Final Documentation of *InternSHare*

Billy Yi
ly1387@nyu.edu

Daisy Huynh
anh422@nyu.edu

Yumeng Lu
yl7002@nyu.edu

May 20, 2022

Contents

1 Description

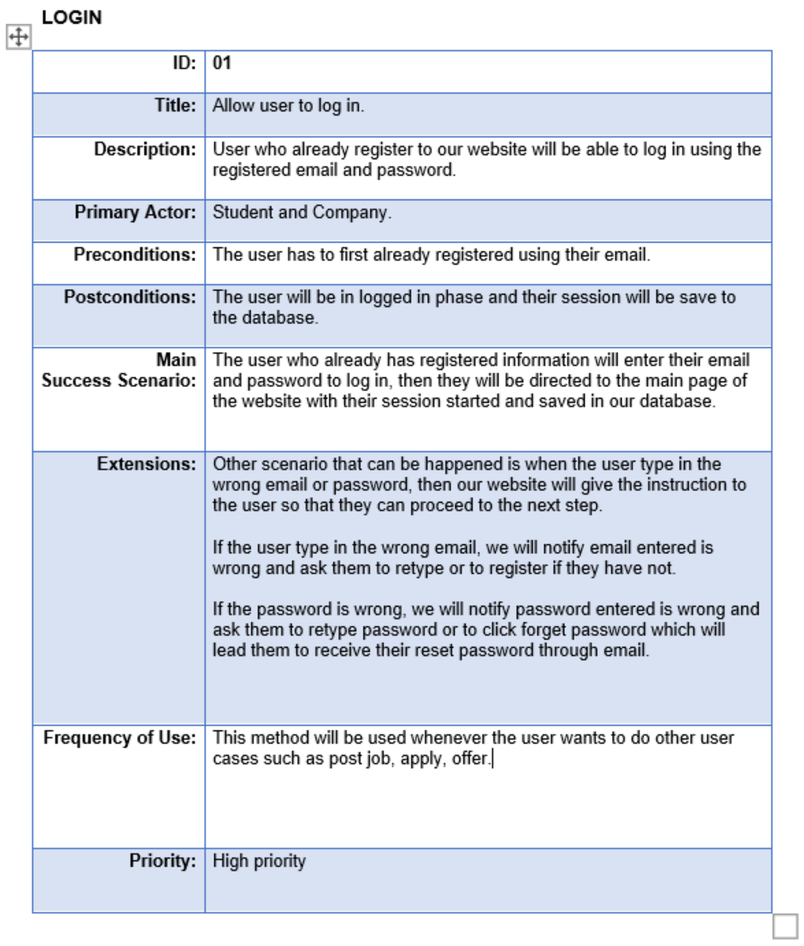
InternSHare is a web system that serves the purpose of finding jobs or internships for NYUSH students and serves the companies, NYUSH students to find the candidates or successors for the jobs that they want to offer. Students can register or log in if they are already a member of the system. On the home page, customized job positions will be recommended to students. Students can also search for their desired jobs using hashtags, filters or direct search and then apply. Meanwhile, students can post a job post with job requirements, job title and job description included. They can view their posted job's candidates and choose to accept or reject their applications. Students can also upload their CV or use the default template as their profile page. Furthermore, they can adjust the status of their profile page to determine if it can be viewed by others. Similarly, companies can register or log in if they are already a member of the system. They can post a job post and decide whether to accept a student's job application. Both company user and NYUSH student user can start a general post, where users can discuss anything related to job hunting, internship experience, course evaluation e.t.c.

2 Process

Our team has followed the process of Extreme Programming (XP) software development process.The reason is that our team has only small amount of members (3 members in total) and that the project do not need exact precision when implementing, in addition, there are many great key practices of XP development process that we think would benefit us during our coding process. Our process for iterative development is that we break down our *InternSHare* project by individual user cases such as `search job`,`login`,`apply`, e.t.c. As the timeline that was given by the Professor where we have a progress meeting every 2 weeks, we therefore have planned to complete one to two user cases per iteration and thus set up a managable timeline in order to finish our project on time. For testing process, our group has followed the white box testing method because we believe that through white box testing method, we can better evaluate our code structures, design and the working flow of the website. From this, we can further improve the design of the project and improve usability for users. For the last month of the working time, we have decided to do collaborative process using pair programming which in our case is ”triple” programming. Eventhough there indeed are difficulty in trying to make use of collaborative coding since we are in different timezones, however, every member has managed to put in their effort and time to do ”triple” programming through Zoom meeting 2 times every week. The description above are the process that our team *InternSHare* has used throughout our project.

