Agora Student Marketplace:
Technical Documentation

1. Running the project in your local environment:

In the project directory (agora_student_marketplace/agora-app), you can run:

```
npm install
```

And

```
npm run build
```

Builds the app for production to the `build` folder.

It correctly bundles React in production mode and optimizes the build for the best performance.

You must then go to the front-end directory (agora_student_marketplace/agora-app/front-end) and run:

```
npm install
```

And

```
npm run build
```

In the project directory (agora_student_marketplace/agora-app), you can run:

```
npm start
```

This will run the back-end server of the project, in the form of the file server.js The server accepts requests on port 5000.

To run the front-end of the project, open a seperate terminal, navigate to (agora_student_marketplace/agora-app/front-end) and run:

```
npm start
```

Open http://localhost:3000 to view the application in the browser.

The page will reload if you make edits and save in the front-end folder. The page won't reload if you make edits and save in the back-end folder, but the backend server will restart, during this time data won't be served, so the pages will most likely have no actual content on them.

NOTE: images uploaded on the Heroku deployment will NOT show on the localhost. This is because the images are saved on the server Heroku is hosting and cannot be accessed by the application instance started on local.

2. Technology and Back-end:

Languages/Frameworks/Other Services: In this project we used MongoDB as our database. For the backend we mainly used Node.js, making use of the various modules that allowed us to listen for and process http requests, and send data to the database. For the front-end we used the React framework to develop responsive web pages. Naturally, we also used CSS and HTML to mark up and style our web pages.

Database: We chose MongoDB because it offers a free cloud based DBaaS, and since it is a noSQL DBMS, it was easy to use with the backend JavaScript framework we used (Node.js)

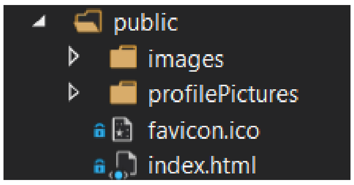
Object Schemas: The schemas for the database objects (users, listings etc.) are found in the back-end/models/ directory. In the files in these directories, the fields of the object are defined as a mongoose schema, then exported.

Routes: The back-end/routes folder contains JavaScript files that define routers and router paths that the server should listen on for requests. These routers are then exported so the server.js file can use them.

Server file: In the back-end directory you can find the server file. The server.js file uses the express module of Node.js to create an express object called app. The routers defined in the routers/ folder are then all added to the app object, as well as the api path. The api paths are named in the convention: api/objectName, e.g. /api/listings. After this, the routes defined in the route file can be accessed by appending /api/routeName + /routeFunction e.g. if the listings route file defined a POST function at the route /create, we could access this function on /api/listings/create. After adding all the route objects, the server file then defines the directory where the image files should be served from. These include the use profile pictures, as well as the listing pictures. The app object then begins to listen for http requests on the port specified in the config.js file and prints this information to the console.

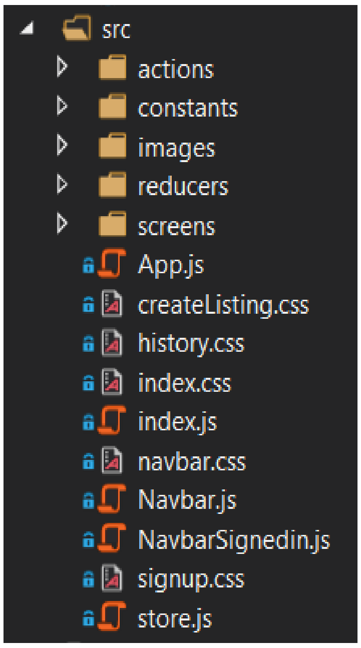
3. Front-end:

Public: The public folder holds two different folders containing images: “images” and “profilePictures”. Also within this folder is the index.html page which acts as the starting block of the app and the icon file to be displayed on the web-browsers tab when on the web-application.



- public
  - images
  - profilePictures
  - favicon.ico
  - index.html

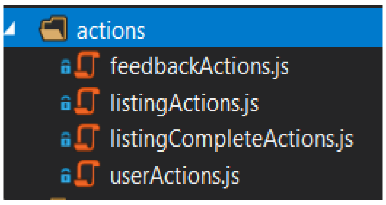
Src: The src or “source” folder consists of the primary front-end components. Components are then placed into folders with other similar components. In doing so we have created folders relating to “Actions”, “Constants”, “Images”, “Reducers” and “Screens” which will be further explained below. Keeping relevant files together promotes a clean file structure which helps both the current team and any incoming team members or external contributors.



