Project Number: MQP MXC-0577

Washburn Shops Lab Management System

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

in Computer Science

by

_____

Samuel M. Baumgarten

March 22, 2019

_____

Professor Michael J. Ciaraldi, Major Advisor

Abstract

Each year, over 1,600 students use the Washburn Shops Manufacturing Labs. As the facility usage has increased, the need has arisen to better understand how the shops are utilized, to ensure that users comply with shop policies, and to enable WPI community members to access the facilities with a minimal amount of friction. This project utilized tools such as Ruby on Rails, Docker, and Swift to develop a software solution to manage shop permissions and compile usage analytics for the manufacturing facilities at WPI.

Table of Contents

1 Introduction

1.1 Background

The WPI Mechanical Engineering Department operates two manufacturing labs on campus – the Washburn Shops and Higgins Labs. An estimated 1,600 students register to use these shops each year to make use of the mills, lathes, and other various tools that members of the WPI community have access to. As the shops have grown, a desire has emerged to better understand and summarize how the shops are utilized as well as to better manage the roles-based permission system utilized by the manufacturing labs.

1.1.1 Lab Roles

With so many people using the labs each year, a permission scheme is used to formalize what each lab user is allowed to do. Each person who enters the shops has one of the following five permission levels:

- Visitor
- Basic User
- Project Space User
- Lab Monitor
- Authorizing Lab Monitor

Each permission level has associated responsibilities and rules that describe how and when users are allowed to utilize the manufacturing labs. Within these roles, there are sub-roles that grant permissions for specific machines.

1.1.2 Prior Management Solution

Prior to this project, there were multiple systems used to manage various aspects of the manufacturing labs. Google Drive was used to keep track of basic users and paper records were used to manage all other levels. Paper records were used to track tool authorizations and no usage statistics were tracked aside from registered users and registered projects. A custom solution was used to manage the current lab monitor in the shop and update the digital signage outside the labs. The lack of centralized data made it difficult to gather information aside from primitive analytics. In addition, this lack of data made it difficult to preemptively identify problems as paperwork was only reviewed retroactively if a problem arose.

Visitors	Not Recorded
Basic Users	Google Drive
Advanced Users	Paper Record
Project Space Users	Paper Record
Lab Monitors	Paper Record
Authorizing Lab Monitors	No Record

Table 1. Methods used to track a user's role.

1.2 Solution Overview

1.2.1 Overview

This MQP focuses upon the development of a web-based system that centralizes the aforementioned management needs as well as handles usage tracking and permission enforcement. This system consists of the four following components:

- Entry System
- Lab Monitor Device
- Backend/API
- Back-Office/Analytics System

1.2.2 Entry System

The entry system is the primary way users interact with the system. Whenever a user enters or exits the shop, he taps his WPI ID on a reader located at the entry point of the shop (See Figure 1). This checks him in or out depending on which kiosk he tapped his ID on. This is also the point where the user is notified whether he has permission to enter the shop. If a user is authorized to enter the facility, a name badge is printed. Users can see the current status of the shop by looking at digital signage located at the entrance of the shop displaying the current lab monitor and other important facility announcements.

Figure 1. A rendering of one of the entryway kiosk devices.

The entry system also serves as the registration on-boarding station for new users. When users first come to the shop, the Washburn system does not have any prior information about the user. When they attempt to sign in through the entry system, they will be prompted to enter their WPI ID number. This is when users' WPI ID numbers are correlated with their unique *RFID card number*s from their *WPI badge*. When a user starts the registration process, the system prompts him for his name, asks him to take a photo, and ensures that he has completed the basic user quiz.

1.2.3 Lab Monitor Device

Whenever a manufacturing facility is open, there must be an individual present who is authorized to be in-charge of the space. Users permitted to fulfil this role are referred to as lab monitors. The lab monitor device is the current lab monitor's primary way of interacting with the system. It enables actions such as closing the shop, switching lab monitors, viewing active users, and checking the permissions of any user in the system. Potential future uses include tool checkouts and other day-to-day administrative tasks. This device is also be used to notify the lab monitor when a user checks-in and when someone is trying to call the lab monitor from

outside the shop (over a *VoIP video call*). The lab monitor device is an iPad Mini running a custom application called Washburn Controller.

1.2.4 Backend/API

The backend is what orchestrates the entire system. This offsite component stores the user data and handles permission validation. All the other components use the API exposed by the backend to interact with the system. The API is designed to allow for future devices to be developed and deployed leveraging existing parts of this system.

1.2.5 Back-Office/Analytics

The back office is where shop staff manage the shop. They can modify users, diagnose problems, and make system-wide changes. Unlike the shop controller device, the purpose of this system is not to make real-time changes but rather for tasks such as modifying user info, updating verbiage, and clearing users. The back-office portal is also where shop staff can approve tool authorizations and view audit logs.

1.3 Evaluation

This MQP is primarily centered around the development of the Washburn management system and does not deeply evaluate its performance. Due to the significant amount of data generated, it wasn't feasible to go to the depth necessary to truly evaluate the system's success in this paper. There is a corollary IQP being conducted that delves into how the data generated by the system can be utilized and visualized as well as how the data can be used to improve the system itself. This IQP's paper provides additional insight into the performance of the system and its success at achieving the desired goals.

2 Entryway

The entryway is the location where users can check-in, check-out, and register for the system. It is located at the front entrance of the Washburn Shops (WB-107). The system consists of multiple entry kiosks, card readers, a digital signage display, and a name badge printer. Whenever a user visits a facility equipped with this system, he will tap his ID on the check-in kiosk card reader. If it's the user's first visit, the touch-screen kiosk will walk the user through the setup process. Once the setup process is completed, or if the user was already registered, the system will check the user in and print a dated name badge that lists all the tools a user is authorized to use. When the user leaves, he will tap his ID card on the card reader that is attached to the check-out kiosk in order to check himself out. This section will detail the components of this system and how they work.

2.1 Card Readers

2.1.1 ID Cards

The primary method by which users identify themselves to the shop management system is with their WPI badges. WPI ID Services issues all members of the WPI community secure identification badges on HID iClass cards with magstripes and barcodes. These cards are encoded in a Corporate-1000 format, a system provided by HID Global. The WPI ID cards are encoded with 35-bits of data including a facility code, card number, and parity bits. The shop management system uses the encoded RFID number and WPI ID number to identify students.

2.1.2 Card Reader Selection

We considered a number of card readers in the planning phase of this project. Card readers were selected based on the following criteria:

- Ease of Implementation
- Maintainability
- Reliability

The final contenders in our selection process were the RF IDeas pcProx Plus readers and the HID iClass SE readers.

Figure 2. An HID iClass SE R10 card reader.

The previous implementation of this system made use of the RF IDeas pcProx Plus readers which were easy to use with devices running Microsoft Windows. They are compatible with HID iClass cards and in our experience very reliable. We had concerns about the maintainability of the readers as we were unsure about the future availability of the pcProx readers. In addition, to our knowledge, the pcProx readers are not used anywhere else on campus.

The HID iClass SE readers are used throughout the WPI campus on most access-controlled doors. The readers support Wiegand and OSDP which require a non-trivial amount of technical implementation to interface with; however, they provide additional flexibility over USB readers. The HID iClass SE readers are extremely reliable and are made by the same company that produces the WPI ID cards. These readers are easily maintainable due to their pervasive use on campus and WPI's existing relationship with a distributor.

Based on our analysis of these two options, we elected to use HID iClass Readers in this component. We specifically utilized HID iClass SE R10s (See Figure 2), the mini-mullion version in the iClass SE series due to its small form factor.

2.1.3 Reader Communication

2.1.3.1 Reader Protocol

HID iClass Readers support two communication protocols: Wiegand and OSDP (Open Supervised Device Protocol). The readers we selected, the HID iClass SE R10s, support both protocols. Wiegand is a legacy protocol that most card access installations still use. This protocol is susceptible to MiTM attacks which has led to a push for new installations to use OSDP. ODSP is a far more secure protocol; however, it is very new and not as ubiquitous as Wiegand. For this installation, we elected to use Wiegand due to its ease of implementation and compatibility; however, the system was designed with future OSDP compatibility in mind.

2.1.3.2 Controller Communication

In standard access control systems, card readers communicate with a local controller that interprets the signals from a reader and sends them to a centralized server to make access control decisions. For a controller to interpret the Wiegand signals, it must have both I/O capabilities and interrupts. For our system, we are using iPads as both controllers and as the primary user interface. The iPads we are using as the primary kiosk interfaces do not have the needed IO capabilities to be directly connected to the readers. To address this issue, we built an interface that allows the iPad to communicate with the card reader over Bluetooth Low Energy (BLE).

The goal of our interface device is to interpret the data generated by the card reader and securely transmit that information to the iPad. The interface device consists of an Arduino microcontroller and an HM-10 BLE module (See Figure 3). The iPad never directly interacts with the card reader, but instead sends commands to the interface which handles all reader communication. We considered three methods for communication between the iPad and the interface device: Bluetooth LE (selected), Lightning through the MFi program, or Lightning through the iPad keyboard interface. Lightning through the Apple MFi program would have been our preferred solution given unlimited time and resources due to the security and reliability benefits of using a wired connection. To use this solution would have required an application to the MFi program and significant hardware development resources which were outside the scope of this project. We excluded the Lightning through keyboard solution because

it would have made it difficult to allow for the on-screen keyboard to be utilized and would only allow for one-way communication. We ultimately selected the BLE option because of its flexibility and bi-directional communication while recognizing its limitations with regard to implementation effort and the need for advanced error recovery.

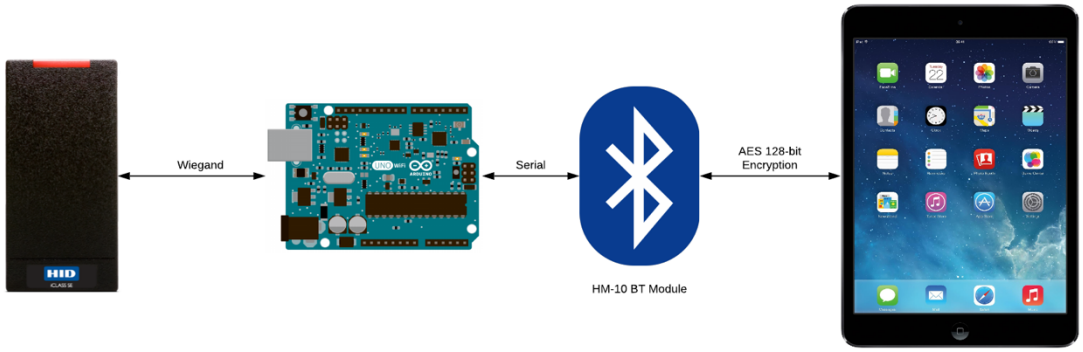

Figure 3. Card reader communication block diagram.

The reader interface is a fairly simple device consisting of three primary components: a microcontroller, a BLE module, and the card reader itself. The responsibility of each of these components is outlined in Figure 4.

Card Reader	Microcontroller	HM-10 (Bluetooth Module)
• Reads RFID Card • Sends card information over data bus • Provides visual and auditory feedback	• Interprets Wiegand signals • Encrypts card data • Computes checksum and forms data packet • Sends data to Bluetooth module over serial bus	• Handles connection state with iPad • Receives commands from the microcontroller and sends data to iPad • Receives data from iPad and sends to the microcontroller.

Figure 4. Analysis of the responsibilities of each component of the card reader interface device.

2.1.3.3 iPad Communication

Securely transmitting the card information from the microcontroller to the iPad is an essential function for the interface device. This task is accomplished using Bluetooth Low Energy (BLE) with a simple protocol that handles data validation and encryption.

All sensitive communications between the microcontroller and the iPad are encrypted using AES 128-bit encryption with a pre-shared secret key.

The connection process begins by the BLE module sending an advertising packet with a unique name. This name is randomly generated and assigned to a specific device in the admin panel (in the “Reader Name” field). Whenever an iPad is not connected to a reader, it will listen for

advertising packets and attempt to connect to any devices with a device name that matches their assigned reader name.

The communication link between the iPad and the microcontroller is bi-directional and can send packets with a maximum length of 16 bytes. This informs the communication protocol design described in the following paragraphs.

Once connected, the iPad will send an initialization vector (IV) over the BLE link. This IV is not sensitive but needs to change for every message. The generated IV is sent in two packets in the format shown in Figure 5. The command for sending the high bits of the IV is 0xA0 and the command for the low bits is 0xA1. If the microcontroller is unable to verify the checksum of either packet, it will send a message with the byte 0xB0 followed 0x00. When the iPad receives this message, it will generate a new IV and send it using the aforementioned procedure. The microcontroller will not send any messages until a valid and complete IV is received.

Command (1 byte)	IV (high/low) (8 bytes)	CRC16 Checksum (2 bytes)	Ignored (5 bytes)

Figure 5. IV transmission packet format from iPad to card reader interface.

Once the microcontroller has received a valid IV, future card reads will result in a message being sent to the iPad. A card read event is different from an IV initialization event as the information contained within the message is sensitive. Card messages follow the protocol expressed in Figure 6. The protocol includes the card data and a 2-byte, CRC16 checksum. The card data field is 5 bytes to allow for the transmission of 35-bit corporate-1000 format cards (used by WPI IDs) as well as standard 28-bit cards. Once the message is formed, it is encrypted with AES using the 128-bit pre-shared key and the IV sent by the iPad.

Card Data (5 bytes)	CRC16 Checksum (2 bytes)	Ignored (9 bytes)

Figure 6. Card read transmission data packet format from interface to iPad

The iPad assumes any 16-byte unterminated message is an encrypted message and attempts to decrypt it using the saved IV and the known key. If the decryption fails or the checksum is invalid, an error is presented to the user. If all the checks pass, the iPad processes the card number and performs the actions needed to check in.

After the iPad receives any communication, it changes the IV value and resends it. The microcontroller will not send any new messages until the new IV is received. This prevents XOR, MiTM, and replay attacks. The changing IV number means that the same card number will generate a different message every time, such that a listener cannot associate a transmission with a particular card, nor can they rebroadcast the same message at a later date as the iPad will not be able to decrypt the message nor will the checksum be validated.

2.2 Kiosk Device

A majority of lab user interaction occurs at the kiosk device, the primary HCI in the entryway system. The kiosk device consists of a touchscreen and a computing device where a user can view feedback or output upon check-in/check-out and complete the initial registration for the system.

2.2.1 Design Requirements

It was decided at the beginning of the project that the kiosk device would be some sort of touch-enabled device that users could quickly interact with when entering or exiting the facilities. This decision was made to ensure the device could be adapted to future policy changes, to give the user relevant feedback as to why admissions were accepted or rejected, and to enable the flexibility to implement interactive features in the future.

