The Princeton Computer Science LabTAs and How an Academic Can Learn to Code

**Alexander Daifotis, B.S.E. Computer Science, Class of 2014**

**Advisor: Brian Kernighan, Professor of Computer Science**

**COS 497, Fall 2013 – Senior Independent Work**

Abstract: In this report, we discuss the goals, accomplishments, and conclusions of the research conducted this semester in the fulfillment of COS 497 and departmental requirements. This independent work consisted primarily of designing, building, and maintaining a newly developed online queuing system for use during the introductory computer science tutoring sessions sponsored by the department, in which undergraduate TAs are available to assist with the students’ coursework. In addition to presenting the reader with the design and use of this system, we will also discuss the lessons learned from its development that are relevant to an academic who has little experience with real world web development. We also relate the pedagogical insights derived from statistically analyzing the data generated by the tool.

Table of Contents

Introduction 3

Motivations and Goals of Research 4

Improving the Day-to-Day Flow of the Lab 4

Collect the Data and do the Analysis Necessary to Gain Pedagogical Insights 5

Learning How to Build and Manage 5

Selection of Web Stack and Platform 5

Design 5

Backend 5

Frontend 5

Observed Usage 6

The Problem Solving Mindset 6

The Mentality of Agile Design 6

When is Technology *Not* the Solution? 6

Building Something That Will Last 6

Data Analysis 6

Conclusion 6

# Introduction

Like many of its peer institutions, Princeton offers various services for academic tutoring outside of class in many introductory courses. For the Introductory Computer Science Sequence (IntroCS), the department runs tutoring sessions 7 days a week that are organized and staffed by undergraduates. This sequence consists of three classes that are meant for majors, and one that is meant for non-majors and those looking for a broader and gentler introduction to the field. The three major courses are COS 126 (Introduction), COS 226 (Data Structures & Algorithms), and COS 217 (Systems Programming). The non-major course is COS 109 (Computer In Our World). Henceforth, we will refer to the classes by their course codes rather than the colloquially naming for the purpose of consistency.

As the chart shows, enrollment over the last four years as accelerated at a staggering pace.[[1]](#footnote-1)[[2]](#footnote-2) COS 126 in particular has been a major driver of these increases, though even the 200 level classes have seen enrollments surge. Along with this increase have of course come increasing demands upon the undergraduate tutors, known in the department as LabTAs, to provide assistance.

As the manager of this program over the last two semesters, I can tell you from first hand experience that it is a bit of a handful. The Head TA is responsible for hiring, coordinating with instructors, staffing the shifts, and many other administrative tasks. And, being the engineering student that I am, I have spent a considerable amount of time thinking about ways of improving and simplifying the way the program is run. Some of these have been simple improvements—putting the schedule in a GoogleDoc instead of writing the HTML for a website, surveying more often, consolidating basic information for the TAs, etc. The main portion of this independent work, however, was a project a bit grander in scale.

One thing that has always been a particularly annoying is the fact that I cannot be in two places at once. The reason this frustrates me is that I cannot monitor what happens in the lab in any more than a semi-regular basis, since that would entail me being there every night of the week. What occurred to me, was that like some other schools, a way of getting the next best thing would be replacing the ad hoc system of asking for help in the lab with an electronic help queue. Historically, if the lab was not busy, a student would just flag down a free TA for assistance. If a lot of people started asking for help, somebody would simply start a list of names on the blackboard that TAs would check off as they were attended to. This is a perfectly functional solution to the problem, but it obviously does nothing to help with the problem of actually gaining insight into what is occurring in the lab. Furthermore, from a UI point of view, a webpage is a bit nicer of a solution.

Thus, for this independent research, I have implemented just such a service. In the rest of this report, I will begin by describing in more detail some of the motivations and fringe benefits of undertaking this project. Then, I will discuss the process of designing, building, and iterating on the project. After this exposition, I will move onto a discussing the lessons learned from both the process of building a design and from the data collected over the semester. Finally, I will end with a brief conclusion and ideas for further work.

