



phpObLib

Document Title

About ObLib

Document Subtitle

An introduction to the ObLib Architecture and the ObLibGation SSDLC

Version Number

1

Published On

10 February 2008

Author

Bashkim Isai

Website

<http://www.phpoblib.com/>

Document Licence

**Creative Commons Attribution-Non Commercial-No Derivative Works 3.0
Unported License**

Licence Website

<http://creativecommons.org/licenses/by-nc-nd/3.0/>

Copyright and Licensing Information

The ObLib Architecture is released under the [Creative Commons Attribution-Share Alike 3.0 Unported License](#). In order to use the ObLib Architecture, you must attribute the ObLib Architecture by inserting the following HTML Meta tag in the head of your document (used for statistical purposes):

```
<meta name="generator" content="phpObLib" />
```

This document is released under the [Creative Commons Attribution-Non Commercial-No Derivative Works 3.0 Unported License](#).

Copyright © Bashkim Isai, 2008.

Some Rights Reserved.

Table of Contents

Copyright and Licensing Information.....	b
Principles of the ObLib Architecture.....	1
ObLib Mission Statement.....	1
Project Success.....	1
The Clashing of two Logics in History.....	1
Problems in Web Development Project Management.....	2
Initial Problem.....	2
General Solution.....	2
Cascading Problem.....	3
General Solution.....	3
Cascading Problem.....	3
General Solution.....	3
Cascading Problems.....	3
Consequences.....	3
The Greater Divide.....	3
Gap-Bridging the Greater Divide.....	5
Structure of the ObLib Architecture.....	7
The Responsibilities of the ObLib Architecture.....	7
Model View Controller [MVC] Programming.....	7
MVC Components.....	8
Controller.....	8
View.....	9
Model.....	10
MVC Lifecycle.....	11
The ObLibGation Process.....	13
ObLib Entity Mapping Process.....	13
Important Considerations for OEMP.....	14
Data Types for Variables.....	14
Class and Entity Names (Singular over Plural).....	15
Programming the ObLib Architecture.....	17
Random Number Generator.....	17
Learning Outcomes.....	17
Project Synopsis.....	17
Processes.....	17
Create an Interrupt Handler.....	18
Reference the Controller.....	19
Create the Controller.....	20
Create the View.....	22
Viewing the Result.....	23
Conclusion.....	23
Retrieving information from MySQL.....	24

Learning Outcomes	24
Project Synopsis.....	24
Processes	24
Setup your Database Parameters	25
Setup Server Identifications.....	25
Identify the Current Server	25
Define the Root Directory	26
Define the MySQL Connection Parameters	26
OEMP: ObLib Entity Mapping Process.....	27
City.....	27
Country.....	27
CountryLanguage.....	28
Creating the OXRM Document.....	28
Create your dataEntity and dataEntities classes	35
Create the Entity Skeleton	35
Reference the Entity Classes	36
Create an Interrupt Handler	37
Create the Controller.....	37
Creating a Search Method	38
Outputting the Results.....	40
Viewing the Result	41
Conclusion	41
Frequently Asked Questions.....	43
Backend Programming.....	43
Forcing NULL values instead of empty values in Databases	43
OXRM.....	44
OXRM type Element in OXRM field Restrictions.....	44
Bibliography.....	45

Principles of the ObLib Architecture

ObLib Mission Statement

The aim of ObLib is to provide a structure for PHP application development where Interface Logic (Graphic Design) and Programming Logic (Backend Logic) are developed concurrently by two separate teams of varying amounts of people without interrupting or disturbing in workflow.

Project Success

For the purposes of this document, a definition must be made to gauge a successful project. In the field of Information Technology, project managers are not able to manage tangible media, which means it is sometimes difficult to determine how a project is progressing through its software development lifecycle. By comparison, [paraphrasing] “a project manager for an engineer firm can stand back and see how far a bridge spans between two riversides” (Jewels, 2007).

In the book *Project Management for Information Systems*, Cadle and Yeates define a successful project as one that finishes “on time, within budget and to the customer’s satisfaction” (Cadle & Yeates, 2001). Furthermore to this definition of success, ObLib also aims to create a methodology of project development which allows the expansion of development through standardised development architecture.

Therefore, a successful project in the ObLib realms can be defined as one that is “on time, within budget, to the customer’s satisfaction, scalable and easy to continue development”.

The Clashing of two Logics in History

When the World Wide Web was created in 1991 by Sir Tim Berners-Lee, the requirement for interface logic was minimal. Websites were developed using simple HTML mark-up and were occasionally accompanied by images. This method of development continued for around half a decade before any major changes to the World Wide Web were made.

Through the late 1990’s, the number of computers in households were rapidly increasing and computers themselves became more readily available. This made way for commercial enterprises to utilise the Internet to reach their customers. Commercial enterprises spent millions of dollars advertising and selling their products through this medium, causing a flurry of new enterprises to be setup for online-only trading; an era

commonly known as the *Dot-Com Bubble* and occurred around the beginning of the new millennium. Shortly after (in 2002) followed the *Dot-Com Bust*, when investors lost vast amounts of money due to Internet sites going out-of-business because of the lack of consumer interest in purchasing goods and services online.

For a list of the top 10 dot-com busts, check out CNET's article at:

http://www.cnet.com/4520-11136_1-6278387-1.html.

Technology economists believe that many factors contributed to the Dot-Com Bust. An abridged list of general problems that arose for online organisations include:

- The lack of trust from the consumer in the merchant to give out credit card information (which was potentially unsafe via the internet) in order for goods to be dispatched and delivered.
- Poorly designed navigational and search features for websites which made it difficult for consumers to find the products that they were looking for.
- Poorly maintained websites with out-dated and useless clutters of information.
- Poorly managed enterprises which did not end up “going live” due to poor development infrastructure. Interface logic being developed by separate teams to backend Programming Logic caused a multiplicity of problems with clashes of personalities, egos and values in what was “more important”; an attractive or a functional service.

Problems in Web Development Project Management

Generally speaking, the managers in web development firms found the following problems.

Initial Problem

Consumers were not able to find the information that they needed from the service online, resulting in less sales and adversely affecting the business's bottom-line. Websites were not visually appealing and it was easier to sell products in a more tangible environment (such as in brick-and-mortar stores).

General Solution

Utilise a specialised team of Graphic Designers who would make the website more appealing and easier to navigate for the Customer.

Cascading Problem

Web Development Managers did not understand how to structure a project so that Graphic Designers and Backend Programmers could develop concurrently without difficulty.

General Solution

Have the interface design of the website be created first so that backend programmers could “plug-in” their functionality to the Website.

Cascading Problem

Programmers couldn't understand the complex nature of the graphic design and negatively reproduced the website in a “bastardised fashion”.

General Solution

Reverse the situation. Programmers would make unappealing and unattractive bare websites (usually just black text on a white background) using simple HTML that designers could plug in their designs.

Cascading Problems

Navigational structure/searching became even more of a problem for consumers accompanied with the requirement that Graphic Designers needed to learn some coding that allowed them to display information to the Consumer (e.g.: database values and variables).

