# *MVC in PHP*

This article series demonstrates how to build an MVC web framework using PHP 5. This article covers the basics of MVC web frameworks, building the foundation classes for a framework that the other three articles in this series will build.

With the introduction of PHP 5 and its new OOP features developers can now seriously talk about building solid APIs and more complex MVC frameworks for the web in PHP. This was possible before, but the new features in PHP 5 make it easier to integrate more advanced features into MVC frameworks, such as SOAP and WSDL.

## What is MVC ?

**MVC** is the idea that you have three different pieces that work in unison to form a complex application. A car is a good real-world example of MVC. With a car you have two views: the interior and the exterior. Both take input from the controller: the driver. The brakes, steering wheel and other controls represent the model: they take input from the controller (driver) and hand them off to the views (interior/exterior) for presentation.

## MVC on the Web

The ideas behind MVC frameworks are quite simple and extremely flexible. The idea is that you have a single controller (such as *index.php*) that controls the launch of applications within the framework based on arguments in the request. This usually includes, at a minimum, an argument defining which model to invoke, an event, and the usual GET arguments. From there the controller validates the request (authentication, valid model, request sanitization, etc.) and runs the requested event.

For instance, a request for /index.php?module=foo&event=bar might load a class called foo and run foo::bar(). The advantages of this method include:

- A single entry point for all applications.
- Removing the headaches involved with maintaining numerous scripts, each with their own relative paths, database connections, authentication, etc.
- Allowing the consolidation and reuse of code.

## Why Create My Own MVC Framework ?

As of this writing, there are very few true MVC frameworks written in PHP. In fact, there is only one that I know of, Solar, that is entirely pure PHP 5 code. Another one out there is Cake, which is trying to be the "Ruby on Rails of PHP." I, personally, have a few problems with both of these frameworks. Both Solar and Cake fail to leverage existing code in PEAR, Smarty, etc. Cake appears a bit disorganized at the moment. Finally, Solar is the work of mostly a single person (not that Paul isn't a great coder or person, but there is only a single gatekeeper at the time of this writing). These may not be issues that concern you, and if they don't concern you, by all means check these two out.

## The Old Way

If I could go back in time and look at code I wrote in early 2001, I would find a file called *template.txt* that looked something like:

```
    <?php

    require_once('config.php'); // Other requires, DB info, etc.

    $APP_DB = 'mydb';
    $APP_REQUIRE_LOGIN = false; // Set to true if script requires login
    $APP_TEMPLATE_FILE = 'foo.php'; // Smarty template
    $APP_TITLE = 'My Application';

    if ($APP_REQUIRE_LOGIN == true) {
        if (!isset($_SESSION['userID'])) {
            header("Location: /path/to/login.php");
            exit();
        }
    }

    $db =
DB::connect('mysql://'.$DB_USER.':'.$DB_PASS.'@localhost/'.$APP_DB);
    if (!PEAR::isError($db)) {
        $db->setFetchMode(DB_FETCHMODE_ASSOC);
    } else {
        die($db->getMessage());
    }

    // Put your logic here

    // Output the template
    include_once(APP_TEMPLATE_PATH.'/header.php');
    include_once(APP_TEMPLATE_PATH.'/'.$APP_TEMPLATE_FILE);
    include_once(APP_TEMPLATE_PATH.'/footer.php');

    ?>
```

Oh man, just looking at this code makes me cringe now. The idea with this approach was that every application fit into this set approach and I could just copy *template.txt* to *myapp.php*, change some of the variables, and then voila, it would work. However, this top-down approach has some serious flaws.

1. What if my boss wanted me to change *myapp.php* to output a PDF in some cases, HTML in others, and SOAP if the request posted XML directly?
2. What if this app required IMAP or LDAP authentication?
3. How would I go about handling various modes in the script (including edit, update, and delete)?
4. How would I handle multi-level authentication (admin versus non-admin)?
5. How would I turn on output caching?

## The New Way

By bringing everything into an MVC framework, I could make my life a lot easier. Compare the following code:

```
<?php

class myapp extends FR_Auth_User
{
    public function __construct()
    {
        parent::__construct();
    }

    public function __default()
    {
        // Do something here
    }

    public function delete()
    {

    }

    public function __destruct()
    {
        parent::__destruct();
    }
}

?>
```

Notice that this code has no apparent concern with connecting to a database, verifying the user is logged in, or outputting anything. The controller handles all of this.

If I wanted to authenticate against LDAP, I could create FR_Auth_LDAP. The controller could recognize certain output methods (such as $_GET['output']) and switch to the PDF or SOAP presenter on the fly. The event handler, delete, handles only deleting and nothing else. Because the module has an instance of the FR_User class, it's easy to check which groups that user is in, etc. Smarty, the template engine, handles caching, of course, but the controller could also handle some caching.

