

Journey

- [Introduction](#)
- [Create Records Script](#)
- [HTML Page Details](#)
- [Auditing](#)
- [Security](#)
- [Microsoft Authentication \(OAuth\)](#)
- [SQLAlchemy](#)
- [Demo](#)

Introduction

The purpose of this project is to replace the current equipment preventive maintenance process and asset tracking. Currently, a different excel spreadsheet is used for every machine in the company. These are used to document the daily, weekly, and monthly maintenance being performed. There is also a separate asset log to track equipment at each location. I wanted to create something that was unified and made it easier to document preventive maintenance. With this app, users can easily see tasks that have not been completed. This can also be used for reporting and will allow necessary reports to be generated for the compliance team and external audits. Power BI can be used for even deeper reporting and visualizations in future developments.

Create Records Script

After building the database, I needed a way to generate the records that would be used for the app. I decided to create a script that is intended to run on a server once daily. This script creates records within the specified date ranges. The logic is a bit complex and could probably be simplified so that it is a little cleaner, but this is the high-level overview. The script iterates through the different frequencies, then iterates through all equipment where there is not a record within the specified date range. The logic branches to obtain the record number to be used for the record creation in the current iteration. It gets the last record for the lab of the equipment in the iteration, then increments by one and returns to the main function where the record is finally created.

HTML Page Details

login.html

This page began as a simple login form. I had a register.html to create an account which would generate a password hash and save it in the database. The login form would check this hash and redirect to index.html if it matched. Once I added user creation logic to the 'users' page, I removed register.html. Later, I added the below functionality:

- Lock user account after 3 invalid login attempts
- Perform checks before checking password hash such as user status and password change required
- If password change is required, then redirect to password_change.html
- Authentication with Microsoft. See [OAuth](#)

Within the login route, a check is performed to compare the last_pwd_chg date with the current date. There is also an environment variable for length of time before passwords expire so that it can be easily changed without needing to re-deploy the app. If the last password change exceeds that number, then the route redirects to password_change.html.

password_change.html

This page is a simple form to allow a user to change their password. The code includes validation to ensure only strong passwords are created. I also added functionality to toggle password visibility. The list of password requirements is displayed as well. I used JavaScript to dynamically change the color of these requirements to indicate when they are met as the user is typing. Once the password change is complete, it redirects back to login.

index.html

Upon successful login, the user is redirected to this page. This page is the primary purpose of the app. It displays the records in separate tables for each frequency. Using CSS, limited the size of the tables and allowed scrolling to prevent the web page from getting too large with hundreds of records. When the user opens a record, a modal pops up and displays the required tasks with checkboxes. Upon submission, the record is marked complete along with the username and completion date/time. The form results are also saved in a table.

Before I had a modal, I used a separate formresult.html to display the form for the opened record. When I learned about modals, I felt that this would contribute to an improved user experience. I then used these modals for all other forms except for the modify_models form where there was too much information to display in a single modal.

Permissions

- Administrator
 - Read-Write; can see all records for every lab.
 - The lab filter includes all labs.
- Global Audit
 - Read-Only; can see all records for every lab.
 - The lab filter includes all labs.
- Manager
 - Read-Write; can see records for current lab.
 - The lab filter only shows current lab.
- Local Audit
 - Read-Only; can see records for current lab.
 - The lab filter only shows current lab.
- Technician
 - Read-Write; can see records for current lab.
 - The lab filter only shows current lab.

manage.html

This page displays equipment and allows admins and managers to add and edit equipment. This could even be used to “transfer” equipment to another lab by changing the LabID. The status is expected to be updated when needed. For example, if the machine is in repair, then the status needs to reflect that. Any equipment that is obsolete is hidden from the table and the view can be toggled to show hidden items.

Permissions

- Administrator
 - Read-Write; can see all equipment for every lab.
 - The lab filter includes all labs.
 - Can edit and add equipment.
- Global Audit
 - Read-Only; can see all equipment for every lab.
 - The lab filter includes all labs.
 - Can only view, unable to edit or add equipment.
- Manager
 - Read-Write; can see equipment for current lab.

- The lab filter only shows current lab.
- Can edit and add equipment.
- Local Audit
 - Read-Only; can see equipment for current lab.
 - The lab filter only shows current lab.
 - Can only view, unable to edit or add equipment.
- Technician
 - No access.

models.html

This page displays a table of all equipment models. The view can be toggled to show disabled models and also allows admins to create new models. When the view button is clicked for a model, the modify_models.html is opened.




