Connect Link

Creating a Job Portal Web Application using Python and Django Module

BINUS University International

Algorithm and Programming

Final Project 2024

Juan Felix Kusnadi - 2802536386

Introduction to Connect Link

Connect Link is a job portal web application that is designed to break the gap between recruiters and employees. This platform aims to provide an easy and direct job searching and recruitment process through a friendly interface. Recruiters are able to post job opportunities, manage available jobs, and view candidate resumes. On the other hand, applicants are able to browse through a wide range of job choices from various industries. Hence, Connect Link is built to break the inequality in finding job opportunities.

Problem Analysis

Connect Link is built to address United Nations Sustainable Development Goals (UN SDG) 8, which is Decent Work and Economic Growth. This SDG highlights the importance of promoting a sustained and inclusive economic growth that improves living standards equally for all. Through providing a job portal that combines easy access to job opportunities and effective recruitment process, Connect Link contributes to achieving this SDG by improving employability.

Features

Recruiters	Applicants
Create and Update Company Details	Create and Update Resume Details
Create Job Ads	Apply to Jobs
Manage Jobs	Manage Applications

Use Case Diagram

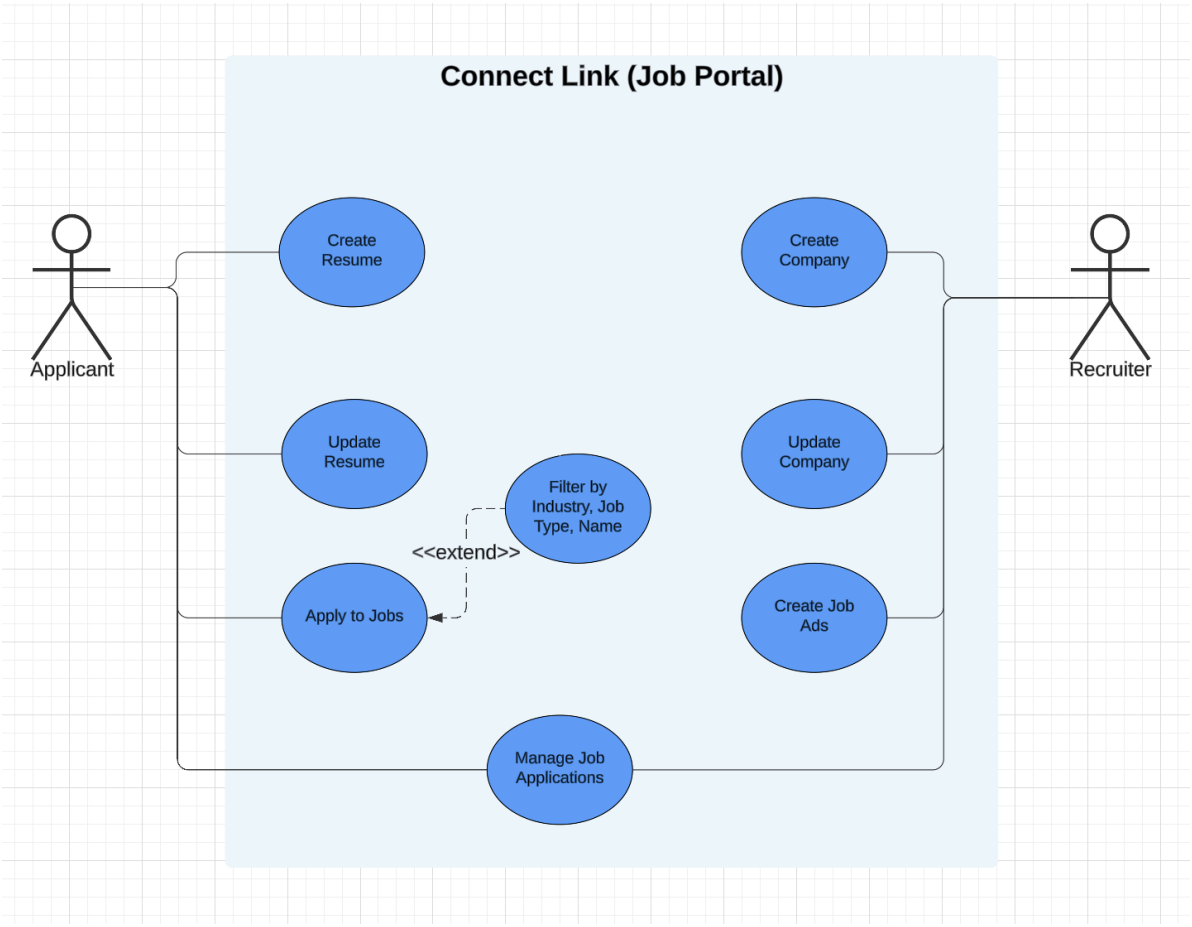


Figure 0. Use Case diagram.

Application Walkthrough

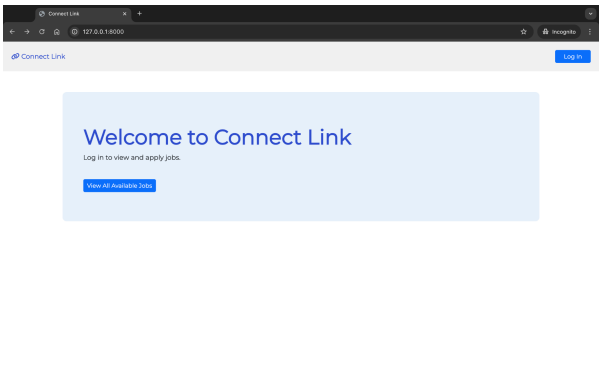


Figure 1. Homepage.

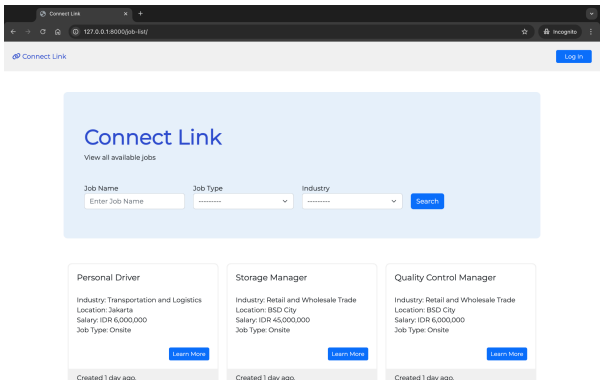


Figure 2. Job List.

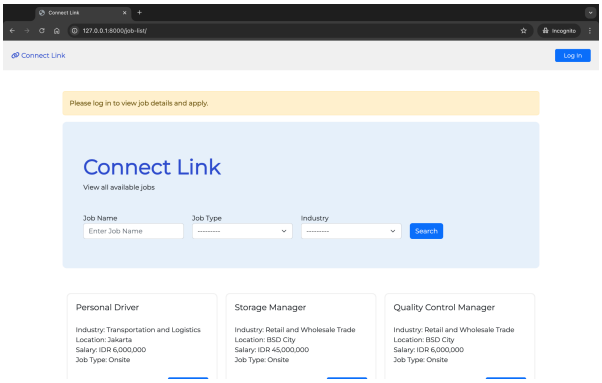


Figure 3. Notification to User.

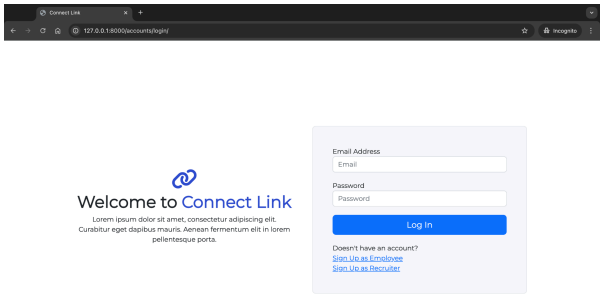


Figure 4. Login Page.

