Milestone 2

Due 06/15/2025

Alumni Management System Functionality

1. Database Requirements - Aziz
2. Files Needed- Mar
3. File Types-Zach
4. Connectivity-Mar
5. System Architecture / Design Approach – Matt
6. Implementation of Functional Requirements - Mar
7. Page functionality – Matt

1. Database Requirements

Overview:

To support the needs of the Alumni Management System, the database structure is designed with simplicity, clarity, and real-world usability in mind. Instead of trying to reinvent the wheel, the structure is built around clear relationships between users and alumni, with additional layers to track important details like degrees, addresses, jobs, donations, and skills. Everything is linked properly to make it easy to query, update, and manage data while keeping the history of each alumnus intact.

Database Name: AlumniManagementDB

Key Tables:

Users

This table controls who can log in and what they can do in the system.

UID (PK) STRING - 20 characters (NOT NULL)
password STRING - 20 characters (NOT NULL)
fName STRING - 20 characters (NOT NULL)
lName STRING - 20 characters (NOT NULL)
jobDescription STRING - 50 characters
viewPriveledgeYN STRING - 1 characters
insertPriveledgeYN STRING - 1 characters
updatePriveledgeYN STRING - 1 characters
deletePriveledgeYN STRING - 1 characters

Alumni

Main profile information for each alumnus.

alumniID (PK) INTEGER (NOT NULL)
fName STRING - 50 characters (NOT NULL)
lName STRING - 50 characters (NOT NULL)
email STRING - 255 characters (NOT NULL)
phone STRING - 20 characters
DOB DATE
gender STRING - 50 characters
ethnicity STRING - 50 characters
website STRING - 255 characters
linkedin_link STRING - 255 characters

twitter_link STRING - 255 characters
facebook_link STRING - 255 characters
instagram_link STRING - 255 characters
guestSpeakerYN STRING - 1 character
newsLetterYN STRING - 1 character
imageThumb STRING - 255 characters
imageNormal STRING - 255 characters
description TEXT
deceasedYN STRING - 1 character
deceasedDT DATE
deceasedNotes TEXT

Addresses

Stores all addresses connected to each alumnus. One can be marked as the main address, but others are kept too.

addressID (PK) INTEGER (NOT NULL)
alumniID (FK) INTEGER (NOT NULL)
address STRING - 50 characters
city STRING - 50 characters
state STRING - 2 characters
zipCode STRING - 10 characters
activeYN STRING - 1 characters
primaryYN STRING - 1 characters

Degrees

Academic history: each entry shows what degree the alumnus earned.

degreeID (PK) INTEGER (NOT NULL)
alumniID (FK) INTEGER (NOT NULL)
major STRING - 50 characters (NOT NULL)
minor STRING - 50 characters
graduationDT DATE
university STRING - 100 characters
city STRING - 50 characters
state STRING - 2 characters

Employment

Work history with support for tracking changes over time.

EID (PK) INTEGER (NOT NULL)
alumniID (FK) INTEGER (NOT NULL)
company STRING - 50 characters (NOT NULL)
city STRING - 50 characters
state STRING - 2 characters
zip STRING - 10 characters
jobTitle STRING - 20 characters
startDate DATE
endDate DATE
currentYN STRING - 1 characters
notes STRING - 100 characters

Donations

Keeps track of any donations made by alumni.

donationID (PK) INTEGER (NOT NULL)
alumniID (FK) INTEGER (NOT NULL)
donationAmt NUMBER - Accept up to 9 digits before the decimal and two digits after the decimal (NOT
NULL)
donationDT DATE (NOT NULL)
reason STRING - 200 characters
description STRING - 200 characters

Skills

A way to log unique or valuable skills that alumni have.

SID (PK) INTEGER (NOT NULL)
alumniID (FK) INTEGER (NOT NULL)
skill STRING - 50 characters (NOT NULL)
proficiency STRING - 10 characters
description STRING - 100 characters

