Recruitment task for Cosuno

Contents

Howto

Running the app

```
$ cp .env.example .env
```

Then modify `.env` with data you want or leave it as it is.

```
// Install deps
$ yarn

// Run the server
$ yarn server:dev

// Run the client
$ yarn client:start
```

The app will be working at https://localhost:3000.

To run tests, simply run:

```
$ yarn test
```

You can also run

```
yarn generate:companies
```

to regenerate (reseed?) companies stored in json files.

Running app is available at https://cosuno.codeneye.io.

Important notes

- I gave myself 8 hours to complete the task (I've done it within ~10hrs not counting the time needed to create this document), as I couldn't find more time and I think that in software development in general **it is NOT about WHAT you did and HOW, but if can you explain WHY you did what you did and WHY you did it that way**. Asking the question WHAT and not WHY often leads to messy code or -- which I think happens even more frequently -- to overengineered code which eventually leads to messy code, because of too many layers, too many tests of the same thing, fancy solutions that are hard to learn and work with, having too many unpredictable dependencies and no abstraction over those, using external solutions without proper research, etc.
- I have only a little knowledge about `Elasticsearch` and that is part of the reasons I haven't used that library here.
- For better context I've added some comments to the code and marked them with **#NotesForReviewer** text.
- All links to Github here lead to my repositories and my code.

General assumptions

- With given time span it is ok not to use pure TDD approach (tests first).
- No need for monorepo -- we need simplest working solution.
- No overengineering, especially on frontend.
- 8hrs to complete the task.
- Typescript used on both frontend and backend, to avoid silly type errors and static code analysis help.
- Frameworks, libraries (like React), database clients, etc should be considered a details and should be easy to replace.
- No need for CI/CD pipeline here, but it would be cool to have working app deployed somewhere.
- No unnecessary dependencies to avoid additional abstractions.

Backend

Assumptions

- The solution should be implemented fast.
- We want this to be scalable and extendable -- let's consider more domain driven approach.
- We don't need to use GraphQL at this stage, but it should be easy to make a transition to such interface in the future.

- Due to given time span there's no need to use sophisticated full-text search engines that would require further research or some database driver mechanisms even available out of the box. Of course, as long as general specs about the app are met.
- We don't need to handle pagination as long as it is clear how we could do that in the future.

Questions

- **Why not GraphQL already and how to transition to it later?** The data passed between backend and frontend is not complex so there is no need for GraphQL. Since the only thing that defines REST on the backend are controllers, if we wanted to use GraphQL we could add another endpoint, handle it with resolvers that could do what controllers (or request handlers if you like) are doing now -- execute the use case and map the data.
- **Do you even know GraphQL**? Yup, checkout my latest project that was using GQL on both sides (BE & FE): https://github.com/eatthatpie/covid-gauss
- **What is `use-cases` catalog?** I wanted to point out use cases explicitly. Of course, I could have a bunch of services and call that a use case, but the way I've chosen helps me understand the business layer better.
- **How about security?** There's not much about that, maybe besides basic `helmet` usage and basic CORS. What we should do more for sure is to prevent XSS attacks by escaping dangerous strings passed via requests.

What would I do if I had more time

- I would add `eslint`, and `husky` maybe.
- New requests and response objects, for the backend to be fully independent from `ExpressJS` framework (or any other).
- More tests, more TDD approach. Here is what I mean by that: https://github.com/skrybeme/skrybe-esl-service/tree/main/test
- Domain models (entities) could be classes to better handle the idea of entities being always in valid state. With classes we could throw domain errors in setters if the data violates business rules and the code would be perfectly readable.
- City name could be an entity, but this is a question we'd need to ask business people.
- I'd add middlewares, like request limiter, xss protection, logger, etc, like I did here: https://github.com/skrybeme/skrybe-esl-service/blob/main/src/http/server.ts

Frontend

Data flows

- General flow I like (not used in the app): the view uses dedicated communication layer (let's call it presenter -- it could be a provider, hook, store, etc) to communicate UI with the rest of the app. View calls presenter function to send/get data. The presenter runs business logic, which has access to data source. The presenter gets the result from business logic, maps it to view model and stores it in a place accessible for the View.
- Simplified flow (used in the app): the View calls presenter (hook) function to send/get data. The function calls data source directly, the result is returned to the View ~~which then maps it to proper view model~~ (this is not implemented at this point).

Assumptions

- We don't need state management at this point.
- No actual data logic (like filtering on actual collection items) on the frontend.
- Responsive layout, basic mobile adjustments.
- It should be doable to implement pagination later.
- We can use `create-react-app` now, but in real app I'd rather create my own webpack config. 😉

Questions

- **Why such a strange file structure?** It is about scalability: such a structure is ready for data sources, domain entities, commons, stores (available not only for the UI layer). You can see what I mean here: https://github.com/skrybeme/skrybe-app/tree/develop/src
- **What are `components` and what are `views`?** Components are pure, presentational components. I consider Views to be components aware of domain data.
- **Why pixels and not `ems` or `rems`?** It is easier to manage.
- **why ListParamsProvider?** It may seem like an overkill. Probably, the easiest way would be to have one component handling params state (query string and selected specialties). But to make it scalable we need something more than component, we need a provider, because theoretically, there may be many places in app that could influence list params. And what the actual implementation is it's a detail. We can use providers and if needed, we could incorporate global state management to handle this.

What would I do if I had more time

- **I would handle pagination, because without it, filtering on backend does not make any sense.** Moreover, current solution does not work well on slower devices, like mobiles.
- I would add `eslint`, and `husky` maybe.
- I would spend more time on loading - loaded transition as it does not look satisfying.
- I would introduce view models, so that the views could work with models defined specifically for UI and not with source models directly (backend's view models).
- I would handle checking checkboxes with TAB and ENTER keys (`tabIndex` and `onKeyDown` should do the work).
- I would handle fonts better and use `src: data(...)` instead of passing that many `ttfs` to the app.
- I would add skeleton loader for specialties in `Nav` component.
- I would add more tests, use more TDD approach -- although I wouldn't unit test each single UI component, as it a big cost for a small prize.
- I would handle tokenization, to display what part of company name matches with query string. But since this is a backend logic, also the backend should return such an info.
- I would handle responsiveness better (better layout, global breakpoints, more adjusted view for mobile, etc).
- I would try to find best debounce timeouts for better user experience. I might add two timeouts: for search query and for specialties separately.