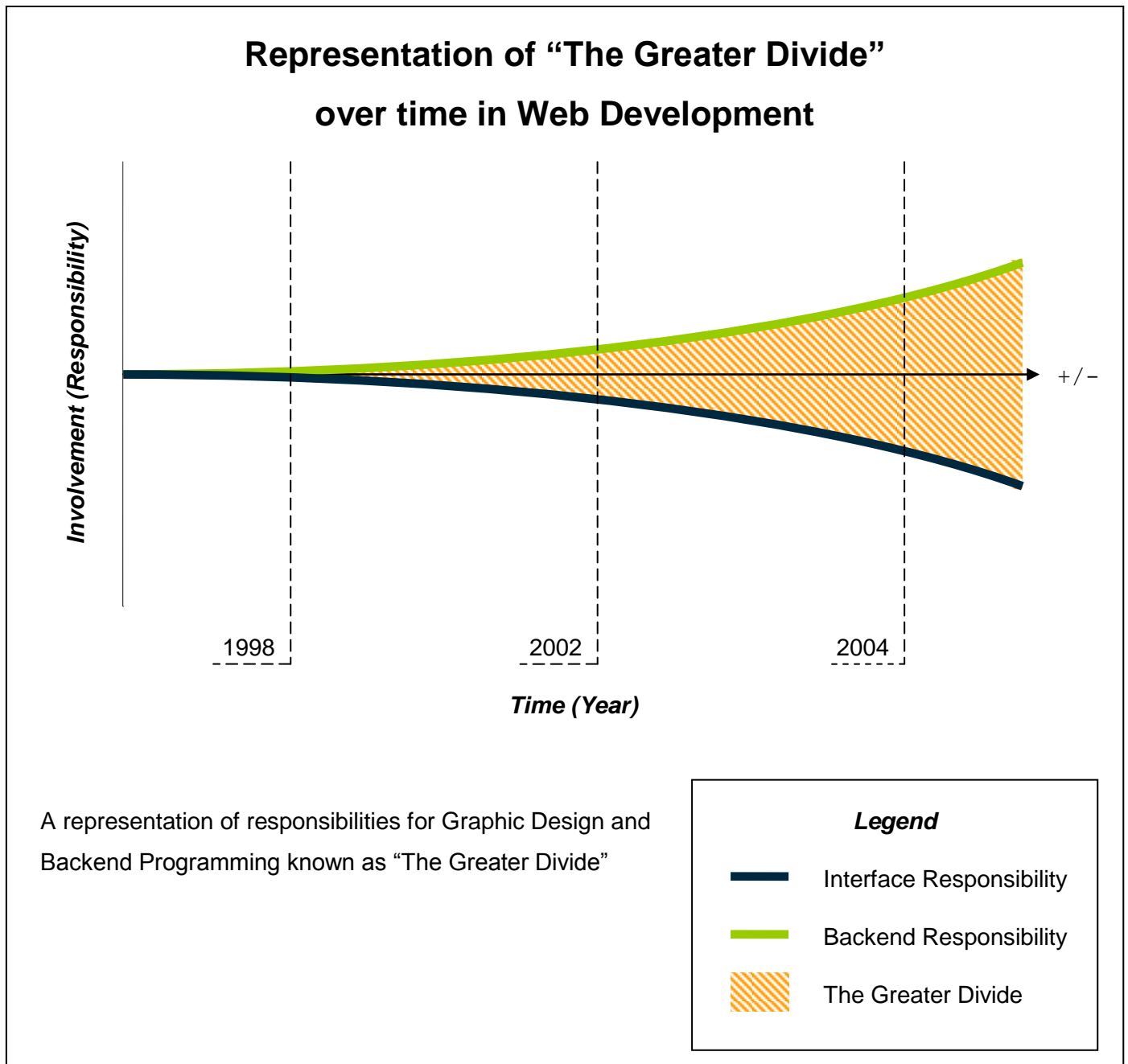
Consequences

By this time, most poorly managed development projects had gone out-of-business because of the continual redevelopment that was required to keep both the customer [the website visitor] and the clients/project sponsors satisfied. Throughout the entirety of development, clients would request changes to the scope of the project which would cause conflicts between the Graphic Design, the Navigation Structure, the Database Design and the Backend Programming Logic. This became yet another difficulty for web development companies to handle in the development of web-based application projects. Usually the project would fail due to its inability to meet the requirements of “project success”.

The Greater Divide

The Greater Divide is a term that I wish to coin which refers to the divide which occurred between the late 20th century through to the early new millennium between Interface Logic and Programming Logic. The

Greater Divide refers to the gradual relinquished responsibility of Interface Logic from a single programmer or programming department and the increased amount of professional work required from a graphic designer.



The Greater Divide has caused problems in project development because it forms disunity and discontinuity between the Interface Logic and Programming Logic. Even with an excellent project specification developed at the beginning of the Software Development Lifecycle (SDLC), projects will still inevitably fall into the Greater Divide gap because of the amount of work required for a Graphic Designer and a Programmer to re-specify the project, defining the changes that need to be made, applying the

additions to the project and updating the previous architecture of the site (both in the Interface design, the database and the class hierarchy) to reflect the new specification.

Gap-Bridging the Greater Divide

The aim of ObLib is to create a flexible and expandable architecture (in PHP for Backend Programming and XHTML/XSLT for Graphic Designers) which allows for the rapid development of web applications with graphic designers and backend programmers developing individual components concurrently without interruptions or disruptions in workflow from the other team.

Structure of the ObLib Architecture

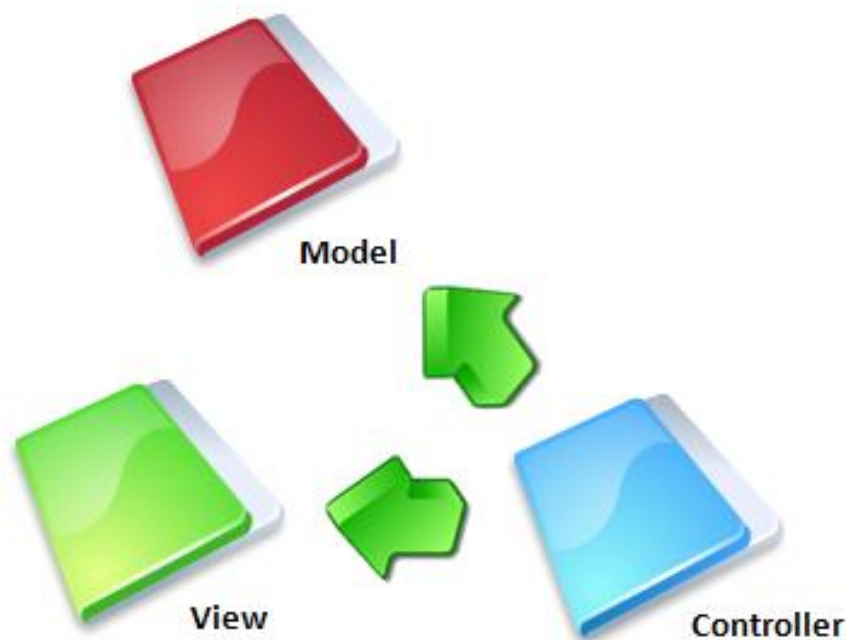
The ObLib Architecture has a particular structure defined for web-based application programming which lends itself to best-practice programming. The most important best-practice ideologies are described below, along with the methodologies of how ObLib implements them.