Switching from the old way to the MVC way of doing things can be a completely foreign concept to some people, but once you have switched, it's hard to go back. I know I won't be leaving the comforts of my MVC framework anytime soon.

## Creating the Foundation

I'm a huge fan of PEAR and the PEAR_Error class. PHP 5 introduced a new class, Exception, which is almost a drop-in replacement for PEAR_Error. However, PEAR_Error has a few extra features that make it a more robust solution than Exception. As a result, the framework and foundation classes will use the PEAR_Error class for error handling. I will use Exception, however, to throw errors from the constructors, as they cannot return errors.

The design goals of the foundation classes are:

- Leverage PEAR to quickly add features to the foundation classes.
- Create small, reusable abstract foundation classes that will enable developers to build applications quickly within the framework.
- Document all foundation classes using phpDocumentor tags.
- Prepend all classes and global variables with FR to avoid possible variable/class/function collisions.

The class hierarchy will look something like this:

- **FR_Object** will provide the ***basic features that all objects*** need (including logging, generic setFrom(), toArray()).
    - **FR_Object_DB** is a thin layer to ***provide child classes a database connection,*** along with other functions such classes might need.
        - **FR_Object_Web** is a ***thin layer that provides session and user information*** for web-based applications.
            - **FR_Module** is the ***base class for all applications*** (AKA "modules," "model," etc.) built in the framework.
                - **FR_Auth** is the ***base authentication class***, which will allow for multiple authentication mechanisms.
                    - **FR_Auth_User** is an ***authentication class*** to use in modules that require a valid, logged-in user.
                    - **FR_Auth_No** is a ***dummy authentication class*** used for modules that require no authentication.
    - **FR_Presenter** is the ***base presentation class*** (the ***VIEW***) that will ***handle loading and displaying*** the applications after they have run.
        - **FR_Presenter_smarty**: the ***presentation layer*** will include the ability to load different drivers. Smarty is a great template class that has built in caching and an active community.
        - **FR_Presenter_debug** is a ***debugging presentation layer*** that developers can use to debug applications.
        - **FR_Presenter_rest** is a ***REST presentation layer*** that developers can use to output applications in XML.

Looking at the foundation classes, you can start to see the separate parts of the MVC framework. FR_Module provides for the basic needs of all of the modules (Model), FR_Presenter provides a way to display applications arbitrarily in different formats (Views). In the next part of this series I will create the controller, which will bring all of our foundation classes together in a single place.

## Coding Standards

Before you start coding a cohesive framework, you might want to sit down with your team (or yourself) and talk about coding standards. The whole idea of MVC programming revolves around reusable and standardized code. I recommend talking about, at least:

- What are your coding standards regarding variable naming and indentation? Don't start a holy war, but hammer out the basics and stick to them, especially when it comes to your foundation classes.
- Decide on a standard prefix for your functions, classes, and global variables. Unfortunately, PHP does not support namespaces. As a result, it might be a good idea to prepend your variables to avoid name collisions and confusion. Throughout this article, I've prepended my global variables, functions and classes with FR_, so as to distinguish core foundation code from simple application code.
- I highly recommend using phpDocumentor to document your code as you actually code it. I will document all of the core foundation classes as well as my initial applications in this article. At my own place of employment, I run phpDocumentor via a cron job to compile documentation frequently from my code repository.

## Coding the Foundation

With all of that theory out of the way, here are the foundation classes. Be sure to read the comments for my reasonings, ideas, and implementation details. I'm presenting a combination of things I've done in the past that work for me, and the results of a few years of trial and error. By no means is this the only way to program an MVC framework, but I think it provides a good overview of how things should work.

## Filesystem Layout

The basic layout is simple and somewhat strictly defined. There is a directory for includes, which will follow a specific pattern to make it easy to use PHP's new __autoload() function. Another directory is for modules, which will have their own layout. The hierarchy looks like:

- */*
    - *config.php*
    - *index.php*
    - *includes/*
        - *Auth.php*
        - *Auth/*
            - *No.php*
            - *User.php*
        - *Module.php*
        - *Object.php*
        - *Object/*
            - *DB.php*
        - *Presenter.php*
        - *Presenter/*
            - *common.php*
            - *debug.php*
            - *smarty.php*
        - *Smarty/*
    - *modules/*
        - *example/*
            - *config.php*
            - *example.php*
            - *tpl/*
                - *example.tpl*
    - *tpl/*
        - *default/*
        - *cache/*
        - *config/*
        - *templates/*
            - *templates_c/*

You're probably thinking that's a lot of code! It is, but you'll get through it. At the end of this article and the series, you'll see that MVC programming will make your life a lot easier and speed up development time.

