Full-Stack Project – Job Listing Web App

Project Overview

Bitbash is hiring full-stack developers and tasking you with developing a full-stack job listing web application over the next few days. This take-home project will assess your full stack skills across the (Flask/Python, SQLAlchemy with PostgreSQL or MySQL, React/JavaScript, and Selenium for web scraping) in a real-world scenario. You will build a job board application that **lists actuarial job postings**, integrating data scraped from the Actuary List website.

Scope:

You'll build a complete web app that shows job listings. It includes three main parts:

1. Create a Flask REST API to manage job listings.
2. Use SQLAlchemy and a PostgreSQL/MySQL database to persist job data.
3. Create a React UI for viewing, adding, deleting, and filtering jobs.
4. Use Selenium to scrape jobs from https://www.actuarylist.com and populate your database.
5. Record a Vimeo demo video explaining and **validating** your implementation.

Note: The submission deadline will be provided by Bitbash's HR or Talent Acquisition team. Submitting earlier than the deadline is considered a **plus** and improves your chances of moving forward. **Late submissions will not be accepted.**

We're not just looking for something that works — we'll also check how clean your code is, how you handle errors, how you explained things, and how nice the UI looks. The main goal is to see how you approach the full project from start to finish, especially when working under limited time.

Allowed Tools/Libraries: You may use any third-party libraries or frameworks to accelerate development (for example, component libraries for UI, form handling utilities, testing frameworks, etc.).Anything that makes you faster and efficient. Make sure, however, that your own coding and architecture are clearly visible. *DevOps tasks (deployment, Docker/containers, CI/CD) are **not** required* – focus on the application code itself. But it's a plus for us.

Project Requirements

1. Backend – Flask API and Database (CRUD Operations)

- **Flask REST API:** Develop a Flask-based RESTful API server for the job listings. Include endpoints to **create**, **retrieve (list)**, **update**, and **delete** job posts. Follow conventional REST patterns (e.g., `GET /jobs`, `POST /jobs`, `PUT/PATCH /jobs/<id>`, `DELETE /jobs/<id>`).

- **Data Model:** Use **SQLAlchemy** with a relational database (PostgreSQL or MySQL preferred) to define a Job model. Each job post should have at least the following fields:
    - `id` (unique primary key)
    - `title` (job title)
    - `company` (company name)
    - `location` (e.g. city and/or country)
    - `posting_date` (when the job was posted – as scraped or provided)
    - `job_type` (e.g. full-time, part-time, contract, internship)
    - `tags` (a list of relevant tags or keywords, e.g. industries or skills like "Life", "Health", "Pricing")
      (You can store tags as a simple comma-separated string or use a related table – your choice. Ensure the additional fields like posting_date, job_type, and tags are incorporated into your schema.)

- **CRUD Functionality:** Implement the logic for all CRUD operations:
    - Clients can create new job entries via the API (e.g., by sending a JSON payload with title, company, etc.).
    - The API should return job data for read requests (single job by ID or list of jobs) in JSON format.
    - Clients can update existing job entries (e.g., to correct or change details).

    - Clients can delete job entries. (Ensure that deleting or updating a non-existent job is handled gracefully with an appropriate error response.)

- **Filtering & Sorting:** Extend the GET/list endpoint to support basic filtering and sorting via **query parameters**. For example:
    - Ability to filter jobs by `job_type` (e.g., `GET /jobs?job_type=Full-time` returns only full-time roles) or by other fields like location or tags (`/jobs?location=London` or `/jobs?tag=Pricing`).
    - Ability to sort the results, for example by date posted (`/jobs?sort=posting_date_desc` for newest first, etc.).
      These parameters should be optional; if not provided, return all jobs in a reasonable default order (e.g., newest first).

- **Validation & Error Handling (Backend)**

  Make sure your API checks incoming data on the server side. Some fields—like **title**, **company**, and **location**—must not be left empty. If a client sends bad or incomplete data, respond with a clear error message and use the correct HTTP status code (like **400 Bad Request**).

  Also, use the right status codes for other responses—for example:

    - **201 Created** when something is successfully added,
    - **404 Not Found** if the requested data doesn't exist.