The Responsibilities of the ObLib Architecture

The PHP aspect of the ObLib Architecture has one single responsibility – to make a standard PHP object, interrelated to other standard PHP objects, output as an XML document. From this XML document that is outputted by the ObLib Architecture, PHP is then able to process the related XSLT document and output meaningful information to the user.

Model View Controller [MVC] Programming

Model View Controller [MVC] programming is a software development methodology designed to help complex data structuring and separation of interface logic (view) from data logic (model). MVC architectures separate interface logic and data logic and introduce a business logic layer (controller).

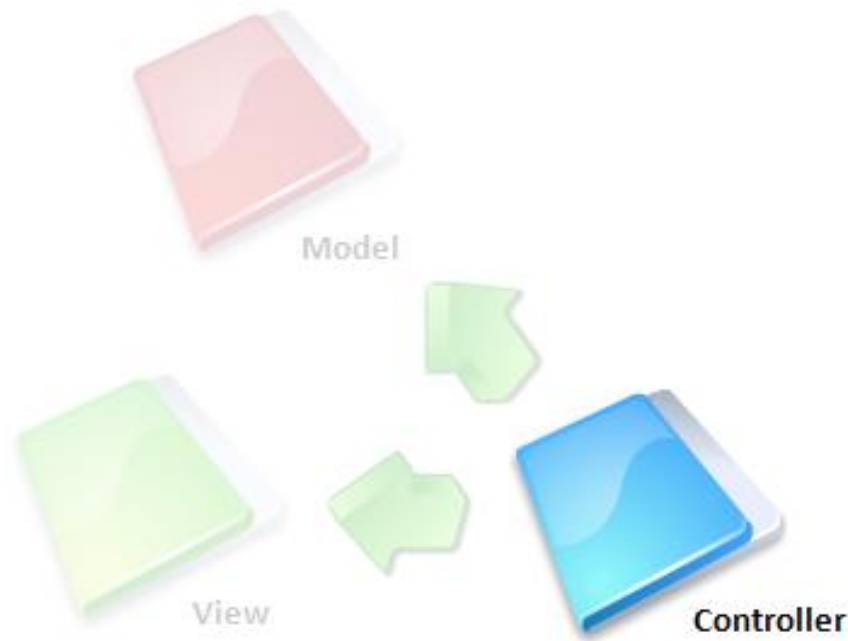


MVC Components

As mentioned before, there are three parts to MVC programming: Models, Views and Controllers.

Controller

A controller is the core of a page in your application. All the processing of user requests is done in the controller aspect of MVC programming.



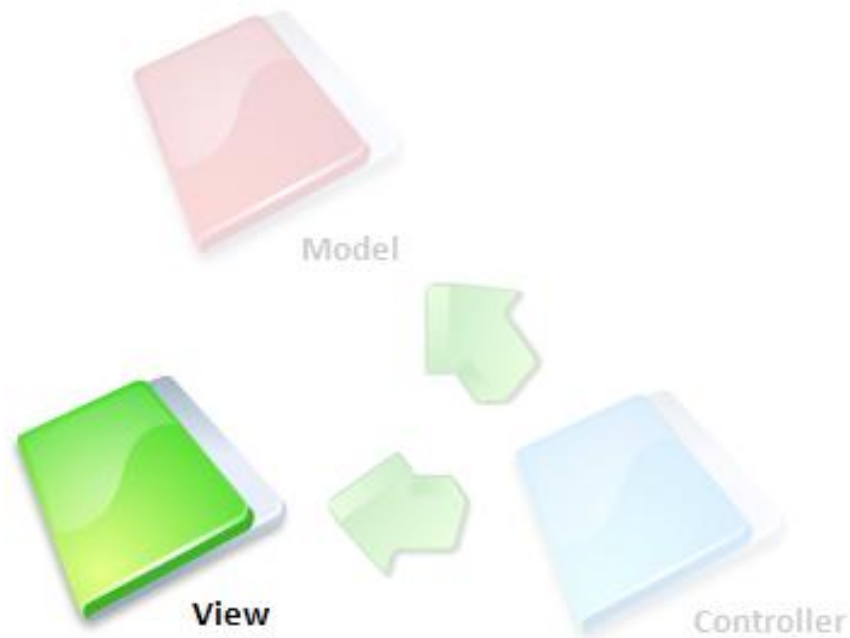
In the ObLib Architecture, controllers typically come in two parts: the dataController class and the dataSection class.

The dataController class represents a single page within a project. A dataController class will always be outputted with the XML tag name of "Controller" and will always appear as a child element of the XML "Section" element. The dataController class will choose which dataSection element applies to it.

A dataSection class will always have the XML tag name of "Section" and will always be the root element of an XML response. The purpose of a dataSection class is to provide templating functionality for PHP, so that multiple common tasks can be repeated without the need for reproducing code.

View

A view is a method in which an interface exists between a user and a system. Views can be in any particular shape or form, but mostly when developing web pages with ObLib, your XSLT views will output HTML to the user's web browser.

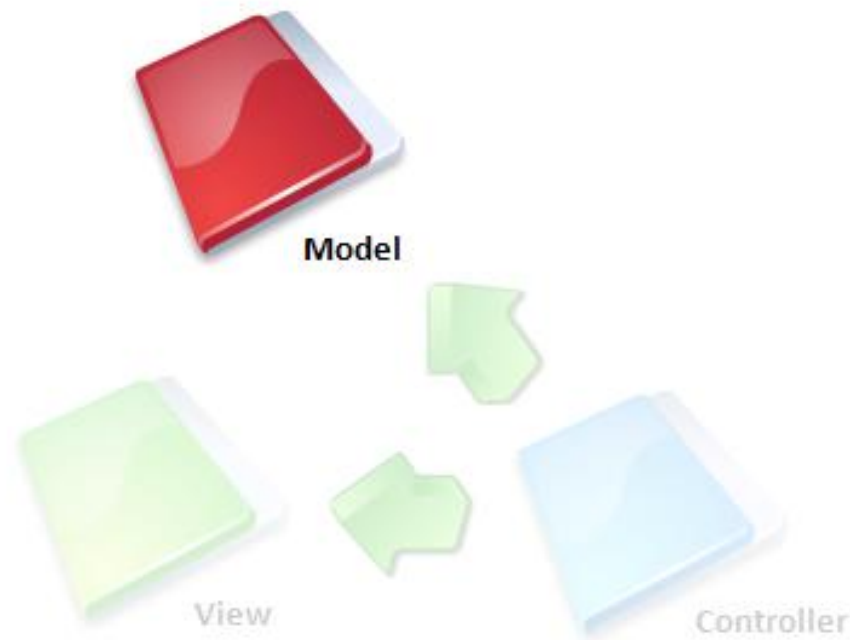


Examples of views that are used commonly in day-to-day computer use include (but are not limited to):

Web Pages	Text Files
ATOM and RSS Feeds	Multipart MIME Emails
Microsoft Office Documents	Images (JPEG, GIF, PNG)
Application Interfaces	COM Port I/O

Model

A model contains data (or information) which is, in some way, useful for the user and can be represented through a view.

