Problem Statement

Allies Interactive Services Pvt. Ltd. also runs World's Largest Vector Design & Distribution Company, Microstock (https://www.microstock.in/), which has portfolio of more than 1 million vector graphics elements and 70000 vector compositions/images.

Designing more than million vector graphics elements and compositions is not a tough task for the Microstock team, but searching and retrieving it from the local server is. The problem arises when the graphic designed long ago was to be reused. It became an overhead for the team to search into directories and retrieving it for further use.

As an Intern at Allies, I am assigned to solve this problem by designing a simple and clean User Interface and User Experience of a search engine that will perform searching by crawling through metadata of the image files which is being indexed by elasticsearch (https://www.elastic.co/).

Solution

The Solution to this problem was to develop and design the search engine with minimalistic UI that'll search within the set of Vector graphics.

Step 1: The important thing for the search engine is the data it searches from. I was suggested to use elasticsearch for

indexing all the files and then search through the JSON file that will be generated after the indexing.

Elasticsearch:

Elasticsearch is a search engine based on Lucene (a free and open-source information retrieval software library). It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. No database is used in this case study. It is extremely fast as compared to Relational Database Management model.

The data is indexed according to indexer built which will fetch all the metadata information, which includes: *File name, File Size, Keywords, Subjects, EXIF information (if any) and directory it is in,* of all the files in the directory into a JSON format.

Step 2: Now that the data is collected, an interface is needed to display all the image files according data which is retrieved by search query. To do this, **Mr. Abdul Qabiz**, CTO at Allies and my guide for this internship, suggested me to design UI using React JS and Redux.

React JS:

React is front end library for building user interfaces developed by Facebook. It's used for handling **view** layer for web and mobile apps (i.e. ‘v’ of MVC pattern). React JS allows to create reusable UI components. It is currently one of the

most popular JavaScript libraries and it has strong foundation and large community behind it. The best thing about React is that it works with the **virtual DOM**, i.e. It does not affect original DOM model, instead it manipulates the components within the DOM.

Redux:

Redux defines itself as a predictable state container for JavaScript apps. Redux is inspired by Flux that makes you think of your application as an initial state being modified by a sequential list of actions, which is a really nice way to approach complex web apps and opens up a lot of opportunities.

Step 3: After designing the Interface, another task is to receive all the images from the search query input by the user. For this, I used Elasticsearch JavaScript API, that fetches all relevant search results matching the search query. And after the data is received in JSON format, the parameter: '*file_name'* is used in *<img>* tag under the *src* attribute.

Step 4: The Interface is created, images are now being displayed after the search input query. Now I have to design the experience, for the user as well as the browser. The experience, that takes minimal efforts and with more functionalities.

Step 5: From the list of images received from the search query, select one vector and display its metadata

information. This can be achieved using **React Router**. React Router is a collection of navigational components that compose declaratively with your application. i.e. It helps in navigating from one component to another.

The Experiences I am integrating currently are:

1. Display all results, even before search query, after loading of the Application.
2. Live Searching, i.e. the results being generated while user is typing.
3. Infinite Scroll, keep scrolling and rendering results until all results are not displayed.
4. Viewing the single vector, and on closing that, back to search results.
5. View the vector in modal, if opened from the results and view it in a separate page, if opened from the direct link.

   Elasticsearch provides 10 results from zeroth index by default per query but this can be tweaked by passing parameters as *size* & *from* parameters.

DIRECTORY STRUCTURE:

