PHP and Oracle

How to develop a application in PHP using Zend Framework, Oracle Database and Core for Oracle

Author: Gaylord Aulke, Zend Technologies GmbH

Table of Contents **Page**

Preface 1
Installing Zend Core for Oracle 1
Installing Zend Framework 1
  Prerequisites 1
  Download and unpack 1
  Configure PHP and web server 2
Building a Zend Framework application skeleton 2
  Bootstrap file 2
  Directory structure 2
  Default Action Controller 2
  Mapping URL's to controllers and Action Methods 3
  More Action Controllers 3
  Creating a Model 3
  Connecting the DB 3
  Using an external configuration file 4
  Separating layout from application logic using View 5
The HR Application (example) 6
  Listing all Employees 7
  Editing an Employee 7
Epilogue 10

Preface

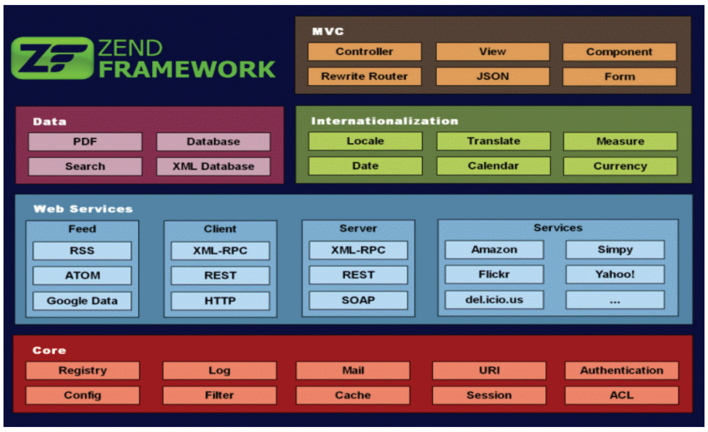
Zend Core for Oracle is a Zend certified and supported version of the open source PHP. It uniquely delivers a seamless out-of-the-box experience by bundling all the necessary drivers and third party libraries to work with the database of your choice. It can be downloaded at: http://www.zend.com/downloads

Zend Framework[1] is an open source framework for PHP. It helps to structure PHP based web applications in a good way and comes with a variety of libraries and tools that increase developer efficiency. The most important development guidelines of Zend Framework were two principles:

1. extreme simplicity
2. use-at-will architecture

In addition to a MVC implementation and lots of libraries, Zend Framework defines common coding standards[2] including naming conventions and a package-like directory structure.

Zend Framework is open source and licensed under the new BSD license. Therefore it can be integrated even in commercial applications without the constraints imposed by other licenses such as the GPL. Specifically, your project is not forced to adopt an open source license.



Zend Framework overall Structure

Installing Zend Core for Oracle

Please refer to the "PHPfest Tutorial: Oracle Database 10g Express Edition and Zend Core for Oracle" for installation instructions. On Windows, just start the setup.exe after downloading, answer some questions right and lean back. Zend Core for Oracle will install on existing Apache and IIS web servers. It also brings its own Apache for a complete new installation if needed.

Installing Zend Framework

Prerequisites

- Apache with mod_rewrite required or IIS with ISAPIRewrite Software[3]
- PHP 5.1.4 or later
- activated OCI8-Extension
  (PHP with OCI8 Extension comes with the Zend Core for Oracle)

Download and unpack

To install Zend Framework, just download the newest zip package from http://framework.zend.com and extract it to a directory that is readable by the web server process, so PHP applications can include the classes in their include_path. A good choice for the location of the framework could be:

```
/opt/ZendFramework/
   ... on linux systems (according to FHS)⁴
/Library/Zend Framework
   ... on Mac OS X
C:\Program Files\Zend Technologies\Zend Framework\
   ... on Windows
```

(Currently install packages for different operating system types are under development, but they are not available yet)

1 Zend Framework: http://framework.zend.com
2 Coding Standards: http://framework.zend.com/wiki/display/ZFDEV/PHP+Coding+Standard+(draft)
3 IIS ISAPI Rewrite Engine: http://www.isapirewrite.com/
4 File System Hierarchy Standard (FHS): http://www.pathname.com/fhs/pub/fhs-2.3.html

Configure PHP and web server

After unpacking, point the PHP include_path (directive set in php.ini) to the lib directory of the Zend Framework installation directory.
Example (excerpt from php.ini):

```
;;;;;;;;;;;;;;;;;;;;;;;;
; Paths and Directories ;
;;;;;;;;;;;;;;;;;;;;;;;;

; UNIX: "/path1:/path2"
include_path = ".:/opt/ZendFramework/library"
;
    ; Windows: "\path1;\path2"
;include_path = ".;c:\php\includes"
```

There are some other ways of setting the include path for PHP. For more information refer to
http://de.php.net/manual/en/configuration.changes.php

