Using Smarty templating for MVC

About

This document explains how to install and use SMARTY templates in the construction of PHP web applications that use an MVC design pattern.

What is SMARTY

SMARTY is a PHP application that enables the separation of the View (the interface) from the Controller (the PHP scripts that actually 'run' / are the application).

Downloading and installing SMARTY

The SMARTY web site contains download distributions of ZIP files that you can use for installation. These require some work creating directories and placing files in the correct location – rather than do this you can download a customised distribution that will work on *studentnet.kingston.ac.uk*

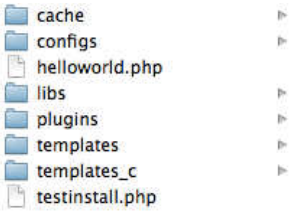


Figure 1

Unless doing something more complex, you should really only need to add two files at a time – a template (.tpl) file into the *templates* directory (the View), and a corresponding PHP file in the *root* of this directroy (the Controller). This file must have the same name as the template file but have a .php extension.

For information

You don't need to know this to use SMARTY, but the directory and files in standard installation are used as follows - the *cache* and *templates_c* directories are used to store temporary versions of the web pages as they are generated by SMARTY. The *configs* directory contains settings for SMARTY, where more advanced users of SMARTY set how long the cached versions are stored to increase the efficiency of their web site. *Libs* and *plugins* contain the actual executable parts of SMARTY. The only directory to add content to is *templates*, which stores .tpl files containing HTML and special SMARTY markup.

Task 1 Creating your SMARTY structure

Download *lab11.zip* from barryavery.com and extract the contents. You should end up with a folder called *lab11* with several directories and files inside (figure 1)

Use an SFTP client to copy the entire lab11 directory over to your WWW directory (it's important to keep exactly the same directory structure as in Figure 1).

SMARTY will create temporary versions of the web site in the *templates_c* and *cache* directory – to do this you will have to change the permissions of these folders so that *Write* access is enabled. To do this in Filezilla right mouse click on the *templates_c* folder and choose *File Permissions* (your FTP client may have different way of doing this). Change the Write permission to be enabled for

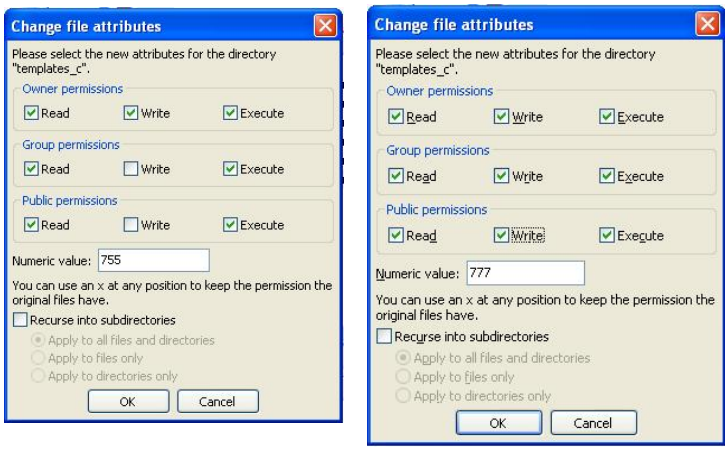


Figure 2

Group and *Public* by ticking the boxes (figure 2). Repeat this for the *cache* directory.

Task 2 Testing your SMARTY installation

A small smarty program has been included to test the install – run the *testinstall.php* file in a browser using the URL *http://studentnet.kingston.ac.uk/~kxxxxxxx/lab11/testinstall.php*

```
Smarty Installation test...
Testing template directory...
/home/ku12492/www/lab11/templates is OK.
Testing compile directory...
/home/ku12492/www/lab11/templates_c is OK.
Testing plugins directory...
/home/ku12492/www/lab11/libs/plugins is OK.
Testing cache directory...
/home/ku12492/www/lab11/cache is OK.
Testing configs directory...
/home/ku12492/www/lab11/configs is OK.
Testing sysplugin files...
... OK
Testing plugin files...
... OK
Tests complete.
```

Figure 3

It should produce a series of lines indicating that all the SMARTY files and directories are in the right places and have the correct permission (figure 3). If you get alternative messages, check your locations and permissions.