  Don't let your server crash if something goes wrong. Catch errors and return helpful responses instead of failing silently or breaking. The goal is to make your API reliable, even when things go wrong.

2. Frontend – React App for Job Listings

Framework & Setup

Use **React** to build a single-page frontend. You can start with **Create React App** or any similar setup. The app should connect to your Flask API to fetch and display data.

Job Listings UI

Create a clean, readable view to show job posts from the backend. Each job should appear as a **card** with:

- **Title** (as the main heading)
- **Company name**, **location**, and **job type** (as subtitles)
- **Posting date**
- A few **tags or keywords**

Keep the design simple but professional. Apply basic styling—consistent fonts, spacing, and a light color scheme—to make the layout visually clear.

Responsive Layout

Make sure the app works on both desktop and mobile:

- On **mobile**, stack job cards in a single column.
- On **larger screens**, use a **grid or multi-column layout**.

The goal isn't perfection—just make sure the layout doesn't break and stays user-friendly across devices.

CRUD Interface: Add, Edit, and Delete Job Listings on the Frontend

1. Add New Job

Create a form on the frontend that allows users to add a new job listing. This form should include input fields for:

- **Title**
- **Company**
- **Location**
- **Job Type**

- **Tags** (can be entered as comma-separated values or through a basic input field)

When the form is submitted:

- Send the job data to the backend via the **POST** API.
- Update the frontend by either refreshing the job list or updating the component state to reflect the new job.

2. Edit Existing Job

Ideally, support editing jobs by adding an **Edit** button on each job card. When clicked:

- Open a form pre-filled with the current job details.
- Allow users to update any of the fields (title, company, location, etc.).
- On submission, send the updated data to the backend via the **PUT** or **PATCH** API.
- Reflect changes in the UI after the update is successful.

If full editing is too time-consuming to implement, at least include a basic version that:

- Edits the latest or a hardcoded job entry using the form.
- Demonstrates the ability to trigger an update request and handle the response.

3. Delete Job

Add a **Delete** button on each job card. When clicked:

- Prompt the user for confirmation (using a simple window.confirm is acceptable).
- If confirmed, send a **DELETE** request to the backend with the job ID.
- On success, remove the job from the UI by updating the state or refetching the job list.

Filtering & Sorting (Frontend)

Implement dynamic UI controls to let users filter and sort the job listings easily and intuitively.

Filtering

Add the following inputs to allow users to narrow down job results:

- **Keyword Search Box:**

  A text field that filters jobs by title or company name. As the user types, update the results live or on submit.

- **Job Type Dropdown:**

  A select box with options like “All”, “Full-time”, “Part-time”, etc.

- **Location Dropdown:**

  A list of cities or remote/onsite options for location-based filtering.

- **Tags Multi-select or Checkboxes:**

  Let users select one or more tags (like “React”, “Remote”, “Internship”) to refine results.

Behavior:

- When any filter is changed, the frontend should **send a new request to the backend** with the selected filter parameters.
- If the app fetches all jobs at once, you can filter client-side—but **server-side filtering is preferred** for up-to-date and scalable results.
- Clearly show **which filters are active** (e.g., with chips or summary text) and offer a **Reset Filters** button to clear them all.

Sorting

Add a sorting control to rearrange job results:

- **Sort By Dropdown:**

  Options like:

  - “Date Posted: Newest First”
  - “Date Posted: Oldest First”

Behavior:

- Changing the sort option should **either trigger a new backend call** (with a sort parameter like ?sort=date_desc) or **sort the data client-side** if already fetched.
- Make sure the UI **reflects the current sort order** clearly.

Together, these features should make the job list easier to explore and more useful for users with specific preferences.

Validation & Error Handling (Frontend)

1. Form Validation Before Submission

- Ensure all required fields in the “New Job” form are validated on the client side before sending any data to the API.
- For example, check that the title, company, and location fields are not empty.
- If a field is missing or incorrectly formatted (like an invalid date), display a clear error message right next to that field.
- Prevent the form from submitting until all errors are resolved.

2. API Error Handling in the UI

- If the backend returns an error (like a 400 response for a missing title), catch that error and show a readable message to the user.
- Example message: “Failed to add job: Title is required.”
- This can be shown in a message box, alert, or a styled message section on the page.

