5-3-2019

Student Engagement Tracking & Behavioral Analytics Software

Steven Christopher McClurg

Student Engagement Tracking & Behavioral Analytics Software

By

Steven Christopher McClurg

An Honors Capstone

Submitted in partial fulfillment of the requirements

For the Honors Diploma

To

The Honors College

Of

The University of Alabama in Huntsville

1 May 2019

Honors Capstone Director: Dr. William Wilkerson

Dean, Honors College

S. Chris McClurg 1 May 2019

Student Date

Director 3-5-19 Date

Department Chair Date

Honors College Dean 3-5-19 Date

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.
In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the chair of the Department, Director of the Program, or the Dean of the Honors College. It s also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Steven Christopher McClurg
Student Name (printed)

S. Chris McClurg
Student Signature

1 May 2019
Date

Student Engagement Tracking & Behavioral Analytics Software

By

Steven Christopher McClurg

An Honors Capstone

Submitted in partial fulfillment of the requirements

For the Honors Diploma

To

The Honors College

Of

The University of Alabama in Huntsville

1 May 2019

Honors Capstone Director: Dr. William Wilkerson

Dean, Honors College

Student	Date
Director	Date
Department Chair	Date
Honors College Dean	Date

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.
In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the chair of the Department, Director of the Program, or the Dean of the Honors College. It s also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

_____
Student Name (printed)

_____
Student Signature

_____
Date

Table of Contents

1 Abstract

In an institution of higher learning, knowing what factors correlate to the academic success of students incredibly useful and actionable data. Finding the correlations in a sea of possible factors, however, can be very difficult, especially using traditional data models. This Capstone Project Report explains the difficulties in looking for connections in traditional data. It goes on to provide an abstract software architecture that can be used to implement a technical solution which helps alleviate those difficulties. All required background knowledge is summarized and included.

Additionally, a specific implementation of the abstract architecture was implemented, generating a proof-of-concept prototype for the design pattern. Limitations and developmental difficulties faced in the implementation are documented. Finally, additional functionalities that would extend the agency of an implementation of the provided design pattern are discussed. It is my hope that either this prototype can be used as a cornerstone for a robust, useful application, and/or that the abstract design pattern provided can help future software development students change the way they think about data.

2 Introduction

2.1 Problem Statement

While details and findings vary greatly, most literature on the subject of student development in higher education theory arrives at two heavily simplified conclusions:

1.) Involved students are more likely to be academically successful

2.) Different types of involvement have varying degrees of impact on student success

3.) The types of involvement which most strongly correlate to academic success varies between institutions

These conclusions imply that, within a given institution, specific types of involvement will correlate to student success. The downfall of this principle, however, is that the data required to see these trends is almost never available in a format that lends itself to viewing them. Since the trends are difficult to spot, they aren't often analyzed in detail.

2.2 Project Overview, Objectives, & Scope

This Capstone Project report details my proposed and partially-implemented solution to create a system that allows users to more easily look for trends. Specifically, the project, at a high level, is to research and create a software architecture for an application that allows the user to do the following:

1.) Easily collect data from traditional spreadsheet-oriented data sources

2.) Connect the data from the disparate data sources

3.) Analyze impact of newly connected data on student academic performance and retention.

In addition to a complete software architecture, the project also includes a partially-implemented prototype user interface with specific utilities hard-coded to allow for use by the Honors College of the University of Alabama in Huntsville.

3 Background

Before the full software architecture plan can be discussed, background on specific utilized technologies is required. The proposed architecture requires an understanding of Graph Databases, Client-Server Architectures, and Reactive Web Design fundamentals.

3.1 Graph Databases

Graph databases are at the core of the proposed application architecture. However, to understand the advantages of a graph database, it is crucial to understand relational databases first.

3.1.1 Relational Databases

Most people, whether they realize it or not, are very accustomed to working with Relational Databases (RDBs) or spreadsheets, which are RDB oriented. Each row in a spreadsheet or RDB table represents what is known as a record. A record is one paradigm for storing information: it contains a set of fields, the values of which may or may not be duplicated in other records. Typically, an RDB will contain multiple tables, which represent a collection of records containing the same fields. One table in the RDB is the primary table, meaning the values it contains are unique, and often referenced by other tables. All non-primary tables will contain non-unique information but will be indexed by a reference to a primary table.