3 Requirements & Specification

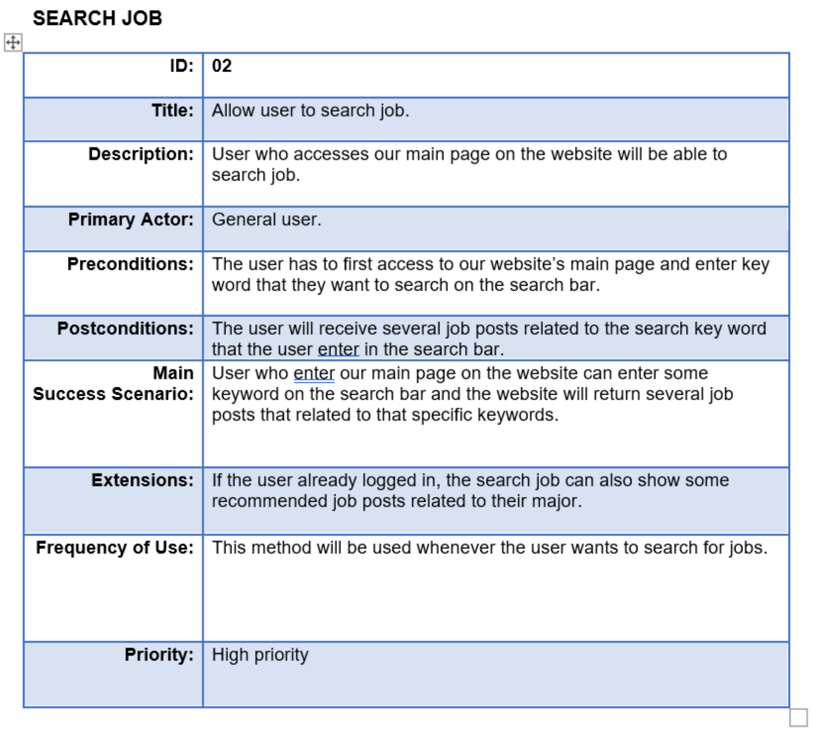
Below are the main use cases in our *InternSHare* project.



LOGIN

ID:	**01**
Title:	Allow user to log in.
Description:	User who already register to our website will be able to log in using the registered email and password.
Primary Actor:	Student and Company.
Preconditions:	The user has to first already registered using their email.
Postconditions:	The user will be in logged in phase and their session will be save to the database.
Main Success Scenario:	The user who already has registered information will enter their email and password to log in, then they will be directed to the main page of the website with their session started and saved in our database.
Extensions:	Other scenario that can be happened is when the user type in the wrong email or password, then our website will give the instruction to the user so that they can proceed to the next step. If the user type in the wrong email, we will notify email entered is wrong and ask them to retype or to register if they have not. If the password is wrong, we will notify password entered is wrong and ask them to retype password or to click forget password which will lead them to receive their reset password through email.
Frequency of Use:	This method will be used whenever the user wants to do other user cases such as post job, apply, offer.
Priority:	High priority

