Staunton Makerspace Communication and Classes Management Systems

A technical report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

Evan Typanski
May 2020

By
Damon Cestaro, Hunter Williams,
Kane Lee, Pranay Dubey,
Evan Typanski, Michael Laterza,
Michael Wood, Samuel Ting

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Signed: ______________________________ Date____________________
Evan Typanski

Approved: Ahmed Ibrahim ______________________________ Date 4/27/2020
Ahmed Ibrahim, Department of Computer Science

Table of Contents

Abstract **1**

List of Figures **2**

1. Introduction **3**

1.1 Problem Statement 3

1.2 Contributions 6

2. Related Work **8**

3. System Design **11**

3.1 System Requirements 11

3.2 Wireframes 13

3.3 Sample Code 19

3.3.1 Models 19

3.3.2 Views 21

3.3.3 Frontend 23

3.4 Sample Tests 24

3.5 Code Coverage 27

3.6 Installation Instructions 30

3.6.1 AWS Setup 31

3.6.2 Create the EC2 Instance 31

3.6.3 Configuring ports 31

3.6.4 Server Setup 32

4. Results **34**

5. Conclusions **36**

6. Future Work **38**

7. References **40**

Abstract

The Staunton Makerspace project solved two problems for the Staunton Makerspace management, the creation of a class management system and improvement of their communication systems, in order to facilitate makerspace communications and begin to run classes for the space. We did this at the request of the Staunton Makerspace in order to allow them to organize, host, and run member classes, as well as public classes to allow for them to have better community outreach in the case of the classes team. The client also provided a vision of a new application designed to increase interaction with makerspace messaging, since existing members of the space were less willing to check the existing methods, creating a physical message board in the space allowed them to check their message without going out of their way. This project was taken on by the communication team, consisting of Evan Typanski, Michael Laterza, Michael Wood, and Samuel Ting. In order to do this, we developed a set of web applications for the makerspace in order to perform these applications. These applications allowed the makerspace management and members to better interact with each other and their members, as well as formalizes the process for creating classes for both members and the public. This application is important to the client as it allows for further public outreach and appeal, allowing them to create classes to help attract new members, as well as enhance internal communication.

List of Figures

Figure 1 - *Communications default kiosk display (no ID is scanned)* p.13

Figure 2 - *Communications user kiosk display (after ID is scanned)* p.13

Figure 3 - *Communications outbox (inbox is similar, but with no send button)* p.14

Figure 4 - *Communications send message page* p.14

Figure 5 - *Classes list view* p.15

Figure 6 - *Classes Details View* p.15

Figure 7 - *Classes Create Class View* p.16

Figure 8 - *Classes Teacher View* p.16

Figure 9 - *Classes User List View* p.17

Figure 10 - *Classes User View* p.17

Figure 11 - *Classes Admin View* p.18

1. Introduction

The primary goal of this project is convenience and utility. Neither of the two halves of this project are solving a key or dire issue, but rather one that looks far simpler on the surface - how to easily communicate information to an average user who, quite frankly, probably has something else he wants to be doing right now.

While this may look far less interesting then, say, building a tool to revolutionize online learning or making a program to enable a revolutionary new system to function, this problem is far more common than either of those two. In fact, this problem is even a core part of both of those two - no project, no matter how revolutionary it's capabilities, will get very far if it's users can barely figure out how to use it.

1.1 Problem Statement

This product is being designed for a non-profit organization called Staunton Makerspace. As the name suggests, this organization is a makerspace based in Staunton, VA. The idea behind a makerspace is that people working on small personal or home projects will often find themselves needing access to complex and expensive industrial tools - like large saw cutters, 3D printers, plasma cutters, or other tools along those lines. But each of these people only need to use these tools once or twice, and can not provide the funding or space to buy their own just for their current small project. To solve that problem, a makerspace will step in. A makerspace will acquire these complex tools, and then allow people to join the makerspace to use the tools (What

is a Makerspace, 2020). Each individual person will only pay a small amount and only use the tools a few times, but in totality the tools will be in almost constant use and the makerspace will earn enough money to justify acquiring these expensive tools. Our customer, Staunton Makerspace, is a makerspace along these traditional lines (Staunton Makerspace, 2018).

Staunton Makerspace faced two separate problems. First, effectively communicating information to their members. Secondly, coordinating and scheduling classes, training, and other events across the makerspace community. Both of these issues will be discussed in further detail in the following paragraphs.

The first problem is effectively communicating with their members. Members of the Staunton Makerspace can be broadly separated into two groups - highly active members and lightly active members. Highly active members are members who routinely come into the makerspace and keep up to date with the makerspace on some of the multiple communication platforms (email, Slack, Google Calendar, text messages) that the makerspace uses to communicate. By contrast, lightly active members tend to only come in a few times a month or perhaps year, and will often not even be present on the various communication platforms, much less will they tend to keep up to date on them and respond to messages quickly. Communicating with the former group is simple for the makerspace, but communicating with the latter group is far more difficult as many of its members may not respond to any messages and need to be talked to, in person, during one of the rare times they come to the makerspace. The system we are designing for the makerspace aims to solve this problem by sending messages directly to these users during one of the rare instances they come into the makerspace. Whenever a member swipes their customer key card to enter the makerspace, we will detect that swipe and display on

a large screen near the entrance any messages the makerspace or its members have sent to this individual which they need to see or act on.

The second problem is scheduling classes, training, and other events. As explained earlier, Staunton Makerspace like all makerspaces contains a multitude of complex, expensive, and extremely specialized pieces of industrial equipment. If improperly used, people not only run the risk of damaging the machines but injuring or killing themselves. Therefore, Staunton Makerspace requires that their members undergo training in a class before they can use certain pieces of equipment, which brings us to the main problem here - scheduling said classes. Currently Staunton Makerspace uses a simple Google Calendar app to schedule their classes. This works fine for telling people when and where classes are, but has several limitations due to the fact that google calendar is a calendar system, not a scheduling system. Some of these problems are: keeping track of who is signed up for a class, keeping track of who has paid the dues for the class or if extra dues are necessary to attend the class, keeping track of who attended and passed the training to hand out permission for use of more advanced machines, and so on. To fix these problems, Staunton Makerspace asked us to create a purpose made class scheduling system for them, with the ability to handle all of these special cases they need out of a scheduling system built into it.

Once delivered, the benefits of these two systems should be clear. Our messaging platform will allow Staunton Makerspace to easily and reliably make contact with their members to bring their attention to new developments, or to ask them to follow up with the leadership on certain issues, such as their membership needing renewed. Our class scheduling system will also make it far easier to schedule and create classes, as well as handling the various issues around it.

