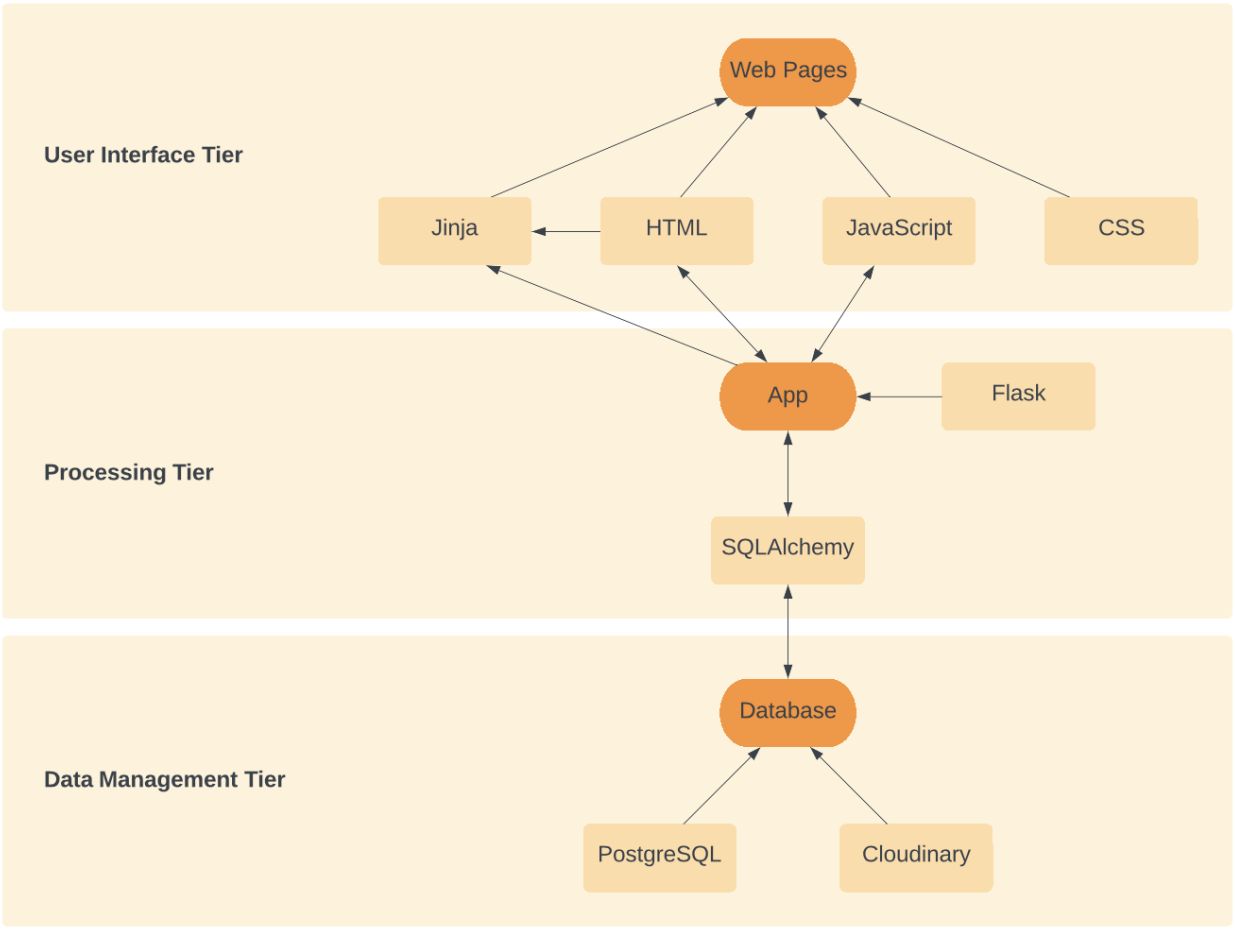
Programmer's Guide

Last Updated May 2, 2022

I. Overview



The GrabAGrad application is divided between three tiers: user interface, processing, and data management. The User Interface Tier is responsible for rendering the web pages and registering user interactions. This tier is built off of Jinja, HTML, JavaScript, and CSS, with the help of Bootstrap. The Processing Tier is responsible for feeding the User Interface Tier the information needed to render the pages and for properly handling user interactions. This tier is also responsible for querying, updating, and otherwise interacting with the database. This tier is coded in Python and is built off of Flask and SQLAlchemy. The Data Management tier consists of the database which stores information about the users of the app. This tier is built using a PostgreSQL database and Cloudinary. In the above diagram, a single-pointed arrow represents the reliance of the pointed-to component on the other component. (e.g. the App is reliant on the Flask framework to function and the Database is built off the PostgreSQL system). A double-pointed arrow represents the transmission of data back and forth between

components (e.g. SQLAlchemy is used to query the Database and the App communicates via Javascript to register user interaction and update data on the web pages).