2.2.2 Initial Prototypes

At the beginning of this project, the decision was made to develop the kiosk device software using web technologies in an effort to allow for multi-device compatibility for future hardware replacement. This led us to implement the kiosk device software using Facebook's ReactJS, a common frontend application framework. This software was deployed on a Raspberry Pi 3B+ with a 7" touch screen (See Figure 7).

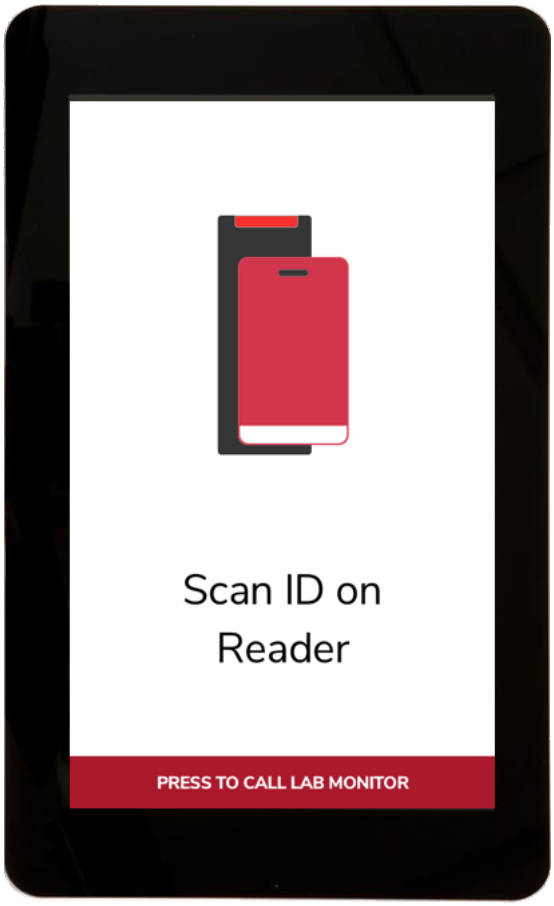

Figure 7. The prototype interface built using a Raspberry Pi 3B+ on a 7" touch screen.

Instead of using the card reader interface described in the Card Readers section, we were able to directly connect the card reader to the kiosk device in the prototype due to the exposed GPIO pins. We designed and manufactured a custom PCB to handle the connection of the card reader to the Raspberry Pi (See Figure 8). This PCB had a screw terminal connecter which the reader attached to and a 2x20 connector for a ribbon cable that would be connected to the Raspberry Pi.

In the testing of this PCB, we identified a problem where the attached display wouldn't power on unless a specific GPIO pin was disconnected due to the logic shifter being active when it shouldn't have been. The peak power consumption of the display occurs as it is first powering on and pulls more current than the rated load of the Raspberry Pi 3B+ as well as a higher load than is cited on the datasheet of the display. This is a

known problem with the display; however, it is only noticeable when there are other devices pulling any current from the Raspberry Pi's 5V bus. When the display turned on, it causes a brownout on the Raspberry Pi leading it to reboot and for the cycle to repeat. This problem was never resolved as other options were considered before a second PCB was produced; however, it could have fairly easily been resolved by either powering the display off a separate power supply or by pulling the enable pin of the logic shifter to ground and not enabling it until the Raspberry Pi had booted.

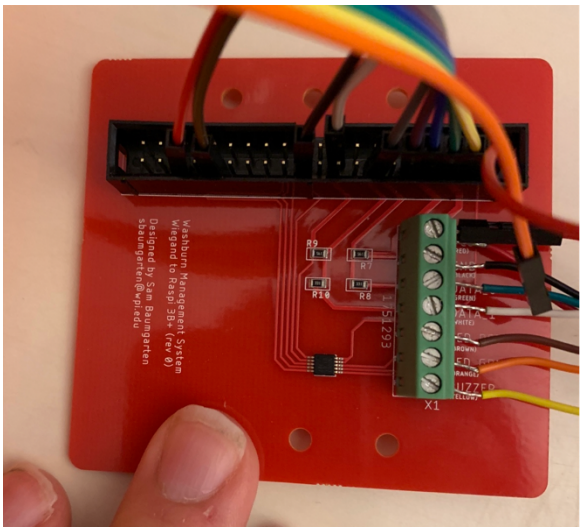

Figure 8. The prototype PCB for connecting an HID card reader to an iPad over BLE.

While this prototype was otherwise functional aside from the power issue described above, our initial testing identified a number of drawbacks found while interacting with the device. Some of the most prominent drawbacks were as follows:

- Touch screen looked low-quality
  - Bad color reproducibility and tilt-shift
  - Poor multi-touch performance
- Animations were not smooth
- Raspberry Pi was susceptible to corruption on power-loss
- Prototype solution involved significant configuration of the device which could lead to future difficulties due to an operating system or dependency change

Some of these concerns could have been addressed with new hardware such as high-quality touch panels or more robust computer hardware; however, we had difficulty sourcing parts such as the touch panels from large suppliers that would be easy to re-order in the future.

Once we better understood the limitations of the prototype hardware, we considered other solutions for the kiosk HCI. One such solution was to use a tablet as opposed to the custom Raspberry Pi hardware. The key benefits to using a tablet were that it would be easily replaceable should it fail, it would automatically handle power-outage situation due to the integrated battery, and components such as a camera, microphone, and speaker were already built into the device.

2.2.3 Final Hardware

The final hardware for the kiosk device consisted of a 9.7" iPad 6th Generation (model A1893). We considered both Android based tablets as well as the iPad. We conducted an evaluation of the benefits associated with these two options summarized in Figure 9. Ultimately, we selected the iPad based on the analysis paired with the extremely limited timeline for this project to be

completed. My existing experience with the iOS operating system significantly accelerated the development process.

Android	iPad
• Lower Price • More hardware options • Lack of experience with development tools • Less standardized and potentially difficult to replace	• Higher cost per unit • Limited hardware selection • Extensive experience with development tools • Easy to purchase initial and replacement device (Apple has an existing relationship with WPI) • Devices immediately available for development • App requires distribution on the App Store, through Apple B2B program, or re-signing the application every year

Figure 9. An analysis of the two tablet operating systems considered.

2.2.3.1 Software Architecture

Details regarding the software architecture of the Washburn Entry application can be found in section 3.2.

2.2.3.2 Device Configuration

The kiosk devices are configured using Apple mobile configuration profiles installed with an mobile device manager (MDM). MDM's are typically used to remotely configure and manage devices allowing functionality such as battery reporting, remote restarts, installing configuration profiles, and installing updates.

We use configuration profiles to secure the devices against unauthorized use, prevent tampering, and configure settings such as wireless configuration. We wrote a custom profile for the entry system to accomplish the following requirements:

- Disable most system functions (including AirDrop, app removal, iCloud backup, touch ID, Siri, connecting to iTunes, etc.)
- Add asset tag to lock screen ("Worcester Polytechnic Institute — Washburn Shops")
- Use App Lock to force iPad to boot directly into the custom application, disable auto-rotation, disable home button, and disable sleep/wake button
- Disable auto-lock
- Restrict removal of profile
- Connect to WPI Wireless using a Washburn Staff Member's wireless certificate (retrieved from https://wpi-wireless-setup.wpi.edu/)

Upon arrival, new iPad's can be configured using Apple Configurator or by manually setting up the iPad and then visiting the enroll URL provided by our MDM provider, SimpleMDM (https://simplemdm.com). When enrolling the device, the SimpleMDM portal will prompt the

user to select a group. The “Washburn Entry” group will automatically configure the device with the settings described above by installing the current version of the configuration profile to the device and updating it whenever it is changed in the MDM portal.

2.2.3.3 Annual Certificate Requirements

One of the drawbacks of using iPads is that all software must be code signed using a certificate from Apple’s certificate authority. Development certificates are only valid for one year so the software on the iPad must be re-signed annually. This burden can be removed by distributing the application on the public Apple App Store or privately through Apple’s B2B program. Distributing the app through the B2B program is likely the best option; however, it will require coordinating with the group that manages WPI’s Apple volume purchase program (VPP) account. Until this is done, the app can be re-signed by attaching the devices using a Lightning cable to a computer running Mac OS with Xcode installed. The app can then be re-built which will sign the app with the updated certificate.

See Section 3.4 for more information regarding the annual certificate requirements.

2.2.3.4 Physical Enclosure

The entryway kiosks are located in a publicly accessible area of the Washburn Shops meaning device security was an essential consideration. The Washburn Shops Manufacturing staff designed and constructed an enclosure for the kiosk device, card reader interface, and the card reader itself (See Figure 10). The enclosure consists of a wooden box with a metal faceplate that holds the iPad. The faceplate is secured with 8-32 #8 spanner security screws to present a minimal deterrent to theft or tampering. The iPad is secured to the faceplate using metal backing bars that can only be accessed when the enclosure is opened.

Figure 10. A rendering of the physical enclosure of an entryway kiosk.

The interface device is attached to the interior sides of the case using an adhesive paste and wires are secured to the sides. An AC line runs into the back of the enclosure where it is split into two AC-DC power adapters to power both the iPad and the interface device. The iPad is resilient against power loss due to its internal battery; however, the card reader is not operational without AC power.

2.2.3.5 Deployment Problems

While the deployment of the entryway devices went fairly smoothly, a few problems and bugs were encountered. There are three primary classes of problems we addressed: incorrect errors, cards not reading, and "ghost" reads.

2.2.3.5.1 Incorrect Errors

One class of error that was identified is the incorrect errors class. This class encompasses all problems where a user taps his ID card and receives an error message even though they should be authorized to access the shop. The two error messages seen within this class are "AES Encryption Error" and "CRC Check Failed."

AES encryption errors occur when the iPad receives a message from the interface device with an invalid message. This could be caused by improper encryption on the interface device; however, in all the cases we encountered it was caused by data corrupted in transit over Bluetooth. This could be resolved by having a signaling protocol to let the interface device know if a packet was or was not received successfully. This was not implemented as we decided to store the sensitive RFID card information in memory on the interface device for as short of a period as possible. By freeing and zeroing out the memory immediately after sending the card number, the is no risk of the card data being extracted from memory or accidently being re-sent later due to a bug.
CRC check failed errors are caused by the data being corrupted in such a way that the AES decryption still succeeds but the data is incorrect. After decrypting the data, the iPad computes the CRC16 checksum based on the received card number and compares it to the checksum in the data packet. If they do not match, it raises an error. This could also be solved by the mechanism described in the prior paragraph; however, this problem is extremely uncommon and rarely seen outside test environments.

2.2.3.5.2 Card Not Reading

The "card not reading" class includes problems where a user taps their badge on the reader and does not receive any feedback. This is distinct from an "incorrect error" problem where the user's badge reads but an error is displayed even though they should have been granted access. The confusion from this error generally stems from the beeping behavior of the HID card readers. By default, the reader will automatically beep whenever it reads a card regardless of if it was successfully processed or not. This leads some users to believe their card has scanned correctly even though it hasn't.

One of the causes of this error was the interface device software crashing due to an invalid memcpy call. This was resolved by fixing the invalid call and enabling the ATMega watchdog timer. The watchdog timer is a function built in to the embedded processor for the interface device where the user calls a reset function every time the program loop runs. If a certain amount of time elapses without the reset function being called, the device will reboot the embedded processor (used in this application) or trigger an interrupt. The combination of these two solutions resolved most of the card not reading problems. This bug still occasionally

manifests and can be resolved with a reboot of the interface device by disconnecting and reconnecting the AC power cord. Disconnecting the power cord will reboot the card reader and interface device but not the iPad as the it has an internal battery. Upon reconnecting the AC power, the reader will start and connect in 1-5 seconds.

This type of issue is difficult to debug due to its infrequent occurrence and the limited debugging tools available for embedded processors.

2.2.3.5.3 Ghost Reads

The ghost read problem has been one of the more elusive problems encountered during the deployment. It could be reproduced when the enclosure was closed; however, was unreproducible when attached to a development console. Ghost reads are when the iPad thinks it has received a card read but no user tapped their card. A number of theories were investigated as to why this may have occurred with the most likely having to do with interference. An investigation into the server logs of invalid card events revealed a pattern that all the invalid swipes were 4-bit swipes which was a supported card length in the interface device software. By excluding all card lengths that were not 28-bit or 35-bit swipes, we were able to eliminate this problem.

2.3 Badge Printing

Whenever a user checks into the shop, an ID badge is printed on a badge printer situated at the entry way (See Figure 11). The purpose of the badge is two-fold: to ensure all users in the facility are authorized and checked-in as well as to make sure users know what tools they are allowed to use. This section will detail how the badge printing solution works and provide background regarding why the badge system was needed.

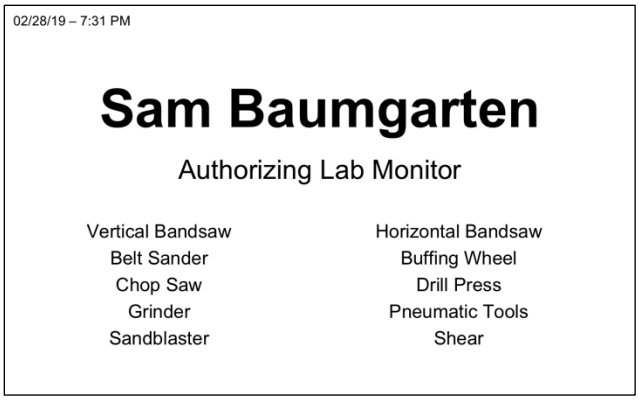

Figure 11. An example of a name badge that is printed when a user checks into a facility.

2.3.1 Background

The badge printing solution was not part of the original system specification. Earlier versions of the system used badges; however, we were hopeful that the lab monitor iPad would make it such that unauthorized people did not walk in. After the system was deployed in January of 2019, we spent several hours monitoring how users interacted with the system. We found a number of people entered the shop without checking in. In addition, we found that some people operated tools they were not checked off on.

Figure 12. The Brother QL-570 printer used for initial testing of the badge printing solution.

2.3.2 Hardware

The prototypes of the badge printing solution were implemented with a Brother QL-570 printer (See Figure 12). The QL-570 printer is a professional desktop label printer with a print speed of 4.33 inches per second. It can print at 330 dpi which is sufficient for this application. The primary drawback to the QL-570 is its print technology. The printer only supports a method of printing called direct thermal printing. Direct thermal printing works using specially made labels made of thermochromic paper which is a paper that changes color when heat is applied. While these labels are extremely convenient, they also are expensive on a per unit basis. The name labels used with this printer from an authentic supplier cost approximately $0.0997 per label on rolls of 300 labels. During non-peak times, the shop can see upwards of 150 shop sessions per day meaning the cost of using the QL-570 would be approximately $14.55 per day and would require shop staff to replace the labels every 1-2 days.

Figure 13. The Zebra ZT-230 industrial label printer used in the production system.

