September 2017

Database for LnL

Matthew Brennan
Worcester Polytechnic Institute

Project Number. GFP0811

Database for LnL

A Major Qualifying Project Report

submitted to the Faculty

of the

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the
Degree of Bachelor of Science

By

Matthew Brennan

Date: September 5th, 2017

Approved:

Professor: George T. Heineman, Major Advisor

Abstract

This project originally sought to replace LnL's Perl web based database system with a more modern Java MVC Servlet solution. Complications in the original project resulted in this project morphing into an analysis of the differences in web based technologies between 1995, 2008, and 2017. This project analyzes the differences in technologies over a twenty-two year period, with a special emphasis on the programming languages used, processing models, and decisions about hosting locations.

Acknowledgements

I would like to give thanks to:

Professor Gary Pollice, for advising the original project;

the members of the 2008-2009 LnL Executive Board, for their participation in surveys and answering various questions during the design phase of the initial project;

Christopher Malone, CFO of Applause, for motivating me to complete this project;

Professor Carolina Ruiz, for providing help with getting this project back on track after a several year hiatus;

Christine Love, Associate Registrar, for helping me get re-enrolled as a student so I could complete this project;

Valerie Moore ‘18, LnL President, and Jake Merdich ’18, LnL Head Projectionist, for their assistance in comparing the system requirements today to those of 2008 and 2013;

Thomas Collins ‘01, Hosting Systems Program Manager, for his assistance in determining the viability of on-campus hosting for the 2017 project version;

and Professor George Heineman, for taking over advising the projector after Professor Pollice retired.

Table of Contents

Abstract i
Acknowledgements ii
Table of Contents iii
Table of Figures v
Introduction 1
Methodology (2008 Project) 4
Requirements Analysis 4
Design Decisions 5
Data Storage Options 10
Application Hosting Options 11
Impact of Hosting Decision 13
Implementation 15
Methodology (2017 Project) 18
Requirements Analysis 18
Design Decisions 18
Data Storage Options 23
Application Hosting Options 25
Implementation 28
WPI Hosted Virtual Machine 29
Amazon Web Services 31
Comparison 36
Currently Running System 36
User Experience 38
Chosen Technology 39
Mobile Access 40
Hosting Options 42
Glossary 44
References 49
Appendix A: List of Use Cases 51
Appendix B: Example Use Cases 52

End-user Login 52
Add an Event 53
Edit Event Details (Requestor) 55
Edit Event Details (LnL Staff) 57

Table of Figures

Figure 1: Perl Application Architecture Diagram (Marr's System) 2
Figure 2: Main Menu of Marr's 1995 database 3
Figure 3: Ruby on Rails Architecture Diagram 8
Figure 4: Java Servlet / MVC Architecture Diagram 10
Figure 5: Virtual Machine Diagram (2008 Project Plan) 15
Figure 6: AWS Hosting Architecture 25

Introduction

Lens and Lights (“LnL”) is a student organization at Worcester Polytechnic Institute (“WPI”) that provides productions services to the WPI community. The services that LnL provides can be as small as setting up a single microphone on an existing sound system, up through large scale concert lighting and sound. LnL also maintains and operates the on campus movie theater in Fuller Labs, including a projectionist-in-training (“PIT”) program.

LnL functions with a few classes of membership. The organization is governed by an Executive Board consisting of a President, Vice President, Technical Director, Head Projectionist, Treasurer, Secretary, and Webmaster. Active Members are full members of the organization and may crew chief events, vote in matters before the organization, and run for office on the Executive Board. Associate Members are any members of the WPI community who express interest in joining the organization. Lastly, Alumni Members are members who were active when they departed the university; they retain all privileges granted to Active Members, except they cannot run for office or vote.

In the early 1990s, the amount of information that was being tracked on paper was becoming too cumbersome. In 1994, Greg Marr ’95 was tasked with developing LnL’s first online web system for tracking this information.

The web system that Marr created was written in Perl using a MySQL back end. The Perl code used data from MySQL to generate HTML code and render as web pages. At this time of its creation, this design was state of the art. Marr’s web system became colloquially known as “the database.”

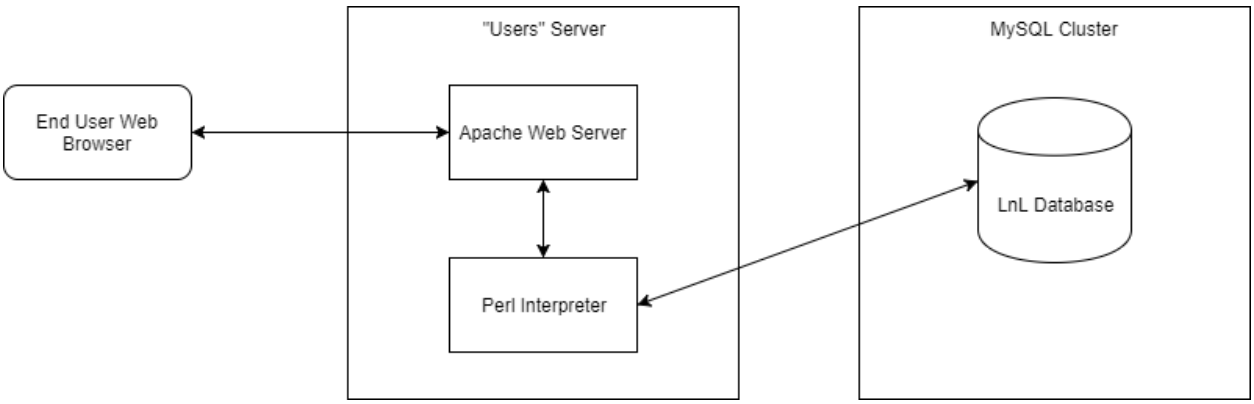

Figure 1: Perl Application Architecture Diagram (Marr's System)

The application started by allowing users to submit a request for services via LnL's website. The form would ask basic questions such as the type of service, location of the event, when LnL could start setup, when LnL had to be setup by, when the event started, when the event ended, who to contact with questions and, lastly, who to bill. The web system would store this information in MySQL and then push a notification to the LnL Vice President ("VP") for their review.

In addition to event specific details, Marr's database also tracked other important aspects of the day-to-day operations. This included tracking active and associate members, hours worked by members, reports from crew chiefs, billing for events, members' PIT progression, and equipment inventory.

Figure 2: Main Menu of Marr's 1995 database

Over the 10 years that followed, Marr's web system started to break down. Functions in the Perl modules that were used were deprecated and, eventually, removed. In some cases, even the modules themselves were deprecated and the Computer and Communications Center ("CCC") removed them. A combination of poor documentation on the original system, and the fact that Perl itself was no longer widely used at WPI, resulted in a lack of people who were qualified to maintain the system. The CCC would continue to work with the LnL Webmaster and other staff to make minor modifications, but the system was constantly in a degraded state.

In 2008, the problems became extremely widespread. Users were receiving exceptions while trying to submit work orders. Executive Board members were unable to get information out of the system, and were sometimes writing paper work orders from email notifications. Data that was entered was sometimes not being recorded. To help support LnL, the CCC was installing older versions of Perl modules which were statically referenced in the system's code – though the maintainers often missed references, resulting in exceptions nested deeper in the code. It became clear that the system needed to be replaced.

Methodology (2008 Project)

Requirements Analysis

The first step in the original project was to perform an analysis of the issues with Marr's system. While the main issue was discontinued Perl modules inhibiting basic functionality, Marr's system had other significant shortfalls. The way in which LnL conducts its day-to-day operations had changed, and this would be the best opportunity to add enhancements matching the current business model.

In order to determine the requirements, in 2008 I surveyed the current members of the LnL Executive Board, as well as Executive Board members from prior years who were still available. The survey asked questions to determine the individual's opinion of the current system and, most notably, what features they felt were necessary and lacking. The responses from that survey were the building blocks for a new design in terms of functionality.

One of the key issues identified by the survey was that Marr's application contained only one role: officer. The only people intended to use the web system were Executive Board members ("Execs"). This means that all information going into or coming out of the system had to be done by an Exec. Paper forms ("work orders") were handed out to Crew Chiefs at meetings. After each event, the work orders were turned back in with the crew list, hours, and Crew Chief report hand written on the form. The VP was then responsible for entering that information into the system for tracking. While this may have made sense in 1994, now it was a hindrance. Crew Chiefs wanted to be able to access information about their events on their smart phones. Members wanted to be able to look up setup times for upcoming events. The VP wanted Crew Chiefs to enter their hours and report directly, streamlining the event close-out process. The new system needed added functionality with multiple user roles:

- Event organizers should be able to look up the history for their events. They should be able to locate the work order from the prior years, and copy the event for this year, reducing the need

to duplicate data and, more importantly, alleviate the burden of work orders being submitted where the only comments were “same as last year.” Copying an event from a prior year could also link those events together, to aid in looking up prior years’ hours worked and reports. A role was necessary for event organizers. However, it was also important to associate that user and role combination with an organization. A given staff member or student might be the event organizer for one or multiple departments or organizations, and they should have access to their organization’s history. Additionally, this would ensure that someone who was not authorized did not submit an event request on behalf of another organization.

- Crew Chiefs should be able to see all the details for their upcoming events. They should also be able to review reports from the same event in prior years for any tips or warning from previous Crew Chiefs. A Crew Chief role was necessary, with that role being tied to a specific user & event combination.
- Members should be able to look up the crew call times for upcoming events, to know when to show up and work. They should also be able to see basic information about the services requested to confirm they have the skill set to assist with the event. They should also be able to get a list of the events they’ve worked in the past, and how many hours they worked at those events. A member role was required which granted this basic level of access.

Design Decisions

Once the functional design was done, it was time to start looking at technical design. Marr’s system had been written in Perl. While part of the issue with his implementation was the lack of proper documentation, the rate at which Perl libraries were deprecated and replaced put a high maintenance burden on LnL’s Webmaster. Even with proper documentation, a Perl based system would be difficult to maintain, and difficult to find people who were capable of maintaining it. WPI’s Computer Science (“CS”)

department also had no classes which taught coding in Perl to mitigate that knowledge gap. In short, Perl was not a viable solution.

Next, we considered building a true Common Gateway Interface ("CGI") application in C++. C++ is an object oriented language which is very widely used in modern Linux applications. A derivative of it, Visual C++, is also commonly used for development of applications on Microsoft Windows platforms. Being a widely used language, it was common for incoming CS and ECE students to already have a basic knowledge of the language. WPI's CS department also had various courses which students could take which used C and/or C++ to teach the core concepts. This meant that there were resources available on campus to assist with learning how to maintain such a code base. Another benefit of using C++ is that the core functions and libraries were fairly stable, with previous versions of base functions only being modified or removed due to vulnerability patches.

A CGI application functions by allowing a web server to execute a Command Line Interface ("CLI") or console application on the web server. The exact implementations can vary based on system architecture and preference of the administrator. The basic function of a CGI is for the web browser to execute a script and either: a) pass all the necessary data in to the script as arguments on the command line; or b) provide an Input / Output ("IO") stream from the web server to the CGI application. The application then takes the information, executes the appropriate code over the data set, and returns information back to the web server. The information it returns is generally Hypertext Markup Language ("HTML") content for the server to send back to the end user's browser, however can ultimately be any content which the developer has handled in the server or end user site.