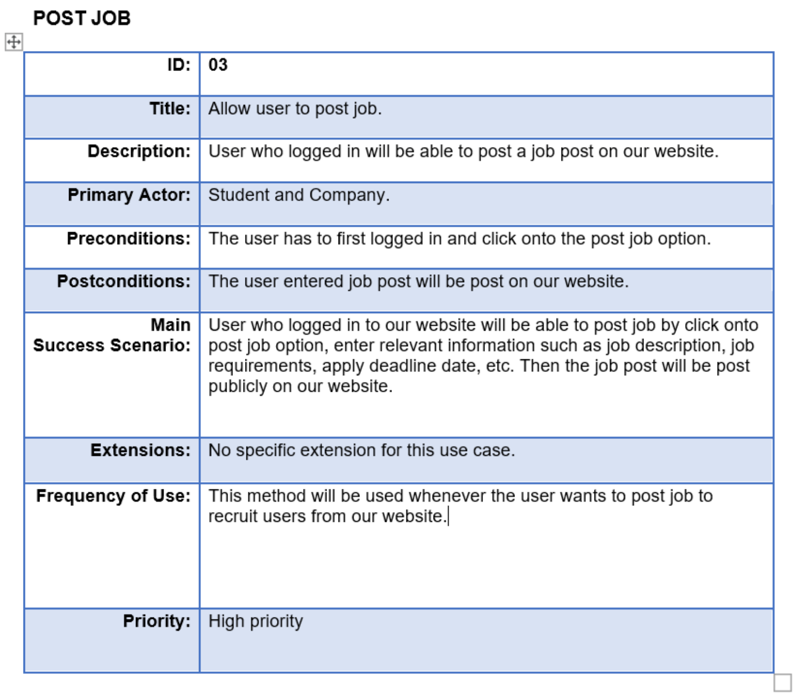
Figure 1: Login use case



SEARCH JOB

ID:	**02**
Title:	Allow user to search job.
Description:	User who accesses our main page on the website will be able to search job.
Primary Actor:	General user.
Preconditions:	The user has to first access to our website's main page and enter key word that they want to search on the search bar.
Postconditions:	The user will receive several job posts related to the search key word that the user enter in the search bar.
Main Success Scenario:	User who enter our main page on the website can enter some keyword on the search bar and the website will return several job posts that related to that specific keywords.
Extensions:	If the user already logged in, the search job can also show some recommended job posts related to their major.
Frequency of Use:	This method will be used whenever the user wants to search for jobs.
Priority:	High priority

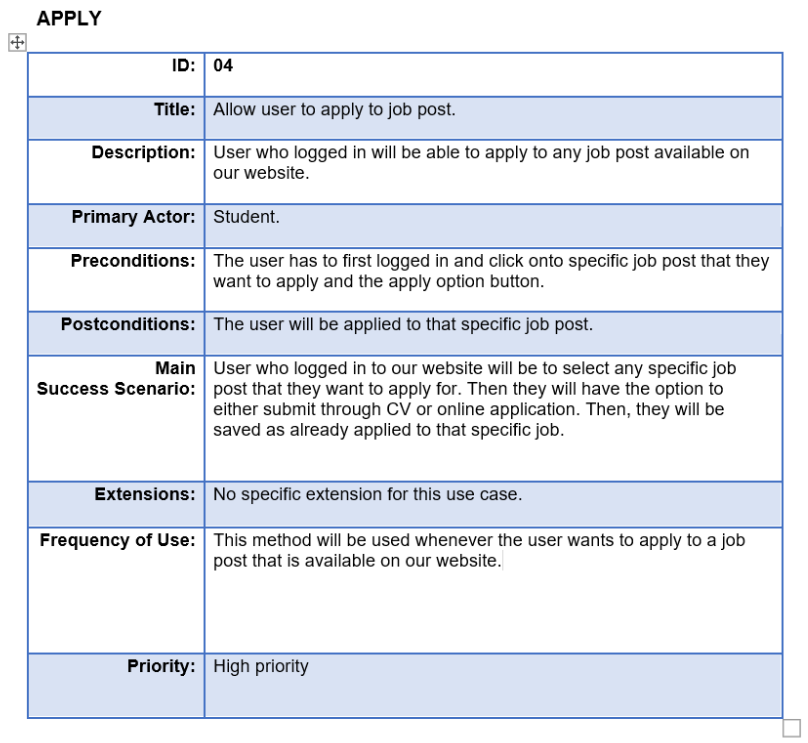
Figure 2: Search job use case



POST JOB

ID:	**03**
Title:	Allow user to post job.
Description:	User who logged in will be able to post a job post on our website.
Primary Actor:	Student and Company.
Preconditions:	The user has to first logged in and click onto the post job option.
Postconditions:	The user entered job post will be post on our website.
Main Success Scenario:	User who logged in to our website will be able to post job by click onto post job option, enter relevant information such as job description, job requirements, apply deadline date, etc. Then the job post will be post publicly on our website.
Extensions:	No specific extension for this use case.
Frequency of Use:	This method will be used whenever the user wants to post job to recruit users from our website.
Priority:	High priority