In the filesystem structure, all of the foundation classes live inside of *includes/*. The example laid out a sample module, as well. Each module has its own configuration file, at least one module file, and one template file. All modules reside in *modules/*. I've become accustomed to wrapping my modules in an outer-page template, which is what the *tpl/* directory is for. Each "theme" or template group has its own template directory. For now, I'm going to use *default/* as my outer-page template. Later I'll show how to create a presentation layer for modules that want to render themselves.

**config.php** - provides a ***centralized location for global configuration variables***, such as the DSN and log file location. Also, notice that I dynamically figure out the installation location on the file system. This will make installing and migrating your code simple if you use **FR_BASE_PATH** in your own code.

**index.php** - This is the ***controller.***

## Object.php

This is the ***base class*** for all of the ***foundation classes***. It provides some basic features that most, if not all, classes will need. Additionally, the child class **FR_Object_DB** extends this object and provides a database connection.

The idea is that, by having all children extend from a central object, all of the foundation classes will share certain characteristics. You could put the database connection directly into **FR_Object**, but not all classes need a database connection. I will talk about FR_Object_DB later.

```
<?php

  require_once('Log.php');

  /**
  * FR_Object
  *
  * The base object class for most of the classes that we use in our
framework.
  * Provides basic logging and set/get functionality.
  *
  * @author Joe Stump <joe@joestump.net>
  * @package Framework
  */
  abstract class FR_Object
  {
      /**
      * $log
      *
      * @var mixed $log Instance of PEAR Log
      */
      protected $log;

      /**
      * $me
      *
      * @var mixed $me Instance of ReflectionClass
      */
      protected $me;

      /**
      * __construct
      *
      * @author Joe Stump <joe@joestump.net>
      * @access public
      */
      public function __construct()
      {
          $this->log = Log::factory('file',FR_LOG_FILE);
          $this->me = new ReflectionClass($this);
      }

      /**
      * setFrom
      *
      * @author Joe Stump <joe@joestump.net>
      * @access public
      * @param mixed $data Array of variables to assign to instance
      * @return void
      */
      public function setFrom($data)
      {
          if (is_array($data) && count($data)) {
              $valid = get_class_vars(get_class($this));
```

```
                foreach ($valid as $var => $val) {
                    if (isset($data[$var])) {
                        $this->$var = $data[$var];
                    }
                }
            }
        }

        /**
        * toArray
        *
        * @author Joe Stump <joe@joestump.net>
        * @access public
        * @return mixed Array of member variables keyed by variable name
        */
        public function toArray()
        {
            $defaults = $this->me->getDefaultProperties();
            $return = array();
            foreach ($defaults as $var => $val) {
                if ($this->$var instanceof FR_Object) {
                    $return[$var] = $this->$var->toArray();
                } else {
                    $return[$var] = $this->$var;
                }
            }

            return $return;
        }

        /**
        * __destruct
        *
        * @author Joe Stump <joe@joestump.net>
        * @access public
        * @return void
        */
        public function __destruct()
        {
            if ($this->log instanceof Log) {
                $this->log->close();
            }
        }
    }

?>
```

## Auth.php

This is the ***base class for authentication***. It ***extends*** from the **FR_Module** class from ***Module.php***. Its major function is to define how a basic authentication class should behave.

An alternate approach is to define this as a variable in the module and then have the controller create the authentication module through a factory pattern. However, this way works too (and is simpler to explain).

Child classes should override the ***authenticate() method***. The **controller** will use this method when determining if a user has access to the given module. For instance, the **FR_Auth_No class** simply returns true, which allows you to create modules that require no authentication.

```
<?php

  abstract class FR_Auth extends FR_Module
  {
      // {{{ __construct()
      function __construct()
      {
          parent::__construct();
      }
      // }}}
      // {{{ authenticate()
      abstract function authenticate();
      // }}}
      // {{{ __destruct()
      function __destruct()
      {
          parent::__destruct();
      }
      // }}}
  }

?>
```

## Module.php

This is the ***heart of all of the modules***. It extends the **FR_Object_DB** class and provides all of its children with database access and an open log file.

Additionally, it defines the ***default presentation layer***, the default template file for the module, default page template file, and a few other variables that the controller and presentation layer use.