In order to work, a C++ application would function as a true CGI application. Unlike other options which were scripting languages, C++ has to be compiled and executed. The CGI URL would be called, and the necessary data would be streamed into the application in a pre-determined format. That data would be

stored inside application variables. The application would continue to run, processing the data, while the user's web browser would seemingly hang polling. The output of the application, which would be streamed back to the end user's browser, would need to be a HTML document.

One of the major draw backs here is the need to compile the application. Even a single line code edit would require the application to be recompiled and re-deployed. This makes testing and debugging a bit more complicated and increases the risk of a small change effecting a large amount of the system. To mitigate this, the application could be broken up into multiple CGI executables for different functions, but this can pose other logistical challenges if there is shared code across multiple executables. At the end of the day, C++ was a potentially viable option, though not an ideal one.

Another consideration was PHP: Hypertext Preprocessor ("PHP"). PHP is a scripting language that runs in line with HTML code. A single file can alternate back and forth between PHP and HTML, and PHP can be used to generate HTML as well. The syntax and structure of PHP is much like C++, though not as strict in regard to formatting and typing. PHP is widely used for web based applications. Its lack of strict formatting gives it a low barrier to entry, and its similarity to C/C++ makes it easy for developers that worked on older CGI applications to adapt. Its similarity to C++ also means there are resources available on campus to assist with learning the language, despite there being no actual classes that use PHP.

PHP code is executed entirely on the server side. The result of the PHP execution, generally HTML code, is then sent over HTTP to the client's computer for the browser to handle. While the content is generally HTML, PHP can also dynamically generate other content such as JavaScript or plain text.

PHP's low barrier to entry is a double edged sword: it makes it easy for people to learn, but in some cases make it too easy to write insecure code. PHP has earned itself a reputation as being an insecure language which makes many system administrators nervous. Calling PHP an insecure language is a bit of a misnomer though. Certainly, a combination of poorly written code and poorly maintained file

permissions can quickly compromise a host. However, properly written and secured code, used in conjunction with proper file permissions, will work just as securely as any other language. It is, however, much easier to write insecure code that works than it is in other languages.

PHP would require proper security testing to be done on the code. It would also require a higher level of proficiency in in secure coding. WPI does have a class to assist with secure software engineering, but it is a 4000 level CS course, making it unlikely that someone looking to be LnL's webmaster would have taken it. PHP was a viable option, but still not an ideal one.

Our next consideration was Ruby on Rails ("Rails"). Rails drew much of its influence from Perl and, at the time of the original project, was starting to gain popularity among web developers. Like Perl, it is intended to run its own code exclusively. The Rails code should then generate all HTML output back to the end user. Unlike Perl, it is more object-oriented. It has various frameworks and tutorials built off of it to aid developers in rapid prototyping and deployment of sites. It also has built in database abstraction and supports classes. WPI, however, does not have any classes which use Rails, nor were there any real resources for Rails at the time. Rails is an option, but is lower on the list.

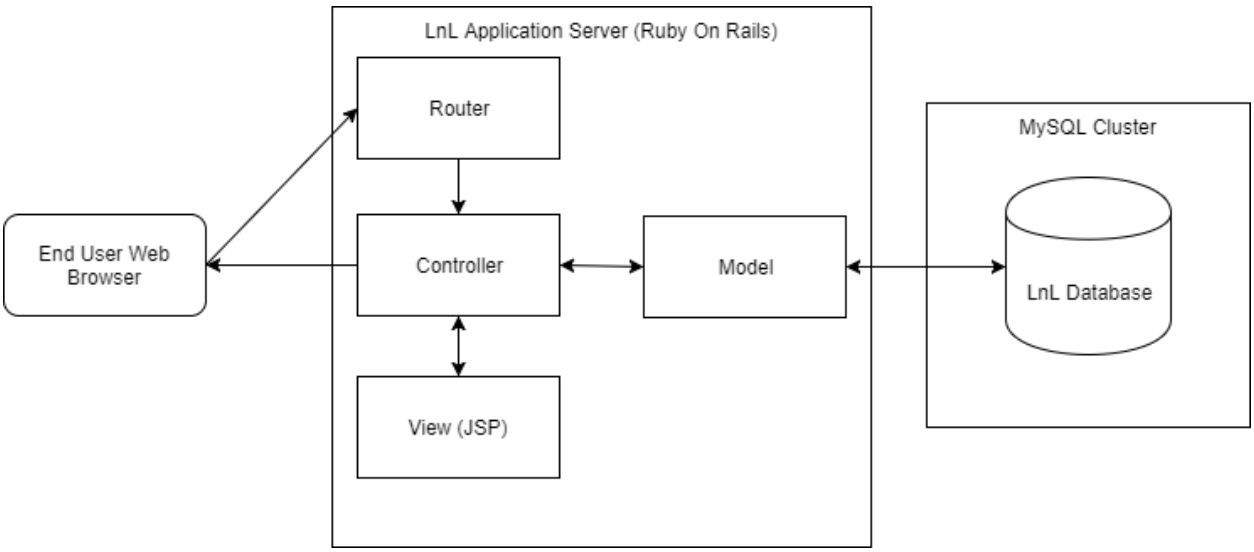

Figure 3: Ruby on Rails Architecture Diagram

The final consideration was a Java servlet application with JavaServer Pages ("JSP"). Java is a common language used for modern development. Like C and C++ it has to be compiled. Unlike C, which is compiled into a native executable, Java is compiled into byte code which is executed by a Java Runtime Environment ("JRE") or Java Development Kit ("JDK"). WPI has several classes which use Java, so there are resources available on campus for assistance with it. Java's structure, while not identical, is similar to C and C++, making it somewhat easier for people familiar with those languages.

JSP's are files which contain both HTML and Java code. Much like PHP, the author can switch back and forth, writing pure HTML code and adding in Java code or function calls only when necessary to produce dynamic content.

The architecture for implementation is referred to as Model View Controller ("MVC"). In the MVC model, the application is separated into three different components. The controller is the servlet, which receives the incoming requests from the client. The controller passes those request in to the Java byte code, or the bean, for processing. The bean then returns the responses to the view – in this case, JSP – which returns the generated content back to the client system.

For the considered method, the compiled servlet would be hosted by a servlet container, such as Apache Tomcat, JBoss AS, or GlassFish. The servlet container acts as a Hypertext Transfer Protocol ("HTTP") host, receiving user requests and relaying them to the proper servlet. The servlet container executes the Java byte code, which produces the output back to the user. Java with JSP was a viable option.

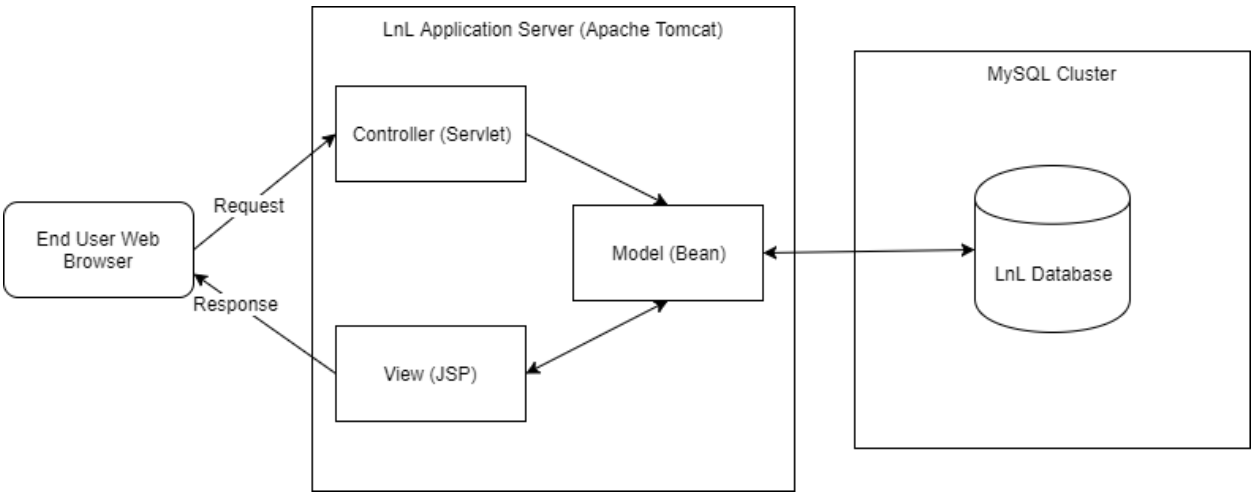

Figure 4: Java Servlet / MVC Architecture Diagram

Data Storage Options

The next question was where to store the data. Marr's database was stored in MySQL. MySQL is an open source, GNU Public License ("GPL") based, popular database engine. MySQL provides user level authentication, with granular permissions. It allows for SSL encrypted connections between the application and the MySQL server. It is diversified, and can function on any virtually any modern flavor of Linux, Windows, or Mac OS. WPI also hosts a MySQL server on campus on which any student or organization can request a database. Given that it is GPL, there are no licensing costs to worry about. Every language we have considered for the system also supports connections to MySQL.

The second option was Oracle. Oracle is a for-profit, closed source, enterprise database solution. It supports all of the same features that MySQL supports including SSL transfer and user-based authentication. Oracle can be run on many flavors of Linux, and on Microsoft Windows Server editions. WPI also has an Oracle cluster on campus which is used for systems such as Banner, WPI's Student Information System. It was unclear whether or not WPI would consider hosting additional databases on their Oracle cluster. Oracle, while known to have better performance than MySQL, also has with it a

steep per-database and per-user cost. All of the languages we looked at natively support connections to Oracle.

Microsoft SQL ("MSSQL") is Microsoft's database solution. Like Oracle, it is a closed source, enterprise application. It is also the hardest of the three options to maintain. MSSQL only runs on certain editions of Microsoft Windows Server, and is not support on Linux or Mac OS. While WPI does have MSSQL on campus, it is a system that is run by the Windows Team for their own infrastructure. It was unlikely that WPI would be willing to support additional databases on this infrastructure. Like Oracle, it has expensive per-instance costs, and would likely be cost prohibitive. All of the languages we looked at do support connections to MSSQL, though some require third party libraries.

Application Hosting Options

The third infrastructure design factor to consider was how the application would be hosted. Marr's database was hosted on the WPI "Users" server: the server where any user can host their own personal website. This has a number of benefits for LnL: LnL does not have to purchase or maintain any hardware; LnL is not responsible for security of the core operating system; LnL does not have to complete standard maintenance tasks such as patches or upgrades; if there is a problem, LnL can refer to the CCC's Service Level Agreement ("SLA") for the server in regards to restoration of service. The lack of administrative burden was a strong benefit in these circumstances. LnL does not have the luxury of selecting the best candidate from a large pool, as would be the case when hiring a full time employee. Typically, there are only one or two people interested in the Webmaster position, and their skill sets will vary from year to year. There is no guarantee of getting a Webmaster that is skilled in system administration, or that wants to acquire those skills.

It has some drawbacks as well: LnL does not have administrative access to the system; updates and patches may be applied at any time; the CCC can decide to stop supporting applications and server systems. This means that deployment of new libraries or server updates may be dependent on approval

from the CCC, possibly limiting the addition of new features. On the opposite side, updates may be applied, or software removed, which result in the application no longer working, leaving LnL in a chaotic situation.

