**YII CODING STANDARDS DOCUMENT FOR OPTISOL**

**1. Model View Controller**

**1.1. Model**

* It contains properties to represent the data
* It contains the business logic (e.g. validation rules) to ensure the represented data fulfills the design requirement.
* Use search method to implement the search functionality
* Should not use the $\_POST and $\_GET inside the model file
* Should not use any embedding HTML or other presentation code.
* Use relations (Belongs to,has\_one,Has\_many and many to many)

**1.2. Controller**

* may access $\_GET, $\_POST or $\_REQUEST and other PHP variables that represent user requests.
* may create model instances and manage their life cycles. For example, in a typical model update action, the controller may first create the model instance; then populate the model with the user input from $\_POST; after saving the model successfully, the controller may redirect the user browser to the model detail page.
* should avoid containing any HTML or any other presentational markup. This is better kept in views.

**1.3 View**

* should mainly contain presentational code, such as HTML, and simple PHP code to traverse, format and render data.
* should avoid direct access to $\_GET, $\_POST, or other similar variables that represent the end user request. This is the controller's job. The view should be focused on the display and layout of the data provided to it by the controller and/or model, but not attempting to access request variables or the database directly.
* may access properties and methods of controllers and models directly. However, this should be done only for the purpose of presentation.
* Layout: common presentational areas (e.g. page header, footer) can be put in a layout view.
* Partial views: use partial views (views that are not decorated by layouts) to reuse fragments of presentational code. For example, we use \_form.php partial view to render the model input form that is used in both model creation and updating pages.
* [CHtml](http://www.yiiframework.com/doc/api/1.1/CHtml) helper class that can produce commonly used HTML code.

**2. Yii Conventions**

**2.1. URL**

By default, Yii recognizes URLs with the following format:

http://hostname/index.php?r=ControllerID/ActionID

The r GET variable refers to the [route](http://www.yiiframework.com/doc/guide/1.1/en/basics.controller#route) that can be resolved by Yii into controller and action. If ActionID is omitted, the controller will take the default action (defined via [CController::defaultAction](http://www.yiiframework.com/doc/api/1.1/CController#defaultAction)); and if ControllerID is also omitted (or the r variable is absent), the application will use the default controller (defined via [CWebApplication::defaultController](http://www.yiiframework.com/doc/api/1.1/CWebApplication#defaultController)).

With the help of [CUrlManager](http://www.yiiframework.com/doc/api/1.1/CUrlManager), it is possible to create and recognize more SEO-friendly URLs, such as http://hostname/ControllerID/ActionID.html. This feature is covered in detail in [URL Management](http://www.yiiframework.com/doc/guide/1.1/en/topics.url).

**2. 2. Code**

Yii recommends naming variables, functions and class types in camel case which capitalizes the first letter of each word in the name and joins them without spaces. Variable and function names should have their first word all in lower-case, in order to differentiate from class names (e.g. $basePath, runController(), LinkPager). For private class member variables, it is recommended to prefix their names with an underscore character (e.g. $\_actionList).

Because namespace is not supported prior to PHP 5.3.0, it is recommended that classes be named in some unique way to avoid name conflict with third-party classes. For this reason, all Yii framework classes are prefixed with letter "C".

A special rule for controller class names is that they must be appended with the word Controller. The controller ID is then defined as the class name with first letter in lower case and the word Controller truncated. For example, the PageController class will have the ID page. This rule makes the application more secure. It also makes the URLs related with controllers a bit cleaner (e.g. /index.php?r=page/index instead of /index.php?r=PageController/index).

**2.3. Configuration**

A configuration is an array of key-value pairs. Each key represents the name of a property of the object to be configured, and each value the corresponding property's initial value. For example, array('name'=>'My application', 'basePath'=>'./protected') initializes the name and basePath properties to their corresponding array values.

**2.4. File**

Class files should be named after the public class they contain. For example, the [CController](http://www.yiiframework.com/doc/api/1.1/CController) class is in the CController.php file.

View files should be named after the view name. For example, the index view is in the index.php file. A view file is a PHP script file that contains HTML and PHP code mainly for presentational purpose. A configuration file is a PHP script whose sole purpose is to return an associative array representing the configurations

**2.5. Directory**

WebRoot/protected: this is the [application base directory](http://www.yiiframework.com/doc/guide/1.1/en/basics.application#application-base-directory) holding all security-sensitive PHP scripts and data files. Yii has a default alias named application associated with this path. This directory and everything under should be protected from being accessed by Web users. It can be customized via [CWebApplication::basePath](http://www.yiiframework.com/doc/api/1.1/CWebApplication#basePath).