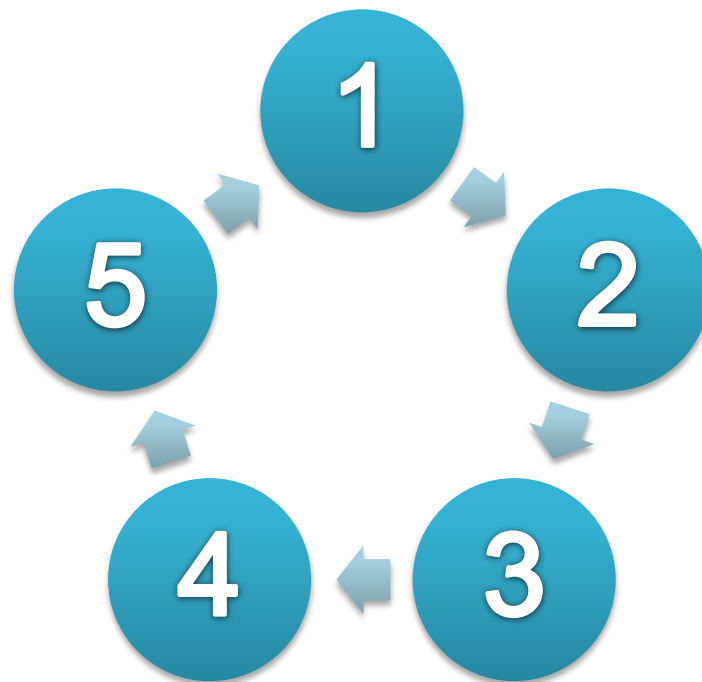


Models represent and control a means of retrieving information, usually from an external source. Instances where Models are used include the possibility of retrieving information from an RSS feed, and querying information from a database (made possible in the ObLib Architecture by using the `dataEntities` and `dataEntity` classes through OXRM).

The relationship between a View and a Model is limited (intentionally) by the Controller. A view can only see properties that have been set by the model itself and cannot query or make method calls (for example) to the Model in order for it to further process information.

MVC Lifecycle

When a page is requested in ObLib, a typical lifecycle will occur in this five step process:



1. An event will occur in the user interface (view) which will call an ObLib page. For example:
 - a. A button will be clicked
 - b. The user will type a URL
 - c. A link will be selected
2. The defined handler (controller) will process the input and gather any foundry information that it will need to process the request.
3. The controller will access the required data (model) and perform an action on it in some way, shape or form. For example: it may authenticate a session, or update a user's profile information.
4. A view is generated and uses information gained from the Model (which it obtains through the Controller) in order to output meaningful information to the user.
5. The user interface is outputted to the user in the required form and waits for a response to process any further actions (which will repeat the cycle).

The ObLibGation Process

ObLibGation (pronounced: obligation) is a supplementary software development life cycle (SSDLC) which can be used in conjunction with any other software development life cycle (SDLC). The most highly recommended SDLC to use in conjunction with the ObLibGation process is the Real World Software Process (RWSP), created by the Queensland University of Technology.

The following definition of RWSP comes from the RWSP website:

The Real World Software Process (RWSP) has four distinct parts: Phase Zero, Phase One, Phase N and Finalisation. Phase Zero describes what takes place to get to the stage where a software project is started; Phase One describes what happens in the first increment of the software development process; Phase N is a repetitive stage where the system is incrementally extended until all the functionality of the system has been delivered; and the Finalisation stage describes what happens after delivery of the completed product. (RWSP, 2002-2007)

For more information about the RWSP, please visit the official RWSP website:

<http://sky.fit.qut.edu.au/~rwsp/>.

Although RWSP recommends the use of a CVS for version control of documents, ObLib recommends the use of Subversion. More information about Subversion is available from: <http://subversion.tigris.org/>.

ObLib Entity Mapping Process

The ObLib Entity Mapping Process is the responsibility of the Backend Programmer.

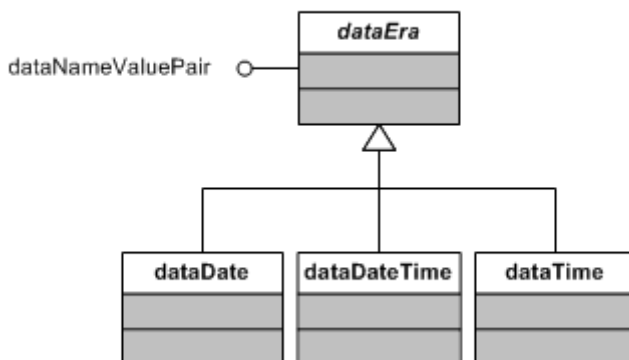
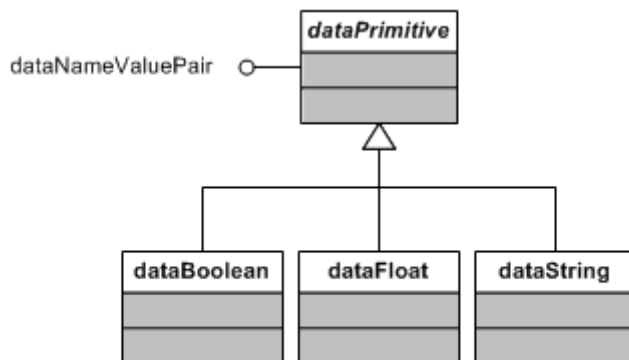
The ObLib Entity Mapping Process (OEMP) defines the entities that are required in the system. Firstly it is recommended that a UML class diagram is created to ensure all the entities are present and interrelate appropriately. From this diagram, an ObLib XML Relationship Map (OXRM) is created which defines the relationships between different entities. The OEMP process is usually completed in Phase One of RWSP.

Important Considerations for OEMP

When developing your UML class diagram(s) and your OXRM document, it is important to consider the following rules and limitations in your design:

Data Types for Variables

It is important to consider the limitations of all the programs which you are using when developing under any environment. During the OEMP process, it is important to consider that databases can only store primitive data types (for example; integers, chars, varchars, blobs, etc). Therefore, for your class variables, only classes which implement the `dataNameValuePair`. The purpose of the `dataNameValuePair` interface is to require a set of methods in a class which force the object to be identified by a name (the tag name of the object) and a value which can be represented as a primitive data type (however, the primitive value can be interpreted into a complex data type, as seen in `dataEra` objects). Common classes which implement the `dataNameValuePair` interface include (but are not limited to):



Class and Entity Names (Singular over Plural)

When choosing class names for entities during the OEMP, use the singular form of the noun rather than the plural form. This is because the plural form of the class name is used to database querying. For example:

Singular	Plural
Car	Cars
Country	Countries
Publication	Publications
User	Users

The singular form of a class name represents an object or row in a database, whereas the plural form of the class name represents the class which methodically retrieves the object/row/singular class from the database. Table names in a MySQL database should also be in the “singular” form.