Currently, enterprising members who decide to hold a training event may end up spending more time coordinating and following up on the event then actually teaching it, but our service can simplify the vast majority of these tasks to reduce the time necessary to handle an event. This not only makes it easier for the members who hold trainings, but increases the number of trainings those members can hold for the same time commitments and may increase the willingness of other members to get more involved, as the system is now easier to understand and become involved in. While in truth neither of these systems add an entirely new capability to Staunton Makerspace, by massively simplifying the amount of effort that needs to be put into these tasks they will defacto open up new options for Staunton Makerspace, as tasks that were previously too difficult to seriously contemplate doing may become possible and become the makerspaces next goals.

1.2 Contributions

To address the first problem mentioned in the previous section, we were able to create a web based application which allows the members of the Staunton Makerspace to easily communicate with the members. The messenger system is able to keep track of each user and allow them to message each other directly on the site for the highly active members to communicate easily, as well as display messages that users receive on the screen upon entrance for the less active members to gain easy access to important news. The members are kept track of their user ids using key cards, which they swipe in to enter Makerspace and allow the large screen near the entrance to display individualized messages.

To address the second problem mentioned in the previous section, we were able to create a web based application which allows the members of the Staunton Makerspace to create and manage scheduling of events. In order to help with organizing the users by which equipment they are authorized to use, the application implements a certification system that can automatically keep track of authorization for each user and allow or prevent users from signing up for certain classes. Each of these classes also keeps track of signed up users, dues, and classes time to be displayed on the site. The Staunton Makerspace staff can use admin features to assign certifications to users once they have passed the classes, manage which certifications are for authorization of which machines, and approval for user-created classes.

The rest of this thesis goes into detail about the system and is organized into four sections. In Section 2, related works are presented. In Section 3, the details about the system requirements and designs that the team used to approach the aforementioned problems. In Section 4, the result of our work is discussed. Finally in Section 5, the thesis is concluded, and in Section 6, future work that can be done to expand the project is discussed.

2. Related Work

In order to consider work related to our system, we must first consider the most basic aims of the system mentioned above: to help the Staunton Makerspace effectively communicate both organization-wide and user-specific information with its members, and to help it schedule classes while keeping track of who's signed up and who's paid for said classes. First, the communication aim. Many other systems exist today that are capable of performing the task of delivering notifications and information to members of an organization (to some extent), but this analysis will focus on two: the *Slack* system and the *AlertMedia* system. Second, the classes/events aim. Similarly, there are a number of systems that accomplish the desired goal of this system, but with a few shortcomings. This analysis will focus on the *Google Calendar* system (currently used by the Staunton Makerspace) and why it does not meet their needs.

Slack is a "team communication software tool [that] enables searchable, annotated file sharing, [and] organizes messaging in channels and direct messages, either public or in private" (Dennerlein et al., 2016). It allows users to message and relay information to whoever needs to receive it in a very efficient manner. Slack is used in a very general manner, serving as the messaging service for all sorts of clubs, fraternities/sororities, and professional organizations. However, despite its efficiency and relatively streamlined design, Slack provides a steep learning curve for those (particularly older or less technologically-inclined folk) who are not accustomed to web-based messaging services. In the case of the Staunton Makerspace, a large proportion of users fall into this category. If a user in this group is not willing or able to learn how Slack works, then he or she will undoubtedly miss out on necessary communication from other

members of the Makerspace. With its RFID functionality, our system will show users their notifications upon entry to the Makerspace, effectively eliminating this steep learning curve where users need to learn all about the system to use it.

AlertMedia is an emergency notification service that companies and organizations can take advantage of in order to send out information to all of their employees (AlertMedia 2020). AlertMedia provides the advantage of being able to broadcast information across a wide variety of media, including text, email, and social media. This means that there is very little effort put forth by both the senders and receivers in order for the information to be passed. This satisfies one of our overarching aims, to reduce the number of actions receivers must take in order to receive pertinent information. However, the main aspect of AlertMedia that causes it to be unsuitable for the purposes of the Makerspace is that it is primarily a notification system, rather than a communication system. At this point, it will be useful to define the two types of AlertMedia users: admins, who have the power to send messages to everyone in (or a subgroup within) an organization, and normal users, who are able to receive these messages based on what subgroups they belong to. With AlertMedia, normal users can also reply to admins who have sent out a mass message to everybody in (or a subgroup within) a company/organization, but the same normal user cannot utilize the system to send messages to another user. In the case of the Staunton Makerspace, our system functions as much less of a one-sided system, meaning there is a greater implementation focus on the abilities and features relating to normal user communication and activity.

Google Calendar functions the way a normal calendar functions. It is good for marking special events on particular days and at particular times, and thus keeping you up to date on

events (Krüger, 2017). The platform allows users to share one other's calendars, but the functionality and experience are tailored around the individual, not necessarily organizations. Staunton Makerspace has a publicly viewable Google Calendar where they display events and the times they will take place. The issue that arises is that there is no streamlined process for members to sign up for events, for verification that they can attend, or for storing any of this information. Using the Google Calendar system is not conducive to smooth transfer of information within the makerspace and is ultimately one of several limiting factors in the previous system.

3. System Design

This system has users, guilds, classes, certifications, and communication. Users can attend classes and receive certifications. Guilds are groups of users. Certifications relate to what workshop machines the user is trained to use and trained to teach others how to use. Communication allows users to receive messages from other users or from guilds.

This system is developed using Python 3 and the Django framework. This language and framework were chosen because of the development team's previous experience with both. Django was specifically chosen because it abstracts away a lot of web application management (database handling, a templating engine, dynamic rendering, etc) that other frameworks leave to the developer to implement. This code is developed under the GNU General Public License v3.0. This license establishes that our software is free to use for any purpose, can be changed to fit the users needs, and shared as necessary.