One of the alternative printers we looked at was the Zebra ZT-230 (See Figure 13). Zebra is an industrial label printer manufacturer and produces label printers designed for high-volume applications. The ZT-230 has a high upfront cost but supports both direct thermal printing as well as thermal transfer printing. Thermal transfer printing works by using regular paper and melting a material off a consumable called the ribbon onto the label to print. Thermal transfer printing requires more advanced printers, but reduces the cost per label to $0.0122 per label. The ZT-230 prints at 203 dpi or 300 dpi and can print at 6 inches per second. The ZT-230 can hold rolls up to 8 inches in diameter which means it can hold up to 2400 name badges. At the current usage rates, each roll lasts 16 days and the cost per day to operate is $1.83.

	QL-570	ZT-230
Upfront Price	$299.99	$712.80
Cost per Label	$0.0997	$0.0122
Cost per Day	$14.55	$1.83
Roll Replacement Time	2 days	16 days

Table 2. An analysis of the cost of a desktop printer (QL-570) and an industrial printer (ZT-230).

Based on the data in

Table 2, the breakeven point for the ZT-230 is 5,518 labels which would be achieved in 36 days. This analysis led us to the conclusion that we should use the ZT-230 printer.

2.3.3 Printing from the Backend

Early in the printer prototyping process, we decided to print the labels from the backend rather than having each client determine how it should print a label. To achieve this, we needed a way to have the server send a print job to a printer. We evaluated two services that can be used to enable printing from a remote server: PrintNode and Google Cloud Print.

2.3.3.1 Cloud Printing Service Selection

PrintNode is a hosted printing service that provides an API that can be used to print PDFs to clients running the PrintNode client application. PrintNode's "Essential" plan includes 5,000 prints per month and costs $9 per month. PrintNode's strengths are its simple API and our experience integrating with it from prior projects. PrintNode's primary drawbacks are its price, when compared with Google Cloud Print and the company's small size. PrintNode has recently revamped their entire service which did not include backwards compatibility with their old API. A change like this would have broken our system if we had written it before the change. This paired with our inability to analyze the strength of PrintNode's business health led us to believe there was a risk of a PrintNode integration breaking in the future.

Google Cloud Print (GCP) is Google's offering for remote printing. It is deeply integrated with Chrome and Google's Chromebook offerings. GCP's strengths include it being supported by Google, its price (free), its flexible API, the potential future availability of GCP ready label printers, and the wide range of host-computers compatible with GCP (any computer able to run Chrome). GCP's primary weakness is its authentication scheme. GCP authenticates with OAuth2 as it is primarily designed for a specific user to associate their account with it, not for a service account to be used. This means if the service account's refresh token expires, a staff member will need to reauthenticate in the admin panel.

Ultimately, we settled on using GCP due to its free price and our confidence that Google will continue to support the service.

2.3.3.2 Authentication

To provide the backend with an authentication token that can be used to print, a member of the Washburn staff can visit https://sms.mfelabs.org/admin/cloudprint/authenticate. This will redirect the staff member to the Google OAuth2 flow. Google provides the backend with a refresh token. This refresh token allows the backend to get new access tokens without the staff member re-authenticating. A staff member will only need to reauthenticate when the Google account's password is changed, or the backend's access is manually revoked. Revoking the backend's access and re-authenticating is a valuable troubleshooting technique and will fix many authentication problems should they come up.

2.3.3.3 Badge Format

GCP supports many content types for printing. The backend application sends content with the content-type "text/html." The HTML is generated by an ERB template called "badges/default." GCP converts this HTML to a PDF before sending it to the printer. GCP's conversion doesn't support some modern HTML standards such as flex-boxes so the badge template heavily relies legacy HTML table layouts to ensure compatibility. The server prints badges with the options listed in Table 3.

Page Orientation	1 (portrait)
Media Size	height_microns: 99800 vendor_id: 275 width_microns: 62000
Margins	All sides 0
Page Range	1-1

Table 3. Print settings used for jobs sent to GCP.

2.3.3.4 Printer Client

The printer is unable to receive jobs from GCP directly. This necessitates an interface between the printer and GCP. Google Chrome has a GCP client built in (https://support.google.com/cloudprint/answer/1686197). This client enables us to share a local printer, in this case, the label printer, with GCP. To connect the local printer with GCP, we use an Intel Compute Stick running Windows 10. The Compute Stick has the Zebra printer drivers and Google Chrome installed on it. With the Compute Stick setup to furnish jobs, the API can submit a print ticket to the GCP API which will forward the ticket to the client associated with the printer being used to print the job. Google Chrome then handles the process of sending the print data to the Zebra drivers which handle the printing of the label.

2.3.3.5 Configuration

Each device can optionally have a printer associated with it. The printer is set using the *badge_printer_id* field on the Device entity. This field should be set to the printer identifier found in GCP. The identifier can be found either through the GCP API or through the GCP console by clicking a printer, then clicking "Show Print Jobs," and copying the ID in the search bar. More information about printer identifiers can be found in Section 4.4.1.4.

By default, a badge will print for every lab user. This can be changed on a per lab user basis by changing the *should_print_badge_on_check_in* field on the LabUser entity. If this field is set to false, a badge will not be printed when the user checks in. This can be used for people that are constantly in the facility and want to use a reusable badge. More information regarding this setting can be found in Section 4.4.1.3.

2.4 Digital Signage

In the entryway to the Washburn Shops Manufacturing Labs, a large TV was mounted to display the current state of the shop. The screen shows an open and close message defined on the Shop model as well as the active lab monitor (See Section 4.4.1.1 Shop).

The software for the signage is implemented as a web application and is discussed in detail in Section 4.3.1 Shop Digital Signage Display.

The hardware driving the TV is a Raspberry Pi 3B+ that is secured behind the TV. On boot, the device will automatically load the webpage for the signage in kiosk mode. The Raspberry Pi connects to WPI-Wireless with the certificate of a Washburn full-time staff member using wpa_supplicant based on the instructions at https://it.wpi.edu/article/Connect-to-WPI-Wireless-Using-a-Raspberry-Pi.

3 iOS Software Architecture

There are two iOS applications that are used to operate this system: Washburn Controller and Washburn Entry. Washburn Controller is an application used by the active lab monitor on the WPI owned iPad Mini to handle day to day operations for the facilities. Washburn Entry is the application running on the kiosk devices (see 2.2 Kiosk Device).

The iOS software is contained within three projects, WashburnAPI (framework), WashburnController (app), and WashburnEntry (app). All three of these projects are written in Swift 4.1 and will be detailed in the following sections.

3.1 WashburnAPI Framework

The WashburnAPI framework is responsible for networking and request handling when communicating with the Washburn backend API (See Section 4, Backend). It is compiled into a dynamic framework and included in all client applications.

Client applications do not directly interface with the backend API, instead routing all requests through the WashburnAPI framework. This provides a number of key benefits including making changes to the API less complicated, increasing code reuse between the client applications, and reducing common bugs due to improper code reuse.

The classes within the WashburnAPI framework are functionally divided into a few key types: Requests, Managers, and Helpers.

3.1.1 Class Types

3.1.1.1 Requests

A request class defines a specific HTTP request that can be made to the Washburn backend. All requests definitions are contained within the WashburnAPI framework and implement the APIRequest protocol which is defined as follows:

```
public protocol APIRequest {
    associatedtype ResponseType

    var method: HTTPMethod { get }
    var path: String { get }
    var parameters: [String: Any] { get }
    var authenticationProvider: AuthenticationTokenProvider? { get }

    func process(response: APIResponse) -> Promise<ResponseType>
}
```

A request must define what HTTP method to use (GET, POST, PUT, DELETE, etc.), the parameters to include, an optional authentication provider (an object to use the authentication

token from), and a method to process the raw response and convert it into the requests return type (usually a model object or an array of model objects).

To execute a request, an instance of a class that implements the APIRequest is initialized (implementations contained in “Requests” folder) and then passed to the execute method of WashburnAPI manager (WashburnAPI.swift) which returns a Promise. The WashburnAPI manager uses the NetworkWrapper to convert the request definition into an HTTP request which is then sent to the server using the Alamofire framework (See 3.1.3 Request/Response Conversion).

To upload an asset such as an image, a multipart form-data request must be used (defined in RFC 2388). The multipart form-data encoding scheme enables a file to be sent in the body of an HTTP request. All asset upload requests subclass MutlipartAPIRequest (See example in WashburnAPI/Requests/LabUser/UpdateLabUserProfilePhotoRequest.swift).

3.1.1.2 Managers

Managers are classes with only static methods that provide some sort of execution functionality (in contrast with helpers which provide generative functionality). The three primary managers in WashburnAPI are WashburnAPI.swift which keeps track of the session state and allows clients to execute requests, ShopSocket.swift which manages the web socket connections, and DeviceManager which keeps track of the current device state including attributes such as the active authentication token and device parameters. The device manager can be accessed using WashburnAPI.deviceManager.

3.1.1.3 Helpers

Helpers are generally used to transform or generate data for use in the rest of the framework. The helpers used in the WashburnAPI framework are RequestHelper which generates authentication headers for the network request builder and ParsingHelper which is used to handle parsing in responses such as converting a string to a date.

3.1.2 Shop Socket Manager

The shop socket manager is the class responsible for keeping an active web socket connection open with the backend and handling subscriptions to various channels. Rather than having each view controller in the client applications handle their socket connections directly, the shop socket manager uses an observer pattern to allow client applications to receive socket messages while leaving the API framework to handle lifecycle events.

Client applications can subscribe and unsubscribe to a socket using the following methods:

```
public func addObserver(shop_id: String, observer: ShopSocketObserver)
public func removeObserver(shop_id: String, observer: ShopSocketObserver)
```

ShopSocketObserver is a protocol to be implemented by the client application to receive delegate events and is defined as:

```
public protocol ShopSocketObserver: class {
    func shopDidChange(shop: Shop?, error: Error?)
}
```

When a client application tries to add itself as an observer for a given shop channel, the socket manager checks if it is already subscribed to the channel. If it is, it will simply add the observer specified to the observer list for that channel. If the manager isn't already subscribed to the channel or if it isn't connected to the web socket server, the manager will create a channel object, add the observer to it, and add it to the list of pending channels. If the manager is connected but not subscribed, it will attempt to flush the pending channels which will attempt to subscribe to them. If it isn't connected it will not do anything. When a connection is established to the socket server, all pending channels will attempt to be flushed again.

3.1.3 Request/Response Conversion

One of the key functionalities of the WashburnAPI framework is to convert a request definition and its associated data into a HTTP request and to process a response and convert it into a usable object model.

3.1.3.1 Request Conversion

Request conversion occurs in NetworkWrapper.swift. First, the RequestsHelper class is used to convert the AuthenticationTokenProvider into a HTTP header (see section 3.1.3.3). Next, the path requested in appended to the base URI (https://sms.mfelabs.org/api/v1/) defined in WashburnAPI.swift to create a full URL. This URL paired with the method and parameters from the request definition as well as the headers computed are used to create an Alamofire request. This request is then executed.

3.1.3.2 Response Conversion

When the server responds to the HTTP request, the response body is converted to JSON. The framework ensures the response is a dictionary with keys of type String. If this is not true, the promise fails with a parsing error. If the check succeeds, the framework next checks if there is a key named "data" in the response. If this key exists, the framework successfully fulfills the promise with an APIResponse object with the data field set to this extracted value. If the key does not exist, the framework next checks for an error key. If this error key exists and is defined in the default error format (a dictionary with field names as keys each with an array of error messages as the value), the framework fails with an error message defined based on this dictionary. If none of these keys exist, the API fails with an unknown error response.

After this level one processing is completed, the WashburnAPI manager takes the APIResponse object and calls the `process(response: APIResponse) -> Promise<ResponseType>` method on the request definition object. The request definition object then parses the response and returns its generic ResponseType.

3.1.3.3 AuthenticationTokenProvider

AuthenticationTokenProvider is a protocol that requires the following method to be implemented:

```
func getOrRenewToken() -> Promise<Token>
```

It returns a promise to allow for it to make asynchronous requests to obtain a token. The two classes that implement AuthenticationTokenProvider in WashburnAPI are LabUser and Device. In the case of device's implementation, it will request a new token from the server using the saved refresh token if the token is set to expire within five minutes. LabUser will simply return the stored token as LabUser tokens cannot be refreshed without re-authenticating.

3.1.4 Dependencies

The WashburnAPI framework has a few external dependencies installed with Cocoapods. The dependencies it uses include:

- Alamofire
  - For low-level networking
- ActionCableClient
  - Handles ActionCable messages (web sockets protocol)
- Promises
  - Framework developed by Google to facilitate writing asynchronous code

3.2 Washburn Entry Application

The Washburn entry application is the iOS app running on the iPads acting as the kiosk devices (See 2.2 Kiosk Device). It is responsible for handling lab user check-ins, check-outs, registration, and lab monitor video calls. It links to the WashburnAPI framework documented in section 3.1.

3.2.1 Navigation Style UX

Most iOS applications are designed to be used by a single user on a personal device. This leads to a number of assumptions about how a user expects to interact with an iOS typical application and how then expect to navigate it. Unlike these typical applications, the entry app is used as a kiosk meaning the user of the application is constantly changing and someone approaching the kiosk likely does not know what state the kiosk is in. For this reason, we chose a hierarchical navigation structure where the application is configured to automatically reset to a known state (See 3.2.2.1 Primary View). The Apple Human Interface Guidelines define hierarchical navigation as a structure where you will:

> *Make one choice per screen until you reach a destination. To go to another destination, you must retrace your steps or start over from the beginning and make different choices.*

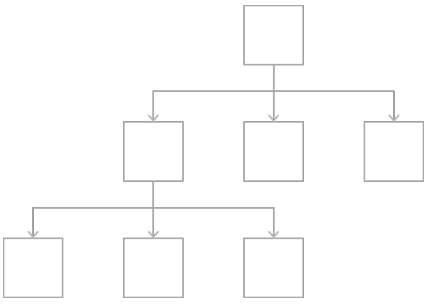

Figure 14. A visual representation of a hierarchical navigation structure.
Apple Human Interface Guidelines

This navigation structure is a good fit for a kiosk style application as there is a set start point where any user can identify a next step and the behavior of the application does not rely on user knowledge of the prior state of the application.

3.2.2 User Interface View Types

3.2.2.1 Primary View

The primary view users will interact with is a full-screen view referred to as the start screen which will be visible when they approach the shop. The start screen serves as a homepage and the user can access any other screen in the application from this point. The screen is very simplistic to aid in creating a simple user experience. The most noticeable elements are the graphic depicting tapping an ID badge on the card reader and the prompt instructing the user to tap their ID badge on the card reader (See Figure 15). At the top of the screen, there are two buttons: one for deliveries and one for visitors. Both of these buttons simply place a video call to the lab monitor (See 3.2.6 below). The only two direct actions a user can take are tapping an ID badge or pressing one of the two buttons. The application will automatically direct them to the correct screen based on their action and the application/backend state.