Next try the supplied *helloworld.php* file which prints "Hello World!!" using the URL *http://studentnet.kingston.ac.uk/~kxxxxxxx/lab11/helloworld.php*

If the words don't appear, recheck the locations and permissions again.

How SMARTY works

SMARTY uses the MVC design pattern. For each displayed XHTML *view*, SMARTY requires a corresponding *controller* page. In this setup, controller pages (.php) files, are placed in the root folder. The code for *helloworld.php* (the controller file in the root folder) has a corresponding *helloworld.tpl* file in the *templates* folder.

In the MVC pattern (figure 4), the *Model* contains data structures (typically designed in a class/object based style) that model real world objects used in the system – these could be things like person, student, invoice, shopping cart. These would be files containing PHP and SQL connectivity to database relations, but no XHTML.

The *View* contains XHTML and a minimal amount of PHP to get the values that are required for the page to be displayed. These can be thought of as XHTML templates that can be reused. There would be no SQL or database connectivity PHP here.

The *Controller* is the main file (or series of files) that use Model or View files to do the required work. These files contain PHP code with no SQL or XHTML.

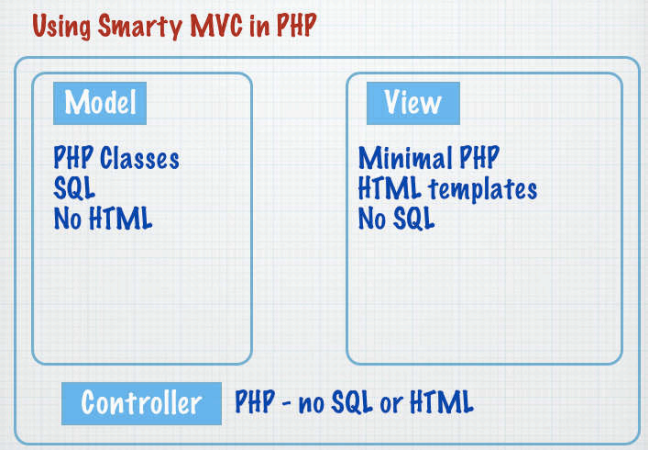


Figure 4

Code for the helloworld.php controller

```
1  <?php
2  //Place controller code here
3
4
5  // These lines initialise the smarty View object
6  // Don't change these unless the files are in different directories
7  require('libs/Smarty.class.php');
8  $smarty = new Smarty();
9  $smarty->setTemplateDir('templates');
10 $smarty->setCompileDir('templates_c');
11 $smarty->setCacheDir('cache');
12 $smarty->setConfigDir('configs');
13
14 //These are the lines to change
15 //Add one assign line for each named piece of data with the data
16 $smarty->assign('hello', 'Hello World!!');
17
18 //This line should be changed to match the template filename in /templates
19 $smarty->display('helloworld.tpl');
20 ?>
```

The code for *helloworld.php* demonstrates a basic SMARTY *Controller* file. Lines 7 – 12 are only changed when the default directories are in different locations. They create a Smarty object (line 8), which is then populated with the directory locations and set up values for all the SMARTY files using various set functions (line 9-12). Remember that for almost all cases these lines don't need to be changed.

Lines 16 and 19 are the important ones that are changed. For every value that we wish to use in the HTML template, we create a unique name for it and give it a value using the assign function – line 16 demonstrates creating a named value "hello" and gives it the value "Hello World!!". This line would be repeated many times with pairs of names and values (it is bad design to perform calculations on a *View* page, so all this work must take place on this page).

Every piece of data that may vary in the XHTML template page must be named and given a value here – note that names in the assign function do not require a $ in front.

Line 19 (which is always the penultimate line) indicates the name of the *View* file which is then displayed – these are in the *templates* folder and it is good practice to use the same names for Controller files as for View files (so the controller *helloworld.php* has a corresponding *helloworld.tpl* view in the template folder).

Code for helloworld.tpl view

```
1  <html>
2  <head>
3  <title>Smarty</title>
4  </head>
5  <body>
6  <p>{$hello}</p>
7  </body>
```

Smarty views are simple text values containing XHTML, which have a .tpl file extension. Values passed in from the controller file are inserted into the template by using a {$name} syntax (line 6) by SMARTY. Note the use of the $ symbol here before the use of names given a value in the controller file.