- src
  - actions
  - constants
  - images
  - reducers
  - screens
  - App.js
  - createListing.css
  - history.css
  - index.css
  - index.js
  - navbar.css
  - Navbar.js
  - NavbarSignedin.js
  - signup.css
  - store.js

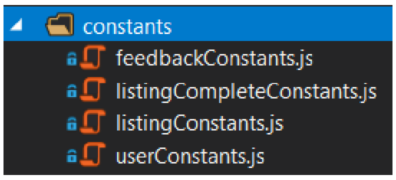
Actions: Using a *ReactJS* framework; the actions folder makes use of *Redux*, “a predictable state container for (the) JavaScript apps”. React-redux is a library available when using *NPM*, which is used to access different use cases of *Redux* into our application. Action types are constants that contain the string value that identifies the type of action uniquely in the application. Actions are the simple functions used to initiate an HTTP call using any HTTP platform, in our case “Axios”, and result in a response from the server (MongoDB). Each action created in the application has an intention to fetch various information from the server using the API, which is created via a combination of Nodejs and Java on our backend.

Agora Student Marketplace for example is a buy and trade platform and therefore have created actions to perform operations such as; fetching user details of who is selling an item, user details of who bought your listed item and fetching the details of the products listed.



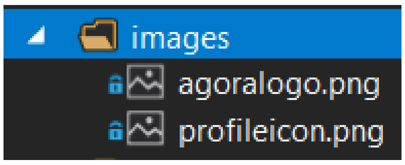
- actions
  - feedbackActions.js
  - listingActions.js
  - listingCompleteActions.js
  - userActions.js

Constants: This folder gathers all action types and therefore keeps the naming consistent. Constants are basically placeholders for string functions. These are used to ensure no human error such as spelling mistakes will occur as most IDE’s (Microsoft Visual Studio) would flag a misspelled constant.



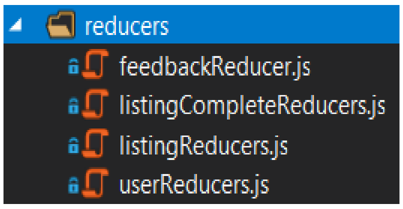
- constants
  - feedbackConstants.js
  - listingCompleteConstants.js
  - listingConstants.js
  - userConstants.js

Images: This folder simply holds the images used within the web-application. The folder location simply keeps it close to other component type files for easy reuse around the web application. These are static files and will not be modified within the application.



- images
  - agoralogo.png
  - profileicon.png

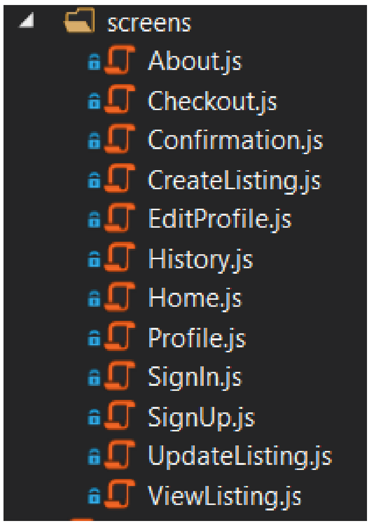
Reducers: The reducers folder consists of functions that manage the entire application’s state. They use the name of the specified action to be generated along with the action data. Based on the action’s data, the state object can be manipulated using different switch cases. Our project consists of four reducer files: feebackReducer.js, listingCompleteReducers.js, listingReducers.js



- reducers
  - feedbackReducer.js
  - listingCompleteReducers.js
  - listingReducers.js
  - userReducers.js

and userReducers.js. Depending on the data an action holds, the reducers will use switch cases to transform the object appropriately and return if criteria is met through the use of true and false Boolean; in any case, all reducer functions end with *"default: return state*;".

This is important as it returns the previous state in the default case. It is important to return the previous state for any unknown action to maintain the system's overall state to provide stability and consistency.



screens
About.js
Checkout.js
Confirmation.js
CreateListing.js
EditProfile.js
History.js
Home.js
Profile.js
SignIn.js
SignUp.js
UpdateListing.js
ViewListing.js

Screens: The front-end's "screens" folder contains the web-applications' different pages and their content's. Every page you see within the screens folder are pages you can access on the web-application. The term "Screens" is interchangeable with "Routes", "Children" and "Views" however "Screens" was an easy to understand term for the team. The app.js folder that sits alone within the *src* folder makes use of screen through the framework of ReactJS. Screens act as components in which the app.js file creates a route path that leads to the specified component and therefore the specified screen file. ReactJS allows for easy reusability. Our project made no explicit use of components but screens function in the same way. By encapsulating the logic of each screen within a file or "component", we can simply call upon them via app.js through specifying their route path.

Remaining Files: The remaining files of "gitignore", "package.json", "package-lock.json" sit within the front-end folder. "Package.json" and "package-lock.json" were created when installing ReactJS and make use of "babel" which is a compiler for JavaScript. These comprise the background processes needed for the application to run using ReactJS. Lastly there is a "README.md" file which explains how to start the application; these instructions are reiterated within the document under "Running the Project".