# Motivations and Goals of Research

As was briefly mentioned in the Introduction, there are a number of related, but ultimately separate motivations and goals that were part of this research project. In this section, I outline these different targets in more detail. Despite the myriad of objectives, I am happy to report that they were all accomplished in degrees ranging from more than expected to truly thorough. As we move through this report, I will describe the successes in more detail.

## Improving the Day-to-Day Flow of the Lab

The most obvious goal of this project is actually improving the daily experience of both the students and TAs in the lab. In addition, I also wanted to help myself, and future Head TAs, get a better grasp on how things were going and provide a way to check up on individual students and TAs. This portion of the project was the primary driver of UI decision-making. In order for this primary goal to be accomplished, it is critical that I created a simple and intuitive system that represented a step forward from what was currently being used (the blackboard).

## Collecting the Data and do the Analysis Necessary to Gain Pedagogical Insights

Although the primary goal, and true barometer of success for this project is ultimately whether or not the system is used, I think the more important and more interesting benefit of the system is the data that was collected. On a practical level, a lot of what the Head TA does, particularly with regard to staffing, is very much based on anecdotal experience. In addition, there have been many outstanding questions in the department regarding the usage of the lab by course and type of student, which have again only been addressed anecdotally. By seeing this system come into use, many of these practical questions can be answered.

In addition to these basic data points, many other insights can be gained from collecting usage data. For instance, one might ask what the distribution of TA work is as measured by students assisted, which students account for most of the requests, how long it takes to help different students, etc. We will return to these questions in detail later on in the discussion of results.

## Learning How to Build and Manage

One of my major motivations in taking on this project was using it as a forcing mechanism for teaching me to learn how to take on projects that face computer scientists in the real world. In my time at Princeton, I have obviously done the gambit of academic work, which has included a good deal of writing code. Taking lots of coursework though, seems to not translate particularly well, at least directly, to doing real things. It should be noted that this is not simply a matter of learning relevant technologies, but also, and I think more importantly, a matter of cultivating a way of thinking and broader mental skills than are derived solely from academia. In addition to personal growth, however, I will seek in this report to relate to others who are in a similar position the important lessons learned from this exercise. In a way, portions of this report can be seen as a sort of cookbook for transitioning from highly academic and contained environments to working in the broader and less defined space that is your typical product.

# Selecting the Web Stack

The first, and perhaps most daunting step, for the enterprising engineer looking to start making products that face actual users, is the myriad of decisions that need to be made about the seemingly endless numbers of platforms and technologies that are available for use. Discussing the full scope of all industrial programming practices would be far beyond the scope of this paper, so I will focus on what I did, and what seems to be a very typical and practical first step: building a full-featured website that looks good with the minimal amount of fuss.

So where does one even start? Although it may seem like a rather naïve question to programmers who have spent a lot of time in industry, for a lot of people who know plenty about writing code, this is actually a very reasonable question. Thus, before actually diving into the tools that I would suggest other beginners should use, I will first discuss what sorts of things that one needs in the first place.

## What is a Web Stack Anyway?

*Web Stack* is a term that gets thrown around a lot, but what does that actually mean? The web stack is just a description of all the technologies that were used at the different layers of the app.



I think the diagram above illustrates quite well why the set of tools/technologies used for the project is referred to as a *stack*. As we can see, starting with the most basic level, the hardware, we layer each piece of technology upon the underlying components. More importantly though, it shows just why it’s so intimidating when we try to take the step. Not only are the decisions at one layer of the stack often complicated enough, but often the decisions you make at one layer have important downstream impacts. As we will see, many new IaaS (Infrastructure as a Serivce) and PaaS (Platform as a Service) providers handle a great deal of this difficulty, let us take a moment now to discuss the various layers in brief so that we have a solid fundamental understanding going forward.