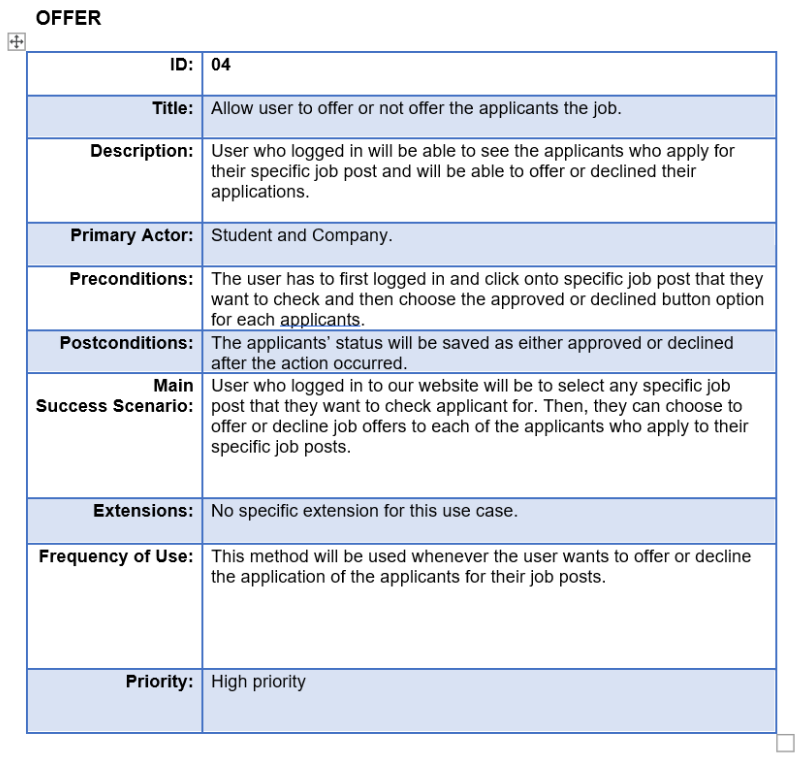
Figure 3: Post job use case



APPLY

ID:	**04**
Title:	Allow user to apply to job post.
Description:	User who logged in will be able to apply to any job post available on our website.
Primary Actor:	Student.
Preconditions:	The user has to first logged in and click onto specific job post that they want to apply and the apply option button.
Postconditions:	The user will be applied to that specific job post.
Main Success Scenario:	User who logged in to our website will be to select any specific job post that they want to apply for. Then they will have the option to either submit through CV or online application. Then, they will be saved as already applied to that specific job.
Extensions:	No specific extension for this use case.
Frequency of Use:	This method will be used whenever the user wants to apply to a job post that is available on our website.
Priority:	High priority

Figure 4: Apply use case



OFFER

ID:	**04**
Title:	Allow user to offer or not offer the applicants the job.
Description:	User who logged in will be able to see the applicants who apply for their specific job post and will be able to offer or declined their applications.
Primary Actor:	Student and Company.
Preconditions:	The user has to first logged in and click onto specific job post that they want to check and then choose the approved or declined button option for each applicants.
Postconditions:	The applicants' status will be saved as either approved or declined after the action occurred.
Main Success Scenario:	User who logged in to our website will be to select any specific job post that they want to check applicant for. Then, they can choose to offer or decline job offers to each of the applicants who apply to their specific job posts.
Extensions:	No specific extension for this use case.
Frequency of Use:	This method will be used whenever the user wants to offer or decline the application of the applicants for their job posts.
Priority:	High priority

Figure 5: Offer use case

4 Architecture & Design

InternSHare is a web application, hence the design can be generally split into front-end (Web), server and the database. To use the application, user will have to first enter the URL (Domain) of the *InternSHare* server, the server will then return the `index.html` file and the related CSS and JS file back to the browser. Note that this is the only time that the server returns the whole HTML file. Once the application is initiated at the browser, later requests will only include pure data requests. Data requests, or API requests, will be first parsed by the server, then the server will operate DAOs to fetch or update the corresponding data. The returned data is in JSON format. An overall system design is shown in Figure 6.

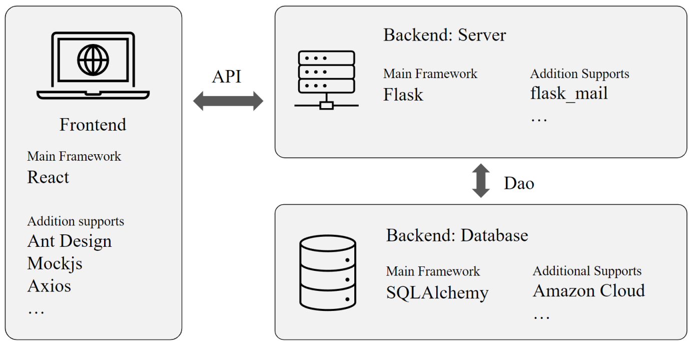


Figure 6: The Overall System design

In the actual implementation, front-end choose React as the main framework, as it is very popular in recent developments. React is famous for high-performance responding and building single page applications. Additionally, front-end also utilize the Ant Design UI module to construct some elements in the web page. Other frameworks and modules including Mockjs and Axios will be further discussed in Section 4.3. The server part choose Flask as the main framework, which is a lightweight server framework with rich functions on the Python platform. The server part also utilized modules including `flask_mail` to handle the mailing tasks. The database side choose SQLAlchemy as the main framework, which is a powerful ORM framework. The database was set on Amazon Cloud.

