Spring 2019

GMS - Guest Management System

Ethan Clark
ejc49@zips.uakron.edu

Introduction

A single application will never meet the needs of all its users. Software must evolve with its userbase, but unfortunately developers and their end users rarely interact. This problem is further expounded when those end users are separately located from the developers. Neither can an organization develop all its software in-house, when reliable products exist to solve relatively simple tasks. Why would resources be wasted developing a new text editor, for example, when so many cheaper alternatives already exist? Thus the question of whether an organization should produce their own custom software or outsource for an existing solution presents itself.

Without software, our expensive devices lose their fundamental abstraction. Software, “characterized as the executable code that [conducts] machine operations,” can best be broken into two categories [1]. Proprietary software “authorized under… the copyright holder” and whose “source code is dependably kept a mystery” due to its nature as a paid product, and open source software “which allows the source code (computer code) to be shared, viewed, and modified by other users and organizations.” It can be argued that both categories offer equivalent features for their respective customers, but many companies need software that meets a very specific set of requirements. With only about one third of software projects ending on-time and on-budget, the fear of beginning a new project to develop custom applications might just strike too much fear in executive committees’ hearts [2]. When faced with so much failure, the safest decision could easily be poaching another company’s work; that is, purchasing a software solution from a third party that has already been developed and implemented by a competitor. If software is what gives our systems meaning, then does using the same software as a competitor risk losing a company’s competitive advantage?

At the University of Akron, our own Department of Residence Life and Housing faces many challenges comparable to other corporations. This department manages the nine residence halls on campus and operates in conjunction with university police in an effort to provide a safe and secure living experience. In each of these nine buildings, community assistants (CAs) work perpetually as a first line of defense against unwanted intruders. Among a CA's duties is checking in guests, a task which has been notoriously cumbersome when considering the task's mundane nature. The current guest management software provider, Residential Management Systems (RMS), has licensed Residence Life a web application that allows guests to be checked into each building on campus. While RMS does its job fairly well in lieu of a more traditional pen-and-paper process, its problems resonate with many CAs.

To address these concerns, I have created a new guest management system, aptly titled GMS. By developing software designed to address specific issues, I can reasonably compare RMS against my own GMS solution. This comparison between features of two applications with fundamentally congruent goals should serve to justify the benefits between proprietary and open source software. While my project can serve as an appropriate case study, this research should not be mistaken for advice on choosing a software solution.

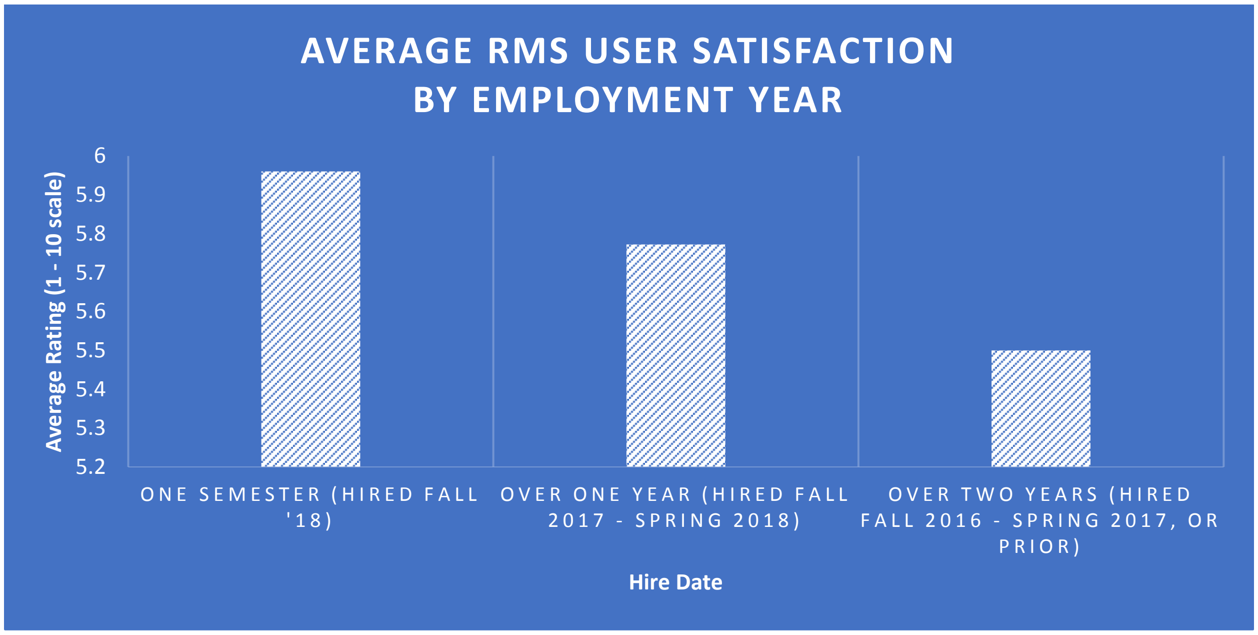
Planning

Further sections of this report will focus on the stages of the software development lifecycle [3]. The lifecycle model has been a proven guide for how companies can effectively build software without going over time or budget. Several different methodologies exist, ranging from the traditional Waterfall approach to the more iterative, Agile approaches [4]. The Waterfall methodology has a company procuring project requirements early and extensively in the development lifecycle, in contrast to most other iterative methods which reexamine requirements with a customer, usually after monthly intervals. All these approaches feature similar stages, just with varying timelines and priorities to them. Given this project's scope, the waterfall methodology was more than acceptable.