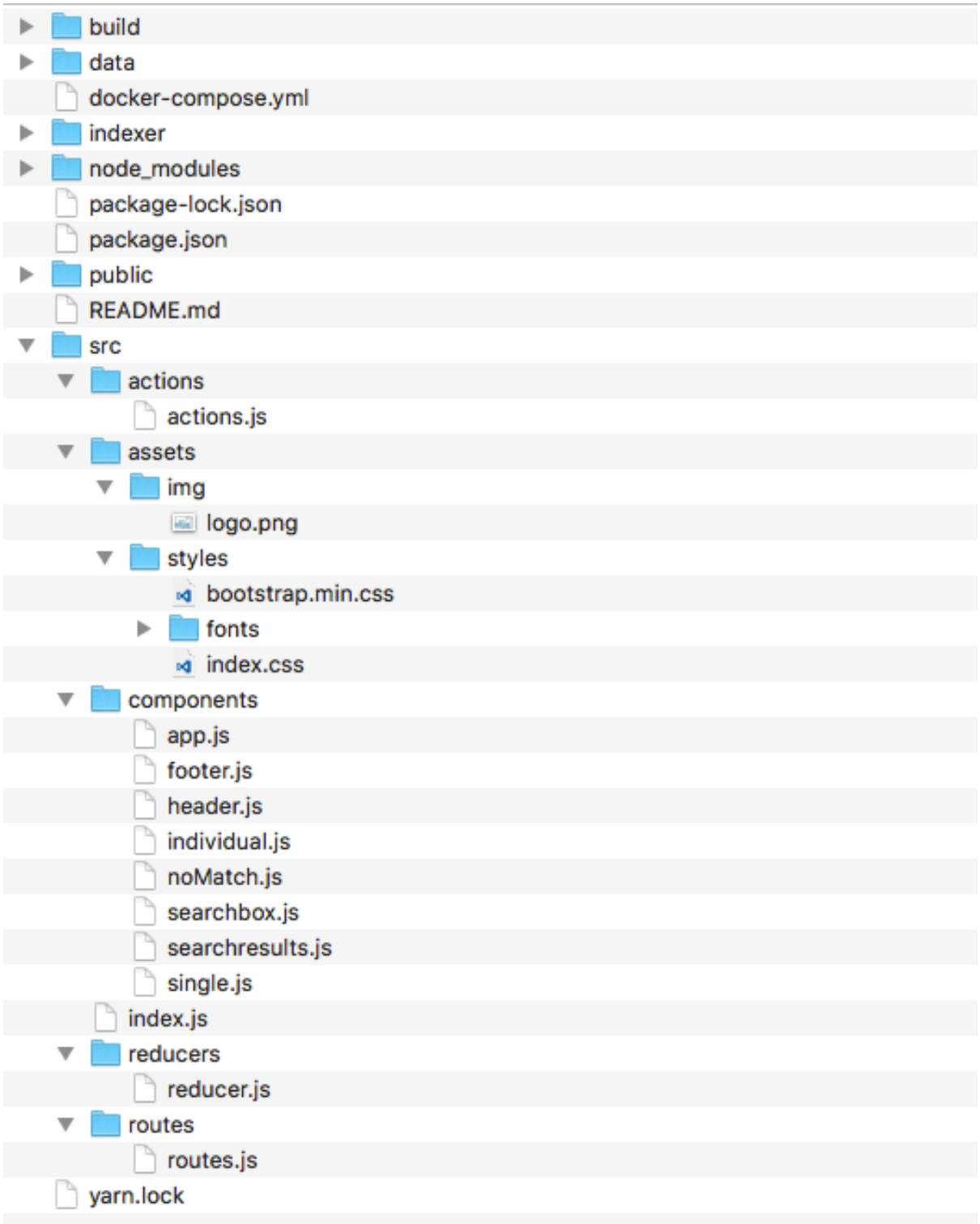


Figure 1: Directory Structure

STEP 1:
Extracting and Indexing the Data

In this step, I used an elasticsearch docker container to create a search engine on the server. After that creating an indexer to search the data according to the indexed data.

1.1 Elasticsearch:

Elasticsearch is a search engine based on Lucene (a free and open-source information retrieval software library). It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. No database is used in this case study. It is extremely fast as compared to Relational Database Management model.

Elasticsearch uses Kibana, which lets you visualize your Elasticsearch data and acts as a frontend for elasticsearch queries and indexed data analysis (https://www.elastic.co/products/kibana)

1.2 Setting Up the Search Engine:

Instead of setting up these tools individually let's use Docker to bring up the official containerized distributions of each of these components in seconds (with functional defaults.) Docker's new *docker-compose* command starts and links multiple containers together based on the configuration in a *docker-compose.yml* file.

