A Comparative Analysis of the LAMP and Microsoft.NET Frameworks.

By: Christo Crampton, g01c1073

Supervised by: Madeleine Wright

2005

Abstract:

The LAMP (Linux, Apache, MySQL, PHP) and .NET web development stacks represent two dominant competing technologies in the web development problem domain. New features in recent major releases of PHP and MySQL show a drive within the LAMP platform towards larger scale enterprise development. Similarly, ASP.NET 2.0 is an impressive upgrade from ASP.NET 1. This project aims to conclusively analyze these new technologies, and compare their relative strengths and weaknesses

To develop a successful understanding of the technologies a web application was produced. This application, in conjunction with stress and load testing was used to critically analyse the relative technologies. Analysis was performed in particular with regards to performance and capability of the applications in terms of data access, security, support for Web Services and XML, and performance.

From the research it is evident that both technologies are well suited to development within the web domain, however, ASP.NET is better suited to large-scale enterprise development. Although LAMP is comparable to ASP.NET in terms of performance and final product, development and maintenance of complex applications using LAMP is more intricate and problematic than the same with ASP.NET. While LAMP is ideal for smaller projects, and is capable of creating high-quality large scale applications, ASP.NET is better suited to larger scale development.

Acknowledgments

Special thanks must go to my supervisor, Madeleine Wright for her consistent help, advice and direction throughout the year; and friendly support and constructive criticism. Thanks also to the Telkom COE for providing testing servers on which comparative tests could be performed. Thanks to Mike Horne for help and technical support with regards to setting up the Linux environment.

ACKNOWLEDGMENTS ..... **II**

TABLE OF FIGURES ..... **V**

TABLES AND CODE LISTINGS ..... **VI**

CHAPTER 1 – INTRODUCTION **1**

1.1. PROBLEM STATEMENT: **1**
1.2. BACKGROUND **1**
1.3. METHODOLOGY **2**
1.4. DOCUMENT STRUCTURE **3**

CHAPTER 2 – THE APPLICATION **5**

2.1. OVERVIEW **5**
2.2 CHANGES TO THE APPLICATION **6**
2.3. ARCHITECTURE OF THE APPLICATION **6**
2.4. DESIGN IMPLICATIONS WITH .NET **9**
2.5. DESIGN IMPLICATIONS WITH LAMP **11**
2.6. CONCLUSION **12**

CHAPTER 3 – DATABASE CONNECTIVITY **13**

3.1. INTRODUCTION **13**
3.2. THE CORE STRUCTURE OF THE DATABASE **13**
3.3. DATA ACCESS WITH PHP **14**
3.3.1 CREATING THE DATA ACCESS LAYER (DAL) 14
3.3.2. PERFORMANCE 18
3.3.3. SHORTCOMINGS 19
3.3.4. CONCLUSIONS REGARDING LAMP DATABASE CONNECTIVITY 21
3.4. DATA ACCESS WITH .NET **21**
3.4.1. CONCLUSION ON DATA ACCESS WITH .NET 26
3.5. CONCLUSION **27**

CHAPTER 4 – SECURITY **28**

4.1. INTRODUCTION **28**
4.2. SECURITY VULNERABILITIES ON THE INTERNET **28**
4.2.1. COMMON WEB-BASED SECURITY EXPLOITS 28
4.3. TECHNOLOGY NEUTRAL SECURITY POLICY **32**
4.3.1. MATCHING INPUT DATA TO A PATTERN 32
4.3.2. RUNNING ON LEAST PRIVILEGES 32
4.3.3. STORAGE OF PRIVATE DATA 33
4.4. IMPLEMENTING A SECURE SYSTEM WITH LAMP **33**
4.4.1. WRITING SECURE PHP 33
4.4.2. SECURITY WITH APACHE 35
4.4.3. SECURING MYSQL 37
4.5. IMPLEMENTING A SECURE SYSTEM WITH .NET **38**
4.5.1. SECURITY IN THE VISUAL ENVIRONMENT 38

4.6. CONCLUSION 42

CHAPTER 5 – WEB SERVICES AND XML 44

5.1. INTRODUCTION 44
5.2. THE USE OF WEB SERVICES 44
5.3. IMPLEMENTING WEB SERVICES WITH LAMP 46
5.3.1. CONCLUSION ON LAMP WEB SERVICES 51
5.4. IMPLEMENTING WEB SERVICES WITH .NET 52
5.5. CONCLUSION 56

CHAPTER 6 – PERFORMANCE 57

6.1. INTRODUCTION 57
6.2. WHAT IS CACHING? 57
6.3. OPTIMIZATION WITH LAMP 58
6.3.1. USING CACHE_LITE FOR CHUNKED, FILE-BASED OUTPUT CACHING 58
6.3.2. CACHING WITH ASP.NET 63
6.4. CONCLUSIONS 67

CHAPTER 7 – CONCLUSION 70

7.1. OUTCOMES AND CONCLUSION 70
7.2. FUTURE WORK 71
7.2.1. EXTENSION OF THE CORE APPLICATION 71
7.2.2. FURTHER ANALYSIS 71

REFERENCES 72

APPENDIX I – SCREENSHOTS I

APPENDIX II – UML AND DESIGN I

Table of Figures

FIGURE 1 - 3 TIERED ARCHITECTURE 7
FIGURE 2 - AN EXAMPLE OF AS MASTERPAGE 10
FIGURE 3 - CORE DATABASE STRUCTURE 14
FIGURE 4 - DATA ACCESS OBJECT [ALUR. ET AL., 2003] 15
FIGURE 5 – CLASS DIAGRAM OF THE DAL DATA OBJECTS FOR PHP 17
FIGURE 6 - DATA ACCESS PERFORMANCE 19
FIGURE 9 – DATASET 24
FIGURE 10 - QUERY ANALYZER 25
FIGURE 12 - VULNERABLE TEXTBOX 41
FIGURE 13 - EXPLOITED VALIDATION CONTROL 42
FIGURE 14 - SCREENSHOT OF THE .NET GUI 45
FIGURE 15 - REST ARCHITECTURE [HINCHCLIFF. 2005] 48
FIGURE 16 - REST SERVICES VS. SOAP SERVICES (.NET), ORANGE IS REST, GREEN IS SOAP, THE RED LINE REPRESENTS USER LOAD 54
FIGURE 17 - .NET SOAP AND REST SERVICES AND LAMP REST SERVICE 55
FIGURE 18 – A CACHED PAGE 61
FIGURE 19 – CREATING THE CACHE 61
FIGURE 20 - A PAGE WITH NO OUTPUT CACHING 61
FIGURE 22 - LAMP HOME PAGE I
FIGURE 23 – LAMP FRANCHISE PAGE II
FIGURE 24 – LAMP GALLERIES PAGE II
FIGURE 25 - LAMP GALLERY PAGE III
FIGURE 26 – LAMP PICTURE PAGE III
FIGURE 27 - .NET HOME PAGE IV
FIGURE 28 - .NET FRANCHISE PAGE IV
FIGURE 29 - .NET GALLERIES PAGE V
FIGURE 30 - GALLERY PAGE V
FIGURE 31 - PICTURE PAGE VI
FIGURE 32 - LAMP DATA ACCESS LAYER I
FIGURE 33 - COMPLETE USE CASE DIAGRAM II
FIGURE 34 - SITE MAP III

Tables and Code Listings

LISTING 1 - STANDARD SQL QUERY CONSTRUCTION USING DB::PREPARE AND DB::EXECUTE 16
TABLE 2 - DATA CONNECTIVITY PERFORMANCE 19
LISTING 3 - REFLECTION OF A DATA OBJECT REVEALING CONNECTION INFORMATION 20
LISTING 4 - STORED PROCEDURES CREATED BY SQL QUERY ANALYSER 25
LISTING 5 - REGISTER GLOBALS EXPLOIT [PHP MANUAL. 2005] 34
TABLE 6 - SQL INJECTION 39
LISTING 7 - USING THE SOAPSERVER AND SOAPCLIENT CLASSES 46
LISTING 8 - CREATING A REST WEB SERVICE 50
LISTING 9 - CALLING A WEB SERVICE USING THE HTML IMAGE TAG 51
TABLE 10 - AVERAGE RESPONSE TIMES FOR WEB SERVICES 55
TABLE 11 - AVERAGE RESPONSE TIMES FOR A CACHED AND UN-CACHED PAGE 61
LISTING 12 - HOW THE ASP.NET TEMPLATING ENGINE CONVERTS AN ASP:LABEL CONTROL TO HTML 65
TABLE 13 - CACHING WITH .NET 65
LISTING 14 - CACHING WITH THE .NET CACHE OBJECT 67
LISTING 15 - APPLYING SQL SERVER 2005 NOTIFICATION-BASED CACHE INVALIDATION TO A SQLDATASOURCE 67

Chapter 1 – Introduction

1.1. Problem Statement:

The primary goal of this project is to provide valuable insight into the inner workings of the LAMP (Linux, Apache, MySQL, PHP) and .NET Frameworks so as to provide a strong comparative analysis of these technologies in terms of both their capabilities and performance for creating web applications.

1.2. Background

LAMP applications and classic ASP (Active Server Pages) have over the years stamped their authority on the internet as the leading and most commonly used server-side scripting languages [Netcraft, 2005]. Prior to the release of ASP.NET it was fairly unanimous that the LAMP platform had the edge over classic ASP.

However, with the release of ASP.NET - Microsoft's next generation web development platform - in 2000, the position of LAMP as the top web development platform suddenly seemed threatened. ASP.NET was not simply an upgrade of classic ASP, but an entire rewrite. ASP.NET offered several important features including a strongly object oriented framework, support for multiple languages (including an entirely new language - C#), and compiled ASP.NET pages boasting (according to Microsoft), twice the speed of classic ASP pages.

ASP.NET quickly established itself as the platform of choice, alongside Java, for large-scale enterprise-level web development. LAMP on the other hand has generally been sidelined to smaller applications and considered not yet ready for the enterprise.

The new releases of PHP 5 and MySQL 5 see both these platforms maturing, with increasing emphasis on enterprise related features. PHP 5, released early in 2005, offers much improved object oriented support as well as improved in-built support for XML and Web Services. MySQL 5, released in November 2005, also sees a maturing

of MySQL with new support for enterprise features such as stored procedures, functions and triggers.

On the .NET side, the .NET framework version 2, alongside Visual Studio 2005, was also released in 2005. ASP.NET 2 includes many impressive new features such as new controls, masterpages and themes which promote rapid and robust development.

In 2000, when ASP.NET was introduced, the question on many people's minds was: Is LAMP dead? In 2005 the new question is: Is LAMP ready for the enterprise? This paper looks at LAMP against a well established enterprise development framework, ASP.NET, and aims to establish a solid unbiased report as to the relative strengths and weaknesses of these development frameworks

1.3. Methodology

Firstly it must be emphasised that this project analyses the framework as a whole. The development frameworks, as mentioned earlier, are: LAMP, running Ubuntu "Breazy" Linux (Debian) server version, Apache 2, MySQL 4.0.24, and PHP 5; and .NET using Microsoft Windows Server 2003, Microsoft IIS (Internet Information Services) web server, Microsoft SQL Server 2005 database server and ASP.NET C# as a programming language.

This paper analyses the performance and capabilities of the entire framework. The LAMP and .NET configurations above represent typical configurations which are widely used in the enterprise. It is important that the framework as a whole is analyzed and not its separate components (such as just PHP or ASP.NET) because any typical web application requires a web server, database engine and coding language. Results from comparative analysis of the single components are easily doctored and are unreliable because performance can be affected by other components used in the testing framework. For example although both PHP and ASP.NET can connect to SQL Server, we would expect the configuration to favour ASP.NET because it is tailored for use with SQL Server. By analysing the defined system as a whole we obtain a far more holistic view of the comparative performance and capabilities of the technologies operating within their optimal frameworks.

Additionally, since the entire platform is analyzed, it is important to bear in mind that the strengths or weaknesses of a single component affect the overall analysis of the entire framework.

From the outset a number of features were decided upon as core features critical to most modern, enterprise–level, web applications. These features are: database connectivity, security, Web Services and XML, and performance. Research was undertaken to critically analyse the frameworks in terms of support for these features. Credit was given not only for high performance solutions, but also for frameworks which encouraged or promoted the development of robust, scalable and maintainable solutions.

To establish an effective analysis of the frameworks a web application was developed using both frameworks. The application served as the basis for feature comparison within the frameworks and provided practical experience upon which to base analysis of the capabilities of the frameworks.

In addition to practical experience and insight, test pages were created to test various features of the platforms. These pages were submitted to load tests as well as being tested for general response time.

Comparative tests were performed on identical separate computers. These computers boast high-end desktop computer specifications: 3Ghz Pentium 4 processor, 1 Gigabyte of RAM and 250 Gigabytes of storage space. Load tests were performed against both technologies using the Microsoft Visual Studio load test and web test tools, and Apache Bench benchmarking tool on Linux.

1.4. Document Structure

Chapter 2 offers insights into the architecture of the web application which was developed for testing. This chapter illustrates both the core features of the application as well as explaining how the application was used to gain expertise in the various sections and why it proved a useful test-bed for analysis. Thereafter, chapters 3 to 6 analyze in depth the issues with regards to data access, security, web services and

performance respectively. Finally, chapter 7 outlines the outcomes of the project and draws relevant conclusions.

Chapter 2 – The Application

In this chapter we explore the example application which was developed as a test-bed for evaluating the frameworks. The changes which were made to the applications and improvements are discussed, in particular with reference to the improved architecture of the new system. Finally a short discussion of the design environments for .NET and LAMP is discussed.

2.1. Overview

www.38.co.za is an existing, live photo gallery website which was developed by the author and was founded in February 2003. The site generates over a million hits a month and to date has over 17 000 photographs, over 200 galleries and a subscribed user-base of over 6000 users. www.38.co.za serves as the perfect test-bed for examining the capabilities of the frameworks because of both the performance and feature requirements of the site.

2.2 Changes to the Application

The original application was developed using LAMP. The existing application has become bloated over the two years over for which it has been running, due to ad-hoc development which has been implemented to supply new features on top of the original framework. As a result code has become riddled with unnecessary or repeated logic and spurious, complicated SQL statements. The goals for developing the new application were to use it as a test-bed for the project, re-creating the basic functionality of the old system but implementing a new robust n-tier architecture. It was not a goal to reproduce the complete functionality of the application, but rather, to produce a smaller, robust application with a solid architecture which would be easier to extend and maintain than the previous application.

The application developed in this project, although not as complete a solution as the original application provides a solid architecture for the application on which future development can be performed.

2.3. Architecture of the Application

The major design goals for the new application were to refactor the 2-tier LAMP code into a solid and robust 3-tier architecture, separating business logic to the BLL (Business Logic Layer) and data access code to the DAL (Data Access Later). The presentation layer, responsible for organizing data and delivering an interface to the end user, was designed to be a thin layer with all complicated logic pushed into the BLL, and all data access code pushed to the DAL. Figure 1 illustrates a typical n-tier architecture, and the architecture of the new application.

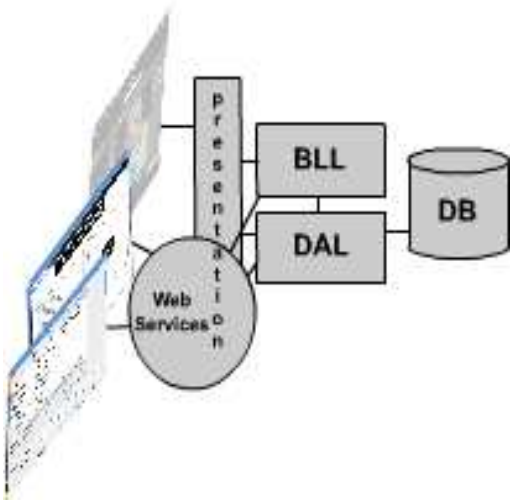

Figure 1 - 3 tiered architecture

Each layer in the application performs a specific task. This provides solid separation of logic, which eases the tasks of future maintenance and extension of the application. Another advantage of a tiered architecture is that it provides a centralized and specialized framework which promotes re-use of objects. The major advantage of such centralization is that changes to code need only be implemented once to reflect throughout the entire application. For example, input validation at the presentation layer is delegated to objects in the BLL. To upgrade validation throughout the entire presentation layer requires only that a single object in the BLL is updated. This improves control of the system because it is easier to keep track of functionality within the application. It also helps in creating and enforcing a policy towards issues such as security throughout the application.

Furthermore, a tiered architecture improves the scalability and flexibility of the application. Theoretically any layer can be swapped or replaced with little or no effect on the system. Although this may not often be the case in practice, it is one of the potential benefits of a layered architecture.

Specialized security can be implemented at every layer. For example, the tiered architecture promotes validation of input at the presentation layer, as well as integrity validation of data at the DAL against the database structure. The layered approach greatly decreases the chances of vulnerabilities in the application.

At the presentation layer there are two interfaces: a public Web Service API (which interacts with client applications through SOAP Web Services and XML, REST (Representational State Transfer) Web Services or RSS (Really Simple Syndication)); and the private HTML presentation layer used to serve web pages to the browser. By placing almost all logic in the BLL and DAL, the presentation layer is kept as thin as possible. Therefore, necessary changes to the structure or logic of the system need only be implemented in the backend layers of the system. The presentation layer is merely concerned with the display and format of information and workflow from the lower layers. The presentation layer is abstracted from the logic and structure issues dealt with by the lower layers.

Each tier represents unique challenges with regards to the project: issues regarding database connectivity, discussed in chapter 3 – “Database Connectivity”, are encapsulated in the DAL. Similarly Web Services and XML, discussed in chapter 5 – “Web Services”, are encapsulated at the presentation layer. Most security issues, discussed in chapter 4 – “Security”, are dealt with in the BLL, although security is implemented at every layer of the application. Finally, performance, such as caching, which is discussed in chapter 6 – “Performance”, is typically implemented at the presentation layer.

The application embodies a typical web application, and highlights challenges facing web developers. Therefore, it is a good test-bed for comparative analysis of the capabilities of the technologies.

2.4. Design Implications with .NET

In the ASP.NET version of the application, the database was a SQL Server database. A lot of the data access functionality was pushed into the database server by using stored procedures so that no actual SQL is written in code. The Data Access Layer (DAL) was built using .NET DataSet objects. DataSets abstract database tables to in-memory programmable objects. Creating the DAL with .NET is discussed in more detail in chapter 3 – "Database Connectivity", in the section "Data Access with .NET".

The Business Logic Layer (BLL) consists of a package of ASP.NET classes responsible for handling business logic and workflow. The presentation layer is a thin layer and is built using ASP.NET Web Forms and SOAP as well as REST Web Services. There are effectively three presentation interfaces, namely: an HTML interface typically used for browsing the site; and REST and SOAP Web Service interfaces used to empower remote syndication via remotely accessibly web methods and services.

At the presentation layer, new features in ASP.NET 2.0 such as master-pages, skins and themes, mentioned in more detail below, greatly improve the ability of the designer to create a uniform look and feel for a webpage. It is important that all possible logic at this level is abstracted to the BLL Layer so that changes to logic and structure can be centralized to changes in the BLL and therefore do not need to be implemented in all three presentation interfaces.

Presentation level design in Visual Studio .NET 2005 is aided by a number of new controls. Most notable are master-pages and themes. Master-pages allow the developer to easily define a uniform structure for a website. Pages inheriting from the master-page automatically inherit the structure and logic encased in the parent master-page. Master-pages effectively abstract the fairly complex issue of creating a uniform look and feel for an entire site, or section of a site. Furthermore, master-pages make it extremely easy to change the entire look and feel of a website from a central location.

Master-pages achieve centralized control of a system similar to, but more flexible and easier to use than CSS (Cascading Style Sheets). Figure 2 shows a screenshot of the core masterpage for the application. This image illustrates how master-pages help maintain a uniform look and feel over a webpage. The core frame of the site is included in all pages inheriting from the master page, and content unique to the specific page is stored in the `ContentPlaceHolder` (the box with the blue heading in the image below).

Figure 2 - An example of a Masterpage

Similar to master-pages, themes enable the developer to attach defined themes to controls. For example, the developer can define the look of a single control in a themed `.skin` file, and easily apply this to all controls of the same type throughout the application. Again, similar to CSS technology, but more powerful because, like master-pages, they are built into the ASP.NET templating engine, not merely used at an HTML level, and therefore can be accessed transparently through C# code.

Every step of the design process in .NET is accomplished through the use of rich visual aids. An impressive feature of Visual Studio, which has always been a strong point, and has been improved with version 2005, is the interaction between visual aids and implementation, as will be discussed in the following paragraphs.

UML (Unified Modelling Language) tools such as class diagrams, ORM database diagrams (see Figure 3 - Core Database Structure) and the visual dataset designer (Figure 9 – DataSet) are integrated into the environment and serve to merge the design and implementation processes, so easing the process of synchronisation between the two and shortening development time by enabling the developer to produce the core implementation of the project at design time.