The class also provides the basic structure of what each module must possess as far as functions. The function set() abstracts the method of setting data into a centralized place, which the getData() function then hands off to the presentation layer for rendering.

```
<?php

  abstract class FR_Module extends FR_Object_Web
  {
      // {{{ properties
      /**
      * $presenter
      *
      * Used in FR_Presenter::factory() to determine which presentation
(view)
      * class should be used for the module.
      *
      * @author Joe Stump <joe@joestump.net>
      * @var string $presenter
      * @see FR_Presenter, FR_Presenter_common, FR_Presenter_smarty
      */
      public $presenter = 'smarty';

      /**
      * $data
      *
      * Data set by the module that will eventually be passed to the
view.
      *
      * @author Joe Stump <joe@joestump.net>
      * @var mixed $data Module data
      * @see FR_Module::set(), FR_Module::getData()
      */
      protected $data = array();

      /**
      * $name
      *
      * @author Joe Stump <joe@joestump.net>
      * @var string $name Name of module class
      */
      public $name;

      /**
      * $tplFile
      *
      * @author Joe Stump <joe@joestump.net>
      * @var string $tplFile Name of template file
      * @see FR_Presenter_smarty
      */
      public $tplFile;

      /**
      * $moduleName
      *
      * @author Joe Stump <joe@joestump.net>
```

```
         * @var string $moduleName Name of requested module
         * @see FR_Presenter_smarty
         */
         public $moduleName = null;

         /**
         * $pageTemplateFile
         *
         * @author Joe Stump <joe@joestump.net>
         * @var string $pageTemplateFile Name of outer page template
         */
         public $pageTemplateFile = null;
         // }}}
         // {{{ __construct()
         /**
         * __construct
         *
         * @author Joe Stump <joe@joestump.net>
         */
         public function __construct()
         {
             parent::__construct();
             $this->name = $this->me->getName();
             $this->tplFile = $this->name.'.tpl';
         }
         // }}}
         // {{{ __default()
         /**
         * __default
         *
         * This function is ran by the controller if an event is not
specified
         * in the user's request.
         *
         * @author Joe Stump <joe@joestump.net>
         */
         abstract public function __default();
         // }}}
         // {{{ set($var,$val)
         /**
         * set
         *
         * Set data for your module. This will eventually be passed toe the
         * presenter class via FR_Module::getData().
         *
         * @author Joe Stump <joe@joestump.net>
         * @param string $var Name of variable
         * @param mixed $val Value of variable
         * @return void
         * @see FR_Module::getData()
         */
         protected function set($var,$val) {
             $this->data[$var] = $val;
         }
         // }}}
         // {{{ getData()
         /**
         * getData
         *
         * Returns module's data.
         *
         * @author Joe Stump <joe@joestump.net>
         * @return mixed
         * @see FR_Presenter_common
         */
         public function getData()
         {
             return $this->data;
         }
         // }}}
         // {{{ isValid($module)
         /**
         * isValid
         *
         * Determines if $module is a valid framework module. This is used
by
         * the controller to determine if the module fits into our
framework's
         * mold. If it extends from both FR_Module and FR_Auth then it
should be
         * good to run.
         *
         * @author Joe Stump <joe@joestump.net>
         * @static
         * @param mixed $module
         * @return bool
         */
         public static function isValid($module)
         {
             return (is_object($module) &&
                     $module instanceof FR_Module &&
                     $module instanceof FR_Auth);
         }
         // }}}
         // {{{ __destruct()
         public function __destruct()
         {
             parent::__destruct();
         }
         // }}}
     }

   ?>
```

## Presenter.php

This is the ***foundation for the presentation layer***. It uses the ***factory design pattern*** to create the presentation layer.

***FR_Module::$presenter*** defines ***which presentation layer*** to use, ***which the controller*** will then create via the factory method. Once the controller has a valid presentation layer, the only thing left to do is to run the common display() function, which presentation classes inherit from **FR_Presenter_common**.

```
<?php

  class FR_Presenter
  {
      // {{{ factory($type,FR_Module $module)
      /**
      * factory
      *
      * @author Joe Stump <joe@joestump.net>
      * @access public
      * @param string $type Presentation type (our view)
      * @param mixed $module Our module, which the presenter will display
      * @return mixed PEAR_Error on failure or a valid presenter
      * @static
      */
      static public function factory($type,FR_Module $module)
      {
          $file = FR_BASE_PATH.'/includes/Presenter/'.$type.'.php';
          if (include($file)) {
              $class = 'FR_Presenter_'.$type;
              if (class_exists($class)) {
                  $presenter = new $class($module);
                  if ($presenter instanceof FR_Presenter_common) {
                      return $presenter;
                  }

                  return PEAR::raiseError('Invalid presentation class:
'.$type);
              }

              return PEAR::raiseError('Presentation class not found:
'.$type);
          }

          return PEAR::raiseError('Presenter file not found: '.$type);
      }
      // }}}
  }

?>
```

# ***Implementing MVC in PHP***

# ***The View***

The ***presentation layer***, as I call it, is the ***View***, in common ***MVC*** terms. Its sole ***responsibility is to display information***. It could care less about authenticating users, what the data is or, for the most part, where it came from. The only thing it has to worry about is how to render it and where to send it once rendered.