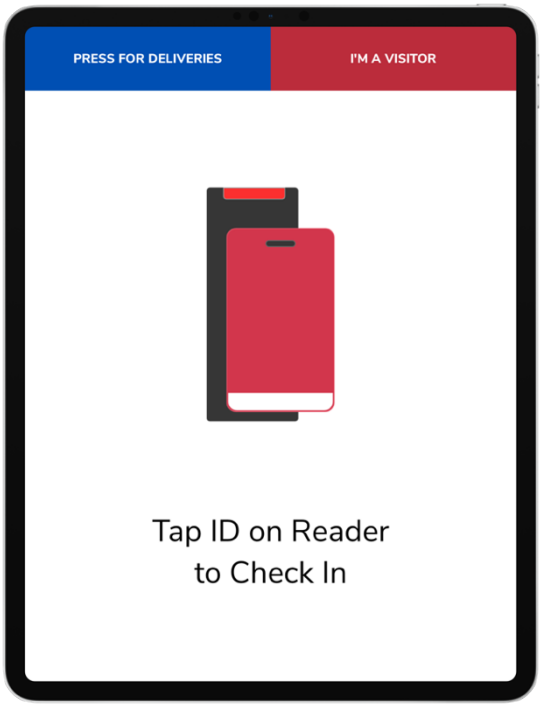

Figure 15. The primary view of the Washburn Entry application.

3.2.2.2 Modal Views

All user interaction with the kiosk device occurs in popups that are referred to as modal popups, form sheets, or alerts which are small modal overlays that partially occlude the primary view, often using a darkened background. The Apple Human Interface Guidelines define this style as follows:

> *Appears centered on screen, but may be repositioned if a keyboard is visible. All uncovered areas are dimmed to prevent interaction with them. May cover the entire screen on smaller devices. Use for gathering information.*

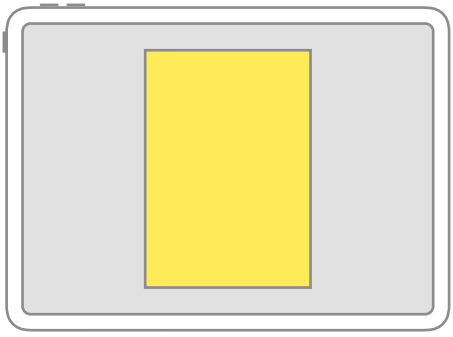

Figure 16. A representation of how a form sheet should appear.
Apple Human Interface Guidelines

The built-in implementation of form sheets supports custom view sizes using the preferredContentSize variable on a UIViewController, but does not support view size changes which is a functionality needed for the entry app.

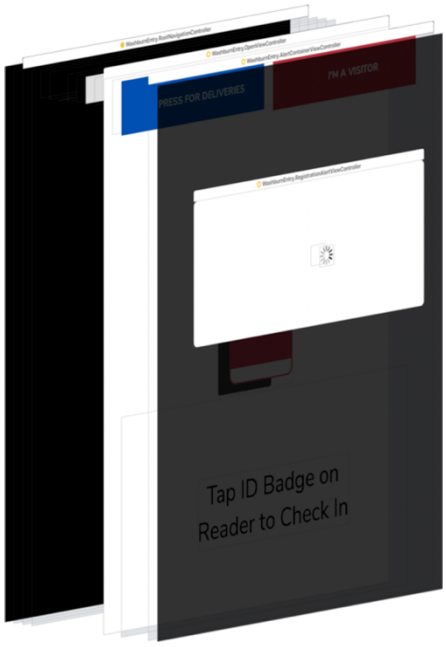

Figure 17. A 3D visualization of the view hierarchy of our form sheet implementation.

Instead of using the built-in implementation, a custom version was written. This custom version gave us more control over the view hierarchy enabling dynamically sized and animated modal views. Alerts using our implementation have a few distinct layers. First is the superview container which is the view that holds the modal view container being presented. In the entry application this is always the navigation view controller which is the root node of the view hierarchy. Next is the modal container (implemented in Alerts/AlertContainerViewController.swift). This view controller contains the background view as well as the content view that will be shown. The container view controller is also responsible for animating the content and background views in/out when the view is presented/dismissed, shifting the view up when the keyboard is presented to keep it centered, dismissing the alert when a tap in the background is detected, and changing the content view without creating a new container.

3.2.3 Alerts/Modal Views Types

Whenever a user performs an action with the entryway kiosk either an error, success, form, or call modal view will appear (See 3.2.2.2 for more information on modal views). These modal view types will be described in this section.

3.2.3.1 Success Alert

Success alerts show when an action has successfully been performed. They are indicated by the presence of a green checkmark in the modal view and have a message label to inform the user of the reason for the success alert. In addition, a success chime is played through the device speakers to provide auditory feedback. Some actions that can trigger a success alert include:

- Successfully checking in
- Successfully checking out
- Completing registration

3.2.3.2 Error Alert

Error alerts are visibly similar to success alerts (both are subclasses of AnimatedAlertIconViewController). The primary differentiator is a red animated “X” in place of

the success alert's checkmark. Error alerts play an error chime to ensure users are aware of the error. Some actions that can trigger an error alert include:

- Not having access to shop
- Card read error occurred (See 2.2.3.5 for common errors)
- Server connection errors

3.2.3.3 Form Alerts

Form alerts are used in cases where more information is needed from the user before an action can be sent. The only place form alerts are used is when a user taps a card that is not already in the system. In this case we need to request the user's WPI ID number to see if they have a shop authorization and just need to associate their RFID card number to their account or if they are in-fact not authorized to access the shop. The user completes this registration process in a form alert called RegistrationAlertViewController.

3.2.3.4 Call Modal Views

When a user places a video call, we present a call modal view. This modal view controller manages the call state, plays an outgoing ringtone, and shows the user a preview of what they will look like. This modal view implements AlertViewControllerProtocol allowing it to end the call whenever the modal view is dismissed. A bug still exists with this view where if the view is dismissed while the video session is being created (0.25-1 second period), the call will continue even though the alert is dismissed. If the lab monitor answers the call then hangs up, the call will terminate; however, if the lab monitor doesn't answer, the call can continue in the background. The kiosk will leave this state when it places another outgoing call, is restarted, or the call connection fails. This bug can be resolved by disabling the cancel button until the video session is created or by keeping track of if the modal view is still presented and dismissing it once a video session is established if it is no longer showing. More information about calling can be found in Section 3.2.6 below.

3.2.4 Device Registration

Each time the app is launched, it shows a view controller called the device registration view controller. This is a captive primary screen that both ensures the device is authenticated with the backend and presents the user interface needed for the initial enrollment with the backend. Whenever the application is launched, it uses the WashburnAPI Device Manager to attempt to authenticate the device. The first time the app is launched, it will not have a saved refresh token so the WashburnAPI framework will make a request to the backend to create a new device and store the returned refresh token (See Section 7.2 Authentication). Once the device has a refresh token, it can make requests to the authenticate endpoint and receive the device information. One piece of information returned is the display identifier. This display identifier enables an admin to identify a device when approving devices in the administration panel. The device registration view will show this display identifier as well as a refresh button that will re-attempt the authentication.

Once a device is approved, the device registration view controller will switch to the start screen. On subsequent launches, the initial authentication request will succeed so this view controller will not show.

3.2.5 Card Reader Connection

When the application is launched, an instance of the BluetoothRFIDManager class is created. This class begins scanning for BLE devices with service UUID 0xFFE0. This service UUID was randomly selected by the manufacturer of the HM-10 module; however, it could be set to any 2-byte number assuming the value is the same as the UUID set on the BLE module. Whenever it finds a service with this service UUID, it checks if the device's advertising packet specifies the reader name entered in the administration panel. If it does, a connection is established. Communication with the reader follows the protocol defined in Section 2.1.3 Reader Communication.

3.2.6 Calling

When users want to initiate a call to the lab monitor from the entryway, they can press one of the two large call buttons at the top of the screen (See Figure 18). When one of the buttons is pressed, a call modal view is presented (See Section 3.2.3.4 above). When the view is presented, the call process is started.

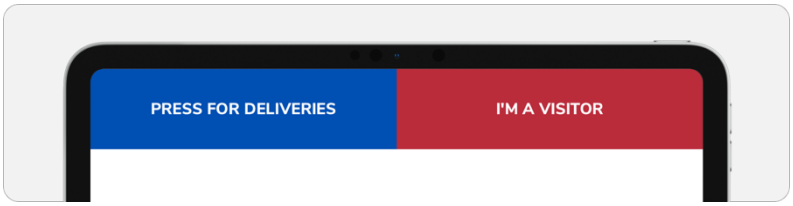

Figure 18. The buttons used to place a call to the current lab monitor.

Video calls in the entry application are powered by Twilio's Programable Video API. The Twilio service simplifies the process of deploying P2P and group video call systems.

3.2.6.1 Call Process

In order to begin a call, a Twilio room must be created and an access token must be obtained. A room is a defined by Twilio as "a multi-party communications session among users in your application, where users can share and receive real-time audio and video tracks with one another." (https://www.twilio.com/docs/video/api). Each time a user places a video call, a room is created.

To do this, a HTTP request is made to the place call endpoint defined in Section 7.3.5. This API endpoint obtains and returns a room SID as well as a Twilio access token. The room SID is a 34-character string generated by Twilio that uniquely identifies a room. The access token is a JSON web token (JWT) that uniquely identifies a user's identity and the room they are able to connect to (https://www.twilio.com/docs/iam/access-tokens).

While this request is being made, the application plays an outgoing call chime to inform the user their call is being placed.

Once the client has a room SID and access token, it uses the Twilio iOS SDK to create a Twilio Video session and connect to the room created by the server. While this is happening, the server is sending push notifications to the lab management devices which will start to ring (See sections 3.3.5 Incoming Calls and 7.3.5 Place Call). When one of these devices accepts the call, the connection will be established. The audio from the lab monitor's device will be streamed to the entry kiosk and the audio and video from the entry device will be streamed to the lab monitor. Right now, there is no mechanism for if a lab monitor rejects the call or doesn't answer. In this case, the kiosk will continue showing the call modal view until the user presses cancel.

When a user ends the call after the call is established, the call will automatically end on both the kiosk and the lab monitor device. If the call is canceled before the lab monitor has answered, the lab monitor device will continue to ring; however, when they answer, no connection will be established. Pressing "hang-up" or tapping outside the modal view will exit this state. In rare cases, if a user presses cancel between when the place call request is made and the Twilio Video session is established, the call can be started on the kiosk even though the call modal view isn't showing.

All the logic described in this section is implemented in the VideoChatViewController class.

3.2.6.2 Call Pricing

Twilio charges a flat rate per participant per minute of $0.0015 for video calls. All Twilio calls are rounded up to the nearest minute for billing purposes (https://support.twilio.com/hc/en-us/articles/223132307-How-do-you-round-minutes-for-billing-). The average video call is less than 1 minute meaning most calls cost $0.003. In the two months since the system was deployed, we have spent $0.62 on video calls.

3.3 Lab Monitor Controller Application

The lab monitor controller app enables the current lab monitor to view all checked-in users, check the privileges of any user, view a limited audit history of any user, grant temporary authorizations (See 7.3.6 Grant Temporary Authorization), receive notifications, and close the facility. The app runs on an iPad Mini that is carried around by the current lab monitor.

3.3.1 Administrative Tasks

The app is designed to facilitate tasks that a lab monitor may have to do while being in-charge of the facility and is not for administrative tasks which are done in the administration. This decision was made because the lab monitor device is unable to positively identify a specific user. The app assumes the current lab monitor is the person using the device. While this form of identification is sufficient for tasks such as closing the shop, it is not sufficient for controlling access to sensitive administrative information. The admin panel has an authentication system that enables the system to positively identify a user which is why it can be used for administrative or sensitive tasks. In the future, if a more robust authentication system is

developed for the lab monitor iPad, some administrative tasks such as approving tool authorizations can be moved to it.

3.3.2 User Interface

The primary Washburn Controller user interface is a split view controller (See Figure 19). A split view controller is a type of view controller that has two child view controllers: a master and a detail. The master view controller is used to show a table or list of all the users currently in the facility. The detail view controller shows information about a specific selected user. In landscape mode on an iPad, the master view shows as a sidebar on the left side of the screen and the detail view shows in the remaining portion of the screen. In portrait mode, the detail view fills the screen and the master view can be accessed by swiping from the left side of the screen. If the application is deployed on an iPhone or other similarly sized device, the master view controller will show as a table view and the detail view will be pushed onto the navigation controller stack whenever a lab user is selected. The user can return to the master view by pressing a back button that shows in the upper left corner.

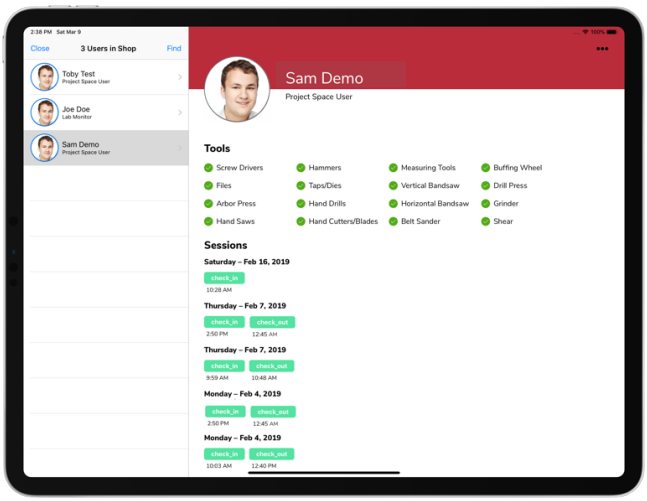

Figure 19. A screenshot of the primary interface of the Washburn Controller application.

3.3.3 Sidebar

The sidebar shows a list of all the users currently in the facility. Each row of the table shows a user's display name, role, and profile picture. The active lab monitor's picture has a red border around it and all other users' profile pictures have a blue border to make it easy to quickly find the current lab monitor in the list. The list will automatically update when the app has an active web socket connection with the server and a change occurs (See Section 3.1.2 Shop Socket Manager). If the user wants to manually refresh the list, he can pull down on the list which will cause a loading indicator to appear and the list to be reloaded (See Figure 20). The sidebar is also where lab monitors can find a user that isn't currently in the facility. When a user taps the find button in the navigation bar, a search form appears as a modal form sheet. This enables the user to check if someone is in the system, view his shop audit history, or grant temporary

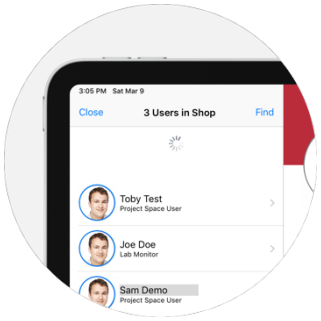

Figure 20. A screenshot of the user list while it is reloading.