A particularly good example of this is the class diagram tool (new to Visual Studio 2005) which enables the developer to design a class diagram in UML, while Visual Studio 2005 creates the respective class stubs in the background. Changes that are made at a later stage to the code are automatically synchronised with the class diagram ensuring that the diagram and code are accurate reflections of each other. The visual design environment is useful because it makes it easier to manage larger applications where design and code or application structure often become unsynchronised.

2.5. Design Implications with LAMP

The LAMP version of the Application was developed using the popular PHP/MySQL combination running on an Apache web server.

MySQL (prior to the recent release of version 5) is a light database which emphasized speed as a priority over issues such as advanced functionality. As a result stored procedures were not used, all data access was written in code and SQL queries were generated dynamically. According to Gulutzan [2005] stored procedures add extra security to applications as well as improving the performance of database queries. However, stored procedures are new to MySQL 5, and even the MySQL website issues a warning regarding their use in applications: “Stored procedures are something new for MySQL, so naturally you'll approach them with some caution. After all,

there's no track record, no large body of user experience that proves they're the way to go" [Gulutzan. 2005]. This debate as well as an explanation of how the DAL was created in LAMP using PHP data objects is taken up in chapter 3 – "Database Connectivity", in particular in the section entitled "Data Access with LAMP".

In the BLL functionality to encapsulate workflow and user logic is stored. In addition we created classes to handle forms similar to controls in ASP.NET. The classes in the BLL and DAL proved a useful testing ground for the new object orientation in PHP 5. Chapter 5 – Web Services, and the section, "Implementing Web Services with LAMP, offer some interesting insights into the complexities of creating Web Services with PHP, and reveal an alternative method of Web Services to the conventional SOAP Web Service.

2.6. Conclusion

A complex and complete 3-tier application was created using both technologies. The application improves on the architecture of an existing web application. Although the new application provides only the core functionality of the existing application, this core is built on top of a solid architecture featuring code-separation and logic centralization which help produce a robust application that is both easier to maintain and upgrade.

The application provides a complete problem domain for successful feature comparison of web development technologies in particular, with relevance to security, data access, performance and Web Services.

Chapter 3 – Database Connectivity

3.1. Introduction

Data connectivity is a vital component of almost all web applications. Whether the data is stored in a database, XML or even a text file, the vast majority of web applications use some form of data storage mechanism to store persistent data. A good application should be able to connect to a data store effectively and efficiently.

This chapter evaluates the technologies, LAMP and .NET, in terms of support for efficient and robust data access. With each technology, we examine the various data access methods which are available, measuring up the strengths and weaknesses of these methods.

It is important that data access is flexible. The main objective of a good data access layer is to decouple the database server (MySQL or SQL Server in this case) from the application code. The DAL provides a defined interface between the database server and application code. This chapter examines the effectiveness of tools and methods provided by the technologies to promote decoupling, and robust data access.

In our evaluation of the relative data access capacities of the technologies we emphasise the requirements that data access logic be contained to the DAL, and that data access is flexible, with emphasis on decoupling the database from the application code.

3.2. The Core Structure of the Database

The core structure of the database is illustrated in Figure 3. The structure for the database was modified from the original database. Extra tables were dropped and the

core structure below was achieved. The database is a relational database with data organized in a hierarchy which represents the data objects in the presentation layer.

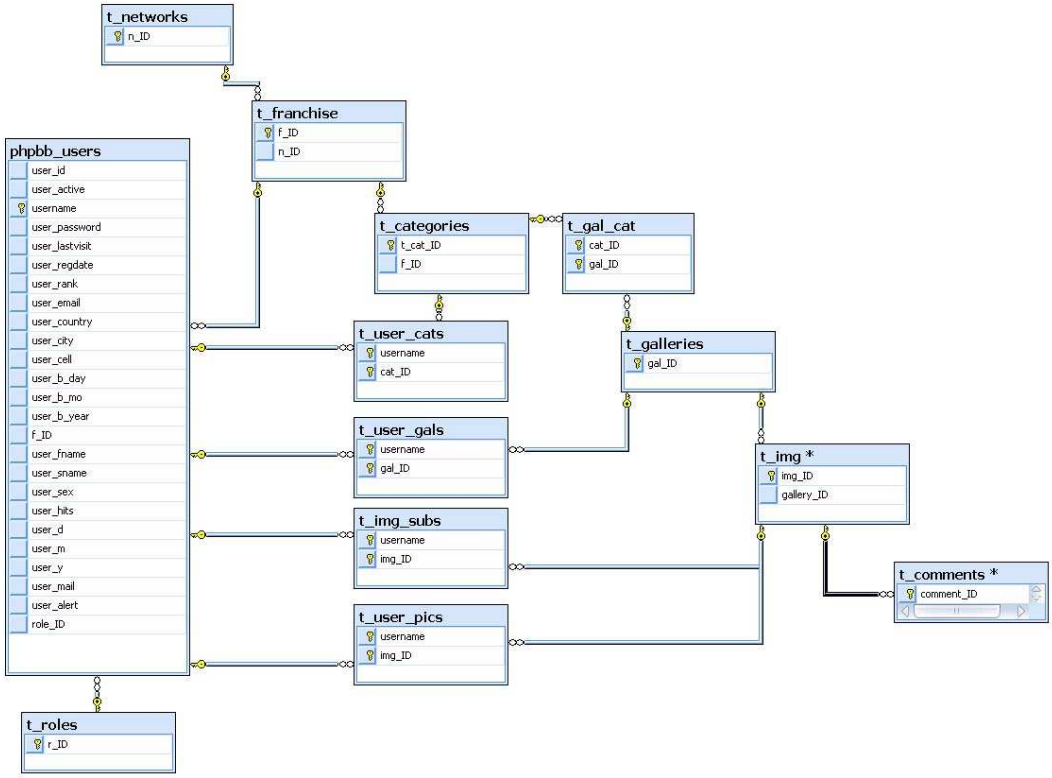

Figure 3 - Core Database Structure

3.3. Data Access with PHP

3.3.1 Creating the Data Access Layer (DAL)

The LAMP DAL was developed with three major components: a database abstraction layer (using `PEAR::DB`), a base-class `DBHelper` object, and data-objects (see Figure 4).

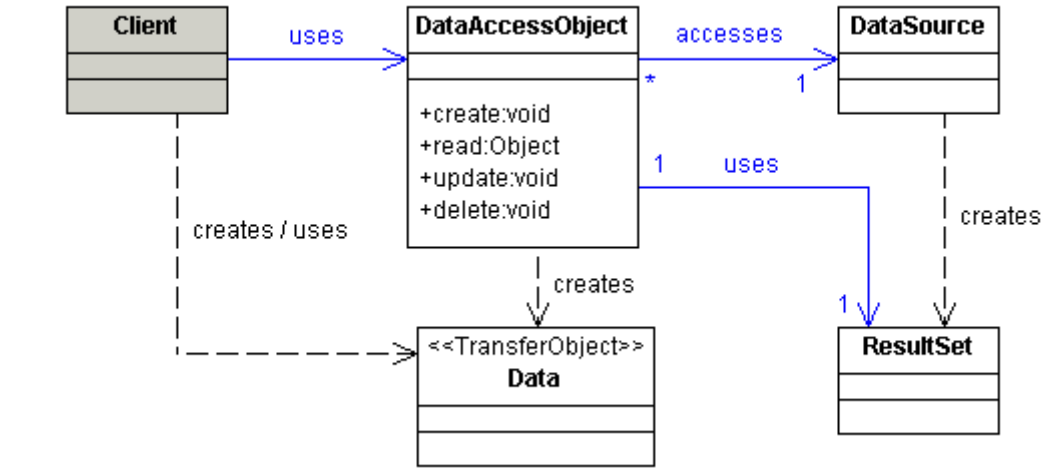

Figure 4 - Data Access Object [Alur. et al., 2003]

The database abstraction layer provides the logic necessary to write pluggable DALs which are decoupled from a particular database server. For example, `PEAR::DB` connects with a "`dsn`" (data source name) such as:

```
$dsn = "mysqli://$this->user:$this->pass@$this->host/$this->db_name";
```

This `dsn` specifies the database server to use as well as the connection details. In theory this suggests that simply by changing the `dsn` one should be able to connect and interact seamlessly with a completely different database server, provided that the table structure of the database is the same. For example, to switch from using a MySQL to a PostgreSQL database one need only change `dsn` as follows:

```
$dsn = "mysqli://$this->user:$this->pass@$this->host/$this->db_name";
to
$dsn = "pgsql://$this->user:$this->pass@$this->host/$this->db_name";
```

Obviously changing the database backend is a more complicated process that simply changing the one line of code, but a database abstraction layer such as `PEAR::DB` (or ADO .NET's table adapters) does go a long way in simplifying the process.

Another complexity with regards to data connectivity is that not all databases support standard SQL (in fact MySQL is notorious for its diversion from the SQL standard [Troels. 2005]). Therefore, any custom written SQL code may not be compatible with other database servers. It is the responsibiliy of the data abstraction layer to ensure that code written *is* compatible. PEAR::DB includes functions (`prepare()` and `execute()`) which contruct standard or compatible queries based on parameters passed to them. For example:

```
$sql = "select t_categories.*, count(t_gal_cat.gal_ID) as num_Gals
               from t_categories join t_gal_cat
               on t_gal_cat.cat_ID = t_categories.t_cat_ID
               where cat_ID = ?
               group by t_categories.t_cat_ID";
$sql = $this->db->prepare($sql);
$this->rs = $this->db->execute($sql,$id);
```

Listing 1 - standard SQL query construction using DB::prepare and DB::execute

Thus, using the `PEAR::DB` abstraction layer for both connections to a database as well as dynamic query construction provides enhanced flexibility at the DAL.

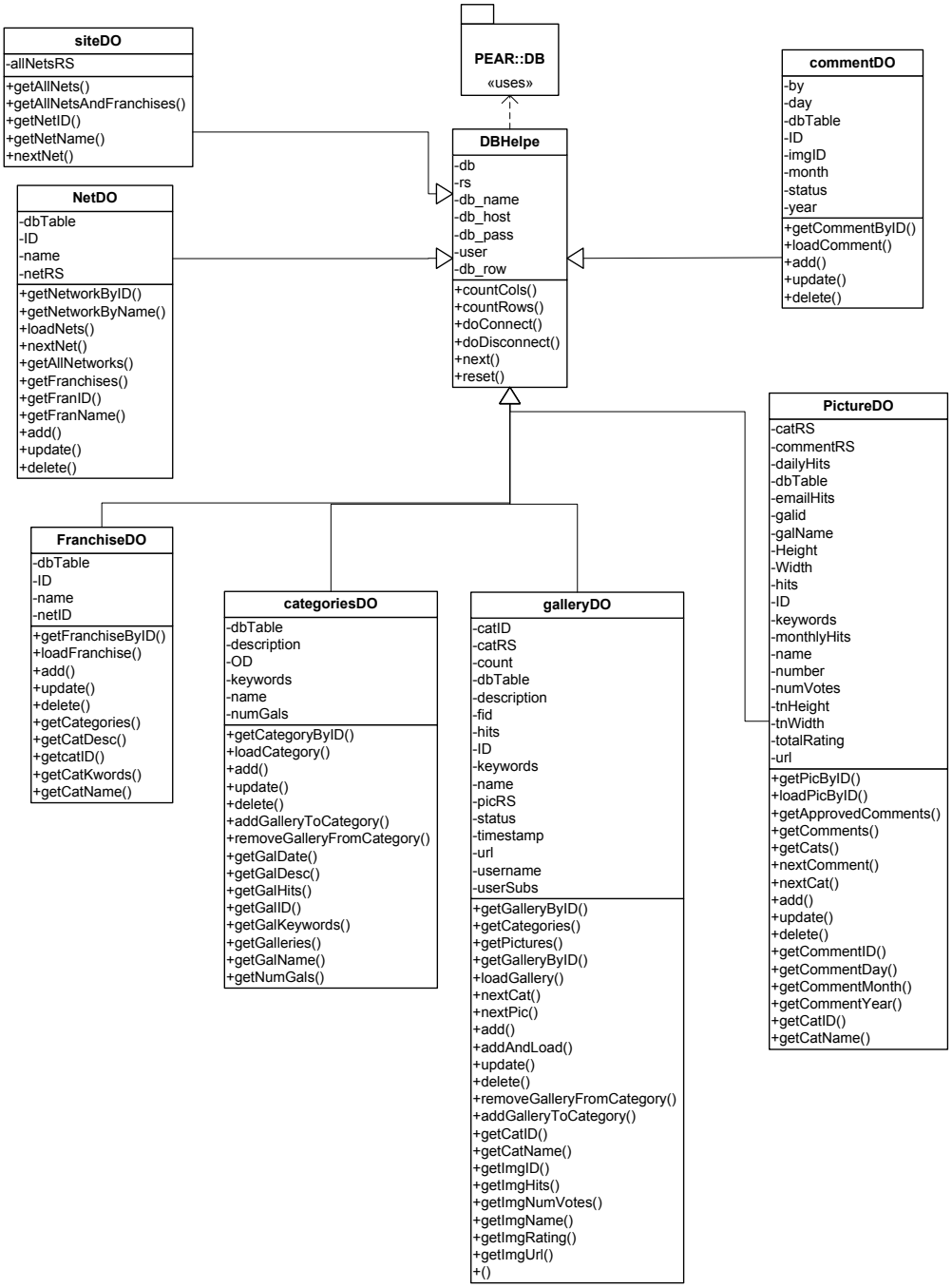

Figure 5 – Class Diagram of the DAL Data Objects for PHP

On top of the PEAR::DB abstraction object, PHP data objects are used to encapsulate the data structures and handle reading and writing to the persistent data store (in this case a MySQL database), see Figure 5 above. Data objects in the DAL provide a centralized interface to the database and improve the structure of the system by

decoupling the code from the database. The UML diagram illustrated in Figure 5 above shows how the data objects created map closely to the core database structure (Figure 3).

Because data is fed to the presentation layer through data objects, the presentation layer is protected from changes to the database structure because table and row names as well as SQL queries are issued only in the DAL. Changes to the database need to be reflected only once in the DAL, and are consequently replicated throughout the application, because the DAL provides a central access point to the database through which all data access is performed.

3.3.2. Performance

In terms of performance the lightweight PHP/MySQL combination is successful. Tests were performed on a page which connected to the database and queries the database for 100 rows. These were in turn rendered to the browser in a table as HTML.

Three pages were tested. One used a .NET `SQLconnection` to connect to a SQL Server database, one used a `ObjectConnection` to connect to a SQL Server database through a `DataSet`, and finally, the LAMP version used a `MySqli` connection to connect to a MySQL database. The .NET templating engine, in order to support controls and view state, adds a fair amout of HTML bloat to web-pages (see the chapter on caching for a more detailed explanation of this). This bloat was removed from the pages tested where possible so that HTML code rendered by the respective pages was almost identical, although some extra form tags needed to be added to the .NET pages so that the `DataList` control on the page was able to function properly and the page could compile. A stepped load test was performed using Visual Studio's load-test tool. The load was increased at intervals of 10 concurrent users every 10 seconds until a maximum of 200 users was reached.

Results from the test are illustrated in Figure 6, and average response times for the period of the load test are tabulated in Table 2. As the graph illustrates, LAMP out-

performed .NET by increasing amounts as the load increased. LAMP performed consistently as the load increased, whereas .NET started off with more impressive response times, but performance decreased consistantly as the load increased. This suggests that in terms of data access the LAMP stack provides a more scalable solution.

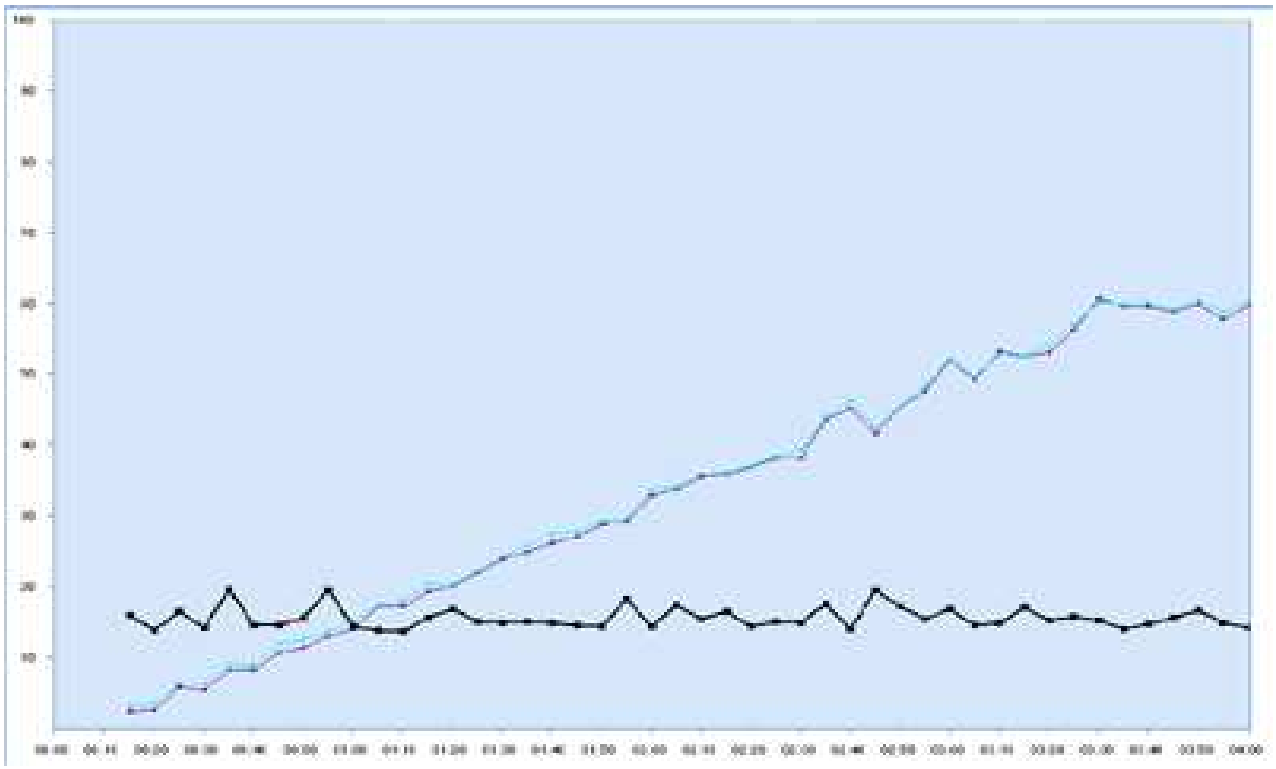

Figure 6 - Data Access Performance

Table 2 - Data Connectivity Performance

.NET SQL connection	.NET DataSet Connection (optimized)	LAMP MySQLi connection
3.36 seconds	3.30 seconds	0.016 seconds

3.3.3. Shortcomings

The lack of a suitable connection pooling facility in PHP means that for larger applications the programmer has to be vigilant as to how he goes about connecting to databases. He needs to be vigilant that connections which are opened are then closed. Object-oriented data access further complicates the process of centralized connection configuration. Traditionally a central file is created which holds connection string

details (host, user, password and database to connect to). This global connection can then be shared by all queries running on a specific page. However, when classes perform data connections page-level variables are out of scope to PHP classes, and the central configuration file cannot be directly accessed. A tempting alternative would be to include a connection class in the DAL. This class would be responsible for holding connection details and connecting to the database. The problem with this approach is that every instantiation of a data object would require a new connection to the database. Thus, for example, on a page such as the home page where three new data objects are created to access network, franchise and category data from the persistent store, we would require *three* database connections when only one would be sufficient.

The simplest solution to this problem is to pass a connection object to a class as an argument. This enables us to use a single connection instance across a number of classes, which improves performance and decreases the load on the database, but it is not an elegant solution. A major security flaw that results from this approach is that connectivity information (username, password, host and database) is included in the class, and can be viewed transparently by using the `print_r` reflection command. Listing 3 illustrates example output from running the `print_r` command on a data object containing connection details.

```
siteDO Object
(
    . . .

            [connection] => Resource id #12
            [dsn] => Array
                (
                    [phptype] => mysql
                    [dbsyntax] => mysql
                    [username] => theusername
                    [password] => *********
                    [protocol] => tcp
                    [hostspec] => localhost
                    [port] =>
                    [socket] =>
                    [database] => toast38
                )

   . . .
)
```

Listing 3 - Reflection of a Data Object revealing connection information

These issues will be discussed further in the section on security (chapter 4 – “Security”)

3.3.4. Conclusions Regarding LAMP Database Connectivity

The use of PHP/MySQL database-enabled dynamic web pages is well established. PHP offers inline connection functionality with a host of `mysql_` and `mysqli_` functions for connecting, querying and getting meta-data from MySQL databases. Direct connections are fast and easy to code.