When the user first enters the web application without being authenticated, the main homepage will be displayed with a button “View All Available Jobs” which allows the user to view all currently active job opportunities.[1] The user is then able to view and filter job opportunities based on their name, job type, and industry.[2] However, upon clicking the “Learn More” button which signals the user as a place to view job details, the web app will notify the user to either log in with an existing account or sign up with a new account before proceeding.[3] Upon clicking the “Log In” button on the top right corner, the user will be redirected to the login page.[4]

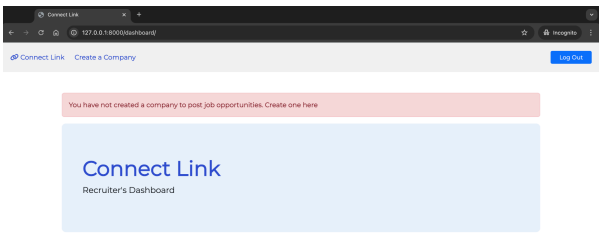


Figure 5. Recruiter's Dashboard.

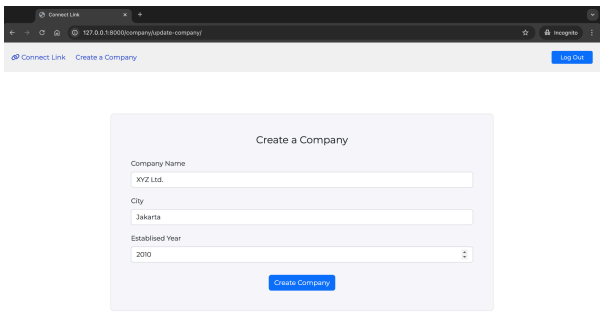


Figure 6. Creating a Company.

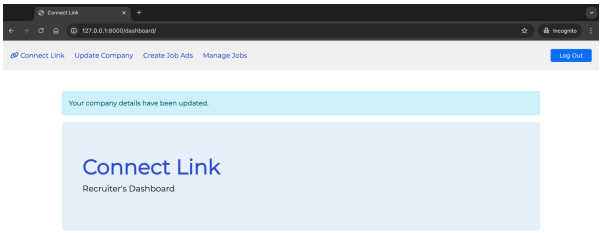


Figure 7. Notification to User.

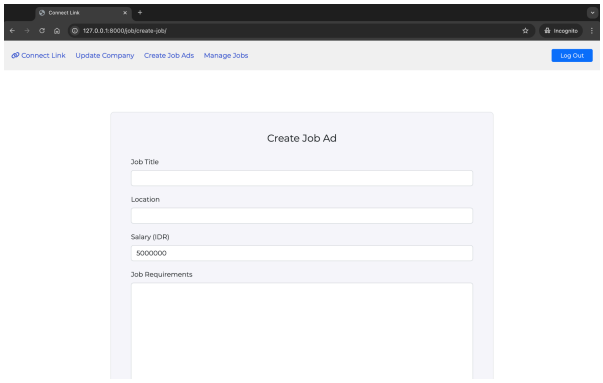


Figure 8. Creating a Job Ad.

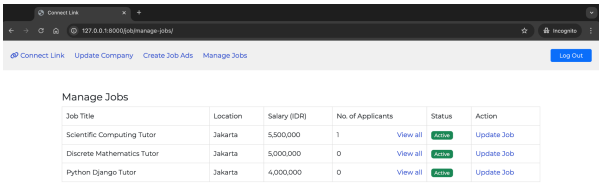


Figure 9. Managing Jobs.

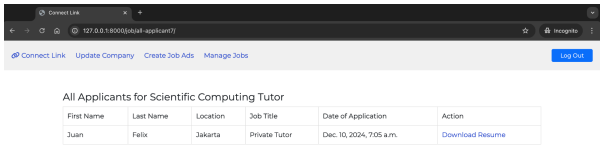


Figure 10. Managing Applicants for a Job.