Programming the ObLib Architecture

This section describes a practical example of how the ObLib Architecture can be used to create professional websites.

Random Number Generator

Learning Outcomes

By the end of this particular tutorial, you should be familiar with the basics of the ObLib Architecture, how ObLib-type objects interact with each other and how a View queries information from a model to display meaningful information to the user through their web browser.

Project Synopsis

The purpose of this project is to display a random number to a user.

Processes

To create this particular project, simply follow these processes:

1. Create an Interrupt Handler
2. Reference the Controller
3. Create the Controller
4. Create the View
5. Viewing the Result

Create an Interrupt Handler

Interrupt Handlers are located in the Base Directory of your ObLib project. These files are called by the web server itself to be interpreted. To create a new Interrupt Handler, copy an existing interrupt handler in the base directory of your project and assign it the name for your new project.

In this project, we will create an interrupt handler named “RandomNumber.php” to represent our random number generator.

Open your new interrupt handler (RandomNumber.php) and edit it to reflect the following template:

```
<?php
    require ('php/oblib.php');

    $Style->Controller (new Controller_RandomNumber);
    $Style->Output ('view/RandomNumber.xsl');
?>
```

Your interrupt handler contains two main parts:

1. The name of the Controller class which contains the information you wish to process.
2. The XSL view file which contains the formatting rules for the information that you wish to output.

The controller for this particular project is named “Controller_RandomNumber”. The responsibility for this controller is simply to generate a random number for the view to display.

The view for this file is located in xsl/view/RandomNumber.xsl. Nota Bene: the “xsl/” path is not included because ObLib will always assume that all view files are located in the XSL folder.

Reference the Controller

Next, we have to define the PHP location for the controller. This can be done either one of two ways:

1. [Strongly Recommended] In the interrupt handler (above, RandomNumber.php), under the “require” statement, add another require statement which points to the location of the controller. This would produce the following Interrupt Handler:

```
<?php
require ('php/oblib.php');
require ('php/controller/RandomNumber.controller.php');

$Style->Controller (new Controller_RandomNumber);
$Style->Output ('view/RandomNumber.xsl');
?>
```

2. [Strongly Advised Against] In the Global Class Definition, in the file “php/classes.php”, by adding a new entry in the array which points to the Random Number Controller. Nota Bene: the global definitions for classes assume that all PHP information is in the “php” folder, therefore there is no need to declare it.

```
// Controller Classes
'Controller_RandomNumber' => 'controller/RandomNumber.controller.php',
```

Create the Controller

In order to create the controller, browse to the “php/controller/” directory and create a new file named “RandomNumber.controller.php”. Our RandomNumber controller will reflect the following code (copy and paste this code in the controller file):

```
<?php
class Controller_RandomNumber extends dataController
{
    function ___construct ()
    {
        // Attach a Random Number to the Controller
        $this->Push (new dataFloat ('RandomNumber', rand (1, 10)));
    }
}
?>
```

This file reflects a standard PHP class. There is one major important difference between this the structure of this class as opposed to a standard PHP class, the function “___construct” has three (3) underscores in front of it, rather than the standard two (2) underscores which you would normally assign to a constructor method. This is due to the following coding standards:

1. Methods or Properties which only contain one (1) underscore affixed at the beginning of the declaration are defined as “private”. The single underscore indicates to the user that this method or property is private (although PHP will not enforce a private member visibility unless it is explicitly marked as private). For more information on member visibility, see the following PHP page:
<http://www.php.net/manual/en/language.oop5.visibility.php>
2. Methods or Properties which are marked with two (2) underscores affixed to the beginning of the declaration are marked as “PHP Reserved” or “PHP Magic Functions”. PHP defines a Magic Function as the following:

PHP reserves all function names starting with __ as magical. It is recommended that you do not use function names with __ in PHP unless you want some documented magic functionality.

<http://www.php.net/manual/en/language.oop5.magic.php>

From the Controller above, we can see that the constructor of the controller will perform one particular processing instruction:

```
$this->Push (new dataFloat ('RandomNumber', rand (1, 10)));
```

In plain terms, this processing instruction does the follow:

Attach to this particular Controller

A floating-point number (a Model)

Named "RandomNumber"

With a value between 1 and 10

For more information about generating random numbers using the rand() function, visit the Rand function documentation at the PHP website: <http://www.php.net/rand>.

Create the View

Finally, before we output the file to the browser, we need to create a view (that we defined in our interrupt handler) to display the Random Number to the user. Browse to the folder "xsl/view/" and create a new file called RandomNumber.xsl. This file will contain the following XSL code:

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml">

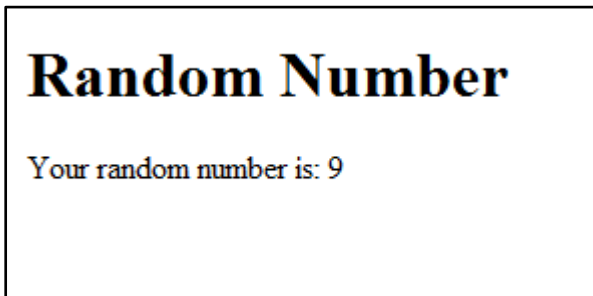
  <xsl:template match="/">
    <html>
      <head>
        <title>Random Number Generator</title>
      </head>
      <body>
        <h1>Random Number</h1>
        <p>
          Your random number is:
          <xsl:value-of select="/Section/Controller/RandomNumber" />
        </p>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Viewing the Result

Open your web browser and point it to the address for your RandomNumber.php.

For example: <http://localhost/RandomNumber.php>

You will see your webpage return a page similar to the following:



Conclusion

Congratulations, you've created your first ObLib project! Although this particular project is very basic, you should now have a firm understanding of how Models, Views and Controllers interrelate.

Retrieving information from MySQL

Learning Outcomes

This section describes how to select, insert, update and delete information from a MySQL database. In order to follow this tutorial, you must download and upload the World Database, as made available by MySQL. For information on how to load the World database into MySQL, and to download the MySQL World database, visit the following website: <http://dev.mysql.com/doc/world-setup/en/world-setup.html>.

Project Synopsis

The purpose of this project is to display information about Countries which exist around the world in a web-based user interface.

Processes

To create this particular project, simply follow these processes:

1. Setup your Database Parameters
 - a. Setup Service Identifications
 - b. Identify the Current Server
 - c. Define the Root Directory
 - d. Define the MySQL Connection Parameters
2. OEMP: ObLib Entity Mapping Process
3. Creating an OXRM Document
4. Create your dataEntity and dataEntities classes
5. Reference the Entity Classes
6. Create an Interrupt Handler
7. Create the Controller
8. Creating the Search Method
9. Outputting the Results

Setup your Database Parameters

Before you being any database-type application, you will need to setup a database connection and define your database connection parameters. ObLib defines database information on a per-server basis, making it easy to define and extend on to multiple servers.