Task 3 Trying out the Controller and View files

Activity: Changing a piece of data in the Controller

Change line 16 in *helloworld.php* so that there is a different value printed – change the assign statement so the value of *hello* is "Another hello" and try it out (remember to FTP the file and to refresh the page).

Activity: Adding a new data value to the Controller and the View

Change line 3 in *helloworld.tpl* so that the text between the title tags is {$title}, then add a new line (at line 17) to *helloworld.php* to assign the text "This is my smarty title" to this name (i.e. $smarty->assign('title', '...

Activity: Check to see if your page reflects this change in a browser window.

Constructing a larger example

The following pages explain the steps required to get a larger example working – one which uses a database table, a PHP class and a form / response page to show results. As the example builds up, it will show the various Model, View and Controller parts.

This example will present a form that requests an *empno* – the details for this employee will then be retrieved from a database table and printed out on a response page.

Form to find an employee using their empno

Response page showing details on the requested emp

The Model will consist of an Emp and EmpDB class that will handle processing the Emp database table, Views will handle the form and response page and the Controller will oversee the whole process.

empno	ename	job	mgr	hiredate	sal	comm	deptno
405	MARCH	ADMIN	938	13/06/1997	18000	*NULL*	2
535	BYRNE	SALES	734	15/08/1997	26000	300	3
557	BELL	SALES	734	26/03/2000	22500	500	3
602	BIRD	MANAGER	875	31/10/1997	39750	*NULL*	2
690	AHMAD	SALES	734	05/12/1997	22500	1400	3
734	COX	MANAGER	875	11/06/2002	38500	*NULL*	3
818	POLLARD	MANAGER	875	36660	34500	*NULL*	1
824	REES	ANALYST	602	05/03/2000	40000	*NULL*	2
875	PARKER	PRESIDENT	*NULL*	09/07/2002	60000	*NULL*	1
880	TURNER	SALES	734	04/06/2001	25000	0	3
912	HAYES	ADMIN	824	04/06/2001	21000	*NULL*	2
936	CASSY	ADMIN	734	37460	19500	*NULL*	3
938	GIBSON	ANALYST	602	05/12/1997	40000	*NULL*	2
970	BLACK	ADMIN	818	21/11/1997	23000	*NULL*	1

Emp database table

Task 4 Creating the database tables used in this project

Download the second zip (lab 11 – more files.zip), which contains various files. Extract the contents and look for the *create EMP DEPT GRADE sql.txt* file. This file contains various SQL statements, which create EMP, DEPT and GRADE tables, and populate them with data. Open the file in a text editor of your choice, copy ALL the SQL and then use phpMyAdmin to run the SQL (paste the SQL lines into the SQL tab and press GO)

You should end up with three tables – the EMP table (above) and a DEPT and GRADE table

deptno	dname	loc
1	ACCOUNTING	LONDON
2	RESEARCH	YORK
3	SALES	BIRMINGHAM
4	OPERATIONS	LEEDS

grade	losal	hisal
1	17000	21999
2	22000	23999
3	24000	29999
4	30000	49999
5	50000	99999

Constructing a Model

A class is used to model the Employee information – open *class.Emp.php* in a text editor to examine its contents (figure 5).

```php
<?php
class Emp {
    protected $empno;
    protected $ename;
    protected $job;
    protected $mgr;
    protected $hiredate;
    protected $sal;
    protected $comm;
    protected $deptno;

    function __construct($new_empno="", $new_ename="", $new_job="", $new_mgr="", $new_hiredate="", $new_sal="", $new_comm="", $new_deptno=""){
        $this->empno=$new_empno;
        $this->ename=$new_ename;
        $this->job=$new_job;
        $this->hiredate=$new_hiredate;
        $this->sal=$new_sal;
        $this->comm=$new_comm;
        $this->deptno=$new_deptno;
    }

    function get_empno(){
    return $this->empno;
    }

    function get_ename(){
    return $this->ename;
    }

    function get_job(){
    return $this->job;
    }
```

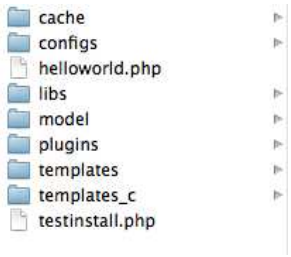
Figure 5

In this example, there is a one-to-one correlation between the *Emp class* and the *Emp database table* (in other words every column in the EMP database table has a corresponding attribute and get/set pair in the EMP class). Note that this is not necessarily true in all designs (for example the class may have functions to generate calculated fields that are not stored or the values in a class may come from more than one table).