Newsletter

Allows for students to create newsletters on frontend website

NID (PK) INTEGER (NOT NULL)
newsDate DATE (NOT NULL)
headline STRING - 200 characters
description STRING - 200 characters
link STRING - 200 characters

fileLoc STRING - 200 characters

SentTo

Intermediary table used to track sent newsletters

alumniID (PK,FK) INTEGER (NOT NULL)
NID (PK,FK) INTEGER (NOT NULL)
sentDate DATE

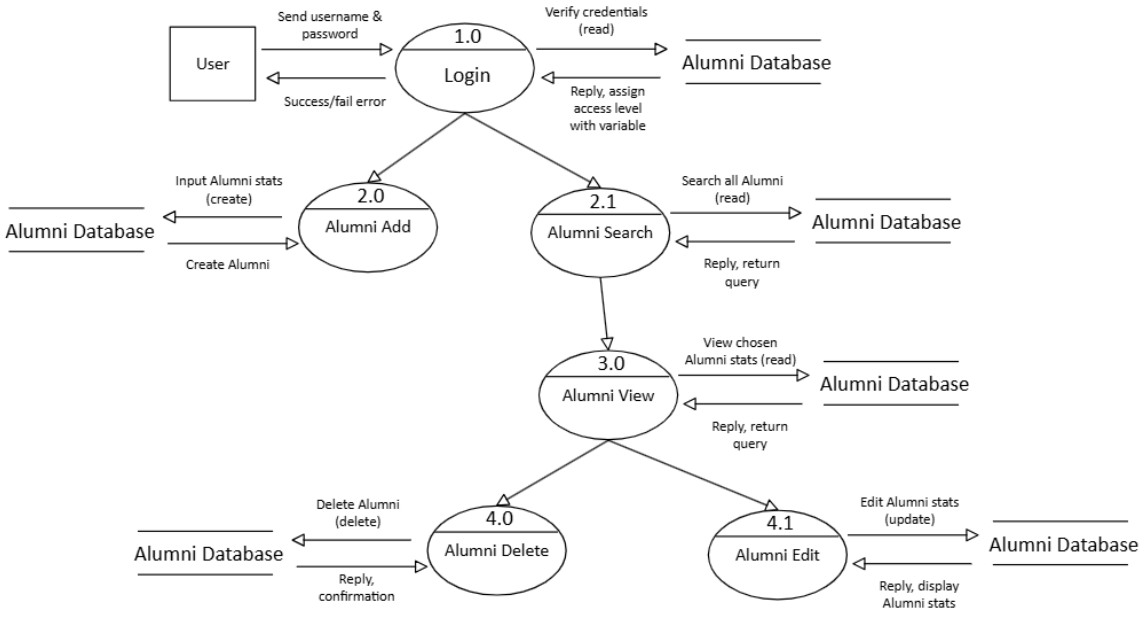
Each table is connected in a way that allows for flexible reporting and updates. Foreign keys ensure proper relationships between records. Data types like Boolean, Date, and Varchar are chosen to suit the kind of information being stored. Fields marked Nullable are optional, allowing for blank entries where needed, such as an ongoing job with no end date.

2. Files Needed:

Module	Front End Files	Backend Files	Purpose
Login	Login.html	auth_routes.py, user.py	Handles authentication using credentials stored in user table
Session Management	Flask login/sessions?	Session_manager.py	Manages login state and permission levels for each user.
Dashboard	Dashboard.html	Dashboard_routes.py	Main admin interface post-login navigation to each data module
Alumni management	Alumin_form.html Alumni_list.html	Alumni_routes.py Alumni.py	CRUD functionality for base alumni records.
Address management	Address_form.html Address_list.html	Address_routes.py Address.py	Allows multiple addresses with flag control. (activeYN, primaryYN)
Degree Tracking	Degree_form.html Degree_list.html	Degree_routes.py Degree.py	Capture of degree history with sorting by graduation date.
Employment History	Employment_form.html Employment_list.html	Employment_routes.py Employment.py	Job tracking with control over currentYN flag.
Donations	Donation_form.html Donation_list.html	Donation_routes.py Donation.py	Recording donation amounts, dates, and reasons.

Skillsets	Skills_form.html Skills_list.html	Skillset_routes.py Skillset.py	Entry of specialized skills and proficiency descriptors
Validation & forms			WTForms setup
Database Connection		Db_connect.py Config.py	Central SQLAlchemy setup and DB config
Application Entry point		Run.py _init_.py	Starts flask app and registers blueprints.