My end purpose for this project was to master the tools and languages used in web development. This would mean learning and utilizing client-side languages such as HTML, CSS, and JavaScript, while also familiarizing myself with the server-side components using C# and ASP.NET. In simplest terms, client-side languages are used to produce the interactivity of a website, while server-side languages assist with database access and logic control flow. Robert W. Sebesta's textbook *Programming the World Wide Web* was used in preparation for understanding these languages and the differences between them [5].

Analysis

A few different methods were used to collect data about RMS' current performance. The first among these was a survey which polled all currently employed CAs. Out of a total 61 responses, CAs were prompted to share their opinion of RMS, why they felt that way, and to share any features that they would want in a new system. Responses generally discussed the limitations of the site, such as how it can only run using Internet Explorer with compatibility settings, how it is impossible to see all currently active guests, and how its interface felt like an "outdated legacy application" despite being newly brought on in 2017. Other responses included anecdotes about the site's lack of reliability and policy complaints which seem to be out of RMS' scope of control. It is no coincidence that as CA longevity increases, overall satisfaction steadily decreases [Figure 1]. Responses in the column *Over Two Years* are likely lower due to all CAs at this point working with RMS' predecessor, Guest Tracker, which included features absent from RMS such as the ability to see all currently active guests in any given building. From this survey I was able to elicit the most highly requested requirements and features that I would eventually incorporate into GMS.



Population Std. Dev: 1.775, Std. Error: 0.227

Figure 1 – Average satisfaction results from survey regarding RMS

To better understand RMS' administrative responsibilities, I held a requirements elicitation session with the associate director and systems administrator of Residence Life and Housing. This meeting was focused on how RMS uses data about residents and their guests. One prominent issue raised was the rough transitions of data between university departments. For example, University Student Conduct requests an Excel file be sent focused around a first and last name whenever a student has broken guest related university policy. RMS has limited capabilities in exporting its data, requiring staff to modify the data columns manually. RMS' underlying database structure was also discussed as to model GMS after the department's current data state as much as possible. The department's data management choices heavily influenced GMS' design.

Design

The end of the Fall semester was reserved for creating GMS' back-end database. This database would need to include data on each building's residents and their guests, at minimum. The diagram below illustrates the relationships between tables in GMS' underlying database [Figure 2].