3.1 System Requirements

In order to adequately meet the client's desires and make a product that is designed to work as they intend, it is massively important to acquire a comprehensive list of system requirements. Some of these are fundamental to the system and are labelled as minimum requirements, others are important features that build on these (desired requirements), and a few more are completely optional requirements. They are listed below:

Minimum Requirements:

- As a user, I should receive pertinent information from a kiosk upon entry
- As a user, I should be able to create a class
- As a user, I should be able to add a description to a class
- As a user, I should be able to send messages to a guild
- As a user, I should be able to send a message to a specific user
- As a user, I should be able to send a message to all users
- As a user, I should be able to send messages through a web app, without needing to interact with the kiosk
- As a user, I should be able to be a member of a guild

Desired Requirements:

- As a user, I should be able to create an account and log in/log out
- As an admin, I should be able to approve or deny the creation of a class
- As a class creator, I should be able to set a maximum capacity for a class
- As a class creator, I should be able to change the class date/time/location
- As a user, I should be able to send messages through slack and have them displayed on the kiosk
- As a user, I should be able to filter my inbox and outbox for easier viewing

Optional Requirements:

- As an admin, I should be able to promote users to guild trainers
- As a user, I should be able to track what level of certifications i have

- As a user, I should be able to request a class be scheduled to train me in equipment with my guild
- As a guild trainer, i should be able to update the certification level of people who attended my training class
- As a class creator, I should be able to automatically promote my class on the Makerspace Instagram and Facebook
- As a user, I should be able to pay the dues for my class through the class sign-up system
- As a class creator, I should be able to see which users paid their dues for the class
- As a class creator, I should be able to send the class information to select group of users
- As a class creator, I should be able to recommend a class to people with a certain training level
- As a user, I should be able to set notification preferences
- As a user, I should be able to view relevant notifications via a ticker on the default kiosk screen

3.2 Wireframes

In order to visualize how a website will be designed, our team found it important to model wireframes of our various web pages.