Another option was for LnL to purchase hardware and host the server on their own. This eliminates basically all of the negatives of the CCC hosting it: LnL would have administrative access, they would control updates and patches, and they would decide what runs on the system. Of course, the pros of the CCC hosting it do not apply: LnL would be responsible for everything on the system and could not rely on the CCC to assist them with it. LnL would also need to find a place to host it. The traffic in many of the LnL offices make them all less than ideal to host a server; history has shown that people sometimes accidentally unplug critical systems to plug in their laptop or mobile phone. The areas with less traffic do not have climate controlled environments and can get very hot and humid: factors which decrease the life of the hardware. With LnL hosting it, there also would be no guarantee of any integration into WPI's authentication databases, or the ability to draw on information from the LDAP servers. These are both things that would be ideal as we rolled out larger scale access to the new system.

At the time of this project – in 2009 – cloud providers were in their infancy. Amazon Web Services ("AWS") and the Rackspace Cloud had both been up and running for about two and a half years (both founded March 2006), but adoption was slow and costs were high. Cloud hosting was not something I considered in the original project.

Due to the limitations WPI's CCC office might impose, I needed to decide which was more important: the technology, or the hosting. Certain languages and technologies may be ideal, but not be supported by the CCC. Similarly, certain hosting choices may be ideal, but limit which technologies we can use. To answer this question, I had a conversation with the then Executive Board. We weighed the pros and

cons of each option together. At the end of the conversation, we felt that the likelihood of issues with an LnL hosted server was too high and elected to use CCC hosted infrastructure.

Impact of Hosting Decision

The decision to use WPI's CCC hosted environment led to a conversation with Jesse Banning, a WPI UNIX Systems Administrator at the time. Banning was the point of contact I was given within the CCC. I sat down with Banning and gave him a bit of background on LnL, showed him Marr's systems, and explained to him what the goal of the new project was. I explained LnL's concerns with them hosting the application on their own and their preference to have it hosted on CCC run systems. Banning agreed with this viewpoint and said he was willing to assist.

The conversation then turned to what the CCC was willing to allow and/or support. From the beginning Banning made it very clear that PHP would not be permitted. He spoke of the extensive issues WPI had had in the past with mass produced insecure code, and the damage it had done: there was no room for conversation on that topic. We started talking about Rails, and Banning informed me that the CCC had not done any testing with Rails yet. While he was willing to consider it, he cautioned that it would likely take a long time for the CCC to adopt, if they allowed it at all. Java Servlets and JSP were something WPI was already getting ready to support for another team, though it was not their first choice. Banning strongly recommended Perl, C, or C++. He said those are the easiest for them to support, and they wouldn't even require us to have our own virtual machine ("VM"). With those, we could run them on the Users server with some small configuration changes.

In discussing databases, Banning said that the Oracle cluster was for Banner only and LnL would not be permitted to host any data on it. He couldn't make a determination on MSSQL, but believed the Windows Team would not be supportive of the idea. This made my decision for me: we were using MySQL. I was happy with that decision as I already felt it made the most sense for the application.

I took this information to my then project advisor, Professor Gary Pollice. Pollice and I had already agreed that Perl was not an option, as discussed earlier in this paper. This left us with a conversation about C++ and Java Servlets. We felt that C++ would be extremely rigid, inflexible, and difficult for future Webmasters to maintain. C++, while it could be adapted to meet our needs, was not designed to generate dynamic web content. Java Servlets with JSP was the only remaining option, and we agreed it was the most appropriate in the given circumstances.

With these decisions made, I scheduled another appointment with Banning. During the conversation Banning agreed to provide a VM on which LnL could host their new application. We agreed that since the VM would be hosted on CCC managed hardware, LnL would not have administrative access. The CCC had decided on using the Apache Tomcat servlet container, so the application would need to be developed and tested under this. Banning agreed to provide a way for LnL to update the code running without having to involve the CCC.

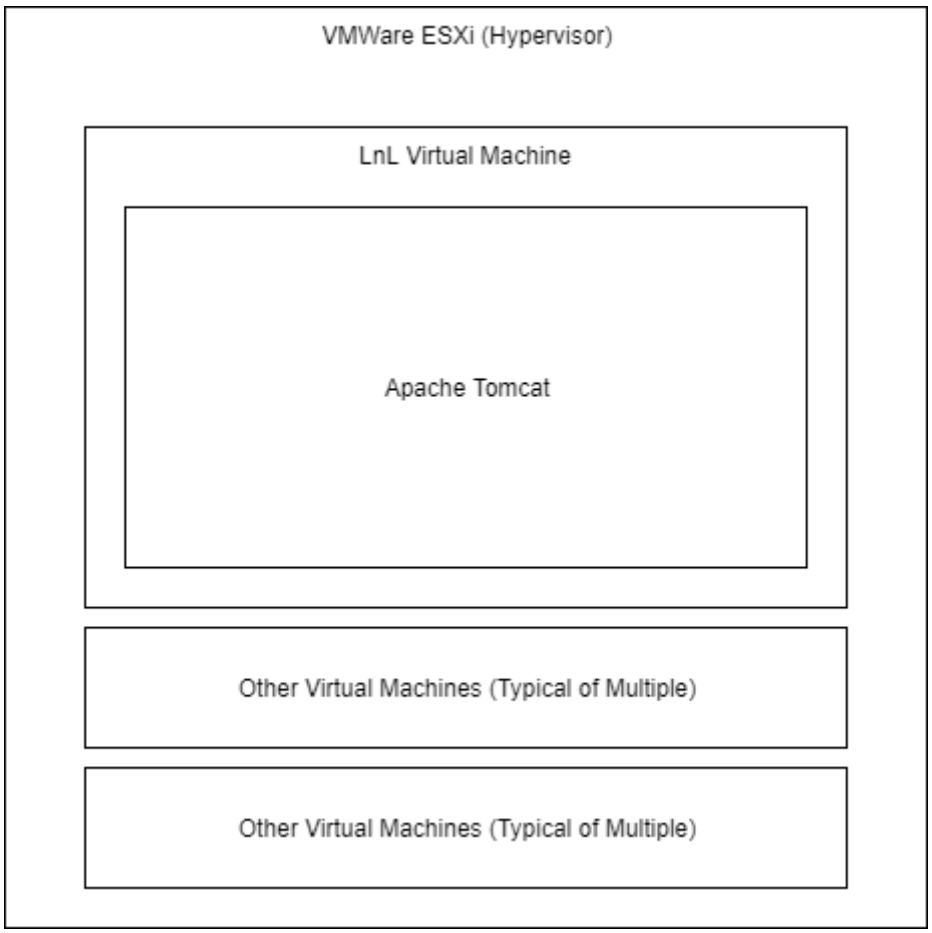

Figure 5: Virtual Machine Diagram (2008 Project Plan)

Implementation

At the time that this project started, WPI was hosting a SourceForge Enterprise server on campus. I setup a project on that server, including a Subversion ("SVN") repository to store the source code. I chose SVN because it allowed continued commits from remote locations with the ability to review history and quickly roll back to prior versions of code. Once I made the decision to use SVN, using WPI's SourceForge server was the obvious, logical choice.

For actually writing the code, I considered three options: NetBeans, Eclipse, and Vi IMproved ("Vim"). NetBeans and Eclipse were both industry leading integrated development environments ("IDE") widely used for Java development. IDE's are application which assist the developer with the code they are

writing. Their user interfaces ("UI") have features such as syntax highlighting, code folding, code validation, code name completion, and even recommendations. Vim is a UNIX and Linux text editor which does minimal syntax highlight and folding, but cannot do any of the validation, correction, or recommendation features.

Up until this point, most of the code I had written was in Vim. Vim's lack of code validation and other features made it a less than ideal choice for this large project. I had used Eclipse for some smaller class projects and found it to be bloated, often having poor performance. Pollice recommended the NetBeans IDE which, after looking into it, I found to be a significant improvement over the other options.

The next step I took was to analyze the schema of the current MySQL database. As a starting point, I felt I needed to at least reproduce this schema while making some improvements. While in the process of doing that, I continued to add other pieces of information I believed would be needed. After a week of this, I had a drawn out MySQL schema for the new system.

In retrospect, this ended up being a poor choice. Over the course of working on the project, I had multiple times where I was attempting to make code match the schema I had designed. About half way through coding, it became abundantly clear that I was spending too much time either making the code fit the schema, or re-working the schema. A more logical approach would have been to base the database schema off of the class object members. This is a lesson I learned from this project and have since employed in other, unrelated projects.

The next step was to begin writing the most basic use cases to start the framework. Use cases like "as a user, I would like to be able to login" and "as a user, I would like to be see a menu of available functions." From there, I was able to determine the basic building blocks necessary to start writing code. I started with the "main menu" use case, and created that page. This led to the need for a filter to check if a user was logged in. The filter, of course, then lead to implementation of the "able to login" use case.

This process continued, implementing different use cases, deploying, and testing their functionality on the CCC provided VM.

Two-thirds of the way through the project, I received an email from Banning saying that he wanted to speak to me. In that meeting, Banning stated that during our initial conversations he was under the impression that LnL was an administrative department; he was unaware of LnL's status as a student organization. Banning stated that the CCC, as a matter of policy, would not support hosting for student organizations. This stemmed out of concern for lack of necessary personnel to handle large scale support for student organizations.

In the last few weeks of the project, I scrambled to find other hosting. I spoke with the ECE department, CS department, and various officials in the CCC. While the ECE and CS department were willing to host the system for a brief time (while it was still academic in nature), neither was willing to commit to hosting a server long-term. The CCC continued to offer their support for Perl or C running on the Users server, but remained unwilling to host a Tomcat server for LnL. As the May deadline reached, the project ultimately went uncompleted and unimplemented.

Methodology (2017 Project)

Requirements Analysis

It's been almost nine years since the original project started and technology has changed quite a bit since then. The introduction of new standards such as HTML5, improvements made to JavaScript, and the push toward responsive web applications has resulted in client based generation and rendering of content, greatly improving the experience for the end user.

As I began to re-examine this project, I sat down with the current President (and former Vice-President) of LnL, Valerie Moore. Moore and I discussed the operations of the Executive Board. In these conversations, we were able to determine that while technology may have changed, the day to day functions of LnL have mostly remained the same. The current database system they are using may not be ideal, but it does what it needs to do, is supported by WPI Information Technology Services ("ITS") infrastructure, and could be maintained and updated by the last two Webmasters. The only feature it lacks is the ability to track trouble reports with broken equipment. That requirement is currently being accomplished through a third-party system: Best Practical's Request Tracker ("RT").

In examining a modern project, our requirements in terms of functionality are the same, with the addition of a "nice-to-have" for equipment trouble tracking. This means all our use cases from the previous project can be used here, and the need to do a separate survey of users is unnecessary. The next aspect of the project would be selection of the technologies to be used.

Design Decisions

The question of basic architecture now comes to the forefront. In 2008, it was assumed that we wanted to have dynamically generated HTML pages. Every click or query would make a call back to the server. The server would process the information and send a new dynamic page for the browser to render. In some cases this could have been done with page divisions to minimize rendering time but, overall, the

server would be handling all content generation. This is now considered a poor design for such applications as page loads are slow and use more resources, especially for someone on a mobile browser.