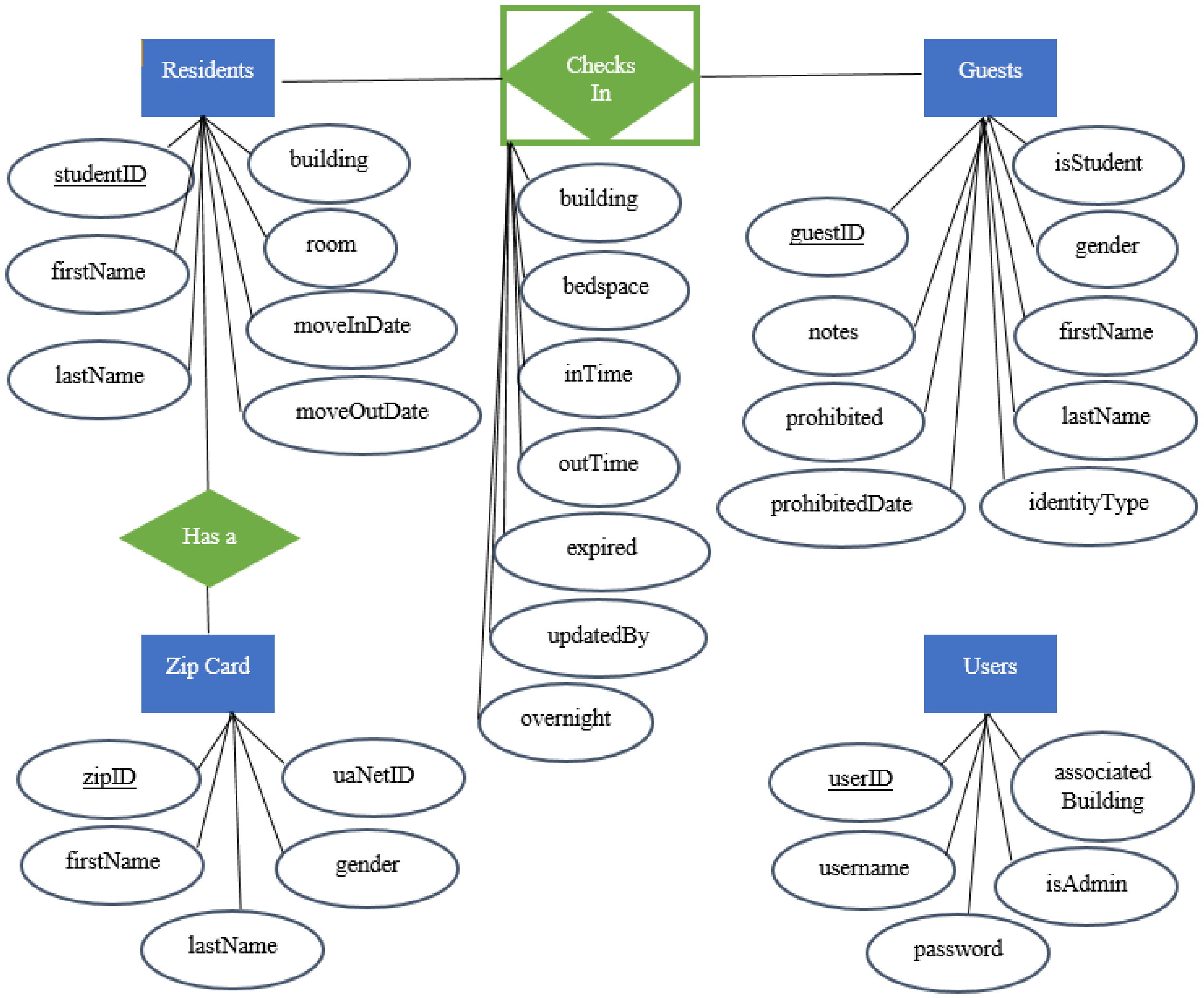


Figure 2 – Entity Relationship (ER) diagram representing GMS' underlying database design

Each rectangle in this diagram represents an entity, or a table in the database. The diamonds represent the relationships between entities, and each oval a different column listed under its table. A resident can check in a guest, for example, and a guest can be checked in by a resident. The attributes chosen here were modelled off RMS' database, but with some notable omissions. RMS keeps data on a guest's phone number and email but allows those values to be null. Since a CA is not required to record this information, its inclusion in GMS was forgone since there were no clear situations in which that data needed to exist. Cutting the fat from RMS and designing for only the necessities proved to be a common theme during the design process.

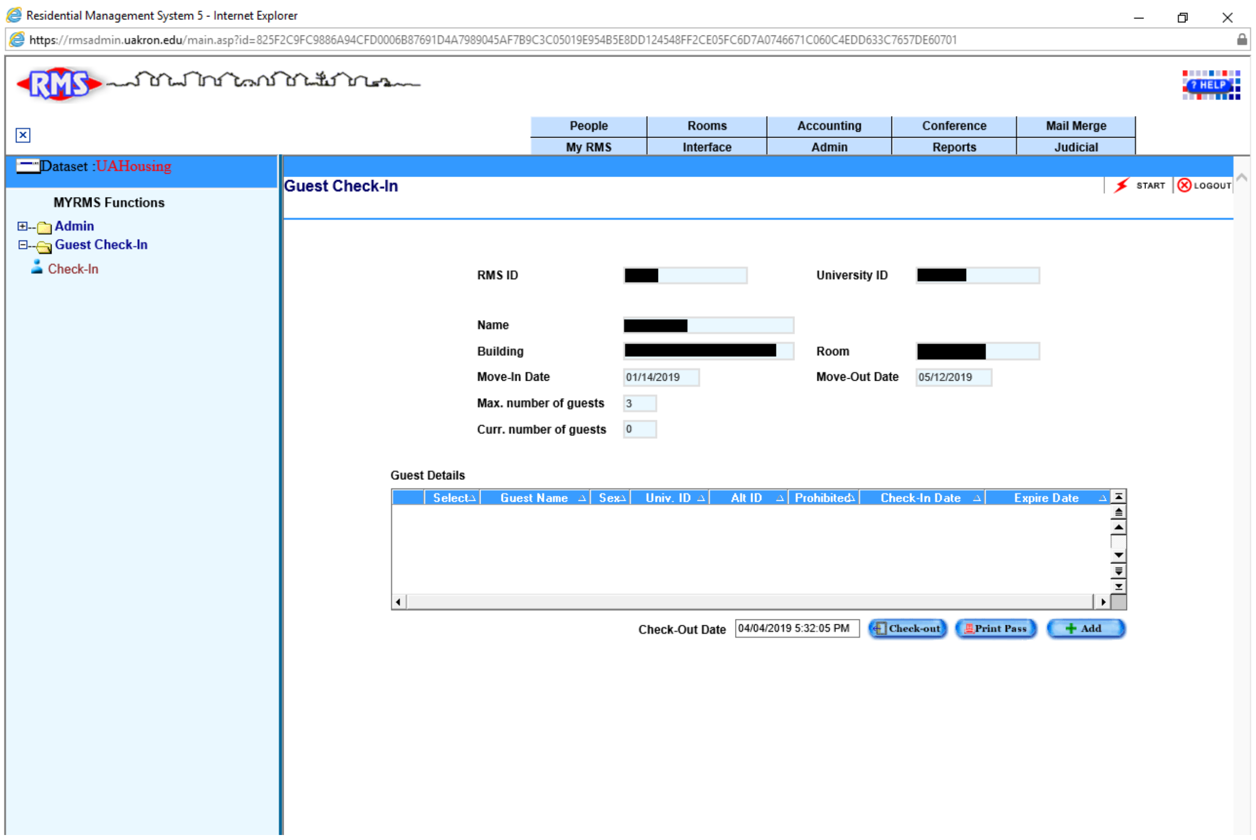
The residents table contains data on all university students living on campus. By default, they must have a zip card, a building, room, and move in date. Any resident can check in at most three guests, each of which is represented by an identification number from a student ID, driver's license, or passport. If a guest has been banned from entering the residence halls, that guest can be flagged as prohibited. These guests will not be permitted to check in, and the CA should notice the error. The department of residence life and housing has several other policies surrounding guest check in, such as limiting the amount of days in which a guest can stay checked in. All these policies were considered under the presupposition that the policies are in place to protect the residents first, to notify the CA of issues highlighted by the data, and to only be acted upon should an unwelcome situation occur. GMS attempts to handle as many of these edge cases as possible behind the scenes while keeping the site logic streamlined.