For example, a traditional RDB for a university might contain two tables, a student table and a financial transaction table. The student table would be a primary table; each record would contain information regarding a specific student, and there would only be one record per student. The financial transaction table would be non-primary table. Each record in the table would represent a financial transaction and would be indexed by the corresponding student's ID. A

student can have multiple transaction records in the transaction table, meaning that the records are non-unique.

Connecting data in an RDB requires utilization of a “Join” statement. Essentially, to find connections in data, the first step is to process each record in the non-primary table one at a time. For a simple sample case, suppose a user ran a query of an RDB which returns a student’s name, email, id, and the details of a financial transaction. To produce the proper output, the RDB has to scan each record in the financial transaction table. For each record in the table, the RDB will find the student ID used to reference the primary table. Then, it has to scan the primary table to retrieve the student’s name and email based on the ID found in the financial transaction table. The RDB then joins the two records together (hence the term “Join statement”) and returns those values. Joining two tables is a fairly fast operation due to the algorithms used to execute the join statement. Essentially, the RDB can sort the transaction table before it begins the join operations. That way, all the matching student IDs are grouped together and the value from the primary table only has to be looked up once. However, that isn’t possible when attempting to join 3 tables together. The page has to scan each record in one table, find a match in the second table, then scan and find all the matches in the third table. Then, it backs up, finds the second value in the second table, and scans to find all the matches in the third table. The more tables that are joined together increase the time and difficulty to execute a query by an exponential factor. In the context of the higher education development principles discussed in the introduction, this makes looking for the correlation between arbitrary relationships that could span as many as 8 or 9 tables very difficult and complicated. Not only would it be difficult to write the queries in the first place, but many queries that span more than 6 tables run so slowly that RDBs never finish and return the results.

3.1.2 Graph Databases

Graph Databases simply utilize a different manner of representing data. If Relational Databases can be thought of as spreadsheets, Graph Databases (GDBs) can be thought of as pushpins and strings in a cork board. For the example from the Relational Database section, you can imagine that there are two types of pushpins: students and financial transactions. For every student, there is exactly one student pushpin. Every time that student makes a financial transaction, a financial transaction pushpin is made. Instead of being stored separately with the student's lookup information like in an RDB, a graph database directly ties a string between the student pushpin and the financial transaction pushpin. A single student pushpin might have many, many strings tied to it, but to find the associated data you only have to follow the strings.

To generate a list of students and their financial transactions in a graph database, you only need to execute a query that searches for a pattern of pushpins and strings (the technical names of which are nodes and relationships, respectively). The query would consist of a pattern that simply states, "Find a student node tied to a financial transaction node". The GDB would produce a list of all the student nodes, just as the RDB would initially. However, instead of having to search a second table for matching values, a GDB only needs to look at the "strings" (the relationships) for each node. The relationship contains the exact location of the related financial transactions: no searching required.

The lack of searching becomes even more significant when searching for data that would be connected across more than 2 tables in an RDB. Where as an RDB has to search each connected to table to see if a match exists, a GDB only needs to check the relationships of a given node to reach the same result. As a result, the complexity of connecting different types of

data together *does not* grow exponentially, meaning that it is significantly easier and faster to search for arbitrary patterns in connected data.

The disadvantage to graph databases is size. The speed at which a GDB can find all matches is orders of magnitude faster than an RDB, but that is because each node knows where the related nodes are initially. This is possible by storing the memory addresses of all related nodes internally within each node. If one node is connected to seven others, it has to store seven memory addresses internally. Assuming all 8 nodes are connected to each other, the nodes contain a collective 64 memory addresses. As opposed to search time growing exponentially, graph databases grow exponentially in memory used.