Setup Server Identifications

Firstly, you must setup the server identification parameters by making a list of all the servers that will be using this project. You can then define this list in the “php/server.php” file. By default, two servers are defined: one named “local” and one named “live.” You can add as many servers as you like.

```
define ('SERVER_LOCAL'      , 0);
define ('SERVER_LIVE'      , 1);
```

Identify the Current Server

After you have created a list of all the servers that will host your project, define the way in which you can identify the server you are currently using. It is best practice to only use the “localhost” server name when dealing with one or two servers. When using more than two servers, try using the name of the computer instead of the “localhost” hostname.

```
// Which server is this?
switch (strtolower ($_SERVER ['SERVER_NAME']))
{
    // http://localhost/
    case 'localhost':
        define ('SERVER', SERVER_LOCAL);
        break;

    // http://phpoblib/
    case 'phpoblib':
        define ('SERVER', SERVER_LOCAL);
        break;

    // http://www.phpoblib.com/
    case 'www.phpoblib.com':
        define ('SERVER', SERVER_LIVE);
        break;

    default:
        throw new Exception ('Server Identification Failed');
}
```

Define the Root Directory

By default, the root directory is not defined on a per-server basis. ObLib will assume that you have placed the project in the root directory of your web server. If you need to put your ObLib project elsewhere, it is recommended that you do this on a per-server basis by adding the “ROOT” definition in the MySQL connection parameters (below). If you are using a per-server basis, ensure you copy the following line to your MySQL definitions:

```
define ('ROOT', $_SERVER ['DOCUMENT_ROOT']);
```

Define the MySQL Connection Parameters

The final parameters that you will need to define on the “php/server.php” file are the MySQL Connection Parameters. These are also done on a per-server basis to make it easy for server-porting. Ensure you change these variables to match your server variables.

```
switch (SERVER)
{
    case SERVER_LOCAL:
        define ('MYSQLI_HOSTNAME'      , 'localhost');
        define ('MYSQLI_USERNAME'      , 'root');
        define ('MYSQLI_PASSWORD'      , '');
        define ('MYSQLI_DATABASE'      , 'phpoblib_com');
        break;

    case SERVER_LIVE:
        define ('MYSQLI_HOSTNAME'      , 'localhost');
        define ('MYSQLI_USERNAME'      , '');
        define ('MYSQLI_PASSWORD'      , '');
        define ('MYSQLI_DATABASE'      , 'phpoblib_com');
        break;
}
```

OEMP: ObLib Entity Mapping Process

Following the ObLib Entity Mapping Process, the first step is to create a UML document which describes the fields and relationships for each particular entity in the project. However, for this example, because of the employed use of the MySQL world database, this part of the tutorial will forfeit the creation of a UML document.

The next step in OEMP is to create a OXRM document from the UML document (or in this case, database times). The ObLib OXRM document will represent the structure of the tables and the relationships between the entities (tables) in the database.

City

Field	MySQL Data Type	ObLib Data Type
ID	int(11)	dataFloat
Name	char(35)	dataString
CountryCode	char(3)	dataString
District	char(20)	dataString
Population	int(11)	dataFloat

Country

Field	MySQL Data Type	ObLib Data Type
Code	char(3)	dataString
Name	char(52)	dataString
Continent	enum('Asia', 'Europe', 'North America', 'Africa', 'Oceania', 'Antarctica', 'South America')	dataString
Region	char(26)	dataString
SurfaceArea	float(10,2)	dataFloat
IndepYear	smallint(6)	dataFloat
Population	int(11)	dataFloat
LifeExpectancy	float(3,1)	dataFloat
GNP	float(10,2)	dataFloat
GNPOld	float(10,2)	dataFloat
LocalName	char(45)	dataString
GovernmentForm	char(45)	dataString
HeadOfState	char(60)	dataString
Capital	int(11)	dataFloat
Code2	char(2)	dataString

CountryLanguage

Field	MySQL Data Type	ObLib Data Type
CountryCode	char(3)	dataString
Language	char(30)	dataString
IsOfficial	enum('T', 'F')	dataString ¹
Percentage	float(4,1)	dataFloat

Creating the OXRM Document

In order to write the OXRM document, open the file “xml/oxrm.xml” in a text or XML editor.

The root element in any OXRM document is the xml element “OXRM”. It contains one attribute which specifies the version of OXRM that is being used. For this tutorial, OXRM 1.0 is being used.

```
<?xml version="1.0" encoding="utf-8"?>
<oxrm version="1.0">
    ...
</oxrm>
```

¹ The MySQL World database specifies the field “IsOfficial” as an ENUM (Enumerative) data type with the values T and F for True and False respectively. In this type of situation, the ObLib Architecture recommends the use of the BOOL (Boolean) data type (which is later interpreted as a TINYINT(1) type). However, for this example, the enumerative type will be used.

Firstly, define the entities which exist in the project. In this case, there are three entities (which also correspond to the tables that exist in MySQL).

```
<?xml version="1.0" encoding="utf-8"?>
<oxrm version="1.0">
  <entity>
    <name>City</name>
    <table>City</table>
    <primary>Id</primary>
    <classes>
      <one>City</one>
      <many>Cities</many>
    </classes>
    <fields>
      ...
    </fields>
    <relationships>
      ...
    </relationships>
  </entity>
  <entity>
    <name>Country</name>
    <table>Country</table>
    <primary>Code</primary>
    <classes>
      <one>Country</one>
      <many>Countries</many>
    </classes>
    <fields>
      ...
    </fields>
    <relationships>
      ...
    </relationships>
  </entity>
  <entity>
    <name>CountryLanguage</name>
    <table>CountryLanguage</table>
    <primary />
    <classes>
      <one>CountryLanguage</one>
      <many>CountryLanguages</many>
    </classes>
    <fields>
      ...
    </fields>
    <relationships>
      ...
    </relationships>
  </entity>
</oxrm>
```

The following table provides a brief explanation of children to the Entity element:

Field Name	Description
Name	The name of the Entity. Usually the Name and the Table element (next) will be the same; however there are some instances when you may need them to be different.
Table	The MySQL table where information for this particular entity will be stored.
Primary	The field (in the list of fields, described later) which is the Primary Key for this Entity. There can only be one field for a primary key and should be (but is not required to be) defined as auto_increment in the database table.
Classes	Contains a listing of PHP classes which correspond to this Entity.
Classes / One	The PHP class name for the Singular for this Entity
Classes / Many	The PHP class name for the Plural of this Entity

For this tutorial, Fields are not described as their application in OXRM is self explanatory. The information for what ObLib data types are assigned to each field are specified in the tables above. It is important to note that you can only use ObLib objects which implement the ObLib dataNameValuePair interface as ObLib data types. For more information about the role of the dataNameValuePair interface in ObLib, see Data Types for Variables on page 14.