One of the most notable problems in the database's design is the data redundancy issue. The Zip Card and Residents table both keep track of a resident's ID number, first, and last names. If these values were to change in the zip card table, another update would also be required in the Residents table, inhibiting the Residents' table future integrity. This problem exists because I do not have proper access to the live pool of students' zip card data; that table is purely a placeholder which forces the design to be slightly sub-optimal.

Due to the confidential nature of a university student's data, mock data was generated to resemble what Residence Life uses. This data would later be imported to Microsoft SQL Server from several Excel documents. Two SQL statements were then executed to create the tables and insert the data into them, while also updating any sensitive data like Student IDs, names, and emails to generic placeholder values.

Having the database covered, it was time to start work on the website itself. Since GMS was being modeled directly off RMS, examining its design was my first consideration. RMS has a profile-centric design, where each guest is tied to a resident's profile [Figure 3]. One consequence of this design choice is that the guest's accompanying resident must be present in order to sign them out. Although this is based off residence life policy, some guests are never checked out as a result, especially when a CA's desk shift goes unfilled. This thereby leads to residents signing in a guest multiple times before they reach their third guest limit, null check out times, and inaccurate information.

GMS attempts to fix these issues by having a building-centric design. Each building was assigned a unique account so that all site components featured only that building's relevant information. So when the Bulger account checks their currently active guests, the CA will only see Bulger's guests and not any other building's. Since GMS keeps track of the building entered on every transaction, guests will be checked out of all buildings upon departure, in case that guest never checked out of a building elsewhere. These simple changes allow CAs to have a higher-level view of their buildings and prevents superfluous null transactions in the process.


Residential Management System 5 - Internet Explorer
https://rmsadmin.uakron.edu/main.asp?id=825F2C9FC9886A94CFD0006B87691D4A7989045AF7B9C3C05019E954B5E8DD124548FF2CE05FC6D7A0746671C060C4EDD633C7657DE60701
RMS
? HELP
People
Rooms
Accounting
Conference
Mail Merge
My RMS
Interface
Admin
Reports
Judicial
Dataset :UAHousing
MYRMS Functions
Admin
Guest Check-In
Check-In
START
LOGOUT
Check-out
Print Pass
Add

Guest Check-In

RMS ID	[redacted]	University ID	[redacted]
Name	[redacted]		
Building	[redacted]	Room	[redacted]
Move-In Date	01/14/2019	Move-Out Date	05/12/2019
Max. number of guests	3		
Curr. number of guests	0		