Permissions

- Administrator
 - Read-Write; can see all models.
 - Can add models.
- Global Audit
 - Read-Only; can see all models.
 - Can only view, unable to add models.
- Manager
 - Read-Only; can see all models.
 - Can only view, unable to add models.
- Local Audit
 - Read-Only; can see all models.
 - Can only view, unable to add models.
- Technician
 - No access.

modify_models.html

This page displays the model details when a model is opened from models.html. Only and admin is able to edit the details in a model. The purpose of this page is to allow an admin to create new tasks as needed. For example, if in the future, it was decided that an annual calibration is

required for a certain model, then an admin would enable the annual frequency and add a task for “Calibration” or other appropriate name.

 Currently, there is no logic to create records outside of the daily script. I’m not sure if this will ultimately be necessary, but I may add it in a future enhancement. [Link to Task](#)


Permissions

- Administrator
 - Read-Write; can modify model details.
- Global Audit
 - Read-Only; can view model details.
- Manager
 - Read-Only; can view model details.
- Local Audit
 - Read-Only; can view model details.
- Technician
 - No access.

users.html

This page displays a table with users along with the ability to create users and toggle disabled users. Each user can be opened to modify the user details. Part of this is a feature to change the password and set the require_pwd_chg value. The idea with that, is if a user forgets their password, they can go to their manager who will set a temporary password. When the user logs in with that password, they will be redirected to the password_change.html. This is also where the manager would go to unlock the user if they lock themselves out. They can simply change the status from ‘Locked’ to ‘Active’. The ‘Disabled’ status is intended to be set when a user is terminated or no longer needs access to the app. I decided against deleting users because this would prevent reports from being run since those would need to query the users table.

The second half of the user details modal displays the lab access for that user. The purpose is to give a user access to other labs so they can assist. Admins can see all labs and access levels while managers can only see labs they have access to and access levels at Manager and below.

 The feature of adding lab access may not be necessary so I might remove it. Also, when logging in with Microsoft, only one lab is available. If I want to expand the functionality, I would need to create a lot more security groups in Azure.



Permissions

- Administrator
 - Read-Write; can see all users at all labs.
 - Can add/edit users.
 - Can assign users to any lab with any permissions.
- Global Audit
 - Read-Only; can see all users at all labs.
 - Can only view users, unable to add/edit.
- Manager
 - Read-Write; can see users at labs they have access to, excluding Admins and Global Audit.
 - Can add users at labs they have access to.
 - Can edit all details for users whose primary lab is the current session's lab.
 - Can edit only password and status for users whose primary lab is different than the current session's lab.
 - Can assign lab access to users for labs the manager has access to. Can only assign Manager, Local Audit, and Technician access.
- Local Audit
 - Read-Only; can see users at current lab, excluding Admins and Global Audit.
 - Can only view users, unable to add/edit.
- Technician
 - No access.

reports.html

I don't yet know what direction I want to take this page. I set up SQL Server Reporting Services and began trying out report creation with Visual Studio. Ultimately, I believe Power BI will be used for running reports and adding visualizations. If set up properly, I could remove the audit permissions from my app since the compliance team would no longer need direct access to the app.

Auditing

When I began adding functionality for adding/modifying models/equipment/users, I decided to implement auditing as well. At first, I used a helper function in logic.py that I could call in the main app. I passed parameters for the values needed then used several if/elif statements to

build the event descriptions for the INSERT statement based on what variables were not null. Later, I realized that this wasn't very scalable and presented potential issues since it relied on variables being empty or not. There was simply too much room for error. I then changed the helper function to only require 2 variables, method and auditdata. The method was an integer that indicated different audit events such as new user created or model modified. The auditdata variable was a dictionary of the necessary data to pass to the function to build the event description. Once I integrated SQLAlchemy, I changed all of the audit function calls to use SQLAlchemy. This added extra lines to the main code, but ultimately it was better since it is now easier to see exactly what is going into the audit event and it can be changed if needed.

Security

This section describes the various security measures and vulnerability mitigations I have utilized in my project.