By default, the framework uses ***Smarty*** to render the framework. I'm not here to argue semantics, but your presentation layer should consist of a template engine of some sort and a few supporting presentation layers.

The idea is that, after the Model runs, the framework hands it off to a child of the **FR_Presenter**_common class.

Actually, the framework uses the **FR_Presenter::factory()** to create the presenter. Each presenter should have a display() method that does the actual rendering. When the factory creates the presenter, it passes the presenter the instance of the model class. The presenter then gets the model's data using its getData() method. From there, the presenter is free to present that data however it sees fit.

## FR_Presenter_smarty

The way I've created my Smarty presenter is a hybrid of two templates. I create a Smarty template, and the outer page template includes the model's template. The model class's $pageTemplateFile can request a particular outer template. If it does not, the default is *tpl/default/templates/page.tpl*. The *page.tpl* template then uses the {$modulePath} and {$tplFile} directives to include the model's template. All model templates should reside in *modules/example/tpl/*.

After assigning the variables, the controller runs Smarty's display function to render the templates. With little modification, you could wrap these calls with Smarty's built-in caching as well. By using Smarty, you could enable an output modifier to output gzipped code instead of plain HTML.

```
<?php

  /**
  * FR_Presenter_smarty
  *
  * @author Joe Stump <joe@joestump.net>
  * @copyright Joe Stump <joe@joestump.net>
  * @license http://www.opensource.org/licenses/gpl-license.php
  * @package Framework
  * @filesource
  */

  require_once(SMARTY_DIR.'Smarty.class.php');

  /**
  * FR_Presenter_smarty
  *
  * By default we use Smarty as our websites presentation layer (view).
Smarty
  * is a robust compiling template engine with an active community.
  *
  * @author Joe Stump <joe@joestump.net>
  * @package Framework
  * @link http://smarty.php.net
  */
  class FR_Presenter_smarty extends FR_Presenter_common
  {
      private $template = null;
      private $path = null;

      public function __construct(FR_Module $module)
      {
          parent::__construct($module);
          $this->path = FR_BASE_PATH.'/tpl/'.FR_TEMPLATE;
          $this->template = new Smarty();
          $this->template->template_dir = $this->path.'/'.'templates';
          $this->template->compile_dir = $this->path.'/'.'templates_c';
          $this->template->cache_dir = $this->path.'/'.'cache';
          $this->template->config_dir = $this->path.'/'.'config';
      }

      public function display()
      {
          $path = FR_BASE_PATH.'/modules/'.$this->module-
>moduleName.'/tpl';;
          $tplFile = $this->module->tplFile;

          $this->template->assign('modulePath',$path);
          $this->template->assign('tplFile',$tplFile);
          $this->template->assign('user',$this->user);
          $this->template->assign('session',$this->session);

          foreach ($this->module->getData() as $var => $val) {
              if (!in_array($var,array('path','tplFile'))) {
                  $this->template->assign($var,$val);
              }
          }

          if ($this->module->pageTemplateFile == null) {
              $pageTemplateFile = 'page.tpl';
          } else {
              $pageTemplateFile = $this->module->pageTemplateFile;
          }

          $this->template->display($pageTemplateFile);
      }

      public function __destruct()
      {
          parent::__destruct();
      }
  }

?>
```

## Other Presenters

You can create other presenters, as well. I've created one called ***debug.php*** that simply displays various debugging information. You could change your model class's $presenter to debug and it would render completely differently.