Upon successfully signing in as a recruiter, the user will be brought to the Recruiter's Dashboard page.[5] In this page, the user will be notified to create a company as soon as possible to create job opportunities.[6] The user can either click the link on the navigation bar or click the text "Create one here." to proceed. The user will then be redirected to the company creation page. On this page, the user is required to enter the name of the company, location, and established date. Upon successfully submitting the form, the user will be redirected back to the Recruiter's Dashboard page with a success notification.[7] Now, more features are opened and can be seen in the navigation bar. This includes updating the company details, creating job ads, as well as managing jobs. Next, the recruiter is now able to create a job ad through inputting job title, location, salary, and descriptions.[8] The recruiter is also able to manage posted job opportunities by clicking the link on the navigation bar.[9] This page also allows the recruiter to update or make changes to a specific job by clicking the "Update Job" link. When the recruiter clicks "View All" for a particular job, they will be redirected to a page where a table containing the details for all applicants will be displayed.[10] The recruiter also has the option to download the applicant's resume by clicking the link under the "Action" table header.

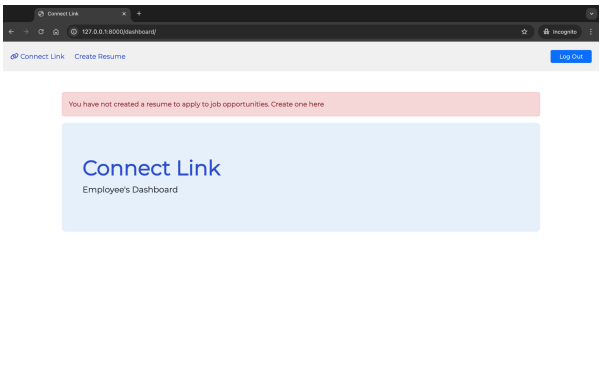


Figure 11. Employee's Dashboard.

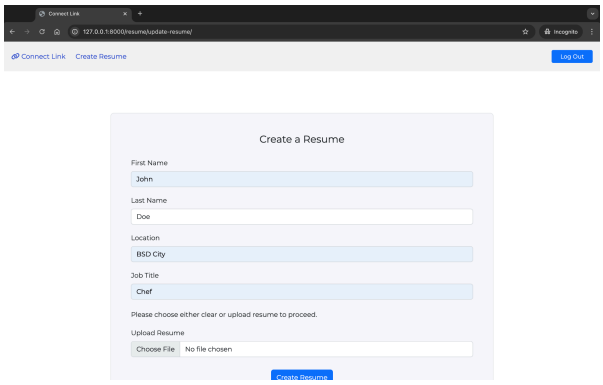


Figure 12. Creating a Resume.

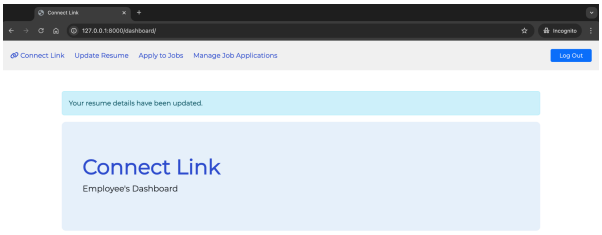


Figure 13. Notification to User.

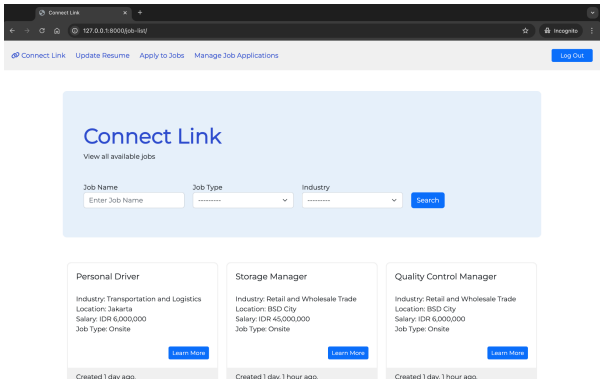


Figure 14. Job List.

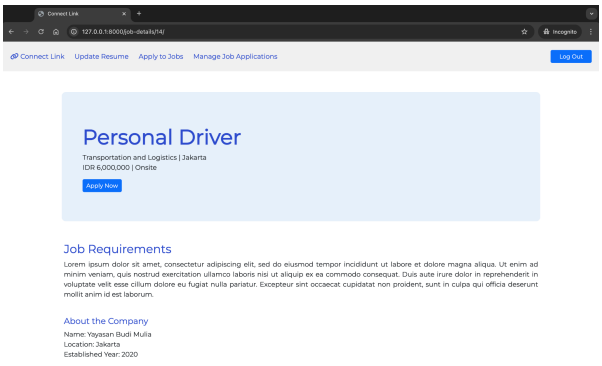


Figure 15. Applying to a Job.