4.1 Database Design & Classes

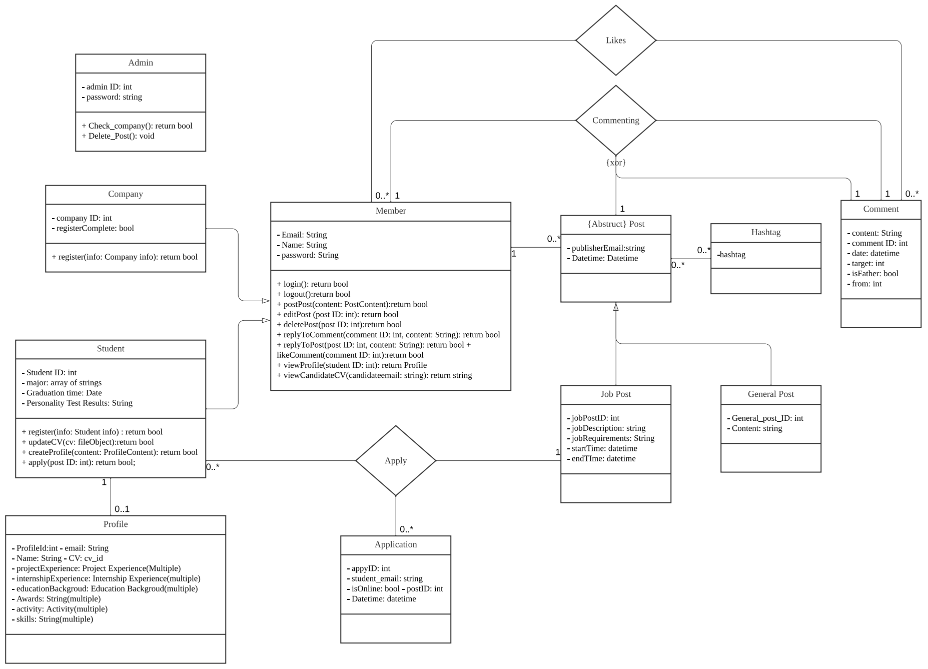

Likes
Commenting
{xor}
Apply
Admin
- admin ID: int
- password: string
+ Check_company(): return bool
+ Delete_Post(): void
Company
- company ID: int
- registerComplete: bool
+ register(info: Company info): return bool
Member
- Email: String
- Name: String
- password: String
+ login(): return bool
+ logout():return bool
+ postPost(content: PostContent):return bool
+ editPost (post ID: int): return bool
+ deletePost(post ID: int):return bool
+ replyToComment(comment ID: int, content: String): return bool
+ replyToPost(post ID: int, content: String): return bool + likeComment(comment ID: int):return bool
+ viewProfile(student ID: int): return Profile
+ viewCandidateCV(candidateemail: string): return string
Student
- Student ID: int
- major: array of strings
- Graduation time: Date
- Personality Test Results: String
+ register(info: Student info) : return bool
+ updateCV(cv: fileObject):return bool
+ createProfile(content: ProfileContent): return bool
+ apply(post ID: int): return bool;
Profile
- ProfileId:int - email: String
- Name: String - CV: cv_id
- projectExperience: Project Experience(Multiple)
- internshipExperience: Internship Experience(multiple)
- educationBackgroud: Education Backgroud(multiple)
- Awards: String(multiple)
- activity: Activity(multiple)
- skills: String(multiple)
Application
- appyID: int
- student_email: string
- isOnline: bool - postID: int
- Datetime: datetime
{Abstruct} Post
- publisherEmail:string
- Datetime: Datetime
Hashtag
-hashtag
Comment
- content: String
- comment ID: int
- date: datetime
- target: int
- isFather: bool
- from: int
Job Post
- jobPostID: int
- jobDescription: string
- jobRequirements: String
- startTime: datetime
- endTIme: datetime
General Post
- General_post_ID: int
- Content: string
0..*
1
0..1
0..*
1
0..*
0..*
0..*
1
1
1
0..*

Figure 7: The Overall Database Class Design

The overall table and class design is shown in Figure 7. Generally, every user in the system is abstracted into `Member`, some generic methods including `login()` and `postPost()` are implemented in the `Member` class. More concretely, there are there types of users in the system, `Student`, `Company`, and `Admin`. Because of time limit, the `Admin` role is not currently implemented. Compared with `Company`, `Student` have additional methods including `apply()` to a job post. An `Application` will be initiated when the apply behavior occurred. `Student` also have a additional association with the `Profile` class. Each `Student` can have at most one `Profile`, and others can only view your profile if any of the following two situations matches: (1) You have set your profile visibility to public. (2) The profile viewer is the publisher of one of your application's target post.