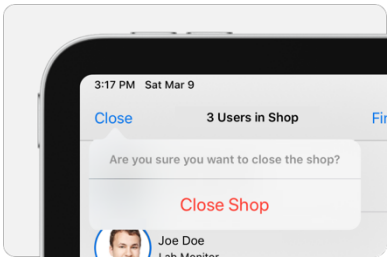

Figure 21. The popover menu to confirm the user intended to close the shop.

authorizations to users that are not basic users (See 7.3.6 Grant Temporary Authorization). The navigation bar is also where lab monitors can close the facility. When a lab monitor presses the "Close" button in the upper left corner, a confirmation prompt appears (See Figure 21). If the lab monitor taps outside the popover menu, it will be dismissed. If he taps the close shop button, a close shop action will be executed (See 7.3.4 Close Shop). A notification will be sent to all the lab management devices, including the one the shop close was initiated from, saying that the shop has been closed. This provides the initiating lab monitor with feedback that the action worked and provides support for future use-cases where multiple lab management devices are in use simultaneously.

The sidebar is implemented in the SidebarViewController class.

3.3.4 Detail View

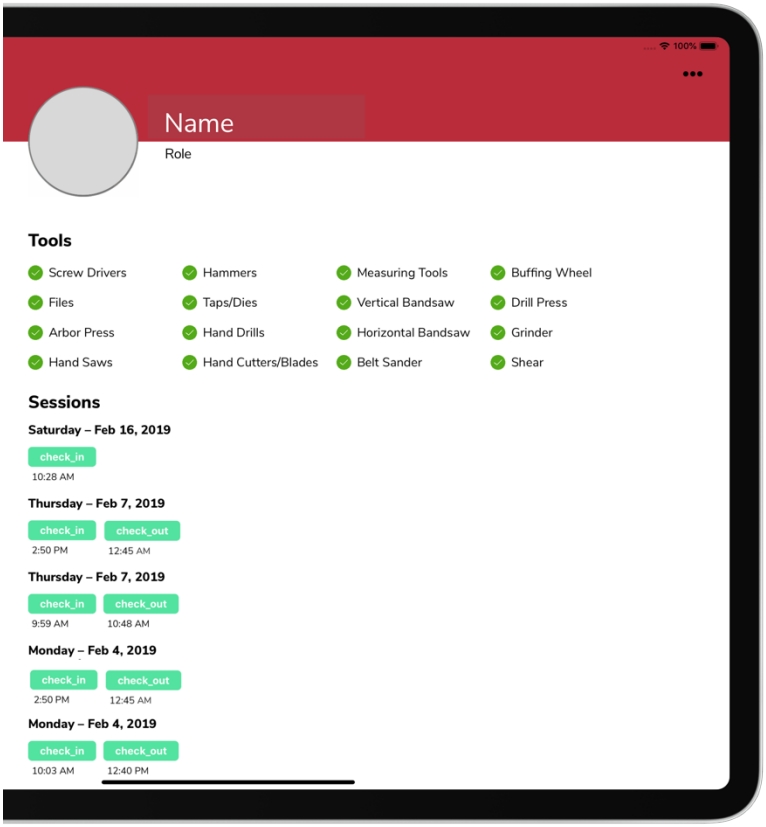

Figure 22. A screenshot of the detail view controller used to display user information.

The detail view shows a lab monitor information about a selected lab user (See Figure 22). It is accessed by selecting a user in the sidebar user list or using the find feature (See 3.3.3 above). The view can be broken down into three distinct sections: the header, tools list, and sessions history.

The header provides information about the user including his name, photo, and role in the shop. The header also is where the action menu button is located. Pressing the icon in the upper right corner will present a popover that allows a lab monitor to perform an action related to a user. The popover will only show actions applicable to the selected user. The possible actions include making the user the current lab monitor, taking a new photo for the user, and granting the user a temporary authorization (See Section 7.3.6).

The tools list shows all the tools a user is authorized on (tools where an active tool authorization exists). Tools the user is not allowed to use will not show in the list. If a user has a

pending tool authorization, it will show in the list; however, it will have an orange circle with a dash instead of the green checkmark shown next to other tools.

The sessions history section shows a user's last 10 shop sessions and the events associated with that session. The session displays the date (from start_time) as the header text and the time each event occurred below the event bubble. Event types that end in "denied" display in red, "check_out_by_close" events show in an orange color, and all other events have a green background. If a shop session ends in a "Check Out by Close" event, it means the user did not check out and was automatically checked out when the lab monitor closed the shop. If a session ends in a "Check Out" event, it means the user did check out. If there is a "Check Out" event preceded by a "Check Out by Close" event, it means the user was still in the shop when it was closed; however, still correctly checked out within 30 minutes from the shop closing when they left (See 7.3.4 Close Shop).

The data for the session history is fetched when the detail view is loaded using the API endpoint defined in Section 7.5.4 Get Shop Sessions.

The detail view is implemented with a UICollectionView. The header is implemented as a supplementary view and the other sections are implemented as UICollectionViewCells. The detail view can always be refreshed by pulling down on the view using the UIRefreshControl behavior built into the UICollectionView class.

3.3.5 Incoming Calls

When a user at the entryway presses one of the call buttons, a push notification is sent to all devices with the role "lab_management" (See Section 3.2.6 Calling). These notifications are received in the AppDelegate class. When there is an incoming notification, the app is started in the background if it isn't already open and the following method is invoked:

```
func pushRegistry(_ registry: PKPushRegistry,
                  didReceiveIncomingPushWith payload: PKPushPayload,
                  for type: PKPushType,
                  completion: @escaping () -> Void)
```

The push payload includes a key called "incoming_call" which is in the following format:

```
{
    "incoming_call": {
        "access_token": "",
        "room": "",
        "from": ""
    }
}
```

The information in the payload is passed to CallController which interfaces with the CallKit framework (https://developer.apple.com/documentation/callkit).

The Apple documentation for CallKit describes its function as to “Display the system-calling UI for your app’s VoIP services, and coordinate your calling services with other apps and the system.”

When CallController dispatches an incoming call event to CallKit, the system incoming call UI is presented to the user. When they accept, the details from the notification payload are used to establish a video session in an instance of VideoChatViewController which is presented as a form sheet (See Section 3.2.2.2 Modal Views). The connection is established in the same manner as the Entry application (See Section 3.2.6 Calling).

3.4 Certificate Requirements

Both the entry and controller applications have certificate requirements that must be fulfilled annually. These requirements can be broken down into two different groups: APNS and code signing.

3.4.1 Code Signing

To maintain control of what software is able to run on iOS devices, all apps must be code signed. Right now, the apps running on the entry kiosks and the controller devices are signed with a development certificate which has the caveat that it expires every year. This means the applications will need to be re-signed once per year. To do this, both an Apple Developer account and Mac with XCode installed are needed (http://developer.apple.com). Once XCode is setup with the developer account, each device needs to be connected to the computer which will necessitate removing the iPads from their kiosk enclosures. After the iPads are connected, the application source code can be re-built and installed on the device. This can be avoided by distributing the apps using Apple’s B2B program which will enable the apps to be signed with a production certificate that doesn’t expire removing this annual maintenance requirement. Should the code-signing certificate expire, the iOS applications will no longer launch and will prompt a message stating that the certificate has expired. These certificates will expire 365 days after the certificate was last renewed and will need to be manually tracked by the shop staff.

3.4.2 Push Notification Certificates

The Washburn Controller application makes extensive use of the Apple Push Notification Sevice (APNS) for both alerts and incoming calls. Unlike code-signing, these certificates are only used on the backend. They expire annually and will need to be renewed. This can be done through the Apple Developer portal. Instructions on how to prepare the certificates can be found at the bottom of the readme file for the Houston library at https://web.archive.org/web/20181201002537/https://github.com/nomad/houston. Both the VOIP Services certificate and Production APNS certificates need to be renewed. They are loaded by the backend in config/initializers/houston_apns.rb. The VoIP certificate should be named “voip_apns_dev_wc.pem” and the alert APNS certificate should be named “wbc-apns-prod.pem.” If an APNS certificate is not renewed, calling will cease to work and the lab monitor will stop receiving notifications when a user completes a check-in or check-out action. The

certificate will need to be renewed 365 days from the prior renewal which should be noted on the shop staff calendar.

4 Backend

The backend is the web application that provides an API for the client applications, receives webhooks from other services, manages the data collected by the system, provides an administration panel, and serves user interfaces such as the entryway digital signage and quiz portal. The backend is also where all shop policies are defined, and authorization decisions are made. Information regarding the backend infrastructure and hosting can be found in Section 5, Web Architecture.

4.1 Software Architecture

The backend is primarily built using version 5.2 of the Ruby on Rails framework (Rails). Rails is a server-side framework for web applications that follows the MVC design pattern and uses the language Ruby.

The backend exposes an API that is used by client applications to manipulate the data stored by the backend. This API is covered in detail in Section 7, API Design.

The backend serves two frontend interfaces that can be viewed by regular users: the signage display and the quiz group embedded view. Both of these user interfaces are written using ReactJS and bundled with WebPack. Webpack is a Javascript module bundler that takes a Javascript project and compiles it into packaged assets for distribution. ReactJS is a reactive Javascript framework built by Facebook for creating responsive user interfaces. React component files, which have the extension JSX, are taken by Webpack and compiled into a single JS file. This file is then used to render the user interface presented to users. All the reactive frontend interfaces are contained within the app/javascript and app/views folders.

The admin interface is implemented using Thoughtbot's administrate library/gem. The interface's implemention files are contained within the app/controllers/admin, app/dashboards, and app/views/admin directories. The dashboards directory contains files that use the Administrate DSL to define how the dashboards for each model should display (more details in Section 4.2.4.4 below).

4.2 Code Structure

Ruby on Rails is designed such that all projects should follow an explicit code structure. The design choices made by the project maintainers are intended to assist engineers in storing their code in a manner consistent with all other Rails projects. All Rails projects have a few relevant root directories: app, config, db, and test. There are other directories in the codebase; however, they will not be covered in this section either because general information about them can be found in the Rails documentation or they are not relevant to the core functionality of the application.

4.2.1 Test Directory

The test directory contains all the tests written for the backend application. The directory almost mirrors the app directory to make it easy to match tests to the component they are intended to validate. The factories folder contains helpers implemented with a library called FactoryBot which are used to quickly create records to be used for test cases. The factory files are broken down by model.

4.2.2 DB Directory

The db directory contains the database schema generated by Rails, a seed file named seeds.rb, and a folder called migrate. The seeds file is a script that can fill the database with fake data for testing. This is extremely useful for development purposes. It can be invoked by executing the command “rake db:seed”. The migrate sub-directory includes Ruby files that define how to create a database. These migration files can either be used to create a new database or to make an older version of the database conform with the latest database schema. The migrations can be run by executing “rake db:migrate”

4.2.3 Config Directory

The config directory contains configuration files and scripts for the backend. Within the config directory, the environments sub-directory contains a file for each environment: development, test, and production. These files define configuration variables that need to change between environments. Common configuration variables are defined in application.rb. Routes.rb is an extremely important file that defines how incoming HTTP requests should be routed. Each route has an entry in this file along with the controller and action it should be routed to. The initializers sub-directory contains scripts that are run whenever the application is started. Adding a file to the initializers sub-directory will cause it to be automatically be run whenever an instance of the application is started.

4.2.4 App Directory

The app directory is where a majority of the application specific code is. This includes frontend javascript, controllers, models, actions, and other types of files that define the backend’s behavior.

4.2.4.1 Channels Sub-Directory

The channels sub-directory is where ActionCable channels are defined. Each file in this sub-directory defines a different channel and specifies the channel’s behavior when lifecycle events occur such as a user attempting to subscribe or disconnecting.

4.2.4.2 Commands Sub-Directory

The commands sub-directory is where service objects for the application are located. This includes the actions defined in section 7.3. Each file makes use of the SimpleCommand library and represents a distinct action.

4.2.4.3 Controllers-Directory

Controllers are one of the main objects in the MVC design pattern. A controller in the backend application defines the behaviors that should occur when an HTTP request is received. Each controller is associated with a scope defined in section 7.2.1 Scopes. All controllers extend application_controller.rb.

4.2.4.4 Dashboards Sub-Directory

The dashboards sub-directory is where the admin interface definition for each model is. The directory is filled with files named "{model_name}_dashboard.rb" that define how a given model should be displayed in the admin panel. This includes specifying information such as what fields to show on each page and what the name should show as in the sidebar.

4.2.4.5 Javascript Sub-Directory

The javascript sub-directory is where all the ReactJS frontend code is. This includes components, uncompiled scss (stylesheets written in a superset of CSS3), and packs. These files are compiled by webpack into consolidated js and css files and placed in the public root-directory so they can be served by the web server.

4.2.4.6 Jobs Sub-Directory

The jobs sub-directory is where the definition for each background job is. These jobs are run by a separate worker and generally queued by a controller. Using jobs ensures that HTTP requests can return information with a reasonable response time while still allowing long running tasks to be completed. All the jobs in this directory extend from ActiveJob::Base.

4.2.4.7 Models Sub-Directory

The models subdirectory is where all the backend's data models are located. Each model extends ApplicationRecord and defines the relationships between models, validations, and relevant helper methods.

4.2.4.8 Policies Sub-Directory

The policies sub-directory is where the shop policy and action policy are location. Policies in the context of the backend application are easy to modify files that define the business logic of a facility. This makes it easy for future changes to be made to policies such as the requirement to check in or check out without requiring a future maintainer to modify code across the codebase.

4.2.4.9 Serializers

The serializers sub-directory is where files that define how to convert objects into response are located. Each serializer is designed to take a certain object or set of objects and convert them into a hash or JSON response. One object type can have multiple serializers that generate different fields. For example, there is a class called ShopSerializer and a class called ShopSignageSerializer. Each of the serializer extends from ActiveModelSerialization.

4.3 User Accessible Views

There are two user interfaces that are designed for consumption in standard web browsers: the signage display and quiz group embedded view. As mentioned in the Software Architecture section, these views are implemented with ReactJS. This section will describe the purpose of each of these views and how they interface with the API.

4.3.1 Shop Digital Signage Display

The digital signage display is a web page that shows on a TV screen positioned in the entryway. The signage is extremely portable as the only hardware requirement for the devices showing the signage is that it must support a modern web browser such as Chrome.

When the signage page first loads, it makes a request to the Get Shop Signage endpoint defined in section 7.4.3. This API endpoint returns the minimum amount of information needed to show this display.

Web sockets are a technology used to enable a server to push information to a client when changes have occurred. The signage frontend makes use of a framework called ActionCable which is built into Ruby on Rails. ActionCable simplifies the process of using multi-channel web sockets by handling subscriptions, messages, and a number of other low-level constructs. Whenever an action occurs on the backend that could cause a change to the signage, a background job called ShopUpdateMessageJob is queued. This job sends a message to any client connected over a web socket connection alerting it of a shop change and sending updated shop information. The signage frontend listens for these messages and updates its UI whenever one arrives.