Docker:

Docker is the world's leading software container platform. Developers use Docker to eliminate "works on my machine" problems when collaborating on code with co-workers. Operators use Docker to run and manage apps side-by-side in isolated containers to get better compute density. Enterprises use Docker to build agile software delivery pipelines to ship new features faster, more securely and with confidence for both Linux and Windows Server apps.

1.2.1 docker-compose.yml configuration file:

```
elasticsearch-1:
     image: elasticsearch
     container_name: elasticsearch-1
     ports:
        - "9200:9200"
        - "9300:9300"
     volumes:
        - ./data/elasticsearch.yml:/usr/share/elasticsearch/config/elasticsearch.yml
     command: elasticsearch

kibana-frontend:
     image: kibana:latest
     container_name: kibana-frontend
     ports:
        - "5601:5601"
     links:
        - elasticsearch-1:elasticsearch
```

The configuration file tells Docker to pull down the official images for Elasticsearch and Kibana from Docker Hub, set up a custom configuration file and connect the Kibana container to the Elasticsearch container. The command: *docker-compose up* starts the elasticsearch engine server at http://localhost:9200 and kibana frontend server at http://localhost:5601 .

Running this command on terminal in the top level directory:

```
$ docker-compose up
```

Now, a powerful foundation for search that can both handle large amounts of data and be scaled out to service a high volume of requests.

1.3 Building the indexer:

Now, an Indexer is built that will extract all the metadata information from the image files in a server directory and will create a JSON format for the indexed files. Also Kibana can be used to display the index and the data in it.

It is pure JavaScript that runs on node.js server. This script is an extension of photo indexer created by GitHub user Jettro Coenradie (https://github.com/jettro/nodejs-photo-indexer) which is a photo indexer based on node.js, elasticsearch and exiftools.

The ***indexer*** directory is for indexer and ***index.js*** file is a JavaScript file which creates and indexes the images from ***photos*** directory.

1.3.1 Running indexer

In the indexer directory, after installing all the dependencies through

```
$ npm install
```

run:

```
$ node index.js photos/
```

1.4 Passing the search query

Now the index created is named **Photos**, and the items in it are named **Photo**. The typical search query that is passed is like:

http://localhost:9200/photos/photo/_search?q=<search_term>

and it returns a JSON format result, that will be later useful for getting the file name of the image, to display on the page.

STEP 2:
Creating the User Interface

In this step, I developed a User interface by creating a React Js application.

2.1 React Js:

React is the JavaScript library for designing the user interfaces by dividing the application into components. These components help in manipulation of DOM and only renders or change the *part* of web app. That is why it is said to be worked on Virtual DOM.

2.1.1 Why use React?

React works on Virtual DOM that is an abstraction and it only works on that part of the application that needs to be changed. That brings up ease of access to browser as well as the user. It is relatively fast and works on “view” part of MVC (Model View Controller) pattern.

2.2 Components:

Several Components were created for this web app. The main directory used to render the UI in this React app is ***src***. The ***index.js*** file in ***src*** directory is the container component that just Renders the Main Component (i.e. ***app.js***) to the HTML file (***index.html***) in the ***public directory***.

app.js is further divided into sub components:

1. ***header.js:*** This file contains the header part of the web application.
2. ***header_items.js:*** This file contains the items that should be included in the header. It is the sub part of the header component.
3. ***searchbox.js:*** This file contains the searchbox, where user will enter the input. It is also one of the main component of the app as it inputs the search query processes it and displays the results in Searchresult component.
4. ***searchresults.js:*** The Component that will display all the search results.
5. ***single.js:*** This component is sub part of searchresults component, and it views single vector image with its details in a modal.
6. ***footer.js:*** This is the footer components, that includes copyright declarations and link to documnetation (Non-Functional Requirement).
7. ***individual.js:*** This component is sub part of searchresults component, and it views single vector image with its details in a seprate page.
8. ***noMatch.js:*** This component is shown when the page requested is not available.

To run this react app, Run this command on top level directory:

```
$ npm start
```

The react app is then by default can be viewed in a browser by visiting:

http://localhost:3000/

And here is how it looks like for now:

Figure 2: Main User Interface of the Web Application

STEP 3:
PASSING QUERIES FROM SEARCH INPUT TO ELASTICSEARCH

In this step, the input from searchbox is sent to elasticsearch server for searching through elasticsearch javascript API.

