CAREER-DAY APP DOC

Developed By: Emmanuel Karanja Ng'ang'a
Email : existentialkaranja@gmail.com
Phone: 0746 422619

Written on: 13th November 2020

Overview:

Career-Day app is a job posting application that allows job seekers to view jobs that have been posted and be able to apply for those jobs. To apply for a job, a job-seeker has to first register as a member and from there he/she can be able to apply for jobs and also track the status of his applications.

Each job posted has a stipulated interviewDate with startTimes and endTime for the interview.

APP STRUCTURE

The app is cleanly divided into two parts; the front-end that runs on the browser and the back-end which runs on the server. The back-end provides the API which the front-end will consume. The back-end is such that it can be used from any client (mobile, browser,standalone etc) provided that the front-end adheres to the API.

Systems Development Requirements:

Each of the two app parts has the following separate and distinct requirements:

I. Front-End
   -React 16 and later
   -Redux

-Bootstrap 4 and later
-Visual Studio Code was used as the development IDE.

II. Back-End
-Java 11
-Spring Boot 2.0 and later
-MySQL database
-Spring Tool Suite as the IDE.

APP DETAILS

I. FRONT-END ARCHITECTURE

The app is implemented using React as the render components with Redux as the state management container.

Any requests that need to access data from the back-end passes through **redux using redux-thunk** to perform asynchronous requests.

The **calls to the back-end** are done **via** the **axios** http request/response library.

There is a very clean API created for use by **redux state container** in the **~/career-day-app/career-day-client/src/api/** directory. The API is cleanly partitioned by the concerns it handles e.g. authApi.js for authentication connection, jobsApi.js to matters appertaining to jobs and jobApplicantsApi.js for job applicants and applications.

All code appertaining to a certain redux slice(reducer, state, action creators,action types ,selectors) is implemented using the **Ducks structure** where all the code appertaining to a given issue are **found within one file.** This co-locates them and makes them easier to modify.

All React components are found within the **~/career-day-app/career-day-client/src/Components** directory and are in eponymous sub-directories(jobs, job applicants, job applications, common etc).

Bootstrap 4 and styled-components library are used to handle the UI work whereas formik and yup are used for the forms and validation.

The React components themselves that display the information only manage ephemeral state e.g. status of whether a component is visible or not that does not require to be stored in the main app state object.

Redux is used because it will **simplify development in the long run.** For such a small app, it may seem like dreadful over-kill but it makes life easier when the app scales and when you want to implement **Progressive Web Applications** e.g. you want the application to be used off-line where there are unstable or non-existent internet connections and when the connections are available the data can be committed to the server. I have not implemented this feature in the current app but it's an easier extension from there.

Create-React-App(cra) template was used for providing the boiler-plate for the front-end app.

Due to the fact that the app uses a JWT based authorization mechanism, localStorage is used to persist the user state from session to session and as such can last even after the browser is shut-down and restarted later.

Use Scenario

You can start using the app by cloning the repository and extracting the zip file to any desired location on your machine where you have access rights to. Then go to ~/career-day-app/career-day-client and run 'npm install'. This will install all the dependencies needed by the app and then you can 'npm start'.

You can then navigate to 'http://localhost:3000' to start using the app.

For a first time run, the user is not logged in and the main view home page will be shown, this page displays the job listings and provides menu options to ‘sign-up’ or ‘sign-in’ on the navigation bar at the top.

When a user signs-up, they can then be able to view the job application options on the job page. There is a simple filter where a user can type and be able to filter the jobs displayed based on the filtering criteria entered.

If the signed in user is the default administrator , then the admin view will be displayed and this provides the additional option to create jobs, and view applicants. Only the admin can edit jobs.

For the purposes of the demo, the admin user is:

Email : existentialkaranja@gmail.com
Password: karanja01

For the sake of this demonstration, the first user created in the system is designated ‘ADMIN’ and subsequent users are designated ‘APPLICANT’.

II. BACK-END ARCHITECTURE

The source for the back-end(server) is located at
‘~/career-day-app/career-day-server’.

The back-end follows a SIMPLE,CLEAN **layered architecture.**
The layers from top-to-bottom are:

Controllers:

At the topmost level, we have Controllers whose only work is pass the validated input data to the Service Layer below. Controllers contain no business login at all, they simply pass the request data to the service layer. Domain level data structures are not visible from the Controllers.

For our purposes there are three controllers:

1. AuthController--handles logins and logouts
2. JobController--handles CRUD operations appertaining to jobs
3. JobApplicantController--handles CRUD operations appertaining to job applicants as well as applications.

**applications are handled within the JobController because you cannot have an application without an applicant and an application belongs to strictly one applicant,.

Services:

The services implement most of the business logic. Here Domain entities are visible. Apart from implementing business login Services perform the work of converting Controller level(external) data models to Domain level(models). This allows for them to change independently and keeps the business logic hidden from the controllers.

This also eliminates the usual JSON circular references problem, because the conversion is often explicit.

ModelMapper is used to perform some of the mappings. However experience teaches that manual mapping is the way to go at scale i.e. you have to create explicit mapping classes to handle the peculiarities of the differences between your Domain model and your Presentation model. ModelMapper is great when the mappings are pretty much one on one.

Note: Mapping between **Domain to Entity** models is **not done.** This is because this is a rather simpler case, but in some cases, a separate model will exist for Persistence and for Domain model and mappings similar to Presentation<=>Domain will exist between Domain<=>Persistence.