The first, and most basic layer is the **Hardware** layer, which is exactly what it sounds like. If you want to run software, you need a computer to run it on. These days, it’s almost always an x86 server. The main thing to worry about here is how powerful we need that server to me. Next is the **Operating System**, which again, is exactly what it sounds like. For web development, the most common choices are some flavor of Linux or Microsoft Server. The choice of Operating System has a great deal of impact on what **Hosting and Database Software** you end up selecting as well. For Linux, the standard choice is Apache with MySQL. For Windows, the choice is almost always Microsoft IIS with Microsoft SQL Server. In any case, these two pieces of software are the basis of your web application. Finally we arrive at what programmers interact with the most, and hopefully spend most of their time working on, the actual language and frameworks they leverage to build their application. I will address this in detail in the next section. Finally, we get to the finally set of tools we need, those for programming the client side of the app, that is, what actually runs in the users’ web browser. This is almost always some combination of JavaScript, CSS, and HTML, but there are still many options for frameworks that can be layered on top of the basics that are widely used.

## How to Pick the Web Stack

Now that we have at least a cursory understanding of the sorts of things we need, the obvious next step is picking from the nearly limitless array of options what we actually want to use. Before making this decision though, it is worth taking the time to figure out what we are trying to optimize for. Obviously we would all like our web stack to be the simplest, fastest, most elegant, etc., etc., but like most things in the world these decisions inevitably involve some amount of compromise. Given the type of person that I am have identified myself as (the experienced programmer but inexperienced in real world application), let me suggest a reasonable hierarchy of concerns below.

1. **Simplicity –** The step in is always the hardest, and making sure the stack we end up using is very simple goes a long way toward making it easier. Learning new technologies is very time consuming, so whatever we decide to learn should be as simple as possible.
2. **Low Startup Costs** – Again, especially since we are inexperienced with a lot of these technologies, it becomes a real problem if whatever stack we choose involves a lot of complex installs and configurations. Ideally, whatever we choose should be mostly a download and run sort of deal. It should work out of the box.
3. **Low Maintenance Costs** ­– Doing a project in your free time is a much different animal than running a website full-time, as such we need to preserver as much of our time as possible for actually building the features as opposed to just make sure the proverbial house is not burning down.
4. **Good Documentation and Wide User Base** – As novices who do not have time to waste, good documentation and plenty of places to go for help save you a lot of time, trust me.

Just to be explicit about the trade-offs we should be prepared to make, note that we are not optimizing for performance, freedom to select individual components, ability to handle complexity, or the ability to select from different design paradigms. I would argue that given our position, these trade-offs are all well worth making if the benefit of doing so is having something that is easy to understand and just works.

## Actually Choosing the Stack

Now that we agree on the sort of features that we are looking for in our stack, it makes picking it a lot easier since we can quickly narrow down the field of options. We can immediately rule out the option of running everything yourself. As somebody who has previous done it, I promise you that the advent of IaaS and PaaS is one of the greatest things ever to come to commercial software development. Actually handling our own software would thoroughly violate all 4 of our guiding principles from above. With that decided, there are many providers out there, but in the spirit of principle (4), there are a few that are nearly completely dominant. In terms of IaaS, that is, services that provide essentially barebones machines with an OS installed and possibly some hosting software, Amazon AWS[[3]](#footnote-3) is the biggest game in town, though the Google Cloud[[4]](#footnote-4) is becoming increasingly competitive. In terms of PaaS, the two major options are Heroku[[5]](#footnote-5) or Google App Engine (GAE)[[6]](#footnote-6). These differ from the IaaS solutions because they cover the entire server side of the stack including development frameworks.

It is at this step, that in my humble opinion many of those just getting started make their biggest mistake. Rather than trying to move as far up the stack as possible by picking a PaaS, they decide that having more control of their deployment is worth the trouble of dealing with a barebones virtual machine. *It isn’t*. After using GAE, I can confirm from experience that unless your applications requires some very specific frameworks/libraries or a lot of high performance operations, you can do basically everything you need trivially and even within GAE there are plenty of options. Although I did not end up using Heroku, my research indicates it is still relatively flexible, like GAE, and is much easier to use than just AWS. Citing our guiding principles, I will once again strongly advise that you do not just use AWS. I will continue arguing for this as I explain in more detail the usage and benefits of the GAE platform.