3.2 Client-Server Architecture

The second technology utilized is the design pattern of a Client-Server Architecture, a generalized structured often used in computer science when designing data-intensive applications. The name is largely self-explanatory: client-server architectures consist principally of two components: a client and a server.

A client simply refers to a user's computer. It means the machine on which the user interface of an application is running. The distinction between client-server architectures and traditional applications is where the data that powers the application lives. In client-server architectures, the data that powers the application is stored remotely in a server, and is fetched in small, bite-sized pieces as needed by the client.

The difference between a stand-alone application is exactly modelled by two popular word processing applications: Google Docs and Microsoft Word. Word is representative of a stand-alone app. All of the relevant data is stored somewhere on the client in a file. The

application can open the file, perform some operations, and then save the file. Additionally, the file has to be manually shared to be viewed on any other computers.

Google Docs, on the other hand, is a client-server word-processor. When the application isn't running, the file is not accessible on the client machine at all. It requires a constant connection to view the file. Several properties of a client-server architecture form the standard features Google Drive users have come to expect. The first is "auto-saving". Google Docs operates by alerting the server every time a button is pushed. The actual file, stored on googles servers, is immediately updated every time. This also means that multiple users can use the file. As long as the server is able to process multiple button presses from multiple users, many people can collaborate because the one file stored is updated simultaneously. Client-server architectures are useful in any situation in which either a.) the volume of data processed by the application is too large for a user's computer, or b.) the data needs to always be automatically synchronized between multiple computers.

3.3 Web Reactivity Frameworks

In web design, there are two predominant types of website: server-rendered static websites, and client-rendered reactive websites. This refers to how the instructions given to your computer about how to display a website are created. At its core, what a user sees when viewing a website is a browser converting some instructions from a web server to a visual format. The instructions which dictate how that data should be displayed is stored in a format most people are familiar with, known as HTML. True HTML is static; the data stored in HTML cannot be changed. Sure, it has some fancy tricks like changing colors or showing menus when a mouse hovers over something or clicks something, but the data stays the same. It is essentially hard coded – for example, a number in an HTML document can never be changed to another number.

Obviously, that doesn't mean websites are all hard-coded. Somehow, websites can handle changing information. Think about a website for grade management. Virtually all grade management websites have a page in which students can perform "What-If" analysis. A student can plug in theoretical grades for various assignments and will immediately see how their overall grade is impacted. How is this possible if the numbers you're seeing are hard-coded in HTML, never to be changed?

The answer is a web reactivity system. It's true that raw HTML cannot be updated. If the HTML is made on the server and sent to the client's web browser without code, it is known as server-side static rendering. For that website to display anything other than that HTML, the client's machine will have to talk to the server and request a new block of HTML instructions. However, a web server can choose to send code to a web browser instead of HTML. The other option for web design is to send a reactivity system to the browser. Essentially, a web reactivity system is a piece of code that creates HTML dynamically. For example, a reactivity system can have a "template" for displaying student grades. This template will resemble HTML, but with one major difference: instead of real values for things like a student's name, it will have a placeholder value. Then, the reactivity system can plug in values where those placeholders are to create HTML. This is significant, since it means that the web browser doesn't need to wait on new HTML when a value changes. Since it already has a template, a web reactivity system can just plug the new values into the template, and the new HTML is ready to go. In the case of the "What-If" analysis, the reactivity framework just plugs the values the student enters into the template to show changes in the overall grade. With enough work, it is possible to use a reactivity framework to create what is called a single-page application, meaning that all the templates needed to produce any content on the entire website is sent in one batch the first time

the first page loads. After that, a single page application only needs values from a server to plug into templates, never requiring any pre-made HTML be created by the web server.

4 Software Architecture

The following section details the proposed architecture for an application that both satisfies the theoretical project objectives and meets the needs set forth for the prototype created for the Honors College.

4.1 Requirements

Aggregating the project objectives and the customer requirements set forth for a prototype produces the following list of requirements.