PHP/MySQL is a lightweight and simple approach to data access for web applications. MySQL offers limited functionality in terms of features such as stored procedures, but in return offers fast response times. LAMP’s performance in terms of speed of connection was impressive and, in the tests executed, out-performed that of .NET convincingly.

PHP/MySQL is an adequate platform for high performance data connectivity. However, larger applications suffer from the lack of support for stored procedures and advanced database features in MySQL (prior to version 5). The lack of effective connection pooling features adds complexity and security vulnerabilities to web applications because database connection details can be vulnerable if not properly gaurded.

3.4. Data Access With .NET

Data access with .NET is easy, slick and powerful. .NET offers two major means of data connectivity: the `DataReader` and the `DataSet`. The `DataReader` is a super-fast, read-only, forward-only cursor [Plourde, 2005]. The `DataSet` is a powerful in-memory data-object which is able both to read from, and update to the database. Both `DataReaders` and `DataSets` build on top of the `SQLDataAdaptor` object. This is .NET’s data abstraction layer, and effectivly decouples the application layer from the persistent data source.

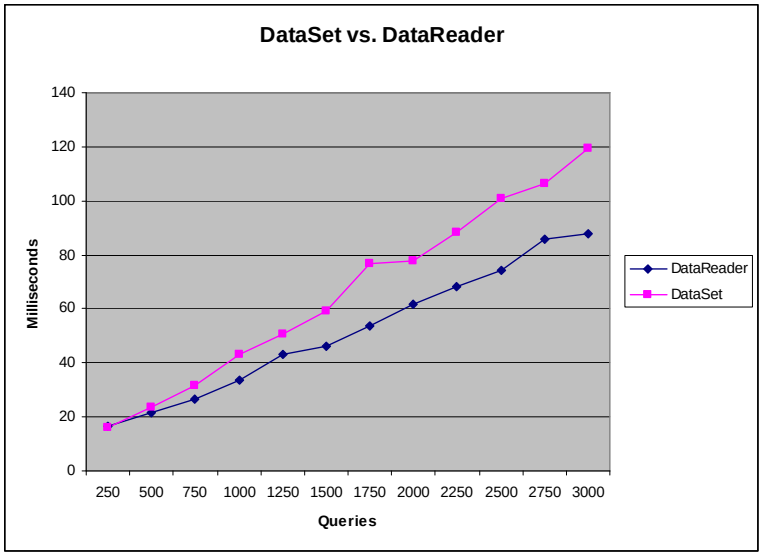

Figure 8 - Dataset vs. DataReader in a GUI application

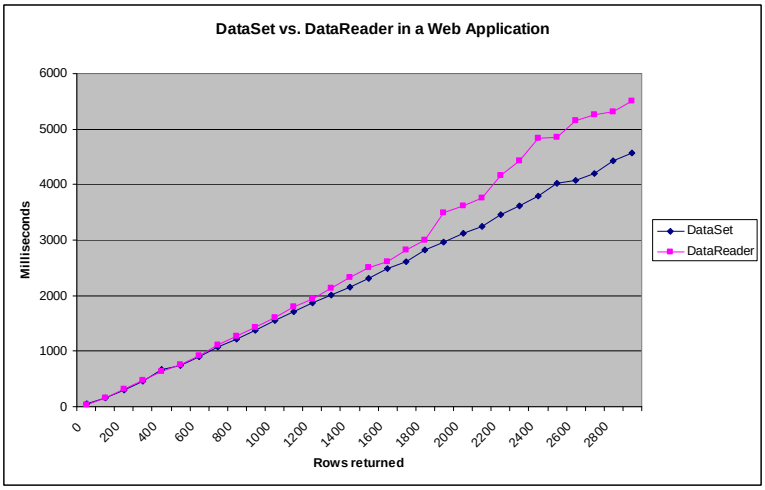

Figure 7 - Dataset vs. DataReader in a Web Application

A major decision at the outset of the project was whether to use the `DataSet` or `DataReader` as the corner-stone of the DAL. Traditionally the `DataReader` has been considered the faster option in terms of performance, while the `DataSet` is preferrable because of benefits in terms of flexibility and functionality. Figure 7 suggests there is a slight performance advantage to using the `DataReader` over the `DataSet`. Figure 7 illustrates the results of running increasingly larger queries against the database. Starting at 0 the number of rows returned was increased until 3000 rows were being returned from the database. These results were returned from a GUI application.

Interestingly Figure 8 paints an entrirely different picture. Figure 8 is an illustration of the same test, returning increasing numbers of rows from the database. However, this time output is bound to and displayed in a `DataList` on a web page. This suggests that in web applications the `DataSet` actually outperforms the `DataReader`. It is also interesting to note that the performance of the `DataReader` drops drastically and becomes erratic when the queries start to return more than 1900 rows. The improved performance of the `DataSet` with regards to web-based queries is probably a result of

the efficient caching mechanisms utilized by the object, and more effieicent databinding to presentation layer controls, such as the `DataList`.

Performance aside, the `DataSet` offers far greater flexibility and functionality as well as greater ease of use. Creating a DAL using the `DataReader` requires the developer to create a series of what .NET terms "business objects" – essentially classes similar to the Data Objects created in the previous section (3.3. Data Access with PHP). Creating these classes is time-consuming, tedious and prone to error. Alternatively, the visual toolkits in Visual Studio enable the developer quickly to create complex and powerful `DataSets` from the persistent layer. `DataSets` can handle complex relationships, and multi-table queries as well as updates, insertions and deletetions from the database. `DataSets` are a useful tool for quickly creating a DAL which is both high performance and flexible.

The `DataSet` object for our application is illustrated in Figure 9. Typed `DataSets` as shown above are a safe option for applications. Typed `DataSets` provide improved security because the integrity of queries or parameters passed to them are checked against the database structure so as to ensure data integrity and improve security through more accurate input validation [Plourde, 2005].

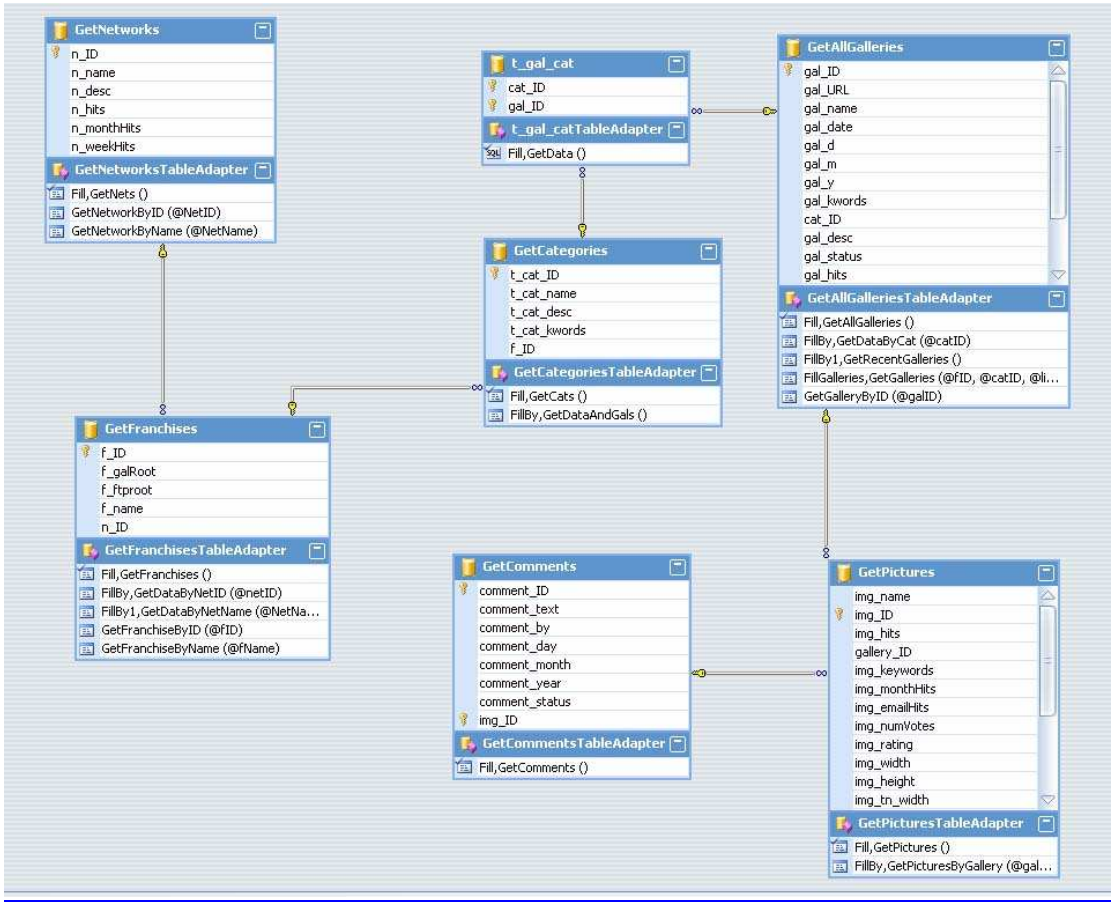

Figure 9 – DataSet

`DataSets` use the Query Analyser (illustrated in Figure 10) to create SQL queries or stored procedures, this eases the process of creating complex SQL queries or stored procedures. For this application we used stored procedures for all queries to the SQL Server database.

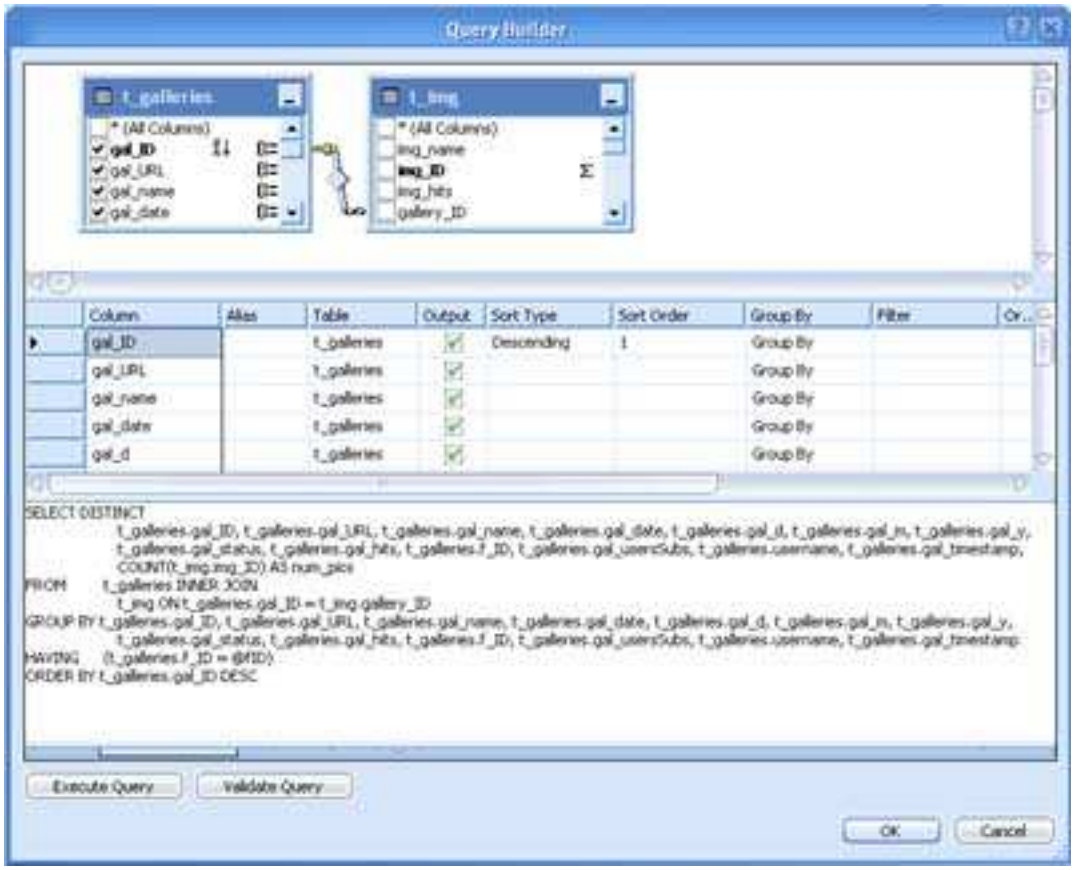

Figure 10 - Query Analyzer

Stored procedures offer improved performance and security compared to inline dynamic SQL code. They also serve to push data access issues down into the actual database server, decreasing the amount of code that needs to be written in the physical application, and improving separation of logic. An example of a stored procedure created by the query analyser is listed in Listing 4.

```
CREATE PROCEDURE dbo.DoLogin
(
	@username nvarchar(25),
	@password nvarchar(32)
)
AS
	SET NOCOUNT ON;
SELECT     user_id, user_active, username, user_password
FROM         phpbb_users
WHERE     (username = @username) AND (user_password = @password)
```

Listing 4 - Stored procedures created by SQL Query Analyser

The combination of `DataSets` and a fully featured database server such as SQL Server provide a platform for the development of secure, powerful and high performance solutions. Unlike MySQL, SQL Server offers mature support for stored procedures. The visual development environment simplifies creation of the DAL in .NET. The query analyzer (Figure 10) makes it easy to create complicated queries in minutes, while the visual dataset builder (Figure 9) enables us to quickly and easily bind stored procedures to `datatables` in the `DataSet`.

Using .NET we created the entire DAL without writing a single line of code. In fact, even binding the data to controls in the presentation layer is managed visually. Not only is code not written, but code is also not even visible. .NET use a feature called partial classes to remove all auto-generated script from the user's code-behind files. This provides an un-cluttered working environment for the developer, as well as protecting auto-generated code from being broken. Unfortunately this approach also has its problems as the developer now has less control over the application, as we can not be certain as to how data is accessed behind the scenes by Visual Studio. Partial classes and the visual programming environment will be analyzed in more detail in chapter 4 – "Security".

.NET also provides for impressive connection pooling facilities. A connection pool provides a global connection string for accessing the database. A connection pool manages efficient use of connections to the database, decreasing the load on the database and the web server. Connection pools are also more secure as they are typically handled by the system and stored in a relatively secure configuration file. With .NET connection pools are stored in the web.Config file. For additional security .NET can handle storing this information in an encrypted format. .NET is even able to store `connectionString` information as a key in the registry on the operating system. This is the most secure option.

3.4.1. Conclusion on Data Access with .NET

Data access is a critical part of any web application. The Visual Studio environment, and .NET 2.0, makes it easy to work with databases. Complex and powerful data

layers can be quickly created and managed. The process of creating stored procedures, queries and `DataSets` which make up the DAL can be completed without even writing a line of code. Even binding data or `DataSets`/Objects to controls, such as the `DataList`, or `DataGrid`, in the presentation tier can be completed without writing any code. This has made the process of data access in .NET extremely accessible and user friendly, although, as we will discuss later there are issues in particular with regards to security with this visual environment.

Data access with ASP.NET 2 has developed on top of ASP.NET 1.x, making data access easier and more flexible through improved databinding features and performance of the `DataSet`. The entire process, from creating the database structure, to creating an in-memory object representation of the data store using `DataSets`, and even binding data results to the presentation layer is user-friendly and manageable.

3.5. Conclusion

Developing the DAL in LAMP was a major task. Conversely, developing a more complex and powerful DAL in .NET was achieved in a fraction of the time. The power and large feature-set of SQL Server meant that we were able to push a great deal of Data Layer logic into the actual database server. Security is also better in the .NET application because data access logic is pushed further down the application layer and is thus harder to access, and connection data can be stored in more efficient ways, such as in the registry of the operating system.

Although a solid DAL was created for the LAMP application, it was a very time-consuming process. By comparison, the DAL could be created quickly in .NET. The final .NET product was more powerful and flexible largely as a result of the functionality of the DataSet, and the preferable feature-set offered by SQL Server over MySQL.

Chapter 4 – Security

4.1. Introduction

Web applications, due to their accessible public nature are natural targets for hacking. As such it is critical that applications are designed securely, and that the platforms on which they are run are locked down and secure. This chapter examines the security features and issues concerning the LAMP and .NET architectures.

This chapter provides a brief overview of common security exploits; in particular those relevant to web applications. This follows with some general methods of prevention. Thereafter details are provided as to how the respective applications were secured and any strengths and short-comings are mentioned.

4.2. Security Vulnerabilities on the Internet

Online internet-facing applications are vulnerable to security exploits at both the application and the server level. To secure a web application it is critical both to write secure code and to run this code on a secure server.

4.2.1. Common Web-based security exploits

4.2.1.1. SQL Injection

Almost all web applications use databases as a means of persistent data storage. SQL injection refers to the art of exploiting vulnerable SQL queries by ‘injecting’ custom SQL into SQL statements used by the application to make these queries act other than their intention, and typically to reveal sensitive data such as passwords, or to obtain unauthorized access to restricted areas.

For example, a query such as:

```
“select * from phpbb_users where username = ‘” + name + “’”;
```

When injected with a name value such as *Molly’ or 1=1*

Will run the following SQL statement:

```
“select * from phpbb_users where username = ‘Molly’ or 1=1”;
```

Thus, instead of returning only the details for Molly, this statement will return the user details for all users (since 1=1 is always true).

Some measures for combating SQL injection include:

- Escaping input
- Using stored procedures
- Strong input validation
- Running on least privileges
- Building SQL statements securely using parameterized commands
- Building SQL stored procedures securely by using the Transact-SQL *quotename* function and the sp_executesql command to execute SQL statements (these are features of SQL Server)
- Store connection strings and data connectivity details securely

[Howard, M and LeBlanc, D. 2005. pp: 397-411]

4.2.1.2. Session Fixation or Session Hijacking

HTTP is a stateless protocol. This can be problematic when it comes to creating stateful applications as there is no way to identify the client between page requests. This causes complexities when it comes to tracking a user’s interactions as they browse a site using, for example, a shopping cart or a system with user authentication. Cookies provide one means of applying state management to the HTTP protocol. Originally developed by Netscape, cookies are an extension to the HTTP protocol enabling session tracking thought the `set-cookie` response header and consequent `Cookie` request header. Therefore, cookies or sessions are a standardized means of managing state in over the stateless HTTP protocol. [Shiflett, C. 2005[a]. pp 40-41]

Session fixation refers to the process whereby a malicious user is able to impersonate a valid user by 'hijacking' a session which identifies the user. The example below illustrates how a malicious user can use the PHPSESSIONID to gain access to another user's identity [Shiflett, C. 2005[a]. pp 41]

.

Consider a php page called *fixation.php*.

```
<?php
session_start();
$_SESSION['username'] = 'chris';
?>
```

When we call this page with the PHPSESSID set to 1234 as such:
*http://example.org/fixation.php?**PHPSESSID=1234***

A session is stored containing the username. This session is stored with a session ID of 1234 as specified in the querystring. This ID is used to access the session throughout the user's session. Traditionally session variables are set to last 20 minutes or until the user closes the browser. Using session fixation, another user is able to hijack the session ID for a valid user, and use it to gain access to this user's information. [Shiflett, C. 2005[a]. pp 41-43]

For example, consider a second page responsible for outputting the information stored in the session variable:

```
<?php
session_start();
if (isset($_SESSION['username']))
{
echo $_SESSION['username'];
}
?>
```

By accessing this page and passing the session ID above, a malicious user is able to obtain access to the session data of our valid user. Entering: http://example.org/test.php?**PHPSESSID=1234** into any new browser page will present output form the initial session variable. [Shiflett, C. 2005[a]. pp 43-4]

Therefore, an attacker can simply link to your website, appending a session ID to the url. This sets up a vulnerable session ID on another server which can be used to hijack sessions from valid users who have followed this link, therefore giving the malicious user access to information stored in user's cookies [Shiflett, C. 2005[a]. pp 44].

4.2.1.3. XSS (Cross Site Scripting)

XXS refers to the act of a malicious user who is able to inject malicious `JavaScript` code into a vulnerable webpage which renders input variables to the screen without properly validating them. [Shiflett, C. 2005[b]. pp 14]

Similarly to session hijacking XXS attacks can lead to leaking of sensitive data, but can also lead to far more serious attacks as the hacker is essentially free to execute `JavaScript` code on vulnerable pages.

The following example explains a vulnerable site and its exploitation:

Consider a page which gets information from a form and renders this information to the screen as HTML:

```
if(isset($_GET['message'])){
        echo $_GET['message'];
}
```

Since the GET variable is not being validated at all, whatever the user submits via GET will be rendered to the screen. Thus a user could run inject a value such as:

```
<script language='JavaScript'>document.location = 'http://evil.empire.org?cokkies=' +
document.cookies </script>
```

This would send all the user's cookies to another page where this potentially private data could be stored [Shiflett, C. 2005[b]. pp 15-6]. In a page as used in this example, with no validation, the seriousness of the attack is limited only by the attacker's imagination and `JavaScript` skills.

XXS attacks are one of the more common exploits of web applications, and are potentially extremely harmful.