The final step in creating an OXRM document is to define the relationship between each entity. ObLib defines relationships between fields on a per-entity-per-field basis. In the World database, the relationships between each entity can be described in the following tables:

Entity	City	Country	Country
Name	Country	Cities	Capital
Candidate Field	CountryCode	Code	Capital
Foreign Entity	Country	City	City
Foreign Field	Code	CountryCode	Id
Frequency	One	Many	One

Entity	Country	CountryLanguage
Name	CountryLanguages	Country
Candidate Field	Code	CountryCode
Foreign Entity	CountryLanguage	Country
Foreign Field	CountryCode	Code
Frequency	Many	One

The final OXRM document would be represented as the following:

```
<?xml version="1.0" encoding="utf-8"?>
<oxrm version="1.0">
  <entity>
    <name>City</name>
    <table>City</table>
    <primary>Id</primary>
    <classes>
      <one>City</one>
      <many>Cities</many>
    </classes>
    <fields>
      <field>
        <name>ID</name>
        <type>dataFloat</type>
      </field>
      <field>
        <name>Name</name>
        <type>dataString</type>
      </field>
      <field>
        <name>CountryCode</name>
        <type>dataString</type>
      </field>
      <field>
        <name>District</name>
        <type>dataString</type>
      </field>
      <field>
        <name>Population</name>
        <type>dataFloat</type>
      </field>
    </fields>
    <relationships>
      <relationship>
        <name>Country</name>
        <candidate-field>CountryCode</candidate-field>
        <foreign-entity>Country</foreign-entity>
        <foreign-field>Code</foreign-field>
        <frequency>One</frequency>
      </relationship>
    </relationships>
  </entity>
  <entity>
    <name>Country</name>
    <table>Country</table>
    <primary>Code</primary>
    <classes>
      <one>Country</one>
      <many>Countries</many>
    </classes>
    <fields>
      <field>
        <name>Code</name>
        <type>dataString</type>
      </field>
      <field>
        <name>Name</name>
        <type>dataString</type>
      </field>
      <field>
        <name>Continent</name>
        <type>dataString</type>
      </field>
    </fields>
  </entity>
</oxrm>
```

```

</field>
<field>
  <name>Region</name>
  <type>dataString</type>
</field>
<field>
  <name>SurfaceArea</name>
  <type>dataFloat</type>
</field>
<field>
  <name>IndepYear</name>
  <type>dataFloat</type>
</field>
<field>
  <name>Population</name>
  <type>dataFloat</type>
</field>
<field>
  <name>LifeExpectancy</name>
  <type>dataFloat</type>
</field>
<field>
  <name>GNP</name>
  <type>dataFloat</type>
</field>
<field>
  <name>GNPOld</name>
  <type>dataFloat</type>
</field>
<field>
  <name>LocalName</name>
  <type>dataString</type>
</field>
<field>
  <name>GovernmentForm</name>
  <type>dataString</type>
</field>
<field>
  <name>HeadOfState</name>
  <type>dataString</type>
</field>
<field>
  <name>Capital</name>
  <type>dataFloat</type>
</field>
<field>
  <name>Code2</name>
  <type>dataString</type>
</field>
</fields>
<relationships>
  <relationship>
    <name>Cities</name>
    <candidate-field>Code</candidate-field>
    <foreign-entity>City</foreign-entity>
    <foreign-field>CountryCode</foreign-field>
    <frequency>Many</frequency>
  </relationship>
  <relationship>
    <name>Capital</name>
    <candidate-field>Capital</candidate-field>
    <foreign-entity>City</foreign-entity>
    <foreign-field>ID</foreign-field>
    <frequency>One</frequency>
  </relationship>

```

```

        <relationship>
            <name>CountryLanguages</name>
            <candidate-field>Code</candidate-field>
            <foreign-entity>CountryLanguage</foreign-entity>
            <foreign-field>CountryCode</foreign-field>
            <frequency>Many</frequency>
        </relationship>
    </relationships>
</entity>
<entity>
    <name>CountryLanguage</name>
    <table>CountryLanguage</table>
    <primary />
    <classes>
        <one>CountryLanguage</one>
        <many>CountryLanguages</many>
    </classes>
    <fields>
        <field>
            <name>CountryCode</name>
            <type>dataString</type>
        </field>
        <field>
            <name>Language</name>
            <type>dataString</type>
        </field>
        <field>
            <name>IsOfficial</name>
            <type>dataString</type>
        </field>
        <field>
            <name>Percentage</name>
            <type>dataFloat</type>
        </field>
    </fields>
    <relationships>
        <relationship>
            <name>Country</name>
            <candidate-field>CountryCode</candidate-field>
            <foreign-entity>Country</foreign-entity>
            <foreign-field>Code</foreign-field>
            <frequency>One</frequency>
        </relationship>
    </relationships>
</entity>
</oxrm>

```

Create your dataEntity and dataEntities classes

If you have reached this section of the tutorial, you should have a database connection defined to your MySQL server and you've defined the entities in your OXRM document. The next stage is to define each of the dataEntity and dataEntities classes for your entities defined in your OXRM document.

Create the Entity Skeleton

In the folder "php/entity/", you will notice that there is a file named "_singular.entity.php". You may want to keep this file for the case of creating new entities; you can use this file as a template. To create entity classes, make a new file in the "php/entity/" directory with the name of the singular (one) form of your entity. In the case of the World database, you will need to create 3 files which include:

1. city.entity.php
2. country.entity.php
3. countrylanguage.entity.php

In each of these files, you should use the following template as a basis for your templates:

```
<?php

class _Plural extends dataEntities
{
    function __construct ()
    {
    }
}

class _Singular extends dataEntity
{
    function __construct ()
    {
    }
}

?>
```

Relating to the OXRM document created earlier, the classes that you will need to create are the following:

File	Entity	_Singular	_Plural
city.entity.php	City	Cities	City
country.entity.php	Country	Countries	Country
countrylanguage.entity.php	CountryLanguage	CountryLanguages	CountryLanguage

As an example, the following is an entity class definition for the “city” entity. This template would reside in the “city.entity.php” file:

```
<?php

class Cities extends dataEntities
{

    function __construct ()
    {
    }

}

class City extends dataEntity
{

    function __construct ()
    {
    }

}

?>
```

From here, you can create the proper classes for the country.entity.php file and the countrylanguage.entity.php file.

Reference the Entity Classes

Because these classes will be used frequently, we need to reference them in the “php/classes.php” file. When you view the classes.php file for the first time, you will notice that there is a section defined for entity classes which should look similar to the following:

```
// Entity Classes
'_Singular'      => 'entity/_Singular.entity.php',
'_Plural'        => 'entity/_Singular.entity.php',
```

All you need to do is add your entity classes underneath this section:

```
// Entity Classes
'Cities'         => 'entity/city.entity.php',
'City'          => 'entity/city.entity.php',

'Countries'     => 'entity/country.entity.php',
'Country'      => 'entity/country.entity.php',

'CountryLanguages' => 'entity/countrylanguage.entity.php',
'CountryLanguage' => 'entity/countrylanguage.entity.php',
```

Create an Interrupt Handler

In the root directory of your ObLib project, create an interrupt handler which will call a controller to show a list of all the countries which are in the word database.

The filename for the Interrupt Handler will be “countries.php” and will contain the following information:

```
<?php
    require ('php/oblib.php');
    require ('php/controller/countries.controller.php');

    $Style->Controller (new Controller_Countries);
    $Style->Output ('view/countries.xsl');
?>
```

For more information on Interrupt Handlers, see “Create an Interrupt Handler” on page 18.