3.1 Initializing JavaScript API

An instance is created in ***app.js*** by:

```
import elasticsearch from (`elasticsearch`);
let client = new elasticsearch.Client({
  host: 'localhost:9200',
  log: 'trace'
});
```

I have used EcmaScript 2015 (ES6) syntax.

3.2 Sending Search request

The query of input from ***searchbox*** is taken under the *value* attribute and the passed to ***searchresult*** component. The ***searchbox*** also contains an attribute *onChange,* which calls a function everytime there is an input. In this case, it is calling a function: *handleChange()* that sends the search query in this form:

http://localhost:9200/photos/photo/_search?q=<search_term>

```
client.search({
```

```
  q: search_query
}).then(function (body) {
  this.setState({ results: body.hits.hits })
}.bind(this), function (error) {
  console.trace(error.message);
});
```

Where, *search_query* is the input provided by the client.

3.3 Fetching results in an array

results is an array here where the JSON data is stored. The data is stored in array of objects. To fetch any data, we use properties of the objects.

For e.g. to retrieve file name of the object, we use *results.file_name* where *file_name* is the property of the image that is indexed in JSON file.

STEP 4:
DISPLAYING THE SEARCH RESULTS

In this step, after the results are being fetched, we need to display the search results in form of a grid and also display all the results using infinite scroll. By default, it will show *30* results per page.

4.1 Displaying images in a grid

results is an array here where the JSON data is stored. The data is stored in array of objects. To fetch any data, we use properties of the objects.

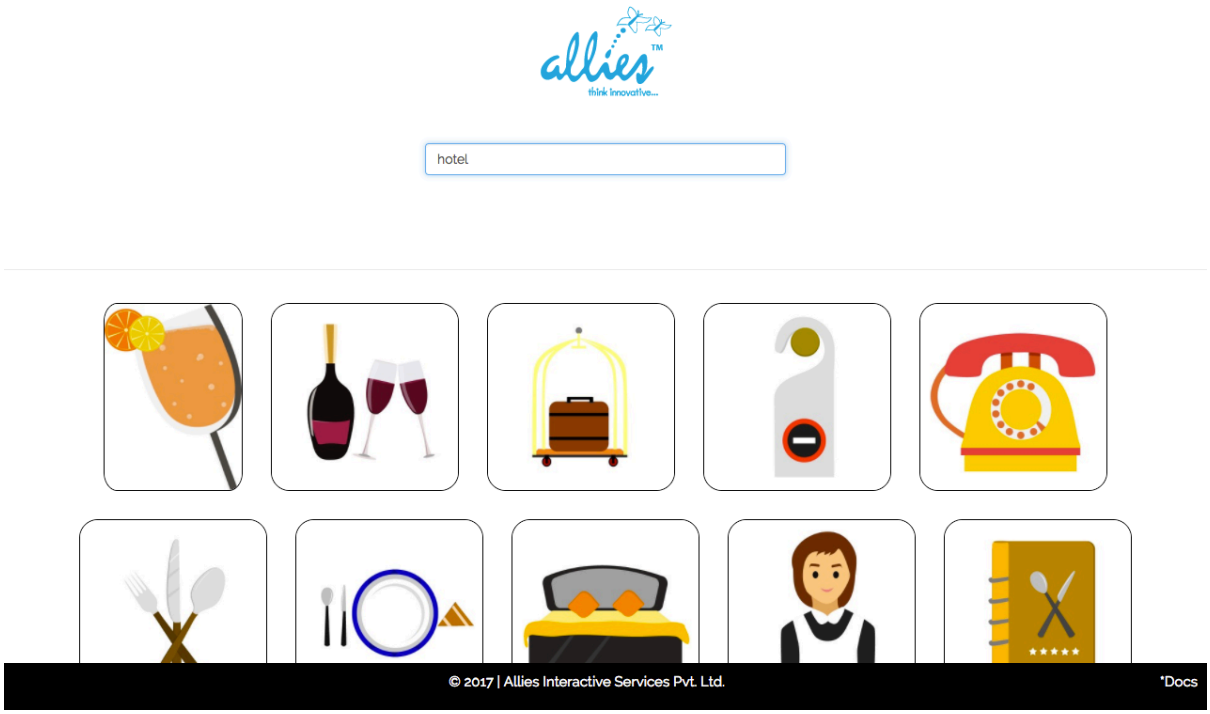
We'll array.map() function and in that we will display image's src attribute as *results.file_name* property. To retrieve file name of the object, we use *results.file_name* where *file_name* is the property of the image that is indexed in JSON file.