Additionally, you could create a presentation layer called ***rest.php*** that outputs the model class's $data variable as well-formed XML. If your model detected a REST request, it would switch the presenter by assigning rest to $this->presenter.

```
    <?php

      /**
      * FR_Presenter_rest
      *
      * @author Joe Stump <joe@joestump.net>
      * @copyright Joe Stump <joe@joestump.net>
      * @license http://www.opensource.org/licenses/gpl-license.php
      * @package Framework
      * @filesource
      */

      require_once('XML/Serializer.php');

      /**
      * FR_Presenter_rest
      *
      * Want to display your module's data in valid XML rather than HTML?
This
      * presenter will automatically take your data and output it in valid
XML.
      *
      * @author Joe Stump <joe@joestump.net>
      * @package Framework
      */
      class FR_Presenter_rest extends FR_Presenter_common
      {
          // {{{ __construct(FR_Module $module)
          /**
          * __construct
          *
          * @author Joe Stump <joe@joestump.net>
          * @access public
          * @param mixed $module Instance of FR_Module
          * @return void
          */
          public function __construct(FR_Module $module)
          {
              parent::__construct($module);
          }
          // }}}
          // {{{ display()
          /**
          * display
          *
          * Output our data array using the PEAR package XML_Serializer. This
may
          * not be the optimal output you want for your REST API, but it
should
          * display valid XML that can be easily consumed by anyone.
          *
          * @author Joe Stump <joe@joestump.net>
          * @return void
          * @link http://pear.php.net/package/XML_Serializer
```

```
        */
        public function display()
        {
            $xml = new XML_Serializer();
            $xml->serialize($this->module->getData());

            header("Content-Type: text/xml");
            echo '<?xml version="1.0" encoding="UTF-8" ?>'."\n";
            echo $xml->getSerializedData();
        }
        // }}}
        // {{{ __destruct()
        public function __destruct()
        {
            parent::__destruct();
        }
        // }}}
    }

?>
```

In the ***REST presentation layer*** I use the PEAR package XML_Serializer to output the **FR_Module::$data** array as valid XML. It's not extremely intuitive, but it does allow me to output my module in valid XML. I use this REST presentation layer later on in one of my applications.

## Conclusion

You get the idea. The presentation layer is an extremely flexible way of displaying the model. To make things even better you can dynamically switch the presenter in the model before the controller renders the module class via the presentation layer.

Before I move on, I'd like to plant the seed for another presentation layer. How about using htmldoc in a presenter named pdf to render your module class as a PDF document?

# ***Implementing MVC in PHP***

# ***The Model***

***The Model*** is where the majority of the application's logic sits. It is where you run queries against the database and perform calculations on input. A good example of what a Model would look like is a simple login script. The login script gets user input from a form, validates it against the database, and then logs in the user.

The first application using the newly created framework will be the users module. I will be creating three Models:

- *login.php* validates input from a form against the database
- *logout.php* logs a user out and destroys the associated session
- *whoami.php* displays simple user information, similar to the Unix program of the same name

Because I am introducing the idea of sessions and users into the framework, I will need to create a few more foundation classes as well as a table in a database. Before I go over the code of *login.php*, I'd like to walk through these classes.

## FR_Session

**FR_Session *is a wrapper*** for the built-in PHP sessions. The code isn't very involved and provides only basic support for starting, destroying, and writing sessions.

```
<?php

  /**
  * FR_Session
  *
  * @author Joe Stump <joe@joestump.net>
  * @copyright Joe Stump <joe@joestump.net>
  * @license http://www.opensource.org/licenses/gpl-license.php
  * @package Framework
  * @filesource
  */

  /**
  * FR_Session
  *
  * Our base session class as a singleton. Handles creating the session,
  * writing to the session variable (via overloading) and destroying the
  * session.
  *
  * @author Joe Stump <joe@joestump.net>
  * @package Framework
  */
  class FR_Session
  {
      /**
      * $instance
      *
      * Instance variable used for singleton pattern. Stores a single
instance
      * of FR_Session.
      *
```

```
         * @author Joe Stump <joe@joestump.net>
         * @var mixed $instance
         */
         private static $instance;

         /**
         * $sessionID
         *
         * The session ID assigned by PHP (usually a 32 character alpha-
numeric
         * string).
         *
         * @author Joe Stump <joe@joestump.net>
         * @var string $sessionID
         */
         public static $sessionID;

         // {{{ __construct()
         /**
         * __construct
         *
         * Starts the session and sets the sessionID for the class.
         *
         * @author Joe Stump <joe@joestump.net>
         */
         private function __construct()
         {
             session_start();
             self::$sessionID = session_id();
         }
         // }}}
         // {{{ singleton()
         /**
         * singleton
         *
         * Implementation of the singleton pattern. Returns a sincle
instance
         * of the session class.
         *
         * @author Joe Stump <joe@joestump.net>
         * @return mixed Instance of session
         */
         public static function singleton()
         {
             if (!isset(self::$instance)) {
                 $className = __CLASS__;
                 self::$instance = new $className;
             }

             return self::$instance;
         }
         // }}}
         // {{{ destroy()
         public function destroy()
         {
             foreach ($_SESSION as $var => $val) {
                 $_SESSION[$var] = null;
             }

             session_destroy();
         }
         // }}}
         // {{{ __clone()
         /**
         * __clone
         *
         * Disable PHP5's cloning method for session so people can't make
copies
```

```
         * of the session instance.
         *
         * @author Joe Stump <joe@joestump.net>
         */
         public function __clone()
         {
             trigger_error('Clone is not allowed for
'.__CLASS__,E_USER_ERROR);
         }
         // }}}
         // {{{ __get($var)
         /**
         * __get($var)
         *
         * Returns the requested session variable.
         *
         * @author Joe Stump <joe@joestump.net>
         * @return mixed
         * @see FR_Session::__get()
         */
         public function __get($var)
         {
             return $_SESSION[$var];
         }
         // }}}
         // {{{ __set($var,$val)
         /**
         * __set
         *
         * Using PHP5's overloading for setting and getting variables we can
         * use $session->var = $val and have it stored in the $_SESSION
         * variable. To set an email address, for instance you would do the
         * following:
         *
         * <code>
         * $session->email = 'user@example.com';
         * </code>
         *
         * This doesn't actually store 'user@example.com' into $session-
>email,
         * rather it is stored in $_SESSION['email'].
         *
         * @author Joe Stump <joe@joestump.net>
         * @param string $var
         * @param mixed $val
         * @see FR_Session::__get()
         * @link http://us3.php.net/manual/en/language.oop5.overloading.php
         */
         public function __set($var,$val)
         {
             return ($_SESSION[$var] = $val);
         }
         // }}}
         // {{{ __destruct()
         /**
         * __destruct()
         *
         * Writes the current session.
         *
         * @author Joe Stump <joe@joestump.net>
         */
         public function __destruct()
         {
             session_write_close();
         }
         // }}}
     }
   ?>
```

