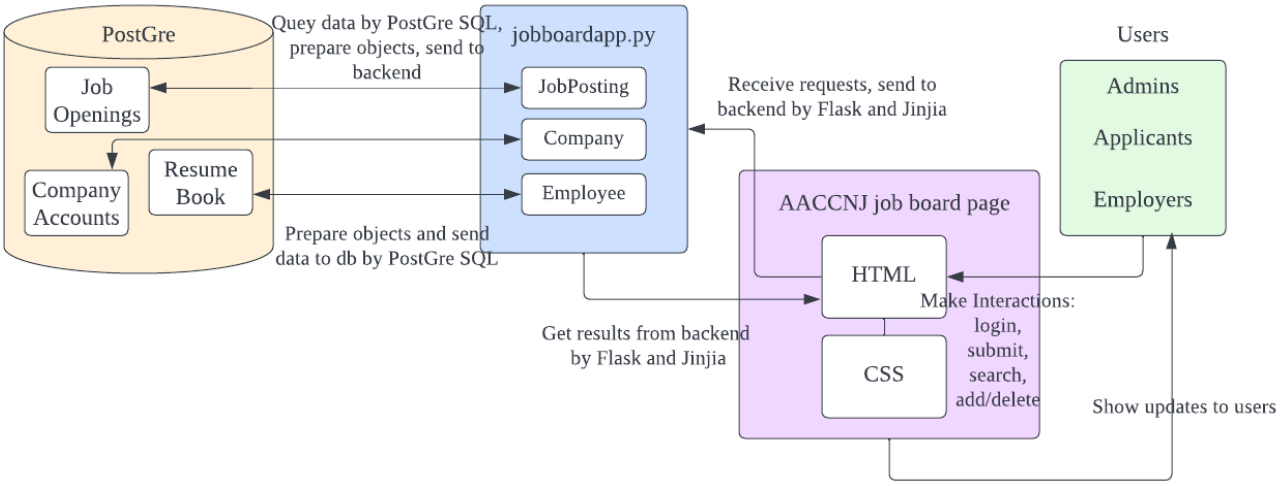
COS333 Final Project: AACCNJ Job Board

Tanushree Banerjee*, Chris Pan*, Haoyuan He*, Yutong Shen*
* denotes equal contribution

Programmer's Guide

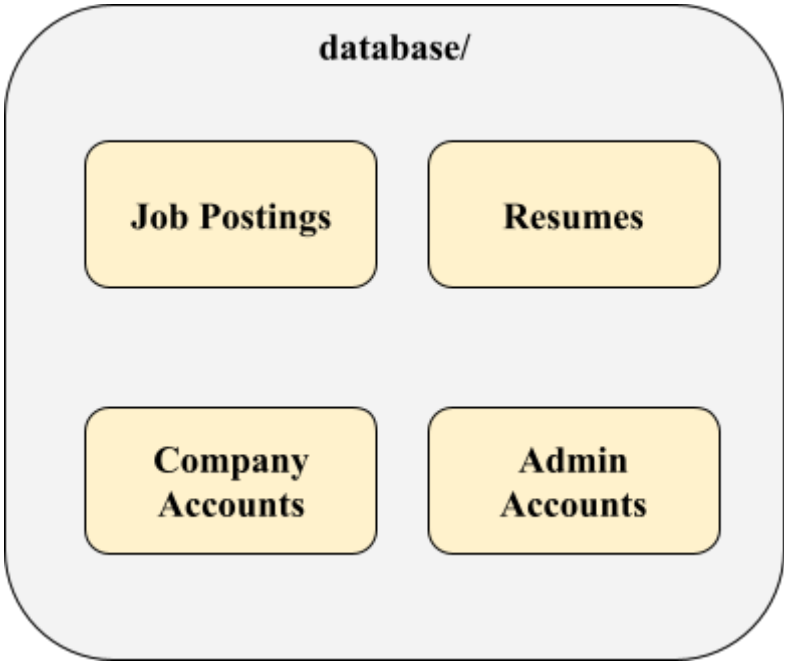
Top-level components of our system



1. The database code, contained in the database subfolder of our project directory. This code handles all the PostGreSQL queries of our codebase. We use PostGre with Elephant SQL in this part of the codebase. In addition to the SQL code, all of the classes defining our fundamental data types. In the MVC model this would correspond to the Model component of our codebase. The backend calls functions defined in the database directory to interact with the SQL database. This is shown in yellow in the above diagram.