To promote modularity and ease of maintenance when developing the GrabAGrad application, we organized our code using the MVC (Model, View, Controller) framework. In the MVC framework, the Model interacts with the database. The View consists of the front end user interface that passes user input to the Controller. The Controller accepts input from the View, passes requests to the Model, and retrieves data from the Model to then relay this data in a meaningful way to the View. The User Interface Tier is composed of the View, and the Processing Tier is composed of the Model and the Controller. The Data Management Tier does not fall within the MVC framework, although the Model interacts with this tier. The files making up this framework will be explained in more detail in each tier's description.

Table of Contents

II. User Interface Tier

The pages in our User Interface Tier are built mainly on HTML, Bootstrap, and CSS. These files are stored within the View directory of our application. HTML is used to build the skeleton of each page. Each page on our site has a corresponding HTML page associated with it consisting of the structure, formatting, and referenced CSS for the page. We utilize the free, open-source tool Bootstrap to make sure that our page layouts are responsive to different window sizes, including when the user is on a mobile phone. However, we wrote CSS code to override Bootstrap for certain features and to further customize our pages.

We handle user interaction on our pages using JavaScript. Using JavaScript, we are able to animate certain features on the site, improve responsiveness, and run functions on each page to allow our pages to be dynamic. JavaScript was also used to help require valid inputs for the Create a Profile/Update a Profile page and to confirm certain tasks with the user. Jinja, AJAX, and JQuery are used for dynamic page rendering and responding to user interactions on each page. For instance, AJAX is used to update the search results on the search page every time a user edits the input fields on the search bars.

Diagram 1: Main Pages Available to Undergraduate vs Graduate User

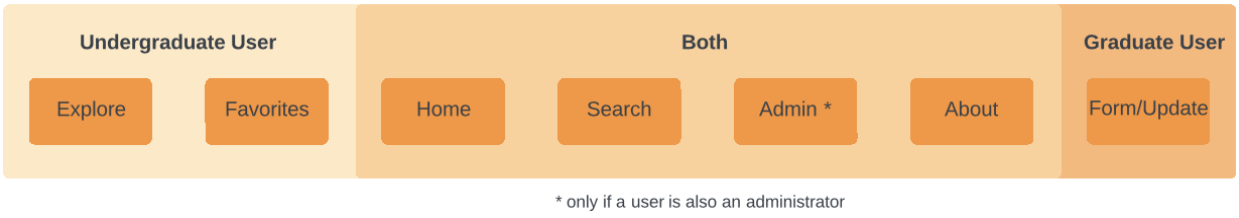
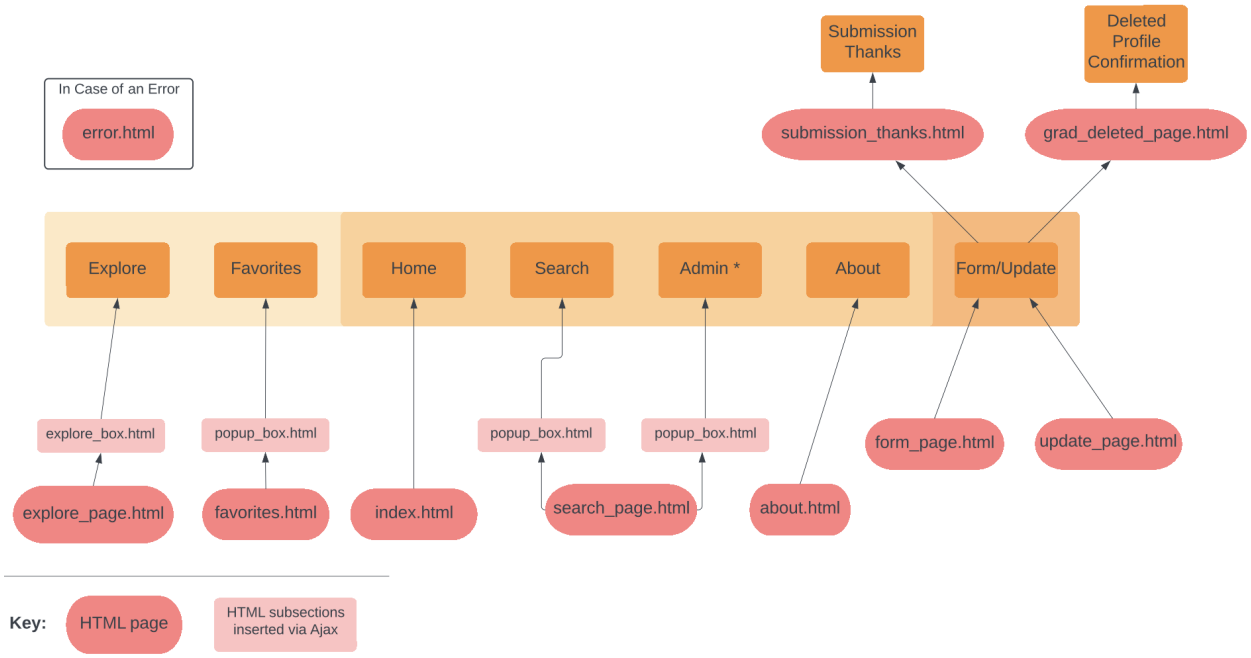