Tokens

This was the first security measure that I used in this project. While I was creating the formsubmit.html page, I noticed that the URL was displaying the parameters such as the record number. This would let users manipulate the URL and potentially submit forms incorrectly. To fix this, I learned about session tokens. Instead of the route redirecting to the formsubmit.html page, it would redirect to a route that creates the token based on the supplied parameters, then save the token in the session. I used these tokens to pass data to my HTML and JavaScript as well as to save data important for the user session.

CSRF

As I understand it, Cross-Site Request Forgery is a way for attackers to trick a browser into submitting POST requests using the user's session since they are authenticated. To mitigate this risk, I am using Flask-WTF CSRF protection. In my Flask app, I enabled CSRF protection globally which ensures that all requests require a valid CSRF token.

```
1 from flask_wtf.csrf import CSRFProtect
2
3 app = Flask(__name__)
4 csrf = CSRFProtect(app)
```

In my HTML forms, I include a hidden input field with the CSRF token which is checked by Flask-WTF upon submission.

```
1 <input type="hidden" name="csrf_token" value="{{ csrf_token() }}">
```

I include the CSRF token in a meta tag in the HTML page so that JavaScript can access it.

```
1 <meta name="csrf-token" content="{ csrf_token() }">
```

In my JavaScript, I then fetch the CSRF token from the meta tag and send it in the X-CSRFToken header for AJAX requests.

```
1 function fetchWithCSRF(url, options = {}) {
2   const csrfToken = document.querySelector('meta[name="csrf-
3   token"]').getAttribute('content');
4   return fetch(url, {
5     ...options,
6     headers: {
7       'Content-Type': 'application/json',
8       'X-CSRFToken': csrfToken,
9       ...(options.headers || {})
10    }
11  });
12 }
```

CORS

Cross-Origin Resource Sharing is used by browsers. This controls how a web page in one domain can access resources in another domain. This is blocked by default, but can be enabled through CORS. By using CORS in my app, I can specify what domains are allowed to access resources within my server. By using CORS(App) I allow any website to make requests by adding the appropriate CORS headers such as 'Access-Control-Allow-Origin:*'. In production, I should restrict CORS to my frontend domain if it is on a different domain or port from my server:

```
1 CORS(app, origins=["https://frontenddomain.com"])
```

HTTPS Redirect

This feature protects against attacks such as Man-In-The-Middle by requiring secure HTTPS protocol instead of HTTP. In my code, I check if the request is HTTP then redirect to HTTPS. Azure App Service and other platforms use the X-Forwarded-Proto header to hold the protocol. In my code, I get this header, then replace http in the URL with https and redirect with a 301 code to ensure the user permanently uses HTTPS.

Security Headers

By using these, I protect against different vulnerabilities such as XSS, clickjacking, and content sniffing. Below are the headers I have added:

- Content-Security-Policy (CSP)
 - Restricts where resources can be loaded from. I use this to ensure I can only load scripts and styles from my own domain.

- When I enabled this, my app stopped functioning because I had inline scripts and inline styles. To fix this temporarily, I set the CSP to allow scripts and styles from 'self' and 'unsafe-inline'. Once I removed the inline scripts/styles, I removed 'unsafe-inline'.
- This prevents XSS attacks.
- X-Content-Type-Options
 - By setting this to 'nosniff', this ensures that a browser will implicitly trust the declared type without guessing.
 - This prevents MIME-Type confusion attacks
- X-Frame-Options
 - By setting this to 'DENY', it ensures my site cannot be loaded within an iFrame.
 - Protects against clickjacking attacks.
- Referrer-Policy
 - By setting this to 'strict-origin-when-cross-origin', it restricts how much referrer info is sent with requests.
 - This protects user privacy.

Cross Site Scripting (XSS)

This security vulnerability allows an attacker to inject malicious scripts into web pages so they can steal cookies, hijack sessions, deface websites, or redirect users to malicious sites. Below are the different ways I mitigate this risk in my project:

- Template Auto-Escaping
 - In Flask Jinja2 templates, variables are auto-escaped by default. This means that special characters like "<, >, “ ” are converted to safe HTML entities.
- ```

1 <div>{{ username }}</div>
2
3 <!-- In this example, if the username is entered as: -->
4 <script>alert(1)</script>
5 <!-- Then it will be converted to plaintext instead of being used as a script. -->

```
- Content-Security-Policy
    - See [CSP](#)
  - No untrusted HTML injection
    - I ensure that raw user input is not rendered as HTML

## Validation and Sanitization


In my app I do not explicitly trust client-side validation. I make sure to validate input server-side as well as sanitize data. Examples of this include the use of Flask-WTF and SQLAlchemy.