A level one Data Flow Diagram of the backend website is shown below:



3. File Types:

Frontend Files (User Interface Layer)

Purpose: HTML templates that provide the user interface components
Total Files: 8 files
File Extension: .html

File Name	Purpose
Login.html	User authentication interface

Dashboard.html	Main dashboard and navigation
Alumni_form.html	Alumni data entry form
Alumni_list.html	Alumni records display
Address_form.html	Address information form
Address_list.html	Address records display
Degree_form.html	Academic degree entry form
Degree_list.html	Degree records display

Backend Routes & Logic (Application Layer)

Purpose: Python files handling API routes, business logic, and request processing
Total Files: 12 files
File Extension: .py

File Name	Purpose
auth_routes.py	Authentication and login routing
Dashboard_routes.py	Dashboard functionality routing
Alumni_routes.py	Alumni CRUD operations routing
Address_routes.py	Address management routing
Degree_routes.py	Degree information routing
Employment_routes.py	Employment history routing
Donation_routes.py	Donation tracking routing
Skillset_routes.py	Skills management routing
Session_manager.py	User session management

Data Models & Extended Forms (Data Layer)

Purpose: Python model classes and additional form templates for data management
Total Files: 11 files
File Extensions: .py, .html

File Name	Type	Purpose
user.py	Model	User account data structure
Alumni.py	Model	Alumni record data structure
Address.py	Model	Address information data structure
Degree.py	Model	Academic degree data structure
Employment.py	Model	Employment history data structure
Donation.py	Model	Donation record data structure
Skillset.py	Model	Skills and competencies data structure
Employment_form.html	Form	Employment data entry interface
Employment_list.html	List	Employment records display
Donation_form.html	Form	Donation entry interface
Donation_list.html	List	Donation records display
Skills_form.html	Form	Skills entry interface

Skills_list.html	List	Skills records display

Configuration & Core Files (System Layer)

Purpose: System configuration, database connection, and application initialization
Total Files: 4 files
File Extension: .py

File Name	Purpose
Run.py	Application entry point and server startup
Config.py	System configuration and settings
Db_connect.py	Database connection management
init.py	Python package initialization

Flask Framework Structure:

- The system follows Flask's Model-View-Controller (MVC) pattern
- HTML templates serve as Views (Frontend)
- Python route files serve as Controllers (Backend)
- Python model files serve as Models (Data Layer)

File Organization Benefits:

- Clear separation of concerns
- Modular design for easy maintenance
- Scalable architecture for future enhancements
- Consistent naming conventions

Technology Stack:

- Backend: Python Flask
- Frontend: HTML templates
- Database: Configured via Db_connect.py
- Session Management: Flask sessions with custom session manager

4. Connectivity:

The Alumni management system uses secure, modular connection architecture to integrate the flask-based application with a MySQL database. It leverages SQLAlchemy as the Object-Relational Mapper (ORM), with connection credentials managed through a centralized configuration file. Sessions and permissions are handled through Flask’s built in session system, ensuring secure and role-based access to all modules.

Connection setup:

The database connection string is stored in config.py:

```
SQLALCHEMY_DATABASE_URI = 'mysql+pymysql://username:password@localhost/alumni_cms'
SQLALCHEMY_TRACK_MODIFICATIONS = False
SECRET_KEY = 'supersecretkey'
```

To promote reusability and clean separation of concerns, a dedicated **db_connect.py** defines the SQLAlchemy instance:

```
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()
```

The instance is initialized in the application factory **(_init_.py**):

```
from flask import Flask
from config import Config
from app.utils.db_connect import db

def create_app():
    app = Flask(__name__)
    app.config.from_object(Config)
    db.init_app(app)
    return app
```

This structure ensures that all model and route files import the same shared **db** object, maintain consistency and minimizing connection overhead.

Upon successful login (handled in **auth_routes.py**), user credentials and permission flags are stored in Flask's **session** object:

```
session['user_id'] = user.UID
session['permissions'] = {
    'view': user.viewPriveledgeYN == 'Y',
    'insert': user.insertPriveledgeYN == 'Y',
    'update': user.updatePriveledgeYN == 'Y',
    'delete': user.deletePriveledgeYN == 'Y'
}
```

These session variables are checked in all module routes before allowing the user to access or modify data.

To initialize Flask Login for advanced session tracking:

```
from flask_login import LoginManager

login_manager = LoginManager()
login_manager.init_app(app)
```

The **User** model implements methods like **is_authenticated** and **get_id()** to comply with Flask login's requirements.

Environment Variable support (optional)

To enhance security and support deployment, sensitive information such as database URIs and secret keys, they can be moved into a **.env** file.