Figure 1 - *Communications default kiosk display (no ID is scanned)*

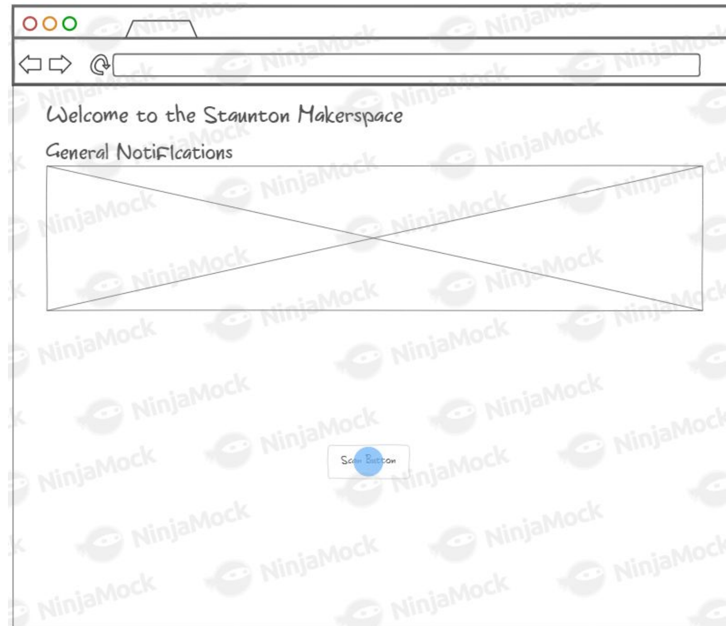


Figure 2 - *Communications user kiosk display (after ID is scanned)*

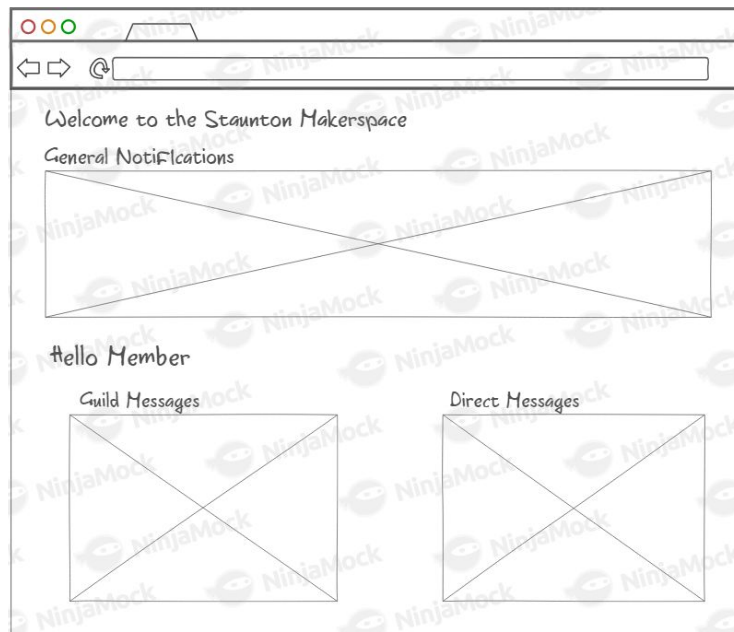


Figure 3 - *Communications outbox (inbox is similar, but with no send button)*

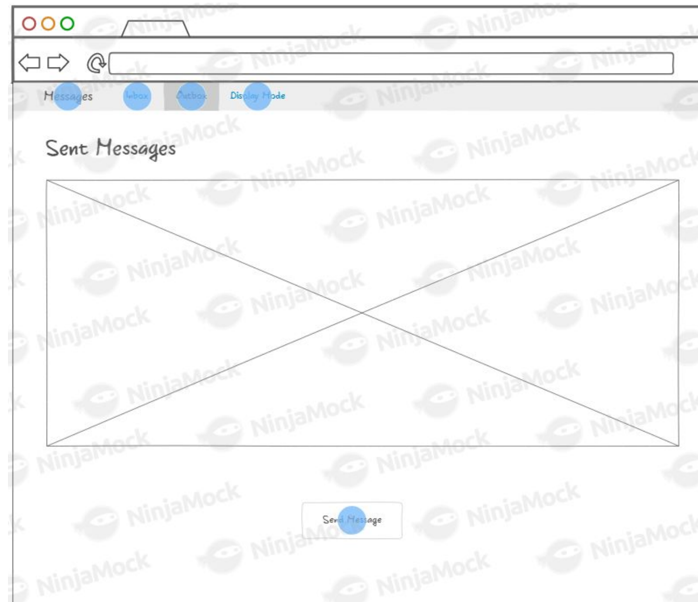


Figure 4 - *Communications send message page*

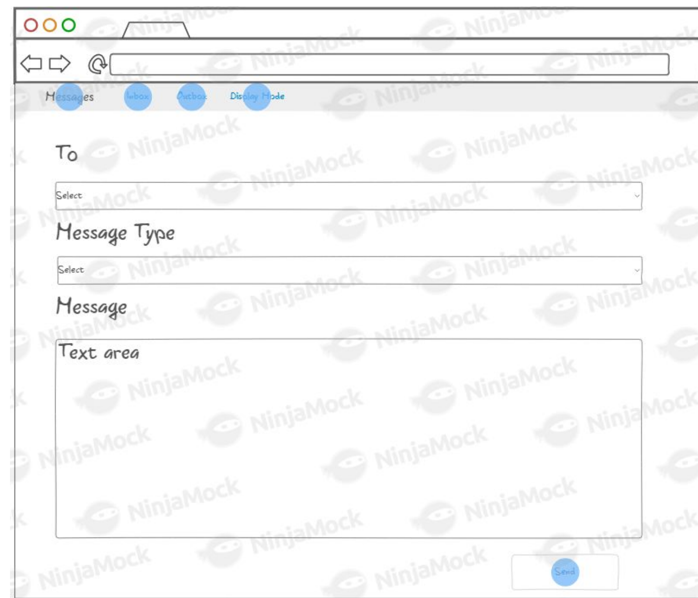


Figure 5 - *Classes list view*

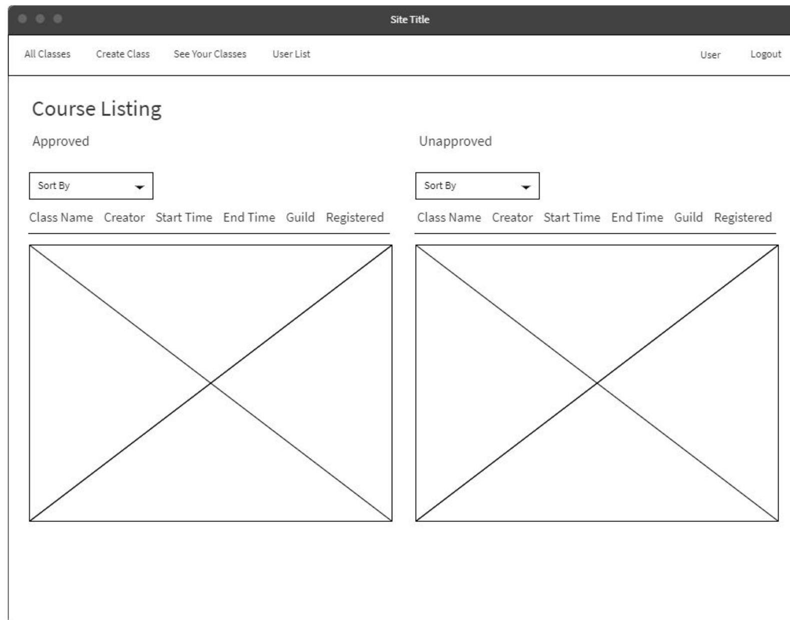


Figure 6 - *Classes Details View*

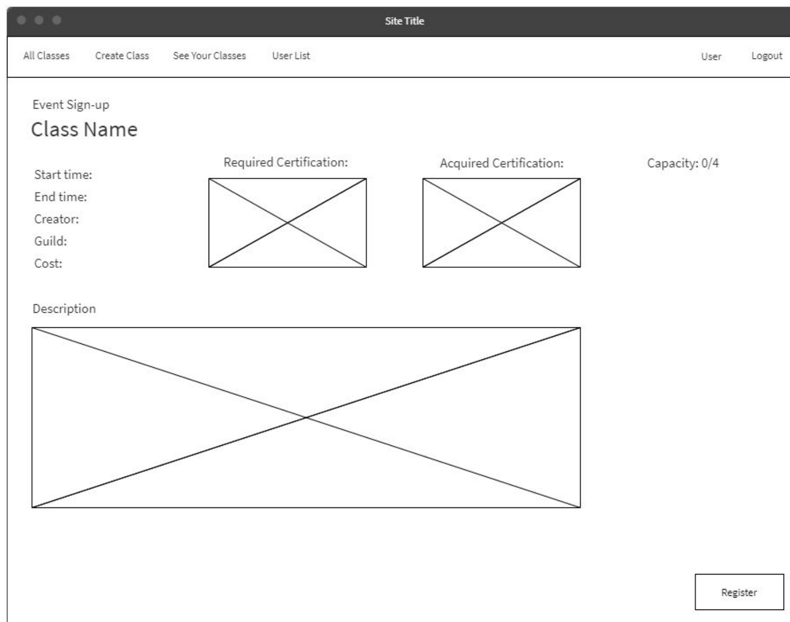


Figure 7 - *Classes Create Class View*

Figure 8 - *Classes Teacher View*

Figure 9 - *Classes User List View*

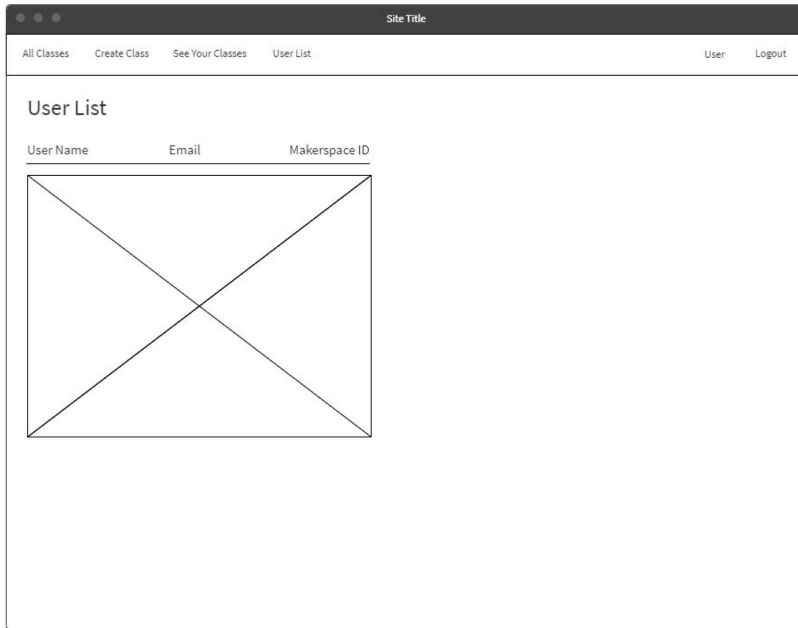


Figure 10 - *Classes User View*

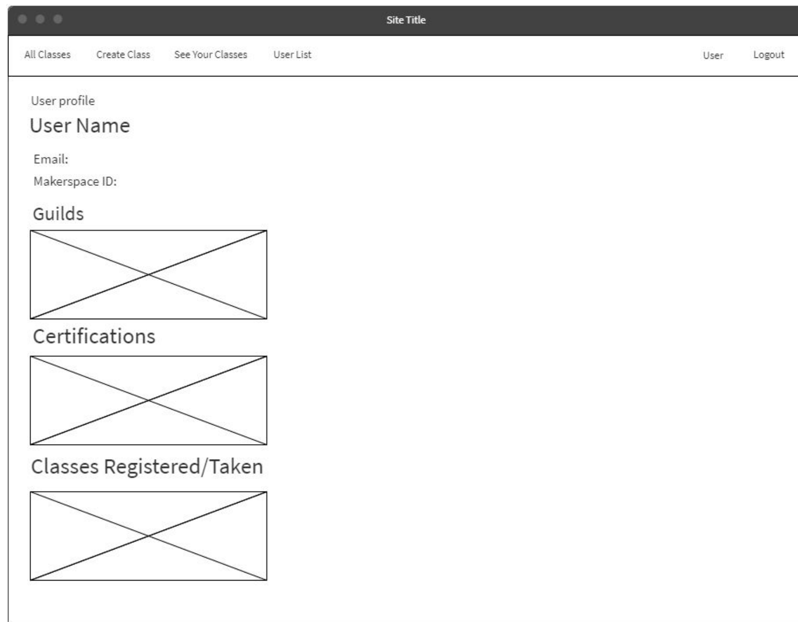
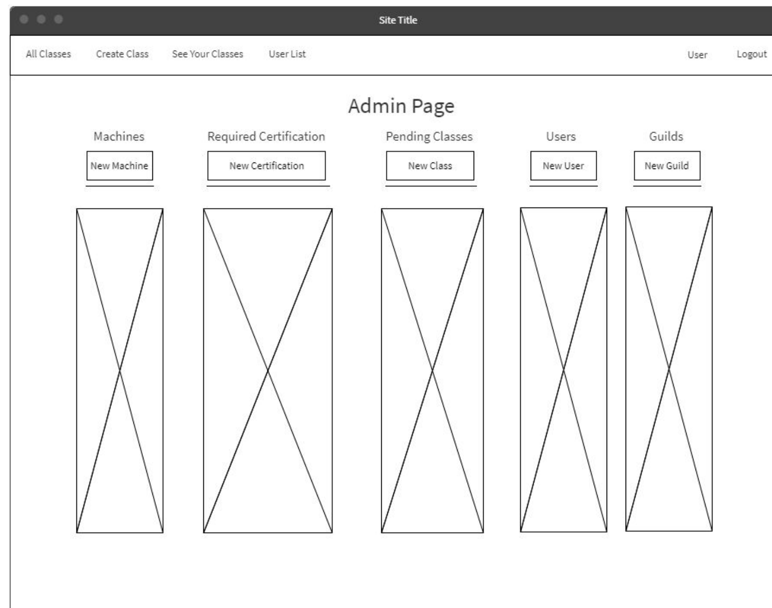


Figure 11 - *Classes Admin View*



3.3 Sample Code

3.3.1 Models

The User model represents any user of the system. Each has identifying information, including their name, email, and makerspace ID. Any user can scan in and retrieve their messages based on the makerspace ID. The password is kept as a hashed password, which is ensured by the API. Any users that are requested from the admin interface in the backend are represented by their first and last name via the __str__(self) method.

```
class User(models.Model):
      first_name = models.CharField(max_length=50)
      last_name = models.CharField(max_length=50)
      # The RFID for the user for accessing the Makerspace
      # TODO: Get actual max number of chars on id
      makerspace_id = models.CharField(max_length=20, unique=True)
      # Machines that this user is authorized to use
      authorized_machines = fields.ArrayField(
            models.CharField(max_length=100),
            null=True,
            blank=True,
      )
      # Machines that this user is authorized to teach
      teaching_machines = fields.ArrayField(
            models.CharField(max_length=100),
            null=True,
            blank=True,
      )
      # Email address associated with user - used as contact info
      email = models.EmailField(null=True, blank=True, unique=True)
      # Hashed password
      password = models.CharField(max_length=250, blank=True, null=True)

      def __str__(self):
            return "{} {}".format(self.first_name, self.last_name)
```

The Guild model contains associations to the User model. This allows users to group themselves into a guild for woodworking, laser cutting, and more.

```
class Guild(models.Model):
      # Name of the guild
      name = models.CharField(max_length=100)
      # Members in the guild
      members = models.ManyToManyField(User)
      def __str__(self):
            return self.name
```