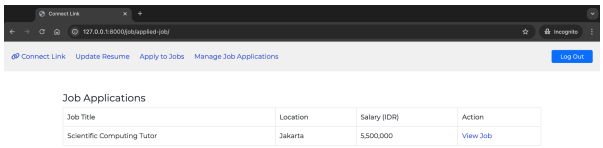


Figure 16. Managing Job Applications.

On the other hand, upon successfully signing in as an employee, the user will be brought to the Employee's Dashboard page.[11] In this page, the user will be notified to create a resume as soon as possible to apply to job opportunities.[12] The user can either click the link on the navigation bar or click the text "Create one here." to proceed. The user will then be redirected to the resume creation page. On this page, the user is required to enter their first name, last name, location, job title, and resume file. Upon successfully submitting the form, the user will be redirected back to the Recruiter's Dashboard page with a success notification.[13] Now, more features are opened and can be seen in the navigation bar. This includes updating resume details, applying to jobs, and managing job applications. Upon clicking the link for applying to jobs, the user will be redirected to the job list page similar to before.[14] However, the user is now able to view job details through clicking the "Learn More" button.[15] To apply to this job, simply click the "Apply" button. The applicant is also able to manage job applications through clicking the "Manage Job Applications" link on the navigation bar.[16]

Program Walkthrough

As this web application requires the use of a database, the building process of Connect Link mainly uses Django as a module to support the creation of most features. Django is chosen as it provides many useful shortcuts, such as login, logout, render, and request, that allow easier and faster platform design. To start a django project, it is necessary to complete the following instructions:

1. Open the terminal.
2. Install Django: *pip3 install django* (use *pip* for Windows / Linux).
3. Start a project: *django-admin startproject project_name.*

4. Navigate to project directory: *cd project_name.*
5. Migrate the database: *python3 manage.py migrate* (use *python* for Windows / Linux).
6. Start the server: *python3 manage.py runserver* (use *python* for Windows / Linux).

Creating a new Django project will create several files by default under the directory, which includes:

manage.py

project_name/

__init__.py

settings.py

urls.py

asgi.py

wsgi.py

manage.py is mainly used to interact with the Django project as a command-line utility. Common commands include runserver, makemigrations, migrate, and startapp. Next, *settings.py* acts as a configuration app under the Django project. It is mainly used to add installed apps to the project. In addition, *urls.py* contains the URL configuration for the Django project. The next step to start coding is to create an app. In Django, an app is simply a way to organize the code such that it can be reused in another Django project. As such, an app must only be created in a way that it only serves a single purpose. This is another benefit provided by Django, allowing users to minimize effort to install new apps whenever starting a new, similar project. To start a

new app, enter *python3 manage.py startapp app_name*. Whenever starting a new app, the following files will be created by default:

app_name/

__init__.py

admin.py

apps.py

migrations/

__init__.py

models.py

tests.py

views.py

Each default file created serves a unique purpose. __init__.py, apps.py, migrations, and tests.py are not commonly used for simple projects, hence they are not being used in this case. Below is the detailed explanation about each file with the example implementation.

1. admin.py

Figure 17. admin.py.

admin.py is mainly used for registering models to the Django admin site, hence allowing modifications to the models from the admin interface. In Figure 17, it is being used to register the Industry class to the admin site. However, it is essential to note that only the superuser is able to create changes to the model. Hence, to create a superuser,

type *python3 manage.py createsuperuser* in the terminal. Credential details will be required in the next step. To open the admin site after successfully creating a superuser, visit http://127.0.0.1:8000/admin/. In this case, I added 9 industry categories under the industry option selector in several forms in the front-end.