 WebRoot/protected/extensions: this directory holds all third-party extensions. It can be customized via [CApplication::extensionPath](http://www.yiiframework.com/doc/api/1.1/CApplication#extensionPath). Yii has a default alias named ext associated with this path.

 WebRoot/protected/modules: this directory holds all application [modules](http://www.yiiframework.com/doc/guide/1.1/en/basics.module), each represented as a subdirectory.

 WebRoot/protected/controllers: this directory holds all controller class files. It can be customized via [CWebApplication::controllerPath](http://www.yiiframework.com/doc/api/1.1/CWebApplication#controllerPath).

 WebRoot/protected/views: this directory holds all view files, including controller views, layout views and system views. It can be customized via [CWebApplication::viewPath](http://www.yiiframework.com/doc/api/1.1/CWebApplication#viewPath).

WebRoot/protected/views/layouts: this directory holds all layout view files. It can be customized via [CWebApplication::layoutPath](http://www.yiiframework.com/doc/api/1.1/CWebApplication#layoutPath).

WebRoot/assets: this directory holds published asset files. An asset file is a private file that may be published to become accessible to Web users. This directory must be writable by Web server process. It can be customized via [CAssetManager::basePath](http://www.yiiframework.com/doc/api/1.1/CAssetManager#basePath).

**2.6. Database**

Most Web applications are backed by some database. For best practice, we propose the following naming conventions for database tables and columns. Note that they are not required by Yii.

* Both database tables and columns are named in lower case.
* Words in a name should be separated using underscores (e.g. product\_order).
* For table names, you may use either singular or plural names, but not both. For simplicity, we recommend using singular names.
* Table names may be prefixed with a common token such as tbl\_. This is especially useful when the tables of an application coexist in the same database with the tables of another application. The two sets of tables can be readily separate by using different table name prefixes.

**3. General Format**

**3.1. General**

|  |  |
| --- | --- |
| For files that contain only PHP code, the closing tag (“?>”) is to be omitted. |  |

**3.2. Indentation**

Use an indent of 4 spaces with no tab characters.

**3.3. Maximum Line Length**

The target line length is 80 characters; i.e., developers should aim keep code as close to the 80-column boundary as is practical. However, longer lines are acceptable. The maximum length of any line of PHP code is 120 characters.

**3.4. Line Termination**

Lines should not contain trailing spaces. In order to facilitate this convention, most editors can be configured to strip trailing spaces, such as upon a save operation.

**3.5. Variable Names**

Variable names should be all lowercase, with words separated by underscores. For example, $current\_user is correct, but $currentuser, $currentUser or $CurrentUser are not. Variable names should be short yet meaningful. The choice of a variable name should indicate to the casual observer the intent of its use.

One-character variable names should be avoided except for temporary variables and loop indices. Common names for temporary variables are i, j, k, m, and n for integers; c, d, e for strings.

* use all lower case letters
* use '\_' as the word separator
* do not use 'l' (lowercase 'L') as a temporary variable
* do not use '-' as the word separator

Justification

* This allows a variable name to have the same name as a database column name, which is a very common practice in PHP.
* 'l' (lowercase 'L') is easily confused with 1 (the number 'one')
* If '-' is used as a word separator, it will generate warnings used with magic quotes.

## 3.6. Class Attribute Names

* All private, static class member attribute/method names should be pre appended with the character '\_'.
* After the '\_' use the same rules as for variable/method names and for public members same rule as variable and method names.
* '\_' always precedes other name modifiers like 'r' for reference.

Justification

* Pre appending '\_' prevents any conflict with method names and it access specifiers.

Example: private $\_fields = array();,public $data = array();

## 3.7. Method Argument Names

Since function arguments are just variables used in a specific context, they should follow the same guidelines as variable names. It should be possible to tell the purpose of a method just by looking at the first line, e.g. getUserData($user\_name).

By examination, you can make a good guess that this function gets the user data of a user with the username passed in the $user\_name argument. Method arguments should be separated by spaces, both when the function is defined and when it is called. However, there should not be any spaces between the arguments and the opening/closing parentheses.

## Global Variables

Global variables should be prepended with a 'g'. It's important to know the scope of a variable.

Example: global $g\_country\_time\_zone\_lookup instead of global $countryTimezoneLookup

* 1. **Functions and Methods**

Function names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in function names but are discouraged.

