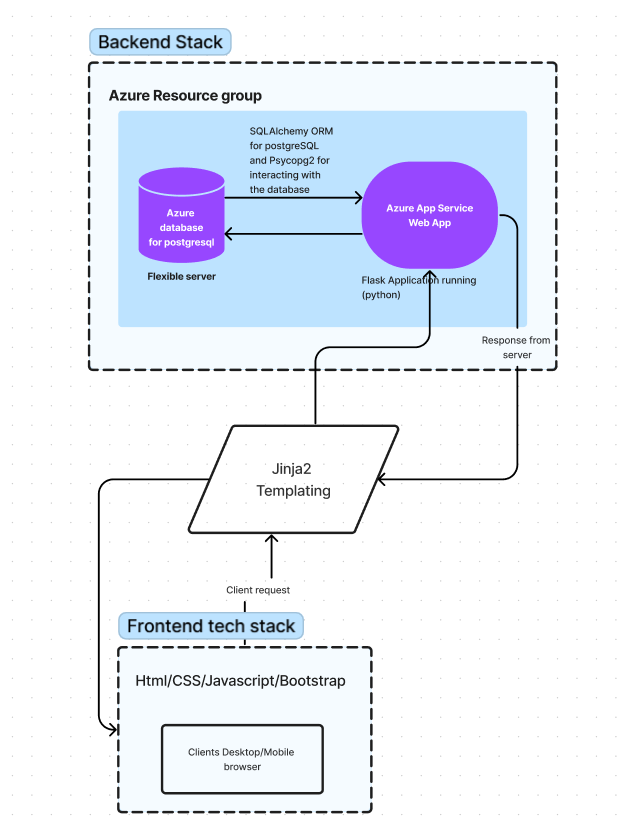
Technical Documentation

**The Tech Stack**

**Introduction:**

This technical documentation provides an overview of a Flask application that is deployed on Azure App Service and utilizes an Azure Database for PostgreSQL for the database layer. The application has a frontend built using HTML, CSS, JavaScript, and Bootstrap. The backend uses Flask, SQLAlchemy ORM, and psycopg2 for database connectivity. Jinja2 templating is used for rendering HTML templates.



# **Backend**

The backend of the application is built using Flask, which is a micro web framework written in Python. Flask is used to define the routes and the functionality of the web application. SQLAlchemy ORM is used as an abstraction layer between the application and the PostgreSQL database. SQLAlchemy allows for easy mapping of Python classes to database tables, making it easier to query and manipulate data.

psycopg2 is used as a PostgreSQL database adapter for Python, which provides a low-level interface for communicating with the database. psycopg2 is used to establish a connection to the PostgreSQL database and execute SQL queries.

The application is horizontally scalable using Azure App Service, which allows the application to be deployed on multiple instances, each handling a portion of the incoming requests. Additionally, the application is vertically scalable as well, allowing for the application to be scaled up or down based on the demand.

# **Frontend**

The frontend of the application is built using HTML, CSS, JavaScript, and Bootstrap. HTML is used to define the structure of the web pages, CSS is used to style the pages, and JavaScript is used to add interactivity to the pages.

Bootstrap is a popular front-end framework that makes it easier to create responsive and mobile-first websites. Bootstrap provides a collection of CSS and JavaScript files that can be used to create a responsive design that adapts to different screen sizes.

Jinja2 Templating: Jinja2 is a templating engine for Python that allows for the separation of logic and presentation. Jinja2 templates are used to define the HTML structure of the web pages and include placeholders for dynamic content. The dynamic content is then provided by the Flask application at runtime. Jinja2 also allows for the creation of macros and templates that can be reused across multiple pages.

**Scaling**

The application can be scaled horizontally and vertically based on the demand. The Azure App Service allows for the application to be deployed on multiple instances, each handling a portion of the incoming requests. The application can also be scaled vertically by increasing the compute resources available to the application.

**The Application**

# **Introduction**

The Flask application package structure includes the following directories and files:

* \_\_init\_\_.py: This file is executed when the package is imported and initializes the Flask application instance.
* static/: This directory contains static files such as CSS, JavaScript, and images.
* templates/: This directory contains HTML templates that are used to render dynamic content.
* models.py: This file contains the database models and schema definitions.
* views.py: This file contains the Flask view functions that handle HTTP requests.
* requirements.txt: This file lists the required Python packages and their versions.

This file structure allows for a clear separation of concerns and easy maintainability of the Flask application code.

# **Running the application locally**

Here are the general steps to run a Flask application:

1. Open the terminal or command prompt.
2. Clone the git repository from https://github.com/Ngala254/Equity-alumni-connect.
3. Navigate to the directory that contains the Flask application.
4. Activate the virtual environment, if you're using one.
5. Run `pip install -r requirements.txt` to install all the required modules.
6. Run the Flask application by running the command **flask run**.
7. Open a web browser and go to the URL provided in the output of the previous command (usually [**http://127.0.0.1:5000/**](http://127.0.0.1:5000/)).