A more modern approach is the use of API's and a progressive client application. An Application Program Interface ("API") allows one application to provide a request and, sometimes, data to another application. The data is generally transferred in either Extensible Markup Language ("XML") or JavaScript Object Notation ("JSON"), though these is no standard which requires it to adhere to that format. The only requirement in data format is that the endpoints respond appropriately to content types they cannot handle, and properly indicate the content type of their return.

When an API call is made, the destination system handles the request and replies accordingly. Representational State Transfer ("REST" or "RESTful") API's work by using only the data necessary to complete a request. REST utilizes HTTP for its requests, making it easy to work with. The call provides a specific request, or instruction, to the web server and can allow a client application to use any of the normal HTTP verbs such as GET, POST, PATCH, PUT, and DELETE. Through these calls, client applications can retrieve, add, or manipulate data that the server application hosts.

Another API format is Simple Object Access Protocol ("SOAP"). Like REST, SOAP can utilize HTTP as its transfer method. One major limitation is that SOAP only utilizes XML for its data transfer, requiring the developer to use applications which support XML on both sides. This factor makes it not as widely implemented, or supported, as REST.

A progressive web application is a site that behaves like a user application rather than a web site. The application requests the data and objects it needs in the background in order to update content without having to reload the page view. To the end user, the experience appears as a faster, more responsive site.

Using REST API's, we can make the backend server generate minimal actual content. This allows the end user application to appear more responsive while minimizing the load on the server(s) and the bandwidth utilization for client activities. This also makes it very easy to build a native mobile app down the line, as the app could use the same API's for requesting and updating data. It would also make integration with external systems, such as RT, easier as many third party tools support integrations via REST.

As I chose to adopt a model of having a REST API with a front end application, the next question I need to examine is what technologies to use to write both the back and front end. Considering the backend application, I will start by examining the same languages and technologies I examined in 2008.

Perl 5, the most widely used version of Perl, was originally released in 1994. In 2008, when the original analysis occurred, the current stable version of Perl was 5.6; that version had been released in 2003 and had seen only minor bug fix improvements in the following five years. Since then, development of the Perl language has picked up. There have been ten major release versions of Perl 5, with the most recent 5.26 in May of 2017. However, as we look at the language, not that much has improved. Many of the features, such as JSON support, year 2038 compliance, and Unicode updates were simply necessary to keep the language viable and compliant with modern standards. Other features, such as the ability to emulate pervious Perl versions, were ironically necessary to allow backwards compatibility, while still allowing use of some newer features.

The long awaited Perl 6 was also released in December of 2015. Perl 6, which had been in the works since 2000, was intended to modernize Perl and clean up the syntax. It was intended to make Perl object oriented, use types, and to generally make it easier for users with experience in other, more modern languages to be able to develop in it. It was released not as a replacement for Perl 5, but rather as

something to complement it. Perl 6 has modules which can allow Perl 5 code to execute under its interpreter.

Still, Perl is a complicated language. Its syntax, while improved, is somewhat convoluted at times making it difficult to understand. There also are still no resources on campus at WPI which would aid students in learning and developing in Perl. This makes it still a poor choice for this project.

Neither ANSI C, nor C++, have made any significant changes in the last nine years. Language bug fixes and minor improvements have occurred to specific libraries, but the language as a whole has remained somewhat stagnant. C++ would actually be significantly easier to use (relative to the 2008 project) if it did not need to generate HTML content. Frameworks exist for the implementation of REST API calls in C++ code. Resources still exist on campus for development with C and C++, but the pitfalls associated with a compiled, rigid application still exist. C++, while still an extremely useful and powerful language, is still not a decent choice for this project.

PHP has also had some development in the last nine years. Since 2008, PHP has had 4 minor version releases. These releases included things like namespace support, allowing type declarations for variables in functions, improved session handling, safe mode, and many other changes which were targeted at improving the security of the language. While the language itself is not insecure, these improvements make it easier for developers to write secure code, and make it more difficult for a beginning developer to leave vulnerabilities unchecked. There were also additional improvements aimed at increasing functionality.

In addition to the minor releases, PHP had an intended major released, PHP 6, which failed. The goal of PHP 6 was to natively support Unicode in all applications, rather than requiring additional libraries to handle it. The release ended up being abandoned due to performance issues introduced into the code by the way in which the developers chose to implement Unicode.

The most recent major release of PHP, PHP 7, was released in December 2015. Remembering that PHP code is interpreted, not a compiled, has a direct impact on performance. The PHP interpreter can be compiled and optimized for the architecture on which it's running, but the end user code itself cannot. PHP 7 aimed to improve performance of end user code by reducing the size of the data structures passed between functions. Additionally, PHP 7 introduced the ability to set return variable types on functions, allowing for another layer of automatic error checking and improved security.

Still, despite these improvements, there is still a negative impression, relating to security, when it comes to PHP. Some education of secure coding concepts, such as CS 4400, would still be necessary to ensure that a developer understood the impact of, and how to properly use, these new features. If the application were written by someone with knowledge of secure coding, it would be more difficult to introduce vulnerabilities while doing minor enhancements or bug fixes. WPI ITS still does not officially support PHP on their servers, however they do support it on the public WPI website. They have also re-opened talks with LnL about supporting virtual machines, maintained by the LnL Webmaster, for other services which are difficult to host on common use systems. As a result of this, PHP is in consideration for the backend server code.

Over the last nine years, Rails has gained a decent amount of popularity and has become more widely adopted. The basic principles of the language, and the MVC architecture, has not significantly changed since 2008. ITS does, now, officially support Rails on the Users server. WPI still does not have any classes which use Rails, nor are there any real resources for Rails. Rails is still an option, but is still low on the list.

The final consideration is a Java Servlet / JSP implementation. While I chose this as the implementation method in 2008, I no longer find that as a good decision. The servlet model requires a heavy weight application (the servlet container) that has to be run with elevated permissions to function properly on

normal web ports. Additionally, Java code is not ideal for writing an implementation of this system. The overhead of the code and the applications required to run it aren't warranted by the desired implementation.

For the front end, the actual pages will need to be HTML. There is no way around that, since that is what the web browser will render. The question comes in the choice of technologies used to deliver that content.

JavaScript is a scripting language that can be run client side, and has access to the Document Object Model ("DOM") to manipulate the content on an HTML page. JavaScript has built in functionality to read, edit, re-arrange, and delete the objects that are present based on user input, or signaling from a backend server.

JavaScript can utilize Asynchronous JavaScript and XML ("Ajax") to send API requests to the web server in the background, transparent to the user. The responses to those requests can then be used to trigger modifications to the current page, a complete replacement of the page, or to redirect the user to a new page. Ajax Long Polling can be used to have a continuous connection to the server to simulate "push' style notifications, allowing near instantaneous update of content. To control appearance, Cascading Style Sheets ("CSS") can be employed. CSS is a framework which controls how the browser renders objects on the page through the use of object types, classes, and ID's. In this model we would ensure that modified and added elements of the page continue to appear as intended by the developer by ensuring the JavaScript sets the appropriate CSS class or ID.

Data Storage Options

For the actual database, we still have the same three major contenders: MySQL, Oracle, and MSSQL. MSSQL still has the same issues of being expensive, difficult to manage, and only supported on Windows. Oracle is also still very expensive. While WPI ITS still maintains both of these DBMS's, they do

not want to support a database for LnL. Since it's unreasonable to expect LnL to pay the licensing costs, or to have a Webmaster that is versed in administration of either of these, they are both still not a consideration.

MySQL, despite having been purchased by Oracle, is still open source, GPL software. ITS still maintains a MySQL cluster on which they allow students to host databases. If LnL wanted to run their own MySQL instance there are various courses at WPI that use MySQL, such as CS 3431, where students could seek assistance. MySQL also has a very active community of end users who support each other. It is important to note that both AWS and Rackspace support MySQL in a hosted model, which would allow LnL to offload the DBMS management to the provider if they were to host this in a cloud environment.

Additional DBMS's that could be considered are Apache Cassandra and MongoDB. Apache Cassandra, introduced in 2008, is a DBMS developed by Facebook for storing large data sets across clustered environments. It is released under the Apache License, and is free to use. It is a NoSQL database, which means that it is non-relational. NoSQL databases are designed for large data sets where one table does not necessarily have a relation to any other table in the database. They are designed for fast access of data, at the cost of potentially lacking reliability of the distributed information. MongoDB, introduced in 2009, is another NoSQL implementation, this one using JSON formatted queries. It is most useful in large, distributed data sets which are being access by systems already using JSON for API and other information exchange.

Due to the nature of the application we're building Cassandra, MongoDB, or any other NoSQL database are not really an option. The dataset we are storing is relational and grows at a predictable rate. We have no need for large, distributed database shards and replicas, and value data reliability and consistency over performance. As in 2008, MySQL is the most appropriate choice for a database engine.

Application Hosting Options

As mentioned previously, ITS does not currently permit running PHP on the Users server. Since our chosen language is PHP, this rules out hosting on the Users server. However, the political climate in ITS has changed along with technology. LnL is currently in negotiation with ITS to be provisioned with a VM for hosting other services. Should this project move forward, ITS may be amenable to allowing LnL to run whatever services are necessary for their operations, provided that ITS will not offer support on maintaining those services.

Another option to look at is cloud based hosting – providers like AWS and Rackspace ("cloud providers"). In the last nine years, cloud providers have become a very common solution for people and companies wishing to host services, but who don't have the resources necessary to host, nor want the burden of hosting, their own highly available servers.

The project could be implemented using the AWS Elastic Compute ("EC2"), Elastic Load Balancer ("ELB"), and Relational Database Services ("RDS") services hosted inside a Virtual Private Cloud ("VPC"). In this design, an EC2 instance would host the application server. The server would store and retrieve data from a MySQL database running on RDS. The ELB would be used for fast failover to a new EC2 instance to minimize downtime during upgrades or in the event of a failure of the original node.

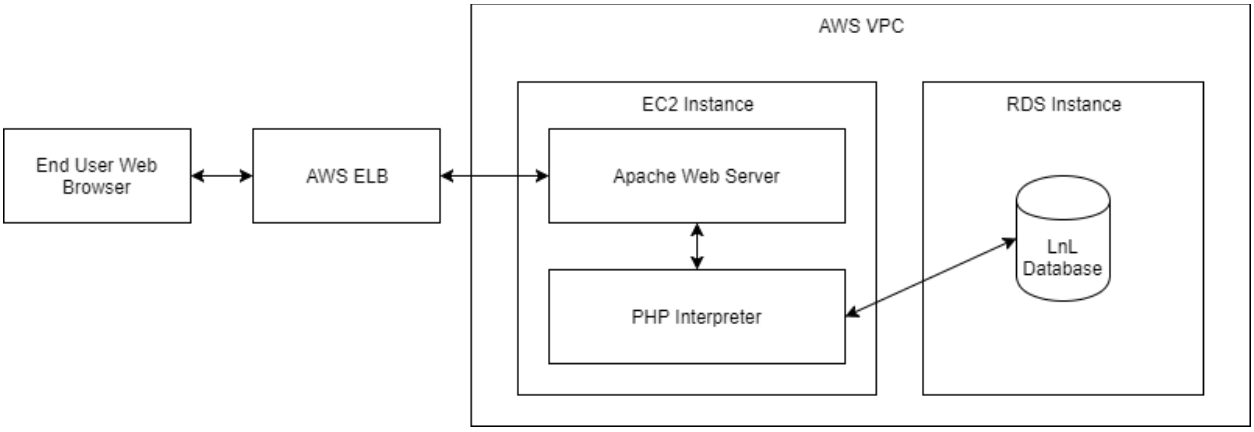

Figure 6: AWS Hosting Architecture

In any environment outside the WPI network, security must be a major consideration. On campus, LnL can rely on WPI's Information Security team to ensure the security of hosted systems; in AWS, LnL would need to take responsibility for ensuring system security. The architecture described above actually makes this fairly simple. The VPC contains all the hosted instances, and utilizes internal RFC1918 IP addresses. RFC1918 is the Internet Engineering Task Force ("IETF") definition for private internets. This means that the IP addresses contained in this specification are not routable over the commodity Internet without the use of Virtual Private Network ("VPN") tunnels. The only part of this architecture that is exposed to the Internet is the ELB, which only supports HTTP(S) requests. This greatly limits the possible attack vectors on the system.

