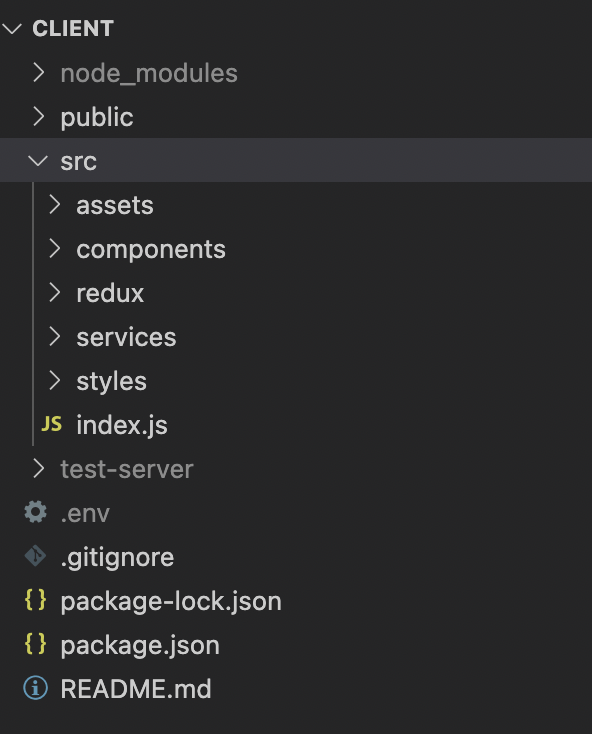
**Goally Documentation**

**Frontend Technical Document**

*Tech Stack used in the app include:*

*React.js, JavaScript, SASS, Redux-Toolkit*

**Directory Structure**

****

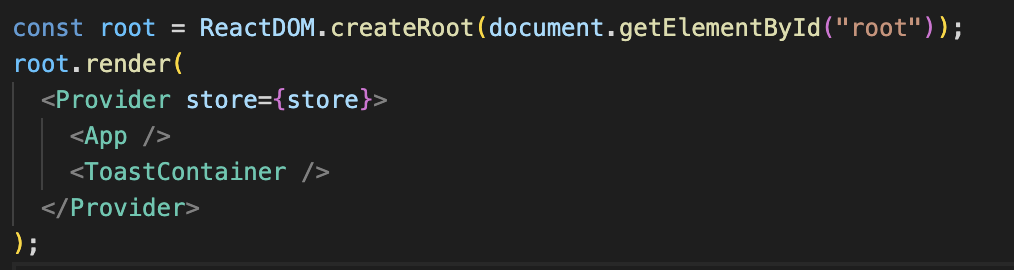
All the source files lie inside the /src folder.

Within /src:

1. /assets contains all the images and icons.
2. /components contains all the web components made in react being used throughout the webapp.
3. /redux contains the codefiles including /slices and store.js to enable app level state management in the entire application.
4. /services contain the service logic and api specifications for the entire app. Put simply, they contain all the API call methods.
5. /styles contain the entire SASS files used inside the application. To note, no styles files were produced inside of the components. It’s a design choice we went forward with keeping all the styles in one place.
6. Index.js marks the main entry point of the entire application. Also mentioned in the package.json.
7. Package.json lists all the packages (third party) being used in the entire application.

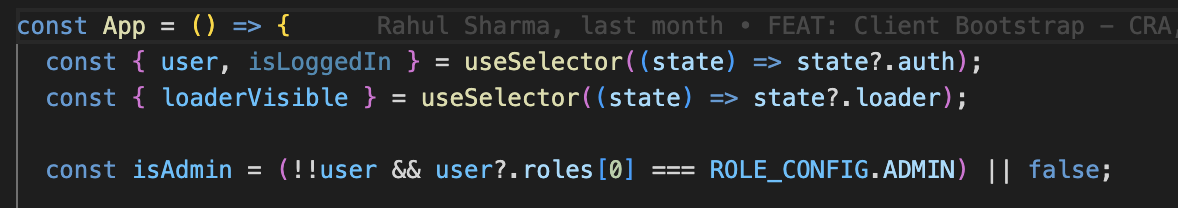
**Entry Point**

The main entry point for the app is /src/index.js



Inside this file, we have enabled an app level state management using Redux store. Also notice the <ToastContainer /> catering to all the toasts (alert messages) being triggered throughout the application at different times.