Task 5 Creating the model

We will need to create a folder for the models used in this example

Create a folder called *model* and place the *class.Emp.php* file in here. All the models used in this example will be placed in this folder (it is good practice to place all the class files in a single place).



Controller and View files to demonstrate the use of the *Emp* class has been included. Place *empTest1.php* in the root and place *empTest1.tpl* in the templates folder.

Testing the Emp class

View *empTest1.php* file in a folder to see it working – it creates a single *Emp* object, fills it with data and then uses a template to display the employees name.

empTest1.php

Line 4 indicates that this file will use the Emp class – line 5 creates an Emp Object called $newEmp and uses the Emp constructor to fill it with values.

Line 19 creates a new SMARTY name (ename) and gives it a value using the *get_ename()* method in the $*newEmp* object

Line 22 indicates the name of the template that will be used – here it is *empTest1.tpl*

```
1  <?php
2
3  //Place controller code here
4  require ('model/class.Emp.php');
5  $newEmp = new Emp(123, "Homer","Manager", 456,"02/02/98",34000,0,2);
6
7
8  // These lines initialise the smarty View object
9  // Don't change these unless the files are in different directories
10 require('libs/Smarty.class.php');
11 $smarty = new Smarty();
12 $smarty->setTemplateDir('templates');
13 $smarty->setCompileDir('templates_c');
14 $smarty->setCacheDir('cache');
15 $smarty->setConfigDir('configs');
16
17 //These are the lines to change
18 //Add one assign line for each named piece of data with the data
19 $smarty->assign('ename', $newEmp->get_ename());
20
21 //This line should be changed to match the template filename in /templates
22 $smarty->display('empTest1.tpl');
23 ?>
24
```

empTest1.tpl

Line 7 is the only line that demonstrates SMARTY here – the name created in *empTest1.php* using the assign statement (*ename – line 19*) is printed between two paragraph tags (remember that names in the template files must have a $ in front of them).

```
1  <html>
2  <head>
3  <title>Display an Emp</title>
4  </head>
5  <body>
6  <h1>Details for an Emp</h1>
7  <p>Employees name is {$ename}</p>
8  </body>
9  </html>
```

Task 6 Showing more Emp details

Edit *empTest1.php* (and *empTest1.tpl*) to show more of Homers details - his *empno* and *job*.

To do this add extra *$smarty->assign…* lines to *empTest1.php* creating two new names. Give these values using the get_empno() and get_job() methods. Next – edit *empTest1.tpl* to show these i.e. something like *<p>{$ename} is number {$empno} in his job which is {$job}<p>*

A shortcut

Creating new (temporary) names in the controller for each piece of data (attributes in the object) can be cumbersome – an alternative approach is indicated in *empTest2.php* and *empTest2.tpl*

Rather than pass individual pieces of data into the template, SMARTY allows the whole object to be passed through (line 19 – the whole *$newEmp* object is passed in as the named *passedEmp* object)

```
17  //These are the lines to change
18  //Add one assign line for each named piece of data with the data
19  $smarty->assign('passedEmp', $newEmp);
20
```

The template can then use the methods inside the object to get at the values (line 7 and 8). Note how the object $newEmp is called $passedEMP in the template. You can use the same name if you wish.

```
1   <html>
2   <head>
3   <title>Display an Emp</title>
4   </head>
5   <body>
6   <h1>Details for an Emp</h1>
7   <p>Employees name is {$passedEmp->get_ename()} and they are a {$passedEmp->get_job()}</p>
8   <p>{$passedEmp->get_ename()} earns {$passedEmp->get_sal()}</p>
9   </body>
10  </html>
```

Task 7 Passing the object into the View

Try running these files and then alter your versions so they work the same way (you will delete lines from your version of *empTest1.php* and change it so that the object is passed through – then add method calls to *empTest1.tpl*).

