Full-Stack Engineer Take-Home Problem: Job Management Dashboard

Introduction

This take-home assignment is designed to evaluate your full-stack engineering skills, with a particular emphasis on frontend development using React and TypeScript, and backend development using Django and PostgreSQL. We're looking for clean, well-structured code, good problem-solving abilities, and an understanding of how to build a deployable application.

The problem should be completable within **approximately 4 hours**. Please focus on delivering a working solution that meets the core requirements. If you have time, feel free to tackle the stretch goals.

Problem Statement: Simple Job Management Dashboard

Rescale helps customers manage complex computational jobs. For this assignment, you will build a simplified "Job Management Dashboard" where users can view, create, and manage the status of computational jobs.

Core Requirements

1. **Backend (Django & PostgreSQL)**
    - Create a Django project with a single application.
    - Define a `Job` model with the following fields:
        - `id`: Primary key (automatically generated).
        - `name`: A string representing the job's name (e.g., "Fluid Dynamics Simulation", "ML Model Training").
        - `created_at`: Datetime field, automatically set on creation.
        - `updated_at`: Datetime field, automatically updated on each save.
    - Define a `JobStatus` model with the following fields:
        - `id`: Primary key (automatically generated).
        - `job`: A foreign key linking to the `Job` model.
        - `status_type`: A choice field with at least four distinct states (e.g., `PENDING`, `RUNNING`, `COMPLETED`, `FAILED`).
        - `timestamp`: Datetime field, captures when the status was recorded.
    - Implement a RESTful API with the following endpoints:

        - `GET /api/jobs/`: List all jobs. **This endpoint should include the *current* status for each job (i.e., the `status_type` from the latest `JobStatus` entry for that job).**
        - `POST /api/jobs/`: Create a new job. Upon creation, an initial `JobStatus` entry (e.g., `PENDING`) should be automatically created for this job.
        - `PATCH /api/jobs/<id>/`: Update a job. This action should involve creating a *new* `JobStatus` entry for the job with the updated `status_type`.
        - `DELETE /api/jobs/<id>/`: Delete a specific job. Ensure that all associated `JobStatus` entries are also deleted.
    - Configure the Django application to connect to a PostgreSQL database.
2. **Frontend (React & TypeScript)**
    - Create a React application (e.g., using Create React App or Vite).
    - Display a list of all jobs fetched from your Django backend. Each job in the list should show its name and current status.
    - Provide a form or input field to allow users to create a new job by entering a name.
    - For each job in the list, provide a mechanism (e.g., a dropdown, buttons, or a modal) to update its status to any of the defined states. This action should trigger the creation of a new `JobStatus` entry on the backend.
    - For each job, provide a button to delete it.
    - Implement basic error handling for API calls (e.g., display a simple message if fetching or updating fails).
    - Implement simple client-side validation for the new job form (e.g., name cannot be empty).
    - Ensure the UI updates dynamically after creating, updating, or deleting jobs.
    - Style the application using a method of your choice to make the UI presentable.
3. **Testing**
    - Include at least one **End-to-End (E2E) test** using Playwright. This test should cover a critical user flow, such as:
        - Creating a new job and verifying it appears in the list with the correct initial status.
        - Updating a job's status to a different available status and verifying the change.
4. **Deployment & Setup**
    - Provide `Dockerfile`s for both your Django backend and your React frontend (or a single `Dockerfile` for the backend and instructions for serving the React build).
    - Provide a `docker-compose.yml` file that orchestrates your Django backend, PostgreSQL database, and React frontend, allowing for easy setup and running of the entire application with a single command.

    - Include a `Makefile` with the following commands:
        - `make build`: Builds the Docker images.
        - `make up`: Starts the entire application stack using Docker Compose.
        - `make test`: Runs your Playwright E2E tests.
        - `make stop`: Stops the running Docker containers.
        - `make clean`: Removes Docker volumes/networks if necessary for a clean slate.
5. **Performance Considerations**
    - While building the application, consider web performance optimization. Imagine a scenario where there could be **millions of jobs** in the database.
    - Address how you would efficiently fetch and display this large dataset in the frontend.
    - Briefly describe any performance considerations you made or optimizations you implemented in your `README.md` file

Stretch Goals (Optional)

If you complete the core requirements and have time remaining, consider implementing one or more of the following:

- **Job Status History View:** Allow users to view the full history of status changes for a particular job.
- **Filtering/Sorting:** Add functionality to filter jobs by current status or sort them by name/creation date.
- **Job Details View:** Allow users to click on a job to view more detailed information in a separate page with a distinct URL from the job list page.
- **Frontend Unit Tests:** Add unit tests for critical React components using React Testing Library.
- **Backend Unit Tests:** Add unit tests for your Django models or API views.

AI Usage Guidelines

We **strongly encourage** the use of AI-assisted developer tooling to implement this project, and if you do, we ask that you document the prompts that you used. Bonus points if you include some discussion on your thought process behind your prompt refinements and about anything the AI got wrong and how you fixed it.

Deliverables

Please provide a link to a Git repository containing your solution. The repository should include:

- All source code for the Django backend and React frontend.
- `Dockerfile`s for both services.

- `docker-compose.yml` file.
- `Makefile`.
- A `README.md` file with clear instructions on how to set up, build, run, and test your application using the provided `Makefile` commands. The README should include the amount of time you spent working on this and a prompt engineering writeup as described in the AI Usage Guidelines section if a copilot was used.

Evaluation Criteria

We will first attempt to build and run tests on a modern Linux or Mac OS installation. You may only assume the following about the environment:

1. The system has the following software installed:
    - `make`
    - `docker`
    - `docker-compose`
    - `bash`
2. The system can access DockerHub over the internet.

After extracting the source code archive, or cloning it from a Git repo, we will enter the top-level project directory and execute `make test` The build and test should be fully repeatable and should not require any software installed on the host system with the exception of what is specified above.

If this step fails, we will not proceed any further with the evaluation

Assuming the initial tests pass, your submission will be evaluated based on the following:

- **Correctness:** Does the application meet all core requirements and function as expected?
- **Code Quality:**
    - **Readability:** Is the code easy to understand and follow?
    - **Maintainability:** Is the code well-organized and structured?
    - **Modularity:** Are components/functions well-defined and reusable?
    - **Error Handling:** Are potential errors gracefully handled?
    - **Type Safety:** Effective use of TypeScript.
- **Testing:** Are the E2E tests well-written, reliable, and do they cover critical functionality?
- **Setup & Deployment:** How easy is it to get the application running using the provided `Makefile` and Docker setup?
- **Problem Solving:** How did you approach the problem? (This can be reflected in your code structure and README).
- **Attention to Detail:** Small touches that demonstrate care and professionalism.

Good luck! We look forward to seeing your solution.