3. Action Feedback

- Show success feedback when a job is successfully added or deleted. This can be a message like “Job added successfully.”
- Disable the submit button while a request is being processed to avoid duplicate entries.
- Optionally, show a loading spinner or status text during the request to indicate progress.

3. Web Scraping – Selenium Bot for Actuary List

1. Purpose:

Use Selenium (with Python) to scrape job listings from the Actuary List website (https://www.actuarylist.com) and import them into your application’s database. This task is meant to:

- Populate your job board with initial seed data.
- Show your ability to extract structured data from a live website using automated browser control.
- Demonstrate your understanding of integrating external sources into your backend system.

Make sure to:

- Identify key job attributes (e.g., title, company, location, posting date, and link).
- Structure the scraped data properly.
- Insert the data into your application’s database using appropriate models or API endpoints.
- Avoid hardcoding values; the scraper should dynamically load and parse listings.
- Handle any potential errors or missing data gracefully during scraping and importing.

2. Target Data (Scraper Requirements):

Your scraper must collect at least the following for **each job listing**:

- **Job Title**
- **Company Name**

- **Location** (both country and city)
- **Posting Date** — this can be an exact date or relative info like “posted X days ago”

In addition, try to extract the following **if available**:

- **Job Type** — if it’s clearly mentioned (e.g., Full-Time, Part-Time, Intern). If not, you can make a smart assumption. For example, default to **Full-Time** unless it’s labeled as something else like **“Intern”** or **“Part-Time”**.
- **Tags / Keywords** — Actuary List displays various tags for each job. These may include:
    - Categories like **Life**, **Health**, **Python**, **Pricing**
    - Role-level labels like **“Analyst (Entry-Level)”** or **“Actuary (Fellow)”**

You don’t need to capture *every* tag, but do try to collect a meaningful and representative set of tags for each job post to give context.

If some fields like job_type are not explicitly provided, you can infer a logical default based on the content.

Implementation: Write a Python script or module using Selenium to navigate Actuary List and collect job data. The scraper should:

- Launch a browser (Chrome or Firefox via webdriver) and load the Actuary List jobs page.
- Possibly interact with the page if needed (e.g., accept any cookies or click a “load more jobs” button if the site requires it to show all listings). If the site uses infinite scroll to load jobs, implement a scroll loop or multiple page loads until you retrieve a substantial number of listings.
- Parse the page content to extract the job details. You may use Selenium’s methods to find elements by XPath/CSS selectors. (Using BeautifulSoup in combination with Selenium is also fine if it helps parse the HTML after Selenium loads it.)
- Save the scraped jobs into your application database via your data model. This could be done by directly using the SQLAlchemy models or by calling your own API endpoints from the script. Ensure no duplicate entries are created if you run the scraper multiple times (you might check if a job title + company combination

already exists, for example).

- **Scope of Scraping:** It's not necessary to scrape all ~900 jobs from the site; a subset (e.g. the first 50-100 jobs or the first few pages) is sufficient to demonstrate functionality. Be mindful of time and do not get bogged down in scraping intricacies – focus on reliability for a reasonable number of records. It's acceptable to limit the scope, but do mention any such limits in your notes or video (for instance, "the scraper currently fetches the first 2 pages of results for demo purposes").

- **Running the Scraper:** Document how to run the scraping script (e.g., via a command `python scrape.py`). It could be a one-off script that populates the database initially. *Note:* We do **not** expect you to deploy a continuous scraper or scheduler – running it manually for the purpose of this test is fine.

- **Third-Party Services:** If you encounter anti-scraping measures or difficulty with Selenium, you may use simple workarounds or third-party tools (like gologin/ undetectable chrome), but this isn't expected.

4. Additional Expectations and Quality Considerations

To make the project comprehensive (yet manageable in a few days), we expect you to also focus on the following aspects:

- **Code Quality & Organization:** Organize your code logically. For the backend, consider using Blueprints or organizing routes, models, and scraping code into separate modules for clarity. For the frontend, break the UI into components (e.g., JobList, JobCard, FilterBar, JobForm, etc.) to avoid one huge component. Name variables and functions clearly. Strive for readability – another developer should be able to follow your code without confusion. Pay attention to consistent coding style and formatting. Comments or README notes should explain any particularly complex sections or assumptions.