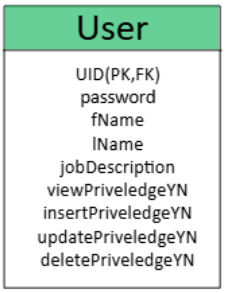
```
Ini
DATABASE_URL=mysql+pymysql://username:password@localhost/alumni_cms
SECRET_KEY=mysecuresecret

Loaded using python
import os
SQLALCHEMY_DATABASE_URI = os.getenv("DATABASE_URL")
```

5. Functions of each page

Login Page

When a user visits the login page, they see a simple HTML form with fields for their username and password. After they fill this out and click "Login," the browser sends these credentials to the Python server. The Python code receives this information and runs a query against the SQL database to find a user with a matching username and password. If a match is found, the code then reads the permission columns (the 'Y' and 'N' values for view, insert, update, and delete) from that same user's row in the database. The server then creates a "session," that stores the user's identity and their specific permissions in this session and sends it back to the user's browser. Finally, the server redirects the user to the alumni search page. For every subsequent page they visit, their browser automatically preserves the session for **two hours** so the system always knows who they are and what actions they are permitted to perform.

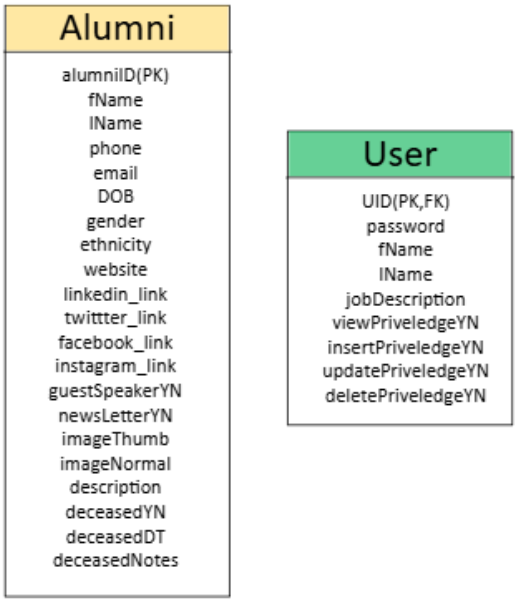


Alumni Search Page

When a user successfully logs in, they are sent to the Alumni Search page. The Python server immediately fetches the entire list of alumni from the Alumni table in the database, specifically attributes "alumniID", "fname", "lname", "phone", and "email" and displays it as a table in the HTML page the user sees. Before showing the page, the server first checks the user's session variable for their permissions. If the variable says they have "insert" permission, the server adds an "Add New Alumnus" button to the top of the page.)

On the page, the user sees search bars above each column. When they type into these bars and hit enter, the browser sends their search terms back to the Python server. The server then performs a new,

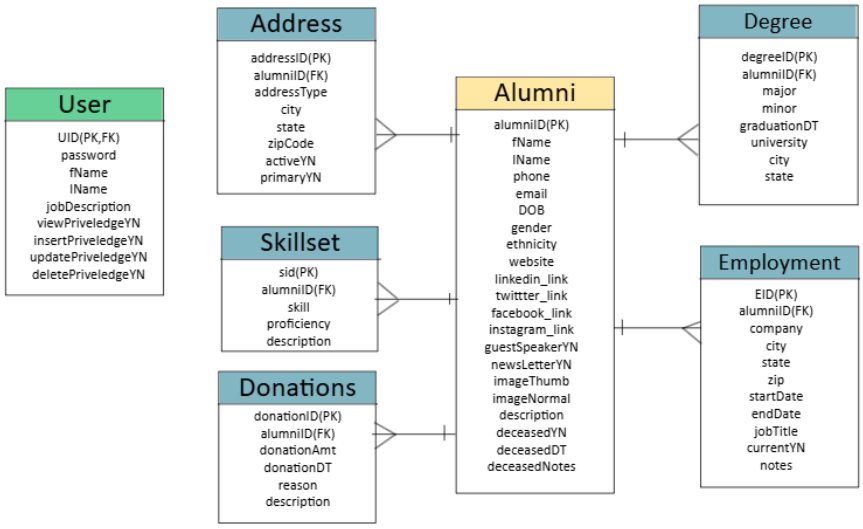
more specific search on the database, asking for only the alumni who match those terms (like "find all alumni whose last name contains 'Smith'"). It then rebuilds the page with just the matching results and sends it back to the user. If the user's session shows they have "view" permission, it creates a "View Details" button for that specific alumnus, with a unique link that leads to that person's full profile. If they don't have view permission, no button appears for them.