Another important question we need to consider is authentication. Since AWS is outside the WPI firewall, this system will not have direct access to WPI's Active Directory ("AD") infrastructure. We can get around this issue by implementing Security Assertion Markup Language ("SAML") 2.0 federated login. SAML allows an Identity Provider ("IdP") server to create a signed assertion for an end user to pass on to a Service Provider ("SP"). WPI maintains a Shibboleth IdP, which is compatible with the SAML 2.0 standard. This would allow the system to utilize WPI's AD infrastructure for authentication without requiring it to have access to the AD servers.

As the system would be hosting student names, email addresses, ID numbers, and mobile phone numbers, we also need to consider implications of the Family Educational Rights and Privacy Act ("FERPA"). FERPA is US federal legislation that dictates, among other things, the minimum necessary privacy and security controls that an educational institution must use to protect a student's personally identifiable information ("PII"). Fortunately, AWS has a guide on how to ensure compliance with FERPA when storing data in AWS (see references for URL).

An additional consideration is whether or not WPI would permit LnL to have a wpi.edu hostname point to a machine which not on campus. At the time of this paper, ITS was not able to say definitively whether or not they would permit this. It was suggested that the answer would more likely be a no. In the event that ITS was not willing to permit a wpi.edu hostname on an off campus system, LnL could work around this by purchasing an additional domain name and redirecting. LnL could purchase a domain name, such as LnLDB.net, for hosting the application. This would also mitigate potential issues with SSL certificates as LnL would hold the ownership of the domain name. From there, LnL could put a redirect in their site (i.e. lnl.wpi.edu/db) which sent the user to the custom domain URL. Any standard links on the page, of course, could link straight to the appropriate URL on the custom domain.

The last consideration, with AWS, is cost and payment. AWS does have a free tier usage, but that free tier only lasts for one year. After the first year, the service costs money. While the cost of the architecture here is certainly within LnL's budget, there are two important, mitigating factors. First, is the organization willing to pay for it? This is a few hundred dollars a year which could be spent on equipment and supplies that serve LnL's primary mission. It is likely that, down the line, people will want to put that money towards something other than hosting. Second is the actual method of payment. LnL is not granted WPI Purchase Cards; the monthly cost of this hosting would need to go on a student's personal credit card, and they would have to expense it every month. They would need to remember to update the credit card on file as students change over and the previous billing contact was no longer enrolled and/or willing to deal with the paperwork. These two items make it highly likely that the system would fail due to the neglect that is inherent in such situations as college organizations.

While a viable option, this decision would need to be thoroughly discussed with the organization's current Executive Board, as well as their advisor, to ensure that the risks and responsibilities were fully understood, and would be properly transferred to future Executive Boards.

Implementation

The 2017 project was theoretical and not actually implemented. I am going to examine how the implementation would have worked.

WPI no longer maintains a Sourceforge server on campus. While the public version of Sourceforge still exists, people have tended to migrate away from SVN in favor of other systems, like git. Git works like SVN in that it tracks changes to source code files. Git works by allowing users to pull (download) and push (upload) changes to repositories via Secure Shell ("SSH") or HTTPS. Git supports branching, allowing multiple variations on a single code base to exist under the same repository. Branches allow two separate issues to be worked on concurrently, and then merged back together into a higher level branch down the line. In the last few years, git has become the most widely used version control system in the world.

For a minimum installation, git can be hosted on any server that is running HTTPS and/or SSH. More feature rich installations exist in systems like GitHub. GitHub is a Software as a Service ("SaaS") provider which provides online hosting of git repositories, a web interface to manage them, and other helpful features. GitHub allows any user to host an unlimited number of repositories for free if they are public. A user can host an unlimited number of private repositories for a small monthly fee. Students using the system solely for university related work are eligible to get that fee waived, having unlimited private repositories for free. A 501(c)(3), like WPI, can also apply for a free corporate account, with unlimited private repositories. Due to the available free hosting, the ease of outside collaboration, and the added features such as issue tracking, I would use a GitHub private repository for this project.

For an IDE, I could again consider options like Vim and Eclipse; NetBeans wouldn't make sense in a PHP environment as it's designed for Java. For the same reasons we discussed in the 2008 project, both Vim and Eclipse would be less than ideal choices. In addition, for this project, Eclipse is primarily a Java IDE.

While it does have plugins to support other languages, it is not as feature rich as some other IDE's for PHP.

A few years ago, while looking for an IDE that is geared toward this type of development, I came across PHPStorm. PHPStorm, made by JetBrains, is an IDE which is geared toward the development of PHP, HTML, JavaScript, CSS and other languages necessary for web development. PHPStorm supports useful features like syntax highlighting, code completion, syntax pop-ups, and even code generation for common code. PHPStorm is an ideal choice for development of this project.

The hosting options analysis really leaves this project open to go in two possible directions for hosting: hosted on a virtual machine at WPI; or hosted in Amazon Web Services. The decision on where to host would require a more detailed analysis than was possible in the scope and timeframe of this project. As such, I am going to examine the implementation details of both projects. These details assume that the code has already been written, thoroughly tested, and is ready for deployment.

WPI Hosted Virtual Machine

To start setup in a WPI hosted model, we would start by creating a virtual machine running a desired operating system. For this request, we would need to enter a ticket with ITS. While my preference is for Debian Linux (most current version), this is simply a personal preference. The architecture that I've described should be independent of the Linux distribution. For their internal systems, WPI currently runs Red Hat Enterprise Linux ("RHEL") version 7. At the time of this report, ITS could not say if they would require LnL to use RHEL or they would allow them to pick their own distribution. It is also unclear if LnL would qualify under WPI's existing licensing for RHEL, which may be a major factor in that decision. A compromise could be Community Enterprise Operating System ("CentOS"), which is the free version of RHEL.

Once the virtual machine was brought online, the next step would be to install the necessary packages. This is generally done through the OS's package manager. Package managers are utilities which allow the user to automatically download and install the most recent version of an application that is available for the operating system they are running. The package manager also takes care of installing any applications or libraries that the new application depends on. The instructions for package installation will different based on the OS selection. Debian's package manager would be Aptitude, while CentOS would use yum. The actual package names would also potentially vary between OS's. The minimum packages we would need to install would include Apache2, PHP, git, and MySQL Client. It should be assumed that we would install the most recent release of each.

The next step is to clone the source code from the git repository onto the system. This is done by the "git clone" command. The code would then need to be copied to the appropriate directories on the server. Ideally, we would create a script which would do a "git pull" and copy the code to the appropriate locations. This would help minimize the amount of work needed to be done for each minor patch to the system. The script could also be set to run automatically, or "cronned," to automate releases after code was promoted to the git master branch. This would allow for "zero-touch" deployments during non-peak times.

The next step would be to setup a database on WPI's MySQL host. This process is accomplished through going to WPI's database management page (http://www.wpi.edu/+mysql) and following the actions to create a database. The user would also need to create a username and password for the application to use when accessing the database. Next we need to configure the MySQL credentials on our application, load the database schema, and load any default or imported data.

For SSL, we could either obtain an SSL certificate from WPI ITS (if they are willing to provide one), or utilize something like the Electronic Frontier Foundation's ("EFF") Let's Encrypt. Let's Encrypt is a free

SSL certificate service from the EFF. The EFF provides an application, called Certbot, which runs on the system needing the certificate. Certbot runs and uses internal mechanisms to verify the authenticity of the hostname and automatically install the certificate in Apache.

At this point, we would need to make any necessary configuration changes to Apache's default configuration. These changes are potentially something that could be included as part of our source control, and copied to the appropriate locations using the deployment script referenced above. Once the configuration is in place, the server process should be ready to be started.

At this point the system should be online. The next step would be extensive testing to ensure that everything was installed and deployed correctly. Once that testing passed, the system would be ready to begin serving real users.

Amazon Web Services

Setting up hosting in an AWS environment is a different process altogether. AWS has a web based control panel (https://aws.amazon.com) for the majority of their operations. There are no tickets necessary to be entered with their team. Additionally, we can be sure that we can use whichever Linux distribution we desire. But, we need to do the VM creation and all the necessary configuration.

The first step of an AWS deployment would be creating an AWS account. Initially, the level of service we require from AWS would be supported under their "Free Tier" model. AWS's free tier allows the use of some services on a trial period free for approximately one year. Because everything we need would fall under free tier, we would not need to initially provide any billing information to AWS.

Our next step would be to create our first Virtual Private Cloud in the data center we chose. AWS has several datacenters across various geographic regions. Since the majority of our users are in Worcester, MA, we would want to pick the data center / region that is closest to Worcester. There are two regions that are geographically close to WPI. Those regions are US-East-1, which is located in Virginia and US-

East-2, which is located in Ohio. The latency between WPI and each of these two locations is minimal, so for the purposes of this write up we will go with the physically closer location: Virginia. The specifics of which availability zone inside US-East-1 are not important for an initial deployment, or for a deployment which we do not intend to cluster, so we can select whichever they give us as the default.

Once the VPC is created, our next step would be to provision an Elastic Compute 2 instance (Amazon's version of a virtual machine). When provisioning an EC2 instance, Amazon will ask us to select from a list of available Amazon Machine Images ("AMI"). An AMI is an image file that someone has put together for rapid deployment to an AWS EC2 instance. There are AMI's available that have been produced directly by Amazon, or by members of the community. For example, since I prefer Debian, I would select a recent release of Debian from the community AMI's.

The next question AWS will ask is the size of machine that we need. Since we're trying to stick with the AWS free tier, we want to select the best machine that is still free tier eligible. This is currently "t2.micro." A t2.micro instance has a single virtual CPU and one gigabyte of RAM. The storage is obtained through Elastic Block Store ("EBS"). As you go through the new machine wizard, it will ask various questions include the VPC to assign, the subnet, shutdown behavior, etc. For these questions, the defaults should all be acceptable answers. One change we will want to make is to the security group configuration. We will likely want to limit the scope of who can SSH into the VM. In my example, I am limiting SSH to WPI's public IPv4 space (130.215.0.0/16).