# **Deploying the app to Azure app services**

The general procedure for deploying a Flask app to Azure:

1. Create an Azure account if you don't already have one.
2. Create a new web app in Azure. You can do this through the Azure portal by clicking "Create a resource" and searching for "Web App". Choose the subscription, resource group, and app name, then select the operating system as "Linux" and the runtime stack as "Python".
3. Set up deployment from a Git repository. Under the "Deployment" section in the web app's settings, choose "Deployment Center" and select "External Git" as the source. Follow the instructions to link your Github to the web app.
4. Configure the Python environment by creating a **requirements.txt** file in the root directory of your Flask app. This file should list all of the Python packages required by your app, one per line. You should also create a **runtime.txt** file that specifies the version of Python to use. For example, if you're using Python 3.8, the contents of **runtime.txt** should be:

Copy code

python-3.8

1. Configure the Flask app by creating a **config.py** file in the root directory of your app. This file should define any configuration settings required by your app, such as database connection strings or secret keys.
2. Push your Flask app to the Git repository linked to your web app. Azure will automatically build and deploy the app.
3. Test the deployed app by visiting the URL of your web app in a browser. If everything is working correctly, you should see your Flask app running on Azure.

# **Modules functions**

## **App.py**

This is the entry point of the web application. It first imports the create\_app function from the website module and then creates an instance of the Flask application by calling this function. Finally, it starts the Flask development server to run the application by calling the run method on the app object. The debug=True argument enables debugging mode for the Flask application, allowing for more detailed error messages and other helpful features during development. The if \_\_name\_\_ == '\_\_main\_\_': block ensures that the app.run() method is only called when this script is executed directly, and not when it is imported as a module in another script.

## **\_\_init\_\_.py**

This document provides technical documentation for the Flask application defined in the code snippet provided. The application utilizes a variety of libraries and modules to create a web application that can interact with a PostgreSQL database.

### Libraries and Modules Used

* Flask: a micro web framework that provides tools for building web applications in Python.
* Flask-SQLAlchemy: a Flask extension that provides a simple way to use SQLAlchemy with Flask.
* os: a module that provides a way of using operating system dependent functionality.
* Flask-Login: a Flask extension that provides user authentication and session management capabilities.
* sqlalchemy: a SQL toolkit and Object-Relational Mapping (ORM) library for Python.
* Flask-Migrate: a Flask extension that handles SQLAlchemy database migrations.
* builtins: a module that provides a set of built-in functions in Python.

### **Application Structure**

* The **create\_app()** function initializes the Flask app instance and sets the necessary configuration parameters.
* The SQLAlchemy object is initialized with the Flask app using **db.init\_app(app)**.
* The app's blueprints are registered using **app.register\_blueprint()** for the views and auth blueprints respectively.
* The User and Note models are imported using **from .models import User, Note**.
* The database tables are created using the **db.create\_all()** method within a Flask app context.
* Flask-Login is initialized using **login\_manager = LoginManager()** and **login\_manager.init\_app(app)**, and a user loader function is defined using **@login\_manager.user\_loader**.
* Flask-Migrate is initialized using **migrate = Migrate(app, db)**.
* A function called **jinja2\_zip()** is defined to pass as a global variable to Jinja2, and is added to the Jinja2 global variables using **app.jinja\_env.globals.update(zip=jinja2\_zip)**.

### **Configuration Parameters**

* **SECRET\_KEY**: A secret key for securely signing session cookies and other security-related needs.
* **SQLALCHEMY\_DATABASE\_URI**: The URI for accessing the PostgreSQL database.
* **SQLALCHEMY\_TRACK\_MODIFICATIONS**: Enables tracking modifications for SQLAlchemy.

## **Auth.py**

This code is a Python script that contains a Flask blueprint for user authentication and related routes.

First, the necessary modules are imported: Flask, Blueprint, render\_template, request, flash, redirect, url\_for, werkzeug.security, db, and flask\_login.

* "***auth***" is created using the Blueprint class from Flask. The blueprint is defined with the route "***/admin***" and takes no arguments. It returns a rendered HTML template called "admin.html", which shows a list of all users in the system, passed in as a variable named "total\_users", and the current user object, passed in as a variable named "user".
* ***"/login***". It is set to accept both GET and POST requests. In the event of a POST request, it takes the user input for email and password, queries the database for the user with the matching email, checks if the password provided matches the hashed password in the database, and logs the user in if the password is correct. If the password is incorrect, an error message is displayed, and if the email is not in the database, a message indicating the need to create an account is displayed. The route returns a rendered HTML template called "login.html", which displays a login form and the current user object, passed in as a variable named "user".
* The next route is "/logout". It requires the user to be logged in, as it is decorated with @login\_required. When a user accesses this route, they are logged out of the system and redirected to the login page.
* "/signup". This route accepts GET and POST requests. In the event of a POST request, it takes the user input for first name, last name, email, password, and password confirmation, checks if the email already exists in the database, and if not, creates a new user with the provided information. The new user is then logged in and redirected to the "/create-profile" page to complete their profile. If the email already exists in the database, an error message is displayed. If any of the inputs are invalid, such as the email being too short, or the password confirmation not matching the password, an error message is displayed. The route returns a rendered HTML template called "signup.html", which displays a signup form and the current user object, passed in as a variable named "user".
* "/create-profile". This route requires the user to be logged in, as it is decorated with @login\_required. In the event of a POST request, it takes the user input for various fields required to complete their profile, creates a new AlumniScholarProfiles object with the provided information, and saves it to the database. If the profile is created successfully, a success message is displayed. The route returns a rendered HTML template called "create\_profile.html", which displays a form to create a user profile and the current user object, passed in as a variable named "user".