*Within app.js*

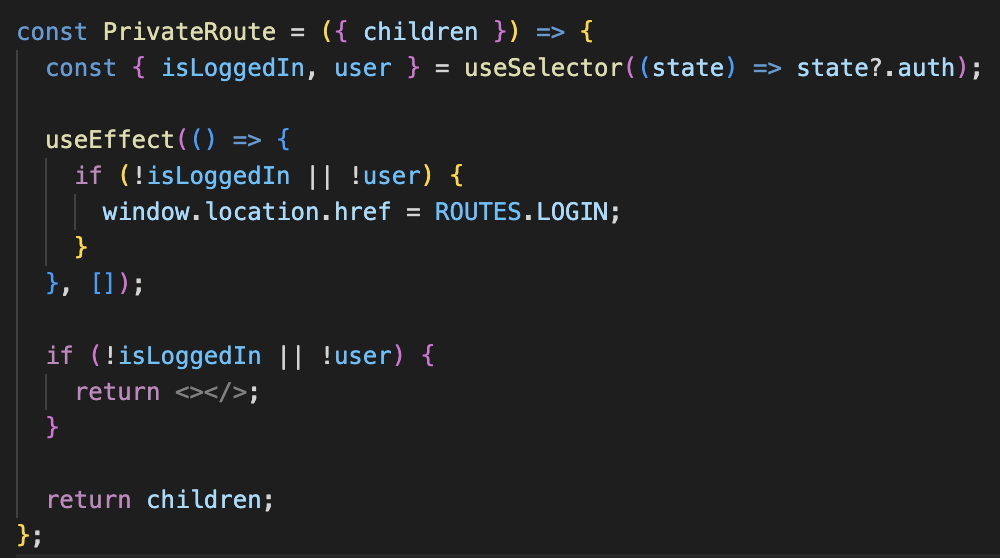


A quick look inside app.js shows we fetch user from the central store and then check if the user is admin or not. isAdmin flag is used to enable the admin level privileges across the application.

Note, the user object that we fetch from the central store is initially {} and is later populated upon login. (check authSlice.js)



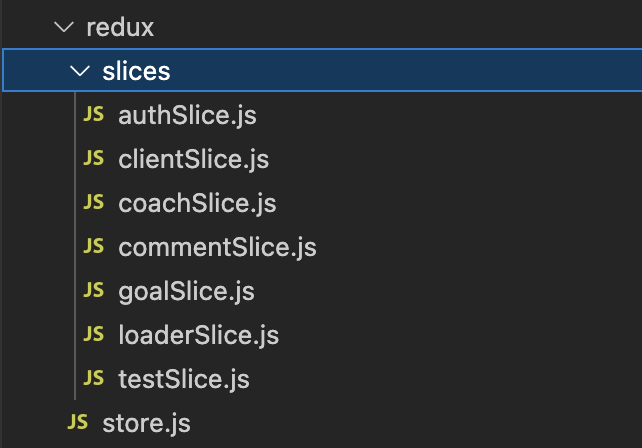
Within app, we have configured the router layer including both public and private routes. Public routes can be accessed by the user who is not logged it whereas the private route components like Dashboard and Survey Forms will not be accessible by the user if he is not logged in.



PrivateRoute component as you can see is just an HOC that returns <></> (nothing!) when the user is not logged in or the user object itself is null. Note, the isLoggedIn flag will also be populated when the user signs in on the platform (check authSlice.js).

If the user object exists on the app level state (redux) and the user is logged in then we simply return the component that was passed to the <PrivateRoute /> component.