Guest Details

Select	Guest Name	Sex	Univ. ID	Alt ID	Prohibited	Check-In Date	Expire Date

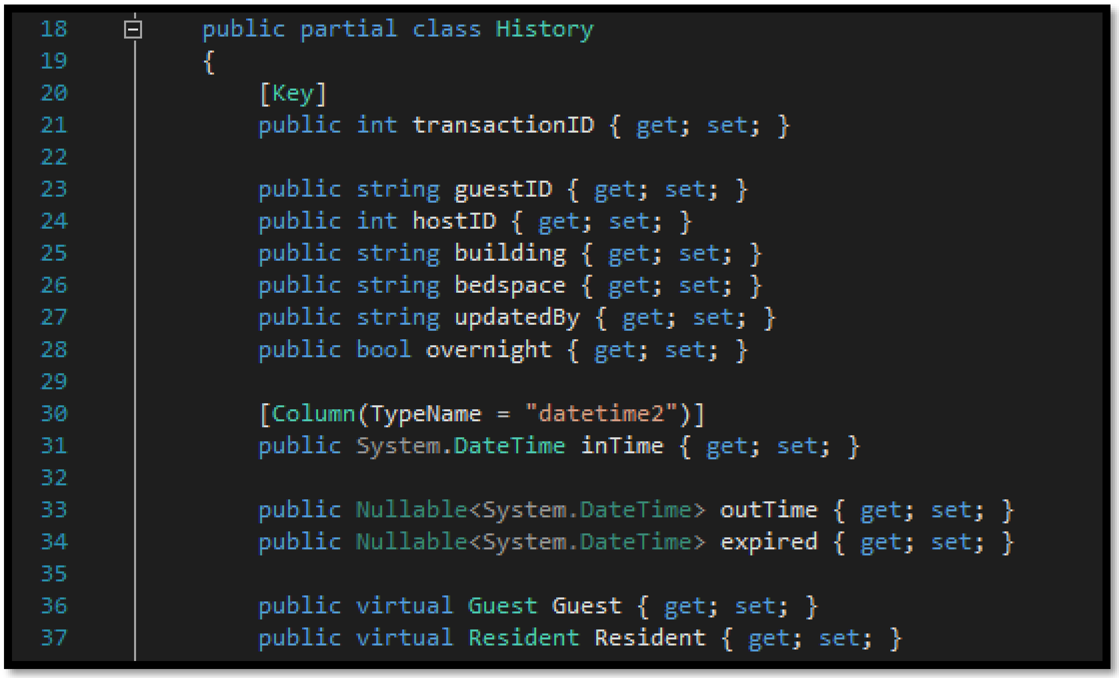
Check-Out Date 04/04/2019 5:32:05 PM

Figure 3 – The home page of the currently in-operation guest management system, RMS

<u>Development</u>

A web application can be thought of as a single resource managed by multiple distinct components. GMS achieves this distinction using the ASP.NET MVC pattern, which "decouples [its] user interface (view), data (model), and application logic (controller)" from each other [6]. In simplest terms, the user interacts with what they view on the page, sends action requests to the controller, and will have data returned back to the view if applicable.

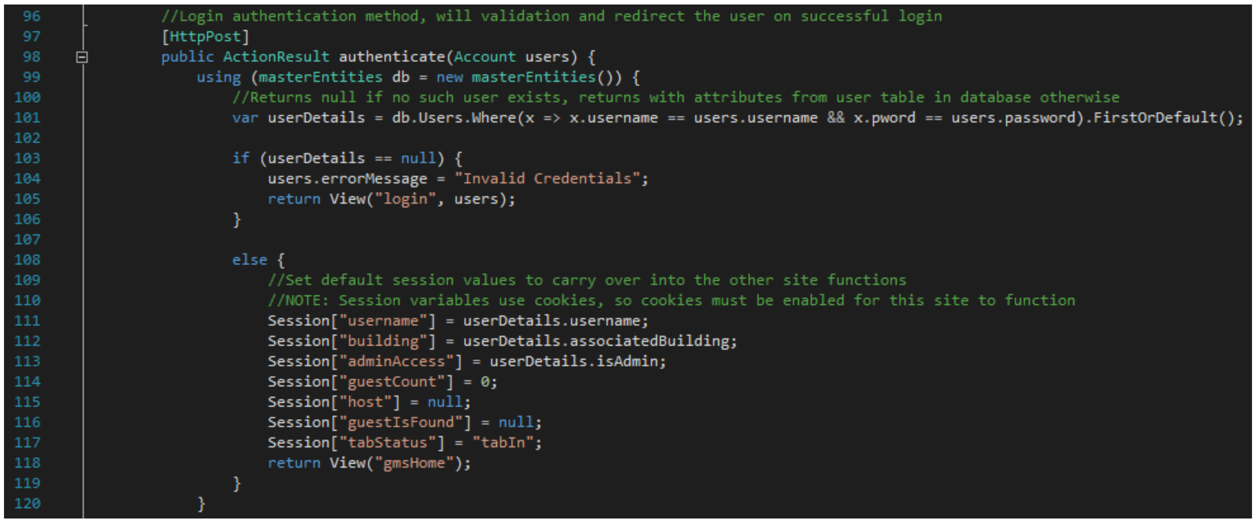
Models are comparable to classes in any object-oriented language. GMS' models were auto-generated with .NET Entity Framework, an extension of MVC which "provides an interface to an already existing database" [7]. An example of one is shown below, where a History object is represented as the collection of its members [Figure 4]. The model shown stores information about a guest's check in history; the Guest and Resident virtual objects refer to their respective models and symbolically act as foreign keys (unique non-null references to other tables). Models can be used to add and find rows from our database, store form data, and a multitude of other useful tasks. Qualifiers such as **[Key]** and **[Column(…)]** were manually added to help validate those members as primary key values and SQL Server valid input respectively. Normally all business logic, trade secrets, and company-sensitive code would be written in each model as different methods. Had these models not been auto-generated by Entity Framework, this would have been the case. Any change to the database required reimporting all models back to an original state, thereby losing all self-added code. If GMS were to enter production, moving the logic to each respective model would be a necessity.



```
public partial class History
{
    [Key]
    public int transactionID { get; set; }

    public string guestID { get; set; }
    public int hostID { get; set; }
    public string building { get; set; }
    public string bedspace { get; set; }
    public string updatedBy { get; set; }
    public bool overnight { get; set; }

    [Column(TypeName = "datetime2")]
    public System.DateTime inTime { get; set; }

    public Nullable<System.DateTime> outTime { get; set; }
    public Nullable<System.DateTime> expired { get; set; }

    public virtual Guest Guest { get; set; }
    public virtual Resident Resident { get; set; }
```