## FR_User

**FR_User** is a ***basic data object*** in the users table. I created a single table called fr_users in a MySQL database called framework:

```
CREATE TABLE `fr_users` (
  `userID`    int(11) unsigned NOT NULL auto_increment,
  `email`    char(45) NOT NULL         default '',
  `password` char(15) NOT NULL         default '',
  `fname`    char(30) NOT NULL         default '',
  `lname`    char(30) NOT NULL         default '',
  `posted`   datetime NOT NULL         default '0000-00-00 00:00:00',
  `status`   enum('active','disabled') default 'active',
  PRIMARY KEY  (`userID`),
  UNIQUE KEY `email` (`email`),
  KEY `posted` (`posted`),
  KEY `status` (`status`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- Special user for anonymous/not-logged-in users
INSERT INTO `fr_users` VALUES
(0,'anon@example.com','','Anonymous','Coward', \
    '2005-06-05 11:33:56','disabled');

<?php

  /**
  * FR_User
  *
  * @author Joe Stump <joe@joestump.net>
  * @copyright Joe Stump <joe@joestump.net>
  * @license http://www.opensource.org/licenses/gpl-license.php
  * @package Framework
  * @filesource
  */

  /**
  * FR_User
  *
  * Base user object.
  *
  * @author Joe Stump <joe@joestump.net>
  * @package Framework
  */
  class FR_User extends FR_Object_DB
  {
      public $userID;
      public $email;
      public $password;
      public $fname;
      public $lname;
      public $posted;

      public function __construct($userID=null)
      {
          parent::__construct();

          if ($userID === null) {
              $session = FR_Session::singleton();
              if (!is_numeric($session->userID)) {
                  $userID = 0;
              } else {
                  $userID = $session->userID;
              }
          }
```

```
            $sql = "SELECT *
                    FROM fr_users
                    WHERE userID=".$userID." AND
                          status='active'";

            $result = $this->db->getRow($sql);
            if (!PEAR::isError($result) && is_array($result)) {
                $this->setFrom($result);
            }
        }

        public function __destruct()
        {
            parent::__destruct();
        }
    }

?>
```

Notice that the constructor looks around for user IDs in various forms. If it receives a user ID, the class fetches the given user ID. If it does not receive a user ID, it checks the session; if a user ID is not present, it assumes the user is not logged in and it loads the special anonymous user record.

The special anonymous user allows me to have a special user for people not logged in. It allows me to easily and quickly check whether a user is logged in:

```
if ($this->user->userID > 0) {
    // User is logged in
}
```

I would have much rather made **FR_User** a **singleton**; however, PHP5 does not allow a child class to have a private constructor when the parent class, **FR_Object_DB**, has a public constructor.

## FR_Auth_User

Along with a session class and a user object, the code needs a proper authentication class to applications for users who are logged in. As a result, I created a new class called FR_Auth_User and put it into *includes/Auth/User.php*.