- **UI/UX Improvements:** While the UI can be simple, an extra level of polish is expected. This includes responsive design (as mentioned), a clean layout, and intuitive interactions (e.g., clickable elements for edit/delete, form inputs labeled clearly). CSS styling can be minimalistic but **consistent** (for instance, use a simple color scheme and spacing to make the app look neat). You can use a CSS framework (Bootstrap, Material-UI, etc.) or write your own CSS – whichever you prefer. The aim is to go beyond a bare-bones look and show an eye for basic design details. Make sure the app is usable: for example, ensure text is legible, buttons are not too small on mobile, etc. *Think of this as delivering a prototype to a client – it doesn't need to be fancy, but it should not be sloppy.*

- **Performance Considerations:** You don't need to implement complex optimizations, but be mindful of obvious inefficiencies. For example, don't re-fetch the entire job list from the server unnecessarily on every minor action if it can be avoided; consider how and when to fetch or update data. Similarly, ensure the scraper doesn't do anything overly inefficient like reloading the browser for every single job. Basic efficient practices will be noted in evaluation (we're not doing load testing, but we value a sensible approach).

- **Assumptions & Trade-offs:** It's okay if you make certain assumptions (e.g., assuming all scraped jobs are full-time unless stated, or not implementing user authentication since it's out of scope). Document these briefly in your README or code comments. Part of this project is understanding how you make decisions to balance completeness vs. time – let us know what you decided to cut or simplify and why (especially if it's something you'd handle differently with more time).

6. Documentation

- README file explains:
    - How to set up and run backend, frontend, and scraper.
    - Any assumptions or shortcuts.
    - Project structure and technology decisions.

5. Communication Video (Loom/Vimeo)

Along with the code, we ask you to **record a short video (approximately 10–20 minutes)** where you walk through your project. This is to evaluate your communication skills and how well you can articulate your thought process. Please use a screen recording tool such as Vimeo (or a similar platform) and include the video link in your submission. In the video, cover the following points:

- **Understanding of Requirements:** Briefly restate what you built in your own words. What is the application supposed to do? This is to confirm you grasped the problem and goals.

- **Approach & Architecture:** Explain how you structured your solution. Describe the overall architecture (how the frontend, backend, and scraper interact), and why you chose that structure. For instance, mention any design patterns, libraries, or organizational approaches you decided on (e.g. "I used Flask Blueprints to separate concerns" or "I chose to use React Context for state management of filters" if applicable).

- **Complete Feature Demo and Implementation Walkthrough**

Your demo should cover the full flow of the application in action:

1. Show the API routes working with full CRUD operations and filters.
2. Demonstrate the React frontend fetching and displaying job listings.
3. Apply filters and sorting to confirm end-to-end functionality.
4. Highlight how the Selenium scraper is used to populate the database.

While walking through, demonstrate how each feature was implemented. Use tools like Postman or the browser console to show the API responses. On the frontend, add a new job, see it appear instantly, and apply filters or sorting to show dynamic updates. Briefly explain key elements like how form validation works and how filtering is handled through the API.

- **Challenges & Solutions:** Discuss any significant challenges you encountered while building the project (for example, difficulties with Selenium and dynamic content, CORS issues between React and Flask, tricky bugs in state management, etc.) and how you solved or worked around them. This gives insight into your problem-solving approach.

Communication tips: Speak clearly and structure your explanation logically (you may want to outline your talking points beforehand). show some code in the video to explain a decision, but try to avoid just reading code – focus on the high-level narrative. Imagine you're presenting your work to both technical team members and a non-technical stakeholder; aim for clarity and completeness.