Create the Controller

In the Interrupt Handler that was just created, we defined that we were using a Controller class named “Controller_Countries” which is located in the file “php/controller/countries.controller.php”. Browse to the “php/controller/” directory and create the file “countries.controller.php”.

The controller for retrieving a list of all the countries in the World database will be quite short. All we are planning to do in the Controller_Countries class is retrieve a list of Countries in the World database, without any particular limits in our search clauses:

```
<?php
class Controller_Countries extends dataController
{

    function ___construct ()
    {
        $denCountry = $GLOBALS ['OxRM']->Entity ('Country');
        $arrCountry = $denCountry->AllCountries ()->Execute ();
        $this->Push ($arrCountry);
    }
}
?>
```

At closer inspection, let it be reminded that the controller constructor class has three (3) instead of the standard two (2) underscores in front of the “construct” name.

The only parts of the controller which are unfamiliar at this point are the following processing instructions:

```
$denCountry = $GLOBALS ['OXRM']->Entity ('Country');
$arrCountry = $denCountry->AllCountries ()->Execute ();
$this->Push ($arrCountry);
```

In order to retrieve information from a database, it is important to define which entity you are retrieving from. The prefix of a variable which links to an entity is “den”. Such as in this example, were retrieving information from the OXRM document which relates to the country entity:

```
$denCountry = $GLOBALS ['OXRM']->Entity ('Country');
```

The entity (in this case, the country entity) has the method “AllCountries” called (although we have not defined the “AllCountries” method yet, it will be defined in the next step). This returns a `dataEntityQuerySearch` object, which all you need to do at the moment is called the “Execute” method to retrieve an array of Countries from the database. You can then attach this `ObLib` array to the Controller so that it can be interpreted by the view.

```
$arrCountry = $denCountry->AllCountries ()->Execute ();
$this->Push ($arrCountry);
```

Creating a Search Method

A search method resides in the `_plural` class and provides a method of querying an entity through MySQL with predefined restrictions to allow for simple and reusable querying.

In the case of the World database, we want to alter the “countries” entity class to reflect the “AllCountries” method described earlier. To do this, open the “`php/entity/country.entity.php`” file and edit the “Countries” class to reflect the follow (do not edit the “Country” class):

```
class Countries extends dataEntities
{
    function __construct ()
    {
    }

    public function AllCountries ()
    {
        $desCountry = $this->__search ();
        return $desCountry;
    }
}
```

In this alteration, we see that there are four (4) new lines which have been added:

```
public function AllCountries ()
{
    $desCountry = $this->__search ();
    return $desCountry;
}
```

The function “AllCountries” (as described earlier) retrieves all the information about Countries from the world database. This is done very basically through the function:

```
$desCountry = $this->__search ();
return $desCountry;
```

The “__search” method returns an object which is basically a means for ObLib to query a database and return all the rows which are in the “Country” table. More information on applying constraints (such as in the WHERE clause of a MySQL statement) are available in the next tutorial.

Outputting the Results

Finally to output the results to browser, an XSL file needs to be created which will display the countries in a table. The filename has been defined earlier in the Interrupt Handler as “view/countries.xml”, so browse to the “xsl/view” folder and create a new file called “countries.xml”. Copy and paste the following XSL code into the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml">
  <xsl:template match="/">
    <xsl:variable name="Countries" select="/Section/Controller/Countries" />
    <html>
    <head>
    <title>Country List</title>
    </head>
    <body>
      <table border="1" cellpadding="4" cellspacing="0">
        <thead>
          <tr>
            <th>Code</th>
            <th>Country Name</th>
            <th>Continent</th>
          </tr>
        </thead>
        <tbody>
          <xsl:for-each select="$Countries/Country">
            <xsl:variable name="Country" select="." />
            <tr>
              <td><xsl:value-of select="$Country/Code2" /></td>
              <td><xsl:value-of select="$Country/Name" /></td>
              <td><xsl:value-of select="$Country/Continent" /></td>
            </tr>
          </xsl:for-each>
        </tbody>
      </table>
    </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Viewing the Result

Open your Interrupt Handler in your web browser and check the output. An example of an address you would open could be: <http://localhost/Countries.php>

The results should reflect the following:

Code	Country Name	Continent
AF	Afghanistan	Asia
NL	Netherlands	Europe
AN	Netherlands Antilles	North America
AL	Albania	Europe
DZ	Algeria	Africa
AS	American Samoa	Oceania
AD	Andorra	Europe
AO	Angola	Africa
AI	Anguilla	North America
AG	Antigua and Barbuda	North America
AE	United Arab Emirates	Asia
AR	Argentina	South America
AM	Armenia	Asia
AW	Aruba	North America

Conclusion

Congratulations, you've created your first ObLib project which connects to a MySQL database! Although this particular project is very basic, you should now have a firm understanding of how to model OXRM documents through the OEMP process, how to command the ObLib Architecture to search for a list of all records in a database and how to display MySQL information in a table to a browser.

Frequently Asked Questions

Backend Programming

The following questions relate specifically to the production of PHP classes.

Forcing NULL values instead of empty values in Databases

Question: Why does ObLib force a value to be NULL rather than allowing a string to be blank? This constraint conflicts with MySQL's article about the Differences between NULL values and Empty Values (<http://dev.mysql.com/doc/refman/4.1/en/problems-with-null.html>), why does ObLib choose to force NULL values instead of allowing empty strings?

Answer: In the SQL-99 standard, there are five rules which must be adhered to when developing databases. They include the following:

1. All information represented in a database must be a true and accurate representation of information presented in its projected environment (whether that environment is real-world or virtual-world)
2. All rows in a database table must be unique; no row can exist twice in the same table.
3. Fields in a database cannot be empty or blank, they must be presented by a NULL value (Nota Bene: that the value 0 for an Integer is not a NULL equivalent value).
4. No matter the order of the fields in a table, the data will still represent the same information.

[Citation Required]

OXRM

The following questions relate specifically to the production of OXRM documents.

OXRM type Element in OXRM field Restrictions

Question: Why must the OXRM type element (when a child of the OXRM field element) implement `dataNameValuePair` and not extend `dataEntity` or `dataEntities`?

Answer: The ObLib `dataNameValuePair` interface is used specifically by PHP ObLib to transact information with databases. The "Name" part of the `dataNameValuePair` dialogue refers specifically to the representation of the field name in the database, and the value represents the row's field (or object) value in the database.

The datatype classes "`dataEntity`" and "`dataEntities`" represent other entities which exist in the system.

If `dataEntity` was to implement `dataNameValuePair`, they would not be able to retrieve the remaining necessary information which is represented in the database about the entity.

The `dataEntities` class cannot implement `dataNameValuePair` interface because it doesn't represent information, it only represents a means for retrieving information about objects of its singular type.

Bibliography

Cadle, J., & Yeates, D. (2001). *Project Management for Information Systems* (3rd Edition ed.). Malaysia: Pearson Education.

Jewels, T. (2007, August). Brisbane, Queensland, Australia.