Technical Document – Ibrahim Said

Introduction:

This application is an organizational management system that can be used to manage employees (CRUD), view hierarchy tree structures of the organization, and restrict user access based on employee roles. It features React and Tailwind for the frontend, Node.js (Express) for the backend, and Prisma (PostgreSQL) for the database storage. The system is able to support employee CRUD and includes charts that allow live searches and customized nodes. The system also includes PDF credential generation for first time users, CSV import and export, and robust admin controls.

System Architecture Overview:

Selected Architecture type: Three Tier architecture (Frontend – Backend – Database).

Deployment strategy: Each layer was deployed separately and then integrated.

Explanation as to why this was the best way:

The system is designed to follow a 3-tier architecture, where the frontend, backend, and database are separated for scalability, maintainability, and security. This architecture type ensures clear separation of concerns which ensures independent deployment for each layer.

- Database layer: Managed through Prisma ORM, where it was hosted on Supabase, which is a reliable cloud-based PostgreSQL solution.
- Backend layer: All the API endpoints along with any business logic was deployed on Railway, as it ensured an efficient and containerized runtime environment. Any pushes made to GitHub would automatically get redeployed onto Railway.
- The frontend: The frontend was built with React and Vite, it is deployed on AWS amplify, which automatically handles hosting, continuous deployment, and environment configurations.

Separation of concerns means that each layer of the system was developed, tested, and maintained independently. This made management over the system a lot less complex, aside from that this also greatly improves scalability.

This approach also means that any business logic or sensitive data in the database is kept away from a user. This improves the handling of security, authentication, authorization, and data validation.

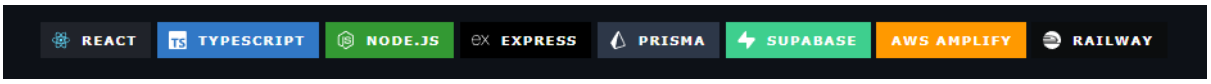
Design patterns used:

- **Observer Pattern**: This design pattern was used in the frontend of the program (hooks). When something changes, all parts of the app that care about it will automatically update (e.g. Different components may not be available depending on which type of user logs into the application). This means that the UI is always in sync with any data received.
- **Singleton Pattern**: Was used for things like the database connection (e.g. Prisma client). This made sure that there was only one connection to the database, which avoided wasting resources and running into unwanted errors.
- **Factory Pattern:** Used in utility functions like creating a PDF of exporting CSVs. These functions are then used to build the files for you instead of repeating the code in multiple places. The factory pattern is also used in the frontend components, such that they these components can be reused to generate specific UI elements.
- **Strategy Pattern:** Used for things like deciding what roles users can have or how promotions work. In terms of promotions for instance, you are able to change rules easily without messing with the rest of the code.
- **Façade Pattern:** This pattern provides a clean and easy to use interface while hiding the complex logic. An example would be the frontend hooks, you are able to use something like useAuth without needing to know how the authentication (backend) actually works.
- **Model View Controller:** Made organizing and updating the backend easier. Used in the backend where the code is split into:
    - **Model:** The database which handles all the data (Prisma).
    - **View:** Was not really used much in the API.
    - **Controller:** Handled the logic and rules.

Security considerations

- JWT tokens were used for session management and API protection where the secret key was stored as an environment variable.
- Supabase Authentication ensured secure credential handling.
- Protected Routes on both the frontend and the backend to restrict access for those not permitted.
- CORS properly configured to prevent unauthorized cross-origin access.
- Environment variables securely stored in Railways, Supabase, and AWS Amplify.
- HTTP enforced on all production deployments.

Tech Stack:



- Frontend made use of React and Vite, which allowed for fast development and provided modular components.
- Frontend was hosted on AWS Amplify as it allowed for easy integration with GitHub and supported my environment variables.
- Backend used Node.js and express as it is lightweight and scalable. Node.js also makes it really easy to integrate with Prisma.
- The backend was hosted onto Railway.
- Made use of Prisma ORM for the database as it is persistent and allowed for simple database access while ensuring type safety.
- Supabase was used to host the Prisma database. This was used as it allowed for secure login and very good session handling.

Things I would have liked to implement:

- Initial Figma designs instead of rushing straight in.
- Testing (unit, integration, nonfunctional (i.e. security and scalability tests))
- Improved documentation with Architecture diagrams elaborating on the system.
- 3d view for the Hierarchy tree.
- A more interactive Hierarchy tree that allows for dragging and dropping of new employees.
- An LLM that allowed checked whether or not an employee is fit for a promotion.
- A messaging system within the application.
- An analytics page.