Diagram 2: HTML Files and Their Associated Pages



Pages, Routes, Components, and Details

Each page in our app is accessed by a unique route and is made up of multiple components including different HTML files or smaller templates written in Python.

Name	Route	Components	Details
Home	/	index.html	The first page a user sees, user must click that they are an “Undergraduate” or “Graduate” to launch the site, or “About” to enter the “About” page.
Search	/search_page?user=X	search_page.html, /popup, /load_header_tabs, /filter_grads	Users can browse entire catalog of graduate students organized in a grid, filter graduates by search queries, and click on graduates to see more information about them
Explore	/explore_page	explore_page.html, /explorebox, /load_header_tabs	Undergraduate users can click through graduates one-by-one
Favorites	/favorites_page	favorites.html, /popup, /loadfavorites, /load_header_tabs	Undergraduate users can see the graduates they have favorited

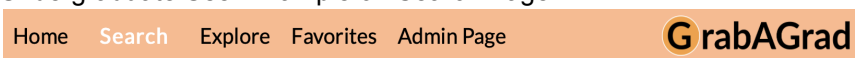
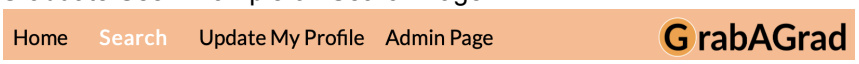
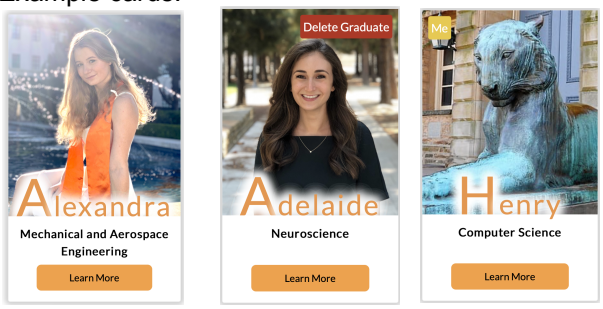
Form/Update	/form	form_page.html, update_page.html, submission_thanks.html	Graduate users can create a profile or update their profile information. Note, in the User's Guide, these pages are called Tell Us About Yourself / Update My Profile
About	/about?user=X	about.html, /load_header_tabs	Users can read about GrabAGrad and how to use the app
Admin	/admin_page?user=X	search_page.html, /popup, /load_header_tabs, /filter_grads	Admin users can delete certain graduates from the database if they choose to
Submission Thanks	/submit	submission_thanks.html	Thanks user for their profile creation, displays the information received
Deleted Profile Confirmation	/self_delete_grad	grad_deleted_page.html	Shows confirmation that the user's profile has been deleted, along with the information deleted