Figure 4 – The "History" model

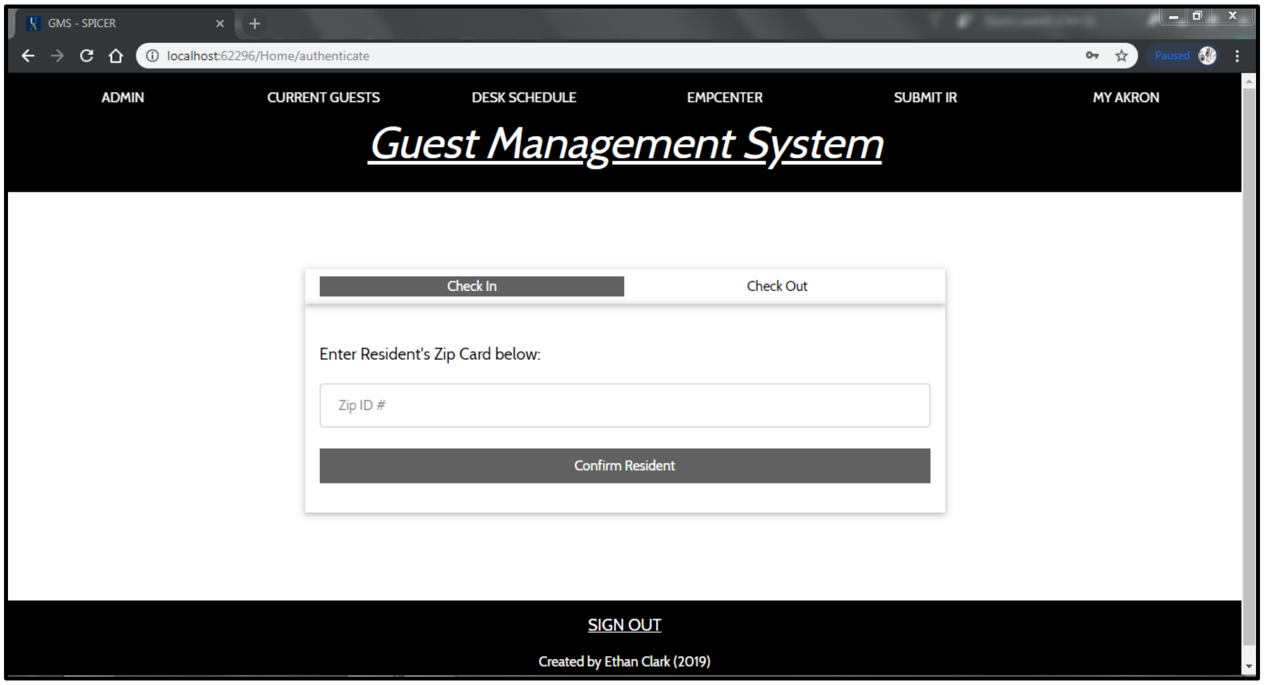
GMS organizes its user interface into four main views. The first page the user will see is GMS' login view. Trying to access other areas of the site while not logged in will kick the user back out to the login page. Because there are a finite number of buildings, GMS can use cookies in the form of Session variables to store building-relevant account details as the user goes from page to page [Figure 5]. This is how GMS can show all currently active guests to a CA in one building while shielding that information from another.



```
        //Login authentication method, will validation and redirect the user on successful login
        [HttpPost]
        public ActionResult authenticate(Account users) {
            using (masterEntities db = new masterEntities()) {
                //Returns null if no such user exists, returns with attributes from user table in database otherwise
                var userDetails = db.Users.Where(x => x.username == users.username && x.pword == users.password).FirstOrDefault();

                if (userDetails == null) {
                    users.errorMessage = "Invalid Credentials";
                    return View("login", users);
                }

                else {
                    //Set default session values to carry over into the other site functions
                    //NOTE: Session variables use cookies, so cookies must be enabled for this site to function
                    Session["username"] = userDetails.username;
                    Session["building"] = userDetails.associatedBuilding;
                    Session["adminAccess"] = userDetails.isAdmin;
                    Session["guestCount"] = 0;
                    Session["host"] = null;
                    Session["guestIsFound"] = null;
                    Session["tabStatus"] = "tabIn";
                    return View("gmsHome");
                }
            }
```