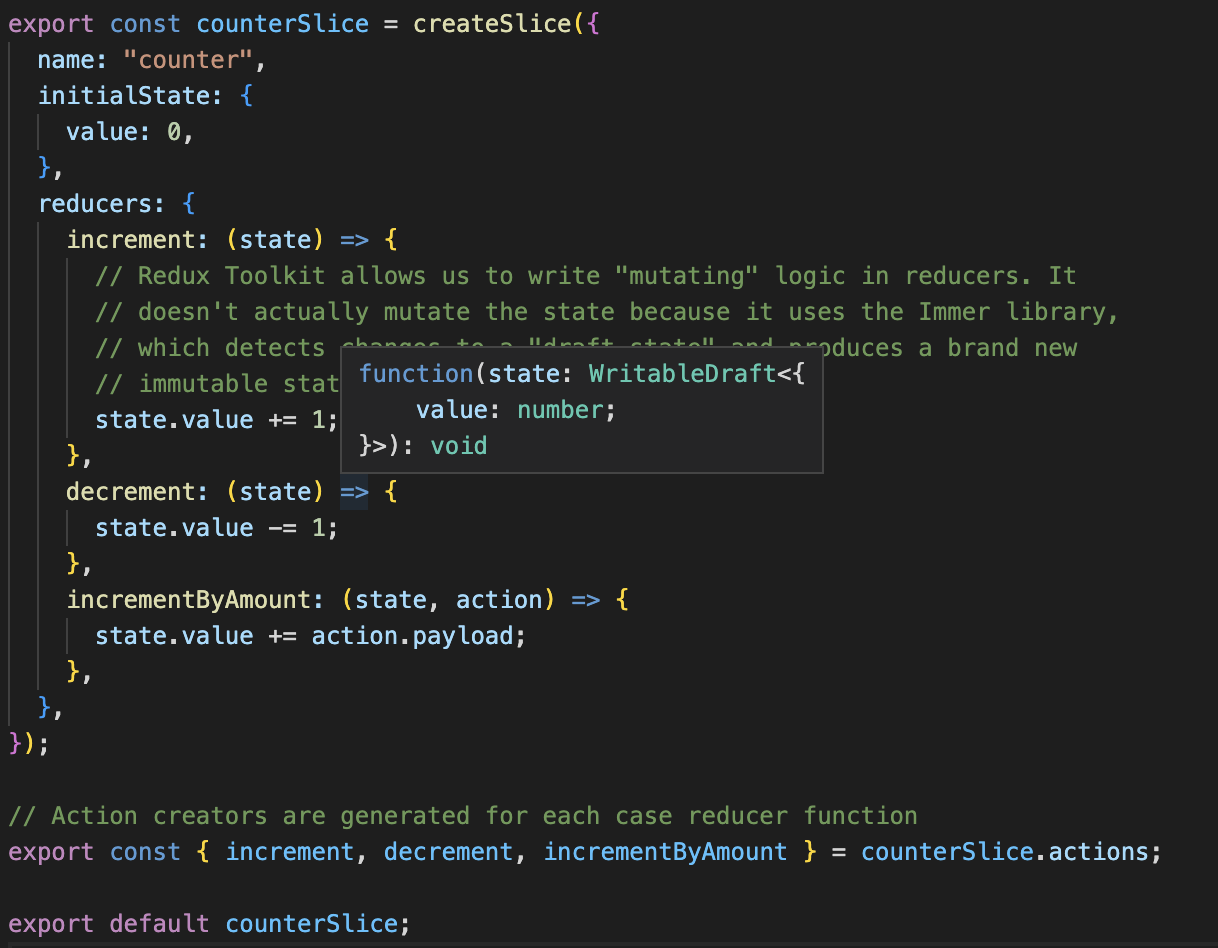
**Data flow through Redux**



Within redux, we have slices and store. Slices contain the (reducer, state, and actions), all encapsulated inside them. Also, it enables calling APIs within an action creator as they create an async thunk for the same.

*Test slice code walkthrough*

We have a testSlice setup which can be used as a template for any new slice we create.



Within the test slice, we have a state entity, named counter whose initial value is set to 0. Furthermore, we have some reducers like increment and decrement which changes the state counter’s value.

This reducers are also exported as actions from the slice itself which then can be called/dispatched using the useDispatch hook provided by react-redux.

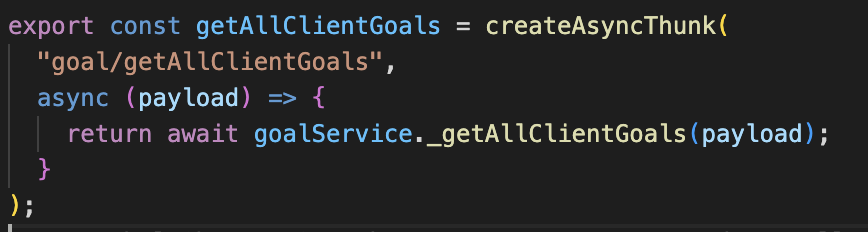
**API Request Flow**

A flow where we fetch all the goals of a client. This walkthrough aims to enable the developer to understand how we use the useDispatch, slices, actions and reducers combined in action to fetch all the goals associated with a client.