In a MVC Application, all requests are routed to one single entry point. For Zend Framework this is a PHP script. Typically, this so called Controller Script would be the only PHP file inside your web server's document root. The framework files and your application code should be stored outside the document root of the web server. They are all loaded via include directives from the one Controller Script that is called for every request.
Typical Zend Framework URLs look like this:
http://myserver.com/
http://myserver.com/start
http://myserver.com/profile/login
To direct all these requests to the Controller Script, URL rewriting needs to be enabled on the web server. This can be done in the httpd.conf inside a Vhost directive or in a .htaccess file in a local directory. For IIS please refer to http://www.isapirewrite.com.
Assume your controller script is in the document root of your web server and it is called index.php. Then, the rewrite rule (for apache mod_rewrite) looks like this:

```
    RewriteEngine On
    RewriteRule !\.(html|php|js|ico|txt|gif|jpg|png|css|rss|zip|tar\.gz|wsdl)$
    /index.php
```

Building a Zend Framework Application Skeleton

Bootstrap File

After the rewrite rule is in place, the next thing needed will be a Controller Script (also referred to as the "bootstrap file") in the document root of your web server:

```
<?php
set_include_path('.:/pathToApp/app:/pathToApp/lib');
require_once 'Zend/Controller/Front.php';
Zend_Controller_Front::run('/pathToApp/app/controllers');
?>
```

If not possible in the php.ini file or .htaccess files, the include path can be set using the set_include_path command in this file.
The run method of the Controller requires the file system path to your action controllers as a string argument. The Action Controllers are your custom codes for doing whatever your application is supposed to do. All the application logic will be implemented in such Action Controller Classes.

Directory Structure

It is recommended that websites built with the Zend Framework share a common directory structure. Conforming to this structure makes your code more easily understandable by someone familiar with the conventions of the Zend Framework.
The suggested directory structure consists of both library directories (from Zend and elsewhere) and application directories.

```
/app
   /models
   /views
    /controllers  - where your action controllers go
/htdocs           - document root of the web server
   /images
   /styles
   .htaccess      - may contain rewrite rule and include_path (apache module)
   index.php
/lib              - symbolic link to /opt/ZendFramwork/library
   /Zend
   /PEAR
   ...etc
```

Default Action Controller

Zend Framework maps all requests to dynamic resources on the web server via the bootstrap file to so called Action Controllers. These are PHP Classes that have one Method per different request they can service. The first action controller that needs to be present in every application is the default controller. It is called whenever the framework did not read a specific controller name from the URL. For example it is used when the homepage of the application is called, i.e. only the domain name is given. The user requests: http://www.myserver.com
In this case, Zend Framework starts the Default Controller which is called IndexController and calls the indexAction in this controller. An IndexController can be coded like this:

.../app/controllers/IndexController.php:

```
<?php
require_once 'Zend/Controller/Action.php';
class IndexController extends Zend_Controller_Action
{
   public function indexAction()
   {
                echo 'Hello from IndexController';
   }
}
?>
```

In Order to be found by the FrontController called in the bootstrap file, the source code file must be stored in the specified controller directory (.../app/controllers in this case). The file must be named IndexController.php and it must contain a class declaration for the class IndexController. This class must extend the Framework Class Zend_Controller_Action (which is first included with the require_once directive) to connect with the FrontController. The Method indexAction() of this class will now be called without parameters whenever the homepage of the server is requested.

Mapping URLs to Controllers and Action Methods

Zend Framework generally maps the first given directory name to an Action Controller Name and the second directory name to a Method inside this controller. When only one identifier is given in the URL, it is used as the controller name an the method is set to 'index'. If no information is given in the URL, the index Method from the IndexController is used:

```
http://framework.zend.com/roadmap/future/
Controller: RoadmapController
Action          : futureAction

http://framework.zend.com/roadmap/
Controller: RoadmapController
Action          : indexAction

http://framework.zend.com/
Controller: IndexController
Action          : indexAction
```

More Action Controllers

Now other Action Controllers can be written and placed in the controller directory. They are all built in the same way as the IndexController described above. For more information on making the assignment from URLs to controllers more flexible see the documentation at http://framework.zend.com

Creating a Model

In MVC, all the actual work the application is supposed to do is implemented in a special class called 'Model' (the 'M' in MVC). This includes all database operations. The Action Controller instantiates a Model and then calls functions on it. To visualize this, let us first start with creating an empty Model class:

.../models/HRModel.php:

```
<?php
class HRModel {
    function helloWorld() {
    return 'Hello World';
    }
}
?>
```

Now we have a (very simple) Model with a function that returns something that can be output to the browser. So let us use it in the Controller. Since we will probably have many methods in our controller that are all very likely to call methods on our Model, we instantiate the Model right at the beginning of the lifetime of our controller and store it for later use in an instance variable for the action controller object:

.../app/controllers/IndexController.php:

```
<?php
require_once('Zend/Controller/Action.php');
require_once('models/HRModel.php');

class IndexController extends Zend_Controller_Action
{
   /**
   * @var HRModel
   */
   protected $hrModel;

   public function init() {
                 $this->hrModel = new HRModel();
   }

   public function indexAction()
   {
                 echo $this->hrModel->helloWorld();
   }
}
?>
```