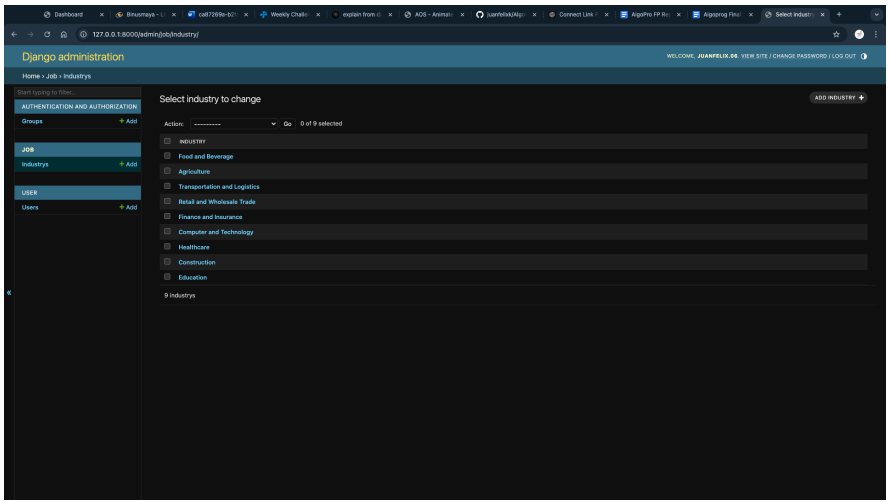


Figure 18. Django admin site.

2. models.py

models.py file is used to create a database schema for the application by creating model classes. In creating Connect Link, several classes made are inheritances of the default class in Django. This includes the classes User, Industry, Job, ApplyJob, Resume, and Company. Some examples are shown below.

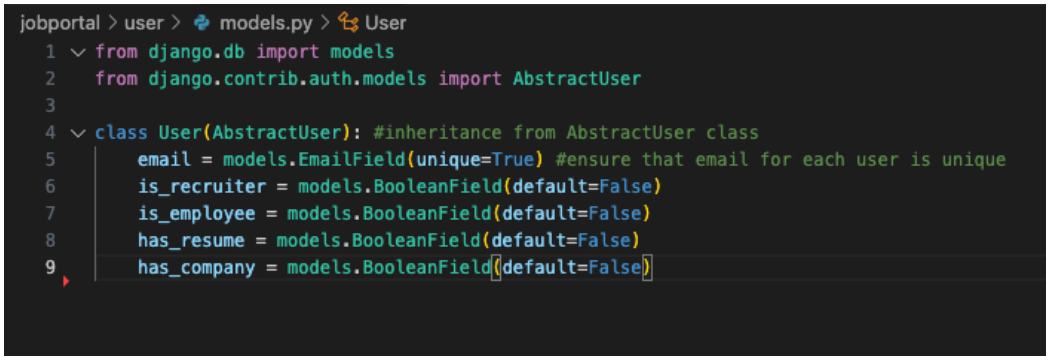


Figure 19. The User class.