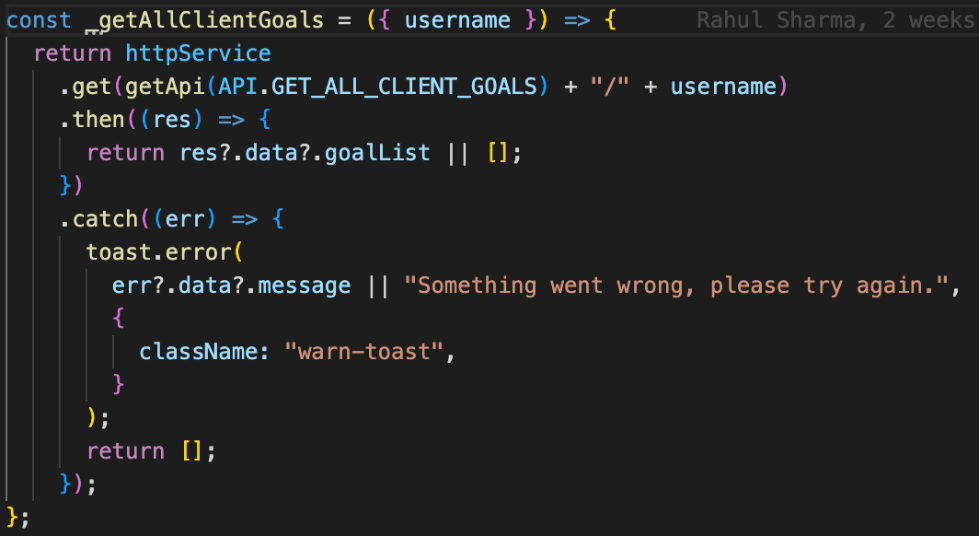
Within src/components/dashboard/index.js, on initial component mount, we dispatch (like explained before) an action, getAllClientGoals() and pass the client’s username as payload.

****

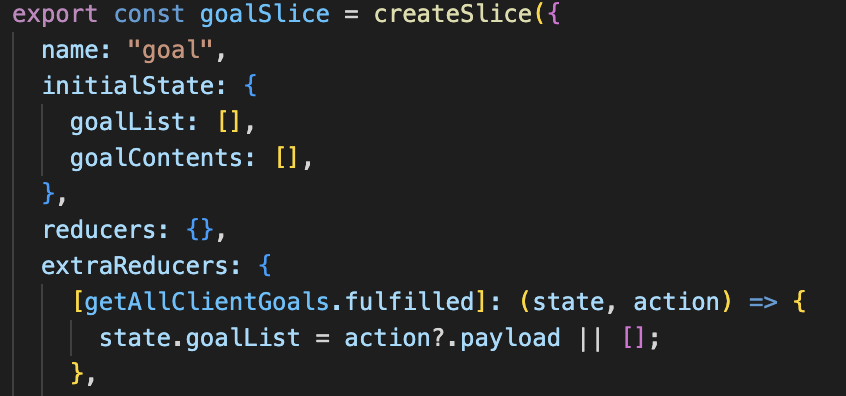
This method is nothing but an action creator method inside goalSlice.js



This calls the \_getAllClientGoals(payload) method of the goalService.js – This method inside the goalService.js is the exact place where the API call lies.



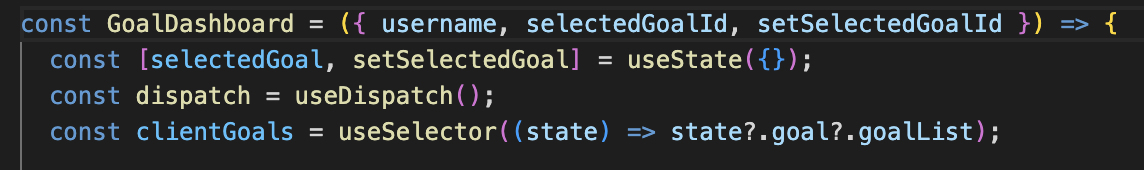
Once the API call succeeds, the res?.data?.goalList [] array returned from the backend is then passed to the reducer of the goal slice which updates the clientGoals [] key inside the central/app level goal state for the client (using Redux store).