The interface can be accessed at https://sms.mfelabs.org/displays/signage.

4.3.2 Quiz Group Embedded View

The quiz group embedded view is an interface that is used for lab users to complete quizzes from their personal computers (See Figure 23). It is intended to be embedded in an external website such as https://mfelabs.org. The embedded view handles user information gathering, displaying a user's quiz progress, and enabling users to take quizzes they haven't completed yet. A quiz group can grant a user access to one or more shops or tools. A single quiz can be in multiple quiz groups which means that taking a quiz in one quiz group can grant a user permissions unrelated to the group they accessed the quiz from.

Figure 23. The quiz group embedded view widget from https://wpi.mfelabs.org.

When the view first loads, it presents a text field and prompts the user to enter his email address. Once the user presses enter or "Find," the view makes an API request to the API endpoint used to authenticate a user without a shop defined in section 7.2.2.2.2. We use the shop-less authentication method because quiz groups aren't explicitly related to a single shop. This API request will either return an authentication token if the user already exists in the system or a 401 error if they have not registered before. Once the authenticate API request returns a response, the view makes a request to the show quiz group API endpoint defined in 7.7.2. If no authentication token is specified on this request because the user is new, only the quizzes marked as "anonymous_allowed" will show a URL. This is used to allow a new user to take the "Basic Information" quiz which is a special quiz that is unscored and will create a new lab user upon completion. After taking that quiz, a user is able to return to the page and complete the other quizzes.

When a user clicks "Take Quiz," they are redirected to Typeform where they can complete the quiz.

The URL that is used to embed this view is https://sms.mfelabs.org/embeddable/quizzes?quiz_group_id={quiz_group_id}. Quiz groups can be made in the administration panel.

4.4 Data Model

The primary datastore for the backend is a Postgres database backed by AWS RDS (See Section 6.2.1 Amazon RDS). This datastore is accessed through Rails' query interface, ActiveRecord. Each model in the datastore is defined in a file in the app/models subdirectory. Relationships in the models are defined using a primary key and foreign key for one-to-many relationships and using a join model for many-to-many relationships. Each record of an entity is assigned a sequential ID to be used as its primary key that is unique within the scope of the entity. This section will discuss the models used in the application and how they relate to each other.

4.4.1 Entity Objects

For the purposes of this project, the term "entity objects" refers to objects that correspond to primary entities entered by a user either through a self-service interface such as the quizzes or through the administration panel. These entities are related by the other models in the system. In this system, the entity objects are as follows: Shop, Tool, LabUser, Device, Quiz, QuizGroup, and QuizResult.

4.4.1.1 Shop

A shop object refers to a physical location where the system is deployed.

Fields

display_name: The name to use when referring to this shop in user interfaces.
open_message: The message to show on signage when the facility is open.
closed_message: The message to show on signage when the facility is closed.
cloudprint_oauth_refresh_token: The refresh token to use to obtain authentication tokens for Google Cloud Print (See Section 2.3.3 Printing from the Backend*).*

Relationships

responsible_lab_user: The current lab monitor.
lab_users: The users currently in the facility (through LabUserShop).
authorizations: ShopAuthorizations referencing this shop
authorized_users: LabUsers that are authorized to use this facility (through authorizations)
quiz_groups: QuizGroups that grant access to this shop (through ShopQuizGroup).
devices: Devices in this shop.
Tools: Tools in this shop.

4.4.1.2 Tool

A tool refers to a piece of equipment in a specific shop that the system will track tool authorizations for.

Fields

name: The name that should be used when referring to the tool.
auto_approve: If false, new tool authorizations created from quiz groups will default to being pending and require an authorizing lab monitor's sign off. If true, tool authorizations will automatically go into effect upon completion of a quiz group.

Relationships

shop: The shop the tool is located in.
authorizations: ToolAuthorizations referencing this tool.

4.4.1.3 LabUser

A lab user object is used to represent a person in the system. A lab user can exist even if they do not have any authorizations. The lab user object is also used for the admin authentication system to retrieve and store an administrator's user information such as his display name.

If a lab user has a *swipe_number* but no *rfid_number*, the user will be prompted to register the next time he checks in. If a user has lost their WPI ID and obtains a new one from WPI, the rfid_number field should be cleared from the administration panel so the user can associate his new ID with his account. This process will be initiated when a user notifies the shop staff he received a new ID or that he is unable to check-in to the shop. This can be done through the call buttons on the entryway kiosk.

Fields
display_name: The user's full name. Users should be encouraged to use correct capitalization to ensure consistency.
swipe_number: The user's WPI ID number.
rfid_number: The RFID card number encoded into the user's WPI ID badge.
email: The user's email address.
created_at: Date the user was created.
updated_at: Date the LabUser record was last changed.
profile_photo: A string used by CarrierWave to keep track of where to find a user's profile photo.
role_display_name_override: An override for the role text displayed on the digital signage (for custom role names).
should_print_badge_on_check_in: Set to false if a user shouldn't have a badge printed on check in.

Relationships
active_shops: Shops the user is currently checked into (through LabUserShop).
shop_authorizations: ShopAuthorizations referencing this lab user.
tool_authorizations: ToolAuthorizations referencing this lab user.
lab_monitor_shops: Shops the user is currently the lab monitor for.
quiz_results: QuizResults referencing this use.

4.4.1.4 Device
A device represents a WPI owned device that is eligible to request authentication tokens. Each device has a specific role which governs which resources it has access to. Device records are generated by client applications calling the Create Device API endpoint defined in 7.6.1.

Fields
approved: If true, the device is eligible to make authenticated requests. If false, the device is still pending approval.
refresh_token: A token generated by ActiveRecord::SecureToken that is used by a client application to request new authentication tokens. If this token is compromised, it needs to be changed immediately.
display_identifier: A 6-character identifier randomly generated by the backend upon device creation that should be shown on a device's screen while it's pending approval to be used so an administrator can identify a specific device in the administration panel. This is not a sensitive field.
display_name: A user readable way to refer to the device in the administration panel.

role: What this device is used for. This will influence the permissions of this device's authentication tokens. Valid options are "check_in", "check_out", and "lab_management".
reader_name: A string used by some clients to identify which RFID reader they should connect to (See Section 2.1.3 Reader Communication*).*
badge_printer_id: The printer identifier assigned by Google Cloud Print that should be used to print badges for users that check in with this device (See Section 2.3 Badge Printing*).*
voip_apns_token: The Apple notification token used to send high-priority VOIP push notifications to this device.
alert_apns_token: The Apple notification token used to send alert push notifications to this device.

Relationships
shop: The shop the device is used for.

4.4.1.5 Quiz
A Quiz object represents a form/quiz created in an external form creation application such as Typeform or Google Drive (not implemented). When adding a quiz from Typeform, calculators and logic jumps should be used to make the quiz perform as expected (not allowing a user to continue until he passes). The max score must be added to the quiz record. Typeform only sends the total score, not the percentage so we need to check if a quiz score Is passing or not. The max_score field can also be used to only allow scores over a certain percentage by setting it to the minimum score needed to pass.

Fields
external_application: The application used to create/take the quiz ("typeform" only valid option).
external_identifier: The identifier used by the external application to identify the quiz. In Typeform, this identifier can be found in the URL of the quiz editor, the URL of the take quiz page, or through the API.
display_name: How the quiz should be referred to in user interfaces such as the embedded quiz view
max_score: The maximum score that can be achieved (for use with Typeform).
allows_anonymous: If true, the quiz can be taken without authentication even if a user doesn't exist. If false, a user must exist to take the quiz. This is used for the basic information quiz which creates user records.

Relationships
quiz_results: QuizResults referencing this quiz
quiz_groups: QuizGroups this quiz is a part of (through QuizGroupQuizzes).

4.4.1.6 QuizGroup
A quiz group is a collection of quizzes where when all the quizzes in the group are passed, a user receives either a tool or shop authorization. A quiz can be a member of multiple quiz groups and a single quiz group can grant access to multiple tools and multiple shops.

Fields
display_name: What the quiz group should be referred to as in user interfaces.
permission_level: For shop authorizations, what level should a user be granted access at.

Relationships
quizzes: Quizzes in this quiz group. All these quizzes must have been passed by a user to get an authorization (through QuizGroupQuizzes).
tools: Tools a user will be authorized on if they pass all the quizzes in this group (through ToolQuizGroups).
sools: Shops a user will be authorized on if they pass all the quizzes in this group (through ShopQuizGroups).

4.4.1.7 QuizResult
A quiz results records the record of a user taking a quiz and his score whether he passed or failed.

Fields
score: The score a user obtained on a quiz stored as a percentage.

Relationships
lab_user: The user that took the quiz.
quiz: The quiz taken.

4.4.2 Authorizations
Authorization models are join models that give a LabUser access to a specific resource.

4.4.2.1 ShopAuthorization
A ShopAuthorization object gives a user access to a particular shop at a given permission level.

Fields
permission: The level of access a user has to the facility ("visitor", "basic_user", "project_space_user", "lab_monitor", "authorizing_lab_monitor").
expires_at: The date the authorization should no longer be valid.

Relationships
lab_user: The user being granted access.
shop: The shop the user has access to.
events: Events that have occurred in this shop session.

4.4.2.2 ToolAuthorization
A ToolAuthorization object represents a user having access to a specific tool.

Fields
approved: If false, the user has completed the requirements to receive this authorization, but it still needs manual approval. If true, the user has access to this tool. The auto_approve field on QuizGroup sets the behavior for this field when the record is created in response to quiz being taken. Non-approved tool authorizations will show in the lab monitor controller with an orange warning next to them (See Section 3.3.4, Detail View*).*

Relationships
lab_user: The user being granted access.
tool: The tool the user has access to.

4.4.3 Auditing
The backend system stores information about how specific users interact with the shop. Models in this section enable that functionality.

4.4.3.1 ShopSession
A ShopSession object represents a single time period a user was in a facility. When a user checks in, a new shop session is created, and the start time is set to the time of check in. When a user checks out, the user's last shop session has it's end time set to the time of check out. In some cases such as when a lab monitor grants a user a temporary authorization (See Section 7.3.6 Grant Temporary Authorization), a shop session will be created with no start or end time with just the one event done to the user.

Fields
start_time: The time the shop session began
end_time: The time the shop session ended
created_at: The time the shop session record was created

Relationships
lab_user: The lab user the shop session is referring to
shop: The shop the session occurred in

4.4.3.2 Event
An event is an action that has occurred in a shop session such as checking in, checking out, becoming lab monitor, or attempting to do an action that was denied.

Fields
event_type: A distinct string that describes an event; usually written in snake case.
notes: A block of text that can provide additional context about the event.
occurred_at: The date and time the event occurred.

Relationships
shop_session: The shop session this event is a part of.

5 Web Architecture

The backend makes extensive use Amazon Web Services as the backbone of the system's infrastructure. We use a combination of AWS services to run the database, authentication, logging, container registry, and container servers. This section will detail how the infrastructure is architected.

5.1 Containers

The backend software uses Docker to enable it to be distributed as a containerized application. Instead of deploying the source code to a server with its own configuration, the backend gets built into an image known as a container. This image can then be started on almost any hardware without worrying about configuration differences.

5.1.1 Elastic Container Registry

Elastic Container Registry (ECR) is Amazon's managed solution for storing docker images. Whenever a new container is built, we push it to the container registry so it can be deployed. ECR stores the latest image and automatically deletes all old images every night to save space. There is no public access to our container registry.

5.1.2 Elastic Container Service

Elastic Container Service (ECS) is a solution by Amazon that takes an image and starts servers that run it without our needing to worry about the servers the image is running on. This enables horizontal scaling as ECS can automatically spin up more containers and will add them to our load balancer. If a container stops responding to health checks, the load balancer will stop sending traffic to it and ECS will start a new instance to replace the unhealthy one.

ECS has two infrastructure choices: EC2 and Fargate. The EC2 option runs your containers off EC2 instances you can directly access. Recently Amazon announced a new solution called Fargate where you pay for vCPU time directly rather than paying for the servers themselves. The backend runs on Fargate; however, it could be switched to EC2 fairly easily.

ECS has an interface accessible from the AWS console. This interface enables system administrators to monitor the current state of the cluster and create task definitions. Task definitions define how the servers should start, configurate, and run the containers. In the case of the backend, the task definition starts two containers: one that runs a web worker and one that runs a background job runner. It's configured to only run one of each unless a deploy is in progress in which case it will start up a new service and ensure it's healthy before shutting down the old one. This leads to zero down time deploys. The task definition also defines what security group the containers will be started in. We have configured the main task to start the application servers in a private Virtual Private Cloud (VPC) that's inaccessible from the public internet. This means clients can't directly connect to the containers. Instead, they send requests to the publicly accessible load balancer and it forwards the traffic to the private containers.

5.2 Authentication

We make use of AWS's Single Sign On (SSO) service for authentication to the AWS console, the admin panel, and other services used by the system. The SSO page in the AWS console allows an administrator to create new admin users and grant them access to different services. Visiting https://sms.mfelabs.org will redirect a user to the SSO login page. Here, they will be able to click on applications they want to go-to and automatically be logged in. Each user with access to the admin panel has a provisioned SSO login. Currently, it is a requirement that any user provision in the SSO console also have a LabUser record in the backend with the same email address.

5.3 Logging

All the application logs generated by the backend are centralized using AWS CloudWatch Logs. AWS CloudWatch Logs enables us to quickly search through logs to identify problems, create metrics based on logs, create alarms based on logs, and create dashboards that present the logging data in an easy to understand format. More information about CloudWatch Logs can be found in Section 6.2.2.

5.4 Database

The primary datastore used by the backend is a Postgres database managed by AWS Relational Database Service (RDS). The database is setup to automatically backup every night and to retain those backups for 7 days. The RDS server is in the same VPC as the application servers limiting the need for it to be publicly accessible. Its access logs are streamed to CloudWatch Logs (See Section 5.3 above).

6 Information Security

The lab management system built as part of this project stores sensitive information that necessitates having controls in place to prevent unauthorized access or dissemination. This section will detail the information handling procedures in place within the system and the infrastructure to prevent unauthorized disclosures.

6.1 Types of Information

Before discussing how information is handled, it is necessary to review what information is stored by the system. The primary data stored by the system is as follows:

- User information
  - Name
  - Email
  - Photo
  - ID number
  - RFID number
- Logs
  - Entry/exit events
  - Access denied events
  - API Logs
    - IP Addresses
    - Web request paths
- Shop/Tool Authorizations
  - Scores on each quiz
  - Users' access levels on specific tools

Each of these pieces of data is stored and retained in a manner consistent with its level of sensitivity and the operational needs of the system and the facility.