Customer Requirements	System Requirements
1.0 The application will be available on both Windows and Mac computers	1.1 The application will be available in a .exe file for Windows and a .app contained within a disk image for Mac
	1.2 The applications should use an identical interface – a user of one version should know the exact steps for use on the other
2.0 Data will be synchronized across all computers and stored securely	2.1 Data will be stored in a stand-alone server
	2.2 Server will implement BASIC security protocol to ensure FERPA compliance
	2.3 Information committed to the database will be immediately available to other computers connected to the server
3.0 Be able to import bulk data sets from Banner using spreadsheets	3.1 The system will support uploading graph data from a spreadsheet
	3.2 The system will support importing multiple node types from one spreadsheet
	3.3 The system will recognize and automatically parse data in spreadsheets using Banner field names
4.0 Execute a student success query suite	4.1 System contains a collection of executable queries whose results reveal correlative trends between involvement types and academic performance.
5.0 Provide tools to track attendance at student events	5.1 Provide Excel spreadsheets that parse A-Numbers from Charger Cards to track event attendance

4.2 Abstract Software Architecture

The proposed generic software architecture details a possible solution to finding correlations in arbitrarily related data.

4.2.1 Data Management

The data for the application will be stored in a stand-alone graph database, which will be completely decoupled from any user interface. A graph database is the obvious choice for the proposed application for the following principle reasons:

1.) The time to execute scales linearly with the number of additional types of connected data searched in a single query, rather than increasing exponentially like a relational database.
2.) Trends can be searched for using arbitrary patterns of connected data, rather than requiring meticulously crafted join statements merging specific data from specific tables.
3.) The definition on how types of data are connected can be changed dynamically, while new fields cannot be added to relational database records.

In addition to the added performance benefits and the ability to implement the queries required, having a stand-alone database implies a client-server architecture. A client-server application will automatically satisfy the requirement that data be synchronized between all computers using the application, since the data store is completely decoupled from the user interface.

4.3 User Interface

The design for the user interface will utilize a JavaScript reactivity framework to create a single-page web application. This is an ideal design, at is contains complete instructions for rendering pages given only data values. All the other logic is contained with the templating instructions. This is ideal for a client-server architecture, as all data values are by definition communicated over the network. This allows for the user interface to adaptably and

automatically re-render anytime new data is received from the server, as well as already having data formatted as the database expects when the client needs to add or update values in the server.

5 Implementation Details & Challenges

5.1 Tools and Frameworks

After designing an abstract software architecture, specific technologies were selected to use in the development of a prototype application.

5.1.1 Graph Database: Neo4j

Graph Databases as a whole are an emerging new technology in computer science. Relational databases have been the long-prevailing database type. As the whole field of graph databases has emerged, Neo4j has emerged as the vanguard product, and the best choice for this project. This stems not only from a solid base implementation of a graph database, but also from the detailed developer resources available and the extensive support for community-driven plugins that simply development and client-server communication. More details under "Communication protocol".

5.1.2 User Interface: Electron-Vue SPA with Multi-Window Support

The JavaScript Reactivity Framework selected is called Vue.js. Vue is a web development framework that is organized into modules, which is reflective of the traditional Object-Oriented Programming paradigm of software development. Vue is particularly effective at creating single page applications. This framework was, in part, chosen because of pre-existing knowledge of the framework. While other frameworks could have been nearly identically effective in the creation of a single-page application, using this framework shortened development delays and did not incur any penalties compared to the other two popular frameworks, React & Angular.js.

A reactive single-page web application lends itself well to a client-service architected application. However, the requirements specify that the prototype is to be delivered as a desktop

application, available on both Windows and Mac computers. To achieve this, the single page web application will be packaged using a tool called Electron. Electron essentially turns a website into a desktop application that can be compiled on any platform. This gives the added advantage of already having the web reactivity framework locally with the application, meaning that there is no need to host the service at a URL as a website.

Electron also adds an additional piece of functionality to a web application that wouldn't normally available: multi-window support. Using a tool called electron-window-manager, the prototype application is able to utilize multiple windows to simplify user work flows.