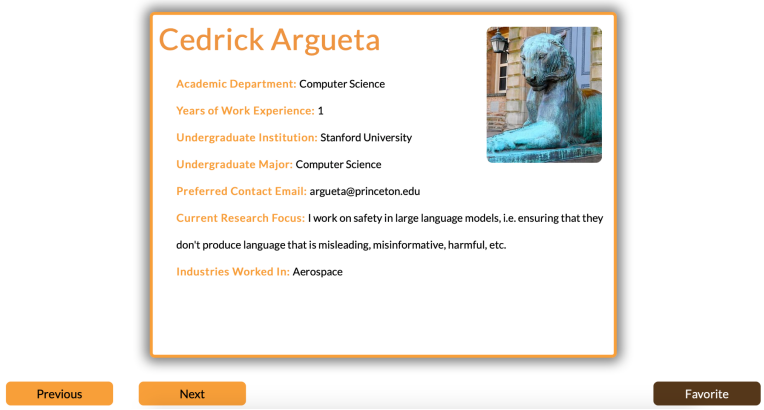
Note: The parameter "user=X" only exists for pages that are accessible by both graduate users and undergraduate users. Whether "user=graduate" or user="undergrad" determines which tabs will be loaded into the menu bar by the request "/load_header_tabs" since certain pages are only accessible to certain users. For instance, the "Form/Update" page is only available to graduate students and shouldn't be an option to click on when an undergraduate user is logged in.

Flask Routes

We utilize Flask requests to actively update certain pages based on user interactions. For instance, the Flask endpoint "/filter_grads" is what updates the search page results when AJAX notices a change in the search bar inputs.

Route and Params	Method	HTML Response
/load_header_tabs ?page=X&user=X	GET	The proper tabs that appear on the top menu bar, with the current page's tab text in white. Params • user=['graduate', 'undergrad'] • page=['search_page', 'favorites', 'about', 'explore', 'admin'] Tabs to show: • Home → Landing page (Both users) • Search → Search page (Both users) • Explore → Explore page (Undergrads only) • Favorites —> Favorites page (Undergrads only) • Admin Page → Admin page (Admins only) • Update My Profile/Create A Profile → Form page (Grads only) • GrabAGrad logo → About page (Both users)

		Undergraduate User Example on Search Page:  Graduate User Example on Search Page: 
/filter_grads ?is_admin=X &name=X &dept=X &industry=X &years_worked=X &un_uni=X &ma_uni=X &sort_by=X &favorites_on=X	GET	The graduate students that match a search query in card format. If on the admin page (is_admin=true), then a “Delete Graduate” button will appear in the top right. If the card represents the user, then a “Me” icon will appear in the top left corner of the card. Params ● is_admin=[‘true’, ‘false’] ○ Whether the current page is the admin page or not ● favorites_on=[‘true’, undefined or ‘false’] ○ Whether the “Only My Favorites” switch is on ● name, dept, industry, years_worked, un_uni, ma_uni, and sort_by are what is in each respective search bar field Example cards: 
/loadfavorites	GET	The graduate students that have been favorited in card format. Similarly to /filter_grads, except with a “Remove” button on the top right corner for the option to remove the graduate from a user’s favorites. Example card: 
/explorebox	GET	The box with graduate information on the Explore page from “explore_box.html”. Also includes the buttons to navigate the explore

?grad=X		page and to favorite/unfavorite a graduate. Params ● grad=[some int] ○ This integer (modulus the number of rows in the graduates database) is used to access the row in the database corresponding to the current graduate on the explore page Example: 
/popup ?id=X &user=X	GET	The box with information about each graduate that pops up whenever a user clicks “Learn More” on a graduate’s card from “popup_box.html” Params: ● id=[netid of the graduate] ● user=[‘undergrad’, ‘graduate’] ○ Need to know this so that Favorite/Unfavorite button doesn’t appear for a graduate user Example: 

Jinja

We utilize Jinja to load in certain parameters to each HTML page that won't change throughout the time that that HTML page is displayed. For instance, template HTML files for features such as the Explore page and Popup boxes take in a different Graduate object as a parameter so that they can turn that object into textual information. This information will not need to be updated within the page. Instead, the page will be reloaded or closed before any information on the template will need to be changed.