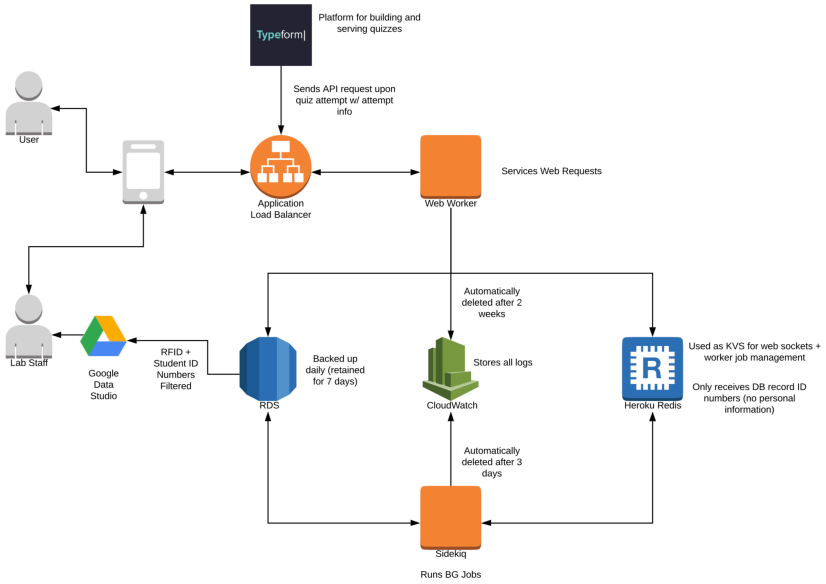

Figure 24. An information flow diagram visualizing where information is stored, how it is retained, and how it is distributed.

6.2 Information Storage Locations

The information listed in the prior section is stored in five locations: Amazon RDS, Amazon CloudWatch Logs, Redis, Typeform, and Google Data Studio. Each of these storage locations is used for a different purpose and has its own retention policy. The information is stored in each of these locations for as short of a time period as is reasonably possible while still achieving the operational goals of the system. Figure 24 shows a visual representation of the data flow within the system.

6.2.1 Amazon RDS

Amazon's Relation Database Service (RDS) is a service provided by Amazon Web Services that is used to manage relational databases. The system uses the Postgres edition of RDS. The RDS database is used as the primary datastore for the application holding all user information and authorizations. Access to the RDS instance is limited. The RDS instance is in a virtual private cloud (VPC) with public access restricted. It has an access control policy in place only allowing connections from application servers and Google Data Studio. Each day, the RDS instance is archived and stored to provide backups should any data loss occur. These backups are automatically deleted after 7 days. Access to these backups is monitored with AWS CloudTrail and any access to the data is recorded. Permission levels also exist within the Postgres database itself. The user account used by Google Data Studio has read-only access and is unable to query sensitive fields such as a user's RFID number.

6.2.2 Amazon CloudWatch Logs

Amazon CloudWatch Logs (CWL) is a log aggregation service provided by Amazon Web Services. CWL is used to provide a centralized location where all the application logs generated by the system are stored. This enables system monitoring and enables the identification of malicious attacks on the system. We have created dashboards in CloudWatch that show recent threats to system and clients that seem to be scanning for vulnerabilities. The logs stored by CWL may contain sensitive information. For this reason, we chose to automatically discard historical logs. Logs generated by the web servers are automatically deleted after 2 weeks so long-term attacks and trends can still be identified. Logs generated by the background works are deleted after 3 days as the current status is more important than historical data in our use-case. RDS access logs are never deleted.

6.2.3 Redis

Redis is used for the web socket connections and background jobs as well as for as a KVS for caching. Information is usually stored in Redis for fewer than 24 hours and the Redis database is constantly cleared. Efforts have been taken to try to limit the amount of sensitive information stored in Redis by storing database identifiers instead of data wherever possible.

6.2.4 Typeform

Typeform is the platform where quizzes are designed by shop staff and taken by lab users. Typeform sends quiz results to the backend whenever a user completes a quiz. They also retain a record of the form entry which can be discarded. Typeform doesn't have a public data retention policy; however, their privacy policy states that they will not disseminate the information collected in the form responses. Care is taken to provide Typeform with as little information as possible.

6.2.5 Google Drive

Google Data Studio (GDS) is a business intelligence tool used for analytics purposes. GDS retrieves information directly from RDS using a restricted access user account (See Section 6.2.1). Google caches this information for fast loads; however, the information is restricted to Google accounts with access to the data studio sources.

7 API Design

The purpose of the API is to enable authorized client applications to manipulate data stored by the backend is a secure manner.

All API requests will use the base URL https://sms.mfelabs.org/api/v1.

7.1 Design Pattern

The API's design pattern is a combination of a RESTful and RPC pattern. The RESTful pattern is used for all direct data retrieval, creation, and manipulation. The RESTful pattern was explicitly designed for this use case which reduced the API's design effort as simplified the implementation.

One scope of the API called actions is implemented with a pattern more similar to RPC. Actions are procedures implemented by the server that the client doesn't execute directly. Examples of these actions are checking in/out a user, placing a call, or closing the shop. RESTful APIs generally are not great for this type of remote procedure. One common way to implement this is by adding an action to the end of a RESTful route. An example of this would be using the following request to check in a user with an ID of 2 in a shop with an ID of 1:

POST /api/v1/shops/1/lab_users/1/check_in.json

The problem with this pattern is that it can create inconsistencies as there is no clear answer as to what the return type of this call should be. It could be a lab user object or some sort of check in object.

Another solution would be to add models to the API that don't exist in the data model. An example of this would be using the following API call to place a call:

POST /api/v1/devices/1/calls.json
Body: {
    "to": 2
}

This is a very reasonable design pattern; however, we decided to use the RPC style pattern instead due to its simplicity and flexibility in adding actions not associated with a specific data type in the future.

With the RPC pattern we used, a user can be checked in by making a request that looks like this:

POST /api/v1/actions/check_in.json
Authorization: Bearer {authentication_token}

The authentication token contains all the information about the user making the request. When a user authenticates, the API returns a list of all valid actions and a subset of invalid actions along with the reason they are invalid. This allows the device to show the user an error message if they attempt to complete an invalid action.

7.2 Authentication

All authentication in the application is done with JSON web tokens (JWT) as defined in RFC 7519. A JWT token is an access token in JSON format that is signed with a private key on the server. Washburn JWT tokens are in the following format:

```
{
        "scopes": [],
        "lab_user_id": null,
        "device_id": null,
        "shop_id": null,
        "expires_at": "{24 hours in future}"
}
```

Our JWT tokens can represent a shop, lab user, device, or any combination of these objects. A token with a device and shop can take actions on a specific shop as a given device. A token with a device, shop, and lab user can take actions such as checking a user into a specific shop. The only time a token doesn't have a device associated is when it was used to authenticate on the embeddable quiz widget (See Section 4.3.2, Quiz Group Embedded View).

JWT tokens are sent to the server in the OAuth2 Bearer token format. This is done by adding the following header to an HTTP request:

Authorization: **Bearer {auth_token_here}**

The scopes field defines what a token is allowed to be used for. The most common scopes are defined in section 7.2.1 below.

7.2.1 Scopes

standard_device
Description: Can perform all standard device actions
Usages:

- Can run close_shop, place_call, switch_lab_monitor, and grant_temp_access actions assuming other scopes and conditions are met (for example device with the role "check_in" cannot call grant_temp_access.

standard_lab_user
Description: Can perform all standard lab user actions
Usages:

- Can run check_in, check_out, and open_shop actions

quizzes
Description: Can request a list of quizzes for a specific user
Usages:

- Can call /api/v1/quizzes.json

7.2.2 Requesting a Token

There are three ways to request tokens depending on what type of token is needed and the required scopes. These methods will be described in this section.

7.2.2.1 Device Tokens

Device tokens always have the “standard_device” scope. The first time a device wants to connect to the Washburn API, it must send an API request to register itself with the system. The authentication token returned can be used on future requests. The refresh_token will be used to request new authentication tokens in the future. Refresh tokens never expire unless manually revoked by the admin. A client device should show the display_identifier returned on its display if possible (to identify the device in the admin panel). An administrator must approve a device before it can use the authentication token to perform actions and requests will respond with a 401 status-code until this occurs.

Devices should only be created the first time they need to connect to the API. In the future, the refresh token should be used.

To create a new device, make the following HTTP request:

Request
POST /api/v1/devices.json

Response


```
{
  "data": {
    "id": 1,
    "display_name": null,
    "display_identifier": "[REDACTED]",
    "approved": false,
    "role": null,
    "token": {
      "value": "[REDACTED]",
      "expires_at": "[REDACTED]"
    },
    "refresh_token": "[REDACTED]"
  }
}
```


Future authentication requests can be made with the following HTTP request:

Request
POST /api/v1/devices/authenticate.json

Response


```
{
  "data": {
    "id": 1,
    "display_name": "Washburn Check In",
    "display_identifier": "[REDACTED]",
    "approved": true,
    "role": "check_in",
    "token": {
      "value": "[REDACTED]",
      "expires_at": "[REDACTED]"
    },
    "shop": {
      "id": 1,
      "display_name": "Washburn"
    }
  }
}
```


7.2.2.2 Lab User Tokens

There are two different types of lab user tokens: lab user tokens with a shop and lab user tokens with no shop. The most common type of token has both a lab user and a shop; however, in the case of authenticating to take the online quizzes, the token will have no shop.

7.2.2.2.1 Request Lab User Token with Shop

This API call will request a JWT token for a lab user in a specific shop for actions such as checking in. This request must be made with an approved device token in the Authorization header.

Request
POST /api/v1/shops/{shop_id}/lab_users/authenticate.json

Parameters
shop_id: The ID of the shop to authenticate for
rfid_number: The RFID number of the user to authenticate
swipe_number: The WPI ID number of the user to authenticate

Only one of rfid_number and swipe_number is required.

Response


```
{
  "data": {
    "id": 3,
    "display_name": "Joe Doe",
    "token": {
      "value": "[REDACTED]",
      "expires_at": "[REDACTED]"
    },
    "actions": [
      {
        "action": "check_in",
        "error": null
      },
      {
        "action": "check_out",
        "error": "You are already checked out of the shop."
      }
    ],
    "role": "basic_user",
    "can_register": false
  }
}
```


The response provides some basic information about the user that was just authenticated. This information includes the user's display name, role in the shop specified (See Section 4.4.2.1, ShopAuthorization), whether the user is eligible to register (i.e. if they have an RFID number assigned to them), and a list of actions they can perform (See Section 7.3, Actions). Then token should be used for the next request you need to make as the user.

7.2.2.2.2 Request Lab User Token Without Shop

This type of token is only used when requesting a list of quizzes, the user has completed, or the user is eligible to complete in a specified quiz group. It will have a scope of "quizzes" and will not be able to make standard action requests.

Request
POST /api/v1/shops/{shop_id}/lab_users/authenticate.json

Parameters
email: The email address of the user to authenticate

Response


```
  "data": {
    "id": 3,
    "display_name": "Joe Doe",
    "token": {
      "value": "[REDACTED]",
      "expires_at": "[REDACTED]"
    },
    "role": "basic_user"
  }
}
```


7.3 Actions

Actions are routines that can be called by a client application on behalf of either a lab user or a device object. This section will detail the actions currently implemented and how the actions are architected.

All the actions in the Washburn API software are implemented in app/commands/actions/. Each action is defined as a class following the structure established by the SimpleCommand library (https://github.com/nebulab/simple_command). Each action has a call method which is where the code the action executes is written. Actions can also have an initializer which is how parameters are passed to the action.

All actions in the backend codebase inherit from ShopAction or DeviceAction.

ShopAction defines an initializer that takes a Lab User, Shop, and an optional Device:

```
1.  def initialize(lab_user, shop, device = nil)
2.    @lab_user = lab_user
3.    @shop = shop
4.    @device = device
5.  end
```

ShopActions are actions that act on a lab user in a specific shop. The device argument specifies which device sent the action and can be used for actions such as printing a badge to the correct printer (See 2.3 Badge Printing).

DeviceAction defines an initializer simply takes a device:

```
1.  def initialize(device)
2.    @device = device
3.  end
```

DeviceActions are actions that are taken by a device either with an anonymous or unauthenticated user. An example of a device action is placing a call because the device doesn't know who the lab user placing the call is (See Section 7.3.5).

Actions are invoked by making an HTTP POST request to /api/v1/actions/{action_name}.json. Some actions have additional parameters than need to be provided (defined in app/controllers/api/v1/actions_controller.rb).

7.3.1 Check-In

The check-in action is a shop action that checks the specified user into the specified shop if allowed or returns an error.

First, the action uses the ActionPolicy class to ensure the check_in policy is met. If the policy is not met, the backend creates a “check_in_denied” event on the lab user that failed to check in.

If the user is eligible for check in, a LabUserShop object is created to join the lab user and the shop (See Section 4.4.1.1, Shop).

Next, a “check_in” event is added to the user’s current shop session if one already exists. If not, a shop session is created and the start_time field is set to the current time.

Lastly, a number of background jobs are queued to do the following: send a shop update message over the web socket, send a push notification to all lab management devices, and print a badge for the user that just checked in if needed.

7.3.2 Check-Out

The check-out action is a shop action that removes a user from a shop if he is eligible for check-out.

First, the action uses the ActionPolicy class to ensure the check_out policy is met. Common cases where it might not be met include if the use did not check-in or if the user is currently the lab monitor. If the policy is not met, a “check_out_denied” event is created on the lab user’s current shop session.

If the user is eligible for check-out, the LabUserShop object created at check-in is destroyed.

A “check_out” event is added to the user’s current shop session and the end_time field is set to the current time.

If the shop is closed, ActionPolicy checks if the user is eligible for a late checkout which enables the user to check-out within 30 minutes of a shop closure (See Section 7.3.4, Close Shop).

Lastly, the shop update and push notification background jobs described in 7.3.1 above are queued.

7.3.3 Open Shop

The open shop action is a shop action that opens the shop and makes the caller the active lab monitor. This action must be called before the user can check in (unless they are able to check in for another reason such as they are a project space user).

First, the action uses the ActionPolicy class to ensure the open_shop policy is met. If it isn't, an "open_shop_denied" event is recorded.

If the user is eligible to open the shop, the action checks if the shop is already open. If it is, it returns an error with the message, "Cannot open shop if it's already open."

Next, the responsible_lab_user_id field of the Shop object is set to the id of the lab user calling the action and an "open_shop" event is recorded on the lab_user's shop session.

Lastly, a background job is queued to send a web socket message to inform other devices that the shop status has been changed.

7.3.4 Close Shop

The close shop action is a device action that removes all users, except the current lab monitor and project space users, from the shop associated with the device that called it and tentatively ends their shop sessions. If the user checks out within the next 30 minutes, the last shop session has a check-out event appended and the shop session end time extended.