And we are using <li> tag to display the images with CSS properties being, display: inline (to display list in a single line) and list-style: none (to remove bullets). Here is the function to display images:

```
{ this.props.results.map((result) => {
       return (
              <li key={ result._source.file_name }>
                     <LazyLoad className="lazy">
                     <div className="istyle">
                                   <Link to={{pathname: '/photo/' +
result._source.file_name + '/' + result._source.keywords, state: {
modal: true }}}>
```

```
        <img src={'/photos/' + result._source.file_name}
className="image grow-shadow" alt="Search Result"
/>
 </Link>
 </div>
 </LazyLoad>
 </li>
 ) }) }
```

Now our application will look like this:



When we are using <Link> from react-router-dom package, we are giving a route to display another component. In this case, we are passing image's filename and keywords as the parameters.

4.2 Pagination using Infinite Scroll

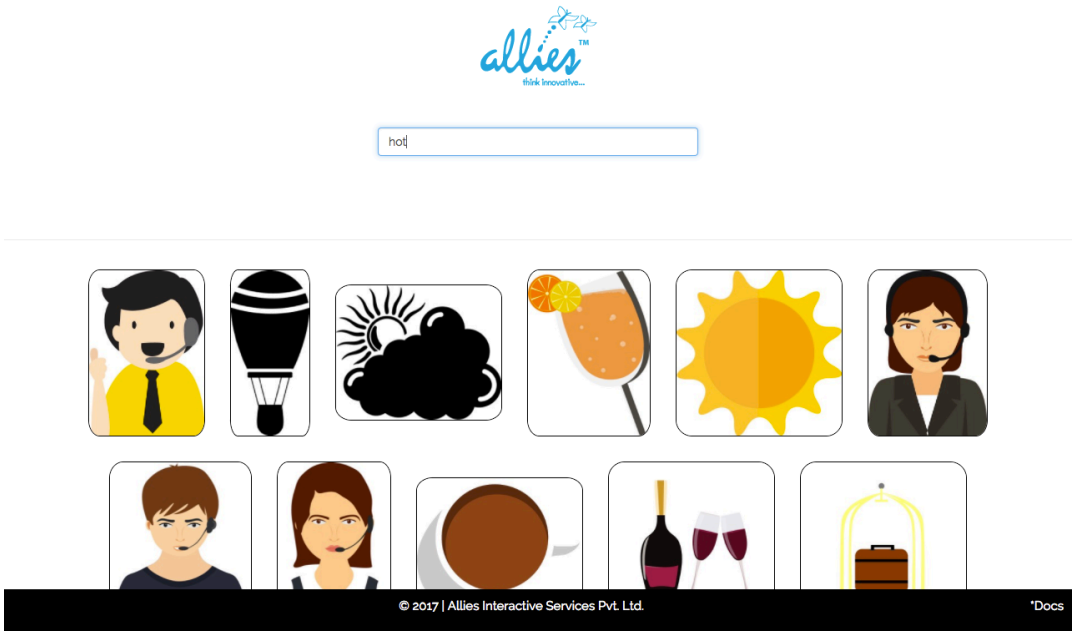
I have set size to 30 for an elasticsearch query for one page. Now the challenge was to display next set of queries, below the older queries in the same page. Elasticsearch uses *from* parameter to scroll to next page. The *from* interval set is *size i.e.* each page will consist of 30 search results and from query will be sent in multiples of 30 (0, 30, 60, 90, and so on). So what I have done is, took all the search results from new query, stored them in a new array and concatenated it with the old results array.