HTML File	Jinja Parameters
about.html	• user=['undergrad', 'graduate'] ○ Determines the value of the 'user' parameter used to load in the proper header tabs when requesting "/load_header_tabs?page=X&user=X"
explore_page.html	• grad=[integer for corresponding graduate row in database] ○ Determines the graduate whose information will be loaded in to the explore page

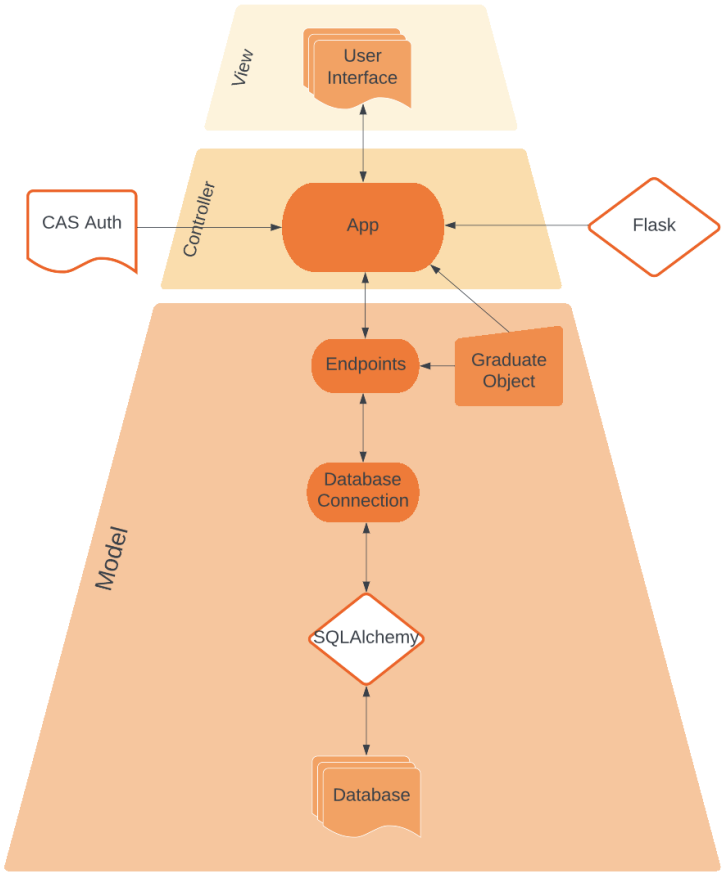
explore_box.html	• grad=[<Graduate object>]
	  ○ The current Graduate being displayed in the explore page
	• favorite=[true, false]
	  ○ Whether or not the current Graduate is favorited (used to set status of favorite/unfavorite toggle button)

form_page.html	• cloud_name=[Cloudinary name]
	  ○ Cloudinary key to access our Cloudinary library
	• grad=[<Graduate object>]
	  ○ The Graduate object with the information already stored for the current user, if user hasn't created an account yet then this will be an empty Graduate object with no information in it
	• dept=[list of departments]
	  ○ List of departments used for suggestions when user types in academic department input field
	• unis=[list of universities]
	  ○ List of universities used for suggestions when user types in undergraduate institution and masters institution fields

popup_box.html	• user=['graduate', 'undergrad']
	  ○ Determines whether or not the Favorite/Unfavorite button will be included in the popup
	• grad=[<Graduate object>]
	  ○ The Graduate object associated with the graduate being clicked, to load in their information
	• favorite=[true, false]
	  ○ Whether or not the selected Graduate is favorited (used to set status of favorite/unfavorite toggle button)

search_page.html	• user=[‘undergrad’, ‘graduate’]
	  ○ Determines the value of the ‘user’ parameter used to load in the proper header tabs when requesting “/load_header_tabs?page=X&user=X”
	• is_admin=[true, false]
	  ○ Whether or not the page is the admin page or not (for admin features to be loaded in as well)
	• grad_names=[list of graduate names]
	  ○ List of all the graduates in the database’s full names for suggestions when the user types in their name in the input search bar
	• depts=[list of departments]
	  ○ List of departments used for suggestions when user types in an academic department in the input search bar
	• un_unis=[list of institutions]
	  ○ List of undergraduate institutions used for suggestions when user types in an undergraduate institution in the input search bar
	• ma_unis=[list of institutions]
	  ○ List of masters institutions used for suggestions when user types in a masters institution in the input search bar

III. Processing Tier



The Processing Tier is built entirely on Flask, SQLAlchemy, and Python. Flask is used to handle inputs from the User Interface Tier and prepare them as inputs to Python functions relying on SQLAlchemy which update, insert, or remove data from our database. The files for the Processing Tier are stored primarily within the Model and Controller directories of our application. The foundational level of this tier, the Model, provides the connection to the PostgreSQL database, formats SQL statements with the open-source Python library SQLAlchemy, and defines the Graduate class for transmitting graduate data between functions. The next level, the Controller, interprets user input received from the View and prepares this input to request corresponding data from the Model. The Controller then receives the returned information from the Model and relays it to the User Interface Tier. Throughout these processes, the user must be verified through CAS authentication. In the diagram above, the three levels of MVC are represented as distinct blocks of a pyramid, with the foundational block ultimately relying on the database. Arrows represent the transmission of data between components.