Figure 5 – GMS' authentication process, which initializes the site's cookies

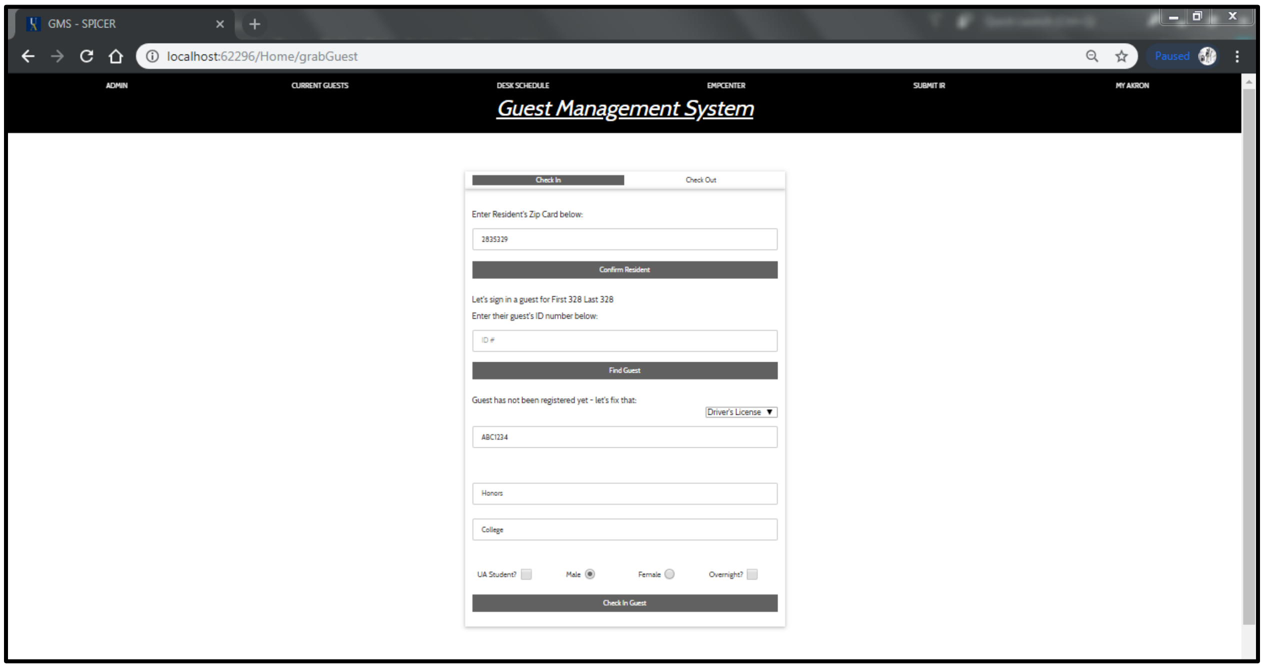
GMS' main view had to allow a CA to check in and out a guest. The home page fulfills this requirement by validating user input and collecting data through HTML forms, several of which are hidden until conditions in the process flow are met. These forms use lambda expressions to populate the History model with the form data, pass the model to the controller, and query the database. Guest transactions all take place on one page in an effort to simplify the CA's site experience. RMS had many links with no functional purpose to the CA, and without proper training many found RMS to have a cumbersome learning curve. GMS attempts to rectify this with a home page that leads the user along, hiding non-applicable forms until the user is ready to see them [Figure 6].



Enter Resident's Zip Card below:

Zip ID #

Created by Ethan Clark (2019)



Enter Resident's Zip Card below:

2835329

Let's sign in a guest for First 328 Last 328

Enter their guest's ID number below:

ID #

Guest has not been registered yet - let's fix that:

Driver's License ▼

ABC1234

Honors

College

UA Student? ☐ Male ● Female ○ Overnight? ☐

Figure 6 – The GMS home page on startup (top),

and GMS' home page, zoomed out, upon registration of a new guest (bottom)

GMS' guest transaction process has a very straightforward process flow. First, the resident must bring their guest to their building's front desk where the CA will validate the resident's student ID. GMS ensures that the resident actually does live in the building, and if not a warning message will be thrown from the controller preventing both the resident and CA from continuing the process. Next the CA will enter every guest's ID number, which will in turn populate the guest's profile into another form. If a guest has not registered their ID yet, the CA can manually enter their name, identity type, gender, and student status to create a new guest profile while simultaneously checking them into the building. Various validation checks occur along the way, preventing the CA from breaking flow. If needed, the CA can switch tabs on the home page to check out a guest by entering their ID. Every text box also works with Residence Life's desk scanners, so that IDs do not need to be entered manually. This also means that if the check out tab is left open while no CA is on duty, a guest would be able to scan their ID and check out even after hours.