Once the build of the VM is in progress, we can go build our Relational Database Services instance. While creating an RDS instance, we want to select MySQL from the list. We will want to select the version covered under the free tier, which is "db.t2.micro." Multi-AZ deployments replicate the RDS instance across multiple availability zones in the geographic region. This provides seamless fail over in the event of data center issues which effect only one availability zone. We will need to opt-out of Multi-

AZ deployments to keep our free tier. Just like with the WPI implementation, we will need to setup credentials for the system to access the database as well. We will want to make sure the RDS instance is launched in the same VPC and subnet group as the EC2 instance. We do not want the database to be publicly accessible.

As the RDS instance is being built, our virtual machine should now be online. Just like in the WPI version, we can sign into the machine and start installing packages. The same packages should be required in this deployment. As with the WPI hosted system, we would also need to clone our code from the git repository once the git client is installed.

Once RDS has finished building the instance, and our code has been cloned to the virtual machine, we can configure access to the RDS instance and perform the initial schema and data load. The process here would be the same as in the WPI hosted version. We would also want to deploy our code, configure Apache and start Apache. While the configuration may differ slightly for the AWS deployed version, the process to configure and the recommendation of using a deployment script would be the same.

The process of obtaining an SSL certificate is not as easy for the ELB as it is for a non-load balanced server. Certbot won't work with an ELB except under strict ownership conditions (which LnL would not meet for wpi.edu). As a result, there are two options. In the first option, WPI ITS could provide LnL with a certification that could be used in the ELB. The second option would be to use the AWS Certificate Manager ("ACM") to obtain a certificate for the specific hostname. ACM provides free, Amazon signed, certificates for services being hosted on AWS. In order to obtain a certificate through ACM, the user fills out the certificate request in the web portal. AWS then sends an email to the registered owner of the domain name requesting authorization to issue the certificate. This authorization is limited to the specific hostname that was requested.

Both of these methods of obtaining a certificate require the approval of WPI ITS. While not ideal, if WPI ITS is not willing to assist with issuing an SSL certificate in an AWS hosted environment, the instructions above for running Certbot could be used on the EC2 instance. In these circumstances, we would forgo setting up an ELB and have users directly access the EC2 instance. This removes the benefits of fast failover, and zero downtime deployments, but would allow us to ensure that our system was protected by SSL. If we were to go this route, we would need to modify the security group for the EC2 instance to permit traffic from anywhere to ports 80 and 443.

Another option would be using a custom domain name. If LnL were to purchase their own domain name, then WPI LnL would be the only party required to authorize issuing the SSL certificate. In this case, we could still have a valid certificate issued by ACM for use on the ELB. This would also be the method to use if WPI ITS was unwilling to point a wpi.edu hostname to an off campus system. Of course, purchasing a customer domain name is an additional cost that needs to be paid either on an annual basis, or prepaid for a number of years up front.

The last step would be to create an Elastic Load Balancer. The ELB creates a VM independent endpoint for web browsers to hit. The use of an ELB allows the system to have a single end point which can point to a single or multiple EC2 instances. This helps us to cluster the system down the line or, more importantly, to be able to quickly recover from a failed EC2 instance. An ELB will also allow deployment of an upgraded version of the system without having any end-user downtime. This is accomplished by building a new EC2 instance with the upgraded code and then moving the ELB to point to the new instance.

When creating the ELB, we will want to be sure to add HTTPS to the list of listeners. We will want to make sure we are deploying it in the same VPC and availability zone as our EC2 instance. We can

configure our certificate settings, the EC2 instance that is the member of the load balancer, and enable it.

If WPI ITS is willing to point a wpi.edu hostname name at the ELB, then this is the stage where that would need to occur. Due to the way AWS ELB's work, ITS would need to create a CNAME entry in their Domain Name Service ("DNS") records to point a hostname name, such as lnldb.wpi.edu, at the AWS provided hostname. A CNAME, or canonical name, is a DNS record which makes one hostname into an alias for another. They allow for flexibility in the way hostnames are resolved, with the tradeoff of a slower lookup time. This is because each CNAME that is returned in a DNS query results in another query needing to occur. An A record is a direct mapping of a hostname to an IP address. Due to the way AWS ELB's work, there is no way to do this with an A record.

If WPI was not willing to point a hostname at the ELB, or if LnL chose to use their own domain name for SSL certificate reasons, this is the point where we would need to add our DNS hosting records to our registrar. Since we are working solely with AWS systems, it would make sense to use Route 53. Route 53 is AWS's hosted DNS solution. Route 53 has direct integration with their internal services, like ELB, to make the DNS provisioning faster and more efficient. To setup Route 53, Amazon provides four host names for the DNS glue records. Glue records are domain name to hostname mappings which state what DNS servers are authoritative for a given domain name. Authoritative means that the DNS servers are trusted to provide accurate responses for the given domain name. From the registrar's control panel, we would delegate the purchased domain name to those four glue records. Once completed, AWS would be authoritative for our domain name.

At this time, the system should be up and running. As testing to ensure that the application deployed correctly, the system will be ready to start taking user load.

Comparison

Currently Running System

In September 2013, under the leadership of then Vice President Jared Erb ‘14, LnL initiated a project to replace Marr’s ailing system. Three people were asked to participate in the initial requirements and design discussion: Matthew Forman, Gabriel Morell, and me.

Forman proposed first, suggesting that the best solution was a Microsoft Access 2013 database. He justified the project saying it was quick and easy, and that he could have it done in a few weeks. Microsoft Access, while good for rapid prototyping and small, temporary projects, doesn’t scale well. It also doesn’t work well across multiple operating systems, nor does it allow for access of information via a web site. Forman elaborated that his plan would be to build an Access database in the short term, and eventually build an integration into the database so it could be accessed via a web browser.

In the conversation that followed it was brought up that ninety-five percent of work orders were currently submitted via the LnL website, with only five percent coming in via telephone or e-mail. That five percent was already taxing enough, as it often required multiple messages back and forth to get all the details from the client. Additionally, its lack of easy and immediate cross platform support would prove challenging. Three out of seven members of the Exec Board were on Mac OS based machines, which would not be able to cleanly use an Access database that was hosted on a Windows machine. There were far too many cons for them to consider this approach.

Without much consideration, I went in and presented the same system I had been working on 5 years prior. Again, we discussed my proposal. While a significant improvement over Forman’s proposal, the same concerns of where it would be hosted gave pause. Erb wanted to make an immediate decision, and no one could guarantee that the CCC would be receptive to the idea of providing VM. Morell also

pointed out that a monolithic Java MVC approach was no longer considered state of the art, and something that people were trying to get away from.

Morell proposed a Python solution using the Django framework. Django is a Python based framework that uses packages to extend its functionality. The framework deals with core operations such as authentication, authorization, intrusion prevention, administrative interfaces, Rich Site Summary ("RSS"), and database abstraction. Morell's proposed solution was to use Django for the base it provides and to implement custom packages for events, inventory, meetings, and projection.

Morell's estimate to get online was short: two months for a beta. Since Django was based written in Python, it was something the CCC would permit on the Users server. It was a more modern approach that was believed to be simpler than the Java MVC I had proposed, and was most definitely more appropriate than Forman's Access proposal. Armed with the database schema and requirements write-up from my 2008 project, Morell began work. Seven months later, in April 2014, the new system went live.

Since 2014 Jake Merdich '18, a former LnL Webmaster now Head Projectionist, has worked to maintain the system. In the last 3 years, Merdich has worked out a decent amount of bugs, added additional features, and customized behavioral flows of the application. They've moved authentication from LDAP to Shibboleth, but kept LDAP functionality to lookup data on community members. All of these are things that are expected when you build a custom system, but the one thing Merdich has commented on it – it's not easy. It really shouldn't be Merdich's job anymore, but both the current and previous Webmasters don't know enough Python to work on it. It's difficult to come in to a code base you didn't write and start modifying it. While Python code is easier to read than Perl, it has many of the same issues with structure. The end result here being that, Python probably wasn't the best choice for the long term.

User Experience

User experience is key in the digital age. Statistics show that if a page load takes more than three seconds, forty percent of users will abort and either abandon the attempt or re-submit (Kissmetrics). In addition to page load times, content presentation is also key. Too much text or poorly organized information can contribute to users giving up on their tasks (Kissmetrics).

As a user navigates these four different systems, we should observe what their experience will be. Marr's original system was entirely text and link based. The system generates and processes all data on the server side, which can lead to blank or hanging pages after a user clicks a link, submits information, or preforms a search. That lag can lead to users believe the system was frozen, not realizing it is processing in the background. As pages hang, users have a tendency to re-click a link, re-submit information, or re-run a search. This can result in higher server load, a longer delay for the end-user, and, in the case of form submission, duplicate data being created.

My proposed 2008 system fixed one part of that issue – the appearance of the user interface. Matching the design of the then current LnL website, the system had images that were used for field headers, buttons, and statuses. It also utilized more color than black, white, and yellow. However, a Java Servlet approach doesn't fix the issue of all the content being processed on the server side. The 2008 solution was still susceptible to poor user performance and feedback when operations were in progress. This can be improved a bit with JavaScript, but that wasn't planned at the time.

Morell's system was much the same as my 2008 system. It added colorful buttons, and some images to make the UI more attractive. Django has the same lack of dynamic content though. Just like Marr's system, and my 2008 system, it required the page to re-load and re-render with every click. Again, JavaScript could be used to improve some of this experience, but the burden of doing so on the Django framework would be high.

Looking at the new proposed system, many of these problems go away. We would certainly employ more colorful pages and images, as planned in 2008. But we're able to learn from the user complaints and take advantage of newer technologies to build progressive web pages, using JavaScript, RESTful API's and AJAX.

For the user experience, this means the system can start to render the page before it has all the information necessary to populate it. As the page renders, or as a user selects different options, the system can pull specific information and refresh specific portions of the page. It can dynamically load content as a user scrolls, or clicks a link. If the user's actions are predictable, it can even preload content in the background for a near instantaneous response. This allows the user to feel more like they are using a desktop application rather than a web page.

Chosen Technology

In 2008 we chose Java as our language for implementation. We chose Java because it would be supported by the CCC, there were campus resources to provide instruction and assistance, and the model was popular in web development. At the time, this seemed the most logical choice.

In the 2013 project, not much group consideration was given to the choice of technologies. It was weighed against two other proposed solutions, but the final decision was based off of the projected time to implement, not off the pros and cons of those technologies.

As we look at a hypothetical project in 2017, it is clear that the technologies chosen in 2008 and 2013 are no longer appropriate for the solution. The use of the Java Servlet / MVC model has been declining due to its monolithic nature. The UI of the application could be adapted to function as a more progressive page, using JavaScript and AJAX, but we still run into the problem of the rigid, monolithic back-end. As previously discussed, the Java byte code requires a recompile and a reload of the servlet for every change made. While this model can work fine in a clustered environment, it's very annoying

for a single server environment. Additionally, WPI ITS has stated that they are still unwilling to support a Tomcat installation, regardless of where or how it is installed.

Python is a good language for scripting and writing small applications. However, as a Python code base gets larger it becomes significantly more difficult to maintain. Generating HTML code, and especially JavaScript, inside Python is cumbersome at best. The Django framework improves some of those, as it provides some of the base functionality necessary for the system. The developer doesn't have to spend time writing code to handle a user base, authentication, database access, authorization, module management, etc. But there's a tradeoff in that the custom modules must be written in Python. The complexity of the language, along with the lack of educated Python coders on campus, makes this a poor choice.

