Client

General:

- Components and business logic must be separated.
    - We can create an internal UI library that components we will use on all our pages. Also, we can group these components by size and complexity (e.g. atoms, molecules, organisms, etc.).
    - Components with business logic we can call `containers`. They will do some logic and pass proper props to UI components.
- Also, we have the highest components - pages. They will contain containers and static UI elements.
- For any constants, I would create a proper `constants.js` file on a level depending on the highest level of using these constants.
- Use code formatter (e.g. Prettier) and save its config file in the repo, then other developers can use it.
    - Create a git hook to run it on every commit or push, if someone doesn't use `formatOnSave`.
- Implement ESLint to support all described here rules.
    - The same about git hooks as for Prettier.
- Avoid using imports like React.useEffect.
- Avoid using imports from `@mui/icons-material/Close` when we can use `import {} from "@mui/icons-material"`.
- All function names must begin with a verb, variable ones - with a noun.
- Use camelCase for all entities.
- Use PascalCase for classes and React components.
- Use UPPER_CASE for primitive-type global constants.
- Avoid huge files like client/src/RTK/Reducers/Reducers.js. It can be split and simplified.
- Keep all text content in JSON files.
    - Initially, we can even keep them in such a way to implement translations with ease.
- I would recommend implementing a module system.
    - We should determine which main modules have the application (auth, user, job, etc).
    - Collect all components, pages, hooks, whatever, linked with a particular module. So we isolate features. It simplifies work in general and helps to work in a team.
    - Each module contains only module-specific entities.
    - All common entities are placed in a Common module.
    - Every module returns outside a fixed structure (translations, routes, any other public entities) to collect them together and use this info in the core logic that will initialize different technologies.
        i. It is difficult to explain this more clearly in such a format.
- We could create a `config` folder, where could keep configs (only some constants grouped by a purpose (config/app.js, config/theme.ts, config/server.js, etc.).
- We need to add a .env file to replace some values dynamically during the start of the app (like API_HOST some tokens, keys).

Technologies

- I would prefer Next.js instead of React, because we have enough static code, code that can be executed on the server (SSR), and work with images that can be optimized by Next.
- I would use GraphQL for communication with the server. If we need REST, I would use ReactQuery and similar to work with requests and provide caching. The last one will help us to get rid of using redux. For the remaining state, React Context will be enough.
- I would prefer using TypeScript, because now it is too difficult to read the code now, so any changes can cause errors, but with proper types, most of them could be avoided.

Server

A lot of remarks for the client part are actual for the server as well. Since the server has quite limited functionality, I don't know how it can be used, that's why I don't mention specific solutions (deploying process, rate limiters, load balancing, validation, and so on).

General

- We should avoid using obsolete syntax like var and require. If we need it for some reason, we can use webpack and babel to transform it into a target version.
- We would automatize attaching routers via app.use by keeping routers in a mapper and running a cycle.
- We should keep all response messages in the json files to implement translations with ease.
- It is better to use async/await wherever it is possible.

Structure

- We should split our server into three layers at least (controller, service, ORM).
    - We already have controllers (our routes), but they call ORM methods directly.
    - Controllers must know only about services. All business logic must be implemented inside services.
- We could create a `config` folder, where could keep configs (only some constants grouped by a purpose (config/app.js, config/database.ts, config/cache.js, etc).
- We need to add a .env file and rewrite proper places to replace some values dynamically during starting the app (like DATABASE_URL or some tokens, keys).
- Using routers in express is wrong. Now we use one router for one route. But we must use routers to group routes by the entity which they work with.
- We should have a custom separate service for mail sender and geoip.

Technologies

- I would prefer to use TypeScript here as well.
- I think it is better to use NestJs, because it provides useful functionality and sets some project structure.

- In my opinion, a relational database is a better solution (Postgres, for instance), because we have obvious relations between entities. We could take TypeORM in this case.