Alumni View Page

When a user clicks the "View Details" button for a specific person on the search page, their browser navigates to a unique URL that includes that alumnus's ID. The Python server receives this request and first verifies the user's session to ensure they have "view" permission. Using the ID from the URL, the server then performs a series of targeted queries on the SQL database. It fetches the main profile information from the alumni table, and then separately retrieves all associated records from the address, skillset, donations, employment, and degree tables that are linked to that specific ID.

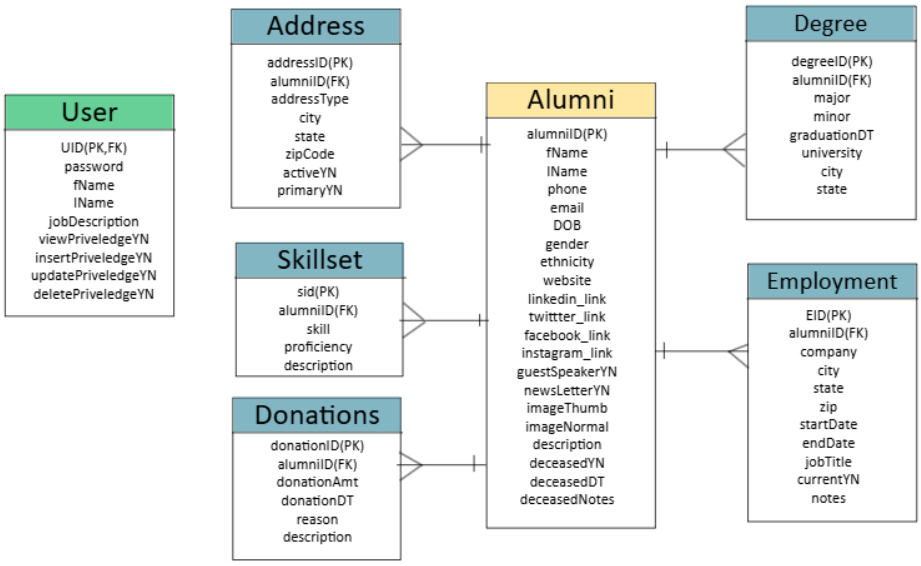
With all the data gathered, the server begins to construct the HTML page. It displays the main profile information at the top, followed by distinct sections for each category of related data, each under its own clear header like "Address History" or "Employment History". Before finalizing the page, the server checks the user's digital pass one more time, this time looking for the "update" permission. If the updatePriveledge attribute in the User table is 'Y', the server adds an "Edit Alumnus" button to the top of the page, with a link that points directly to the edit page for that specific alumnus. If the attribute is 'N', the button is simply not included in the HTML. Finally, the complete page, showing all the alumni's details and the correct button visibility, is sent to the user's browser



Alumni Edit Page

On the Alumni edit page, the server uses the alumnus's ID to gather all their information from the main alumni table and all related tables like address, degree, employment, skillset, and donations. However, instead of displaying this data as plain text, the server renders it inside editable HTML form fields. The alumnus's name appears in a text box, their description in a larger text area, and so on, turning the entire page into one comprehensive form.

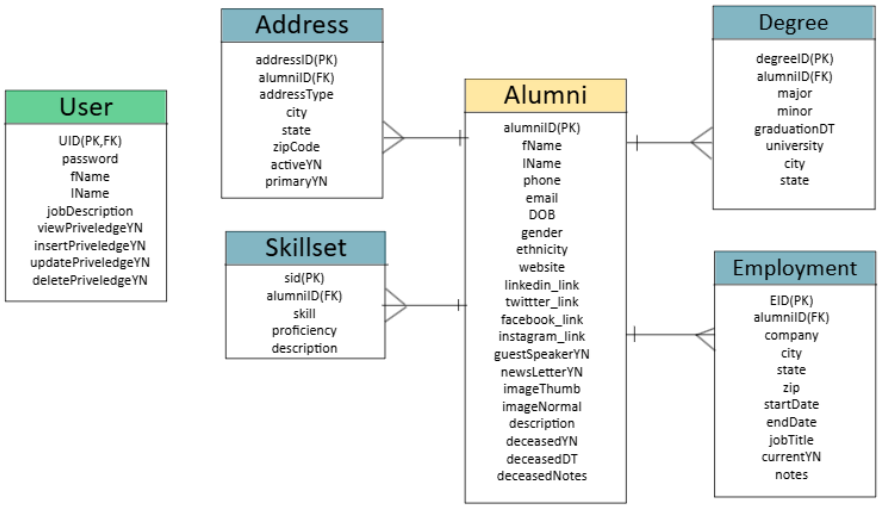
Before sending the page to the user, the server performs one final permission check, this time for deletePriveledgeYN. If the user's digital pass shows a 'Y' for this permission, a "Delete Alumnus" button is added to the page. If the permission is 'N', that button is not included in the HTML. The user can then make changes to any field and click a "Save Changes" button to submit the form, which tells the server to update the database with the new information.