(Add your Vimeo Demo Link to your Repository)

Deliverables

By the end of this project, you should deliver the following:

- **Code Repository:** A GitHub repository containing your source code for the backend (Flask app, models, scraping script) and frontend (React app). Include a README file at the root with instructions and your Vimeo Demo Link to your Repository

- **README Documentation:** In the README, provide clear instructions for setting up and running the application. This should include:

    - Environment setup steps (e.g., required Python version and Node version, how to install dependencies with `pip` and `npm/yarn`).
    - How to configure the database connection (include any setup scripts or .env example if needed for credentials).
    - How to run the Flask server and the React development server.
    - How to run the Selenium scraper (and any prerequisites like installing ChromeDriver/GeckoDriver).
- **Video Link:** A link to your Loom or Vimeo video (make sure the link is accessible – e.g., not private or requiring a password). Put this link prominently in the README (and optionally in your email submission if that's how you deliver).
- **Submission Package:** providing a link to your repo should suffice. Ensure that all parts of your project are included and that the project can be run with minimal fuss according to your instructions. We will be testing the setup from scratch.
- **Optional/Bonus Materials:** If you want, you can include additional documentation such as design diagrams, or additional notes on what you did. This is not required, but sometimes candidates provide a short write-up or design rationale. We will primarily look at the code and the video, so consider this optional.

❌ What NOT to Include

To keep the project focused:

- No authentication/login system.
- No Docker, CI/CD, or deployment tasks.

No test suites (unit or frontend tests are not required).

Submitting

- Submit your work before the deadline given to you.
- Early submissions are viewed favorably.
- Submissions received **after the deadline will not be considered**

Good luck!

Full-Stack Project Tips and Guidance

Hello! The team at Bitbash and our CEO wants to see you succeed on this project. We know building a full-stack app can be challenging, but following some best practices will make your life easier and impress anyone reviewing your work. Think of these as friendly tips to help you create a solid project — not a secret checklist you'll be graded against. Now, let's dive into each part of the project with practical advice to boost your confidence and results.

🔧 Backend (Flask API) Tips

- **Implement All CRUD Endpoints:** Make sure you have endpoints for **Create, Read, Update, and Delete** jobs. For example, an endpoint to list jobs (GET), create a job (POST), update a job (PUT/PATCH), and delete a job (DELETE). Test each one by calling it and verifying it works. The more complete your endpoints are, the smoother your project will be.

- **Validate Input:** Double-check that your API handles bad or missing data gracefully. If someone tries to create a job without a title or with invalid fields, return a clear error (like HTTP 400 Bad Request) and an explanation. This shows you care about user mistakes and prevents your app from crashing unexpectedly.

- **Use RESTful Conventions:** Stick to standard HTTP methods and status codes. For example, use POST /jobs to create, GET /jobs to list, GET /jobs/<id> to fetch one job, PUT/PATCH /jobs/<id> to update, and DELETE /jobs/<id> to remove. Return 200-series codes on success (201 for created, 204 for no content on delete), and 404 if a job isn't found. Structure your JSON responses clearly (e.g., include an "error" message on problems) so it's easy to understand.

- **Handle Errors Smoothly:** Don't let your app just crash or return a generic server error. Use try/except or Flask error handlers to catch issues like "job not found" and return a friendly 404 message. This is easier for users (or developers) to debug. Think of it as guiding the user when something goes wrong, rather than leaving them guessing.

- **Organize Your Code:** Keep your Flask app modular. You might put your database models in one file (like models.py) and your route definitions in another (like routes.py).

You can also use Flask Blueprints to group related routes. A neat structure (with clear folders and filenames) makes the code easier to read and maintain — which our CEO always appreciates. Avoid one giant file where everything is jumbled together.

- **Follow Best Practices:** Even in a quick project, attention to detail stands out. Don't hardcode sensitive data like database passwords — use environment variables or a config file. Make sure you're not vulnerable to SQL injection (using SQLAlchemy's query methods normally handles this for you). Also, remember to turn off debug mode or other development settings in any "production" configuration, just to show you understand security basics.

By focusing on these backend practices, you'll build a reliable API that feels professional and well thought out.

💾 Database Design and Data Handling Tips

- **Choose the Right Data Types:** When defining your Job model (using SQLAlchemy or similar), pick appropriate types. For example, use a date or datetime field for the posting date rather than a plain string. For fields like tags, decide how to store them (maybe as a comma-separated string or in a separate table for many-to-many relationships). There's no single "correct" approach to tags, but be ready to explain your choice if asked.