4.3. Technology Neutral Security Policy

Apart from technology-specific security details (which are outlined below), there are a number of technology-neutral security principals which where applied to the application to improve overall security, and decrease the seriousness of possible security violations.

4.3.1. Matching Input Data to a Pattern

Input validation forms the basis of much secure coding practice. To ease the process of strict validation an attempt was made to limit input text where possible. Therefore numeric IDs were selected to represent key values for data queries. These keys are typically passed from page to page in the URL and accessed using GET, making it easy for a malicious user to change their values and achieve unexpected results. By forcing these values to be numeric we are able to validate data ensuring that it is both numeric and of limited length.

4.3.2. Running on least privileges

The principle of running an application as a user with minimal privileges was applied to the database server. The website requires data access. However, it does not require unlimited data access. At the very most, a user may run SELECT, INSERT, UPDATE, and DELETE queries. A user was created with only SELECT, UPDATE and INSERT privileges. Deleting of data was handled by creating a field called ‘deleted’ and setting this value to 1 (true) on deletion by a user. A separate process, with only SELECT and DELETE privileges then periodically searches the database deleting rows where the field ‘deleted’ is set to true.

4.3.3. Storage of Private Data

Although no particularly sensitive data is stored in the system (such as credit card details), details such as passwords are hashed using a one-way md5 hash algorithm. Thus, should a malicious user gain unauthorized access to information, sensitive data is stored in a format which is of no use to the hacker.

4.4. Implementing a secure system with LAMP

4.4.1. Writing secure PHP

The golden rule of securing an application revolves around input validation. Because of PHP's lower level nature (by comparison to ASP.NET) the developer is well aware of all input data and can deal with it accordingly. The class `PEAR::Validate` provides a useful class for validation of data in PHP [Freuks[a]. pp: 159-63]. PHP also provides useful functions such as `strip_tags()` (which strips HTML tags from data), and `addslashes()`. `Addslashes()` can be used to escape input data for database queries and help guard against SQL injection. Furthermore, the `prepare()` and `execute()` functions of the `PEAR::DB` library, which was used throughout, provide a secure means of writing SQL which is resilient to injection.

A major vulnerability which is still present is the '`register_globals`' setting in PHP.ini (the configuration file for the PHP interpreter). Register globals enables all request variables to be accessed transparently as normal variables. Thus, assume we use POST to send a variable named `$authorized`; with register globals on. This variable can be accessed transparently in code as `$authorized`. On the contrary, with register globals off, this variable needs to be accessed explicitly as: `$_POST['authorized']`. Because, in PHP, variables do not have to be initialized, a malicious user can inject values into the `$authorized` variable by passing an appropriately named request header to the page. The following code snippet illustrates the vulnerability:

```
<?php
// define $authorized = true only if user is authenticated
if (authenticated_user()) {
   $authorized = true;
}

// Because we didn't first initialize $authorized as false, this might be
// defined through register_globals, like from GET auth.php?authorized=1
// So, anyone can be seen as authenticated!
if ($authorized) {
   include "/highly/sensitive/data.php";
}
?>
```

Listing 5 - register globals exploit [PHP manual. 2005]

By passing a header response such as authorized=1 (in this example done by passing the GET header via a URL such as: `auth.php?authorized=1`), the malicious user is able to set `$authorized` to true, and therefore gain unauthorized access to the restricted area.

This vulnerability can be avoided by initializing variables correctly. However, with register globals on, it becomes very easy inadvertently to write insecure code. This vulnerability highlights a security issue with PHP code. The PHP scripting language is written to be an easy to use, flexible and convenient language. Features like the fact that variables do not need to be explicitly declared, and that the language is not strongly typed make writing insecure code easier. Potential vulnerabilities (such as above), may go by undetected by the human eye, and will not be picked up by the compiler or interpreter as they would be in more formal, strongly typed languages such as C# or Java.

A major short-coming of the PHP engine is the lack of an adequate connection pooling mechanism. As a result connections are typically stored in a central configuration file. This is a severe vulnerability as all database connection details: username, password, database and host are typically contained in this file in plain text. Very insecure systems store configuration details in .inc text files. Since these are not parsed by the PHP interpreter, browsing to the file will display the file in a browser as a text file. A simple Google search such as: *config filetype:inc* can reveal some very interesting results.

The steps taken to limit this vulnerability are to store configuration data in a parsed file such as a .php file. Thus a user browsing to the files location will only see a blank file. In our applications the configuration file is stored above the web root so that it is not accessible from the internet. However, should a user compromise the server they could obtain the database connectivity data from accessing the unencrypted configuration file.

As an extra layer of security, the data in the configuration file could be encrypted, but this is not a standard method, and was not implemented in the application.

4.4.2. Security with Apache

Secure access to the remote Apache server was facilitated via SSH (Secure Shell). FTP (File Transfer Protocol) is a useful, yet infamously insecure tool used for the transfer of files. As a result, as far as possible SFTP/SCP (Secure FTP) or HTTPS was utilized to manage file uploading to the remote server. These protocols are used by the Web Developer to upload code files to the website. However, due to the nature of the website it is necessary that photographers are also able to upload photos to the website so that they can be added to the gallery by the administrator. A tool was developed which enables users to add their own galleries to the user submissions transparently. This tool uses FTP to transfer the files, introducing potential security vulnerabilities – a user could sniff the access details for the FTP used by this client and use these details to obtain unlimited FTP access to the server. Using this they could upload an executable file and run this file on the remote server, thus compromising the server.

However, using Apache mime-types we were able to eliminate this vulnerability. The FTP account used by the upload client was given access to a single folder. This folder was placed above the web root and as such is not directly available from the internet. Further, by editing the `.htaccess` file for the folder we are able to limit the accepted mime-types for this folder to accept only the jpg/jpeg extension, thereby eliminating the threat of users uploading executable files via FTP and running these on the server to compromise the server.

A good feature of Apache is its modular design. Two modules which add a further layer of security to the application are `mod_security` and `mod_rewrite`.

`Mod_rewrite` is a very popular tool for re-writing URLs. A major use of `Mod_rewrite` is to produce 'search-engine-friendly urls'. Therefore, using `Mod_rewrite` the developer can re-write urls from the form: www.38.co.za/galleries.php?catName=Recent&catID=23 to www.38.co.za/galleries/Recent/23. As may be seen, the latter form is far friendlier to the human eye, but more importantly it adds a layer of security by hiding implementation details from the user. In the example above, information revealing the language in which the applications is written is hidden by removing the .php extension from the URL. Furthermore, the names of the GET variables (`catName` and `catID`), are also removed from the URL. Although these are minor issues, they do add another layer of security to the application.

A more important security gain as a result of `Mod_rewrite` is that it uses regular expressions to rewrite urls, and as so adds an extra level of validation to url and GET input.

For example, the following rule:

```
RewriteRule ^galleries/([0-9])/$ galleries.php?c=$1
```

will force the value of c to be a single numeric value between 0 and 9. A value not matching this pattern will return a 404 page not found error. Using regular expressions with `Mod_rewrite` ensures that urls match suitable patterns, and hence adds another layer of security to our application.

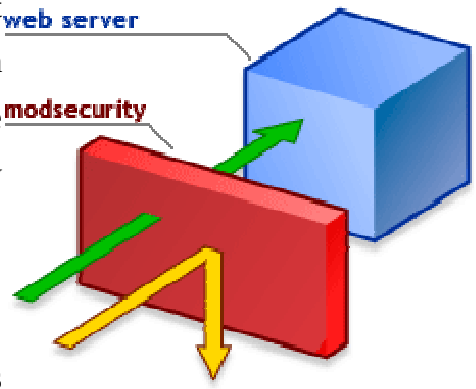

Figure 11 - Mod Security

`Mod_security` is an Apache module which acts

as an “intrusion detection and prevention engine for applications (or a web application firewall)” [modsecurity. 2005].

`Mod_security` was used to add yet another layer of security to the application. Rules where defined to help prevent attacks such as XXS, path traversal, SQL injection, as well as limiting potentially dangerous privileges assigned to the web server – such as limiting HTTP file uploads to only valid image files.

In addition to the general rules mentioned above, `Mod_security` also played a useful role in securing our Web Services from attack. Since HTTP/HTTPS (the protocols on top of which Web Services are usually conveyed) typically are able to pass unobstructed through a firewall, `Mod_security`, as an applications firewall, plays an important role in securing Web Services running on Apache, in particular against attacks such as variable-length buffer injection, meta-character injection, SQL injection and SOAP fault code disclosure. [Shah, S. 2005]

4.4.3. Securing MySQL

A number of MySQL accounts were created, all accounts with least privileges enabled. A web user account was also created. This is the main account used for drawing the required data for basic user browsing of the website and some interactions as well as tracking. This account was enabled with only SELECT, UPDATE and INSERT privileges. Although deletions are necessary, these are handled by the front-end initially simply by setting a flag which marks a row as deleted. An automated process is executed periodically using a `cron` script which searches the database and physically deletes all files from the database. Cron is a linux tool for performing chronological scheduled tasks. This script utilizes a different database user account which has SELECT and DELETE privileges only.

Another inherent advantage of the PHP/MySQL combination is that only a single SQL command can be executed per statement. This further decreases the likelihood of successful SQL injection.

For example, a common SQL injection exploit such as the following:

```
SELECT * from phpbb_users where username = ‘$username’ and password =
‘$password’

Injected with $password = a’;drop table galleries /*
$username = a;

To produce:

SELECT * from phpbb_users where username = ‘a’ and password =
‘a’;drop table galleries /*
```

Would not be successful as the malicious second statement would not be executed by the PHP `mysql_query()` command.

4.5. Implementing a secure system with .NET

4.5.1. Security in the Visual Environment

The ASP.NET, and indeed the entire .NET programming environment is largely visual and as such, the typical user relies fairly heavily on the Microsoft programming environment to implement security checks against common exploits. With Visual Studio 2005 and ASP.NET 2.0 this has been taken one step further with the introduction of the partial class. Partial classes are classes which are hidden away in the assembly and are not accessible to the developer. All auto-generated code is stored in these partial classes. This has the advantage that the development environment is less cluttered, and that the developer is unable to break auto-generated code. However, the obvious disadvantage is that developers do not know what they are actually doing! Furthermore, there is no way even to see what is being done as all this code is locked away in a partial class.

4.5.2. Database Security

Database security is particularly important in relation to databinding and SQL injection. As mentioned in the chapter on data connectivity (chapter 3 – "Database Connectivity"), with .NET 2.0 it is possible to perform multi-tiered data-connectivity without writing a single line of code. Since all this code is stored in partial classes, should the user employ the visual tools to connect to the database, no clue whatsoever is given as to how data in sanitized prior to insertion into a SQL statement or stored procedure.

While it is possible to add validation logic to variables and pass these sanitized variables to the SQL statements or stored procedures, given the visual nature of the ASP.NET environment, this is an unlikely step for most developers using Visual Studio.

To understand how pages were sanitized in the Visual Studio environment prior to insertion into stored procedures or SQL queries, a series of typical SQL injections attacks were launched against a page which was created to be as vulnerable as the automated databinding tools would allow.

The page, `vulnerable.aspx`, runs a query against the database which accepts a String input in the WHERE clause:

```
"select * from t_img where (img_keywords = @input)".
```

The basic SQL injection tactics outlined in Table 6 were run against the page. The first attempt is to alter the SQL so that all rows of the table are returned. The second injection attempts to execute a malicious SQL query by injecting it into the querystring.

Table 6 - SQL injection

Input	Injected SQL
`a' or a=a'`	`select * from t_img where (img_keywords = 'a' or 'a'='a')`
`');drop table t_img--`	`select * from t_img where (img_keywords = '');drop table t_img--)`

[Howard. and LeBlanc, 2002]

All attempts at injection returned an empty `resultset`. To further check the robustness of query validation the single quote in the injected string was replaced by its hex value (char(0x27)). This substitution can foil attempts by the validator to escape dangerous characters such as the single quote. The attempts were, again, unsuccessful.

It would appear that Visual Studio performs basic sanitization of data such as escaping potentially dangerous characters like the single quote. Some form of type checking is also performed against the types defined in the database structure. It is evident that input data is checked to conform to the appropriate type (int32, int64, decimal, string etc.) expected by that column in the database. It is also possible that checking is performed against the database for the length of input variables according to restraints placed on them within the database.

It is evident from the examples above that with regards to data-access Visual Studio performs both security checks against input data and integrity checks against the database structure. It appears that solid base-line validation and security is implemented inherently by the application.

One criticism of the visual data-access approach is that pattern matching of input data (for example matching an expected email or code (such as a student number) input against a regular expression) is obviously not performed by Visual Studio. This is often the most secure means of input validation.

Furthermore, since the vast majority of development, if not all, of ASP.NET web applications are likely to be created in Visual Studio or Visual Web Developer (a smaller, limited version of Visual Studio for ASP.NET development) should a vulnerability be spotted in the auto-generated code, it is safe to say that the vast majority of ASP.NET applications will be vulnerable to such exploits. Fixing these vulnerabilities in web applications would be no small task, especially since developers do not have access to all the code.

The visual development environment makes it harder for inexperienced users to write vulnerable code. However it limits the level of control that more experienced developers have over the application. More importantly, it limits the clarity of knowing exactly what is being input into the application, which could lead to a more lax approach to input validation.

Most security exploits attack logical vulnerabilities in badly written code by injecting unexpected values. In contrast to PHP, C#, a fully fledged strongly typed and properly object oriented language, improves the robustness of code written because insecurities arising from improperly instantiated variables and irregular type checking are picked up at compile time by the compiler.

4.5.3. Validation Controls

The .NET validation controls are a popular feature of the Visual .NET environment. The controls enable easy client and server side validation of form input data. However, vulnerability also arises out of common misconceptions of the workings of these controls. This is also a good example of how assumptions made in the visual environment can lead to insecurities in code.

To illustrate this example, consider a textbox with a `RequiredFieldValidator` attached to it, as illustrated in Figure 12.

Figure 12 - Vulnerable Textbox

The following code is attached to the button

```
protected void Button1_Click(object sender, EventArgs e)
    {
        if (TextBox1.Text == "")
        {
            Label1.Text = "should NEVER get here";
        }
    }
```

Because the textbox is theoretically protected by a `RequiredFieldValidator` it is expected that the label's text should never be set to: "should NEVER get here". However, by turning off `JavaScript` in the browser the code is executed on the on the server-side without proper validation taking place, as illustrated in Figure 13(note that the label's text is now set to "should NEVER be here".

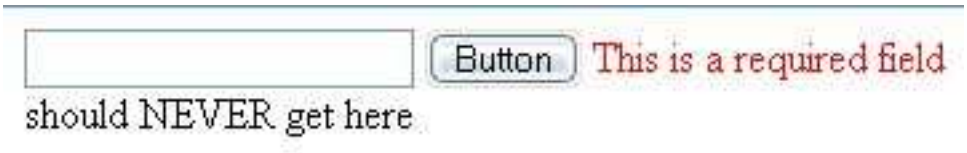

Figure 13 - Exploited Validation Control

This exploit is not really a weakness in .NET framework, but rather a result of poor coding due to assumptions made by the developer as to what the visual environment will do behind our backs. Fixing this validation is simple, as illustrated by the code below:

```
if (Page.IsValid)
        {
            Label1.Text = "should NEVER get here";
        }
```

Although not explicitly a weakness of the .NET framework, this does show how reliance on the visual environment to produce secure code may be a misconception and lead to insecure code.

4.6. Conclusion

Security is critically important for web applications due to their public nature. While many security features are technology-neutral, some technology-specific features can strengthen security. Apache in particular is renowned as having a good security record and is a secure base on which the LAMP infrastructure is built. PHP, due to implementation being at a lower level than the more abstracted ASP.NET environment, helps the developer to write secure code as little happens behind the developer's back, and he is able to keep track of inputs and outputs fairly closely. However, the flexibility of PHP, the fact, for example, that it is not strongly typed and

that variables do not need to be instantiated, enables the inexperienced or un-conscientious developer to produce looser, more insecure code. By contrast, the more formal constraints of a language such as C#, help produce tighter, more secure code.

The visual environment manages a good deal of security for the developer, and in many respects the developer can rely on the development environment to oversee many of the security issues. However, this can lead to developers making assumptions as to what the environment will handle for them, which can lead to vulnerabilities.

Ultimately, .NET provides built-in solid base-line security, while security with LAMP is more the responsibility of the developer. An experienced developer using the LAMP platform has greater flexibility and control over the environment and thus is more aware of security issues. However an inexperienced developer has far less protection from creating vulnerable applications.

Chapter 5 – Web Services and XML

5.1. Introduction

XML and Web Services offer a means for standardized communication between machines and applications over a network. This chapter discusses the general need for Web Services in the modern web development environment as well as within the problem domain of this application. Thereafter we expand on the process of implementing Web Services within our application.

5.2. The Use of Web Services

Web Services offer an excellent means of producing a public API for an application. Web Services empower the user to produce a public API which can be used by others to extend and add functionality on top of the basic service architecture provided by the core application.

Web Services offer an excellent means of producing a public interface for an application. Influential companies such as Amazon, Google, Yahoo and e-Bay all offer a web service interface to their applications. This enables developers to interact with code made accessible by the vendor (Amazon, Google, Yahoo or e-Bay). Developers can build useful applications on top of the core functionality provided by the vendor. For example, developers can create applications which can search Google or Yahoo, or get the best offers on Amazon or e-Bay. Web Services contain highly abstracted business logic. With Web Services, because interaction is controlled by standards such as SOAP and XML, implementation details are not important to a client trying to consume the service. All a client needs to know is the location of the service and the methods which the service makes available [Trenary 2002: p. 370].

In the project we dealt with both creating the core Service architecture, and creating clients which were able to consume these services. Web Services were used to

provide an interface for possible future development by users as well as to enable remote administration.

Because the website is heavily image based, there are a number of sandbox issues associated with development using a purely server-side language such as PHP. For example, for security reasons, PHP running on a remote host cannot save files to the user's hard-drive. However, a client-side GUI has the appropriate privileges to perform these tasks. Therefore, by combining the power of a GUI application, with Web Services we where able to produce a useful tool which is able to browse the website and download entire galleries to the user's computer with the click of a button (see Figure 14).

Figure 14 - Screenshot of the .NET GUI

This illustrates how useful Web Services and XML are when it comes to interoperability and remote syndication. The GUI uses web services and XML to interact with a remote site (www.38.co.za), and provide extra functionality to the user. The GUI above was developed in C# .NET 2005, and runs on the windows platform.

Using Web Services and XML it is able to interact with code written in PHP running on a remote Linux server.

5.3. Implementing Web Services with LAMP

Support for Web Services is relatively new in PHP. Although the `PEAR::NuSOAP` classes enabled the creation of XML-RPC style Web Services in previous versions of PHP; It is the `SoapServer` and `SoapClient` functions (see Listing 7), new to PHP 5, which seriously deal with SOAP Web Services in PHP.

The code below (Listing 7) provides an illustration of the use of these functions both to create and to consume web services with PHP.

Listing 7 - using the SoapServer and SoapClient classes

CODE:

Server.php

```
function HelloWorld(){
      return "Hello World";
}

ini_set("soap.wsdl_cache_enabled", "0"); // disabling WSDL cache
$server = new SoapServer("test.wsdl");
$server->addFunction("HelloWorld");
$server->handle();
```

Client.php

```
$client = new SoapClient("test.wsdl");
$return = $client->HelloWorld();
echo $return;
```

Despite the simplicity of this code, creating SOAP web services in PHP is not so straightforward.

A major short-coming in terms of web service support in PHP is the lack of a WSDL generator – requiring developers to create their WSDLs by hand – which is tedious, frustrating and non-trivial. There are a number of attempts at WSDL generators for

PHP; however, those that where tried out for this project were found to be incomplete and/or buggy.

A major reason for the lack of a suitable generator is: “PHPs lack of function parameter prototyping” and the fact that PHP is not a strongly typed language (a fact which can make *consuming* Web Services a much simpler task) – hence generators are unable to extract sufficient META-information to create WSDL on-the-fly [Schlossnagle, G. 2005]. The problem springs from the structure of the language and finding a solution is not a trivial task.

The need for the manual creation of WSDL documents is a hindrance to successful production of SOAP Web Services, and led to many complications in attempting to create and debug PHP web services. Once a successful server had been created, the service was consumable using only a PHP client. Attempts to consume the service using .NET simply returned nothing instead of the desired results. Attempts to consume services in Java failed.

As a result of the complexities of creating web services with PHP, an alternative method of implementing the SOA (Service Oriented Architecture) was researched. This alternative is REST (Representational State Transfer).

REST is described by Roy Fielding, who coined the term in his doctoral dissertation, as “an architecture style for networked systems” [Costello. 2005]. Unlike those web services which are RPC-style with remote method calls, REST is an extension of the normal use of the HTTP protocol. REST services deal with web requests and responses just as normal HTML web pages do. However, instead of rendering responses in HTML, responses are rendered in XML, and are thus accessible and consumable by remote clients. Figure 15 illustrates the REST architecture.