When any form is submitted, its data is passed to the controller via the History model. Each form calls a method within the controller which processes the data. In almost every case, the method called creates an instance of the database, runs a query, checks if any rows were returned from the database table or not, and returns to the original view. The methods within the controller are used to authenticate user login, check in and check out a guest, perform administrative transactions such as returning a guest's archived check in history, and return a list of currently active guests. If a form exists in the view, then a matching method for it exists within the controller.

While GMS' development is technically over, it does have room for future improvement. The largest technical concern would be publishing the site to a remote server. For development and testing purposes, GMS was hosted locally using Microsoft's IIS Express as a test server. Publishing GMS would require a remote server, which would also likely create costs outside the scope of this project. GMS' stability online is also unknown without first running it on a remote machine; this is a relevant problem when a large fraction of CAs surveyed cited RMS' downtime as a major defect. GMS also keeps track of when a guest's check in period has expired 24 hours after the initial check in, but without running on a remote server all day, GMS has no way to log that record in the database as it is now. There is also room to grow GMS' administrative responsibilities, especially concerning the topics of overnight and prohibited guests, as only three distinct admin functions are currently offered.

Conclusion

The creation of GMS sought to address all the requirements addressed by Residence Life and their respective CA employees. These features included the ability to:

- See all guests currently checked into a given building
- Scan a guest's driver's license as their guest ID
- Use Google Chrome or other browser without extra compatibility options
- Check out a guest without the accompanying resident being present
- Link to CA resources like EmpCenter and the Desk Schedule without leaving the site
- Pull a guest or resident's transaction history by name into an Excel file format

Apart from scanning a guest's driver's license which appeared to be a hardware issue, GMS accomplished every requested requirement. This is in addition to reverse engineering every core feature in RMS to be more user friendly. While it would appear that GMS achieved for the University of Akron in nine months what a company could not provide years later, this notion could not be any further from the truth.

RMS as it has been referenced in this paper refers to just one tool from Mercury Inc.'s student management software toolkit [8]. Mercury provides software services such as a cloud database, bed hosting interface, and as described here, a guest management system. Comparing GMS to what appears to be a side project from a large corporation would be a laughable attempt. GMS focused on a set of very specific requirements elicited from University of Akron administrators and employees and delivered on every front. Alternatively, Mercury RMS works with many different universities which understandably have different data requirements, internal policies, and software needs. Should GMS ever be repurposed for another university, it too would face the same complaints of IT professionals working with third party proprietary software such as “incompatibility of code, software crashes, slow installation” and other cost factors [9]. Spending too much time learning someone else’s code rather than implementing an innovative new solution does call into question how necessary an on-site development team might be. An open source application can be customized for any number of niche features, but liabilities like project cost and low success rates may be intimidating enough to warrant anybody else doing the job. Any given organization will absolutely have different needs from another, but whether those needs are worth meeting remains to be seen.

References

1. Singh, A., R.K. Bansal, and N. Jha, Open Source Software vs Proprietary Software. International Journal of Computer Applications, 2015. **114** (18). Retrieved from https://pdfs.semanticscholar.org/48b7/64286fde00991c9b8ffc2b88ee8a6c7207b3.pdf
2. The Standish Group International (2009). *Extreme Chaos*. The Standish Group International, Inc. Retrieved from https://www.classes.cs.uchicago.edu/archive/2014/fall/51210-1/required.reading/Standish.Group.Chaos.2009.pdf
3. Tricia Hussung (2016). *What is the Software Development Lifecycle?* Husson University. Retrieved from https://online.husson.edu/software-development-cycle/
4. Bhuvaneswari, T., Prabaharan, S., "A Survey on Software Development Life Cycle Models", International Journal of Computer Science and Mobile Computing, Vol. 2, Issue. 5, pp. 262 – 267, May 2013. Retrieved from https://www.ijcsmc.com/docs/papers/May2013/V2I5201384.pdf
5. Robert W. Sebesta (2015). *Programming the World Wide Web, Eighth Edition*.
6. Microsoft (2019). *ASP.NET MVC Pattern* | *.NET*. Retrieved from https://dotnet.microsoft.com/apps/aspnet/mvc
7. Microsoft (2019). *Getting Started with Entity Framework*. Retrieved from https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/database-first-development/
8. Mercury RMS (2019). *Student Management Software*. Retrieved from https://mercury.rms-inc.com/mercury-tools.html
9. Sahu, P., & Bhadury, K. (2017). Standard vs. custom Software: How does a company make right decision? *International Journal of Latest Engineering and Management*

Research (IJLEMR), *2*(2). Retrieved from http://www.ijlemr.com/papers/volume2-issue2/15-IJLEMR-22058.pdf