Services use Repositories to store and retrieve entities from the database.
Repositories :

Repositories are the Data Access Objects that are used to persist and retrieve data from the database. For our case, I have used **JPA 2** and later for **ORM** for that specifically JpaRepository. Repositories hide the underlaying DB details and hence will allow the specific persistence mechanisms to change. To be sure a pure interface can be create and implementations provided for each of the specific types of data persistence methods e.g. you can have a pure interface JobRepository and have a MongoJobRepository and JpaJobRepository that implement said JobRepository for those specific types of Databases. JPA is used for RDMBS whereas Mongo is just one example of a NoSQL database.

The service layer will be able to store and retrieve the data via the JobRepository for example, and not have to care about whether the underlaying data store is Relational or NoSQL.

Entities

Entities are the core domain objects that encapsulate data and behavior of domain objects. They have a unique id and are persistable. For our case, the domain is not sufficiently to warrant use of **Domain Driven Design** methods to manage the complexity, so everything here is pretty simple and straightforward. These are at the innermost or deepest layer.

DTOS(Data Transfer Objects)

These are objects that contain the input and output data for the system. As I have previously mentioned, the **Entities don't cross the service boundary** into controllers. There are many of those in a self-explanatory package and are essentially POJOs with getter/setter fields for primitives.

NOTE:
Security is implemented using **method-level authorization using AOP**(Aspect Oriiented Programming) technique. This is what I find to be a preferred way to implement cross-cutting concerns.

The API

Documentation:

The API is automatically documented using Swagger 2.
You can view this by visiting ‘http://localhost:8080/swagger-ui to view the full API. I prefer this to using manual documentation since it allows any changes to reflect immediately.

The BASE ROUTE is configured to be ‘http://localhost:8080/api/v1’ where ‘V1’ indicates the API version.

The URLs are configured as follows:
Jobs

~/api/v1/jobs for jobs CRUD operations

URL	Http Method	FUNCTION	RESPONSE
~/api/v1/jobs	GET	Returns all jobs in the database	All jobs if successful apiResponse with success, and message fields
`~/api/v1/jobs/{id}	GET	Returns one job given the job’s id {id}	Returns a job if successful or error if it does not exist
~/api/v1/jobs	POST	Creates a job with the new job JSON object passed in the request body.	Returns the created job’s details or apiResponse with success and message fields
`/api/v1/jobs/{id}	PUT	Updates a job given the id and the update job’s	Returns the updated job’s details or a

		JSON object as input in the request body	apiResponse with success and message fields
~/api/v1/jobs	DELETE	Deletes the job with id {id} allowed only for admin users	Returns an apiResponse with success as true and an appropriate message f successful or success false if failure with appropriate message.

Job Applications

URL	Http Method	Function	Response
~/api/v1/job-applicants	GET	Returns all job applicants for admin only	All job applicants
~/api/v1/job-applicants/{id}	GET	Returns one job applicant given id	Returns one job applicant or apiResponse with succes : false and message showing what went wrong
~/api/v1/job-applicants	POST	Creates a new job applicant with the details passed in the request body	The freshly created job applicant or apiResponse with success and message fields set
~/api/v1/job-applicants/{id}	PUT	Updates the job applicant indicated	The updated job applicant or

		by the {id}	ApiResponse with success and message fields set.
~/api/v1/job-applicants/{id}	DELETE	Deletes the job applicant indicated by {id}	ApiResponse with success and message fields set appropriately
~/api/v1/job-applicants/{id}/applications	GET	Gets all applications belonging to applicant denoted by {id}	All applications for a given applicant or ApiResponse with success and message fields set
~/api/v1/job-applicants/{id}/applications/{appId}	GET	Gets the application denoted as appI belong to applicant denoted by {id}	The application object or apiResponse with success and message fields set
~/api/v1/job-applicants/{id}/applications/	POST	Creates a new application for an applicant, the application details(jobId) is sent in the request body	The freshly created application or apiResponse with success and message fields set
~/api/v1/job-applicants/{id}/applications/{appId}	DELETE	Deletes the application denoted by {appId} for applicant denoted by {id}.	ApiResponse with success set to true or ApiResponse with success set to false and an error message in the message
~/api/v1/job-applicants/email-availa	GET	Checks if a given phone number or	Returns an AvailabilityRespo

ble or ~api/v1/job-applicants/phone-available		email is already being used by another applicant registered in the system.	nse with true or false

*Note that there is no API method for updating applications.
Authentication

URL	Http Method	Function	Response
~/api/v1/auth/login	POST	Logins in a user given email and password credentials passed in the request body	An authentication response with the user details and a JwtToken or apiResponse with success and message fields set appropriately.

Other Back-end Details:

The back-end uses Spring Security for both authentication and authorization, to be sure, I have implemented a JWT based authentication mechanism where on successful login a JWT token is generated and sent back to the client. The client can then attach it to the request headers and make sure it's sent to the server on every request. The JWT is decoded and if valid, the user email hashed into it is used to retrieve and load the user and attach that user to the context. From this user, information such as roles can be accessed and used in authorization.

ADDENDUM:

Known Issues:

1. There is a maven issue with use of @Before annotations for the automated tests. When you try to run the app with on the command line with 'mvn

install' or 'mvn package' or 'mvn springboot:run' ,the maven build tool throws up errors but the app runs fine from the IDE with the STS task-runner.

2. The objects used(Entities and DTOs) are simple POJOs with getter/setters for mostly primitive types I therefore didn't see any qualitative value in writing unit tests for them, in my view(I stand to be corrected), services, repos and controllers are more vital for automated tests and from experience integration tests are usually more valuable when dealing with APIs. If there is more business login in the entities, then it makes more sense to write more unit tests for them.

3. There is an issue with react-router-dom for the front-end. This is an issue that am still looking into, I believe it has to do with the latest release not being able to work well with redux.
4. There is also the issue of Spring Security throwing up errors when accessed by the front-end app, whereas it runs fine when tested via curl and postman.