implement MVC cleanly.

Architecture

The architecture needed to run this framework is basically a classic LAMP (Linux, Apache, MySQL, PHP) setup. The versions of Linux, Apache and MySQL don't really matter, only PHP has to be version 5.

Framework design

Component and class design are two major steps when engineering an application.

Component design

When building any application (in our case a framework), preliminary design must be done before starting to write the code. A UML component schema can help us outline the main components of our framework, like in Fig 2.

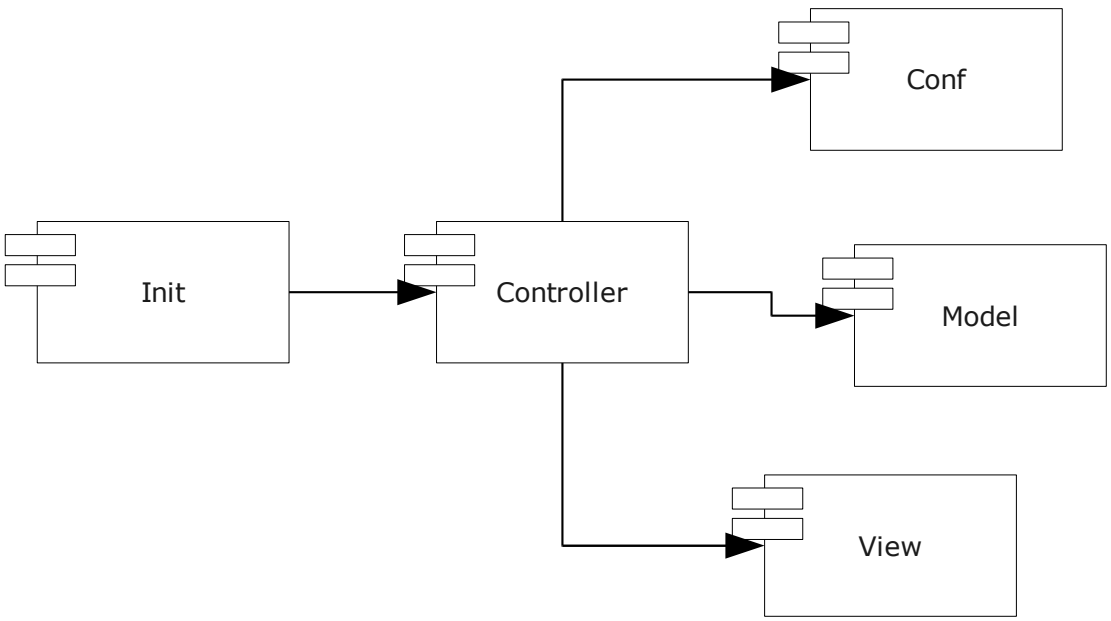

Fig 2: Components

Basically we have an Init (from initialization) component that starts the framework. Other frameworks call this *Bootstrap*. The Init component handles URLs, and based on the requested URL it dispatches the Controller.

The Controller is the central node of our framework, that does all the work. First it loads the Conf component that contains all the configuration details necessary for the framework and application. After the Conf is loaded, it loads the Model, that connects to the database and allows us to work with it. Finally the View component is loaded, that handles the User Interface.

After all this, we can start working with the Controller, Model and View to effectively write our application on top of the framework.

Class design

The object-oriented design scheme provides us with a deeper understanding of the framework's internal workings. This is also a UML scheme, but specifically for classes (Fig 3)

Init is the first component called when the framework is started. The autoload function is responsible for dynamically loading any class that the application or the framework uses. The init function parses the URL and based on the URL it instantiates the corresponding controller and calls an action (method) from it. If for example, the URL is http://example.org/blog/add/ then the controller will be *blog*, and the action will be *add*. In our schema BlogController is instantiated as an example. If no controller is called then MainController is instantiated. If no action is called then main( ) is called by default. This controller is where the application business logic happens.

BlogController extends the Controller class and inherits its properties and methods. In the Controller's constructor, the Conf, Model and View classes are instantiated and the objects are attributed to their corresponding properties ($Conf, $Model, $View). The Controller's properties are all protected so they can be referenced only from an inherited object. Note that the $controller and $action properties are lowercase. This is because the values they hold are strings (the name of the controller, and the action called) as opposed to $Conf, $Model and $View which hold objects.