Function names must always start with a lowercase letter. When a function name consists of more than one word, the first letter of each new word must be capitalized. This is commonly called the “camelCase” method.

|  |
| --- |
| These are examples of acceptable names for functions:filterInput() |
|  |
| getElementById() |
|  |
| widgetFactory()   * 1. **Single or Double Quotes** * Use single quote when you don’t want to substitute any variable value within the string. * Use doublr quote when you want to replace a variable within a string with value.   Example:  $element\_name = 'foo\_bar';  $count = 3;  $message = “$count records found.”;   * 1. **Braces {} Guideline**   Of the three major brace placement strategies two are acceptable, with the first one listed being preferable:   * if ($condition) { while ($condition) { * } } |
|  |

* if ($condition) while ($condition)
* { {
* } }

## If Then Else Formatting

Layout

It's up to the programmer. Different bracing styles will yield slightly different looks. One common approach is:

if (condition) // Comment

{

}

else if (condition) // Comment

{

}

else // Comment

{

}

If you have *else if* statements then it is usually a good idea to always have an else block for finding unhandled cases. Maybe put a log message in the else even if there is no corrective action taken.

Condition Format

Always put the constant on the left hand side of an equality/inequality comparison. For example:

if (6 == $errorNum) ...

One reason is that if you leave out one of the = signs, the pharser will find the error for you. A second reason is that it puts the value you are looking for right up front where you can find it instead of buried at the end of your expression. It takes a little time to get used to this format, but then it really gets useful.

## Switch Formatting

* Falling through a case statement into the next case statement shall be permitted as long as a comment is included.
* The *default* case should always be present and trigger an error if it should not be reached, yet is reached.
* If you need to create variables put all the code in a block.

Example

switch (...)

{

case 1:

...

// FALL THROUGH

case 2:

{

$v = get\_week\_number();

...

}

break;

default:

}

* 1. **Tags**

Tags are single words prefixed by a "@" symbol. Tags inform phpDocumentor how to present information and modify display of documentation.

*Here are some of the phpDocumentor tags:*

@abstract

@access *private*

@author *Gregory Beaver <cellog@php.net>*

@copyright *Copyright (c) 2002, Gregory Beaver*

*@final [document a class method that should never be overridden in the child class]*

*@global bool $var\_name*

*@ignore [Prevents documenting of an element]*

@internal *the class uses the private methods*

@license *http://opensource.org/licenses/gpl-license.php GNU Public License*

*@link http://www.valuepitch.com*

*@name [Specifies an aliasto use for linking]*

*@param bool $flag*

*@return [Specify the return type of a function or method]*

*@see [Display a link to the documentation for an element]*

*@static [Document a static property and method]*

@staticvar *integer used to calculate the division tables*

@package *pagelevel\_package*

@subpackage *data*

@todo *make it do something*

@tutorial *phpDocumentor.pkg*

*@var [Document the data type of the class variable ]*

* 1. **Source Code Control System (SVN) Comments**

Some issues to keep in mind:

* On checking in the code to repository comments should be written describing for what the change has been made and format of log comments should be like as following:

# No Magic Numbers

A magic number is a bare-naked number used in source code. It's magic because no-one has a clue what it means including the author inside 3 months. For example:

if (22 == $foo) { start\_thermo\_nuclear\_war(); }

else if (19 == $foo) { refund\_lotso\_money(); }

else if (16 == $foo) { infinite\_loop(); }

else { cry\_cause\_im\_lost(); }

In the above example what do 22 and 19 mean? If there was a number change or the numbers were just plain wrong how would you know?

Heavy use of magic numbers marks a programmer as an amateur more than anything else. Such a programmer has never worked in a team environment or has had to maintain code or they would never do such a thing.

Instead of magic numbers use a real name that means something. You should use define(). For example:

define("PRESIDENT\_WENT\_CRAZY", "22");

define("WE\_GOOFED", "19");

define("THEY\_DIDNT\_PAY", "16");

if (PRESIDENT\_WENT\_CRAZY == $foo) { start\_thermo\_nuclear\_war(); }

else if (WE\_GOOFED == $foo) { refund\_lotso\_money(); }

else if (THEY\_DIDNT\_PAY == $foo) { infinite\_loop(); }

else { happy\_days\_i\_know\_why\_im\_here(); }

1. **Security**

**4.1. General security**

* Validate the user input (see below for details).
* Protect (escape) your application output according to context (see below for a few output types, mostly HTML and SQL).
* Test your application in debug mode.  
  Set the constant YII\_DEBUG to true (by default, it is defined in index.php) and put alongside error\_reporting(E\_ALL);. Then errors and warnings will stop the execution and Yii will display the message, the source code and the call stack. Even an undefined key in an array (which is just a "E\_NOTICE" level) can cause security problems.
* Disable the debug mode in production.  
  Make sure your error messages don't contain sensitive information.
* In production, keep logs. Parse them regularly for warnings and errors.  
  There are two levels of logs : application logs (handled by Yii) and server logs (handled by PHP and usually Apache). Yii logs are described in [The Definitive Guide to Yii, Logging](http://www.yiiframework.com/doc/guide/1.1/en/topics.logging). PHP logs are usually on by default. Please check your server configuration and your rights on the file system for accessing these log files.