5.1.3 Communication Protocol

The graph database Neo4j is very robust in its ability to model and query connected data. It is highly adaptable and can execute searches on connected data in milliseconds that would take hours to execute using a relational database. The biggest limitation on Neo4j, however, is the lack of specification on default query return values. Essentially, it is very easy to write a query to execute in Neo4j, but the format in which it returns data is complicated and bogged down with unnecessary data. To make the return values more concise and consistent, I elected to utilize a middleware communication protocol called GraphQL. GraphQL functions by creating a "contract" between the user interface and the Neo4j server. It utilizes GraphQL Queries, a request for data from the Neo4j database. Functionally, GraphQL converts its own queries into Neo4j queries once they arrive at the server. In the process, however, it also creates and sends requirements about what data is returned and how it is formatted. Additional GraphQL code running on the Neo4j server receives the GraphQL query, converts it to and executes the Neo4j query, and then formats and returns the results as dictated by the GraphQL query. Essentially, it simplifies the process of guaranteeing that data requests meet the developer's expectations.

5.2 Project Difficulties

A number of difficulties did arise over the course of the project, stemming principally from two areas.

The first problem that arose during development was lack of proper project planning. While a timeline was provided with the initial proposal for this project, the estimates for delivery were very rough estimates at best. Approximately a month after the project proposal was submitted, I was taught some basic project management estimation principles as part of the course “Management Systems Analysis”.

Stemming from an unrealistic project management plan was insufficient dedication of time to research and development tasks, resulting in insufficient dedication of time to the project as a whole. Based on the lack of documentation, I can find no evidence of anyone using a multi-windowed application in combination with Vue.js, so I had to develop logic to make the application functional myself. Additionally, and more significantly, I can find no online documentation to support that any developers have publicly made applications in Vue.js that use GraphQL to query a Neo4j database. While the prototype demonstrates that this architecture is both simple and efficient, the complete lack of resources for combining all three components together created significant delays in project development. This (coupled with delays in getting model data from the Office of Institutional Research) delayed development even more. Completing even the proof-of-concept application was only achieved through several all-nighters. The importance of proper project velocity estimation from Management Systems is not a project I shall take for granted when dealing with future projects, especially in regard to research and development related tasks.

6 Future Work & Possible Project Supplements

The abstract architectural solution proposed is one possible approach to developing a technical solution which bypasses the restrictions limiting the application of the higher education principles in action. Specific technologies aside, this solution could be given to a team of students for a Senior Design project, and the prototype application would likely be far more fully implemented that the prototype I was able to generate as an individual developer in a fraction of a semester. The generated prototype, however, does serve as a proof-of-concept application. A user interface was successfully generated, which compiles successfully on both Windows and Mac. The application successfully implements and utilizes GraphQL to query a Neo4j database from a Vue.js application, proving that the integration of all of these components is possible despite the lack of documentation online. The implemented executable schema, included in Appendix A, allows for the execution of all the required queries.