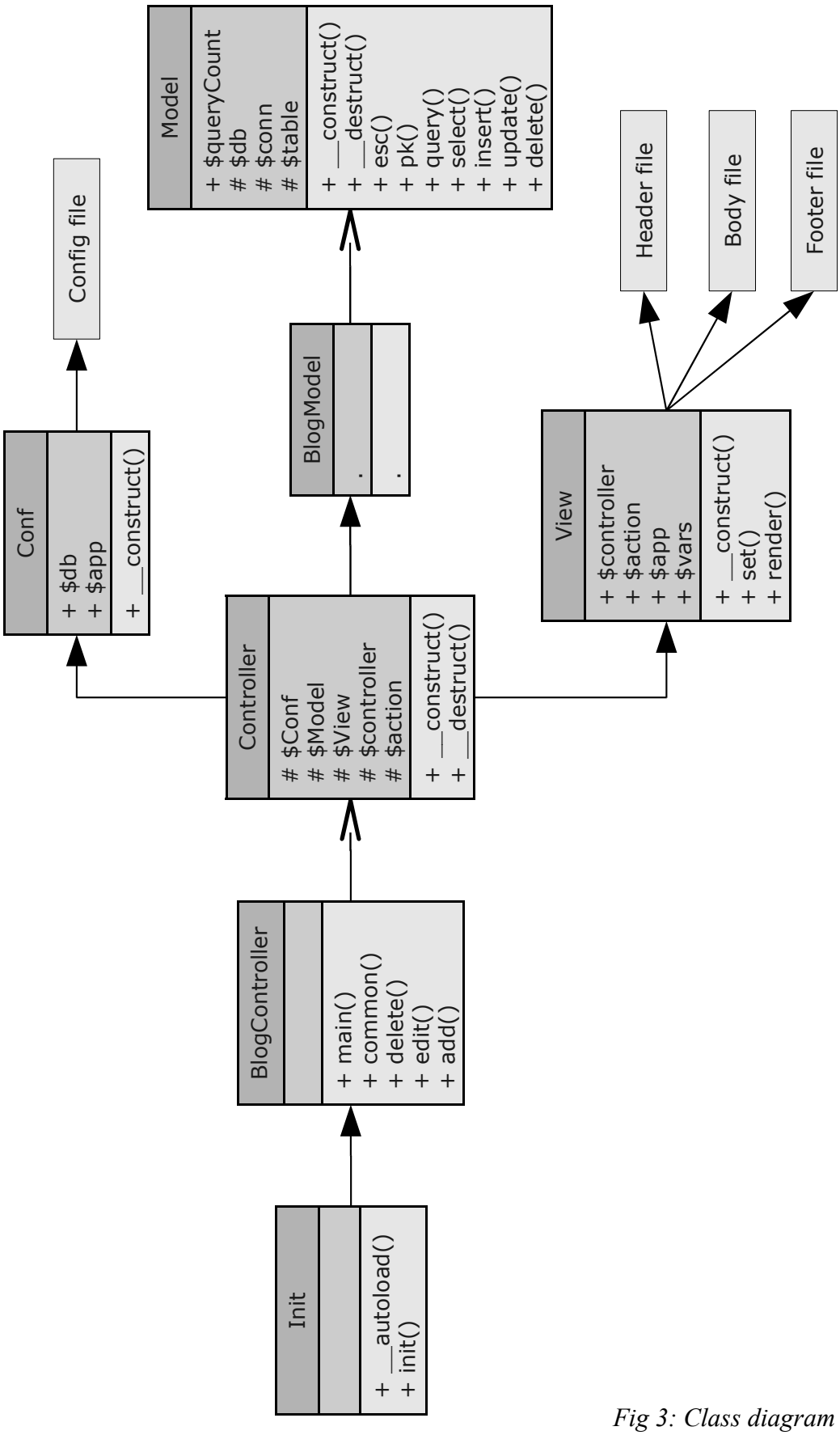

Fig 3: Class diagram

The Conf class in instantiated in the Controller. The Conf class reads a config.php file and transposes its contents into a object with properties. From the Controller, parts of Conf are passed to the Model and the View. Conf is also accessible from the BlogController at any time.