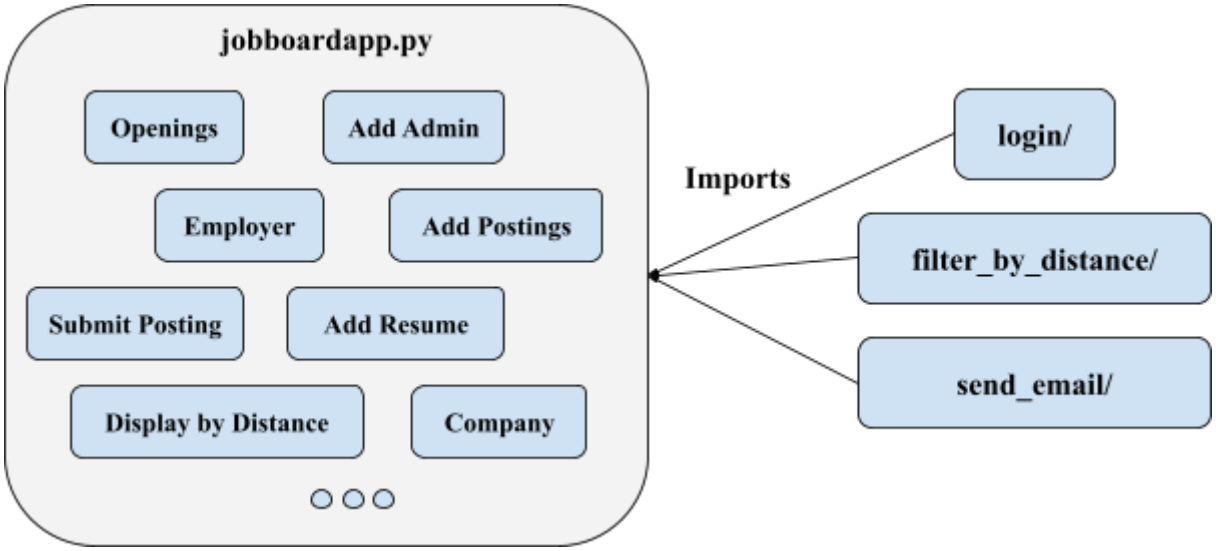
2. The second component is the backend. The majority of the backend code is stored in the jobboardapp.py file in the codebase. The backend is written in the Flask format with Python, using Jinja2 templates (which are defined in more detail below). In addition to the code in jobboardapp.py there are helper functions located in external folders. For instance, helper functions for the filtering by distance feature are in the filter_by_distance directory. Helper functions related to login are in the login directory, and helper functions and sending emails are in the send_email directory. In the MVC framework this would correspond to the Controller part of the codebase. This is shown in blue in our diagram.
3. The last component of our codebase is the front-end. All front-end components are located in the templates directory in our codebase. The frontend is rendered in HTML, styled with CSS, and uses embedded Javascript to make it responsive. We rely heavily on flask-tables, and in particular this repository. In the MVC framework this would be the View component that handles the appearances of our website. In our diagram this is the purple area.