Thus, our decision now comes down to selecting between Heroku and GAE. As you are likely expecting at this point, I once again optimized solely for the four principles outlined before and decided that GAE was what made sense. The install is trivial, the documentation is nearly uniformly excellent, it comes preconfigured correctly out of the box, etc. I believe Heroku also does a good job with all of these things as well, but the bottom line is that it seems nearly impossible to beat GAE for easy of use and learning curve. As long as you have Python installed on your development machine, you are essentially good to go after just installing the SDK from Google.

After selecting GAE, all that is left to decide upon is what GAE platform we want to run on (they give you a choice of databases and languages) and what client side tools to use. Google gives you the choice of Python, Java, PHP, and Go. I would suggest going with Python because it is a flexible and simple language and comes with the equally simple and easy to use webapp2 web framework. For the database, I decided to go with their schemaless NoSQL datastore, though they also offer MySQL that is likely simpler. I used the NoSQL mainly because I wanted to learn about that technology. In retrospect, however, unless there is a compelling reason (like my curiosity), I would suggest going with the standard MySQL database, again, for simplicity. For the client, I would suggest using jQuery[[7]](#footnote-7) for your JavaScript and Bootstrap[[8]](#footnote-8) for HTML/CSS. jQuery is a widely used and easy framework to get started with. Bootstrap may be what most excited me though on the frontend, since it makes setting up a basic layout an absolute breeze and makes things look good even if you have not the faintest sense of design like myself.

Long story, short…



# Design and Implementation

With the selection of the web stack components out of the way, the next step for any project is creating the mental visualization of what the product should be like, then a plan for how to realize that vision, and finally doing the implementation itself. In this section, we discuss the actual building of the Lab Help Queue, which was the product of the research.

## UI Design and Feature Set

## Backend

Awefwf

## Frontend

Awefwaf

Wefwef

# Thoughts on Various Technologies and Their Features

GAE, Bootstrap, NoSQL, jQuery, discuss what was nice, not so nice, concluding thoughts, ability to iterate, etc, learnability

# Observed Usage

Maybe this is more relevant to discuss with Data Analysis? But this is more of a UI and feedback perspective I suppose. Might not be enough here to make it a section.

# The Problem Solving Mindset

Wefwefwefwef

## The Mentality of Agile Design

Be fast, be simple, avoid complicated things, focus on the goal

## When is Technology *Not* the Solution?

Understand the costs of technology are great, time, maintenance, learning, etc. You have to ask, is this getting you close to the goal?

## Building Something That Will Last

Difference between a hack and a project, not a one-time thing. How to ensure things continue being used that are meant for posterity. What are important considerations?

# Data Analysis

Many subsections here, break out further when we come to it

# Conclusion

Blah blah awesome blah

1. Data for the Spring 2014 semester is as of 12/18/2013. Final numbers were not attainable at the time of submission. Note they are likely inflated since students typically drop. [↑](#footnote-ref-1)
2. The seasonality of enrollments reflects the fact that typically most engineers take COS 126 in the Spring, most majors take COS 226 in the Spring, and then COS 217 the following Fall semester. [↑](#footnote-ref-2)
3. <http://aws.amazon.com/> [↑](#footnote-ref-3)
4. <https://cloud.google.com/> [↑](#footnote-ref-4)
5. <https://www.heroku.com/> [↑](#footnote-ref-5)
6. <https://cloud.google.com/products/app-engine/> [↑](#footnote-ref-6)
7. <http://jquery.com/> [↑](#footnote-ref-7)
8. <http://getbootstrap.com/> [↑](#footnote-ref-8)