**4.2. Validating User Input**

#### Validating through a model

* Validating through controller

## 4.3. HTML output and XSS

* If the application prints unfiltered user input inside a HTML page, then it allows a malicious user to change the display of this page, and to inject client code (usually JavaScript) that can be run by other users. One typical use of these XSS attacks is to steal user sessions.

## 4.4. SQL Injections

* When some user data is put unfiltered in a SQL query, it allows a malicious user to send its own SQL in the query.

<?php

// warning, dangerous code

Yii::app()->db

->createCommand("DELETE FROM mytable WHERE id = " . $\_GET['id'])

->execute();

$comments = Comment::model->findAll("user\_id = " . $\_GET['id']);

## 4.5. Cross-site Request Forgery (CSRF)

## Please note that HTTP requests that modify the server state (create, update, delete) should be with the POST protocol. This is a good practice, as recommended by REST, and it helps web browser to prevent accidental re-send of these requests.

## 4.6. Authorization

Authorization is ensuring users only have access to the resources they have permissions on. This is a lengthy subject, and Yii provides many useful classes to handle permissions and roles. To learn about this, please read The Definitive Guide to Yii from [Access Control Filter](http://www.yiiframework.com/doc/guide/1.1/en/topics.auth#access-control-filter) to [Using Business Rules](http://www.yiiframework.com/doc/guide/1.1/en/topics.auth#using-default-roles).

## 4.7. Authentication

### 4.7.1 Password strength

The validation rule must reject any weak password. Writing its own validation method is easy: just require a minimum size, and check that different classes of characters are present.

### 4.7.2 Encrypting passwords

This section considers only internal authentication, i.e. through passwords managed by the application. It does not consider LDAP, SSO, OpenID, or any other external service.

If the authentication process is internal, then of course you shouldn't store the passwords in plain text. The easiest solution for encryption is to use the well-known library [PHPass](http://www.openwall.com/phpass/). With Yii, it can be as simple as the following "User" model:

* 1. **Massive assignment**

Once your model's validators have approved all the fields, it's time to make use of the data produced by the form, in bulk. This happens during form submission by calling the controller's action.

Here we'll look at the post/update code:

// protected/controllers/CommentController.php

public function actionUpdate()

{

$model = $this->loadModel();

if (isset($\_POST['Comment']))

{

$model->attributes = $\_POST['Comment']; // Massive Assignment

....

When ever we use massive assignment we must assing the default value in corresponding controller.

## Cross site request forgery prevention

Cross-Site Request Forgery (CSRF) attacks occur when a malicious web site causes a user's web browser to perform an unwanted action on a trusted site. For example, a malicious web site has a page that contains an image tag whose src points to a banking site: http://bank.example/withdraw?transfer=10000&to=someone.

If a user who has a login cookie for the banking site happens to visit this malicous page, the action of transferring 10000 dollars to someone will be executed. Contrary to cross-site, which exploits the trust a user has for a particular site, CSRF exploits the trust that a site has for a particular user.

To prevent CSRF attacks, it is important to abide to the rule that GET requests should only be allowed to retrieve data rather than modify any data on the server. And for POST requests, they should include some random value which can be recognized by the server to ensure the form is submitted from and the result is sent back to the same origin.

## Cookie Attack Prevention

Protecting cookies from being attacked is of extreme importance, as session IDs are commonly stored in cookies. If one gets hold of a session ID, he essentially owns all relevant session information.There are several countermeasures to prevent cookies from being attacked.

An application can use SSL to create a secure communication channel and only pass the authentication cookie over an HTTPS connection. Attackers are thus unable to decipher the contents in the transferred cookies.

Expire sessions appropriately, including all cookies and session tokens, to reduce the likelihood of being attacked.

Prevent cross-site scripting which causes arbitrary code to run in a user's browser and expose his cookies.

Validate cookie data and detect if they are altered.