Intermediate-level components of our system



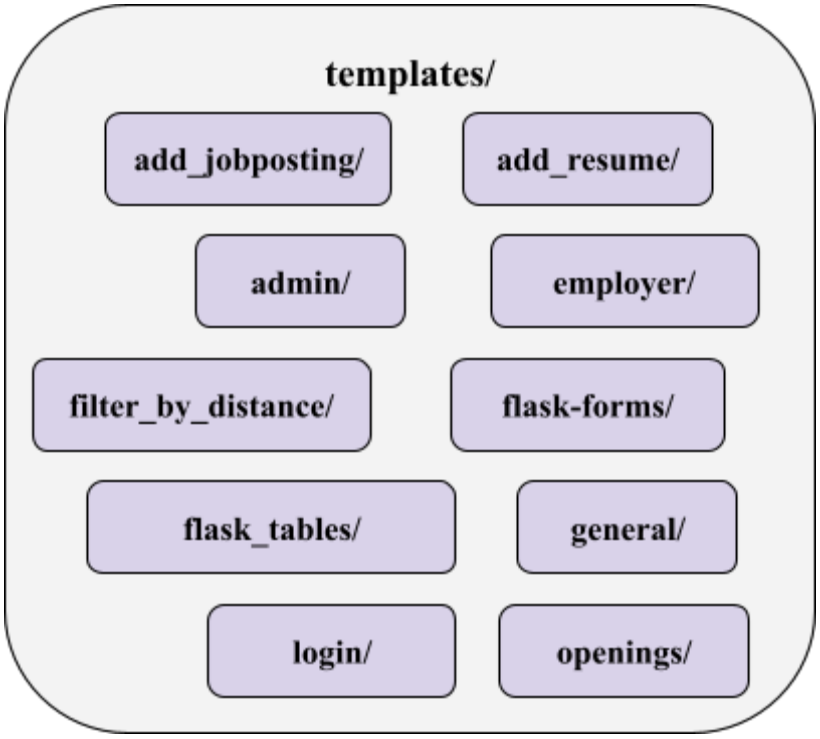
The database section consists of four main files. In our schema, we rely on four databases, shown above, that store fundamentally different data types. Given this we found it natural to divide our codebase in this way as well. Within the database folder, there are four files each containing code related to one of the four database schemas. In each file, we define the python class used to represent that data type, as well as any functions relating to that data type.

For instance: If you need to access a helper function to delete an admin account, you would find “delete_admin_account(email)” in database/admin_accounts.py.



The backend of our website is mostly contained in jobboardapp.py. This file contains a python method for every URL subdirectory of our website. Any GET/POST request has a corresponding method in this file, using the Flask API with Jinja templates. When accessing the backend code for a specific URL, you can find “@app.route(“/subdirectory” …” within the jobboardapp.py file.

This main controller files relies heavily on imports. In particular, imports from the database directory are used to access the SQL database, as well as imports from any feature specific helper methods, such as google login, sending emails, or filtering by distance. As the diagram illustrates, separate helper methods are in their own subdirectories, so that the jobboardapp.py file doesn’t contain too much extra code.



The front-end of the website is contained in the templates directory in the codebase. In this directory, there are various sub-directories named appropriately for the clusters of templates that are useful to them. The general subdirectory within templates contains html files that are used in other html files, (headers, footers, navigation bars etc.). The HTML files in this directory use the Jinja2 template structure, so feel free to use Jinja2 syntax.

Some of the HTML files contain CSS used to render the appearance of the front-end. Some of the HTML files also contain embedded Javascript used to make the page more responsive. A notable example of this is the openings/openings_table.html file which contains extensive Javascript code to make the table very responsive and searchable.

Database schema

Table 1: Resume book profiles

1. id(str): Unique applicant id
2. first_name(str): Applicant's first name
3. last_name(str): Applicant's last name
4. skills(list[str]): Applicant's skills
5. resume_link(str): Url to resume
6. profile_stat(str):Profile status

Table 2: Job postings

1. id(str): Unique job id
2. title(str): Job title
3. role_name(str): Name of the role
4. company_name(str): Company's name
5. location(str): Job's location
6. zip_code(str): Zip code of the job location
7. hours_per_week(int): Work hour / week
8. work_style(style): In-person/Hybrid/Remote
9. app_link(str): Url to apply
10. description(str): Job description
11. summary(str): Summary of job description
12. skills(list[str]): Expected skills
13. post_stat(str):Post status
14. post_date(date): Post date

Table 3: Company accounts

1. id(str): Unique company id
2. comp_name(str): Company's name
3. com_url(str): Url to company web
4. email(str): Email of contact
5. logo(str): url to company logo
6. status(str): Registration status

Table 4: Admin accounts

1. first_name (str): Admin First Name
2. last_name (str): Admin Last Name
3. email (str): Admin Email

Our database consists of four separate schemas for each of the fundamental data-types we used. We will describe them in depth below:

1. **Resume Book Profiles:** This database schema is used to represent an applicant's resume. We store the first and last name, as well as the skills and resume link. The applicant id is a randomly generated UUID that ensures that two applicants of the same name are not treated as equal. The profile_stat field tracks whether administrators have approved the applicant resume or not.
2. **Job Postings:** This database tracks a job posting entry made by a company. Many of the fields contain information about the job posting. The zip_code field is used in the filter_by_distance feature to locate jobs nearest to a query location. Similarly to the resume table, the id is a unique identifier used to make sure two jobs of the same title aren't treated as equal. Post_stat is similarly used to track whether administrators have approved this job or not.
3. **Company Accounts:** We use this database to store accounts created by companies. Since we use google login, we don't need to store passwords, but we still store the email, URL and name of the company. We also store a logo URL to display their logo whenever appropriate. Lastly, we keep track of the registration status, to indicate whether or not the account is approved by administrators or not. If their account is not approved by administrators, they are not allowed to view company account-sensitive pages, such as applicant resumes.
4. **Admin Accounts:** Our last database stores accounts for administrators. We only store the first and last names, as well as admin emails. Note that unlike previous data schemas, the admin table does not have a status, since there is no such thing as an

unapproved admin account. Only administrators can add other administrators, so there is no need for additional admin approval at this time.

Interesting design problems we encountered

1. **Too many database files:** We initially used so many database query files that it was difficult to keep track of what code was in which file. The code used to store admin accounts and company accounts were in the same file, however the code used to interact with job postings was split into three separate files. It made searching through the codebase very difficult..
    a. We solved this problem by realizing a simple solution was to cluster all the code for one data-type into a self-contained file, and creating one file for each data-type. This fits in the design philosophy of strong cohesion, weak coupling, since each module is now very self-contained.
2. **Where to put helper functions:** As we added new functionality into the website, we originally added many helper functions in the jobboardapp.py file. This resulted in a file where half the methods were flask endpoints and the other half were unrelated helper functions. We tried clustering helper functions at the top of the file, but we still weren't happy with how cluttered the backend was.
    a. We realized that we could use the principle of high modularity to pull the helper functions out into separate files. We created additional folders in our directory to house our helper functions, and we grouped them by functionality. Again there was a very weak coupling between helper folders and strong cohesion within a helper folder.
3. **How to organize the MVP functionality as well as stretch goal functionality in each of the three interfaces, and what tabs to show each functionality on:** Deciding which tabs would be the homepage, and what sequence of screens each user would see in each possible use case and scenario (unapproved employer or admin accounts, job details, location-based search, etc.) was challenging.
    a. We took inspiration for this from existing websites that serve similar functionality such as indeed.com, as well as inputs and insight from our external organization (AACCNJ) contact, Ms. Gates.

Adding New Features

Adding a new data table / schema:

1. To create the data-table, add a new python script in the database/dummy_db_creation file mimicking the format of database/dummy_db/create_job_posting.py.
2. Run the python script to create the database in the ElephantSQL backend.
3. Add a file in the database directory, titled appropriately.
4. Add a class that represents the data schema appropriately. While it is not strictly necessary, it is helpful to name the fields of the data in the same format as the dataframe.
5. Add any helper functions necessary in this same file.
6. In the back-end, import the necessary components from the file you just created, and use them in jobboardapp.py.

Adding a new page:

1. A new page requires a new flask endpoint. You can add this by defining a new method in jobboardapp.py. You must also add @app.route(directory…) above the method declaration.
2. This method must return a flask response. You can use flask.render_template and flask.make_response to do so. Use the Jinja template convention in render_template.
3. To define any templates, create html files in templates/. If your page is very related to an existing subdirectory of templates/ you can create your files in that subdirectory. If not, feel free to make a new subdirectory.
4. Within the template you can define subtemplates, scripts and CSS styling.

Adding API keys:

1. Sometimes you want to include new API keys for new APIs you want to implement.
2. Best practice is to define this in an environment variable. Make sure to not add API keys to the repository!
3. You can use os.environ[KEY_NAME] to access the key in your code.
4. Make sure you also add the key to the render settings Environment Variables section.

Adding a new feature / API

1. You would not want to add helper methods and API code in the jobboardapp.py backend file, since this would clutter the file unnecessarily. (We want to maintain that all methods in jobboardapp.py are flask endpoints.)

2. Instead, create a new appropriately titled directory within the repository.
3. In this repository, create any python files you might need, containing all helper methods or data type definitions.
4. Import this file in jobboardapp.py and call the methods you defined in the external directory you just made.