Connecting the emp object to the emp table

There are alternative approaches to wiring a class to a database – in this instance a new class *class.EmpDB.php* will be used, which inherits methods and attributes from *class.Emp.php*.

```
1   <?php
2
3   include_once "model/class.Emp.php";
4
5   class EmpDB extends Emp {
6
7       function __construct($empno){
8           //Create connection to database server
9           $link = mysql_connect('localhost:8889', 'root', 'root')
10             or die('Could not connect: ' . mysql_error());
11          //Select the employee database
12          mysql_select_db('employee') or die('Could not select database');
13
14          // Construct and Perform the SQL query
15          $query = 'select * from emp where empno='.$empno;
16          $result = mysql_query($query) or die('Query failed: ' . mysql_error());
17
18          mysql_close($link);
19
20          //Convert the result to an associative array
21          $result=mysql_fetch_array($result);
22          //Either we have only one result which matches the empno, or nothing
23          if ($result['empno']){
24              //Fill up Emp with the values by using the emp parent constructor
25              parent::__construct($result['empno'], $result['ename'], $result['job'],$result['email'],
26                      $result['job'], $result['mgr'], $result['hiredate'],$result['sal'],
27                      $result['comm'],$result['deptno']);
28              }
29          else
30              {
31              //We didnt get any so fill up with nothing
32              parent::__construct();
33
34          };
35      }
36  }
37  ?>
```

Line 3 and 4 indicate that this will use the *Emp* class. When an object of this class is created, an *empno* is passed into the constructor function (line 7 - $empno). Lines 8-20 look like the standard mysql connectivity routines – the SQL statement uses $empno to get a single row back ($empno is appended onto the end of the SQL on line 15). To get this working on studentnet, line 9 must be edited to include your specific *database server*, *username* and *password*, with line 12 being changed to your *databasename*.

$result is an associative array containing either the one row (record) that matches the empno, or an empty null result. Line 23 tests to see if *$result['empno']* is true (i.e. there is a returned result) - if so the parent constructor (i.e. the Emp constructor) is used to populate the object with data (lines 25-27). If no row has been returned then the object is created empty (line 32).

empTest3.php uses this class to get the details for the employee with empno 405.

Line 4 indicates the class we will be using (note the use of *class.EmpDB.php* rather than *class.Emp.php*). Line 5 creates an EmpDB object using the details retrieved from the emp table, for employee number 405.

The rest of the code is the same (including the View file).

```
1  <?php
2
3  //Place controller code here
4  require ('model/class.EmpDB.php');
5  $newEmp = new EmpDB(405);
6
7
8  // These lines initialise the smarty View object
9  // Don't change these unless the files are in different directories
10 require('libs/Smarty.class.php');
11 $smarty = new Smarty();
12 $smarty->setTemplateDir('templates');
13 $smarty->setCompileDir('templates_c');
14 $smarty->setCacheDir('cache');
15 $smarty->setConfigDir('configs');
16
17 //These are the lines to change
18 //Add one assign line for each named piece of data with the data
19 $smarty->assign('passedEmp', $newEmp);
20
21 //This line should be changed to match the template filename in /templates
22 $smarty->display('empTest3.tpl');
23 ?>
24
```

Task 8 Using the EmpDB class

To get this working, you need to place three files in different locations:

- *class.EmpDB.php* must be placed in the model folder
- *empTest3.php* must be placed in the root
- *empTest3.tpl* must be placed in the template folder.

Assuming that you have successfully constructed the EMP, DEPT and GRADE tables (on page 4) - you now need to edit lines 9 and 12 of *class.EmpDB.php* with your *mysql server name*, *username*, *password* and *database name*.

You should now be able to bring up *empTest3.php* in a browser window to see person 405s details.

Now edit line 5 to retrieve details on a different person – try 734 or 818 in the constructor function.

Creating the form controller

As we are using SMARTY, we will need two files to use a form to get an employer number from the user – a controller *findanemp.php* and a view *findanemp.tpl*

The controller file has very little extra functionality in it – just an indicator which view should be used (line 18).

```
1  <?php
2  //Place controller code here
3
4
5  // These lines initialise the smarty View object
6  // Don't change these unless the files are in different directories
7  require('libs/Smarty.class.php');
8  $smarty = new Smarty();
9  $smarty->setTemplateDir('templates');
10 $smarty->setCompileDir('templates_c');
11 $smarty->setCacheDir('cache');
12 $smarty->setConfigDir('configs');
13
14 //These are the lines to change
15 //Add one assign line for each named piece of data with the data
16
17 //This line should be changed to match the template filename in /templates
18 $smarty->display('findanemp.tpl');
19 ?>
20
```

Task 9 Create the findanemp.php controller

Copy one of your controller files and name it *findanemp.php.* Edit the contents so that it matches the controller code above (you will probably need to remove a few lines and then edit line 18). This file needs to be placed in the root folder.