The Authenticator is how the backend stores cookies for a given user session. Each is associated with a given user and has a unique value to differentiate user sessions.

```
# The authenticator used for a given user's session, tracked through a
cookie
class Authenticator(models.Model):
      # User must be provided
      user = models.ForeignKey(User, on_delete=models.CASCADE)
      # Authenticator for tracking session
      authenticator = models.CharField(max_length=250, primary_key=True)
      # Date the authenticator was created
      date_created = models.DateTimeField(auto_now=True)

      # If the authenticator is valid or not
      @property
      def valid(self):
            now = timezone.now()
            return now - AUTHENTICATOR_TIMEDELTA < self.date_created

      def __str__(self):
            return self.authenticator
```

3.3.2 Views

Backend views are mostly view sets, which is managed by Django Rest Framework. This sets up the endpoint logic for a User.

```
class UserViewSet(viewsets.ModelViewSet):
      """
      A ViewSet for combining logic associated with User objects
      """
      queryset = User.objects.all()
      serializer_class = UserSerializer
      filter_class = UserFilter
```

The Communication app mostly uses views to set up User information.

```
def inbox(request):
      # Get the user associated with this session
      current_user = get_user(request)
      # If no user is logged in
      if not current_user:
            # Then force the user to log in
            return HttpResponseRedirect("{}?next_url={}".format(
                  reverse("login"),
                  reverse("inbox")))

      # Else get the current user and render them in the view
      context = {'user': current_user}
      return render(request, "inbox.html", context)
```

The Classes app also uses views to retrieve database information through API calls.

```
"""
View holding details of a selected class
"""
def class_details(request, id):
      # Retrieving a class with specified id
      url = 'http://root:8002/api/classes/' + str(id)
      thisclass = requests.get(url).json()
      # To get the currently enrolled number
      signedup = len(thisclass['students'])
      # Retrieving the creator of the class
      url2 = 'http://root:8002/api/users/' + str(thisclass.get("creator"))
      creator = requests.get(url2).json()
      # Retrieving the guild of the class
      url3 = 'http://root:8002/api/guilds/' + str(thisclass.get("guild"))
      guild = requests.get(url3).json()
      # Retrieving the acquired certification of the class
      url4 = 'http://root:8002/api/certifications/' +
str(thisclass.get("acquired_certification"))
      acqcert = requests.get(url4).json()
      # Retrieving the required certification of the class
      url5 = 'http://root:8002/api/certifications/' +
str(thisclass.get("prerequisites"))
      reqcert = requests.get(url5).json()
```

