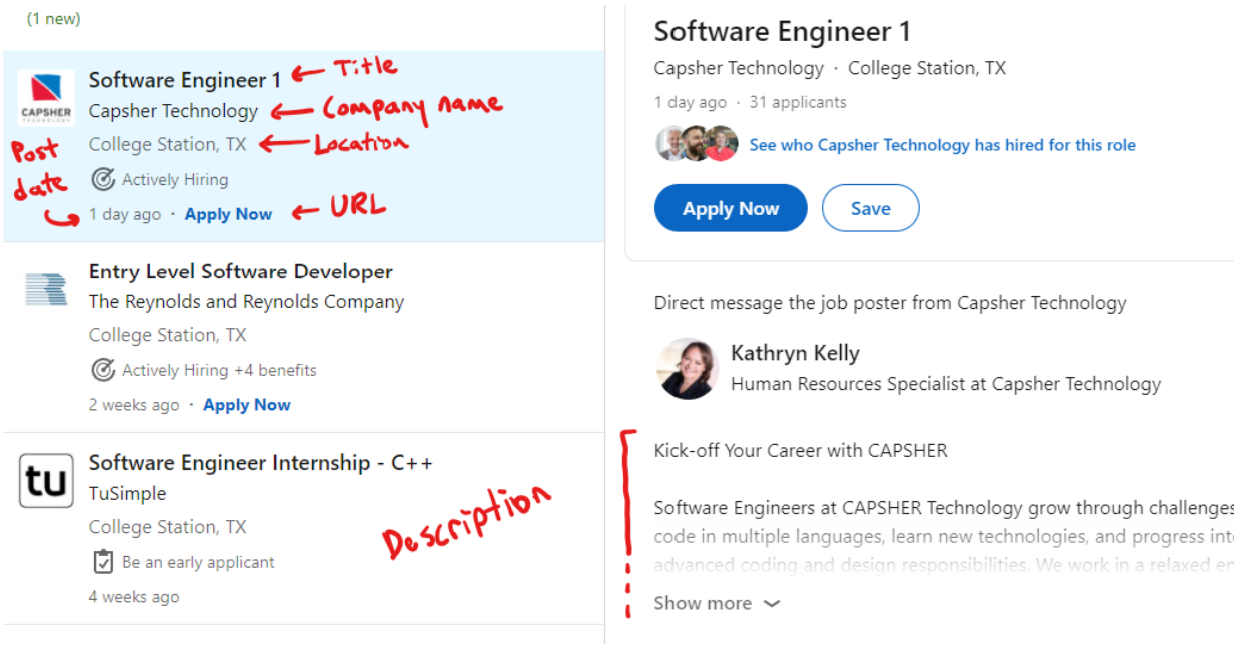
C. Linkedin Web Scraper

C1. Overview and Job Page Details

The implementation of the Linkedin web scraper is done in python3. It mainly relies on the selenium package to navigate the linkedin page and gather elements from the page. The sqlalchemy library is also used to upload the data to the database and make changes to the database.



Each object that we want to grab has to be selected as a css element on the page. Since the Linkedin page is fairly structured, grabbing these elements is fairly simple. The main Linkedin job search page is formatted as shown above, where each job gets an individual card on the left hand panel, and only after being selected does a job's details appear on the right hand panel.

This interactive view on Linkedin is only activated when the window is a certain size, so an important detail is to have the window maximized. Only 25 jobs at a time are displayed in this view, so to load all the jobs from a search, the window must be scrolled to the bottom to load the next 25 jobs. After

doing this several times, a “load more jobs” button will appear, requiring the agent to click on the button to load more jobs. Another important detail to note is that after loading roughly 1000 jobs in a single session, the Linkedin application will not load any more requests for that session (at least for a decent amount of time). This can be problematic, since there is no way to close the Linkedin window and open a new search starting from where the last one left off, since this interactive view is essentially an infinite scroll. So it must be ensured that the agent only gathers ~1000 jobs in a session.

C2. Code Implementation

The Linkedin web scraper is run using “python main.py CHUNK_SIZE" where the chunk size command line argument represents the amount of jobs that will be scraped before writing a chunk of them to the database. For example, a chunk size of 50 for 1000 jobs would mean that every 50 jobs scraped, the scraper would write those 50 to the database, therefore writing in 20 different sessions in total.

The implementation uses environment variables to gather whether the run is a production run and to get the heroku credentials (so they are not stored directly in the code). If the run is a production run, we write to the production database, if it is not we write to a test database.

C2i. Initialization

The first step in the process of the web scraper is to initialize everything. If it is a production run, the postgresql connection string is built from environment variables. Included in the initialization process is a function to connect to the database and run a sql command to delete jobs in the database that were scraped at a date older than an allotted time (this is 2 weeks as of now). This is so that we don’t keep outdated jobs in the database, obviously since we don’t want to recommend jobs that are no longer available to our users. As a caveat, this could be improved upon since not all jobs have the same lifespan after being posted.

The next step in initialization is to start the selenium webdriver. This function takes all of the selenium arguments we want to start the driver with and creates a chromedriver instance using an installed version of chrome. One important argument is to start the window maximized so we get the deterministic result described in (C1). This function returns the driver to now be used.

The last initialization function is responsible for getting the scraper to the point described in (C1) where all the available jobs are loaded on the page in the “infinite scroll” format. First, the driver loads a predetermined url that includes:

- Search for the linkedin jobs page.
- Query for what type of jobs (computer science)
- Query for location (United States)
- Time limit for jobs posted in the past t seconds (86400)

After the page is loaded, the function proceeds to load all of the job cards onto the page. It repeatedly scrolls to the bottom of the page, either waiting for more jobs to load or clicking the “load more jobs” button that appears after loading a certain amount of jobs. This process will end either after all of the jobs are loaded (which is indicated by a certain element on the page) or after a certain threshold is reached, so there is not an infinite loop.

C2ii. Scraping

After the initialization process, job data can then be gathered from the page. First, all of the job cards are stored as a list, which can be conveniently grabbed from the Linkedin page as a list element. Next, we gather the data for a given chunk. Most details that we want are already loaded on the page at this point and can be scraped from the cards:

- Job title
- Employer

- Location
- Posting date
- URL

This is defined as the “metadata” and is scraped in a function that returns lists containing the data for each type.

Next, the most important and intricate part of the design is scraped; the description. Job descriptions are the most important part for the resume matcher since that is the data it uses to compare against the resume data and make recommendations. On the Linkedin page, descriptions are only loaded one card at a time, and when that card is selected. Therefore, to get the description, the driver needs to select a card, wait for the details to load, scrape the details, then proceed to the next card. This process has its own function and takes the longest amount of time of any of the processes.

One interesting detail of the description gathering is that even after selecting a different job card, the old jobs cards description is still loaded on the page until the new one is finished loading, so we must be sure that the new description is loaded before scraping it. This loading process is indeterminate, and must be handled dynamically. One way we improved the process is by clicking away to the previous card and back on the selected card. This process re-prompts the page to load the card details and has proved to be successful to speed up the process time.

To avoid grabbing the previous description, we check the current description on the page against what the previous description was. If they are the same, we wait until they differ before scraping the description. Also, two back-to-back jobs will sometimes have similar descriptions. To avoid getting stuck, we implemented a string similarity matcher to set a threshold when comparing the previous and current description, which has also turned out successful.

If the description is not loaded in a reasonable amount of time, that job description is marked as an error and will be removed at a later step, so the program can proceed. After everything is scraped, a function is run to filter out jobs where the description could not be gathered, or where some other error occurred. Now, this chunk is written to the database. After all of the chunks are written, the program is complete.