To note, these methods which call the API service fall under the extraReducers part of the goalSlice and their completion or success is identified by the key - *.fulfilled – This is a function which expects the state and action as its arguments. The action.payload will contain the same list that we returned from the goalService’s method.*

Now when the central/app state level goal list populates. It can be easily accessed anywhere across the app using the hook useSelector provided by react-redux.

For e.g. inside GoalDashboard component, see how we select the goalList into the clientGoals.



Now once the clientGoals are updated from the redux, it triggers an update on the GoalDashboard component, which calls its render method and updates the view where clientGoals is being rendered.

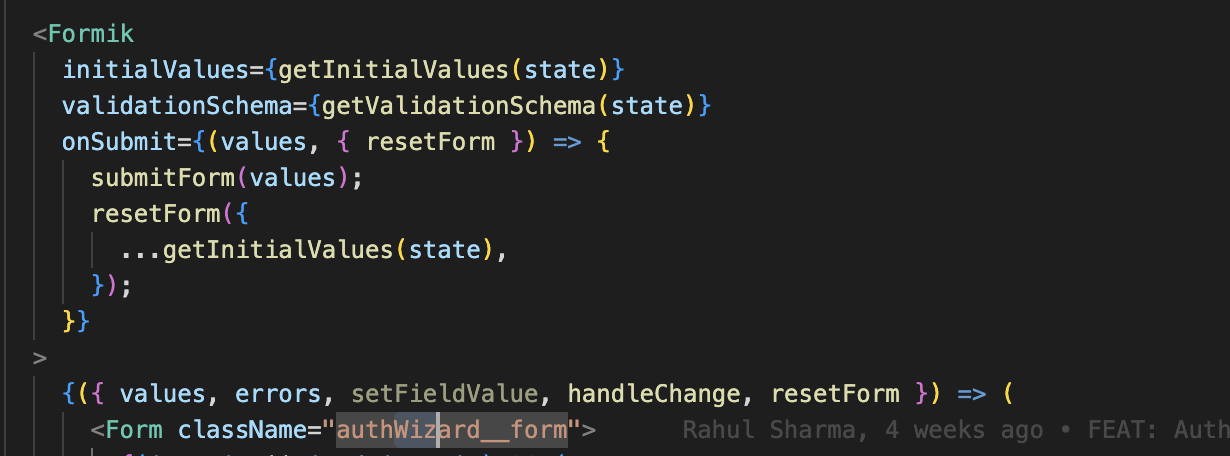


With this update, all the client goals are made visible on the goal dashboard component.

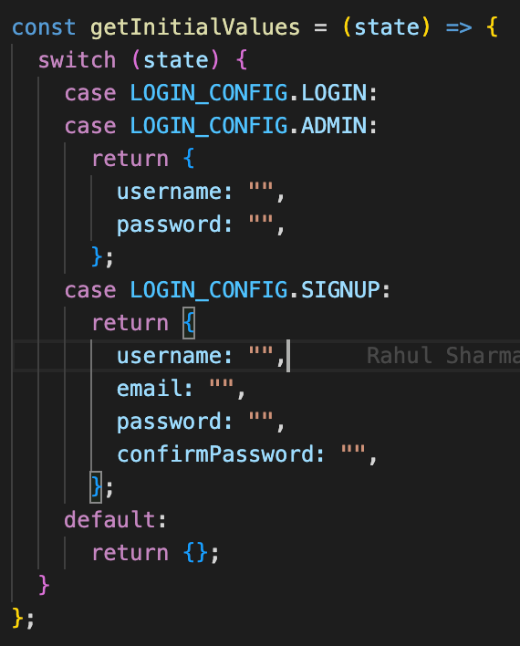
**Authentication Flow**

Initially when the user lands on the platform, he sees the landing page with the login and signup wizard, we call it the AuthWizard on the component level.

AuthWizard is a generic component which uses Formik to enable form handling including validation and field handling.



If we check the render method of the AuthWizard application, we see that the Formik expects some initial state and a validation schema. Now since we have 3 users (client, coach and admin) on the platform and one AuthWizard for all of them to enable them to have access to the portal, we provide custom init values and validation schema based on which user it is and what action he is performing (state).