The first thing we added is the require_once in line 3. This includes the source file in which the Model is defined. Then, in the init() function which is called when the controller object is created, an instance of the HRModel is made and stored in the instance variable hrModel that was previously declared. The PHPDoc comment before the variable declaration gives the Zend Studio IDE the information needed to provide code completion for the instance variable hrModel. After this, an hrModel object is present in the IndexController and can be used by any ActionMethod by referencing $this->hrModel. We try this in the indexAction: We call the helloWorld Method we have implemented in our hrModel and output the returned value to the browser using the echo command of php (later we will render all output via so called views to separate logic from layout but for now this should do).
If we now call the home page of our web application, the output will be as follows:
Browser:
http://www.myserver.com/
Result:

```
Hello World
```

Connecting the DB

Since our application will display database content, we will need a database connection to get to the required information. As an example we use the predefined HR schema that comes with Oracle XE and the normal Oracle installation as an example schema. (Please make sure to unlock the Oracle user called HR first and assign it the password hr.)
To connect to a DB from Zend Framework, we use a class called DB Adapter. It can be found in the Package Zend_Db in the Framework distribution. We will also need some credentials to connect to the database. We assume that we use the DB on the local host and login with username and password 'hr'. So where do we put the connection code? Since the Model will use the DB connection in most of its methods, it makes sense to open a connection whenever the Model is instantiated. Therefore we put the connection code into the constructor of the Model and store the connection in an instance Variable of our Model. To check if the connection was successful, we implement a new Method in our Model that fetches the Sysdate from the oracle instance and replace our nice but useless helloWorld function with it:

.../models/HRModel.php:

```
<?php
require_once('Zend/Db.php');

class HRModel {
    /**
    * @var Zend_Db $db
    */
    protected $db = null;

    function __construct() {
        $params = array (
            'username' => 'hr',
            'password' => 'hr',
            'dbname' => '//localhost/XE'
                                                        );

        $this->db = Zend_Db::factory('oracle', $params);
        $this->db->query("alter session set
        NLS_NUMERIC_CHARACTERS = ',.'");
        $this->db->query("alter session set
        NLS_DATE_FORMAT = 'dd.mm.yyyy'");
    }

    /**
    * return the current sysdate of the oracle server
    *
    * @return string
    */
    function getSysDate() {
        $res = $this->db->fetchRow("SELECT sysdate FROM dual");
        return $res['SYSDATE'];
    }
}
?>
```

Note that we needed to include the Zend DB Adapter prior to using it with a require_once statement in line 2. The Model now creates a connection to the Oracle DB in the Constructor of the Model and stores it in the instance variable $this->db for later use. At the same time, some session variables are set for the oracle connection to control number and date formatting. This can be left out if you set the according values generally for the oracle instance. Our new function getSysDate() then uses the Oracle connection that was built in the constructor to select the current system date from the oracle database using the fetchRow() method of the DB object in $this->db. This method sends a query to the database and returns the first row of the result in form of an associative PHP array. We then return the column 'SYSDATE' from this array to the caller of the method. Of course, there are simpler methods in PHP to determine and format the current date. This is only to show that we can make the DB do things for us. Now we need to modify the IndexController to output the result of our brand new database interaction:

.../app/controllers/IndexController.php:

```
<?php
require_once('Zend/Controller/Action.php');
require_once('models/HRModel.php');

class IndexController extends Zend_Controller_Action
{
    /**
    * @var HRModel
    */
    protected $hrModel;

    public function init() {
        $this->hrModel = new HRModel();
    }

    public function indexAction()
    {
        echo $this->hrModel->getSysDate();
    }
}
?>
```

This little change now results in displaying the current date (in German format) instead of the simple Hello world message. The output looks like this:
Browser:
http://www.myserver.com/
Result:

```
01.02.2007
```

Using an external configuration file

With our example model, we could successfully connect to the database. In order to make the example easy to understand, we "hardcoded" the database credentials into the model's source code. In real world applications we do not want to do this since we might have different environments where the code should work without changes afterwards. Therefore we separate this kind of environment information in configuration files. Zend Framework supports different methods of storing such information. In this example we use the ini-file method that parses windows like ini-files with sections and parameters. This is how we integrate the Zend_Config_Ini Component with our existing HRModel:

.../models/HRModel.php:

```
<?php
require_once('Zend/Db.php');
require_once('Zend/Config/Ini.php');

class HRModel {
    /**
    * @var Zend_Db $db
    */
    protected $db = null;

    function __construct() {
        $config = new Zend_Config_Ini('hr.ini', 'staging');

        $params = array (
            'username' => $config->database->user,
            'password' => $config->database->passwd,
            'dbname' => $config->database->dbname
            );
        $this->db = Zend_Db::factory('oracle', $params);
        $this->db->query("alter session set
        NLS_NUMERIC_CHARACTERS = ',.'");
        $this->db->query("alter session set
        NLS_DATE_FORMAT = 'dd.mm.yyyy'");
    }
/**
  * return the current sysdate of the oracle server
  *
  * @return string
  */
  function getSysDate() {
       $res = $this->db->fetchRow("SELECT sysdate FROM dual");
       return $res['SYSDATE'];
  }

}
?>
```