The chosen implementation is much more flexible. Creation of a progressive web using HTML, JavaScript, AJAX, and API's allows for a faster loading, more dynamic, more adaptable web based system. The use of API's provides for easy integration with a mobile application, a third party system, and multiple active front-end UI's, for ease of testing and migration. PHP permits the developer to make and test small changes to the system without having to re-compile, or reload the server.

Mobile Access

Users want information at their fingertips. In all of the example systems, we're working toward that goal by providing access to information via a web browser. With proliferation of smart phones and similar devices, it's important to consider mobile access. Standard pages don't render well on mobile browsers.

In the 1990's, mobile phones were still limited to phone calls and, if you had a state of the art one, text messages. There was no such thing as a mobile web yet and I'm sure that mobile access wasn't even a consideration that Marr made. While Marr's system was annoying to scroll around in, the lack of any rich media content made the load times for retrieving information very quick.

Even in 2008, mobile still was barely a consideration. The early versions of Android devices had started to come out in beta releases, but were not on the consumer market yet. The first iPhone had hit the shelves, but was very expensive and not common on college campuses. Some people had Blackberry devices, but the web browsers on those functioned very poorly. Instead of focusing on mobile development, I focused on the ability to push email and text messages notifications out of the system. This seemed the most relevant way, at the time, to communicate via mobile. It is unknown how well the 2008 system would function in a modern web browser since it's not online anywhere to test.

In 2013, smart phones were becoming much more popular. Lots of users with Androids and iOS devices were on campus. However, the scope of time given for the project, and the failure to meet that deadline, resulted in Morell not considering mobile development. Considerations have been made as to what would be necessary, but the custom code and the Django framework make it difficult to produce a responsive page, and virtually impossible to produce an application. The page renders and functions on a mobile browser well enough, though the performance is not ideal.

In 2017, the question ends up coming down to seamless experience and performance, versus ease of maintenance for cross platform support. Given the lack of time that would be dedicated to development, I would choose to build the pages in a responsive fashion. The important part, though, is that mobile would need to be a primary consideration this time around.

In the modern world, there are two solutions to this problem: mobile optimization and mobile application. Mobile optimization is accomplished through the use of a responsive web design. A responsive page uses HTML and CSS to automatically detect the dimensions of your device and re-size the elements on the page so that it displays properly. Often this includes pushing content to be more vertical, or removing content that isn't necessary for the page view. The task of building a responsive page certainly isn't trivial. It requires careful consideration of size and positioning of elements and, in

the longer term, extensive testing on devices of various sizes. It can also be slower than an application, as applications can more easily cache static elements like images and page layouts. However, responsive web design is the easiest method of ensuring your page works across any device OS / size combination.

On the other hand, creating a mobile application helps to have better performance, easier data entry, and overall a more seamless user experience. Since the web system is based on a RESTful API, creating a mobile application shouldn't require much back-end work. We should be able to use the existing API endpoints. The application is written to pull data from those endpoints, and have a native display on the mobile device. Applications do have downfalls though. There are currently three major mobile operating systems: Android, iOS, and Windows Mobile. All three of these have their own specific programming nuances. Even to cover the two higher user base platforms, Android and iOS, would require two separate applications with limited codebase re-use. As new upgrades to these OS's come out, new versions of the application need to be released to maintain compatibility.

Hosting Options

In 2008, we had two hosting options: self-host on a machine in an LnL controlled location or host on a CCC provided VM. We chose to host on a CCC provided VM which, in the long term, ended up not actually being an option. As we review that hosting decision, it was the most logical one to make with the information we had at the time. While an argument can be made for LnL hosting the system on their own, history has shown this to be a poor decision. Over the last ten years, LnL has attempted to self-host several systems for tracking inventory and related problems, as well as an internal Wikipedia. All of those systems, except the most recent one brought online last year, are no longer functional. The most recent one suffered over a month of downtime this past summer after the location in which it was being hosted was shut down.

In 2013, not much consideration was given to the hosting decision. It was an simply assumed that the system would be hosted on the CCC Users server. Since the chose technology was supported on the Users server, there was no need to discuss other options.

As we look in 2017, we have two real options to consider: an ITS provided VM, or an AWS VM. There are many benefits of the AWS provided virtual machine: built in redundancy for storage and database; fast failover to a new EC2 instance in the event of a VM failure; full control over the VM; root access to the database server; load balancers allowing the quick switching between VM's, for upgrade or testing. The main drawback to the use of AWS is billing. AWS still has no model to allow for a pre-pay or billing, resulting in a high likelihood that a student run organization would have issues resulting in the account and services being suspended. AWS, at this time, does not offer free services for 501(c)(3) organizations, beyond the initial year of free tier services.

The other option, an ITS hosted VM, lacks pretty much all of the pros of AWS: no root; no built in redundancy; no fast failover. Though, it has the benefit of the administrative burden being handled by ITS, with any cost associated being billed through the campus Inter-department Transfer ("IDT") system. The likelihood of ITS suspending service because LnL failed timely payment of an IDT is virtually non-existent.

This is ultimately a decision for the LnL Exec Board to make, however I believe the decision made in 2008 and 2013 was correct. LnL should stick with an ITS VM as long as ITS is willing to allow it. In consideration of this, a written agreement, such an SLA or memorandum of understanding ("MOU") should be put in place between LnL and ITS outlining each group's requirements to ensure that neither side unfairly burdens the other.

Glossary

501(c)(3) – The chapter and verse reference for a public charity under the US Internal Revenue Service tax codes. A 501(c)(3) organization is a public charity, recognized by the IRS, entitled to no-tax status. Donations to such an organization are, generally, tax exempt.

A Record – A mapping of a hostname name to an IP address on a DNS server. Multiple A records can exist for a single host name.

Active Directory ("AD") – Activate Directory is a user, workstation, and network management system developed by Microsoft. AD is frequently used as a master for authentication, authorization, and accounting in large institutions.

Ajax Long Polling – A process in which the client application initiates an Ajax request to the server with a long timeout. The server then pauses the response to that request until it has content to return to the client. This is a method of simulating push style notifications in a web browser.

Amazon Machine Image ("AMI") – An Amazon Machine Image is operating system image that is ready to be deployed to an EC2 instance. AMI's are either built by Amazon or end users. End users can make AMI's they build available to the public.

Amazon Web Services ("AWS") – A geographically distributed collection of cloud based hosting services provided by Amazon. The services include virtual machines, load balancers, data base services, file storage, certificate services, and a large number of other services for web hosting.

Apache Cassandra – A free NoSQL based DBMS, created by the Apache Software Foundation, which is designed to be clustered for performance and high availability.

Apache Tomcat – A free Java Servlet Container created by the Apache Software Foundation.

Application Program Interface ("API") – A set of well-defined functions and routines that allow an external program or application to interface with a running application.

Asynchronous JavaScript and XML ("Ajax") – A method of sending and retrieving data, in the background, from a client web browser without requiring a page reload.

Authoritative – In terms of DNS, a term used to describe a server which is trusted to provide accurate information for the given domain name. Authoritative servers are established through glue records.

AWS Elastic Compute ("EC2") – AWS's name for a virtual machine hosted on their infrastructure.

C – A general purpose programming language that focuses on imperative programming.

C++ - An enhancement on C that was intended to be more object oriented.

Canonical Name ("CNAME") – A mapping of one hostname to another hostname for DNS name lookup; sometimes referred to as an alias.

Cascading Style Sheets ("CSS") – A document that is designed to describe how a markup document should appear when rendered. It is most commonly used for rendering HTML on web browsers.

Certbot – A utility, created by the EFF, which enrolls a server in the Let's Encrypt program, authenticates a request, and obtains a signed SSL certificate.

Crew Chief ("CC") – The LnL term for the person who is in charge of the crew for a particular event. For large events there can sometimes be more than one Crew Chief.

College Computing Center ("CCC") – The former name of WPI's ITS department.

Common Gateway Interface ("CGI") – A protocol for web servers to run console based applications to generate dynamic content on web page.

Command Line Interface ("CLI") – An interface that allows a user to run text based comments and receive text responses.

Community Enterprise Operation System ("CentOS") – The free version of Red Hat Enterprise Linux. The free versions lacks the enterprise support given to RHEL customers.

Database Management System ("DBMS") – An application that allows a user to interact with data stored in a computer database.

Debian Linux – A free distribution of Linux released under the GPL.

Document Object Model ("DOM") – A language independent API that allows code to interact with an HTML document as a tree / an array of objects.

Domain Name Service ("DNS") – A service which allows users to locate Internet based systems by translating hostnames to IP addresses.

Eclipse – Eclipse is a free, open source IDE. It is the most widely used Java IDE.

Elastic Load Balancer ("ELB") – AWS's name for their implementation of a hosted load balancer.

Electronic Front Foundation ("EFF") – A non-profit digital rights group dedicated to ensuring user rights and privacy in their interactions with computer systems.

Executive Board – The governing body of LnL. The Executive Board is made up of 7 members, each with an equal vote, that decide the direction and run the day to day operations of the organization.

Extensible Markup Language ("XML") – A markup language that is independent of the data contained or the consumer of the data.

Family Educational Rights and Privacy Act ("FERPA") – A set of US federal laws which dictate how information about students of a college can be stored, accessed, and disseminated.

Git – A version control system which tracks branches and modifications to source code files.

GitHub – A SaaS system which hosts Git repositories, and other related features, for developers.

GlassFish – Oracle's implementation of a Java Servlet Container. Released under the GPL.

Glue Records – A mapping of a domain name to a host name, usually with the root name servers, which identifies where to find an authoritative DNS server for a given domain name.

GNU Public License ("GPL") – A widely used software license which grants the end user free rights to run, share, and modify the application as long as proper credit is given to previous authors.

Hypertext Markup Language ("HTML") – A markup language used for creating web pages. HTML is the standard template used for web pages.

Hypertext Transfer Protocol ("HTTP") – The primary protocol web browsers use to retrieve HTML pages from web servers.

HTTP Secure ("HTTPS") – The SSL encrypted version of HTTP.

Identity Provider ("IdP") – In the SAML model, a server which provides authentication and authorization to a SAML service provider.

Integrated Development Environment ("IDE") – A software application which is designed to aid the user in developing software in particular languages. IDE's offer benefits like syntax highlighting and verification, code completion, and VCS integration.

Internet Engineering Task Force ("IETF") – A standards body which develops and maintains Internet standards.

Information Technology Services ("ITS") – WPI's department which handles hosted services for the campus. ITS is responsible for most hosted applications on campus, including AD, Banner, and the Users server.

Java Development Kit ("JDK") – A JRE which contains additional resources necessary to assist in the development of Java based applications. This includes the Java byte code compiler.

Java Runtime Environment ("JRE") – An implementation of Java which allows the system to execute compiled Java byte code.

Java Servlet – A Java program that provides services by extending the endpoints hosted by a Java Servlet Container. Java Servlets typically host HTTP requests.

Java Servlet Container – An application which providers an HTTP interface to Java Servlets, allowing them to share content.

JavaScript – An interpreted programming language, often used to generate dynamic and progressive web pages. JavaScript facilitates content on web pages changing without the need to reload the entire page.

JavaScript Object Notation ("JSON") – A specific data format used to transfer objects between clients and servers. It is often used in API requests and responses.