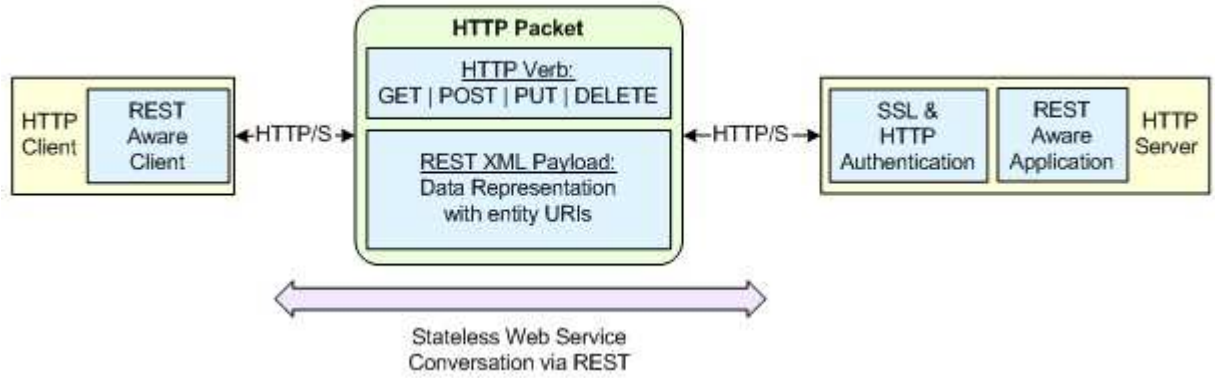

Figure 15 - REST architecture [Hinchcliff. 2005]

REST web services define a 'resource' (instead of an object). A resource is a page, for example: http://www.38.co.za/services/REST/getGallery.php. These resources are accessed via a URI (Uniform Resource Identifier). Parameters can be sent to the page using HTTP verbs such as GET or POST. For example, a web service equivalent to a SOAP call such as `getGallery(int id)`may be called using REST simply by sending a web request to a URI such as: http://www.38.co.za/services/REST/getGallery.php?id=5. Here the parameter is sent using GET and is simply appended to the URL. Using mod_rewrite this can be further simplified to: http://www.38.co.za/services/REST/getGallery/5.

REST is the architectural style of the internet; REST web services use this architectural style of HTTP requests and responses to provide distributed remote access to networked resources. REST web services typically use XML as a transport medium for returning responses to the user. Therefore, a typical response to a Web Service call to the above URL might be an XML document like that below:

```
<gallery id=5>
        <name>The Union Fri 5 Feb 2004</name>
        <desc>Friday night at the Rhodes Union</desc>
        <keywords>Rhodes Union </keywords>
        <hits>10232</hit>
</gallery>
```

There are a number of benefits to using REST Web Services, namely: scalability, performance, security, reliability, and extensibility [He. H. 2004]. Unlike SOAP, REST profits from a lower meta-data overhead in message passing, making it faster

and more scalable. Furthermore, since REST has been termed merely a specification of the architecture of the internet [Fielding. 2000], all the security features and concerns of the HTTP and HTTPS protocols also apply to REST. REST is growing in popularity and has been adopted as an alternative platform for web services by influential companies such as Amazon [He. 2004], Yahoo, e-Bay and Flickr [Frietag, 2005].

New features native to PHP 5 offer improved support for dealing with XML which is critical for development of REST web services. The `DomDocument` object offers good support for both creating, reading and validating XML.

The following code (Listing 8) illustrates the creation and consumption of a simple REST Web Service with PHP. The service simply returns a list of all the networks in the database as an XML documents; or a specific network should the user specify this using GET:NetName=Name.

```
<?php
/*
 * Service.php
 * Returns XML output of all the networks or a single network if specified with
$_GET['NetName']
 * Example URLS: http://www.38.co.za/services/REST/Network -> Shows all Networks
 *   http://www.38.co.za/services/REST/Network?NetName=Schools   ->   Shows   only   the
selected Network
*/

. . .
/*
 * Perform any necessary business logic here…
*/

//Set the content-type to render XML
header("Content-Type: application/xml; charset=utf-8");

if(Validate::string($_GET['NetName'],$NameCheck_options)) //chosen single network
{
        $n = new netDO(); //create a network Data Object
        $n->getNetworkByName($_GET['NetName']); //Get the specified name
        $dom = new Networks(); //declare the DOM
        $dom->load("Networks.xml"); //load the dummy networks XML file
        $dom->addNetwork($n->getName(),$n->getID()); //add a network node to the XML
}
else //show all nets
```

```
{
        $site = new siteDO();
        $site->getAllNets();
        $dom = new Networks();
        $dom->load("Networks.xml");

        while($site->nextNet())
        { //add all networks as nodes to the xml document
                $dom->addNetwork($site->getNetName(),$site->getNetID());
        }

}
//print results to the HTTP response
print $dom->saveXML();
?>
```

Listing 8 - Creating a REST Web Service

This code renders the following XML output:

```
<Networks>
        <network>
             <title>Universities</title>
             <ID>1</ID>
        </network>
        <network>
              <title>Schools</title>
              <ID>2</ID>
        </network>
</Networks>
```

This XML can be consumed in PHP with the following code:

```
<?php
/*
* Client.php
* Create a network Data Object, populate it and list all its child franchises
*/
//Galleries:
$dom = new DomDocument();
//Call the web service
$dom->load("http://localhost:3000/38_2005_ii/public_html/services/REST/Network");
$titles = $dom->getElementsByTagName("title");

foreach($titles as $node) { //read the titles of all nodes in the XML document
   print $node->textContent . "<br/>";
}
?>
```

Listing 9 - A PHP client

Or in C#.NET with the following code:

```
string lcUrl = "http://localhost:3000/38_2005_ii/public_html/services/REST/Network";

//request the page (necessary to perform server-side logic)
WebRequest r = WebRequest.Create(lcUrl);
//read in the XML stream
XmlReader xReader = XmlReader.Create(lcUrl);
while (xReader.Read())
{
   responseTxtBx.Text += xReader.Value; //add the text from the nodes to a textbox
}
```

Listing 10 - A .NET client

Furthermore, any code included in `service.php` will be executed on the server-side, and thus code can be executed remotely from a client script such as the one above. In fact, PHP offers the ability to execute web service calls like the above in the background, without affecting the load-time of pages. The method, used in applications such as pseudo-cron (www.bitfolge.com), uses a simple HTML image tag to call the web service. By passing the web service URI as the 'src' attribute for the image, we cause the browser to spawn a new process to fetch the URL (see Listing 9). The service is effectively executed in a separate thread, and does not effect the loading time of the current HTML page. This method can of course only be used for services which do not return values which we wish to use in the current page.

```
<img src = "http://localhost:3000/38_2005_ii/public_html/services/REST/Network"
height=0 width=0 />
```

Listing 9 - Calling a Web Service using the HTML image tag

5.3.1. Conclusion on LAMP Web Services

Support for XML and web services is fairly new to PHP with most support being new or rewritten for PHP 5 [Stocker, C. 2005]. As such the technologies are new and in many case simplistic.

The tools provided by PHP for consumption of XML and web services, such as `SoapClient`, `SimpleXML` and `DomDocument` are easy to use and work well. In fact, the SoapClient excelled as the only client able successfully to consume PHP SOAP web services as well as .NET services. In the interests of this project it was deemed preferable to keep web services simple, and the features provided by PHP were well

suited as they were both non-complex and functional. It is highly possible that PHP web service consumption would suffer with more complex web services dealing with complex types and arrays.

Creating web services in PHP was a difficult task: attempts with `SoapServer` were unsuccessful and, because no useful WSDL generator was available, creating SOAP Web Services in PHP was tedious.

Support for SOAP web services in PHP is new and at present is not well implemented. REST web services seem far better suited to PHP. Using the new improved XML support offered in PHP 5 by `DomDocument` we were able to successfully implement a REST service architecture which could be consumed with relative ease and transparency by multiple clients in multiple languages.

5.4. Implementing Web Services with .NET

By contrast to the support for web services in the LAMP platform, support for both consuming and creating web services is built into the .NET framework. Web services are an important component of the .NET framework as they improve interoperability and cross-platform integration between applications, lessening the impact of the platform lock-in characteristic of Microsoft applications. WSDL creation and deployment are all handled within the Visual Studio environment.

The .NET environment uses reflection to manage the intricacies of deploying and creating web services. The `[WebMethod]` attribute is used to describe to the .NET CLR (Common Language Runtime) the function which follows. The `WebMethod` attribute has properties which enable even more control over the service environment, properties such as: `BufferResponse`, `CacheDuration`, `Description`, `EnableSession`, `MessageName` and `TransactionOption` These simple directives enable the developer to perform tasks such as caching, buffering, session-state management and transactions, which would typically be extremely difficult to program manually. The `MessageName and Description` properties can be used to

provide further information to the end user of a service [Ferrera and MacDonald. 39-44].

.NET provides extensive support for creating both simple and complex web services. The .NET framework has abstracted many of the complex tasks associated with web service development enabling the developer to concentrate on code rather than on internal intricacies.

Consumption of web services with .NET is also handled by the environment. Services can be added to a project as a "web reference". After a web reference has been added to the project it is accessible from the project in the same way as a normal namespace. Web service consumption of .NET services is, as one would expect, well supported with the ability to easily pass and return arrays and complex data-types between the client and the service.

XML support in the .NET framework is also well established, and REST services can be created using the `XmlWriter` object of the `System.XML` namespace. REST services may be consumed using the `XmlReader`.

Succesful implementation of both SOAP and REST Web Services were easily developed using .NET and were used to test the performance of REST and SOAP Web Services. Although Amazon's Jeff Barr claims that REST Web Services are six times faster than SOAP Services [Trachtenburg] our tests (illustrated by Figure 16) seemed to suggest the opposite. Figure 16 illustrates the results from a load test performed in the Visual Studio testing suite. The orange line represents a REST service, the green line represents the SOAP service, the red line represents the user load. The user load was incremented by 20 users every 10 seconds and reached a maximum of 400 concurrent users. The respective average response times (average time to last byte) for the SOAP and REST service over the four minute test period were 1.42 seconds and 1.60 seconds respectively.

The faster response times of the SOAP implementation by comparison to the REST services is probibly due to poor implementation of the REST web services. Both the

`XmlWriter` object in .NET and the `DomDocument` object in LAMP require that responses are written to files on the file-system. This extra read-write time, as well as the use of DOM instead of the more efficient SAX implementation, may well have costed the REST services in terms of response time.

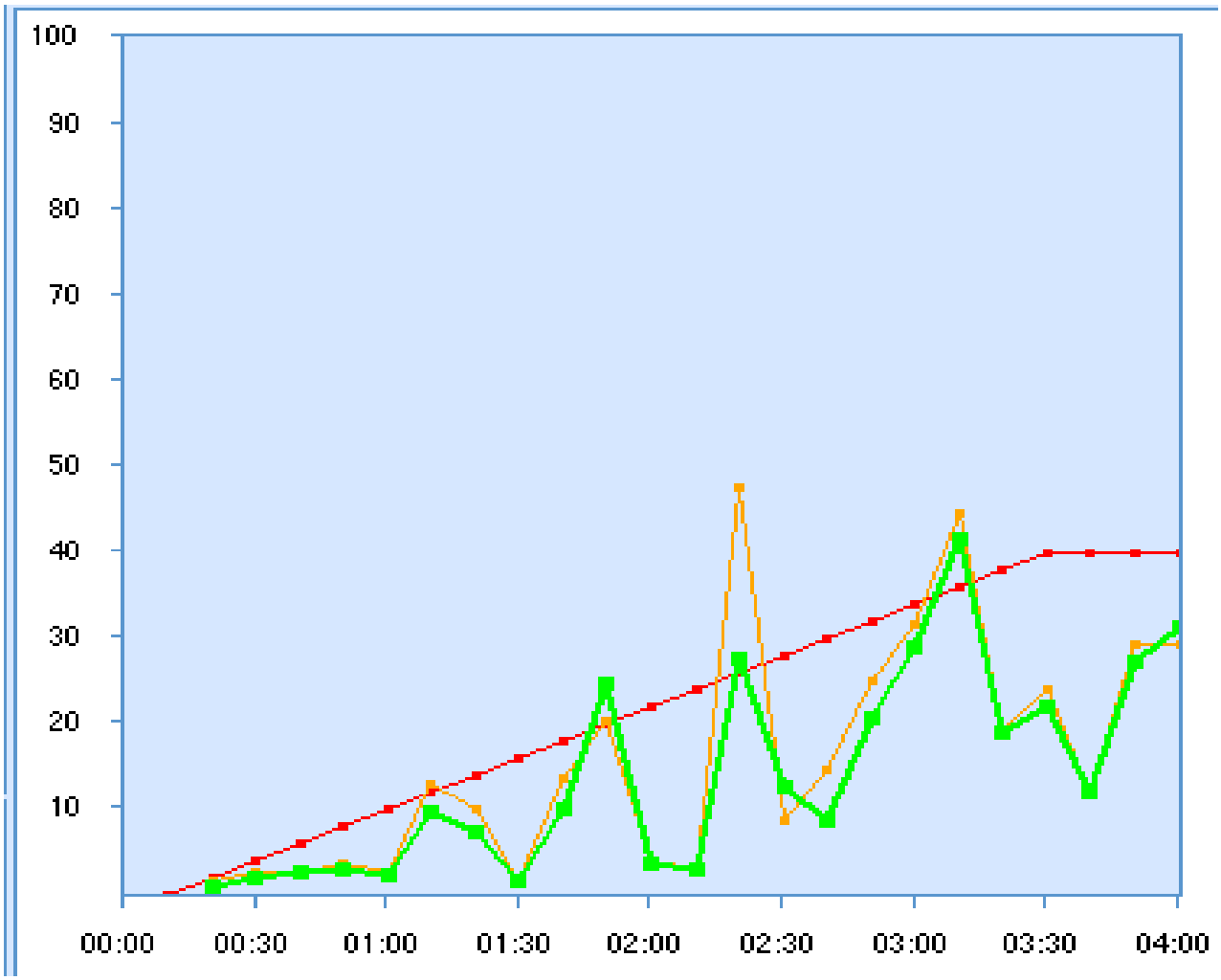

Figure 16 - REST Services vs. SOAP Services (.NET), Orange is REST, Green is SOAP, the red line represents user load

After testing the .NET implementations of REST and SOAP, the LAMP REST service was added to the test. The results from this test are illustrated in Figure 17 below. From Figure 17 it is clear that the both the .NET services far out-perform the LAMP service. The LAMP service (depicted in green) returned an average of 35 seconds response time, by comparison to the .NET REST and SOAP services which returned 0.27 and 0.30 seconds respectively. Furthermore once the load hit 360

concurrent users, the LAMP service started failing. The average times for this test are anotated inTable 10.

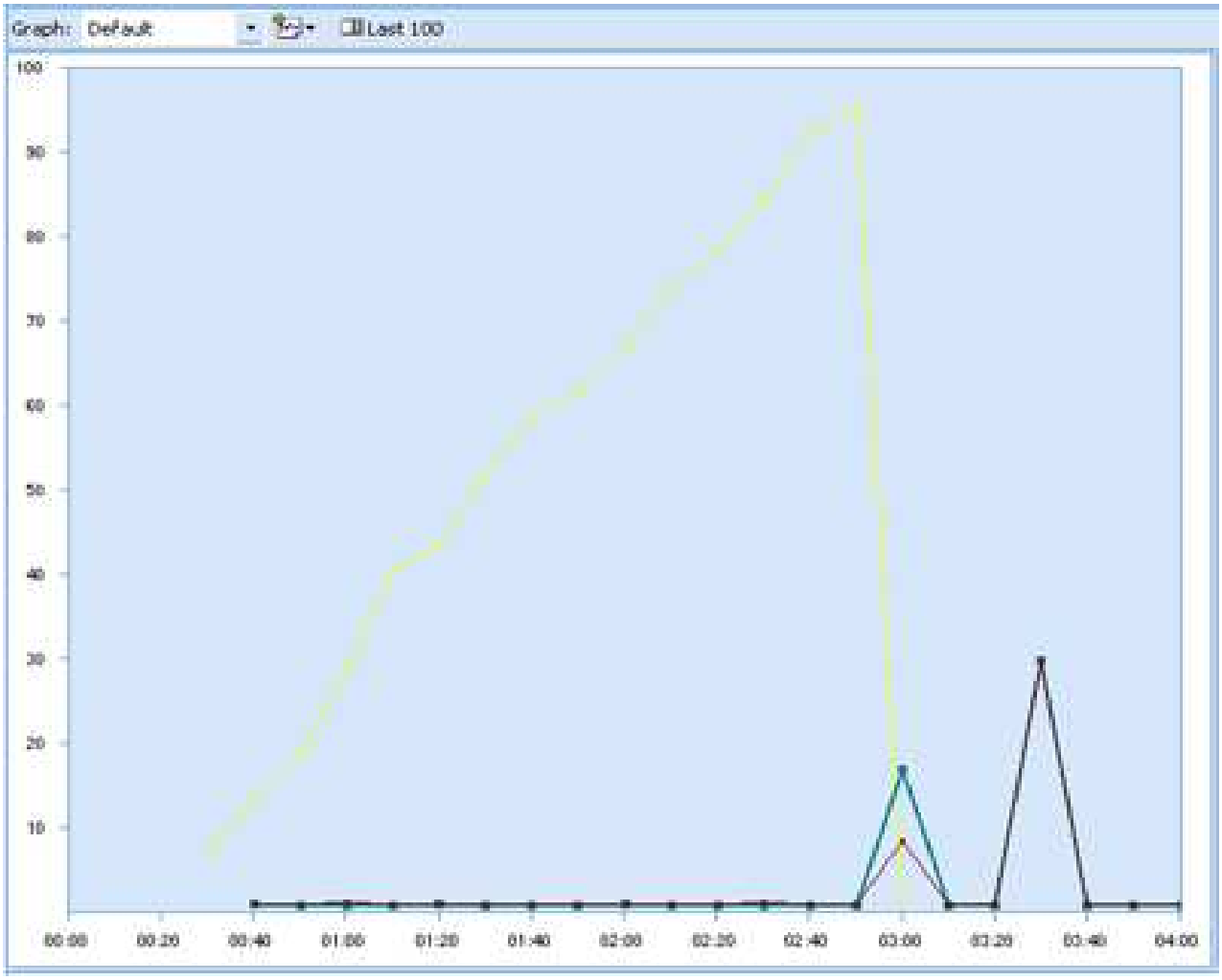

Figure 17 - .NET SOAP and REST services and LAMP REST Service

Table 10 - Average response times for Web Services

PHP REST Service	.NET SOAP Service	.NET REST Service
35 seconds	0.27 seconds	0.30 seconds

5.5. Conclusion

Web service support in the LAMP platform is fairly new. By comparison, web services are a mature technology within the .NET framework. The .NET environment far out-performs LAMP in terms of both capability and performance.

New features such as `SoapServer` and `SoapClient`, and particularly the new XML support offered by the `simpleXML` and improved `DomDocument` objects in LAMP show a step in the right direction by the LAMP platform in terms of XML and web services; but these implementations are far behind the mature web service development environment plumbed into Visual Studio 2005 and the .NET framework. The .NET environment for web wervices was shown to be far superior to that offered by LAMP in all cases which were tested for this project.

Chapter 6 – Performance

6.1. Introduction

This chapter evaluates the optimization techniques offered by the technologies. In particular we evaluate the caching mechanisms offered by the relative technologies. The effectiveness of optimization attempts is gauged by the relative speed-up achieved by the mechanisms as well as by the flexibility with which the techniques can be applied.

Some benchmarking tests are also performed so as to gauge the relative efficiency of the technologies against each other. In particular we test performance in the important fields of data access and rendering, and for both SOAP and REST Web Services. The Apache and IIS web servers are also benchmarked.

The section on LAMP examines the PEAR `Cache::Lite` caching tool as well as taking a look at other optimization mechanisms such as 'pseudo-cron'. The section on .NET looks at the caching options offered by this technology. The section on .NET takes a look at the various caching mechanisms available in ASP.NET. Caching in SQL Server stored procedures is also investigated.

6.2. What is caching?

With modern web applications much content is dynamic, and is pulled from a persistent data store such as a database. While this adds manageability, flexibility and functionality to an application, it does have performance implications. Dynamic pages utilizing server-side technologies like ASP.NET or PHP typically have longer response times than simple HTML pages because a page is required to repeatedly fetch and parse data from the persistent data tier as well as to perform business logic. This results in extra processing as queries are run against the database, business logic carried out, results returned and in turn rendered to HTML. For data which changes infrequently, all this extra processing is unnecessary. Output caching provides a mechanism to store the response either to memory or the hard-drive for optimized

retrieval, thus by-passing unnecessary processing and improving the raw response time of the page.

