# The City Lit Institute

##### Department of Computing

## Keeley Street, Holborn, London WC2B 4BA

**OO PHP**

**(with Apache server)**

**LECTURER** : **ALEXANDER ADU- SARKODIE**

MSc. Telematics (IT & Telecom), MSc. Eng., Dip. Russ. Lang., Teach. Cert, AMIAEng (UK), MBCS (UK), MIfL(UK)

[**http://www.blogger.com/profile/14800490193632788559**](http://www.blogger.com/profile/14800490193632788559)

**Introduction to Zend Framework**

**Zend Framework** is an open source, object-oriented web application **framework** implemented in PHP 5. It is also a design pattern. Zend Framework requires PHP 5.2.4 or later since version 1.7.0

Components of Zend Framework:

* Provides **fully** object-oriented support (PHP 5)
* Uses **flexible architecture** with **loosely coupled components** and **minimal interdependencies**
* Extensible **MVC (Model, View, Control)** implementation supporting **layouts** and **PHP-based templates by default**
* Support for **multiple database systems and vendors**, including MySQL, Oracle, PostgreSQL and others.
* Flexible **caching sub-system** with support for many types of back-ends, such as memory or a file system.

**Patterns**

Patterns are ways to describe best practices and good designs. They show a flexible solution to common programming problems.

**MVC**

Model View Control (MVC), is a **design pattern** that **simplifies application development and maintenance**. Adopting this pattern leads to **separation** of **presentation layer, business logic,** their **relationships,** cleanliness of code, ease of upgrading and maintenance.

The **MODEL**  is responsible for the **business logic**.

The **VIEW** is responsible for **design template, styling and layout**.

The **CONTROLLER** glues everything together, **merging the styling of the view with the functionality of the model.**

**Concepts**

**Bootstrapping:** This integrates the web server environments with Zend Frameworks. Sending **request for the site**, and handle **application despatching**. The Bootstrapping mechanism is responsible for including files and **initializing the front controller**. Using **auto-loading mechanism i.e require components**.

**Front Controller:** The job of the Front Controller is **to parse the URL ‘s** and **to instantiate and invoke the action method**. This is a singleton class.

So, request comes first to bootstrapping and then forwarded to front controller for parsing before despatching.

**Auto-loading**: Many developers writing object-oriented applications create one PHP source file per-class definition. One of the biggest annoyances is having to write a long list of needed includes at the beginning of each script (one for each class).

In PHP 5, this is no longer necessary. You may define an **\_\_autoload** function which is automatically called in case you are **trying to use a class/interface which hasn't been defined yet.** By calling this function the scripting engine **is given a last chance to load the class before PHP fails with an error.**

**Singleton**: A singleton is a **responsibility pattern**. It creates a single point of functionality within an application. The Singleton ensures that **there can be only one instance of a Class and provides a global access point to that instance**. The Singleton pattern is often implemented in Database Classes, Loggers, Front Controllers or Request and Response objects.

The Singleton pattern is one of the more controversial patterns(defeating at times loosely coupled component property between objects). Critics argue that Singletons introduce **Global State** into an application and tightly couple the Singleton and its consuming classes. This leads to hidden dependencies and unexpected side-effects, which in turn leads to code that is harder to test and maintain.

Critics further argue that it is pointless to use a Singleton PHP where objects are unique within the Request only anyways. It is easier and cleaner to create collaborator object graphs by using Builders and Factory patterns once at the beginning of the Request.

Singletons:

* Must have a **constructor**, and must be **marked private**
* Contain a **static member variable** that **withholds an instance of a class**
* Contain **a public static method** to **access the instance**

**Factory pattern:** Factory patterns are used in **polymorphism**. The Factory pattern allows for **the instantiation of objects at runtime**. It is called a Factory Pattern since it is responsible for "**manufacturing**" an object. A Parameterized Factory receives the **name** of the class to instantiate as **argument**.

**Creating the Index Controller and the View**

1. **Create Index Controller:**

This is the default controller filename.

This file should be saved to this repository **application/controllers/IndexController.php**

<?php

#Class {**Type**}Controller **extends** Zend\_Controller\_**Action**

**#extends** allows inheritance from Zend\_Controller\_Action

**#{**Type of Controller} **= Index**

#Method name : {**Type**}{**Action**}

Class **Index**Controller **extends** Zend\_Controller\_Action {

public function **IndexAction**() {

$this->view->name = ‘Kevin’;

}

}

?>

1. **Create the View:**

This file should be saved to this repository **application/views/scripts/index/index.phtml**

<html>

<body>

<p> Welcome: <php echo $this->name; ?> </p>

</body>

</html>

**Creating the Customer Model, Controller and View**

1. **Create Customer Model:**

This model should be saved to this repository **application/models/customers.php**

<php

Class Customers **extends** Zend\_DB\_Table {

protected $\_name = ‘name’;

protected $\_primary = ‘customer\_id’;

}

?>

1. **Create Customer Controller Action**

This controller should be saved to this repository **application/controllers/CustomersController.php**

Here, we set up an environment with a class that

* Inherits from the parent class (Zend\_Controller\_Action)
* Write a method (IndexAction in this case) with a scope descriptor (public)
* Instantiate an object from the inherited class
* Pass or expose the object instance to the view

<?php

Class CustomersController extends Zend\_Controller\_Action {

public function indexAction() {

$table = new Customers(); //**instantiate** an object from the **Customers** class

//we expose the object (instance) to the view

$this->view->customers = $table->fetchAll();

}

}

?>

1. **Create Customer View**

This view should be saved to this repository **application/views/scripts/customers/index.phtml**

Next step is to create the view for this information. Note you will need to create a new directory to contain the views for the customers controller. Since an IndexAction method was created in the controller so the route to the customer page will be: application/views/scripts/customers/index.phtml

<tbody>

<?php

//We use the foreach loop to return individual unique records from the dataset

//In the data set we return node properties (name value pairs)

// from the object which is expected as an array

//$customer= Array (‘customer\_id’= ‘1’ , ‘name’= ‘Alex’, ‘customer\_id’ = ‘2’, ‘name’ = ‘Ben’);

foreach($this->customers as $customer) {

?>

<tr>

<td>

<?php echo $customer->customer\_id ?>

</td>

<td>

<?php echo $customer->name ?>

</td>

</tr>

<?php } ?>

</tbody>