JavaServer Pages ("JSP") – JSP is a server-side Java implementation, allowing the generation of dynamic web pages.

JBoss AS – An implementation of a Java Servlet Container, originally authored by JBoss. The application was purchased by Red Hat and is now called WildFly.

Lens and Lights ("LnL") – An organization at WPI which provides sound, lighting, power, rigging, and projection services to the WPI community.

Microsoft SQL ("MSSQL") – Microsoft's implementation of a SQL based DBMS.

Model View Controller ("MVC") – A user interface architecture which separates the request and response sections of an application.

MongoDB – A NoSQL based DMBS which is engineered for clustered high performance.

MySQL – An open source, relational, DBMS produced by Oracle.

NetBeans – An IDE written in, and geared toward development of, Java. It is released under the GPL.

NoSQL – A term used to describe databases which are not relational in nature. It does not mean that they do not use SQL to access the data in the database.

Perl – An interpreted language often used to implement scripts, and sometimes used to generate dynamic web content.

PHP: Hypertext Preprocessor ("PHP") – A server side interpreted language that is primarily used for the generation of dynamic web content.

PHPStorm – An IDE by JetBrains geared toward the development of web pages which use PHP as the server-side code.

Projectionist in Training ("PIT") – The LnL program for training members in the equipment used, as well as the Massachusetts laws regarding, the projection equipment under LnL's control.

Python – A widely used, interpreted language which was developed for scripting operations on Linux systems. It is also used, executing server side, to generate dynamic web content.

Rackspace – A cloud based hosting provider. Rackspace is like AWS, but has a smaller footprint and service offering.

Red Hat Enterprise Linux ("RHEL") – A paid version of Linux released by Red Hat, targeted at enterprises and large institutions. RHEL comes with an SLA from Red Hat.

Relational Database Services ("RDS") – AWS's name for their hosted databases.

Representational State Transfer ("REST") – A stateless API implementation.

Request for Comments ("RFC") – A publication from the IETF that defines systems and protocols for systems connected to the Internet. The documents function as a set of standards.

Ruby on Rails ("Rails") – A server side web programming language released under the MIT license.

Secure Shell ("SSH") – An encrypted method of accessing the command line of a remote server. SSH also support tunneling IP requests over the session.

Security Assertion Markup Language ("SAML") – SAML is a protocol which allows an authentication mechanism in one network to provide authentication to a service provider on another network without the need for direct communication between the two servers.

Service Level Agreement (“SLA”) – An agreement between parties as to what the expected service level is for a given system, what the response time is to correct deficiencies in that service level, and what recourse is available if the agreement is not met.

Service Provider (“SP”) – In the SAML model, a Service Provider is an application with consumes an assertion given by an Identity Provider. The SP uses the information to authenticate the user.

Simple Object Access Protocol (“SOAP”) – A specification for transferring information about objects, often in a client server model. It is most commonly used in API’s.

Software as a Service ("SaaS") – The term to describe companies providing access to hosted software for the periodic payment of a fee.

Structured Query Language (“SQL”) – A syntax for querying information from, updating information in, or adding information to a DBMS.

Subversion (“SVN”) – A system for tracking revisions to source code.

Uniform Resource Locator (“URL”) – A specification for how to locate a resource on the Internet. This includes the protocol used, the remote server address, and location on that server.

User Interface (“UI”) – A view, command line, or other portion of an application that allows the user to interact with the application.

Vi IMproved (“Vim”) – A free, open source text editor for UNIX / Linux operating systems.

Virtual Machine (“VM”) – The emulation of a computer which allows a single physical computer to host multiple logical computers.

Virtual Private Cloud (“VPC”) – AWS’s service in which they provide a private container of hosted systems.

Work Order – A request for services to be provided. The work order details the services required, and the time frame for delivering those services.

References

Amazon Web Services, Inc. "Overview of Amazon Web Services." Whitepaper. 2017. 08 2017. <https://d0.awsstatic.com/whitepapers/aws-overview.pdf>.

Bray, T. *RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format*. n.d. 08 2017. <https://tools.ietf.org/html/rfc7159>.

Computer Hope. *What is C++?* 26 04 2017. 08 2017. <https://www.computerhope.com/jargon/c/cplus.htm>.

Django Software Foundation. *Django Overview*. n.d. 08 2017. <https://www.djangoproject.com/start/overview/>.

GitHub, Inc. *Pricing - Git hosting software and tools*. n.d. 08 2017. <https://github.com/pricing>.

Javascript-coder.com. *A Modern Reintroduction To AJAX*. n.d. 08 2017. <http://javascript-coder.com/tutorials/re-introduction-to-ajax.phtml>.

Kissmetrics. *How Loading Time Effects Your Bottom Line*. 28 04 2011. 08 2017. <https://blog.kissmetrics.com/loading-time/>.

—. *What Makes Someone Leave a Website?* 01 09 2011. 08 2017. <https://blog.kissmetrics.com/leave-a-website/>.

Oracle. *What is a JSP Page?* n.d. 08 2017. <http://docs.oracle.com/javaee/5/tutorial/doc/bnagy.html>.

—. *What is a Servlet?* n.d. 08 2017. <http://docs.oracle.com/javaee/5/tutorial/doc/bnafe.html>.

—. *What is MySQL?* n.d. 08 2017. <https://dev.mysql.com/doc/refman/5.7/en/what-is-mysql.html>.

Raggett, David. *Getting started with HTML*. 24 05 2005. 08 2017. <https://www.w3.org/MarkUp/Guide/>.

RAILSGUIDES. *Getting Started with Rails*. n.d. 08 2017. <http://guides.rubyonrails.org/getting_started.html>.

Refsnes Data. *HTML Responsive Web Design*. n.d. 08 2017. <https://www.w3schools.com/html/html_responsive.asp>.

Robert, Kirrily. *Perlintro*. n.d. 08 2017. <http://perldoc.perl.org/perlintro.html>.

Robinson, D. and K. Coar. *RFC 3875 - The Commin Gateway Interface (CGI)*. n.d. 08 2017. <https://tools.ietf.org/html/rfc3875>.

Skerrett, Ian. *Eclipse Community Survey 2014 Results*. 23 06 2014. 08 2017. <https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/>.

Wikimedia Foundation. *C++*. n.d. 08 2017. <https://en.wikipedia.org/wiki/C%2B%2B>.

Woolf, Chad. *FERPA Compliance in the AWS Cloud*. 08 06 2015. 08 2017. <https://aws.amazon.com/blogs/security/ferpa-compliance-in-the-aws-cloud/>.

Worcester Polytechnic Institute. "Undergraduate Catalog 2008 - 2009." 2008. *WPI Registrar's Office.* <https://www.wpi.edu/sites/default/files/docs/Academic-Resources/Academic-Catalogs/ugrad0809.pdf>.

—. "Undergratuate Catalog 2016-17." 2016. *WPI Registrar's Office.* <https://www.wpi.edu/sites/default/files/docs/Academic-Resources/Academic-Catalogs/WPI_UGcat16-17FinalWEB.pdf>.

Appendix A: List of Use Cases

End-user Login
User Login Intercept
View Main Menu
View List of Events
Add an Event
View Event Details
Edit Event Details (Requestor)
Edit Event Details (LnL Staff)
View List of Events Pending Approval
Approve an Event
Enter Crew Hours
Enter Crew Chief Report
Bill Event
Mark Event as Paid
View List of Locations
Add a Location
View Location Details
Edit Location Details
View List of Clients
Add a Client
View Client Details
Edit Client Details (Client)
Edit Client Details (LnL Staff)
View List of Users
Add a User
View User Details
Edit User Details (User)
Edit User Details (LnL Staff)
View List of Projectionists
Add a Projectionist
View Projectionist Details
Edit Projectionist Details
View List of Equipment
Add Equipment
View Equipment Details
Edit Equipment Details
Generate a Meeting Notice
Enter a Semester of Films

Appendix B: Example Use Cases

End-user Login

End Objective:	The user is logged in
Created by:	Matt Brennan
User/Actor:	Any end-user
Frequency of Use (1-5):	5

Preconditions
N/A

Basic Flow	
Step	**Actions**
1	User navigates to log in page
2	User enters their username and password and submits
3	The system validates the user's credentials
4	The system displays the page which the user was attempting to access (main menu if none)

Exception Flow: Invalid Credentials	
Step	**Actions**
4	The system displays the login page containing an error

Post conditions
Successful: The user is logged in Unsuccessful: The system state is unchanged

Other Notes
N/A

Add an Event

End Objective:	The user has submitted an event
Created by:	Matt Brennan
User/Actor:	Any end-user
Frequency of Use (1-5):	2

Preconditions
The user is logged in

Basic Flow	
Step	**Actions**
1	The user selects “New Event”
2	The system confirms the user has access to create events
3	The system displays the add event form
4	The user enters and submits the information
5	The system validates the entered information
6	The system adds the event to the database
7	The system sends relevant notifications
8	The system shows a confirmation page

Exception Flow: Access Denied	
Step	**Actions**
3	The system displays an “Access Denied” error

Exception Flow: Invalid Input	
Step	**Actions**

Exception Flow: Invalid Input	
Step	**Actions**
6	The system re-displays the event add form with an “invalid information” error

Post conditions
Successful: The event has been added to the database Unsuccessful: The system state is unchanged

Edit Event Details (Requestor)

End Objective:	The event details have been updated
Created by:	Matt Brennan
User/Actor:	Event Requestor
Frequency of Use (1-5):	1

Preconditions
The user is logged in

Basic Flow	
Step	**Actions**
1	The user clicks “Edit” from the View Event Details page
2	The system confirms that the user has access to edit the event
3	The system displays an editable page
4	The user updates the information and submits the event
5	The system verifies that the data is valid
6	The system updates the information in the database
7	If the event was already approved, the event is marked as pending approval
8	A notification is sent to relevant users
9	The user is brought back to the View Event Details page with a confirmation message

Exception Flow: Access Denied	
Step	**Actions**
3	The system displays an “Access Denied” error

Exception Flow: Event Doesn’t Exist

Step	Actions
3	The system displays a “No Such Event” error

Exception Flow: Invalid Input	
Step	**Actions**
6	The system re-displays the event edit form with an “invalid information” error

Post conditions
Successful: The event data is updated in the database Unsuccessful: The system state is unchanged

Edit Event Details (LnL Staff)

End Objective:	The event details have been updated
Created by:	Matt Brennan
User/Actor:	Crew Chief, LnL Exec
Frequency of Use (1-5):	2

Preconditions
The user is logged in

Basic Flow	
Step	**Actions**
1	The user clicks “Edit” from the View Event Details page
2	The system confirms that the user has access to edit the event
3	The system displays an editable page
4	The user updates the information and submits the event
5	The system verifies that the data is valid
6	The system updates the information in the database
7	The user is brought back to the View Event Details page with a confirmation message

Exception Flow: Access Denied	
Step	**Actions**
3	The system displays an “Access Denied” error

Exception Flow: Event Doesn’t Exist	
Step	**Actions**
3	The system displays a “No Such Event” error

Exception Flow: Invalid Input	
Step	**Actions**
6	The system re-displays the event edit form with an “invalid information” error

Post conditions
Successful: The event data is updated in the database Unsuccessful: The system state is unchanged