3.3.3 Frontend

The communication app does most database calls through Javascript in the frontend to have dynamically loaded information. Separate utility methods are used to get messages from the API.

```
// Get all direct messages tied to user
function getDirectMessages(user_id, successCallback) {
	// Retrieve base url, localhost if locally running or IP if remote
	let base_url = $('head base').attr('href');
	// Get user messages from API
	$.ajax({
		url: base_url.concat(`/api/messages/?to_direct=${user_id}`),
		success: successCallback
	});
};
```

This is displayed through mostly JQuery calls on a sorted list.

```
function displayMessages(selector, display) {
	// Filter for only general messages
	selector.empty();
	// Get all user ids who sent messages in unique array
	let userIds = Array.from(new Set(display.map(
				message => message.author)));
	getUserIdToName(userIds, function(userIdToName) {
		// Sort the messages by their date
		display.sort(compareDates);
		// For each message
		for(let i = 0; i< display.length; i++){
			// Display it
			displayMessage(
			selector,
			userIdToName[display[i].author],
			display[i].body,
			new Date(display[i].created_at).toDateString());
		}
	});
}
```

3.4 Sample Tests

Testing proves the validity and accuracy of software. Without testing, there are no assurances that software performs as intended. Testing also allows all edge cases to be considered, which helps developers find programming bugs in unexpected places that would not be discovered with typical usage.

Below are sample tests used to verify and validate the operation of this system.

- test_del_certs() - this tests to make sure that 1) the remove_cert url produces the correct output message, "Successfully Deleted", and 2) deletes certification testing data

```
def test_del_certs(self):
    certs = views.get_all()['cert_list']
    for cert in certs:
        e = c.get(reverse('remove_cert'), {'id': cert['id']})
        self.assertTrue("Successfully" in e.url)
        self.assertTrue("Deleted" in e.url)
        self.assertEqual(e.status_code, 302)
```

- test_validation_name() - this tests that the create_class form will not allow the class name to be empty or a string containing only whitespace characters

```
def test_validation_name(self):

    form_data = self.generate_valid_form_data()

    # Empty fails
    form_data['name'] = ""

    is_valid, x, y =
        create_class_views.check_validity(form_data, [])
    self.assertIs(is_valid, False)

    # Only spaces fails
    form_data['name'] = "   "

    is_valid, x, y =
        create_class_views.check_validity(form_data, [])
    self.assertIs(is_valid, False)
```

- test_message_home_get() - this tests the message page by mocking the possible users and guilds that could be sent to that are displayed on the page

```
def test_message_home_get(self):
      # Mock Users as that's first URL used by page
      user_url = "http://root:8002/api/users/"
      # Setup mocked response - expects a list of users who have
      #   a first name, last name, and id
      user_body = json.dumps([{
            "first_name": "Sam",
            "last_name": "Ting",
            "id": 1
            },
            {
            "first_name": "A",
            "last_name": "Tree",
            "id": 2
            }
            ])
      # Actually do the mocking with request type, url, and body
      httpretty.register_uri(httpretty.GET, user_url,
                             body=user_body)

      # Mock Guilds as that's second URL used by page
      guild_url = "http://root:8002/api/guilds/"
      # Setup mocked response - expects a list of guilds who
      #   have a name and id
      guild_body = json.dumps([{
            "name": "Runecrafting Guild",
            "id": 1
            },
            {
            "name": "Woodcutting Guild",
            "id": 2
            }
            ])
      # Actually do the mocking with request type, url, and body
      httpretty.register_uri(httpretty.GET, guild_url,
                             body=guild_body)

      # Now test if page reachable and has certain text
      response = self.client.get(reverse('message_home'))
      self.assertEqual(response.status_code, 200)
      # Response will have Woodcutting because I made
      #   a guild with that,
      #   which is then loaded on the page in an <li>
      self.assertContains(response, "Woodcutting")
```

- test_inbox_redirects_without_token() - this tests that the inbox page redirects if a user is not logged in, that is no valid cookie is present

```
def test_inbox_redirect_without_token(self):
    # Empty token to make get_user return nothing
    self.client.cookies.load("makerspace_auth=")
    response = self.client.get(reverse('inbox'))
    self.assertEqual(response.status_code, 302)
    self.client.cookies.load("makerspace_auth=" + self.token)
```

- test_certification_creates() - this test creates a certification and ensures that it has the correct instance

```
def test_certification_creates(self):
    # Create a certification object
    certification = Certification.objects.create(
        name="test name",
        teaching=True,
        machines=["test machine 1", "test machine 2"])
    # Ensure it was created by checking its instance type
    self.assertTrue(isinstance(certification, Certification))
```

- test_message_message_type_filter() - this test ensures the message type filter properly filters by message type, that is general, direct, or guild

```
def test_message_message_type_filter(self):
    # Create a message to get
    message = Message.objects.create(
        message_type="GE",
        body="test message body")
    message.save()

    # Ensure we've a message
    self.assertTrue(Message.objects.count() >= 1)
    # Make sure request gets an item
    url = '/api/messages/?message_type=GE'
    response = self.client.get(url, format='json')
    self.assertTrue(len(response.data) >= 1)
```

3.5 Code Coverage

Code coverage was determined using the python3 package coverage. To install and use coverage, install python3 (which is outside the scope of this report). Pip is the package management software installed with python3. Running the command `pip3 install coverage` will install the coverage package. To generate code coverage reports, navigate to the src/ folder in the Staunton Makerspace source code repository. Tests require the service to be running. This means running `docker-compose up` in src/. Then, run `docker exec -it CONTAINER bash`, where CONTAINER is either src_classes_1, src_communication_1, or src_root_1, depending on which system is being tested. Once inside the docker container, run the command `coverage run --source=. manage.py test && coverage report` to generate a comprehensive code coverage.