Creating the findanemp.tpl view

The view for the form has little/no smarty – all it contains is a form and input text entry to get the employee number.

```
1  <html>
2  <head>
3  <title>Find an Emp</title>
4  </head>
5  <body>
6  <form name="findanemp" action="responseEmp.php" method="POST">
7  <p>Enter an empno: <input type="text" name="empNo"></p>
8  <p>Press to show details <input type="submit" name="find" value="find" /></p>
9  </form>
10 </body>
11 </html>
```

Task 10 Create the findanemp.tpl view

Create the *findanemp.tpl* view using the markup above. This file will need to be placed in the *templates* folder.

Load *findanemp.php* in a browser window to ensure that it is working (remember that you will need to FTP both the controller and the view files).

Creating the response controller – responseEmp.php

The response file needs to find the *empno* entered in the form, get the details from the *emp* table that match this employee, and then pass these details (in the form of an object) to the appropriate view.

Line 4 uses the _POST array to get the employee number – line 6 uses this value to create a new *EmpDB* object containing the details of this person.

This object is then passed into the responseEmp.tpl view (line 20 and line 23).

```
1  <?php
2  //Place controller code here
3
4  $empNo=$_POST["empNo"];
5  require ('model/class.EmpDB.php');
6  $newEmp = new EmpDB($empNo);
7
8
9  // These lines initialise the smarty View object
10 // Don't change these unless the files are in different directories
11 require('libs/Smarty.class.php');
12 $smarty = new Smarty();
13 $smarty->setTemplateDir('templates');
14 $smarty->setCompileDir('templates_c');
15 $smarty->setCacheDir('cache');
16 $smarty->setConfigDir('configs');
17
18 //These are the lines to change
19 //Add one assign line for each named piece of data with the data
20 $smarty->assign('passedEmp', $newEmp);
21
22 //This line should be changed to match the template filename in /templates
23 $smarty->display('responseEmp.tpl');
24 ?>
25
```

Task 11 Create the responseEmp.php controller

Copy empTest3.php to a new file called responseEmp.php and alter it look like the code above. This will need to be in the root folder.

Creating the response view – responseEmp.tpl

The response view is almost identical to versions that were previously created – the passedEmp object has get methods which are used to print out the various issues in the correct places in the template.

```
1  <html>
2  <head>
3  <title>Display an Emp</title>
4  </head>
5  <body>
6  <h1>Details for an Emp</h1>
7  <p>Employees name is {$passedEmp->get_ename()} and they are a {$passedEmp->get_job()}</p>
8  <p>{$passedEmp->get_ename()} earns {$passedEmp->get_sal()}</p>
9  </body>
10 </html>
```

Task 12 Create the responseEmp.tpl view

Copy one of your existing views to a new file called *responseEmp.tpl* and alter it look like the code above (only minimal changes will probably be required) – this will need to be in the templates folder.

Try out the form with various user details and see if they are correctly retrieved and displayed

A complete version of all the files used in this project are available from the web site (lab11 final.zip). Remember that to get this working you will still need to edit the database settings in the *class.EmpDB.php* file.

Summary

It can be difficult to see the justification for using the MVC design pattern – the common criticism is that it requires a lot of extra code and files. This may be true for very small projects, but once a project becomes larger to any extent, it becomes much quicker to create pages because of the reusability and maintainability offered by the clear subdivision of the parts.

Further Activities

Any employees that earn 40000 or more are to be classified as "executive" – otherwise they are known as "employees".

This is an example of a calculated field (and should not be stored in the database table) – all the work should be done in the *Emp* Class.

Change the Emp class so that it includes a method (function) called get_classification that returns the string "executive" or "employee" depending on their salary (use an if statement). Then change the rest of the code so that the classification is displayed in the template.

You should find that this change is easy to implement (a small change to the emp class and the response template).