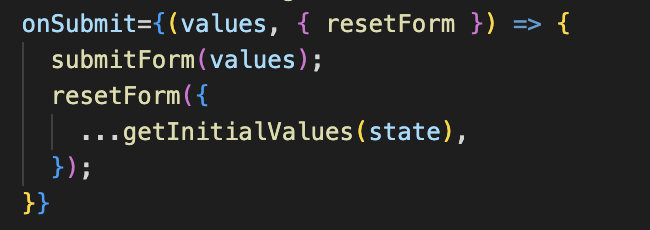


For e.g., it the user is a client/coach who is trying to login into the portal, we return an object with only username and password.



Within Formik, we have used <TextField /> component for different fields in the login/signup form. A quick look in the code above shows the initial value for the username is mapped to the username key in the values object that is inputted into Formik (which initially has our INIT\_VALUES).

For each field in the forms, we use in Formik, (also all other forms), we have a handleChange handler which updates the value of the field into the values object.



On a successful form submit, we call the submitForm method with our values object and then reset the form to its initial state for a seamless user experience.



Within the submit form method, we display the loader and based on the state being received in the AuthWizard we either dispatch the signup or signin action along with the values payload and appropriate role (in case of signup).

,

Upon signin, within the authService.js, the user is navigated to different routes based on his/her role in the application. Also, we save the user object within the local storage of the browser as well.

**HTTP Service / Axios Request-Response Interceptors**

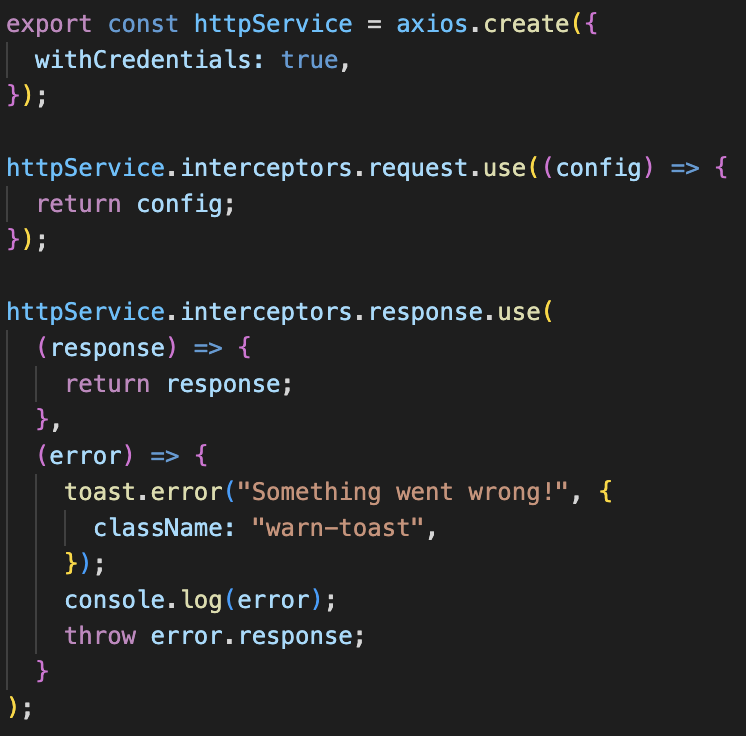
All the service/API related stuff lies inside the /src/services directory. The apiConfig.js file houses all the APIs currently being used in the application.



It also includes some helper functions to return the exact endpoint for the API.



The httpService.js, exports the httpService object which is an axios instance and contains the interceptors for both the API request and response. Any common code, for e.g. common API headers, must be placed inside the request/response interceptors.



**Backend Technical Document**

**Tech Stack**

1. Spring-Boot 2.0
   1. Spring Security
   2. JWT (0.9.1)
   3. Spring Mail (2.5.6)
   4. SpringFox – SwaggerUI (3.0.0)
2. MongoDB (6.0.3)

**SpringBoot**

1. Folder Structure
   1. Model – Pojos for entities
   2. Controller – REST controller for API access
   3. Services – service classes for different functional controllers
   4. Repository – Mongo Repositories for entities
   5. Security – Spring security code
   6. Request – custom request pojos
   7. Response – custom response pojos
   8. Utils – common util classes

A picture containing text

Description automatically generated

1. API Documentation

To make API documentation simple and central we have used **Swagger.**

You can access the swagger resource using the below urls

* 1. Launching spring app from local machine :- http://localhost:8080/swagger-ui/
  2. From AWS EC2 instance :- http://{ec2-address}:8080/swagger-ui/

Graphical user interface, text, application

Description automatically generated

1. Swagger Testing

To use swagger follow the below steps

* 1. This is the API definition :-

Graphical user interface, application

Description automatically generated

* 1. Click on “Try it Out” and input your value in the field:-

Graphical user interface, application

Description automatically generated

* 1. Click on execute to execute the API. You can see the expected output, or you will get the appropriate error: -

Text

Description automatically generated

1. Logging :-

For application logging we have used SLF4J logging.