All `Member` in the system are allowed to publish `Post`. `Post` is also an abstraction, because there are two types of `Post` in the system: `Job Post` and `General Post`. `Job Post` is the post that are intended to seek for suitable applicants, and `General Post` can be anything, for instance, internship experiments sharing etc. `Post`s can have `Hashtag`s, which are used for recommendation and filtering. `Student`s are allowed to apply to `Job Post`s, the `Application` will be then created. If a `Student` or a `Company` is the publisher of some `Job Post`, he or she will be able to view all the applicants of that `Job Post`, and also `accept` or `reject` some `Application`s. He or she will also be able to delete or modify the `Post`. Additionally, `Member`s are able to create `Comment`s to `Post` or other `Comment`s. He or she is also able to like a `Comment`, the like information will be preserved in a special `Like` table.

In actual implementation, SQLAlchemy is utilized to serve as the ORM framework. The choose of this framework highly improves the development efficiency, it decoupled the table relationship the with SQL statement, and provides a convenient, DAO-like API interface for the Flask controller.

4.2 API Design

Table 1: API reference

Path	Method	Params	Frontend Page	Flask Blueprint
/api/login	post	email, password	2	loginRegister
/api/login/student/sendemail	post	email	2	loginRegister
/api/login/student/verify	get	email, code	2	loginRegister
/api/login/register/student	post	email, password, confirPW	2	loginRegister
/api/homepage/searchsuggestions	post	content	1.1, 1.2	homepage
/api/search/jobpost	post	filter, pagenumber	1.2	search
/api/homepage/recommendpost/jobs	get		1.1, 1.3	homepage
/api/homepage/applystatus	post		1.1, 1.3.3	homepage
/api/job/detailedinfo	post	jobpost_id, method, publisher	1.5	jobpost
/api/job/getpostcomment	post	jobpostid	1.5	jobpost
/api/job/create/comment	post	content, jobpost_id, target_id	1.5	jobpost
/api/job/update/comment	post	comment_id, new_content	1.5	jobpost
/api/job/delete/comment	post	comment_id	1.5	jobpost
/api/job/like/comment	post	comment_id	1.5	jobpost
/api/job/apply	post	jobpost_id, method, publisher	1.5	jobpost
/api/apply/cancel	post	application_id	1.3.3	apply
/api/profile/get	post	email	1.3.1	profile
/api/profile/update	post	project_experience, . . .	1.3.1	profile
/api/profile/download	post	cv_id	1.3.1	profile
/api/profile/getname	get		1	profile
/api/profile/changesvisibility	get	status	1.3.4	profile
/api/mypost/get	get		1.3.2	mypost
/api/mypost/create	post	post_title, company_name,. . .	1.3.2	mypost
/api/mypost/viewall	post		1.3.2	mypost
/api/mypost/delete	post	job_id	1.3.2	mypost
/api/mypost/update	post	id, post_title, company_name	1.3.2	mypost
/api/mypost/accept/application	post	application_id	1.3.2	accept
/api/mypost/reject/application	post	application_id	1.3.2	reject

The API design follows the RESTful format. All APIs are using either `GET` method or `POST` method, and the route all starts with a prefix `/api`. This is to differentiate the backend (API) route with the frontend (URL) route. Whenever an HTTP request arrives at the Flask server, Flask will first parse the route to see if it starts with the `/api` prefix. If not, the `404Handler` will return the index page `index.html`. This also enables the user to refresh the page that contains sub routes. In real deployment, the `/api` prefix also be a reference for production servers like Apache or Nginx to redirect the request.

The APIs are split into sub categories for better management. Flask Blueprint can help create sub routes and thus form a routing table. In actual implementation, there are 7 sub routes: `/login`, `/homepage`, `/job`, `/search`, `/profile`, `/apply`, `/mypost`. These sub routes also fits well according to the frontend page design.