Because my *login.php* module class sets a session variable called userID to the user ID of the person logged in and that value is numerical, it becomes quite easy to validate that the user is logged in. The user ID of zero indicates an anonymous surfer, which means that if the user ID is greater than zero, the person is logged in.

```
<?php

  /**
  * FR_Auth_User
  *
  * @author Joe Stump <joe@joestump.net>
  * @copyright Joe Stump <joe@joestump.net>
  * @license http://www.opensource.org/licenses/gpl-license.php
  * @package Framework
  * @filesource
  */

  /**
  * FR_Auth_User
  *
  * If your module class requires that a user be logged in in order to
access
  * it then extend it from this Auth class.
  *
  * @author Joe Stump <joe@joestump.net>
  * @package Framework
  */
  abstract class FR_Auth_User extends FR_Auth
  {
      function __construct()
      {
          parent::__construct();
      }

      function authenticate()
      {
          return ($this->session->userID > 0);
      }

      function __destruct()
      {
          parent::__destruct();
      }
  }

?>
```

As you can see, the ***authenticate()*** method simply checks to make sure the session variable is greater than zero.

## login.php

That's everything necessary to create a program: a simple application that validates a login form. The minimum necessary files for creating an application in any module are:

- *login.php*, the Model. It houses all of the PHP code that validates the user, queries the database, and sets the appropriate session variables.
- *login.tpl*, the Smarty template used by the default presenter. It's just a file with a simple HTML form.

For this specific module class, I have created the module users in the *modules* directory. Thus the module class file is *modules/users/login.php* and the template file is *modules/users/tpl/login.tpl*.

[code for modules/users/tpl/login.php]

You may be thinking, *I read 8,000 words for this?* Consider what the framework allows you to do. When my module loads and I finally start programming, I don't have to worry about creating sessions, connecting to databases, opening log files, or authenticating. For once, I don't have to worry that I missed an include file or forgot a snippet of common code.

This, in a nutshell, is the beauty of MVC programming. Because the controller and views have already been programmed and are in working order, you can focus on what you get paid to do: creating applications. I remember the pain of creating large sed and awk scripts to change database connections, config file locations, and so on. I remember having a *header.php* and a *footer.php* at the top and bottom of every single script.

I remember having hundreds of files that ran a function check_user() or something like that. What if I had to change that function name or integrate a different authentication method? If I added another check for a different authentication method, then that code/check would be run in *every single file* on every request. Now I just create a new FR_Auth_MyAuth and change the applications that need that specific authentication to extend that auth class.

## logout.php

Create a user account in fr_users for yourself, and go to the login module and log in. If you have the mod_rewrite rules turned on, you can simply go to *http://example.com/users/login*. If you don't, then go to *http://example.com/index.php?module=users&class=login*. Once you have logged in, you should be redirected to the home page, which should now recognize that you are logged in.

Before you click on the Submit link, right-click on it and copy the link to your clipboard. I'll explain why in a moment. OK, now go ahead and click on the link labeled Submit, which will log you out. (The links assume you have the rewrite rules turned on.)

The logout module should destroy your session and then send you back to the home page, which should no longer show that you are logged in. Now copy and paste the logout URL, and try to go to the module again. You should see the error "You do not have access to the requested page!"

Because the class logout extends FR_Auth_User, it requires that the user be logged in. The function FR_Auth_User::authenticate() checks to make sure $this->session->userID is greater than zero. When the controller runs logout::authenticate() (inherited from FR_Auth_User) and a person is not logged in, the function returns false, which tells the controller to bomb out.

## whoami.php

As a small example of how you can combine the various pieces to the MVC puzzle, I created a simple module class called *whoami.php*. The script simply displays who the user is. However, it checks for a GET argument called output. If output is rest, the module switches to the FR_Presenter_rest view, which outputs the module's $data variable in valid XML using PEAR's XML_Serializer class.

```
<?php

  /**
  * whoami
  *
  * @author Joe Stump <joe@joestump.net>
  * @copyright Joe Stump <joe@joestump.net>
  * @license http://www.opensource.org/licenses/gpl-license.php
  * @package Modules
  * @filesource
  */

  /**
  * whoami
  *
  * @author Joe Stump <joe@joestump.net>
  * @package Modules
  */
  class whoami extends FR_Auth_User
  {
      public function __construct()
      {
          parent::__construct();
      }

      public function __default()
      {
          $this->set('user',$this->user);
          if ($_GET['output'] == 'rest') {
              $this->presenter = 'rest';
          }
      }

      public function __destruct()
      {
          parent::__destruct();
      }
  }

?>
```

### Conclusion

Some people think that MVC frameworks are heavy. That may be; however, my own MVC application run web sites that regularly receive more than 10 *million* page views per month. If the load gets too heavy, I simply turn on caching in my presentation layer.

The gains I have experienced in my development cycles vastly outweigh the perceived problem of heaviness. If programmed and documented properly, MVC frameworks reduce the time you spend debugging your code; they increase your productivity; and they greatly reduce the barrier to entry for new and junior programmers.