## SQL Injection

See [SQLAlchemy](#)

## Access Control

I used access control throughout my project, both in frontend and backend. First, I made sure each Flask route required the user to be logged in, so I created a login-required decorator that could easily be added to each route. I then created a decorator for managing access requirements for specific routes. For example, the Users route requires Administrator, Manager, Global Audit, or Local Audit access. Since I used a decorator, I can easily modify access. I also use access control with Jinja 2 in my HTML pages to show/hide the navigation links and modify behavior of some elements. In my JavaScript code, I use access control to modify certain behavior. For example, only Administrators can modify equipment models while Managers, Global Audit, and Local Audit can only view the data. So, I was able to disable or hide buttons to prevent modification.

 Currently, I only use frontend access control for read-only vs edit access. I may need to change the logic in my Flask routes to prevent modification if permission denied. [Link to Task](#)

---

## Microsoft Authentication (OAuth)

Since the company uses Microsoft for authentication, I decided to integrate OAuth into my app. I added the app to Microsoft Entra ID in App Registrations. I then created different security groups that aligned with the access levels in the app. In the 'auth/callback' route, the required info is extracted from Microsoft, such as the groups and user info. The security groups are specifically named so that I can pull the lab and access level needed. This info is assigned to the appropriate session cookies. Just like the login route, this route also performs the necessary queries for labs, classes, etc. and saves them to the session. Upon logout, the user is redirected to the Microsoft logout endpoint, then returned back to the login page. For additional details, see [OAuth](#)

---

## SQLAlchemy

At the start of this project, I did not consider vulnerabilities in my code. Later, I did a security review and found that I was at risk of SQL injection in several areas, including the SQL files and inline SQL. I was using string concatenation to build my queries, even in my helper functions in logic.py. To mitigate the risk of SQL injection, I learned about parameterization and

subsequently modified all raw SQL to use this technique. While this technique protected my project from SQL injection, it wasn't very scalable. I then looked into other solutions, which brought me to SQLAlchemy, specifically Flask-SQLAlchemy. I learned that the features include built-in parameterization and connection pooling which I figured would improve performance since this app relies very heavily on SQL queries. Prior to SQLAlchemy I had touched on this a little bit by created a helper function to create single connections. I found that I was opening/closing connections several times within most routes which would probably lock up the database at some point with dozens of users utilizing the app. Since SQLAlchemy manages the opening/closing of connections, I can remove all that from my code, as well as the helper function.

With the integration of SQLAlchemy, I have mixed feelings about its usefulness as it pertains to code cleanliness and simplicity. In many instances, it simplified the code where it replaced raw SQL. For example, 2-3 lines turned into 1 line. I can also see that it will make future changes easier because of how objects are assigned as variables. If I modify tables in the database, it will also be easier to integrate those changes in my code. Where things get complex is the more complex SQL queries, specifically in the "get" functions where I use joins then transform the results into dictionaries to be used in the JSON for the frontend. The more of these I did, the easier it got as I understood it better, so I think with more practice I will see the benefits more clearly.

---

## Demo

My project has now reached a point where I can share it with others. In fact, it is ready to go to production as a first version. I needed a way to deploy a demo version so that others can try it out publicly. With my knowledge of Azure, I thought that would be a good start, but it can be costly. I researched the cost of the services I would need and eventually found that I could make it work for a minimal cost as long as I limit usage. Now that I identified the platform, it was time to build the demo.

### SQL Server

I started with migrating my SQL Server database to azure. I chose Azure SQL Server with DTU-based usage. This was the cheapest option. What I discovered later during testing was that this tier contributes to very slow performance. It's fine for a demo but would be wholly inadequate for any production solution. Once I created the SQL Server resource, I provisioned a SQL database. I then used Azure Data Studio on my local machine to access the database and replicate my current SQL Server database. To simplify this process, I used SSMS to generate

the “CREATE TO” queries for each table. I put these together into a single query, copied it to Azure Data Studio and ran the query to create all required tables in my database. I then added the base data such as Frequency, UserStatus, Access, etc. I also added some sample data to all tables instead of using any confidential or proprietary info.

