Content Management System for Tabletop

Author: Axel Ancona Esselmann, student at SFSU (axel.esselmann@gmail.com)
Date: 11/16/2015
Revised: 11/17/2015

This document is meant to serve as a tutorial for how to use our content management system. The first section describes the basic workings of the MVC pattern and how it is applied for our web application. Part two is comprised of an annotated view controller exemplifying the concepts discussed in part 1.

IMPORTANT DIRECTORIES:

```
public_html/src/htmlViewControllers    for html view controllers
public_html/src/jsonViewControllers    for json view controllers
public_html/src/app                    for models
public_html/templates/html             for template files
public_html/css                        for css files
public_html/js                         for js files
```

1) TUTORIAL:

Our content management system uses the **Model View Control** (MVC) pattern. Wikipedia has a very good synapses on this.

In short, complexity is tamed by very strictly defining who can talk to whom.

The model:

> This is the back end. This includes all of the data and logic associated with an object. An example for a model in our application is the `User`. The user is an abstraction that groups information related to a user with functionality acting on that information. Some of this functionality might be related to creating a user or recovering a password.
>
> Models don't display anything. Ever. The idea is to separate business logic from display logic.
>
> In our application all models should live in the application folder:
>
> ```
> public_html/src/app
> ```
>
> and be in the namespace app. Each model lives in it's own file. The class name should match the file name in order for auto-loading to work properly.

View Controllers:

> View controller talk to models through their API. This is where the US and German team will be interacting with each other for the majority of the project. View controllers are in charge of relaying the model's output to the view. You might ask why the model can't just talk straight to the view. This way we decouple the model from the display. The benefit is that we can easily reuse the same logic and data and present it in a different way by just swapping out the view controller. This is incredibly powerful.

View controllers live in the following directory:

`public_html/src/htmlViewControllers`	for html view controllers
`public_html/src/jsonViewControllers`	for json view controllers

The View:

Views in php applications are pretty dumb. The don't have any logic whatsoever. We pretty much just use straight up html templates in which we can insert data. Views or templates shouldn't even have any javascript or css in them, we keep that separate, but they can include files containing the former.
The benefits: We can develop one template and change it's appearance completely by just replacing the css script.

Templates live inside the following directory:

```
public_html/templates/html
```

css is stored in:

```
public_html/css
```

and javascript lives in

```
public_html/js
```

templates can have variables to insert other templates or strings. Variables have the following format:

```
{$varName}
```

To access that variable with a template, refer to it as “`varName`".

How it all comes together:
The magic lies in the index.php file at the web root and the little obscure `.htaccess` file. Inside the `.htaccess` file we make use of the rewrite engine to direct all requests to the `index.php` file. There the requests are parsed and the appropriate view controller is called.

When working with view controllers one often would like to perform an action on them. For example:

A request to the server might ask of the User to be created. Since we can't communicate directly with the model, we ask for the UserViewController and we perform the create action on it.

The request to the server then looks like this:

```
www.example.com/user/create
```

we might want to pass some arguments to the view controller, for example we might want to see the restaurant with id 13. We translate this into MVC speak: We want the RestaurantViewController, we want to perform the view action, and we give it the argument 13. As a complete url request that looks like this:

`www.example.com/restaurant/view/13`

Naming:

View controllers should end with "ViewController". Classes with that ending in the appropriate namespace make up the web api, meaning certain functions on those classes are are exposed to the web. The name of the view controller should start with an uppercase letter, but not have any other upper case letters, except for the ViewController ending. The reason for this is that the standard that defines urls defines them as case insensitive. Linux is case sensitive, which can create some weird inconsistent behavior when reusing code on other machines.

The functions on the view controllers that are exposed to the web also follow a specific naming convention:

All functions ending in "`Action`" are part of the web api. Everything before the "`Action`" ending should be lower case.

One special function exists with the name “`defaultAction`".

Create this one if you don't want to pass an action name.

View controllers are either `htmlViewControllers` or `jsonViewControllers`. content-type headers sent to the server determine which one is served. This also happens in the `index.php` file.

To see hot to make a json request, look at `js/test.js` and look at the json view controller `public_html/src/jsonViewControllers/TestViewController.php`

2)Example View controller

The following is an example view controller that demonstrates the CMS in action:
you can find this file on the server at `public_html/src/htmlViewControllers/TestViewController.`

```
namespace htmlViewControllers {
    /**
     * This view controller uses the following files:
     *
     *    public_html/css/main.css
     *    public_html/css/test.css
     *    public_html/js/main.js
     *    public_html/js/test.js
     *    public_html/templates/DefaultView.html
     *    public_html/templates/top.html
     *    public_html/templates/bottom.html
     *    public_html/templates/TestView.html
     *
     * @author Axel Ancona Esselmann
     */
```

```
class TestViewController {
    protected $model;

    public function __construct() {
        $this->template = new \aae\ui\HtmlTemplate(
            "templates/html",
            “DefaultView"
        );

        // css and js can be included into the project with setCss or setJs
        $this->template->setCss('css/main.css');
        $this->template->setCss('css/test.css');
        $this->template->setJs('js/main.js');
        $this->template->setJs('js/test.js');

        // here we are loading two templates into the main template.
        // DefaultView has three variables that HtmlTemplate can import into, top,
        // main and bottom. Since all views associated with this view controller
        // share the same top and bottom, we set them in the constructor.
        $this->template->set('top', $this->template->getTemplate('top'));
        $this->template->set('bottom', $this->template->getTemplate('bottom'));
    }
    /*
     *  All functions that end with "Action" are exposed to the web. Functions not
     *  ending in “Actions" are not part of the web API and can not be reached
     *  from the web.
     */

    /**
     * The defaultAction function is called when the view controller is called
     * without an action.
     * ex:
     *     For the url www.example.com this function executes when calling:
     *
     *     www.example.com/test/
     *
     */
    public function defaultAction() {
        // load the main template. The template is set up in UserViewController.
        $this->template->set('main', $this->template->getTemplate('TestView'));

        // Test view also has a variable: testVar1. The way we imported the
        // TestView we have to build the template first to be able to insert into
        // testVar1
        // After the template is built, css and js can not be added any more.
        $this->template->build();

        // we can set and build in one step with insert:
        $this->template->insert('testVar1', "bonjour");
        return (string)$this->template;
    }



    /**
     * This action is called when the url consists of a view controller and an
     * action with the name
     * test_function.
     *
     * The url that executes this action is:
     *
     *  www.example.com/test/test_function
     *
```

```
         */
        public function test_functionAction() {
            // top and bottom were already set in the view controller. Since the
            // template has not been built,
            // They can still be replaced. I am using insert, which means after the
            // insert the variable can not
            // be replaced any more.
            $this->template->insert('top', 'Hello from test_function. I was top');
            $this->template->insert(
                'bottom',
                'Hello from test_function, I was bottom.'
            );

            // after a variable has been inserted or set and build, changes will not
            // be visible any more.
            // This function call does nothing:
            $this->template->set('top', 'This will not show up.');

            // This one works though, since "main" has not been build yet.
            $this->template->set(
                'main',
                "test_function says: Hello from the html view controller!!!"
            );
            return (string)$this->template;
        }

        /*
         * This code is deployed on the server. Visit http://sfsuswe.com/~f15g09/test
         * to see it in action.
         *
         * If something doesn't make sense, please ask me!
         */
    }
}
```