The differences to the previous version of our Model start with the "require_once" statement that includes the declaration of Zend_Config_Ini. Then, before we make the DB connection in the constructor of our Model, we create a Zend_Config_Ini Object from the ini-file hr.ini that we will store in the include directory of our app. The Zend_Config_Ini Object will filter this ini file for the label 'staging' and extract all parameters that match this filter. For detailed information on the format of the ini file and other information about Zend_Config see:
http://framework.zend.com/wiki/display/ZFDOCDEV/4.+Zend_Config
The parameters array for creating the Oracle DB adapter is now no longer assembled from static text but the individual settings are taken from according parameters from the Zend_Config. The config file looks like this:

.../app/include/hr.ini:

```
[staging]
database.user=hr
database.passwd=hr
database.dbname=//localhost/XE
```

Separating Layout from application logic using Views

The last element of MVC that we did not introduce yet is the View. This Component renders the data we got from the model into HTML. In an MVC application, the Model gathers data and triggers transactions, the Controller controls user interaction and page flow and the View is responsible for the output of data in a nice and shiny format. The Zend_View is a class with lots of helpers for the output of plain HTML and Forms. The Controller Methods can create such a view, populate it with dynamic data they want to display on the page and then hand this over to a template like view script. We illustrate this mechanism using the IndexAction in our IndexController:
.../app/controllers/IndexController.php:

```
<?php
require_once('Zend/Controller/Action.php');
require_once('models/HRModel.php');
require_once('Zend/View.php');

class IndexController extends Zend_Controller_Action
{
  /**
  * @var HRModel
  */
  protected $hrModel;

  public function init() {
       $this->hrModel = new HRModel();
  }

  public function indexAction()
  {
       $view = new Zend_View(array('scriptPath' =>
       '<pathToMyApp>/app/views'));
       $view->sysdate = $this->hrModel->getSysDate();
       echo $view->render('index.phtml');
  }
}
?>
```

At first we fetch the Zend_View definition by including the according source file with require_once in line 4. Then in the indexAction, we first instantiate a View Object giving it the path where to find the view scripts (please enter the base path of your application instead of <pathToMyApp> there). In the next line we assign the sysdate we got from the Model to this view object and then we use the view script 'index.phtml' to render an output string from the data we have put into the view before. Note that render() does not output anything but returns a string. This can then be output to the browser using the echo command. To make this example complete, we need our new view script index,phtml. It must be stored in the subdirectory 'views' of our application:

```
.../app/views/index.phtml:
<html>
<body>
Sysdate: <?= $this->sysdate ?>
</body>
</html>
```

Of course this is not a very pretty example of a HTML output page. But it shows the concept: The view script is basically a php script that is executed inside a method of the Zend_View object. Therefore it can access all resources of Zend_View referencing $this->. An example is the sysdate that was put into the view object by the controller before the view script was executed during the call to $view->render(). Sysdate is an instance variable of the view object and thus can be referenced using $this->sysdate. Since the php interpreter is in HTML mode when the view script is executed, it treats all content of the view script as HTML until it finds an opening PHP tag (<?). Generally it is possible to embed any PHP code in a view script but it is recommended to use as little php as possible there. To output variables from the viw, the short print notation of php can be used (as in the example). The format of this notation is:
<?= <expression> ?>
... while expression can be any valid php expression including calls to php internal functions etc. In this case we just output the contents of the instance variable sysdate. When we now call our home page again, we still get:
Browser:
http://www.myserver.com/
Result:

```
Sysdate: 01.02.2007
```

In contrast to the output we have seen before, result page is now well formatted HTML with start and end tags for HTML and BODY.

The HR Application

We now want to write a small Application that lists the employee records from the HR database that comes with the default installation of Oracle 10g or Oracle XE.

Listing all Employees

First we need a Controller Action that fetches the list from the db and outputs the result to the browser. To make things as simple as possible, we use the indexAction from the IndexController for this. The user will therefore be prompted with the employee list when he starts the application by calling the homepage.
The SQL query to fetch all employees and format them nicely is given as follows:
SQL:

```
SELECT employee_id, substr(first_name,1,1) || '.  '|| last_name as
employee_name,
    hire_date, to_char(salary, '9999G999D99') as salary,
    nvl(commission_pct,0)*100 as commission_pct,
    d.department_name,        j.job_title
    FROM employees e, departments d, jobs j
    WHERE e.department_id =d.department_id   and e.job_id = j.job_id
    ORDER BY employee_id asc
```

As a first step, we integrate this query in our HR Model and return the resulting rows in form of an array of result rows. Each of these rows is an associative array itself.