#### Problems

- When I was trying to create my database in Azure, I kept getting an error that the resource could not be created due to subscription limits. I didn't target the source of the issue, but I thought it may be caused by the SQL server database through Azure Arc. So, I attempted creating the resource in a different region which was successful.

### App Service

After the SQL database was created, it was time to provision the app service and webapp. Provisioning was the easy part. Next, I had to prepare my code for deployment. The first step of this was to prepare my environment variables, including the SQL Server authentication info and Flask environment info. I added all necessary environment variables to the app service environment variables setting, then modified my code to use them. The last step was to deploy using the Azure CLI in PowerShell, which is where I ran into issues.

#### Azure CLI Commands

##### Deploy Code

```
1 az webapp up --name <app-name> --resource-group <resource-group-name> --runtime <runtime>
```

##### View Logs

```
1 az webapp log tail --name <your-app-name> --resource-group <your-resource-group>
```

#### Problems

- I could not deploy because the resource group was not found. After listing the resource groups, I found that the CLI was using the wrong subscription. After changing the default subscription, I was able to deploy successfully.
- Once deployed, the webapp would not open. Using the logs, I found that the flask session secret was empty. The issue was where the secret was being called in my code. I needed to move it so that it would be called in the beginning before the environment was set.
- After fixing the issue of the session secret, the logs showed another error where 'timedelta' was not defined. I had mistakenly removed 'import timedelta' prior to deploying.

- When I fixed the session secret and timedelta issues, my app still would not open, and the logs showed the same errors. I eventually learned that the logs seem to be lagging behind. When I viewed the logs in Azure, it would start them from several minutes' prior, then repeat everything. I kept making code changes and redeploying and it seemed that the same errors would occur. I also found that I needed to navigate to the webapp to initiate the first build. This took a couple minutes, but once complete, as long as the app service remained in the running state, I could access the app right away.

## OAuth

When I deployed the app, I had commented out the OAuth logic just in case it was the cause of my previous errors. Once those were addressed, I proceeded to re-implement OAuth so that it would work for the webapp in a production environment. To do this, I had to modify the app registration in Microsoft Entra ID so that it used redirect URIs for my webapp, instead of just localhost.

I added functionality for users to choose to sign in with Microsoft from the login page. Before, it would automatically route to Microsoft login and bypass the login route. I added the button to the login page by using Microsoft's requirements for the HTML/CSS. I then added a route that would be used specifically for this button, that way I could keep the normal login route, and have another login with Microsoft route. After verifying that I could log in both ways, I realized I needed a way to logout depending on how the user logged in. Before, it would just redirect to Microsoft logout. So, I added a session cookie for 'ms\_login' that was set at login. For normal login, this was set to false, and for Microsoft login it was set to true. Then, I was able to create a single logout route that would first check this session cookie and redirect to the login page or Microsoft page as needed.

### Problems


- I still could not login using Microsoft which turned out to be because of HTTPS. My code was using HTTP, but the redirect URIs required HTTPS. The fix for this was to add the scheme in the routes depending on whether the environment was production or development

```
1 url_for('auth/callback', _external=True, _scheme='https'))
```

- Whenever I logged out with Microsoft, it wouldn't redirect back to my login screen, even though I had this set in my code. I actually needed to add a new redirect URI in Entra ID for "/login" so that it would logout, then redirect back to my login page.

## Custom Domain

Once my app was successfully deployed, I wanted to add my “tybax.com ” custom domain. This was easier said than done because it just showed errors or would get stuck on “Get Certificates”. These are the steps that ultimately proved successful:

 App Service must be running before creating certificates, otherwise it will fail.

1. Add custom domain in Azure App Service for equipmansys.tybax.com.
2. In GoDaddy, add the DNS records using info generated from Azure.
  - a. asuid token
  - b. CNAME for equipmansys subdomain
3. Check for DNS propagation using a tool such as [DNS Checker - DNS Check Propagation Tool](#)
4. In App Service > Certificates, add a new managed certificate using the custom domain created in step 1. This should happen automatically but for me it was not, so I needed to do it manually.
5. In App Service > Custom Domain, add binding using the new certificate if it did not automatically bind.
6. In Entra ID app registrations, make sure to add new redirect URIs for this custom domain.