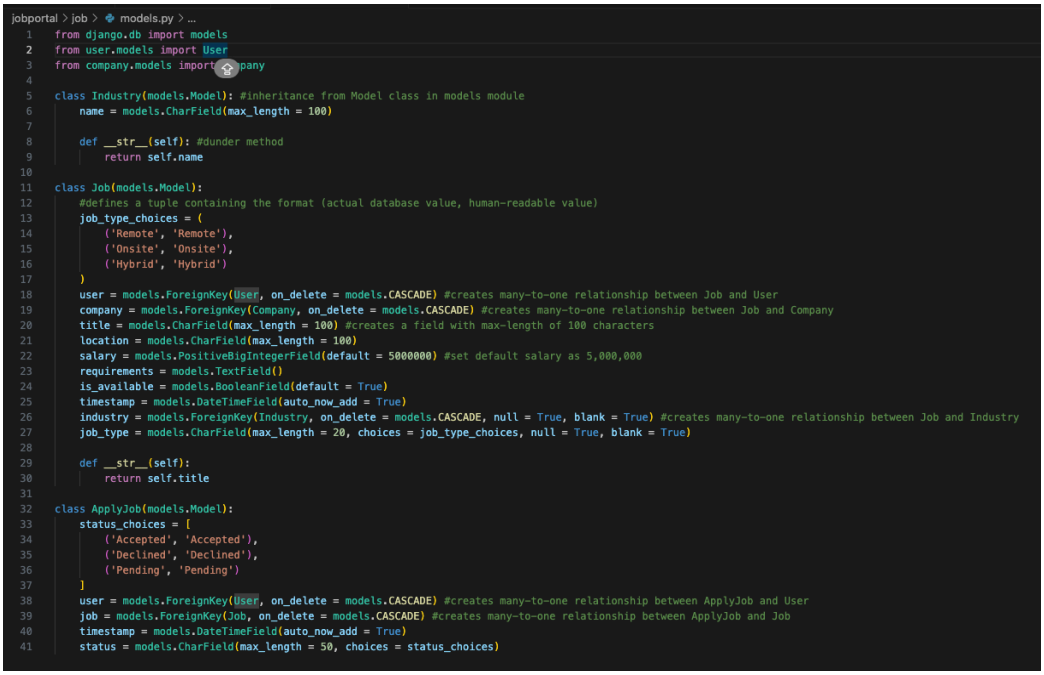
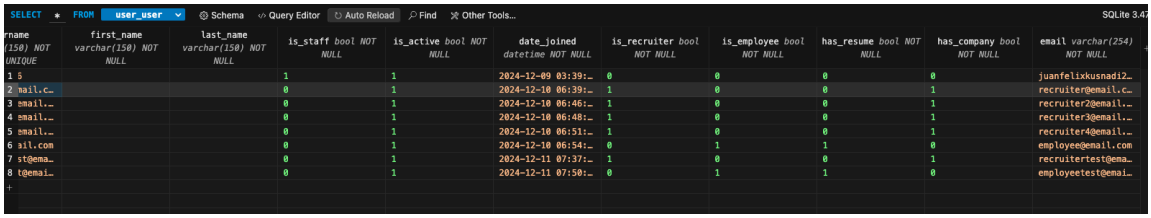


Figure 20. Industry, Job, and ApplyJob class.

As an example, the class User contains 5 different fields, which are email, is_recruiter, is_employee, has_resume, and has_company. These fields, with their respective values are reflected on the database in the form of a table. Figure 21 shows this table.



SELECT * FROM user_user Schema Query Editor Auto Reload Find Other Tools... SQLite 3.47

	rname (150) NOT UNIQUE	first_name varchar(150) NOT NULL	last_name varchar(150) NOT NULL	is_staff bool NOT NULL	is_active bool NOT NULL	date_joined datetime NOT NULL	is_recruiter bool NOT NULL	is_employee bool NOT NULL	has_resume bool NOT NULL	has_company bool NOT NULL	email varchar(254) NOT NULL
1	5			1	1	2024-12-09 03:39:…	0	0	0	0	juanfelixkusnadi2…
2	mail.c…			0	1	2024-12-10 06:39:…	1	0	0	1	recruiter@email.c…
3	email.…			0	1	2024-12-10 06:46:…	1	0	0	1	recruiter2@email.…
4	email.…			0	1	2024-12-10 06:48:…	1	0	0	1	recruiter3@email.…
5	email.…			0	1	2024-12-10 06:51:…	1	0	0	1	recruiter4@email.…
6	ail.com			0	1	2024-12-10 06:54:…	0	1	1	0	employee@email.com
7	st@ema…			0	1	2024-12-11 07:37:…	1	0	0	1	recruitertest@ema…
8	t@emai…			0	1	2024-12-11 07:50:…	0	1	1	0	employeetest@emai…

Figure 21. user_user table.

3. views.py

views.py is a file under an application that contains functions and/or classes that handle HTTPResponse request objects and return responses. As such, views.py is mainly used to return render or redirect function to the user, with the purpose of rendering (displaying) and redirecting (bringing to another page). A further discussion about views.py will be done in the following section.