## **Views.py**

This is code includes several routes to render different HTML templates for a the web application. It imports necessary modules and models from Flask and SQLAlchemy, sets up a Blueprint object, and defines several routes.

The routes include:

* home(): renders the home page ('home.html'), which includes the current user as a parameter.
* about(): renders the about page ('about.html'), which also includes the current user as a parameter.
* events(): renders the events page ('events.html') and queries the Events model to return all events in ascending order by their date. The current user is also included as a parameter.
* create\_event(): renders the 'create\_event.html' template and allows users to create new events by entering information in a form. If the request method is POST, the new event is created and added to the database, and the user is redirected to the events page.
* careers(): renders the careers page ('careers.html') and queries the Careers model to return all job listings in ascending order by their date. The current user is included as a parameter.
* create\_job\_listing(): renders the 'create\_job.html' template and allows users to create new job listings by entering information in a form. If the request method is POST, the new job listing is created and added to the database, and the user is redirected to the careers page.
* find\_alumni(): renders the 'find\_alumni.html' template and queries the User model to return all alumni in alphabetical order by their first name. The current user is included as a parameter.
* delete\_note(): receives a POST request with a note ID to delete and, if the current user is the owner of that note, deletes the note from the database.

The code also includes necessary imports of modules and models, and uses Flask's login\_required decorator to ensure that certain routes (create\_event()) require a logged-in user. Additionally, it uses Flask's generate\_csrf() function to generate a CSRF token for form submissions to prevent cross-site request forgery attacks.

## **Models.py**

The provided code defines four different SQLAlchemy models: Note, User, AlumniScholarProfiles, Events, and Careers. These models are used to create corresponding database tables with the specified columns.

The **Note** model has four columns: **id**, which is the primary key of the table and an auto-incrementing integer; **data**, which is a string that can hold up to 10000 characters; **date**, which is a datetime object with a default value set to the current time; and **user\_id**, which is a foreign key that refers to the primary key of the **User** model.

The **User** model has six columns: **id**, which is the primary key of the table and an auto-incrementing integer; **email**, which is a unique string that can hold up to 150 characters; **password**, which is a string that can hold up to 150 characters; **firstname** and **lastname**, which are strings that can hold up to 150 characters each; and **notes**, which is a relationship that connects the **User** model to the **Note** model.

The **AlumniScholarProfiles** model has twelve columns: **id**, which is the primary key of the table and an auto-incrementing integer; **scholars\_code**, which is a string that can hold up to 150 characters; **primary\_number** and **secondary\_number**, which are strings that can hold up to 20 characters each; **country**, **home\_county**, and **current\_county**, which are strings that can hold up to 150 characters each; **equity\_home\_branch**, which is a string that can hold up to 150 characters; **school\_university\_college**, which is a string that can hold up to 150 characters; **course**, which is a string that can hold up to 200 characters; **interests** and **hobbies**, which are strings that can hold up to 500 characters each; and **user\_id**, which is a foreign key that refers to the primary key of the **User** model.

The **Events** model has eight columns: **id**, which is the primary key of the table and an auto-incrementing integer; **event\_poster**, which is a binary object that can hold a large amount of data and is nullable; **event\_name**, **event\_venue**, and **event\_description**, which are strings that can hold up to 500, 500, and 2000 characters respectively; **event\_date** and **created\_date**, which are datetime objects with default values set to the current time; **phone\_number** and **email**, which are strings that can hold up to 20 and 150 characters respectively; and **user\_id**, which is a foreign key that refers to the primary key of the **User** model.

The **Careers** model has nine columns: **id**, which is the primary key of the table and an auto-incrementing integer; **job\_title**, **company**, **location**, and **job\_description**, which are strings that can hold up to 100, 100, 100, and 2000 characters respectively; **salary**, which is a string that can hold up to 100 characters; **date\_published** and **deadline\_date**, which are datetime objects with default values set to the current time; **phone\_number** and **email**, which are strings that can hold up to 20 and 150 characters respectively; and **user\_id**, which is a foreign key that refers to the primary key of the **User** model.

**Note**: incase of a change in the model make sure to run the commands

* + *$ flask db migrate*
  + *$flask db upgrade*

This would ensure that the azure database for postgresql (elp-server) is updated with the latest fields