John Squibb

Build a PHP MVC Framework in One Hour - Part Three: Additional suggestions, wrapping up.

If you haven't already completed parts One and Two of this series, I highly recommend doing so before continuing with this section.

At this point, you should have a basic framework established. Keep in mind, that you have a very basic system at this point, and a lot would have to be done to make this tutorial production worthy. Up to this point, we have made a lot of security concessions for the sake of speed and demonstration!

Below are some additional suggestions for rounding out some of the components we whipped up thus far.

Rounding out the View_Model

We demonstrated a very simple templating system in the first part of the tutorial. What if we wanted to have nested views, such as a header, footer, menu, or other all inside a master template? This is very common practice for web sites, and actually pretty easy to do with a couple of modifications to the View_Model.

It's been a while since we have worked with it, so let's recap the contents of the View_Model so far:

```php
<?php
/**
 * Handles the view functionality of our MVC framework
 */
class View_Model
{
    /**
     * Holds variables assigned to template
     */
    private $data = array();

    /**
     * Holds render status of view.
     */
    private $render = FALSE;

    /**
     * Accept a template to load
     */
    public function __construct($template)
    {
        //compose file name
        $file = SERVER_ROOT . '/views/' . strtolower($template) . '.php';

        if (file_exists($file))
        {
            /**
             * trigger render to include file when this model is destroyed
             * if we render it now, we wouldn't be able to assign variables
             * to the view!
```

```
             */
            $this->render = $file;
        }
    }

    /**
     * Receives assignments from controller and stores in local data array
     *
     * @param $variable
     * @param $value
     */
    public function assign($variable , $value)
    {
        $this->data[$variable] = $value;
    }

    public function __destruct()
    {
        //parse data variables into local variables, so that they render to the view
        $data = $this->data;

        //render view
        include($this->render);
    }
}
```

Toggle plain-text

First, we would want to move the render logic out of the __destruct method into another method, such as render(), or display(), etc. This way, we could call that method directly, and optionally, still have __destruct invoke that method.

After moving it, we would have two options for controlling the render output:

1. Have render() accept an optional argument that dictates whether content is output or returned (more on this later)
2. Have a separate method, such as get_content() that returns the generated output rather than displaying it directly to the web page.

Using either method and a little output buffering, we can begin to nest templates into a single master template. If you haven't worked with output buffering before, I suggest reading up on it.

I like method one, because it means less code duplication, so let's go with that.

Modify the contents of the View_Model so that it looks like this:

```
<?php
/**
 * Handles the view functionality of our MVC framework
 */
class View_Model
{
    /**
     * Holds variables assigned to template
     */
    private $data = array();

    /**
     * Holds render status of view.
     */
    private $render = FALSE;

    /**
     * Accept a template to load
     */
    public function __construct($template)
```

```
    {
        //compose file name
        $file = SERVER_ROOT . '/views/' . strtolower($template) . '.php';

        if (file_exists($file))
        {
            /**
             * trigger render to include file when this model is destroyed
             * if we render it now, we wouldn't be able to assign variables
             * to the view!
             */
            $this->render = $file;
        }
    }

    /**
     * Receives assignments from controller and stores in local data array
     *
     * @param $variable
     * @param $value
     */
    public function assign($variable , $value)
    {
        $this->data[$variable] = $value;
    }

    /**
     * Render the output directly to the page, or optionally, return the
     * generated output to caller.
     *
     * @param $direct_output Set to any non-TRUE value to have the
     * output returned rather than displayed directly.
     */
    public function render($direct_output = TRUE)
    {
        // Turn output buffering on, capturing all output
        if ($direct_output !== TRUE)
        {
            ob_start();
        }

        // Parse data variables into local variables
        $data = $this->data;

        // Get template
        include($this->render);

        // Get the contents of the buffer and return it
        if ($direct_output !== TRUE)
        {
            return ob_get_clean();
        }
    }

    public function __destruct()
    {
    }
}
```

Toggle plain-text

Now, since we have direct control over the render methods, we can establish views for each section and render them as needed:

```
<?php
// Now we can nest our templates using multiple views
$header = new View_Model('header_template');
$footer = new View_Model('footer_template');
$master = new View_Model('master_template');
$master->assign('header', $header->render(FALSE));
$master->assign('footer', $footer->render(FALSE));
$master->render();
```

Toggle plain-text

In the example above, we have a master view that receives the output of a header and footer view. To display this, we could use the following master HTML/PHP template:

```
<?php
// Master Template with header, footer nested.
?>
<html>
    <head></head>
    <body>
    <?=$data['header'];?>

    ...body stuff...

    <?=$data['footer'];?>
    </body>
</html>
```

Toggle plain-text

That's all there is to it! You can create as many different regions as you need, and assign them all to the master template as necessary.

Download Source

You can download the source files from the following links. They contain the full framework from parts 1 and 2 of this tutorial, with the modifications made in this section of the tutorial.

- .zip (click to download...)
- .tar.gz (click to download...)

Tags: PHP,MVC,framework,one hour,part three,wrapping up, additional thoughts
Short URL: http://sqb.in/D2tq