.../models/HRModel.php:

```
<?php
require_once('Zend/Db.php');
require_once('Zend/Config/Ini.php');

class HRModel {
    /**
    * @var Zend_Db $db
    */
    protected $db = null;

    function __construct() {
    ...
    }
    ...

    /**
    * returns a list of all employees in an assoc. array
    *
    * @return array
    */
    public function queryAllEmployees() {
        return $this->db->fetchAssoc(
            "SELECT employee_id,
            substr(first_name,1,1) || '.  '|| last_name as employee_name,
            hire_date, to_char(salary, '9999G999D99') as salary,
            nvl(commission_pct,0)*100 as commission_pct,
            d.department_name, j.job_title
            FROM employees e, departments d, jobs j
            WHERE e.department_id =d.department_id AND e.job_id
            = j.job_id
            ORDER BY employee_id asc");
    }

}
?>
```

We just wrapped the complete Query in a Method declaration and a call to $this->db->fetchAssoc(). The result of fetchAssoc is the desired list in form of a list of associative arrays (one for each row).
Now we call this new method from our indexAction and store the resulting array in the View Object:

.../app/controllers/IndexController.php:

```
<?php
require_once('Zend/Controller/Action.php');
require_once('models/HRModel.php');
require_once('Zend/View.php');

class IndexController extends Zend_Controller_Action
{
    /**
    * @var HRModel
    */
    protected $hrModel;

    public function init() {
        $this->hrModel = new HRModel();
    }

    public function indexAction()
    {
        $view = new Zend_View(array('scriptPath' =>
        <pathToMyApp>/app/views'));
        $view->employeeList = $this->hrModel->queryAllEmployees();
        $view->sysdate = $this->hrModel->getSysDate();
        echo $view->render('index.phtml');
    }
}
?>
```

Now we have the result in our View Object. The next step would be to display the result in the View Script index.phtml:

.../app/views/index.phtml:

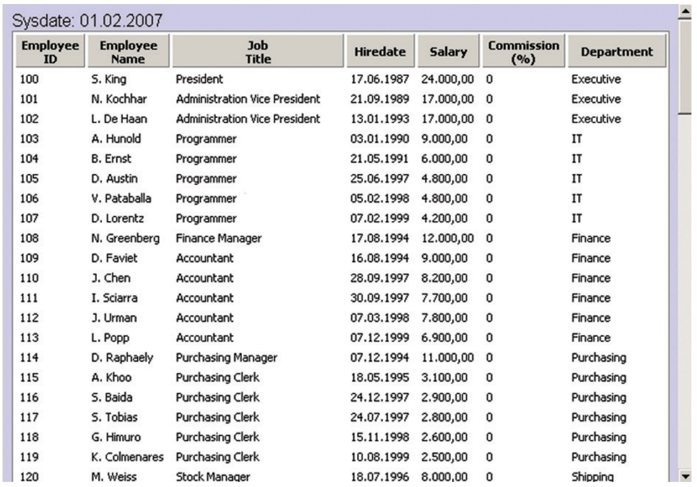
```
<html>
<head>
<link rel="stylesheet" type="text/css" href="/style.css" />
</head>
<body>
Sysdate: <?= $this->sysdate ?>
<table>
    <tr>
        <th>Employee<br>ID</th>
        <th>Employee<br>Name</th>
        <th>Job<br>Title</th>
        <th>Hiredate</th>
        <th>Salar[y</th>
        <th>Commission<br>(%)</th>
        <th>Department</th>
    </tr>
<?php
    // Write one row per employee
    foreach ($this->employeeList as $emp):
    extract($emp);
        echo <<<END
    <tr>
        <td align="right">$EMPLOYEE_ID</td>
        <td>$EMPLOYEE_NAME</td>
        <td>$JOB_TITLE</td>
        <td>$HIRE_DATE</td>
        <td align="right">$SALARY</td>
        <td align="right">$COMMISSION_PCT</td>
        <td align="right">$DEPARTMENT_NAME</td>
    </tr>
END;
    endforeach;
?>
</table>
</body>
</html>
```

In addition to the basic view script that only displays the sysdate, some changes were introduced here. At first, we added a link to a style sheet named style.css that improves the layout a little:

.../htdocs/styles.css:

```
body {
    background:     #CCCCFF;
    color:          #000000;
    font-family:    Arial, sans-serif;
}
h1 {
    border-bottom: solid #334B66 4px;
    font-size: 160%;
}
table {
    width:          100%;
    font:           Icon;
    border:         1px Solid ThreeDShadow;
    background:     Window;
    color:          WindowText;
}
td {
    padding:        2px 5px;
    vertical-align: top;
    text-align:     left;
}
th {
    border:         1px solid;
    border-color:   ButtonHighlight ButtonShadow
                    ButtonShadow ButtonHighlight;
    cursor:         default;
    padding:        3px 4px 1px 6px;
    background:     ButtonFace;
}
```

Further, we integrated a HTML table that will show the result list. This table has a first row with column titles in plain HTML. Then for the output of the actual row content, we switch to PHP mode in the view script (<?php). Using the foreach control structure of PHP, we iterate over the employeeList array which had been stored in the view object by the Controller Action before.
To make the loop more visible inside the HTML fragments in this file, we do not use the normal bracket notation to denote the foreach-block but we use an alternative format. This format starts the block with a colon and ends it with the endforeach statement. The bracket notation could also be used here but the alternative notation seems better inside templates.
The foreach block is executed once per row in the employeeList variable, meaning once per result row. Inside the block, the current row can be referenced via the variable $emp. The individual columns of the result row could now be referenced by using their names as an index to the array $emp. For example $emp['EMPLOYEE_NAME'] would reference the name column from the current row.
To make references to the fields of the result row simpler, we use the extract() command of php on $emp. This command creates one local variable for every key in the given array and copies the associated value in this new variable. Instead of using $emp['EMPLOYEE_NAME'] it is then possible to reference the name by simply using $EMPLOYEE_NAME.
After the extract command we use the HEREDOC syntax of PHP to echo one table row to the browser. This notation allows multi line strings in which variables are automatically replaced by their values. Also, quotation marks and other special characters can be used without the need to escape them inside a string. The string to echo ends with and END marker at the beginning of a new line. Calling our home page now results in a employee list:
Browser:
http://www.myserver.com/
Result:



Sysdate: 01.02.2007

Employee ID	Employee Name	Job Title	Hiredate	Salary	Commission (%)	Department
100	S. King	President	17.06.1987	24.000,00	0	Executive
101	N. Kochhar	Administration Vice President	21.09.1989	17.000,00	0	Executive
102	L. De Haan	Administration Vice President	13.01.1993	17.000,00	0	Executive
103	A. Hunold	Programmer	03.01.1990	9.000,00	0	IT
104	B. Ernst	Programmer	21.05.1991	6.000,00	0	IT
105	D. Austin	Programmer	25.06.1997	4.800,00	0	IT
106	V. Pataballa	Programmer	05.02.1998	4.800,00	0	IT
107	D. Lorentz	Programmer	07.02.1999	4.200,00	0	IT
108	N. Greenberg	Finance Manager	17.08.1994	12.000,00	0	Finance
109	D. Faviet	Accountant	16.08.1994	9.000,00	0	Finance
110	J. Chen	Accountant	28.09.1997	8.200,00	0	Finance
111	I. Sciarra	Accountant	30.09.1997	7.700,00	0	Finance
112	J. Urman	Accountant	07.03.1998	7.800,00	0	Finance
113	L. Popp	Accountant	07.12.1999	6.900,00	0	Finance
114	D. Raphaely	Purchasing Manager	07.12.1994	11.000,00	0	Purchasing
115	A. Khoo	Purchasing Clerk	18.05.1995	3.100,00	0	Purchasing
116	S. Baida	Purchasing Clerk	24.12.1997	2.900,00	0	Purchasing
117	S. Tobias	Purchasing Clerk	24.07.1997	2.800,00	0	Purchasing
118	G. Himuro	Purchasing Clerk	15.11.1998	2.600,00	0	Purchasing
119	K. Colmenares	Purchasing Clerk	10.08.1999	2.500,00	0	Purchasing
120	M. Weiss	Stock Manager	18.07.1996	8.000,00	0	Shipping

Editing an Employee

Now that we have a list, we might want to display a record in detail and edit its contents. For example we could want to add a commission or to change department, job title or telephone number. To do so, we first need a link to a new action called editForm in the List. When the user clicks on the id of an employee, he gets to a form with the details of the according user. So first we add the new link. This can be done in the View Script:

```
.../app/views/index.phtml:
<html>
...
<?php
    // Write one row per employee
    foreach ($this->employeeList as $emp):
    extract($emp);
          echo <<<END
          <tr>
                <td align="right"><a href="/index/edit/id/
                $EMPLOYEE_ID">$EMPLOYEE_ID</a></td>
                <td>$EMPLOYEE_NAME</td>
                <td>$JOB_TITLE</td>
                <td>$HIRE_DATE</td>
                <td align="right">$SALARY</td>
                <td align="right">$COMMISSION_PCT</td>
                <td align="right">$DEPARTMENT_NAME</td>
          </tr>
END;
    endforeach;
?>
</table>
</body>
</html>
```

The new link calls the method editAction() in our IndexController. But what does the rest of the URL mean? To edit an employee we need to select which record to edit. We do that by transferring the EMPLOYEE_ID along in the URL. Usually this is done with GET parameters in the style 'http://www.myserver.com/index.php?id=xyz'. To make the URLs prettier, Zend Framework offers another option to pass values to the action controllers: If controller name and action name are both given, all path elements suceeding the action name are handled as key/value pairs for parameter passing. Therfore the URL in our new link is interpreted in the following way:
example URL: "/index/edit/id/100"

Controller:	IndexController
Action:	editAction
Parameter1:	id=100

What we need next is a new action method in the indexController. It looks like this:
.../app/controllers/IndexController.php

```
<?
require_once ...
class IndexController extends Zend_Controller_Action
{
    ...
    public function editAction() {
          $id = (int)$this->_getParam('id');
          die('edit called for id:'.$id);
    }
}
?>
```

By calling the method $this->_getParam($name) of the Action Controller, our new method obtains the value of the id passed in the URL as mentioned above. For security reasons, we cast the returned value to an integer. This eliminates all characters that are not digits and therefore makes attacks such as Cross Site Scripting and SQL injection impossible. Note that all data that is coming from outside the application should go through "white list" input filters. That means, only values that are definitely legal values for the according variable should be accepted. Everything else must be filtered out or trigger an error. The die() command tells php to stop the execution of the program at once after displaying the contents of the first argument of the die() function. In our case, it is only a debug output to show that the method was called and the value of the id field was transferred correctly.
Now we have created a new action that can be called by clicking on the Employee-ID in the employee list. Clicking on the id of the first user in the list generates the following output:
edit called for id:100
Now we want to replace the die() call with something useful: The form to edit the profile of an employee. To do this, we need to accomplish a number of tasks:

1. fetch the record of the employee from the database
2. fetch the option lists for departments and job names from related tables to display them as choices in dropdown lists
3. display a form with different input fields with the data in them pre filled

We start with a function to fetch data from the database. This function is implemented in the HRModel:

.../app/HRModel.php:

```
<?php
require_once ...
class HRModel {
    ...
    /**
    * find a specific employee by his employee_id and return
    * the data in an assoc. array.
    *
    * @param int $eid
    * @return array
    */
    function findRecord($eid) {
          $myvars['EMPID'] = $eid;

          $res = $this->db->fetchAssoc("SELECT employee_id, first_name,
          last_name,
                email, hire_date, salary, (nvl(commission_pct,0)*100) as
                commission_pct,
                department_id, job_id
                FROM   employees
                WHERE  employee_id = :empid", $myvars);
          return $res[$eid];
    }
}
?>
```

It should be easy to replace the die() in the controller by a call to this function and a var_dump() of the returned array. This way the function can be tested before we continue. This is skipped here.
The next we need to do is to store the information obtained from the Model in the view object and write a view script to render a form for us that has these values as default values in the form elements. So at first we modify our action method:

.../app/controllers/IndexController.php:

```
<?
require_once ...
class IndexController extends Zend_Controller_Action
{
    ...
    public function editAction() {
        $id = (int)$this->_getParam('id');
        $view = new Zend_View(array('scriptPath' =>
<pathToMyApp>/app/views'));
        $view->employeeDetail = $this->hrModel->findRecord($id);
        echo $view->render('edit.phtml');
    }
}
?>
```

The action method now stores the result of the findRecord call in the view object and then uses the view script called 'edit.phtml' to render the form. Now we code edit.phtml:

.../app/views/edit.phtml:

```
<?php
    extract($this->employeeDetail);
?>
<HTML>
<head>
<link rel="stylesheet" type="text/css" href="/style.css" />
</head>
<body>
    <form method="post" action="/index/save">
    <?= $this->formHidden('EMPLOYEE_ID',$EMPLOYEE_ID) ?>
    <table>
    <tr>
        <td>First Name</td>
        <td><?= $this->formText('FIRST_NAME',$FIRST_NAME) ?></td>
    </tr>
    <tr>
        <td>Last Name</td>
        <td><?= $this->formText('LAST_NAME',$LAST_NAME) ?></td>
    </tr>
    <tr>
        <td>E-Mail</td>
        <td><?= $this->formText('EMAIL',$EMAIL) ?></td>
    </tr>
    <tr>
        <td>Hiredate</td>
        <td><?= $this->formText('HIRE_DATE',$HIRE_DATE) ?></td>
    </tr>
    <tr>
        <td>Salary</td>
        <td><?= $this->formText('SALARY', $SALARY) ?></td>
    </tr>
    <tr>
        <td>Commission (%)</td>
        <td><?= $this->formText('COMMISSION_
        PCT',$COMMISSION_PCT) ?></td>
    </tr>
    </table>
        <input type="submit" value="Save" name="save">
        <input type="submit" value="Cancel" name="cancel">
    </form>
    </body>
</HTML>
```

At first, we switch to PHP mode in this view script to extract the associative array from $this->employeeDetail into the local scope. This way, all the different fields of the currently edited record are accessible via variables with the name of the according DB field (as seen already in the employeeList). The we start with linking our style sheet again and opening a normal HTML form. The form action will be index/save, a new action that we must integrate in our IndexController. The rest of the page is all like a normal HTML form with the exception that all the input fields are generated dynamically via so called View Helpers. These helpers are a library of methods that the Zend_View provides for rendering certain HTML tags. In this example we use $this->formHidden() to generate a hidden field and $this->formText() to generate normal text input fields. The first parameter to these Helpers is always the name of the input field and the second parameter is the current value. Information about other field types can be obtained from the Zend Framework Website.
Now we code the saveAction to store the changed values in the database. This is done again in the IndexController. In order to do this we must first add an update function to the hrModel. This will take an EMPLOYEE_ID and an associative array of fieldnames and values and generate an update query.