Create Alumni Page

Once the user is authorized via the createPriveledgeYN attribute in the User table, the server constructs and sends a completely blank version of the alumni form to the user's browser. This page is structured identically to the edit page, with empty input fields for the core alumni details, address, employment history, degrees, and skills. The donations section is omitted by design, as it’s not assumed every alumnus has donated.

The user then fills out all the necessary information for the new alumnus. When they click the "Create" button, all the data from the various form sections is bundled together and sent to the Python server. First, it inserts the main profile information into the alumni table, which causes the SQL database to generate a new, unique alumniID for this person. Using this brand new ID as a key, the server then inserts the corresponding address, employment, degree, and skill records into their respective tables. After all the data is successfully saved to the database, the server redirects the user to the "View Details" page for the alumnus they just created.


User
UID(PK,FK)
password
fName
lName
jobDescription
viewPriveledgeYN
insertPriveledgeYN
updatePriveledgeYN
deletePriveledgeYN
Address
addressID(PK)
alumniID(FK)
addressType
city
state
zipCode
activeYN
primaryYN
Skillset
sid(PK)
alumniID(FK)
skill
proficiency
description
Alumni
alumniD(PK)
fName
lName
phone
email
DOB
gender
ethnicity
website
linkedin_link
twittter_link
facebook_link
instagram_link
guestSpeakerYN
newsLetterYN
imageThumb
imageNormal
description
deceasedYN
deceasedDT
deceasedNotes
Degree
degreeID(PK)
alumniID(FK)
major
minor
graduationDT
university
city
state
Employment
EID(PK)
alumniID(FK)
company
city
state
zip
startDate
endDate
jobTitle
currentYN
notes

6. Implementation of functionality

User authentication

Files:

Auth_routes.py

User,py

Login.html

```
POST /login:
    get UID and password from form
    query user table for UID/password match
    if valid:
        store UID and permission flags in session
        redirect to /dashboard
    else:
        return login error
```

Alumni CRUD Operations

Files:

Alumni_routes.py, alumni.py

Alumni_form.html, alumni_list.html

```
GET /alumni/list:
    if session['permissions']['view']:
        query all alumni from Alumni table
        render alumni_list.html with results

POST /alumni/add:
    if session['permissions']['insert']:
        read form fields
        create new Alumni object
        db.session.add(obj)
        db.session.commit()

GET /alumni/edit/<id>:
    if session['permissions']['update']:
        load alumni record by ID
        render alumni_form.html with fields pre-filled

POST /alumni/update/<id>:
    update alumni table with form data
```

Address management with Flag Control

Files:

address_routes.py, address.py

address_form.html, address_list.html

Special Logic: Only one primaryYN = 'Y' and multiple activeYN = 'Y' allowed.

```
POST /address/add:
    if form['primaryYN'] == 'Y':
        update all other addresses for this alumniID to primaryYN = 'N'
    insert new address with given flags
```

Degree history tracking

Files:

degree_routes.py, degree.py

degree_form.html, degree_list.html

```
GET /degree/list/<alumniID>:
    load all degrees for that alumni
    sort by graduationDT ascending
```

Employment history with **currentYN** logic

Files:

employment_routes.py, employment.py

employment_form.html, employment_list.html

Special Logic: Only one job per alumni should have **currentYN = 'Y'.**

```
POST /employment/add:
    if form['currentYN'] == 'Y':
        set all existing employment entries for alumni to currentYN = 'N'
    insert new job record
```

Donations

Files:

donation_routes.py, donation.py

donation_form.html, donation_list.html

```
POST /donation/add:
    create new Donation object with donationAmt, donationDT, reason
    link to alumniID
    db.session.add(obj)
    db.session.commit()
```

Skillset Entry

Files:

skillset_routes.py, skillset.py

skills_form.html, skills_list.html

```
GET /skills/<alumniID>:
    load all skills for that alumnus

POST /skills/add:
    insert new skill with description and proficiency
```

Permission Enforcement

```
if not session['permissions']['insert']:
    return redirect('/unauthorized')
```