Below is the output of running this command in src_root_1:

```
Name                                          Stmts   Miss  Cover
-----------------------------------------------------------------
api/__init__.py                                   0      0   100%
api/admin.py                                     12      2    83%
api/apps.py                                       3      3     0%
api/filters.py                                   51      0   100%
api/migrations/0001_initial.py                    7      0   100%
api/migrations/0006_auto_20191120_1932.py         5      0   100%
api/migrations/0007_auto_20191120_2107.py         4      0   100%
api/migrations/0008_auto_20191121_2040.py         4      0   100%
api/migrations/0009_auto_20191129_0013.py         5      0   100%
api/migrations/0010_auto_20200208_0733.py         4      0   100%
api/migrations/0011_auto_20200208_0736.py         4      0   100%
api/migrations/0011_scan.py                       4      0   100%
api/migrations/0012_auto_20200130_2150.py         4      0   100%
api/migrations/0012_auto_20200208_0739.py         4      0   100%
api/migrations/0013_auto_20200205_2018.py         4      0   100%
api/migrations/0013_auto_20200209_1839.py         4      0   100%
api/migrations/0014_auto_20200317_1959.py         4      0   100%
api/migrations/0015_merge_20200329_0035.py        4      0   100%
api/migrations/0016_auto_20200329_0035.py         5      0   100%
api/migrations/__init__.py                        0      0   100%
api/models.py                                    77      0   100%
api/serializers.py                               56      0   100%
api/tests.py                                    233      0   100%
api/urls.py                                      12      0   100%
api/views.py                                     47      0   100%
manage.py                                        12      2    83%
root/__init__.py                                  0      0   100%
root/settings.py                                 20      0   100%
root/tests.py                                     6      0   100%
root/urls.py                                      4      0   100%
root/views.py                                     3      0   100%
root/wsgi.py                                      4      4     0%
-----------------------------------------------------------------
TOTAL                                           606     11    98%
```

Below is the output of running this command in src_classes_1:

```
Name                                        Stmts   Miss  Cover
----------------------------------------------------------------
classes/__init__.py                             0      0   100%
classes/settings.py                            21      0   100%
classes/urls.py                                 6      0   100%
classes/views.py                                3      1    67%
classes/wsgi.py                                 4      4     0%
classes_pages/__init__.py                       0      0   100%
classes_pages/admin.py                          1      0   100%
classes_pages/apps.py                           3      0   100%
classes_pages/forms.py                         12      0   100%
classes_pages/helpers/auth.py                  36      2    94%
classes_pages/helpers/helpers.py               17      0   100%
classes_pages/migrations/__init__.py            0      0   100%
classes_pages/models.py                         1      0   100%
classes_pages/templatetags/__init__.py          0      0   100%
classes_pages/templatetags/custom_tags.py      14      0   100%
classes_pages/tests.py                       1060     33    97%
classes_pages/urls.py                           3      0   100%
classes_pages/views/__init__.py                 7      0   100%
classes_pages/views/admin_cert_view.py         89      2    98%
classes_pages/views/admin_guild_view.py        66      4    94%
classes_pages/views/admin_mach_view.py         53      3    94%
classes_pages/views/admin_user_view.py         85      5    94%
classes_pages/views/admin_views.py             82      1    99%
classes_pages/views/create_class_views.py     254     13    95%
classes_pages/views/views.py                  273      0   100%
manage.py                                      12      2    83%
----------------------------------------------------------------
TOTAL                                        2102     70    97%
```

Below is the output for running the coverage command in communications:

Name	Stmts	Miss	Cover
communication\communication__init__.py	0	0	100%
communication\communication\context_processors.py	5	1	80%
communication\communication\settings.py	21	0	100%
communication\communication\urls.py	5	0	100%
communication\communication\views.py	0	0	100%
communication\communication\wsgi.py	4	4	0%
communication\frontend__init__.py	0	0	100%
communication\frontend\admin.py	1	0	100%
communication\frontend\apps.py	3	3	0%
communication\frontend\forms.py	12	0	100%
communication\frontend\helpers\auth.py	35	0	100%
communication\frontend\helpers\helpers.py	33	0	100%
communication\frontend\migrations__init__.py	0	0	100%
communication\frontend\models.py	1	0	100%
communication\frontend\test_auth.py	41	0	100%
communication\frontend\test_helpers.py	48	0	100%
communication\frontend\tests.py	246	0	100%
communication\frontend\urls.py	3	0	100%
communication\frontend\views.py	96	0	100%
communication\manage.py	12	2	83%
TOTAL	566	10	98%

All lines that the function determines are “missed” are either not testable or do not need testing, as they are automatically generated.

3.6 Installation Instructions

The following subsections will contain the installation instructions that are provided to get our software properly running on an EC2 instance.

3.6.1 AWS Setup

Go through the AWS portal in order to create an AWS account. An AWS EC2 instance will be created to host the server, so add a credit card or payment method if necessary.

3.6.2 Create the EC2 Instance

- Navigate to Amazon EC2 Dashboard
- Select "Instances"
- Select "Launch Instance" on the top left of the page
- Select a "Ubuntu Server 18.04 LTS (HVM), SSD Volume Type" x86 type
- Choose the instance type desired. For testing, the t2.micro should be sufficient
- Select "Review and launch"
- Select "Launch"
- Either create a key pair, or use an existing key pair, and select that option
- Select "Launch instances"

The instances should now be launching.

3.6.3 Configuring ports

Ports 80 and 8002 need to be open to the public. Do so as follows:

- On the EC2 instance listed, click the link to the security group
- Select "Actions" dropdown
- Select "Edit Inbound Rules"
- Add two rules:

- Port Range: 8003
- Source: Select custom, in box put 0.0.0.0/0 (or default value)
- Port Range: 8002
- Source: Select custom, in box put 0.0.0.0/0 (or default value)

Now you should be able to SSH into the instance (make sure you didn't delete port 22 from inbound rules). Go back to the instances page, find your instance, and type:

“ssh -i /path/to/private/key.pem ubuntu@ipaddress”

Where the path to the private key is where you saved your private key and ip address is from the instances screen (looking in the bottom right after selecting the instance).

3.6.4 Server Setup

Now that you're on the server, necessary setup is straightforward, agreeing to any prompts:

“ubuntu@ip-xxx-xx-xx-xxx:~$ sudo apt update”

“ubuntu@ip-xxx-xx-xx-xxx:~$ sudo apt-get install docker docker-compose”

“ubuntu@ip-xxx-xx-xx-xxx:~$ sudo systemctl enable docker --now”

“ubuntu@ip-xxx-xx-xx-xxx:~$ git clone”

“https://github.com/uva-cp-1920/Staunton_Makerspace.git”

“ubuntu@ip-xxx-xx-xx-xxx:~$ cd ~/Staunton_Makerspace/src”

“ubuntu@ip-xxx-xx-xx-xxx:~/Staunton_Makerspace/src$ sudo docker-compose up -d”

Once the server is up, install the migrations with:

“ubuntu@ip-xxx-xx-xx-xxx:~/Staunton_Makerspace/src$ sudo docker exec -it src_root_1 python manage.py migrate”

Now the server is hosted on port 8003. Should it be required to be hosted on port 80, add the inbound rule and change a few files:

File 1: “Staunton_Makerspace/src/docker-compose.yml” - In the “proxy” section, change the “ports” from “8003:8003” to “80:80”.