There are two typical mechanisms for caching page output. The first, which is the type predominately used by ASP.NET, is memory caching. Memory caching refers to the process whereby output is stored in memory for quick retrieval at a later stage. File-based caching uses the file system to store output as text on the hard-disk for quick retrieval. The intricacies of memory-based caching are discussed in the section on .NET with specific relevance to the `Cache` object, while file-based caching is discussed in the section on LAMP.

6.3. Optimization with LAMP

The LAMP system offers support for both memory-based output caching and file-based output caching. This section gives a brief overview of memory-based caching options, but concentrates on the file-based memory-caching options made available by the PEAR `Cache_Lite`.

The PEAR class, `Cache_Lite` provides a user-friendly interface to implement file-based output caching. `Cache_Lite` also supports memory caching if the `memoryCaching` parameter is set to true - although this is not the default. With this option, every time a file is cached it is stored in an array within the `Cache_Lite` object; this memory cache can be accessed through the `saveMemoryCachingState` and `getMemoryCachingState` methods of `Cache_Lite`. The advantage of such memory caching is that the complete set of cache files can be stored in a single file, thus decreasing disk read/writes by reconstructing the cache files into an array to which code has access. Memory-based caching is best suited to larger sites [Fuecks, H^2., 2004]. The following section, ‘6.3.1. Using `Cache_Lite` for Chunked, File-Based Output Caching’, illustrates and analyzes the process of using file-based caching with `Cache_Lite`.

6.3.1. Using `Cache_Lite` for Chunked, File-Based Output Caching

Cache_Lite provides a mechanism for saving the HTML output from a response to a file on the storage system. The following code illustrates how caching may be performed using Cache_Lite:

```
//include Cache_Lite
require_once('Cache/Lite/Output.php');
//set the parameters for the cache object
$options = array(
                        'cacheDir' => './.cache/',//directory where responses are stored
                         'fileNameProtection' => false,//do not md5 hash name
                        'writeControl' => TRUE,//checks file is successfully written
                        'readControl' => TRUE,//checks files being read for corruption
                        'readControlType'  =>  'strlen'//how  files  are  checked  for
corruption
                        );
//instantiate cache object
$cache = new Cache_Lite_Output($options);
echo time(); //output the uncached time
$lifetime = 60;
//set the lifetime of the cache to last 2 minutes before updating the cache
$cache->setLifeTime($lifetime);
//start caching a chunk
if(!$cache->start('content','Static')){

        echo time(); //cached time value
         /* perform logic and render HTML for caching */
        //stop caching this block
        $cache->end();
}
```

The first time the above code is called, it will write a chunk of output code, generated from code starting at the `$cache->start('content','Static')` command and ending at the `$cache->end()` command, to the file system. This code is stored in the specified `cacheDir` directory (.cache), and will be named '`cache_Static_content`' as specified by the arguments passed to `$cache->start`. Thereafter, every time it is called within its specified lifetime, in this case every two minutes, the cached output file will be served to the browser instead of the code physically generating the output as would normally be the case. Once the page is called after the lifetime has expired, the "`cache_Static_content`" file is regenerated with fresh content.

Output caching is generally accepted as one of the best means of improving the performance of a page. To test the effectiveness of `Cache_Lite`, and file-based caching, tests where run against a page with considerable overhead. The page in question runs a query against the MySQL database. 1000 rows are returned and these are output to the browser in a table.

Two version of this page were created, one which supports caching, and another which doesn't. Profiling the pages using the Zend Profiler revealed some interesting results as shown below, Figures, 17, 18, and 19. The figures illustrate a breakdown of the core time-distributions associated to files loaded when a single url is called. This tool is useful for establishing bottlenecks, and understanding the execution process of a PHP script which includes a number of other scripts. The red portion of the pie illustrates the file with the longest execution time; blue is second longest and finally green is the file with the lowest proportion of execution time.

Figure 18 illustrates the process of serving a cached file to the browser. As this figure illustrates, execution time is dominated by the `Output.php` file which is responsible for fetching and displaying the cached output file. Only a small proportion of time is dedicated to the main `MySQLi_withCache.php` file because the code normally executed within this file has been cached, and thus is not executed. Figure 19 illustrates a page where the cache is being renewed; you can see here that a far greater proportion of processing time is dedicated to `MySQLi_withCache.php`. This is because the logic contained in this file needs to be executed so that the output can be created and cached. Notice the impressive difference in execution time between the major component in Figure 19 (`MySQLi_withCache.php`, executing in 247.98 ms), and the major component in Figure 18 (`Outout.php`, executing in 6.52 ms). Finally Figure 20 illustrates a page with no output caching. This page reported the longest response time.

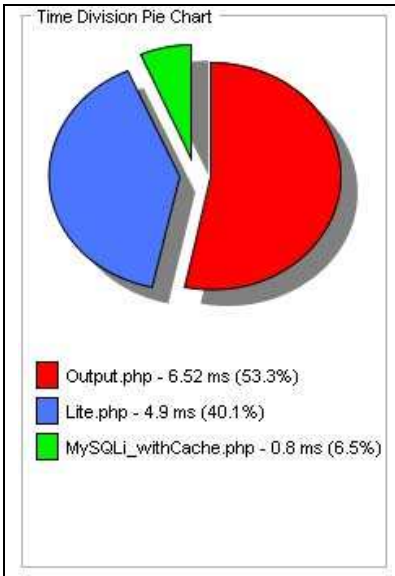

Figure 18 – A cached page

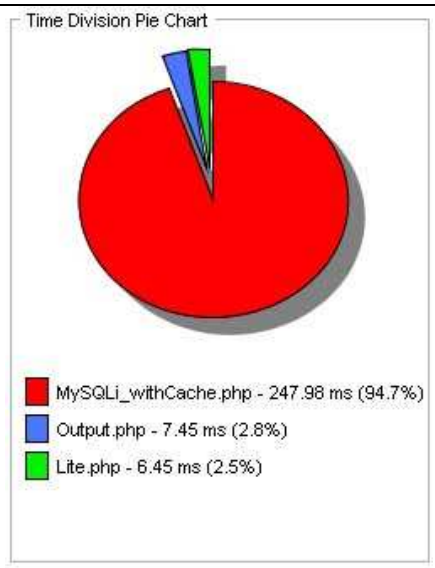

Figure 19 – Creating the cache

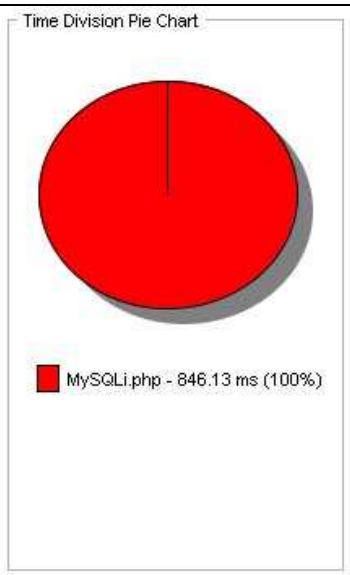

Figure 20 - A page with no output caching

While the above illustrations provide useful information as to how the cache object works, response times are the results of a single request, and do not provide a suitable illustration of the overall improvement in response time. To establish this, the average response time of 100 requests was recorded. For two pages, one with caching, one without. The results are illustrated in Table 11. This test shows a 39.67% improvement in response time from the cached page.

Table 11 - Average response times for a cached and un-cached page

Average response time over 100 web requests	
MySQLi_withCache.php (cached)	146 ms
MySQLi.php (not cached)	242 ms

While the above example serves to illustrate the advantages of output caching, there are a number of considerations which need to be understood when using output caching. Firstly, it is important to bear in mind that there is some overhead incurred in using the cache object because the object has to write the files to the file directory. Thus, while in the example above caching provides for great speedup, this may not always be the case, in particular if caching is used too frequently and for short lifetimes as in the following example:

```
$cache->setLifeTime(60);

if(!$cache->start('content','Static')){

        echo time(); //cached time value
        //stop caching this block
        $cache->end();
}
```

A breakdown of this page's execution is illustrated in Figure 21. (Figure 21in this example is equivalent to Figure 19 in the previous example). This image represents a situation in which a page is being cached for the first time. As is illustrated by the example, the page `Output.php`, responsible for creating the cached text-file actually requires more execution time than the `BadCache.php` file itself. Thus caching actually increases the average response time for the document.

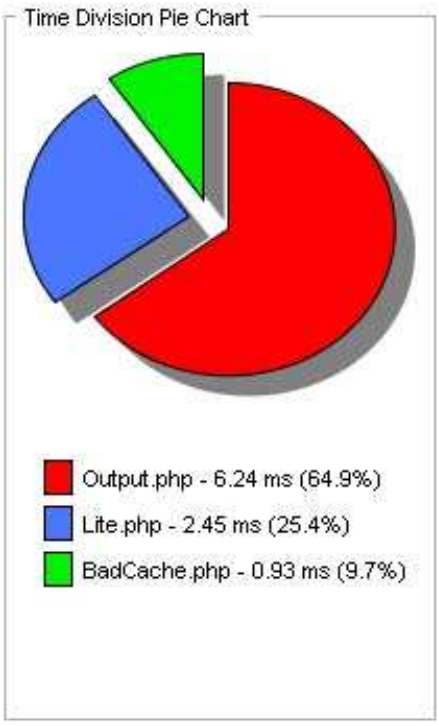

Figure 21 - A bad use of cache

Another consideration when using caching for dynamic pages is that dynamic pages often vary according to parameters passed to them. For example, in the application, the gallery page will display images in a gallery dynamically selected according to the gallery ID and page number passed to it in the querystring. Using a simplistic example of caching as above, this would be problematic, since the gallery page would be cached, and no new pages would be loaded until the time-limit exceeds.

For example, with caching such as illustrated above, the url:

http://localhost:3000/38_2005_ii/public_html/ga

llery.php?g=**278**

would be synonymous to the another url such as http://localhost:3000/38_2005_ii/public_html/gallery.php?g=**231**, because both would load the same cached file.

This is easily be averted by naming the cache files appropriately, for example:

```
$cache->start('gallery_'.$_GET[‘g’],'Static')
```

Finally, when using text-file caching as with `Cache_Lite`, there are some security implications which must be taken into account. The example above reveals a potentially insecure use of output caching. Since output is stored in plain-text, care needs to be taken so as to store cached output safely. In particular, personal details must be cached safely. Typically, the cache folder should be stored above the web-root on the server, and thus not be accessible directly from the internet.

6.3.2. Caching with ASP.NET

ASP.NET offers caching at a page level, using page directives, and with the `Cache` object. Page directive caching enables blanket caching of an entire page, or fragment caching of specific user controls within a page. The ASP.NET `Cache` object is similar to PHP's `Cache_Lite` used in the examples above. `Cache` allows for application level caching. Although using the `Cache` object is slightly more complex, it provides the developer with improved flexibility and control. A new feature offered in .NET 2.0 is SQL cache invalidation. This option caches data pulled from a table in a SQL Server 2005 database until changes are made to the table, at which stage the cache is renewed. SQL cache invalidation is a simplification and improvement on SQL cache dependency which was available in older versions of the .NET framework.

The simplest form of caching in .NET is by use of the `@OutputCache` page directive. Placing:

```
<%@ OutputCache Duration="120" VaryByParam="None" %>
```

This will cache the page for a duration of 120 seconds. This example was used to test caching performance improvements under the .NET framework. A file similar to that in the previous example was run. Results from the test are listed in Table 13 –

"Caching with .NET". The pages, `DataSetCache` and `DataSetNoCache` are similar to the pages used in the LAMP example above. The .NET pages collect 1000 rows from the SQL Server 2005 database using a DataSet, and render these pages to the browser. As illustrated in Table 13, an impressive 45.31% improvement in performance is achieved through using memory caching with .NET. However, the size of the file created by Visual Studio was a massive 2.44 MB by comparison to the skimpy 424KB file-size served by LAMP. The reason for the bloated filesize in the .NET page was two-fold, firstly, the view-state feature (a state management feature unique to ASP.NET which maintains state by passing variables between pages using an encrypted hidden form field), and bloated HTML code.

Setting the page level attribute `EnableViewState` to false enabled us to remove the hidden `viewstate` field and so decrease the file-size to 1.9 MB. Setting `EnableViewState` to false does not completely remove the `viewstate` hidden field, although it considerably shortens it.

From the HTML code generated by Visual Studio it is evident that there is substantial bloat added to the HTML code by the ASP.NET templating engine. For example, a label control in Visual Studio will be converted to the following HTML code by the templating engine:

```
<span id="DataList1_ctl00_img_nameLabel">IM000795.jpg</span>
```

The only information that the user needs, and indeed which is relevent in the HTML is the text contained within the tag. The tag itself is completely irrelevent in normal HTML. However, the Visual Studio templating engine requires that all ASP.NET controls be form elements – this is how the templating engine and server-side code are able interact to achieve a seemingly event-driven programming environment on top of the stateless HTML protocol. For a label to be accessible using code ,such as `Label1.Text = "Some Text"` , in the code-behind file, it must be served in HTML as a valid form element (such as the tag above). The ASP.NET templating engine facilitates the conversion from ASP control to rendered

HTML (Listing 12) while the CLR (Common Language Runtime), manages the application code behind our backs.

```
<asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
To
<span id="Label1 ">Label</span>
```

listing 12 - How the ASP.NET templating engine converts an ASP:label control to HTML

However, even with the bloated HTML removed the .NET pages were still almost twice the size of the LAMP equivalent.

To improve the performance of the page we removed the labels from the DataList control and instead just left the plain HTML text. This further decreased the filesize to 811KB, and response times impoved from 642 to 269 ms and from 1117 to 493 ms for the cached and non-cache pages respectively.

Table 13 - Caching with .NET

Average response time over 100 web requests			
Name of File	Response Time	File size	Relative improvement from caching
DataSetCache.aspx (cached)	823 ms	2.44MB	45.31% improvement
DataSetNoCache.aspx (not cached)	1505 ms		
DataSetCache.aspx (cached)(no viewstate)	642 ms	1.90MB	42% improvement
DataSetNoCache.aspx (not cached)(no viewstate)	1117 ms		
DataSetCacheStripped.aspx (cached)(no View State)(Stripped of bloated HTML)	269 ms	811KB	45% improvement
DataSetNoCacheStripped.aspx (not cached)(no View State)(Stripped of bloated HTML)	493 ms		

The caching mechanism in .NET can handle the problem described in the section on caching with LAMP, where different variables are passed in the URL, as in http://localhost:3000/38_2005_ii/public_html/gallery.php?g=**278** and http://localhost:3000/38_2005_ii/public_html/gallery.php?g=**231** by setting the `VaryByParam` attribute to true. This ensures that pages are refreshed when the specified parameters are passed to the page using either GET or POST. The `OutputCache` directive also supports attributes such as `VaryByControl`, `VaryByCustom` and `VaryByHeader`. These help the developer to produce cached pages which are sensitive to events, and to state.

For example, consider a web form which accepts variables via POST. By setting `VaryByParam` to the name of the POST variable we will be accepting we can ensure that the various different responses to the form are successfully cached.

The `OutputCache` directive can also be applied to user controls within a Web Form. This is called fragment caching [Webb, 2003.], and is similar to the chunked caching method explained in the LAMP section of this chapter. By specifying `PartialCaching` in the definition of a web control, caching can be applied to individual user controls throughout an application. Chunked caching, or fragment caching, can also be applied at the page level by setting the `VaryByControl` parameter to the ID of a specific control on the page to cache various different versions of a page control [Webb. J. 2003].

Caching is also available through the `Cache` object of the `System.Web.Caching` namespace. This offers improved flexibility because caching can be performed explicity in code as opposed to the blanket manner in which it is applied using page directives. With the `Cache` object application logic as well as simple output responses can be cached. Data can be added to the cache object using the `Add` and `Insert` methods, or on assignment. As mentioned earlier, use of the `Cache` object is similar to that of `Cache_Lite` as Listing 14 illustrates.

```
if(Cache[“NewItem”] == null){
      Label1.Text = “No Cache, lest add some”;
      Cache.Add(“NewItem”,”Some String data”);
}else{
      Label1.Text = Cache[“NewItem”].ToString();
}
```
[MSDN. 2005]

Listing 14 - Caching with the .NET Cache object

Finally, ASP.NET 2 offers “SQL Server 2005 notification based cache invalidation” [MSDN, 2005]. This is an impressive and useful feature which “uses the query change notification mechanism of SQL Server 2005 to detect changes to the results of queries” [MSDN. 2005]. When a command is executed, ASP.NET and ADO.NET automatically create a cache dependency. This listens to change notifications sent from SQL Server. When data is changed in the SQL server, the web server is informed and invalidates the appropriate caches [MSDN. 2005].

Notification-based invalidation can be applied using the `SqlDependency="CommandNotification",` attribute of the `OutputCache` page directive. Notification can also easily be added to a databound control using the code in Listing 15.

```
<asp:SqlDataSource EnableCaching="true"
SqlCacheDependency="CommandNotification" CacheDuration="Infinite" ...
/>
```

Listing 15 - Applying Sql Server 2005 Notification-based Cache Invalidation to a SQLDataSource

This is one of the more useful and practical caching features available. It is extremely useful for caching objects such as look-up tables which draw rarely changing data, such as lists of categories, from the database.

6.4. Conclusions

Both LAMP and .NET offer effective mechanisms for caching. Both platforms also offer the option to cache to disk or to memory, although in LAMP disk caching is the default while in .NET memory caching is the default.

Both platforms offer impressive responses to caching, although ASP.NET produced a slightly greater improvement at 45.31% over the 39.67% improvement experienced by LAMP. However, it must be noted that the size of the data cache by ASP.NET was considerably larger, and therefore, the advantages to caching would be expected to be greater as more data is retrieved directly from the cache.

The .NET caching mechanism offers useful tools to enable the developer to easily manage most of the intricacies of data caching such as dynamic pages with varying responses according to the variables passed to them. Features such as these must be handled in code by the `Cache_Lite` object in LAMP. Particularly important is the tight integration between SQL Server 2005 and ASP.NET which enables SQL notification caching. This is an impressive and unique feature to ASP.NET. The LAMP platform does not have the tight integration necessary to enable lower level synchronisation to the same extent. Although similar capabilities would be possible using the LAMP system, these would be complex to implement, would rely on some kind of polling mechanism using cron and would be unlikely to be as efficient a solution to the problem.

Although optimization through caching for ASP.NET slightly outperforms the equivalent in LAMP in terms of both capability and improved performance, in terms of overall performance there is considerable bloat added to ASP.NET code through both the visual environment and the templating engine. It is important to note too that much of the bloat experienced in ASP.NET pages is necessary for the ASP.NET framework to operate as intended. ASP.NET controls, the cornerstone of ASP.NET web applications, need to add extra formatting to web pages so that they can function properly. Viewstate too adds a great deal of bloat to a web page, but is necessary for many ASP.NET applications as it is a useful and relatively secure means of maintaining state throughout a user's session. The example above is an excellent example of the bloated output created by the default configuration in Visual Studio. By simply removing viewstate and using text instead of the ASP label server control, the page-size was decreased from 2.44 MB to 811KB. Response times decreased from 823 to 269 ms on cached pages and 1505 to 493 ms on non-cached pages. Even the

optimized ASP.NET code was still considerably slower than comparative LAMP code.

Although LAMP code may not execute as fast as compiled ASP.NET code, its generated HTML is typically tighter. This leads to smaller HTML files and ultimately better response times. Extra functionality which is plumbed in to the ASP.NET Visual environment, such as viewstate and controls, although useful in producing high quality solutions, and powering the ASP.NET event-driven web programming environment, do incur a certain performance overhead due to added bloat to pages. Although often necessary in ASP.NET applications, unnecessary HTML code is often generated to enable these features, even when not in use.

Even though ASP.NET slightly out-performs LAMP in terms of caching capabilities, LAMP excels in creating neat and optimized HTML code compared to the bloated HTML code necessary to drive the ASP.NET model of server-side programming. As such, the average page created with LAMP is likely to be of a smaller file-size than its .NET counterpart, and thus we can expect faster response times.

Chapter 7 – Conclusion

7.1. Outcomes and Conclusion

The recent releases of PHP 5 and MySQL 5 within the LAMP platform signify a large push into the enterprise web development domain space by this framework. Similarly, the release of ASP.NET 2.0, the first major release to the .NET framework since its initial release in 2000, also represents major development to this platform. These new releases all signify a significant drive towards maturity of these already well established platforms.

ASP.NET 2.0, and the .NET framework 2.0 offer a complete and powerful environment for developing robust large-scale web applications quickly and easily. Visual tools, new controls and new features such as the login controls for role management, as well as masterpages, themes, over 40 new controls and efficient Object re-use allow for straightforward rapid development of robust applications with the .NET framework.

In addition to the new features mentioned above, mature and solid support for features such as Web Services and robust data access is also offered.

The .NET environment is scaled for top-end enterprise level development, .NET emphasises power and flexibility at small cost to performance due to bloated file-sizes - although compiled .NET files are optimized and run extremely efficiently. .NET is aimed at the high-end market and targets high-performance hardware, assuming that scalability and performance can be improved through hardware upgrades.