The BlogModel class is also instantiated in the Controller. In our example, BlogModel is empty because the Model abstracts the database sufficiently, so we won't build another abstraction on top of that. We will use the methods from the Model directly in the BlogController. The model name is also the table name on which we make SQL operations. For example if we have BlogController, the table used will be *blog*.

BlogModel extends the Model, which in turn works with the database.

The Model connects to the database in the constructor and disconnects in the destructor. The Model is an abstraction layer on top of the database. It has functions such as select, insert, delete or update that follow the SQL command names closely, but builds on top of them and creates a library that eases the developer's work. For example, the insert method from our Model can be given an array as input, which simplifies inserting data from the programming language. Also the select method will output an array, which again eases the work with the database significantly.

The Model also features an esc( ) method - used by select, insert, update and delete to escape values before feeding them to the database, and a queryCount property, which is a SQL query counter.

The View is the class which handles the User Interface. It is instanced in the Controller, so we can access the View's set( ) method from the BlogController. Using this method we can send data to the View, to be displayed. However, the User Interface itself is rendered last, when Controller object is destroyed. When the render( ) method is called, the View takes the header, the footer and the body corresponding to the current controller's action and assembles the page. The View can emit events using GET and POST HTTP requests, which are captured by the BlogController.

After the interface is rendered no more processing takes place and the application exits, until another URL is requested.

Developing the framework

During the development stage the code itself is written and files organized.

Conventionally, file names are also class names. The following directory structure was chosen for the framework:

/	holds the index and the config file. The index redirects to the Init component.
library/	holds all the base classes: Controller, Model, View and also the Conf class and Init.
controllers/	holds application controllers, such as BlogController or MainController. They extend the base Controller.
models/	holds application models, such as BlogModel or MainModel. They extend the base Model.
views/	holds the interface files. These are not classes, just HTML files with a minimal amount of PHP in them. For each controller there must be a directory with its name (such as blog, or main). Each directory stores the interface files separately (separate files for: add, edit, delete, etc.). The views directory also includes the header and footer files, that are common for all views.

Also note that the code was commented using the JavaDoc [5] syntax, which ensures a clean and standard method of commenting code (with the possibility to auto-generate documentation if desired).

Mercurial (a distributed version control system) [6] was used for tracking versions. The source for the framework can be found here: http://bitbucket.org/wooptoo/simplemvc/

Example application

The best way to demonstrate how the framework works is by building an example application on top of it. We chose to build a simple blog. A working demo can be found at: http://wooptoo.com/demo/ The blog application can be accessed by clicking on *blog*.

The controller for the blog is controllers/blogcontroller.php. This is basically a class – BlogController that extends the base Controller class. The main( ) method selects all blog entries by default and sends the results to the View to be displayed. The common( ) method is called every time the Controller is instantiated, regardless of what other methods are called. This provides a way to set View header and footer data, and a common method between different actions of the same controller.

The other methods handle blog entries manipulation. For example, the edit( ) method opens a View where an entry is editable. After the entry is edited by the user, the edit( ) method receives data sent by the View using HTTP POST and updates the corresponding database entry. If we want to edit a inexistent entry then a 404 HTTP header will be sent by the controller, and the View will display “Not found”.

Likewise, the delete( ) method deletes a blog entry. It knows which entry to delete based on the HTTP GET request that it received from the View.

Views for the blog are stored in views/blog/. Each view is in a separate file which correspond to the BlogController methods. The header and footer are common across all views.

Conclusions

This web applications framework was developed mostly for educational purposes and for a deeper understanding how MVC works in today's production frameworks. However, it can be used as a skeleton to further develop the framework into a more powerful one, that has all features requested by today's developers.

References

[1] http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[2] http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[3] http://en.wikipedia.org/wiki/Observer_pattern

[4] http://en.wikipedia.org/wiki/Ruby_on_rails

[5] http://en.wikipedia.org/wiki/Javadoc

[6] http://mercurial.selenic.com/

References as available on 27 January 2010.