Algorithm Walkthrough

This walkthrough will summarize the major algorithms used in this project.

```python
# Update Resume Details
def update_resume(request):
    if request.user.is_employee: #check whether user is an employee
        resume = Resume.objects.get(user = request.user) #retrieve resume object of the authenticated user
        if request.method == 'POST': #check whether form is submitted
            form = UpdateResumeForm(request.POST, request.FILES, instance = resume) #retrieve information and file inputted by the user
            if form.is_valid():
                update = form.save(commit = False) #save is form is valid, commit=False allows modifications before saving
                user = User.objects.get(pk = request.user.id)
                user.has_resume = True
                user.save() #updating user object
                update.save()
                messages.info(request, 'Your resume details have been updated.') #notify the user
                return redirect('dashboard') #redirect user to the url path named "dashboard"
            else:
                messages.warning(request, 'An error occured. Please try again later')
        else:
            form = UpdateResumeForm(instance = resume)
            context = {'form': form}
            return render(request, 'resume/update-resume.html', context) #display the webpage when user first enter the page
    else:
        messages.warning(request, 'Request denined to due permission error.')
        return redirect('dashboard') #redirect user to url path named "dashboard" if user is not an employee
```

Figure 22. update_resume function.

Figure 22 shows a function named *update_resume*, whose main purpose is to update an existing resume that has been previously created by the user. This function takes the *request* object as an argument. First, it checks whether the user is an employee since this feature should only be accessible to employees. Next, it displays the webpage to the user when the user first enters the url. When the method is *POST*, meaning that the form is submitted, the function retrieves the data inputted from the user and saves this to the database. It finally returns a redirect, which redirects the user to another web page with a url path *dashboard*.

```
# Applying to a Job
def apply_to_job(request, pk):
    if request.user.is_authenticated and request.user.is_employee:
        job = Job.objects.get(pk = pk)
        if ApplyJob.objects.filter(user = request.user, job = pk).exists(): #check whether a user has applied to a particular job
            messages.warning(request, 'You have applied to this job.') #doesn't allow the same user to apply twice
            return redirect('dashboard')
        else:
            ApplyJob.objects.create(user = request.user, job = job, status = 'Pending')
            messages.info(request, 'You have successfully applied to this job.')
            return redirect('dashboard')
    else:
        messages.info(request, 'Please log in to continue.')
        return redirect('login')
```

Figure 23. apply_to_job function.

Figure 23 shows a function named *apply_to_job*, whose main purpose is to allow applicants to apply to an active job opportunity. This function takes the *request* object and *pk* as arguments. First, the function ensures that the user is an employee and has been authenticated (logged in). Next, it retrieves the job object with the associated pk (primary key) value which uniquely identifies a user. It then checks whether the user has applied to this job, hence preventing the same user from applying twice. If this is valid, it creates an applyjob object and redirects the user to the webpage with a url path *dashboard*.

```python
# Create Job
def create_job(request):
    if request.user.is_recruiter and request.user.has_company: #check whether user is a recruiter and has a company
        if request.method == 'POST': #check whether form is submitted
            form = CreateJobForm(request.POST) #retrieve information inputted by the user
            if form.is_valid():
                job = form.save(commit = False) #save if form is valid, commit=False allows modifications before saving
                job.user = request.user
                job.company = request.user.company
                job.save() #updating job object
                messages.info(request, 'New job has been created.')
                return redirect('dashboard') #redirect user to the url path named "dashboard"
            else:
                messages.warning(request, 'An error occured. Please try again later.')
                return redirect('create-job')
        else:
            form = CreateJobForm()
            context = {'form': form}
            return render(request, 'job/create-job.html', context) #display the webpage when user first enter the page
    else:
        messages.warning(request, 'Request denined to due permission error.')
        return redirect('dashboard') #redirect user to url path named "dashboard" if user is not a recruiter
```

Figure 24. create_job function.

Figure 23 shows a function named *create_job*, whose main purpose is to allow recruiters to create a job opportunity. This function takes the *request* object as an argument. First, the function ensures that the user is a recruiter and has a company. Next, it displays the webpage to the user when the user first enters the url. When the method is *POST*, meaning that the form is submitted, the function retrieves the data inputted from the user and saves this to the database. It finally returns a redirect, which redirects the user to another web page with a url path *dashboard.*

Resources

- Link to GitHub Repository