First, the action finds the current lab monitor for the shop associated with the calling device. The ActionPolicy class is used to verify that user is able to perform the action "close_shop." If access is denied, a "close_shop_denied" event is added to the user attempting to perform the action.

If the policy is met, the action iterates over all users currently in the shop and adds a "check_out_by_close" event to their shop sessions if they are not the current lab monitor and they are not a project space user (See Section 4.4.2.1, ShopAuthorization). The end time of these shop sessions is also set to the current time.

The responsible_lab_user field on the shop is set to nil and lab users currently in the shop are removed (if they do not meet the conditions defined above).

Lastly, a shop update and push notification background job are queued.

7.3.5 Place Call

The place call action is a device action that is called when a user wants to initiate a video call with the lab monitor. It handles generating tokens and sending VOIP notifications (See 3.2.6 Calling for more details on video calls).

First, the action generates a 24-character long Base58 string to act as the room SID (See Section 3.2.6.1 Call Process). This room SID is sent back to the client initiating the call as well as in the notification to the lab monitor devices so they can join.

Next, the action finds a list of all the lab management devices (devices with the role “lab_management”) assigned to the initiating device’s shop. For each device in this list, we first generate a Twilio VideoGrant access token (https://www.twilio.com/docs/iam/access-tokens). This access token is tied to a specific user id (the device’s display name) and the room identifier generated earlier. Next, an Apple Push Notification Service (APNS) notification is generated with the custom data shown in Figure 25. This push notification is sent as a PushKit VOIP notification.

```
incoming_call:{
    from: "{display_name}",
    room: "{room_identifier}",
    access_token: "{twilio_access_token}"
}
```

Figure 25

Lastly, the room identifier and an access token for the initiating device is returned.

7.3.6 Grant Temporary Authorization

The grant temporary authorization action is a device action that creates a new shop authorization for a given user that expires 24 hours after its creation. This action is used to manually approve visitors that are in the system; however, have not completed the basic user training.

The initializer for this action takes both a device and a lab user to grant the temporary authorization to.

```
1.  def initialize(device, lab_user)
2.    @device = device
3.    @lab_user = lab_user
4.  end
```

When the action is invoked, it attempts to find a non-expired shop authorization for the specified user. If an existing authorization is found, a "User is already authorized" error is raised.

If no pre-existing authorization is found a new ShopAuthorization object is created and an event is added to the newly-authorized user with the type “granted_1_day_auth.” This enables tracking of how often users are given a temporary authorization.

7.3.7 Switch Lab Monitor

The switch lab monitor action is a device action that changes the active lab monitor.

The action has an initializer that takes two arguments, the device and the new lab monitor. The current lab monitor is able to be derived from the device.

```
1.  def initialize(device, new_lab_monitor)
2.    @device = device
3.    @new_lab_monitor = new_lab_monitor
4.  end
```

First, the action uses the ShopPolicy class to ensure the new lab monitor is able to pass the "become_responsible_user" policy. If this fails, a "switch_lab_monitor_denied" event is recorded on the new lab monitor.

If the policy check is successful, the responsible_lab_user field on the initiating device's shop is set to the new lab monitor and a "become_lab_monitor" event is added for the new lab monitor.

Lastly, shop update and send notification background jobs are queued.

7.4 Shops Scope

This section will detail all the API requests available for retrieving shop information. The base URL for this scope is /api/v1/shops. API requests in this scope are read only as there are no defined API calls for modifying or creating a shop. These tasks are done using the admin panel.

7.4.1 List All Shops

The API provides a mechanism to list all shops registered in the backend.

Authentication – Device Token: ["standard_device"]

Request
GET /api/v1/shops.json

Response

```
{
  "data": [{
      "id": 1,
      "display_name": "Washburn"
    }]
}
```

7.4.2 Get Shop

The API provides a mechanism to get information about a particular shop. It is used by the lab monitor controller device to fetch the current status of the shop (including the current users and current lab monitor).

Authentication – Device Token: [“standard_device”]

Request
GET /api/v1/shops/{shop_id}.json

Parameters
shop_id: The ID number of the shop to request information on. This ID can be obtained from list all shops request or from another API request such as a device authentication request.

Response


```
{
  "display_name": "Washburn",
  "responsible_lab_user": {
    "id": 1,
    "display_name": "Sam Demo",
    "profile_photo": "[REDACTED]",
    "role": "authorizing_lab_monitor"
  },
  "lab_users": [{
      "id": 1,
      "display_name": "Sam Demo",
      "profile_photo": "[REDACTED]",
      "role": "authorizing_lab_monitor"
    }]
}
```


7.4.3 Get Shop Signage Information

The get shop signage endpoint is used by the digital signage web application to get the current status of the shop (See Section 2.4, Digital Signage). This endpoint is similar to the get shop endpoint except it is unauthenticated and reveals as little information as is reasonably possible.

Authentication – None

Request
GET /api/v1/shops/{shop_id}/signage.json

Parameters
shop_id: The ID number of the shop to request information on. This ID can be obtained from list all shops request or from another API request such as a device authentication request.

Response


```
{
  "data": {
    "display_name": "Washburn Shops",
    "open_message": "{message to show when shop is open}",
    "closed_message": "{message to show when shop is closed}",
    "responsible_lab_user": {
      "id": 1,
      "display_name": "Sam Demo",
      "profile_photo": "[REDACTED]",
      "role": "authorizing_lab_monitor"
    },
    "lab_users_count": 17
  }
}
```


7.5 Shop Lab Users Scope

The shop lab users scope contains all the API endpoints related to fetching and manipulating lab users in the context of a specific shop. All endpoints in this scope start with /api/v1/shops/{shop_id}/lab_users.

7.5.1 Authenticate Lab User

This API endpoint is used to get a user's personal information and an authentication token. API reference information can be found in section 7.2.2.2.1 Request Lab User Token with Shop.

7.5.2 Get Lab User

This API endpoint can be used to get information on a specific lab user by ID. The endpoint uses the LabUserDetailSerializer to render the specified lab user. The information provided includes the user's basic information, the tools they are authorized on, and if they are authorized to be in the shop (has_authorization).

Authentication – Device Token: ["standard_device"]

Request
GET /api/v1/shops/{shop_id}/lab_users/{lab_user_id}.json

Parameters
lab_user_id: The ID number of the lab user to request information about.

Response


```
{
  "data": {
    "id": 1,
    "display_name": "Sam Demo",
    "profile_photo": "[REDACTED]",
    "has_authorization": true,
    "role": "authorizing_lab_monitor",
    "tool_authorizations": [{
      "id": 1,
      "approved": true,
      "tool": {
        "id": 1,
        "name": "Vertical Bandsaw"
      }
    }]
  }
}
```


7.5.3 Search for Lab User

This API endpoint is used by the lab monitor controller device to find a lab user by their display name or their WPI ID number. It accepts two parameters which are both used as filters. The display name parameter will find lab users with a display name that contains the query. The WPI ID number parameter (swipe_number) will find lab users with an exact match to the ID number. The first 50 matches will be returned.

Authentication – Device Token: [“standard_device”]

Request
GET /api/v1/shops/{shop_id}/lab_users/search.json

Parameters
display_name: The search query for the user’s display name (does not need to be complete)
swipe_number: The query for the user’s WPI ID number (needs to be exact match)

Response


```
{
  "data": [{
    "id": 1,
    "display_name": "Sam Demo",
    "profile_photo": "[REDACTED]",
    "role": "authorizing_lab_monitor"
  }]
}
```

7.5.4 Get Shop Sessions

This API endpoint is used by the lab monitor controller to get a given user's last 10 shop sessions. This endpoint will also return the full event log from these sessions.

Authentication – Device Token: ["standard_device"]

Request
GET /api/v1/shops/{shop_id}/lab_users/{lab_user_id}/shop_sessions.json

Parameters
lab_user_id: The ID number of the lab.

Response


```
[{
  "id": 1,
  "events": [
    {
      "event_type": "open_shop",
      "occurred_at": "2018-12-13T19:54:47.022Z",
      "notes": null
    },
    {
      "event_type": "check_in",
      "occurred_at": "2018-12-13T19:54:47.144Z",
      "notes": null
    },
    {
      "event_type": "become_lab_monitor",
      "occurred_at": "2018-12-13T20:05:10.090Z",
      "notes": null
    }
  ],
  "start_time": "2018-12-13T19:54:46.995Z",
  "end_time": null
}]
```


7.6 Devices Scope

The devices scope contains all the API endpoints related to creating, updating, or authenticating devices. A device in the context of this application is a WPI owned device that has API access to the backend (See Section 4.4.1.4, Device). Each kiosk and lab management unit has a corresponding device record on the backend.

7.6.1 Create Device

The create device endpoint creates a new device on the backend and is called the first time a client is started on a new device. The device created will default to being unapproved and

having no permissions. An administrative user can manually approve and configure a device after calling this endpoint.

More information on this endpoint is available in section 7.2.2.1 Device Tokens.

7.6.2 Authenticate

The authenticate API endpoint is used to obtain an authentication token using a previously stored refresh token. More information on this endpoint can be found in section 7.2.2.1 Device Tokens.

7.6.3 Update Device

The update device endpoint is used to update specific parameters on the device model. The only parameters that are able to be updated are “voip_apns_token” and “alert_apns_token”. “voip_apns_token” is the token generated by PushKit on a user’s iOS device for sending high-priority VOIP notifications. “alert_apns_token” is the token for sending standard alert notifications. Both of these tokens should be sent each time the app launches or if the client application is aware, they have changed.

Authentication – Device Token: [“standard_device”]

Request
PUT /api/v1/devices/{device_id}.json

Parameters
device_id: The device ID of the device to be updated (this must be the device associated with the authentication token used to make the request.
voip_apns_token: The new VOIP APNS token to set on the device model.
alert_apns_token: The new alert APNS token to set on the device model.

Response


```
{
  "data": {
    "id": 1,
    "display_name": "Washburn Entry",
    "display_identifier": "[REDACTED]",
    "approved": true,
    "role": "lab_management",
    "reader_name": "",
    "shop": {
      "id": 1,
      "display_name": "Washburn Shops"
    }
  }
}
```


7.7 Quizzes Scope

The quizzes scope is used for non-internal requests related to obtaining a user's current quiz progress and quizzes they are eligible to take. These API endpoints can be accessed using a lab user token with the scope "quizzes." These tokens can be obtained with a standard shop + lab user token or with a limited lab user token which can be obtained with just an email address. For more information on limited lab user authentication, see section 7.2.2.2.2 Request Lab User Token Without Shop.

7.7.1 List All Quiz Groups

This API endpoint returns an array of all the quiz groups visible to a given user. If a quiz has a URL of null, it means the user is ineligible to take that quiz; however, the user does have permission to see the record of it. This commonly comes up when a user has already taken a quiz that can't be retaken or if the user is not authenticated and needs to take an anonymous quiz (such as the basic information quiz which will register the user).

Authentication – Lab User Token: ["quizzes"]

Request
GET /api/v1/quizzes.json

Response


```
{
  "data": [{
    "id": 1,
    "display_name": "Basic User Quiz",
    "quizzes": [
      {
        "id": 1,
        "display_name": "0.0 Basic Information",
        "url": "https://mfelabs.typeform.com/to/XwIgMO"
      },
      {
        "id": 2,
        "display_name": "1.0 Operating Principles",
        "url": null
      }
    ]
  }]
}
```


7.7.2 Show Quiz Group

The show quiz group API endpoint is very similar to the list all quiz groups endpoint except it only returns a single quiz group. This reduces the server-load and request response time. This endpoint should be used whenever possible to reduce server load.

Authentication – Lab User Token: [“quizzes”]

Request
GET /api/v1/quizzes.json

Response


```
{
  "data": {
    "id": 1,
    "display_name": "Basic User Quiz",
    "quizzes": [
      {
        "id": 1,
        "display_name": "0.0 Basic Information",
        "url": "https://mfelabs.typeform.com/to/XwIgMO"
      },
      {
        "id": 2,
        "display_name": "1.0 Operating Principles",
        "url": null
      }
    ]
  }
}
```

7.8 Webhooks Scope

The webhooks scope is where all the API endpoints intended to be used as webhooks by other applications are located. The intended structure of a webhooks URL is /api/v1/webhooks/{service_name}/{action_name}. Currently, the only webhook implement is for Typeform, the quiz solution used with the system.

7.8.1 Typeform Webhook

The Typeform webhook is called whenever a user completes a quiz on Typeform. When the user presses submit, a HTTP POST request is sent to this endpoint. Typeform generates a signature (HMAC-SHA256) with a secret shared between the backend and Typeform's server which is validated to ensure the request originated from Typeform. The body of the request is used to find the associated quiz in the backend's database and create a quiz result record to track quiz attempts and completions. After every request to this endpoint, an action is called that recalculates the authorizations of the user that submitted the quiz. This ensures that once a user completes all the quizzes in a quiz group, he is granted the associated shop or tool authorizations.

Request
POST /api/v1/webhooks/typeform/submitted

Parameters
Defined in Typeform documentation (https://www.typeform.com/help/webhooks/)

Response

```
{
  "success": true
}
```

8 Future Work

One of the underlying technical goals of this project was to build the system to be extremely extendible to allow both future modifications to existing features as well as the addition of new functionality. Throughout the project, a number of future expansions have been identified.

One of the expansions identified was the connection of machine tools in the facility to this system. While the system is designed to track tool authorizations, there is a desire for these permissions to be enforced at the tools themselves. This system would be similar to the front entry system wherein a user taps his WPI badge on a machine whenever he wants to use it. The machine would then either use a visual indicator, such as a stack light, or simply not turn on if a user did not have access to use the tool. In addition to the obvious safety benefits, this would provide useful analytics to the shop staff regarding tool usage. This could be relevant for making decisions regarding capital acquisitions or for ensuring usage-based maintenance is completed.

Another potential expansion would be the integration of tool checkouts into this system. In the Washburn shops, users often borrow cutting tools to be used on the machines in the facilities. Currently, these tool check-outs are tracked on a clipboard kept in the tooling area. This is not ideal for keeping track of real-time inventory levels or ensuring tools are returned. The system could be expanded such that it tracks tool check-outs or even allows for self-checkouts wherein a user could check-out tools on their own without needing to find a lab monitor.

Finally, a number of smaller expansions could be completed. One of these smaller expansions is the replacement of the iPad Mini used as the lab monitor controller with a smaller device. Since the system was deployed, we have observed that many lab monitors don't carry the lab monitor device since it is too large. It would be useful to deploy the lab monitor controller app to a smaller device such as an iPhone or an iPod Touch to pilot changing the device type. Another smaller expansion would be the conversion from Wiegand to OSDP on the card reader interface device (See Section 2.1.3 Reader Communication).

There is both interest and funding for the expansion of this system and I am hopeful that the existing work can be expanded on to better fulfill the needs of the manufacturing facilities at WPI.