All API returns are in JSON format. The returns usually contains two fields: `status` and `result`. `status` specifies whether the request have achieve its intention, and why not. `result` usually contains the desired data, if any. If the requested data is a single object, for instance, the profile info, then `result` will be a dictionary (or JavaScript Object). If the requested data is several objects of the same type, then `result` will be a list. The following provides an example of the API returns. (`/api/job/getpostcomment`)

```
{
  "result": [
    {
      "color": "#bf3f84",
      "company_email": null,
      "content": "Change successfully",
      "datetime": "04/30/2022, 18:56:39",
      "descendent": [...],
      "id": 1,
    },
  ],
  "status": "ok"
}
```

All APIs in the system are shown in Table 1. In actual implementation, The Flask framework provides a solid foundation for responding API requests. We are also glad that we have stick to the Python language based server framework and ORM, as it greatly reduces the risk of being stuck with some difficult technical problems.

4.3 Frontend Design

Frontend designs can be split into pages, shown in Figure 8. Each leaf node in this tree is generally a React Component, and are controlled by React Router to mount according to specific URLs. React Router will be listening to the URL when the application is initiated, and mount or unmount specific components once the URL changes. Among all pages, the Main Page is used for all the major operations, while the Login Page handles the login and register behavior. Main Page can be divided into five sub pages, the Home Page provides a integration of search, position feed, check recent apply status functions, which plays a good role in navigation. The Search Page provides detailed search functions, including category filter and sorting (not currently implemented). The Home Page and the Search Page can be accessed from the global Header in the system. The User Page is a place to view other's profile, the Post Page is to view the detailed information of a post and all the corresponding comments. Users will be able to apply to a post on the Post page, as well as creating comments or like comments. The User Page and the Post Page can be accessed by clicking usernames or posts respectively, as these two pages need the user's email and the `postId` to render. My page can be access by clicking the avatar after logged in. It's also split into four sub-sub-pages. The profile page is a place to view the current profile info, update online profile, upload or download the CV PDF file. The My Posts Page and My Applies Page function as viewing all the user's posts and applies respectively. The function of updating posts, deleting posts and creating posts can also be accessed from the My Posts Page.