6.1 Additional Functionality

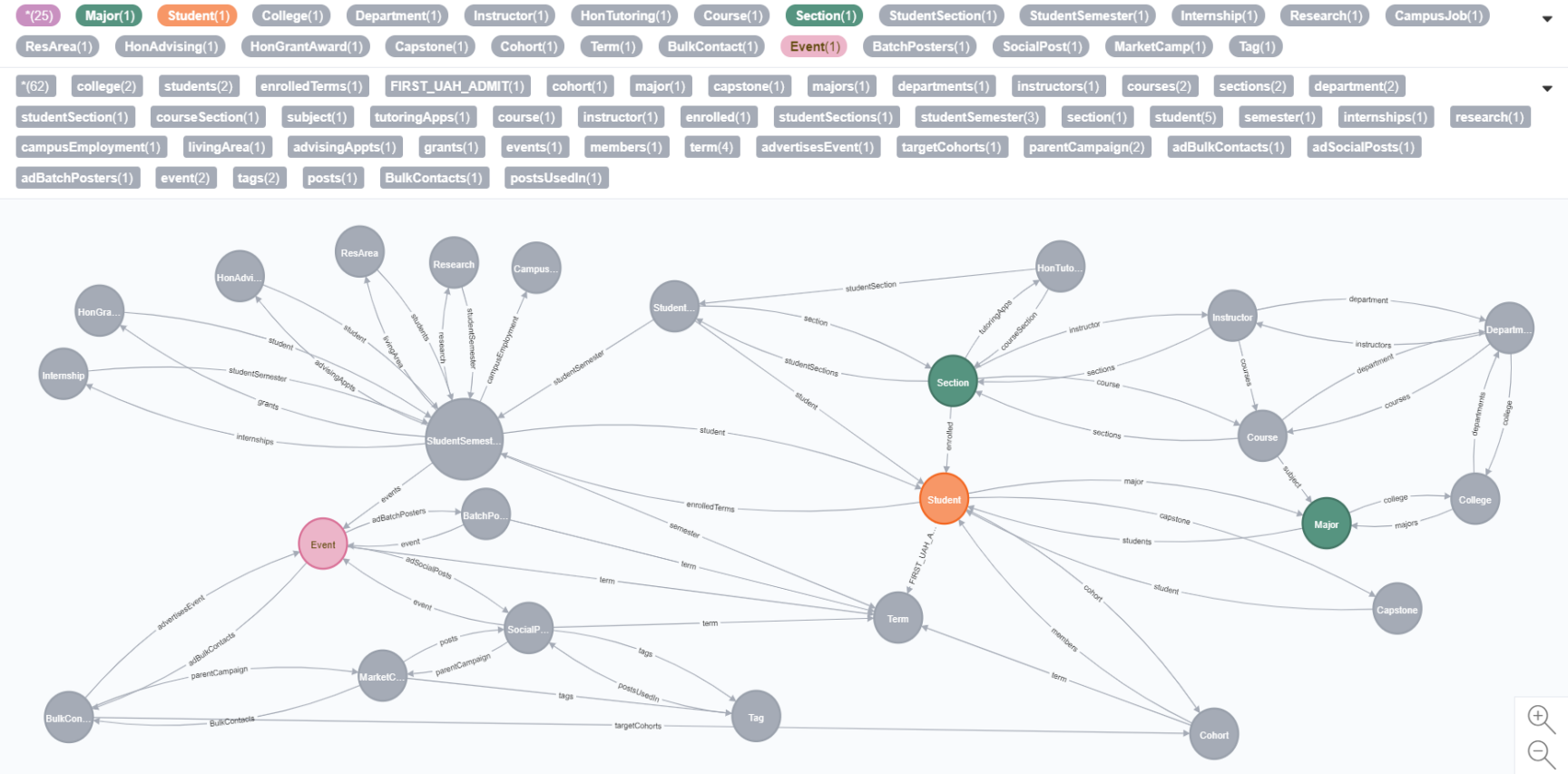
As a proof of concept, the application works fantastically. The limiting factor for the application and its theoretical agency lies in its ability to collecting data about student involvement. Though the application is able to transform data from a spreadsheet into a format usable by Graph database, the biggest limitation is that some fields are not tracked anywhere – in any format. Involvement records such as event attendance, club involvement, and more are critical metrics that no organization currently tracks. Additionally, social media posts have to be added manually to the system.

The project can currently return the trends found in the data, but as of current there is no way simple way to collect the required data. The most useful addition to this project would be the creation of built-in tools for data collection. For example, adding a feature which (for a

specified duration) creates a website that students could go to from their phones to sign into an event. Signing in could automatically flag the student as having attended the event in the database. Students who sign in could then be entered into raffles as a method of encouraging students to sign in. Another useful utility might include automated tracking for social media posts on Twitter, Instagram, and Facebook. By linking an account to the application, it could in theory detect hashtags.

Any number of features could be added to assist in the easy collection of data. However, once the application is collected, the provided design pattern can and will allow for the observation of trends previously illusive to higher education faculty.

Appendix A



This image represents the schema defined for the prototype application. The schema defines how different pieces of data connect to one another. This schema is assumed to be the one used for the User Interface; however, the database itself is capable of utilizing a new schema dynamically if more connected fields, such as Clubs, were to be added in the future.