.../app/models/HRModel.php

```
<?php
require_once...
class HRModel {#
    ...
        /**
        * update all fields that are present in the assoc array
        * $row for the employee given in $eid
        *
        * @param int $eid
        * @param array $row
        * @return int Number of affected rows
        */
    public function update($eid, $row) {
        $where = $this->db->quoteInto('EMPLOYEE_ID = ?', $eid);
        return $this->db->update('employees', $row, $where);
    }
}
?>
```

To accomplish this task, we use the update function of the Zend_DB Adapter. It requires a table name, an associative array of key/value pairs for the updated values and a where clause to determine which records to update. The where clause must first be generated by inserting the given employee-id into a query fragment with placeholders. The where clause is generated this way to enable the DB Adapter to quote the value of $eid correctly for the given database. This prevents SQL injection attacks in the where clause. The other values are quoted in the update method of the DB adapter.
After we have added the needed functionality to the Model, we can now extend our IndexController to fetch the data from the HTML form, store it in the array that the update function needs and call the update function in the Model:

.../app/controllers/IndexController.php

```
<?
require_once ...
class IndexController extends Zend_Controller_Action
{
    ...
        function saveAction() {
        // read input variables (should be checked in real application)
        $row = array();
        $row['FIRST_NAME'] = $_POST['FIRST_NAME'];
        $row['LAST_NAME'] = $_POST['LAST_NAME'];
        $row['EMAIL'] = $_POST['EMAIL'];
        $row['HIRE_DATE'] = $_POST['HIRE_DATE'];
        $row['SALARY'] = $_POST['SALARY'];
        $row['COMMISSION_PCT'] =
        number_format($_POST['COMMISSION_PCT']/100,2,',','.');
        $this->hrModel->update($_POST['EMPLOYEE_ID'],$row);
        $this->indexAction();
    }}
?>
```

Please note that this action contains no input filtering or validation. In a real world application you would use the Zend_Validate and Zend_Filter or other components to check all the data fields from the $_POST array with a white list approach before using them in the application to prevent hacking. And also you would want to check if the values submitted by the user and re-display the form with according error message if the user entered invalid data or left required fields blank. This has been left out here to keep the example simple. In the current preview release of Zend Framework there is no standard form component in Zend Framework yet that helps you with this part. Such a component is under development and will be available in future versions of Zend Framework. After copying all input values from the POST request to the new associative array $row, the action method passes this array along with the EMPLOYEE_ID to the update Method of the hrModel. After a successful update the index action is called which results in re-displaying the Employee-List.

Epliogue

We have seen in this session how to install Zend Framework and how to build a simple web application with it based on the HR schema that comes with every Oracle installation as an example database schema. Our sample application lists all employees and offers a way to edit and save employee records with a simple form. This shows some of the basic functions and components of Zend Framework. To turn this rough example into a usable application, a number of additional things would be needed:

- First of all the save method needs to be extended to do input validation on the submitted fields.
- Some fields from the database were left out in this example application because they require some more coding. These are the fields DEPARTMENT_ID and JOB_ID which should be implemented as drop down boxes with all rows from the according database tables as options.
- There is a trigger in the database that requires to update the field HIRE_DATE to the current date whenever JOB_ID or DEPARTMENT_ID are changed. This logic would also need to be implemented.
- Apparently, a function to add new Employees would be needed and a delete function would also make sense.
- The structure could be further optimized by extracting the SCRIPT_PATH variables from the source code lines in which view objects are instantiated. These path information as well as the Controller Path in the index.php should be taken from an ini file.

Although there is much room for improvement, this little example shows how a structured PHP application can be built with the help of Zend Framework. The standard application structure and naming conventions help to build the application in the right way. This gives other developers the chance to start quickly into application development when they join the project later and makes maintenance and extending the project much easier and more efficient.

For further details about Zend Framework look at http://framework.zend.com

About Zend Technologies

Zend Technologies Inc., the PHP Company, is the leading provider of products and services for developing, deploying and managing business-critical PHP applications. PHP is used by more than twenty-two million Web sites and has quickly become the most popular language for building dynamic web applications. www.zend.com

ZEND: The holistic approach to PHP

- Application management and availability with Zend Platform™
- Development of PHP applications with Zend Studio™, the leading development environment for PHP
- Certified and officially supported PHP installations with Zend Core™
- Access to expertise of leading PHP experts with Zend Professional Services™
- Improved PHP knowledge through Zend Training™ offers
- Protection of intellectual property and source code and administration of licensing models with Zend Guard™
- The certified and manufacturer-supported collection of PHP components and PHP libraries – Zend Framework™
- First class 24/7 support via the Zend Network™

Zend
The *php* Company
w w w . z e n d . d e

Corporate Headquarters:

Zend Technologies, Inc.
19200 Stevens Creek Blvd.
Cupertino, CA 95014
Tel: 1-888-PHP-ZEND
1-888-747-9363
Fax: 1-408-253-8801

Central Europe:

Zend Technologies GmbH
Bayerstrasse 83
80335 Munich, Germany
Tel: +49-89-516199-0
Fax: +49-89-516199-20
E-Mail: info-germany@zend.com

International:

Zend Technologies, Ltd.
12 Abba Hillel Street
Ramat Gan, Israel 52506
Tel: 972-3-753-9500
Fax: 972-3-613-9501

UK:

Zend Technologies
50 Basing Hill
London NW11 8TH, United Kingdom
Tel.: +44 20 8458 8550
Fax: +44 20 8458 8550

Italy:

Zend Technologies
Largo Richini 6
20122 Milano, Italy
Tel.: +39 02 5821 5832
Fax: +39 02 5821 5400

France :

Zend Technologies SARL
5, Rue de Rome, ZAC de Nanteuil
93110 Rosny sous Bois, France
Tel : +33 1 4855 0200
Fax : +33 1 4812 3132