- **Schema Clarity:** Include all required fields (title, company, location, posting date, job type, tags). Make sure column names are clear. If you can, add comments or a short description somewhere explaining your schema design choices – it shows you thought about it.

- **Easy Setup:** Provide an easy way to create the database schema. You could use SQLAlchemy's create_all() or include a migration script (Alembic migrations are a plus but not required). In your README or docs, clearly explain how to set up the database. We suggest at least a step like "run python init_db.py" or similar, so someone can start with a fresh database without guessing.

- **Prevent Duplicate Data:** If your Selenium scraper or any import script runs multiple times, think about duplicates. For example, if the scraper sees the same job twice, it shouldn't create two identical entries. You could check if a job with the same title and company already exists before adding. This extra step improves data quality and shows you care about consistency.

- **Efficient Queries:** When you implement filtering or searching, try to do it at the database level. For instance, use SQLAlchemy query filters (Job.query.filter(...)) instead of fetching all jobs into Python and then filtering. This way, your app stays fast even if there are many jobs. It’s a bit more advanced, but it’s a good habit to form.

- **Use the ORM Correctly:** If using SQLAlchemy, manage sessions properly (commit after writes, close or remove sessions to avoid leaks). Use relationships if needed. Avoid putting raw SQL in your code; the ORM handles common cases safely. Handling the database gracefully (no leftover sessions, clear transactions) keeps your app stable.

By designing a sensible schema and handling data carefully, your project will be more robust and reliable — something that will definitely earn you points in the eyes of the team and CEO.

🖥️ Frontend (React) Development Tips

- **Fetch Data on Load:** When your React app starts, it should fetch the list of jobs from your backend API. Use useEffect to call the API on component mount, and store the jobs in state. Then display them in a list (e.g., a list of job cards). Testing this early ensures your frontend and backend are talking to each other properly.

- **State Management:** Keep track of the jobs (and any form data) in React state. For example, after adding or deleting a job, update the state so the UI reflects the change right away (or trigger a refetch). This way the user sees immediate feedback. Good state management (maybe with hooks like useState and useEffect) will keep the UI in sync with the database.

- **Implement CRUD in the UI:** Let users add, edit, and delete jobs from the interface:

    - For adding, have a form that submits a POST to your backend. Clear the form or reset state after a successful add.

    - For deleting, maybe a button on each job card that sends a DELETE request. Remove that job from state or refetch.

    - If you include editing, allow the user to modify a job and send a PUT/PATCH, then update the list.

- After these actions, make sure the list updates (either immediately in state, or after a page refresh/refetch). Seeing these operations work in the UI is always satisfying!

- **Form Validation and UX:** Before even sending data to the backend, check the form on the client side. For example, if someone tries to submit a blank title, show a message or highlight the field instead of sending the request. This is friendlier for users. Also, handle API errors gracefully: if the backend returns an error (like 400 or 500), catch it and display a polite message instead of just crashing. Little touches like disabling the submit button while loading or showing a "loading…" message can make the form feel polished.

- **Filtering and Sorting:** If your app has filters (e.g., by job type or search keywords) or sort options, make sure they actually work. You could implement filtering by:
    - Having the frontend call the API with query parameters when a filter changes (preferred).
    - Or filtering the already fetched jobs in JavaScript if the dataset is small.
- Either way, the displayed jobs list should update when you select a filter or sort option. This shows depth in your implementation. If you don't have time for all filters, include at least a basic example (e.g., a dropdown for job type that narrows the list).
- **Component Structure:** Build reusable components. For instance, a JobCard component for each job, a JobList component, a JobForm component, etc. Splitting your UI into logical components keeps your code tidy. Avoid dumping all logic and UI into one big component. Even if your app is small, demonstrating good component design is a sign of strong React skills.

Following these tips will make your frontend smooth and user-friendly, which the reviewers will definitely notice.