A different logging level and pattern can be set as per requirements.

“Application.properties”

Text, letter

Description automatically generated

1. Application Properties

Comments are added for each config property.

Text

Description automatically generated

**MongoDB**

We have used MongoDb 6.0.3 for the project.

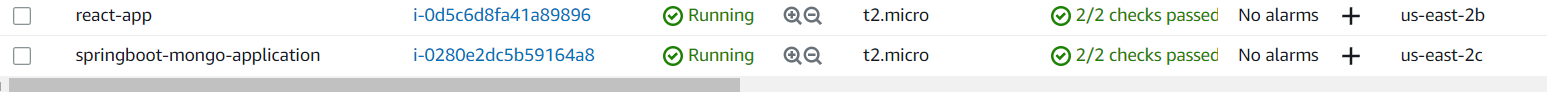
To setup on local machine.

1. Install mongo on your local machine. (default port is 27017)
2. Create a database goally – “use goally”
3. We need to create 3 static collections and import the data from excel files.
   1. “roles” – roles.csv
   2. “content” - Personalized Content.csv
   3. “tags” - Tags.csv
4. You can refer the “Database – Steps to Add Collection.docs” for more information
5. Rest all required collections will be created dynamically by SpringBoot.

**Deployment Documentation**

For deploying the application we have used 2 EC2 instances due to the system memory issue.

1. “springboot-mongo-application”:- SpringBoot and MongoDb
2. “react-app: - ReactJS application



User needs to SSH or use putty into the correct EC2 instance. “.ppk” keys can be downloaded from the AWS console to connect to these instances.

**SpringBoot Deployment**

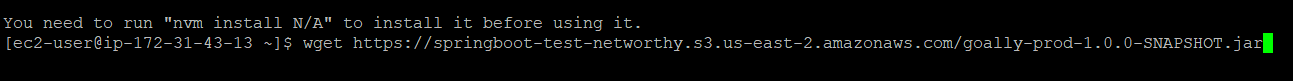
1. Upload the latest jar file to your S3 bucket.

Graphical user interface, text, application, email

Description automatically generated

1. Copy the url for the jar file and go to your EC2 instance ssh/putty screen.

Use “wget {URL}” to download the latest jar file in your EC2 instance.



1. Go to the correct directory and use the below command to run the springboot app as a background task.

“nohup java -jar goally-prod-1.0.0-SNAPSHOT.jar”

1. Go to your swagger and check if the Spring app is up and running:- Graphical user interface, text, application, website

   Description automatically generated

# **MongoDB Deployment**

1. On the same instance use “mongosh -u admin -p password” to access the mongo shell with admin privileges.
2. In case you add a new database. There will be read and write permissions required for a user that will login into that database used in your SpringBoot sonfig.

Text

Description automatically generated

1. You can now connect this Mongo remote instance with your local Mongo compass :-

URL :- “mongodb://admin:password@ec2-18-220-177-210.us-east-2.compute.amazonaws.com:27017/test2022?authSource=admin&readPreference=primary&appname=MongoDB%20Compass&directConnection=true&ssl=false”

You can see all the collections for the DB “Goally”. To add the static tables roles, content and tags refer to the MongoDB section in Backend Technical Document.

Graphical user interface, text, application, email

Description automatically generated

# **React Application Deployment**

SSH into the react-app EC2 instance.

We have linked the react github with this instance. The user just needs to pull the latest changes and run the app.

1. To pull the latest changes, go to the react app directory and run “git pull”

Text

Description automatically generated

1. We are running this app on PM2. Use the following command to start the app  
   “pm2 start --name client npm -- start”

A screenshot of a computer

Description automatically generated with medium confidence

1. The Application is now up and running.