File 2: “Staunton_Makerspace/src/haproxy/haproxy.cfg” - In the “frontend http-in” section, change “bind *:8003” to “bind *:80”

If ports were changed, rerun docker with “sudo docker-compose up”. Reach the site at the IP address SSH'd into, either at port 8003 or no port, depending on if it was changed to port 80 or not. A simple 3 links should be shown.

The database is set up and the objects can be created through the API at “/api/”.

4. Results

Creating our web applications allowed for the makerspace to better fulfill its goal of being a place that encourages collaboration and creation. The two separate subsections of the application were the classes management system, and the communication system created for the space. These systems fulfilled two different goals that the makerspace leadership desired, creating a new system to allow them to create classes with one, and another system that was designed in order to better facilitate communication between members of the space.

For the classes management system, the team was able to design and create a system that allowed for the creation, and management of classes for members, and the public alike in a single house managed area. This resulted in an application that members of the makerspace can use to create classes and sign up for classes to be taken at the makerspace. In addition, this system also frees up public members to sign up for and take classes at the makerspace. This is all distilled into one singular system whereas before, classes were created as events, requiring new pages to be created on their existing website, as well as placing it on a google calendar, and communicating it out to members and the public through various email lists and other outreach methods. While the amount of time saved was not measured, the development of our application reduces the amount of systems needed from around 4 to 5 to hopefully only 1. In addition, the classes system allowed the makerspace to create an internal certification system to allow for safer regulation around the various tools in the space. In order to use some of the tools, there needs to be a certain level of training attained. This allows for the makerspace to have a codified system for tool access making it more clear for members of the space.

The communication system was created to create a centralized place for the members who are more resistant to using other technologies to still be easy to reach from the other members. Since the makerspace is a physical location that all members go to, creating a kiosk that automatically shows members their messages creates a low barrier system that more of their members would be willing to use. The system created will result in a kiosk established where all users can scan their existing RFID badges and get their messages to show. This allows the members who are less willing to check the systems like Slack and email to receive the messaging better. In addition, there is some built-in integration to existing systems so that using this new system does not create additional hassle. This should hopefully extend the readership of the general membership to around 100% of the members who actually go to the space. In addition, this allows the leadership to be more confident in messaging members who do not typically respond to things like email or slack messages, saving them time in looking for the members phone numbers, other contact methods, or relying on them physically showing up when leadership is there to tell them in person. The makerspace management team can interact with the system through the web application itself, to create, send, and read new messages to all members of the space. In addition, all users that are on the makerspace slack channel can send direct messages to the entire maker space on the system through the creation of a slack bot. This allows the members of the makerspace to save time when checking messages as well, with easy access to the messages upon arrival at the makerspace.

5. Conclusions

Issues previously faced by the Staunton Makerspace can be attributed to a lack of a strong central platform for communication and organization. These functions were instead fulfilled by separate web applications where the only link between said applications would be volunteers at the makerspace and the notes they took. The lack of cohesion that existed in this previous system created unnecessary friction in the everyday running of the makerspace and lowered its potential for growth.

The new system fills the preexisting void with two web based applications to address both the communication gap and the lack of organizational structure. For communication, it does so through a single, straight-forward messaging platform where messages can be viewed upon entering the makerspace and from the web portal. The solution addresses organization by presenting all class information, signup procedures, scheduling, and user profiles in the same place such that they are interconnected. The result is a much more accessible and structured base for the core communication and organization operations that transpire in the makerspace. As such, the burden on volunteers has decreased, the experience and ability to stay connected for members has improved, and the propensity for growth of the Staunton Makerspace has increased.

While each of the individual requirements presented by the Staunton Makerspace could be handled by a particular web application, the lack of integration between these applications resulted in disorganization and inefficiency. There exist platforms that seek to address this issue, however, they each have certain barriers to entry that eliminated them from contention. In the

end, a more customized system was required to satisfy their requirements. It seems likely that other organizations would face similar issues, particularly those of similar size and where membership is largely non committal. For such organizations, having a solution developed around their specific needs, as was done for the Staunton Makerspace, appears to be an effective way to improve the manner in which they operate and allows them more flexibility for growth.

6. Future Work

Given the relatively small amount of time and people appropriated to this task, and the limits of our experience to this point, there are many avenues of expansion for this project. For one, there are many optimizations or add-on features that would make our website work more effectively. We also do not have access to some of the information that the Makerspace needs to keep confidential, which limits exactly how much we can do.

The communications team realized along the way that setting up a system to send email messages through our site would require an additional email server setup, which is something the Makerspace or a future team could pursue. There is also likely a way to expand the number of forms of communication that our site can obtain if we had more prior research, instead of just using Slack. In addition, our fairly simple interface should be very easy to expand upon if desired.

To further the work done by the classes team, one significant development would be the use of a central styling guide. All required core functionality is present in the application, however, the team was never provided a styling sheet as they were expecting. This resulted in inconsistent styling practices which take away from the user experience and fluidity of the application, ultimately harming the human-computer interaction that transpires. The organization of models in the application will make it easy to modify their structure, and allows for more to be implemented.

There is also additional work that can be done in research to aid this project. Our site is secure and passwords are hashed, but there are more advanced cybersecurity methods out there

that could be used if desired (if, perhaps, this site grows to include more sensitive data). Research can also be done to see how people react to our system and how they choose to use it, and further modifications can then be made to improve the users' experience.

7. References

Dennerlein, S., Gutounig, R., Goldgruber, E., & Schweiger, S. (2016). Web 2.0 messaging tools for knowledge management? exploring the potentials of slack. Kidmore End: Academic Conferences International Limited.

Krüger, L. (2017, October 17). Time for a refresh: meet the new Google Calendar for web. Retrieved March 2, 2020, from https://www.blog.google/products/g-suite /time-refresh -introducing-new-look-and-features-google-calendar-web/

Staunton Makerspace (2018). Retrieved November 10, 2019, from http://www.stauntonmakerspace.com/

What is a Makerspace? Retrieved March 4, 2020, from https://www.makerspaces.com/what-is-a-makerspace/

AlertMedia Employee Notification System (2020) Retrieved March 29, 2020 from https://www.alertmedia.com/employee-notification-system/

GNU General Public License. (2007, June 29). Retrieved from https://www.gnu.org/licenses/gpl-3.0.en.html/