On the other hand, LAMP is still better suited to smaller applications; although it is capable of handling large-scale enterprise level problems. The LAMP framework affords the developer better flexibility, but does not have the power features of .NET, such as `DataSets` and ASP.NET controls. Furthermore, support for XML and Web Services is new and limited by comparison to the mature Web Service suite offered by .NET. The LAMP stack is a scalable and efficient, yet lightweight solution, which is capable of producing high quality web-based applications - but is better suited to

smaller projects because of the longer development time requirements of the framework.

The LAMP platform, hosted on the robust Apache web server is scalable and high-performance enough to serve large web applications under high-demand or load. Furthermore, the PHP scripting language and MySQL database server are capable of producing high-quality high-performance and secure applications equivalent to .NET applications. However, the development of such solutions with the LAMP framework is more difficult and likely to take longer.

In conclusion, .NET is better suited to larger-scale application development, however, LAMP is capable of creating comparable and even better solutions to large problems, and is well suited as an alternative to ASP.NET for large-scale development where cost or operating platform is an issue.

7.2. Future Work

7.2.1. Extension of the core application

This project developed the core functionality and architecture of a refactored version of the www.38.co.za website. Future development could extend the core application, using either LAMP or ASP.NET to encapsulate the full functionality of the existing application. This could be performed using either the .NET or LAMP environment and could facilitate a more in depth analysis of the specific framework.

7.2.2. Further analysis

ASP.NET 2.0 reflects a major release and much development on top of ASP.NET 1.x. Work done in this project to evaluate the ASP.NET 2.0 platform could be used to compare ASP.NET 2.0 with ASP.NET 1.x in terms of performance and capability. Similarly, analyses could be performed on PHP 4.x and PHP 5 or MySQL 4.x and MySQL 5.

References

Note: Most technologies being tested are relatively new. ASP.NET 2.0, SQL Server 2005, PHP 5 and MySQL 5 where only released out of the BETA testing phase during the course of the year. As a result, there is little literature published on the topics, and this accounts for the high proportion of web references below.

1. [Alur *et* al. 2003] Alur, D., Crupi, J., Malks, D. 2003. **Core J2EE Patterns: Data Access Object.** Prentice Hall/Sun Microsystems. Available at: http://corej2eepatterns.com/Patterns2ndEd/DataAccessObject.htm. Accessed: 9 August 2005
2. [Costello. 2005]. Costello, R.L. **Building Web Services the REST Way.** Available at: http://www.xfront.com/REST-Web-Services.html Accessed: 19 Oct. 2005.
3. [Ferrara, A., and MacDonald, M. 2002] Ferrara, A., and MacDonald, M. **Programming .NET Web Services.** O'Reilly Press, United States of America, 2002
4. [Fielding, R. T. 2000]. **Architectural Styles and the Design of Network-based Software Architectures.** Doctoral dissertation, University of California, Irvine, 2000
5. [Fleet Berry, D. 2005] Fleet Berry, D **15 Seconds: Writing a Custom Membership Provider fo the Login Control in ASP.NET 2.0.** 15 Seconds.com. Available at: http://www.15seconds.com/issue/050216.htm . Accessed 17 Aug 2005
6. [Frietag, P. 2005] **REST vs SOAP Web Services.** Available Online: http://www.petefreitag.com/item/431.cfm. Accessed: 19 Oct 2005.
7. [Fuecks, H[a]. 2004] **The PHP Anthology Volume I: Applications.** Sitepoint press.
8. [Fuecks, H[b]. 2004] **The PHP Anthology Volume II: Applications.** Sitepoint press.
9. [Greant, Z. and Richter, G. 2004] **Using ext/mysqli: Part I - Overview and Prepared Statements.** www.zend.com. Accessed: 2 Nov 2005 Available online: http://www.zend.com/php5/articles/php5-mysqli.php?article=php5-mysqli&kind=php5&id=4524&open=0&anc=0&view=1

10. [Gulutzan, P. 2005] **MySQL 5.0 Stored Procedures**. Accessed: 2 Noc 2005. Available online: http://dev.mysql.com/tech-resources/articles/mysql-storedprocedures.pdf
11. [He, H. 2004]. **Implementing REST Web Services: Best Practices and Guidelines.** XML.com. Available online: http://www.xml.com/lpt/a/2004/08/11/rest.html Accessed: 10 Oct. 2005.
12. [Hinchcliffe, D. 2005] **The Hidden Battle Between Web Services: REST and SOAP.** Available online: http://hinchcliffe.org/archive/2005/02/12/171.aspx. Accessed: 6 Nov 2005.
13. [Howard, M. and LeBlanc D. 2002] **Writing Secure Code.** 2 ed. Microsoft Press, Redmond, 2002
14. [Modsecurity. 2005]. **Open Source Application Firewall.** Thinking Stone. Available at: www.modsecurity.org Accessed: Mon 17 Oct. 2005.
15. [Moroney, L. 2005] **SOAP's Alive: Try the New Native SOAP Extensions for PHP.** DevX. Available online: http://www.devx.com/webdav/Article/22338. Accessed: 28 Aug 2005.
16. [MSDN. 2005] SQL Cache Invalidation. Available online: http://beta.asp.net/QUICKSTART/aspnet/doc/caching/SQLInvalidation.aspx. Accessed: 3 Nov 2005.
17. [Patterson, D. 2004] **Simplify Business Logic with PHP DataObjects**. Onlamp.com. Available at: http://www.onlamp.com/pub/a/php/2004/08/05/dataobjects.html Accessed: 9 Aug 2005
18. [Plourde, W. 2005] **Creating a Data Access Layer in .NET.** Available online: http://www.15seconds.com/issue/030317.htm. Accessed: 27 Oct. 05.
19. [Schlossnagle, G. 2005] **WSDL Generation.** George's Blog. Available Online: http://www.schlossnagle.org/~george/blog/index.php?/archives/234-WSDL-Generation.html Accessed: 28 Aug 2005.
20. [Shah, S. 2005] **Securing Web Services with mod_security.** Available online: http://www.onlamp.com/pub/a/onlamp/2005/06/09/wws_security.html. Accessed: Fri 14 Oct. 2005. www.ONLamp.com
21. [Shiflett, C. 2005[a]]. **Essential PHP Security.** O'Reilly Press. United States of America

22. [Shiflett, C. 2005[b]]. **PHP Security Guide.** ApacheCon. Available at: http://phpsec.org/php-security-guide.pdf. Accessed: 2 Nov 2005 .
23. [Stocker, C. 2005]. **XML in PHP 5 – What's New?** Available online: http://www.zend.com/php5/articles/php5-xmlphp.php. Accessed 25 October 2005. Zend
24. [Trachtenburg, A. 2005]**. PHP Web Services Without SOAP**. OnLamp.com. Available online: http://www.onlamp.com/pub/a/php/2003/10/30/amazon_rest.html accessed: 6 Nov 2005.
25. [Trenary, J. 2002] **Developing XML Web Services and Server Components with Microsoft Visual Basic .NET and Visual C#.NET.** Microsoft Corporation. Redmond, Washington, USA.
26. [Troels, A. 2005] **A Comparison of different SQL implementations.** Available at: http://troels,arvin.dk/db/rdbms/ Accessed 25 May 2005.