🎨 UI/UX Design Tips

- **Clean and Consistent Look:** Aim for a neat interface. Use consistent fonts, colors, and spacing. Even basic styling (like making job titles bold, company/location smaller) can make the app look professional. A simple CSS framework (Bootstrap, Material UI, etc.) is fine if you're comfortable with it — it helps ensure things align nicely. The goal is a polished but not cluttered design.
- **Responsive Layout:** Test your app on different screen sizes (shrink the browser window). Ensure content doesn't overflow or become unreadable on mobile. A single-column layout for narrow screens is usually a safe default. If using a grid or multiple columns on wide screens, make sure it shrinks gracefully. This shows you cared about the real user experience.
- **Interactive Feedback:** Provide visual feedback for user actions. For example, when a button is clicked, maybe change its color briefly or show a spinner if loading. After adding or deleting a job, you might display a temporary message like "Job added successfully!" or "Job deleted." These small touches reassure the user that their action was received. Also consider disabling buttons while a request is processing to prevent duplicate

submissions.

- **Complete Feature Visibility:** All the main features should be obvious in the UI. If you planned a filter or search, include visible controls (even if basic). Display important fields like job type or tags on each job card, so it's clear what they mean. If you support tags, showing them as distinct colored labels or "chips" can make the interface more lively and understandable. Essentially, make sure the UI reflects the full data model you built.

- **Smooth User Flow:** Put yourself in the user's shoes. Can they easily add a job and see it listed? Is it clear how to delete or edit? If something requires a page refresh or isn't obvious, consider adding hints or automatic updates. For example, after adding a job, you could redirect the user back to the job list where the new entry appears. A good candidate experience is intuitive and doesn't require magic to figure out.

Taking care with UI/UX polish demonstrates attention to detail. Our team (and the CEO!) really appreciate when an app not only works well, but also looks and feels professional.

🤖 Selenium Scraping Tips

- **Use Selenium Wisely:** Your scraper should use Selenium to navigate to *actuarylist.com* and gather job data. Include waits (time.sleep or Selenium's implicit/explicit waits) so you're sure the page loads before scraping elements. If the site has a "Load more" button or infinite scroll, use Selenium to click it or scroll until all jobs appear. This shows you can handle dynamic content.

- **Extract All Fields:** Make sure each job's title, company, location, posting date, type, and tags are captured. Double-check that nothing important is left empty. If posting dates come as "16h ago," consider storing them meaningfully (maybe as text or converting to an actual timestamp). If tags are listed, grab them too. Accurate, complete data means your app will look rich and realistic.

- **Handle Selectors Robustly:** Use stable element selectors (like class names or element attributes) instead of fragile ones (like absolute XPaths). This makes your scraper less likely to break if the page layout changes slightly. Adding try/except around your scrape loop (for example, skipping a job if one field is missing) can prevent the whole script from crashing mid-run.

- **Avoid Duplicates on Scrape:** Like the database tip above, think about running the scraper twice. It should ideally not create duplicate job entries. Maybe check if a job already exists by title+company or a unique ID. This shows you understand data hygiene.

- **Integration with Your App:** Provide an easy way to run the scraper. It could be a simple Python script (e.g., python scrape_jobs.py) or even an API endpoint in your Flask app that triggers it (bonus if you do this!). Include instructions in your README on how to execute the scraper. After running it, verify that the new jobs appear in your app. A seamless flow from scraping to viewing in the UI is a great demonstration of integration skills.

Even if this is the most time-consuming part, a working scraper filled with real jobs will really make your project stand out. It proves you can automate data collection and connect it to your database.

🗂️ Code Organization and Quality Tips

- **Organize Your Repository:** Keep the backend and frontend in separate folders (e.g., /backend and /frontend). In the backend folder, separate files for models (data definitions) and routes (API endpoints) help others find things quickly. In the frontend folder, group related components together (e.g., a components/ folder). A clear folder structure is a small thing that goes a long way in showing professionalism.

- **Use Clear Names:** Name your functions, classes, and variables descriptively. For example, get_jobs is better than func1. Good naming makes your code self-explanatory. This matters because if someone else (or your future self) reads the code, they'll understand it without guessing what a variable like x stands for.

- **Comment Smartly:** You don't need a comment on every line, but add comments for any complex logic or important decisions. For instance, if you did something tricky in your scraper or handled a special case in validation, write a quick note. Also, a brief comment at the top of your main files (or a good README) explaining the overall approach can help others navigate your thought process.