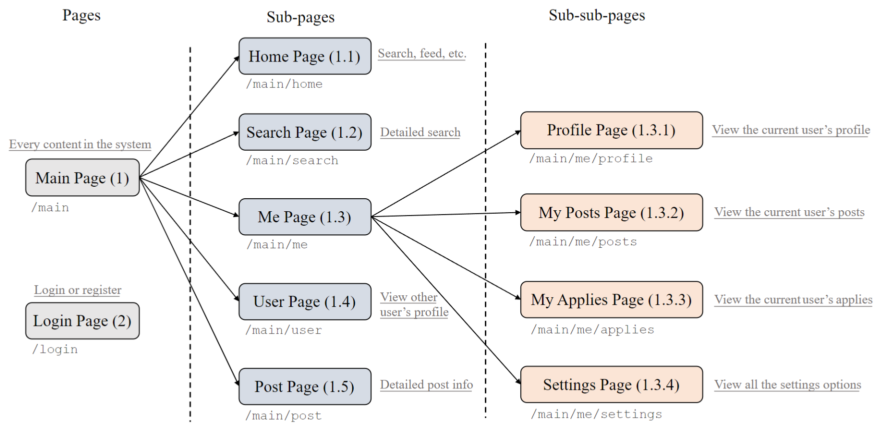


Figure 8: The page organization of the frontend design

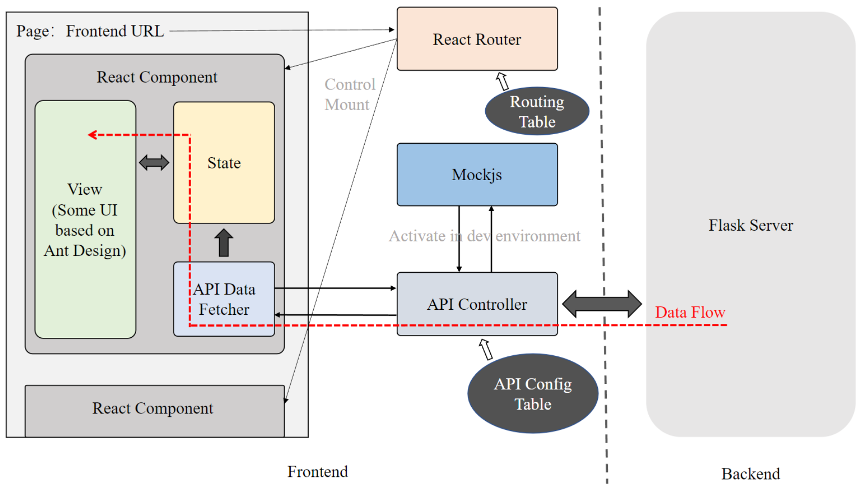


Figure 9: The data flow and the API controller structure in the frontend design

Frontend designed a unified API controller class to handle all the API requests. In development, backend usually does not implement the functions of API in time, but the front end still needs to be developed as planned. Hence, the Mockjs module is used to generate mock data. Each time the React Component is mounted, it will first initiate a new API fetching request, the request is consisting three parts: the key of which API it is going to use, a list of parameters required by this API, and the callback function after the data is returned. The request will be then sent to the API controller, where the controller will search in the API config table for the path, method, and the variable names according to the API key, and then start the HTTP request. In development environment, the HTTP request is responded by Mockjs, while in production it is responded by the actual backend. Once the data is returned, the API controller will call the callback function to update the state of the React Component. This pattern is very similar to the Observer pattern, where each React Component is an observer. A complete data flow is shown in Figure 9.

Figure 10 is the UML diagram of this structure. But notice that this UML does not reflect the actual implementation, because React Components are not following the OOP philosophy. But the UML do reflect the logic of the design.

In actual implementation, the choose of the React framework makes it very easy to separate the frontend and backend development, and provides a solid foundation for the frontend to realize complex functions.

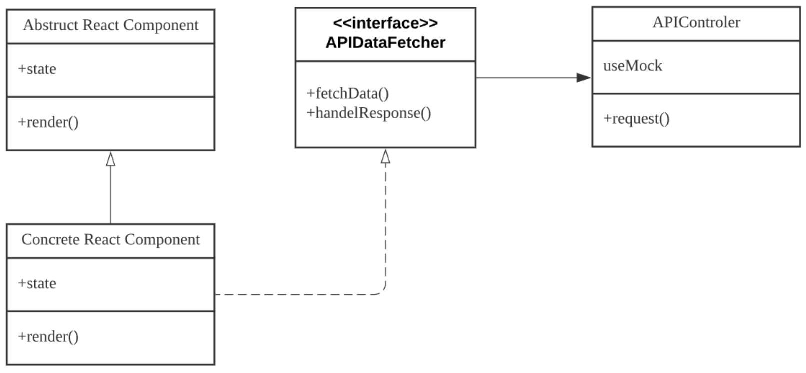


Figure 10: Some visual designs

5 Reflections

- **Frontend reflection - Billy**: In the design and development process, frontend have not encounter any serous problem. However, the inadequate communications with the backend have still cause some problems when doing joint debugging. The inconsistency of some variable names in the APIs have been very troublesome and it cost us huge amount of time to fix it. Although we have use API documentations to communicate, but sometime the backend updated the documents and the frontend don't know, or sometime the backend updated the code without updating the documents. This experience taught me the importance of detailed communication.

- **Database reflection - Daisy**: For me, this is my first time using SQL Alchemy in implementing the database which at first I was really unfamiliar with and the process was a little slow. However, after about 2 iterations, I have started to get used to the flow and able to produce the code quicker. It was difficult at first to translate the query for Lucy who is in charge of the back end code since we have some misunderstanding about what the SQL code should return. After some time, we were able to better cooperate by doing more communications and increase our time of Zoom meetings per week to discuss our process and our schedule. I think the hardest problem for me would be to set up the Cloud Database. This is because we first decide to use Google Cloud Database and already finish implementing all the requirements to connect to the Cloud Database, however due to the fact that we have to upload our codes on GitHub, the Google Cloud Database have locked our account and connections. Thus, I had to spend another week to implement another Cloud Database which is the Amazon AWS and also the one that we are using right now for our project. Overall, I definitely have learned a lot from the class and my teammates. We are always ready to answer any questions or to discuss about an error that we have in our codes which makes the process a lot easier.

- **Backend reflection - Yumeng**: Back-end CRUD operations are not very difficult, except when dealing with the data type of Date, one need to pay special attention to the format of data written to the database. Also, the data extracted from the database should be converted into String or other data type that can be used directly by the front-end. Another problem is that since this is my first time to use API as a backend, I lack effective communication with the frontend in terms of incoming parameters and outgoing data. Sometimes the parameters requested by the back-end are not supported by the front-end, and sometimes the data returned by the back-end is incomplete. Meanwhile, I used flask's blueprint to rearrange the whole API routes when refactoring. These somehow lead to the failure of connecting front-end and back-end due to unmatched content. Next time, I will determine the required APIs with the front-end as early as possible, along with the required parameters and returned data. I will also notify the front-end whenever any change is made to APIs to ensure the smooth development.