Python Files

The relevant Python files for this tier are listed below along with a description of the purpose and functionality of each. In the source code, each file has comments underneath each function describing the individual purpose of the function.

Name	Description
flask_app.py	Main file, defines all Flask routes and serves as the connection between the back-end and the front-end by transmitting data back and forth to each

auth.py	Authenticates that each user is a valid Princeton user via the Central Authentication Service
model/**database_connection.py**	Creates the SQLAlchemy engine that provides the connection to the database and handles the execution of all SQL commands
model/**endpoints.py**	Built off of database_connection.py, contains all of the back-end functionality for updating and querying the database
model/**graduate.py**	Contains the Graduate Class for storing all information about a graduate student in one object
model/**universities.py**	Accesses a list of university names using the Hipo API (https://github.com/Hipo/university-domains-list-api) for suggesting universities as the user enters input on the front-end
model/**departments.py**	Returns a list of the departments at Princeton (list was created locally) for suggesting departments as the user enters input on the front-end. Also enables users to type in acronyms for departments in the department search bar.
controller/**search.py**	Takes input from flask_app.py and relays it to endpoints.py for querying specific graduates according to search criteria, returns whether the search was successful along with the results of the search or an error message
controller/pustatus/**pustatus.py**	Determines whether or not the user is an administrator for GrabAGrad, can also determine whether or not the user is a graduate student or undergraduate student based off of their netid (We have not taken advantage of this feature fully yet, however)

flask_app.py Routes

Each route facilitates communication between the front-end and the back-end.

Route and Params	Method	Description
/searchfavorite ?grad=X	POST	Favorites a graduate when the user clicks "Favorite" on that graduate's popup box on the Search page or Favorites page Params: • grad=[netid of the graduate]
/searchunfavorite ?grad=X	POST	Unfavorites a graduate when the user clicks "Unfavorite" on that graduate's popup box on the Search page or Favorites page Params: • grad=[netid of the graduate]
/removefavorite ?id=X	POST	Unfavorites a graduate when the user clicks "Remove" on that graduate's card on the Favorites page

		Params: ● id=[netid of the graduate]
/admin_delete_grad ?id=X	POST	Deletes a graduate when the user clicks "Delete" on that graduate's card on the Admin page Params: ● id=[netid of the graduate]
/self_delete_grad	POST	Deletes the user's information from the database when a graduate user clicks "Delete Profile" on the Update My Profile page
/exploretogglefavorite ?grad=X	POST	Favorites a graduate when the user clicks "Favorite" and unfavorites a graduate when the user clicks "Unfavorite" on the Explore page Params: ● grad=[integer for corresponding graduate row in database]
/submit	POST	Takes graduate user input and inserts it into the database accordingly. If the user has already made an account, then updates the database. If not, then adds new information to the database. Form params: ● first-name, last-name, academic-dept, undergrad-institution, undergrad-major, masters-institution, masters-degree, email, phone-number, years-worked, image_link, research-focus, industry-experience

Two examples of the workflow of the processing tier are described below:

1. **User favorites a graduate on the Search page**

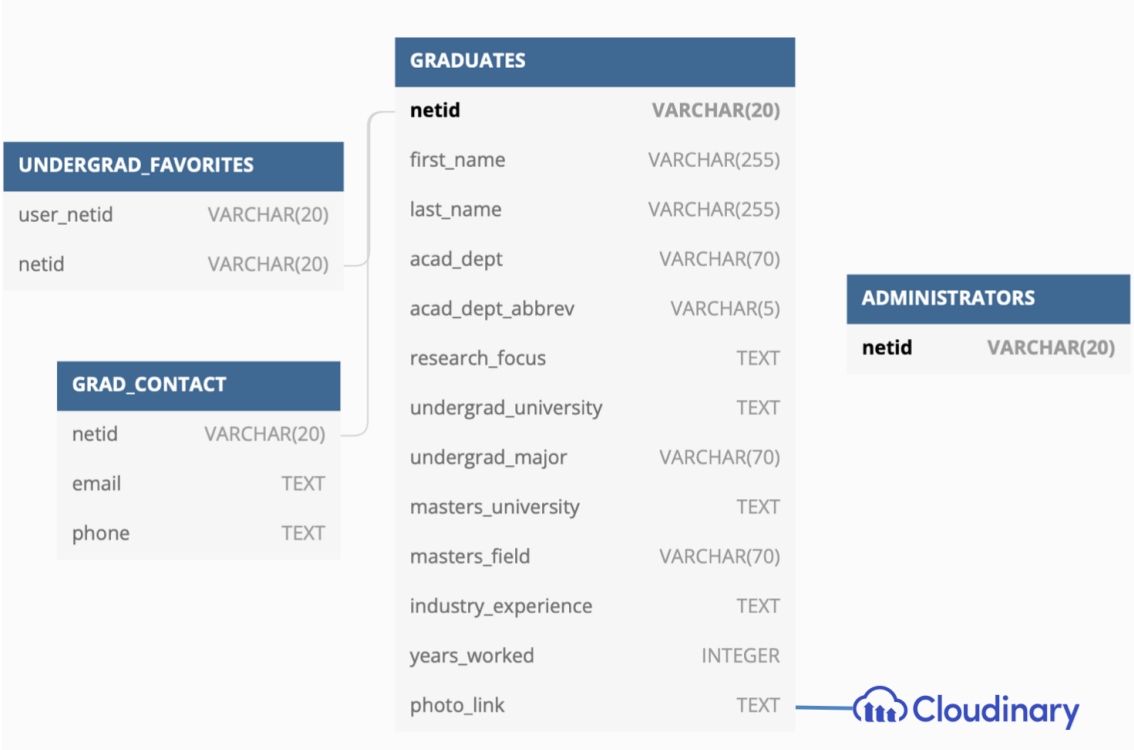
If a user clicks "Favorite" on the popup of a certain graduate student with netid "exampleid" on the Search page, then the User Interface tier will trigger the Flask route "/searchfavorite?grad=exampleid". From here, the app will verify the netid of the user by ensuring that a CAS session is still open. Then, the app will call on a Python function in endpoints.py to add "exampleid" as a favorite associated with the current user's netid in the database. This function in endpoints.py will take in the two netids as parameters and format a SQL statement for updating the database accordingly. It will then run a function in database_connection.py to execute this SQL statement using the created SQLAlchemy engine. Following this process, the User Interface tier will reload the pop-up of the chosen graduate student and verify that the graduate student has been successfully added as a favorite for the current user, updating the pop-up "Favorite" button to say "Unfavorite" if the favoriting was indeed successful.

2. **Graduate user updating their information on the Update My Profile page**

If a Graduate user fills out the required information on the Update My Profile form page, the User Interface tier will trigger the Flask route “/submit” to start the process of updating the database according to the submitted form. The code in flask_app.py will read in the parameters of the form request and prepare them for the database. Then, flask_app.py will call on an update function in endpoints.py, passing in the parameters needed to be updated. This function will then format a series of SQL statements to update the correct rows in the database before calling on database_connection.py to execute these statements. After this process has terminated, Flask will reroute the site to land on the Submission Thanks page, inputting the associated Jinja parameters that will then render on the User Interface side of the information that the user just submitted.

IV. Data Management Tier

GrabAGrad data is stored in a PostgreSQL database hosted by Heroku Postgres which we built and set up with the help of pgAdmin4. In the database, we have 4 tables. Three of our tables are relational while one is stand-alone. Besides this textual data, GrabAGrad also stores images in a Cloudinary library. Our database schema is shown below:



The graduates, undergrad_favorites, and grad_contact tables contain the core data of our app. The graduates and grad_contact tables are linked by primary key “netid” in each. This value corresponds to the CAS verified netid associated with a graduate user after they have created a profile. In the graduates table, each row corresponds to an individual graduate’s information. In the grad_contact table, each row corresponds to contact information (email and phone number) for a graduate. Each user’s inputted information is hence tagged to their netid in our database. For the undergrad_favorites table, when an undergraduate user favorites a graduate student, a new row consisting of this undergraduate user’s netid and the corresponding favorited graduate student’s netid will be created. The administrators table consists of one column with all the netids of the designated administrators of our app.