Appendix I – Screenshots

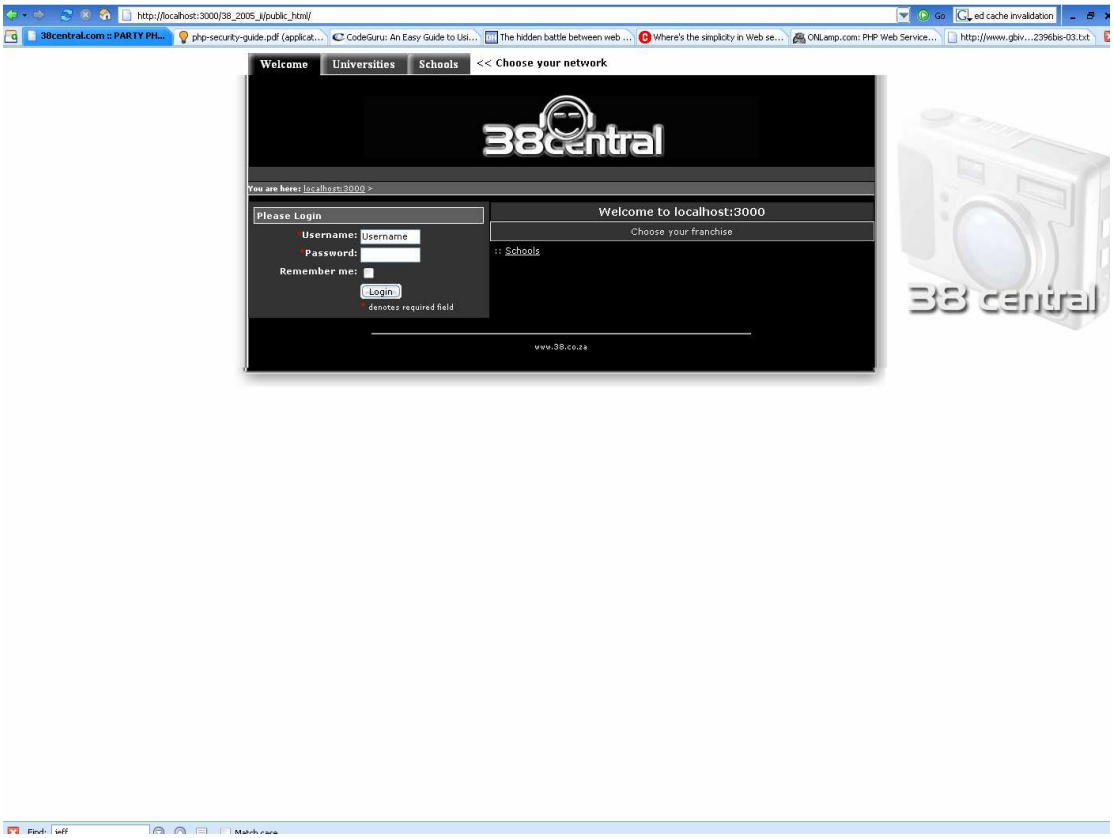

Figure 22 - LAMP Home page

Figure 23 – LAMP Franchise page

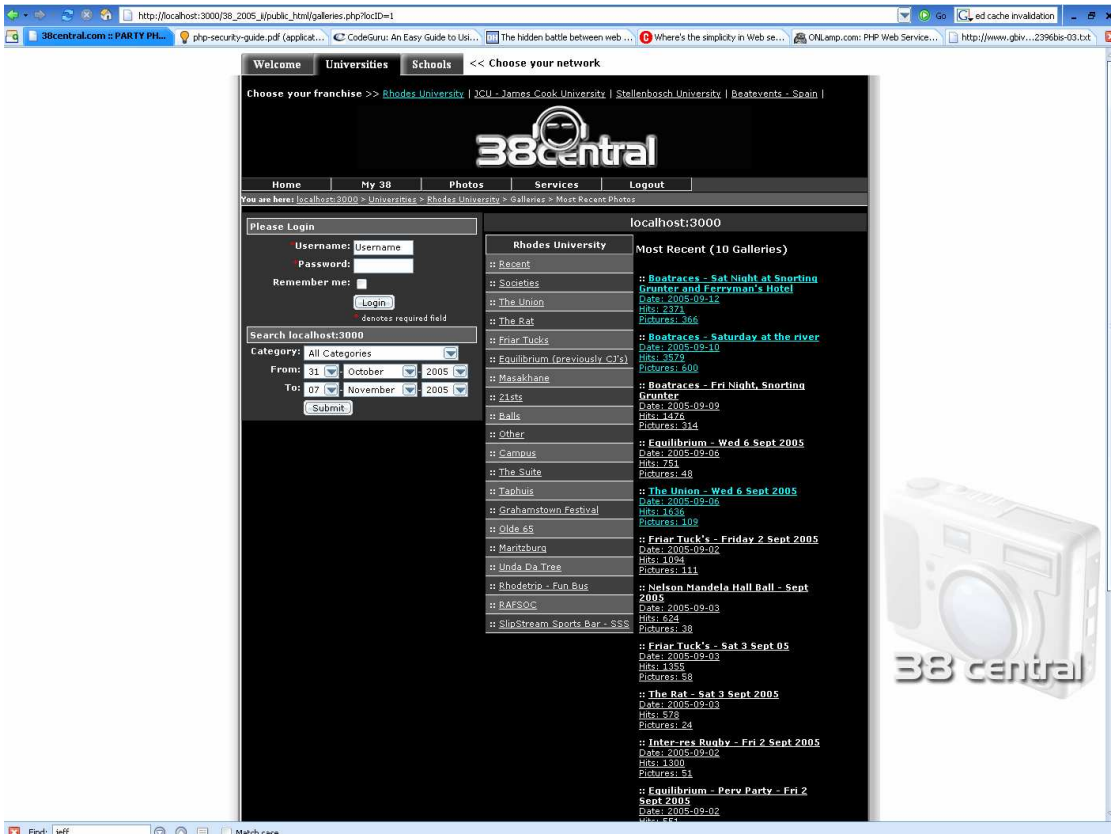

Figure 24 – LAMP Galleries page

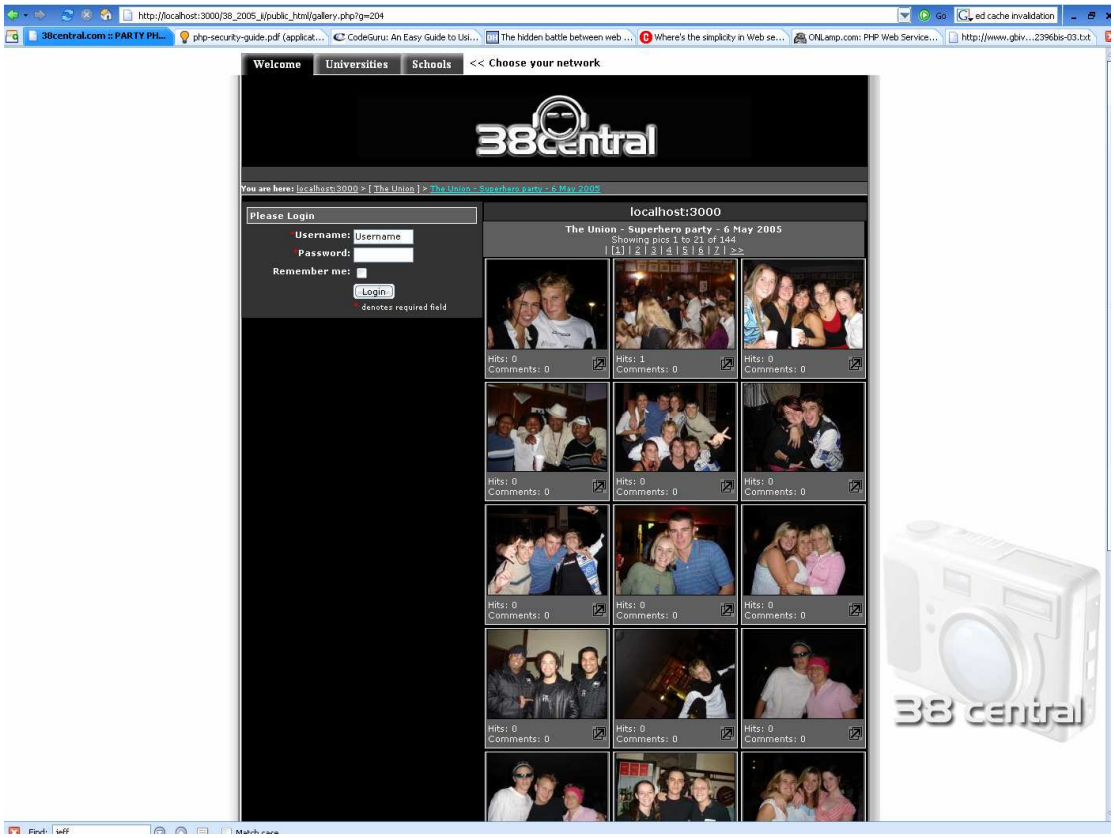

Figure 25 - LAMP Gallery page

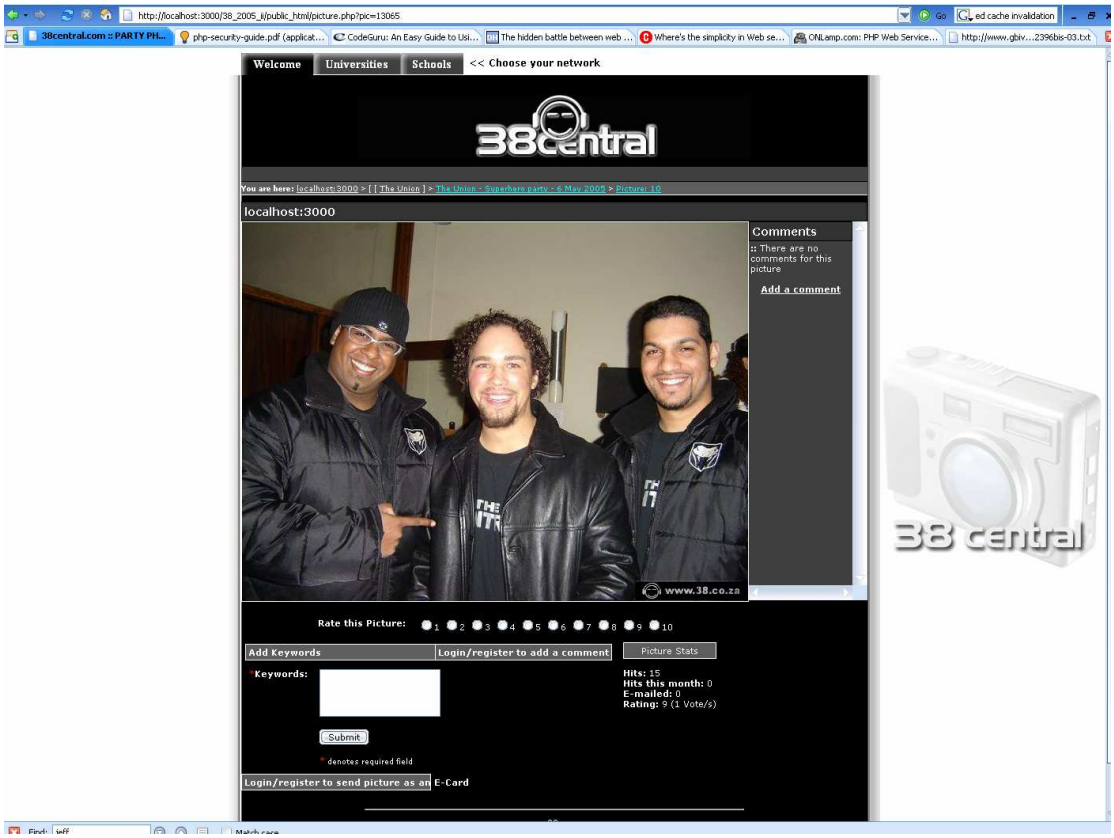

Figure 26 – LAMP Picture page

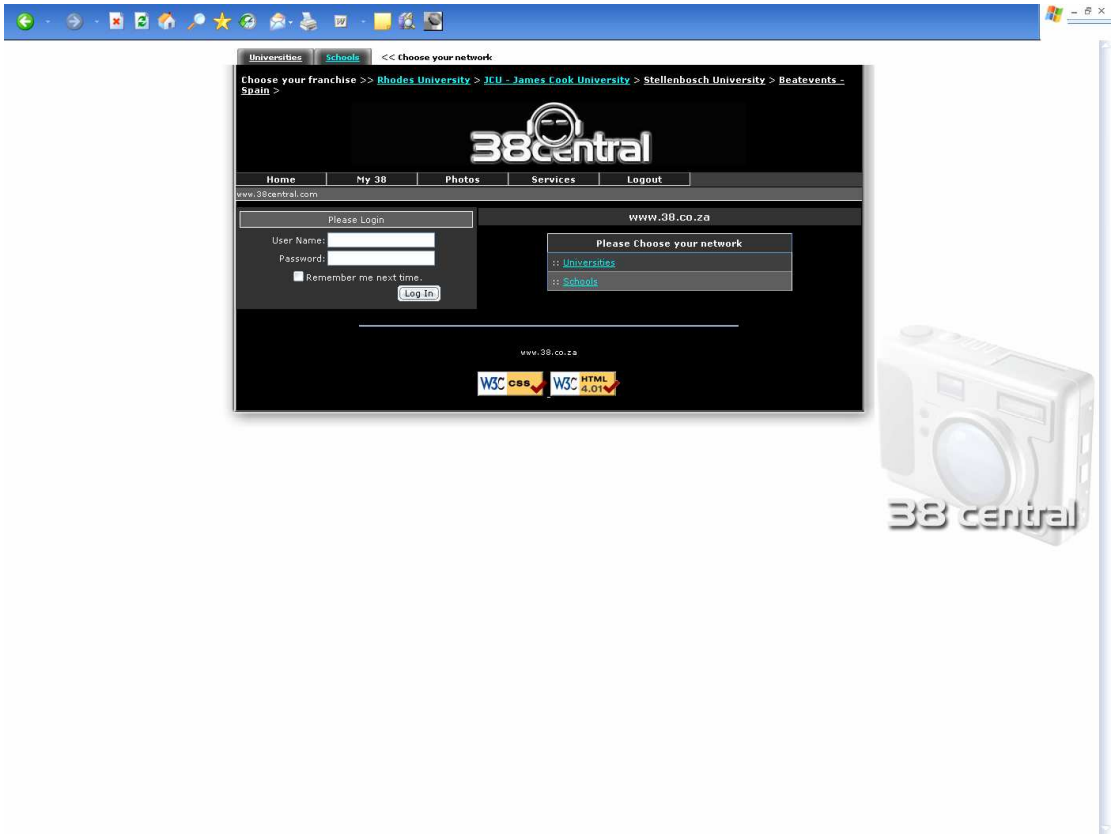

Figure 27 - .NET Home page

Figure 28 - .NET Franchise page

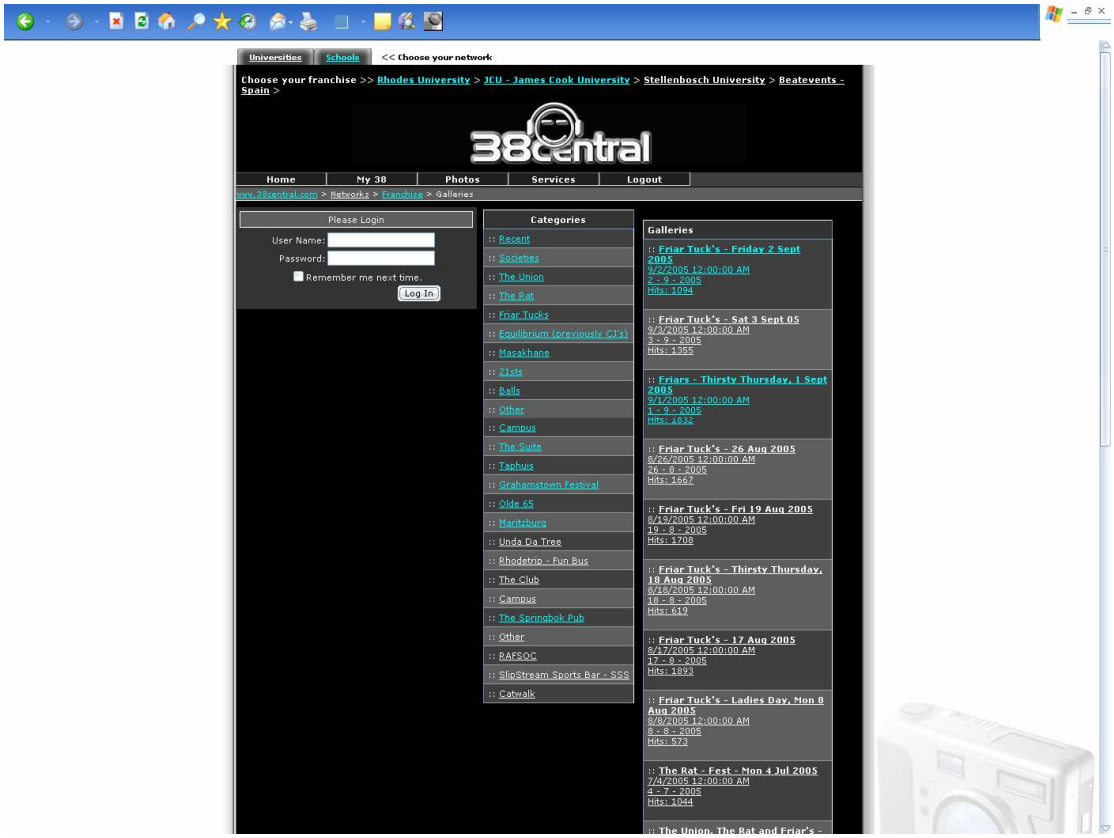

Figure 29 - .NET galleries page

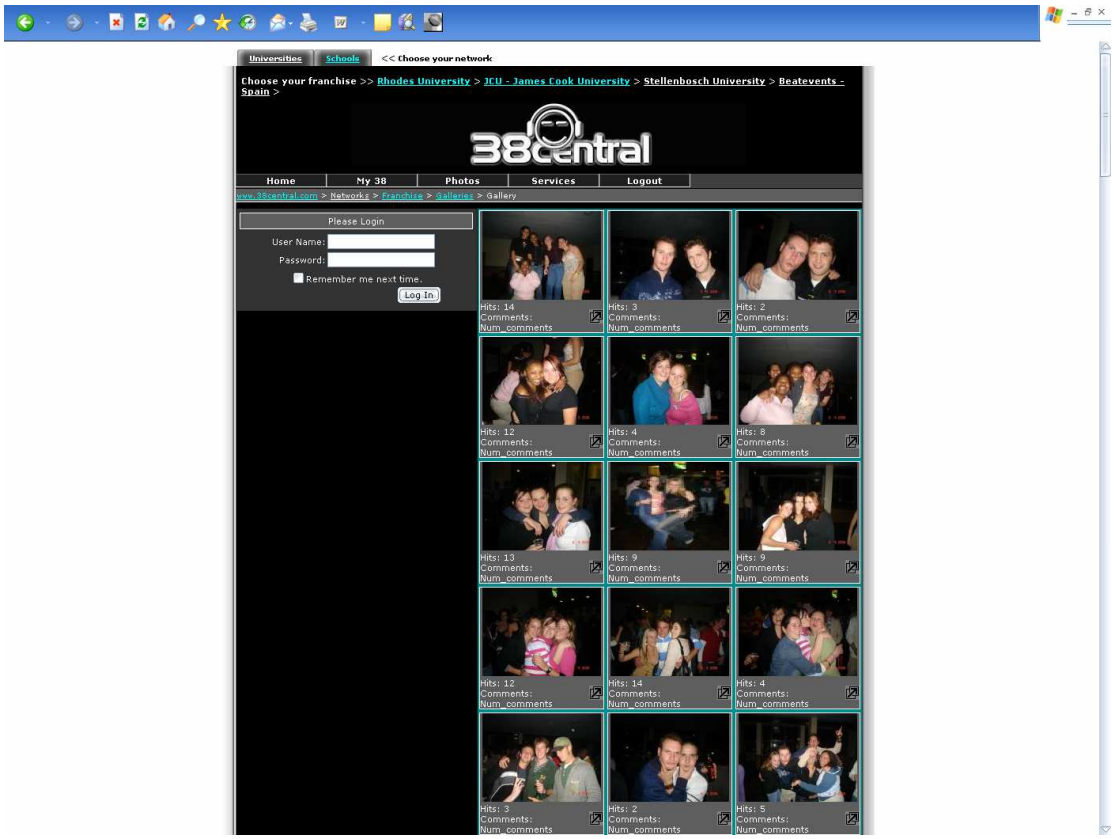

Figure 30 - Gallery page

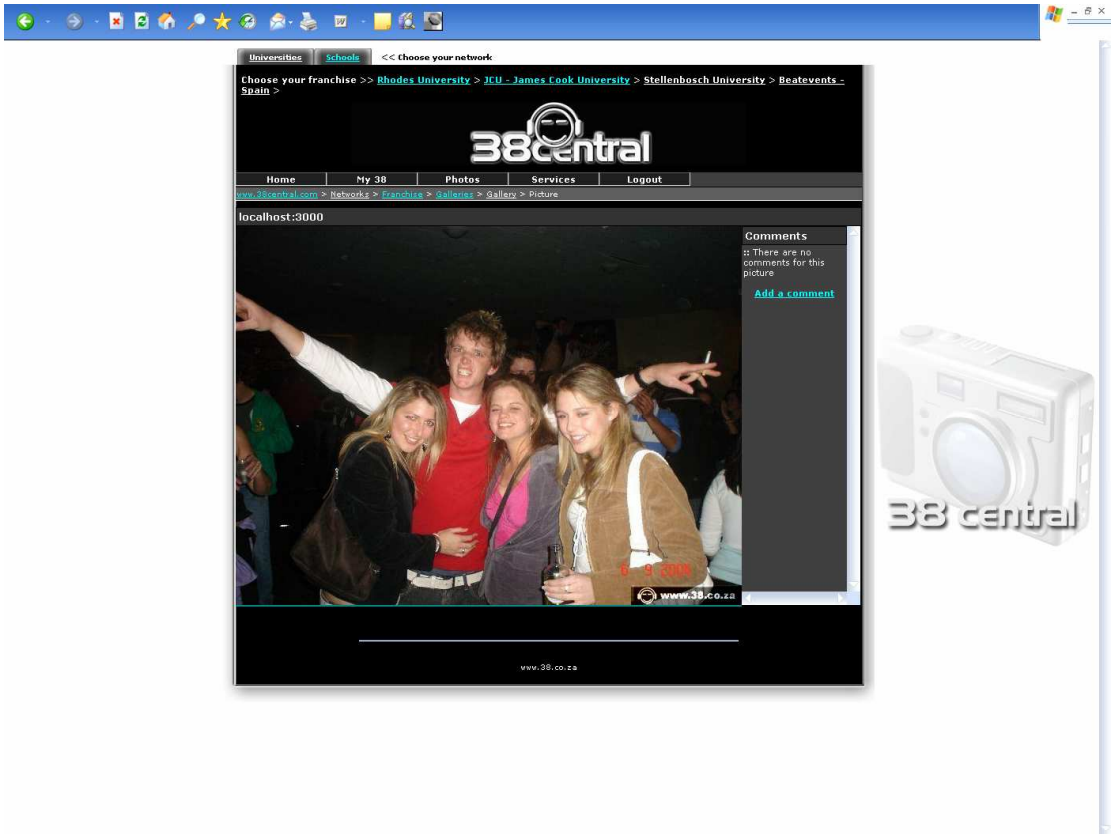

Figure 31 - Picture page

Appendix II – UML and Design

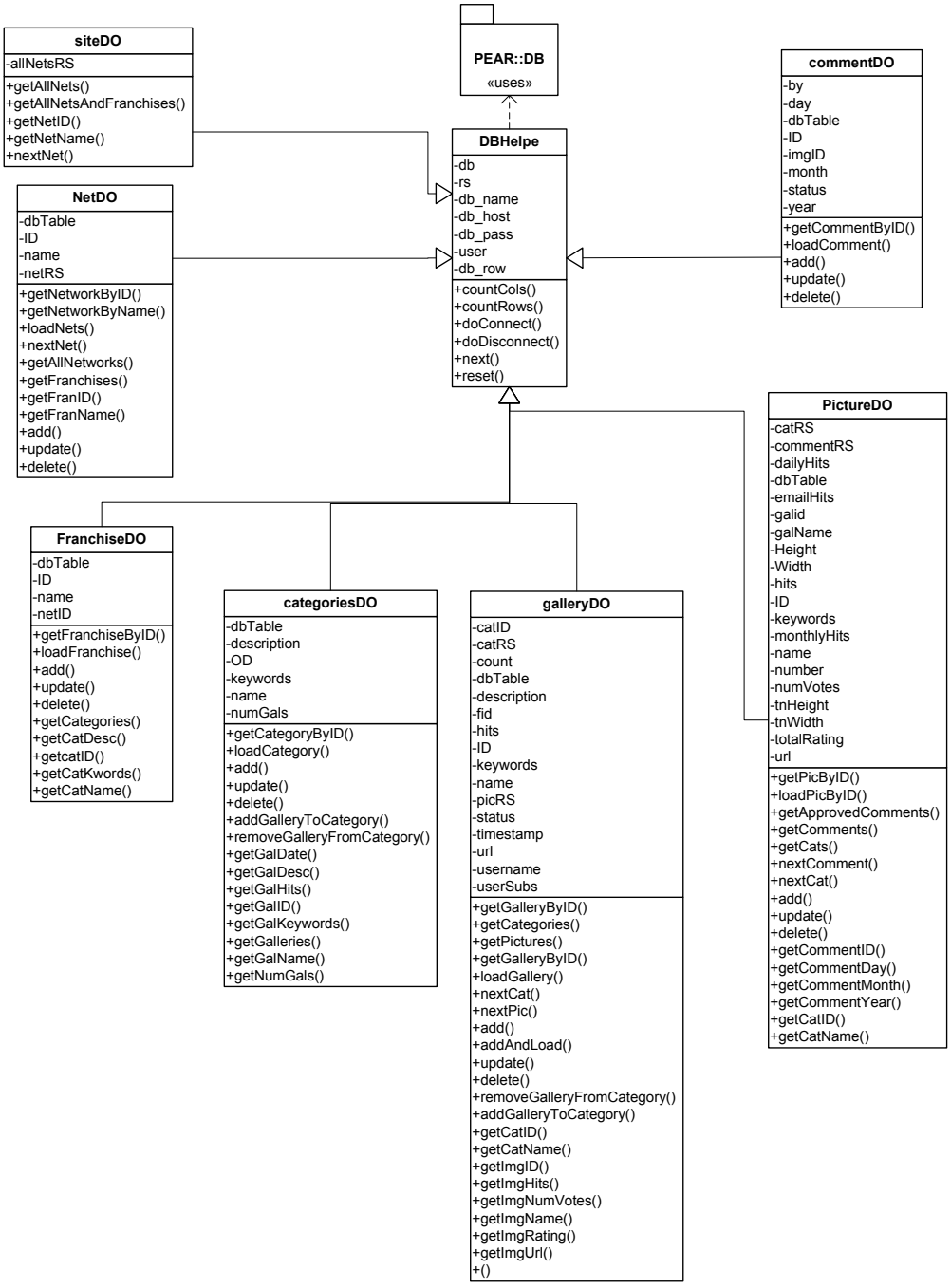

Figure 32 - LAMP Data Access Layer

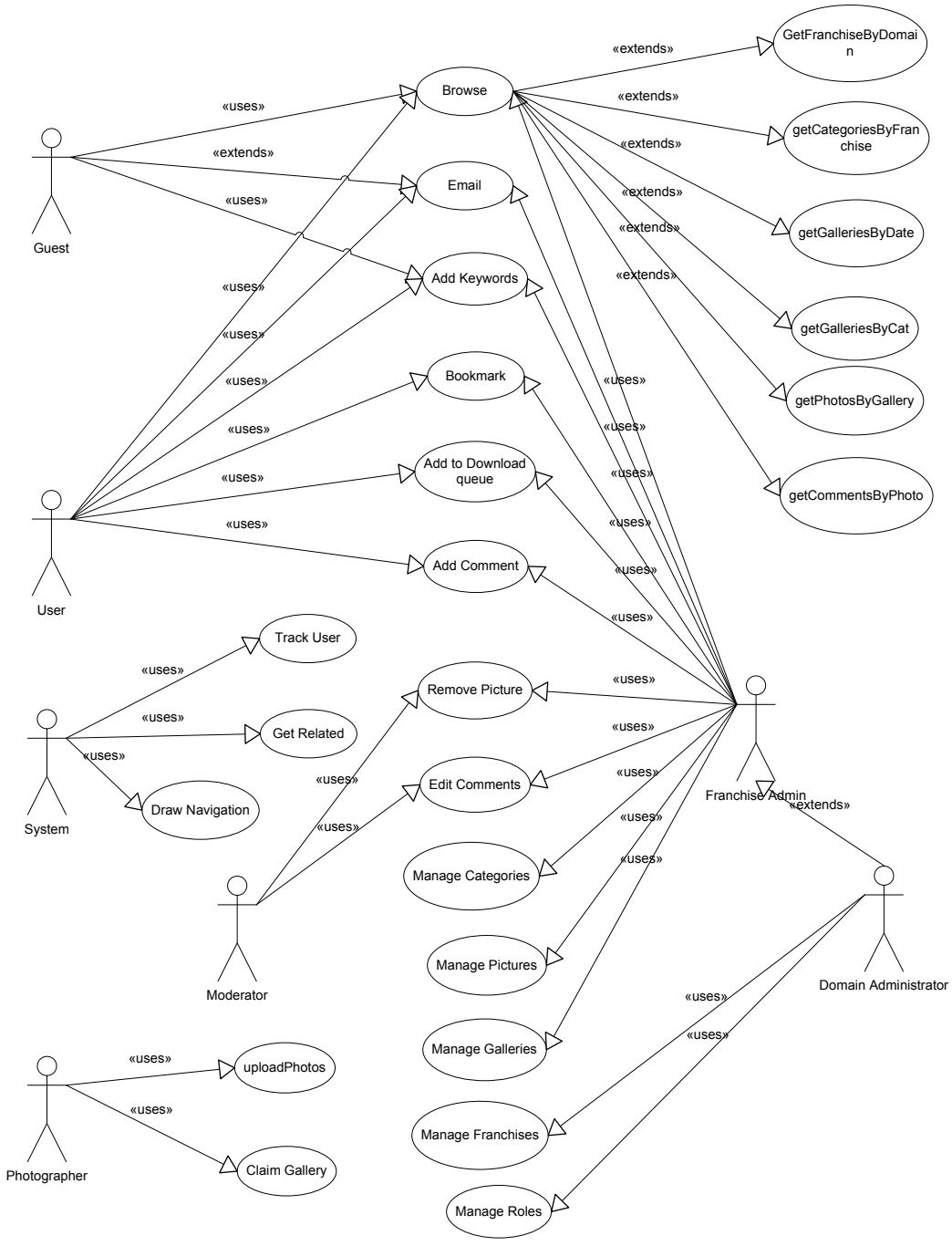

Figure 33 - Complete use case diagram

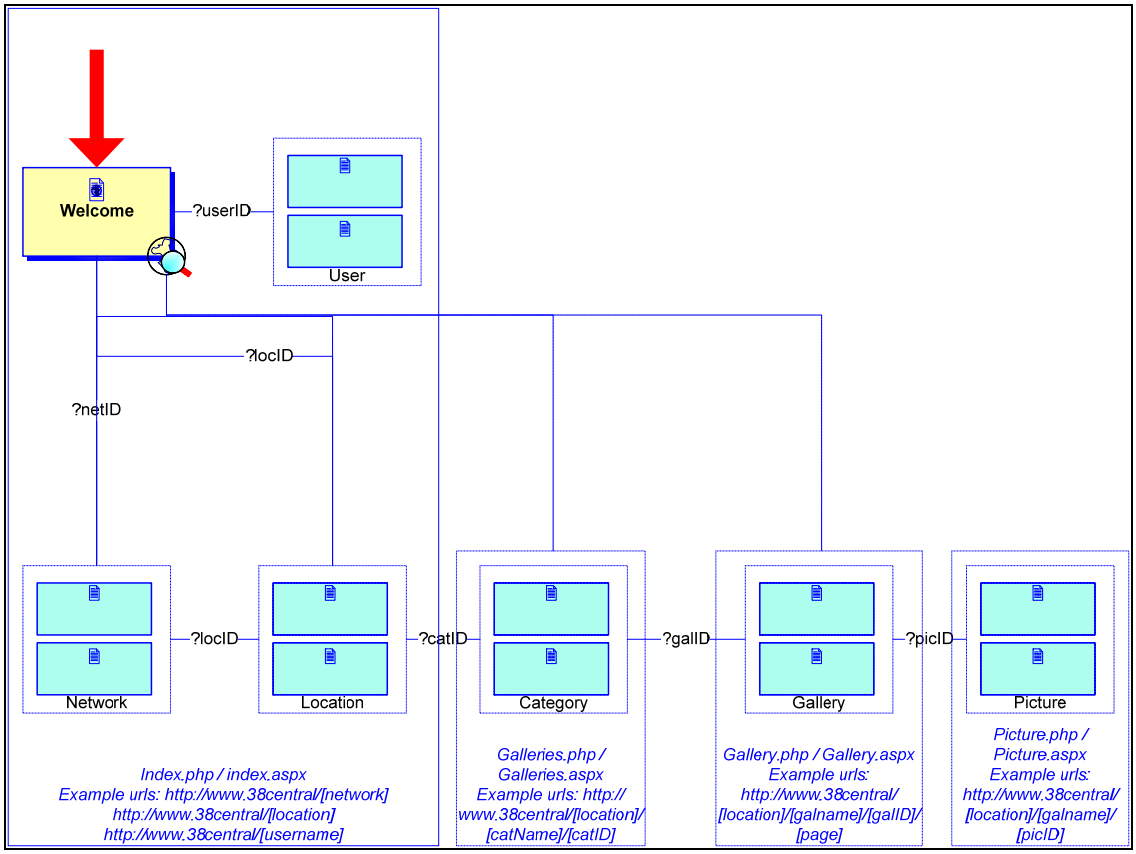

Figure 34 - Site Map