- **Consistent Style:** Follow a consistent code style. In Python, try to stick to PEP8 (proper indentation, reasonable line length). In JavaScript/React, be consistent with semicolons, quotes, and indentation. Using a formatter like Black for Python or Prettier for JS automatically is a plus. Consistency makes the code look clean and cared-for.

- **Remove Dead Code:** When starting from templates (for example, React's default starter), delete any boilerplate you didn't use (like unused sample components). Also remove commented-out code that's not needed. This cleanup shows that the project is finished with intention, not just cobbled together.

By writing clean, well-organized code, you make it easier for reviewers to understand your work. Our team will appreciate that you thought about readability and maintainability — it's a sign of a professional developer.

🎤 Communication and Presentation Tips

- **Explain Your Project Well:** If you're submitting a video (like a Loom or Vimeo recording), communicate clearly and confidently. Introduce yourself briefly, then outline what you built and why. Describe your architecture choices (for example, why Flask and React, how the data flows). Speaking in a friendly, logical way helps us follow along.

- **Demonstrate the Features:** Show the running app on your screen. Walk through key features: list the jobs, add a job, filter or search if implemented, delete a job, etc. If you can show the code for a moment while explaining a tricky part, that's even better. Visuals help reinforce what you're saying.

- **Mention Challenges and Solutions:** Be honest about any difficulties you encountered. For example, "I struggled with the date format, so I decided to store posting dates as strings for now." Explaining how you solved (or would solve) problems shows problem-solving skills. If a feature is incomplete, say what you would do with more time. This shows you understand priorities and have a plan.

- **Stay Organized and Engaging:** Keep the video focused. A good structure is introduction → main features → challenges → wrap-up. Try to stay within a reasonable time (around 10–20 minutes). You don't need to rush; speaking clearly and at a natural pace is key. Even if English isn't your first language, clear and courteous speech wins points.

- **Show Enthusiasm and Responsibility:** Finally, let your passion for coding show a bit. A calm but positive tone is ideal. If you make a small mistake in the demo, it's okay – the way you handle it matters. For example, "Oops, I forgot the title field there, let me fix that." This shows responsibility. Also mention briefly what you learned or what you would improve. Hiring managers love to see a growth mindset: "Next time I'd add automated testing" or "I'd improve the UI by…".

Great communication ties together all your technical work. It proves you can not only build a project but also explain and defend your work — a crucial skill for any developer.

🌟 Going the Extra Mile (Bonus Opportunities)

- **Add Extra Features:** If you have time, include something beyond the basic requirements. Maybe a search bar that filters by keyword, or pagination for the job list, or an option to mark favorite jobs. Even a simple detail view (click a job to see more info) can impress. These are not expected, but they show creativity and initiative.

- **Use Advanced Tools:** If you're comfortable, try incorporating extra technologies: for example, use React Context or Redux for state management, or TypeScript for type safety. Integrating a UI library (like Material UI or Tailwind) for a nicer look can also score bonus points. Just make sure the basics still work first; extras should enhance, not break, your app.

- **Document Thoroughly:** A great README goes a long way. Explain how to set up and run your app (including the scraper), list any assumptions, and note any "to-do" items. If you have more time, add comments or docstrings to important functions or classes. A well-written readme and clean comments show professionalism.

- **Optimize and Think Product:** Think about the end user. For example, could you cache the scraped data so the scraper doesn't overload the site? Or, add a feature to sort by posting date? These optimizations or thoughtful touches demonstrate deeper understanding. They also show you can think like a product owner, not just a coder.

- **Deploy Online (If Possible):** If you can deploy the app somewhere (even a quick Heroku or Vercel deploy), share the link. It's not mandatory, but running your app live makes testing easier for reviewers and shows you can handle deployment basics.

Remember, these extras should be icing on the cake. The core functionality (working API, data flow, working front-end) is the base. Any additional features or polish can distinguish your project and show your passion for the work.

We hope these tips give you a clear roadmap for approaching the project confidently. Focus first on making each part work correctly, then refine the details. Keep the user in mind: write clean, clear code, build an intuitive interface, and explain your work well. Good luck — we can't wait to see what you build!