Description of Columns

Each column contains unique information about each user.

Table	Column	Description
graduates	netid	The netid of the graduate
	first_name	First name
	last_name	Last name
	acad_dept	The Princeton academic department the graduate is studying in
	acad_dept_abbrev	The acronym associated with the academic department, if an acronym exists
	research_focus	A description of research the graduate is involved in
	undergrad_university	The undergraduate university the graduate attended
	undergrad_major	A graduate’s undergraduate major
	masters_university	The university the graduate got a Master’s degree from
	masters_field	The field in which the graduate got a Master’s degree
	industry_experience	A description of industry experience the graduate has
	years_worked	Number of years worked in industry after college
	photo_link	A link to a graduate’s photo saved on Cloudinary
grad_contact	netid	The netid of the graduate
	email	An email associated with the graduate

	phone	A phone number associated with the graduate
undergrad_favorites	user_netid	The netid of an undergraduate who has favorited a graduate
	netid	The netid of a graduate that has been favorited
administrators	netid	A user who has administrative privileges on our site

V. Design Challenges

Attempting to keep track of user status

One of the most significant issues that we realized midway through our project was how we were going to keep track of whether or not the current user was a graduate student or an undergraduate student as they navigated around different pages. As soon as a user clicked “Undergraduate” or “Graduate” on the landing page, we wanted to remember that decision. The main reason for keeping track of this is the fact that the pages available to undergraduate users are different than the pages available to graduate users. For instance, undergraduate students shouldn’t see the “Create my Profile” page and graduate users shouldn’t see the “Explore” or “Favorites” pages. Even the pages that are available to both have slight differences. For instance, on the Search page, undergraduates have the option to “Favorite” graduates while graduate students do not. Originally, we used the sloppy solution of storing a global variable in the Flask app which would keep track. However, Heroku doesn’t support global variables. Therefore, we had to switch to passing in the user’s statuses as parameters for certain Flask routes that could be accessed by both types of users. As a result, routes such as ‘/about’, ‘/admin_page’, and ‘search_page’ require a ‘user’ parameter which can be set to ‘undergrad’ or ‘graduate’. Then, on each page we use Jinja to display the appropriate tabs in the header menu and to turn on certain features that are only accessible to each user.

Making the application aesthetically pleasing on all screen sizes

Because of all the functionality placed into the application, along with all the specific search fields and form inputs, it became a bit difficult at times to properly display all the application’s features on mobile layouts, when there was so much information to display, but such a small screen size to display it on. In order to mediate this issue, JavaScript was utilized to rearrange the contents of each page based on the size of the screen as well as responsive font sizes (in terms of the viewport width) to enable the text on each page to change dynamically based on the screen size.

Restricting Flask endpoints that edit our database

A significant issue with our app was that originally we designed all of our Flask endpoints to trigger with a GET method. This inadvertently allowed users to type in to the URL bar on a browser a route such as /admin_delete_grad to delete a graduate from the database without actually being an administrator. To avoid this security problem, we changed all of the Flask endpoints that would insert, update, or delete data from the database to instead only be activated with the POST method. This restricts users from being able to corrupt the database by typing in routes in the search bar and instead an error message will pop up stating “Method Not Allowed”. However, this still allows the AJAX code in our Javascript to call on these routes as long as we changed the method in which it did so.

VI. Sources

We found StackOverflow, GeeksForGeeks, and W3Schools to be particularly helpful throughout our project. We also referenced the Cloudinary (https://cloudinary.com/documentation), Bootstrap (https://getbootstrap.com/docs/5.1/getting-started/introduction/), and JQuery autocomplete (https://jqueryui.com/autocomplete/) documentation files. In addition, we relied heavily on COS333 example code from lectures, our past assignments in the class, and the occasional visit to Professor Dondero within office hours.

In composing the Programmer’s Guide, the User’s Guide, the Project Evaluation, and the Product Evaluation, we referred to TigerSnatch and TigerThrift’s documentation to gain inspiration on how to style our documents, such as including our project’s logo in the top corner of the page, and including a title page for our documents.

TigerThrift documentation: https://github.com/PrincetonUSG/TigerThrift/tree/main/docs

TigerSnatch documentation: https://github.com/PrincetonUSG/TigerSnatch/tree/main/docs

VII. GitHub Repository

https://github.com/HKnoll42/GrabAGrad