For calling the function to retrieve next set of results, I have used react-scroll-pagination npm package. This Package is called in searchbox.js file which calls next() function, that sends query to elastic search with from size being: *from=from+size.*

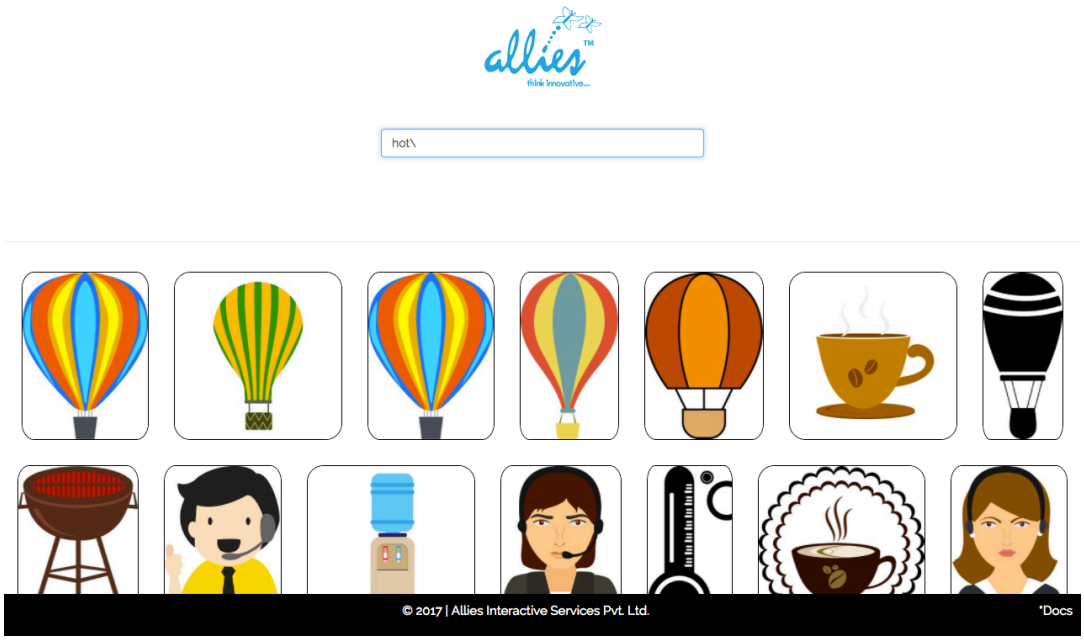
So whenever the client reaches the end of first page's search result, the next set of results are generated until it reaches the end of the total results retrieved from the search query.

More npm packages used in the project are specified in package.json file.

To get the results of exact keyword, type an `\` in the end of query. For example, hot will return results of **hot, hot**el, **hot**line, etc.



So in order to get search results of just ‘hot’, enter `\` in the end of hot (i.e. **hot**).



STEP 5:
DISPLAYING THE SINGLE VECTOR IMAGE WITH ITS METADATA INFORMATION

Now, another task is to display single vector on being clicked from the search results. And the challenge is to display the single vector with its information in a modal, if it is clicked from the search results and to display it in an individual page, if accessed directly from the URL.

5.1 Creating a single page component to be displayed in a modal

The parameters passed to this component (single.js) are filename and keywords. So in our Modal, we need to display the image, that we retrieve using *file_name* parameter like we did while retrieving the images for search results, and the keywords using *results.keywords* properties. To display it in a modal, we need to create a modal container, and modal itself.

We defined Styles for

i) modal container:

```
const ModalWrapper= {
                    position: 'fixed',
                    top: 0,
                    left: 0,
                    bottom: 0,
                    right: 0,
                    background: 'rgba(0, 0, 0, 0.15)'
                }
```

ii) modal:

```
const Modal = {
                position: 'fixed',
                background: '#fff',
                top: 25,
                left: '10%',
                right: '10%',
                padding: 15,
                border: '2px solid #444'
            }
```

and used them to display the content in modal. The URL will change to the *file_name.*

5.3 Creating a single page component to be displayed on a single page

This component (individual.js) will display the information when directly accessed from the URL and not any other state. The parameters passed to this component are also filename and keywords. So in this component also, we need to display the image, that we retrieve using *file_name* parameter like we did while retrieving the images for search results, and the keywords using *results.keywords* properties.

5.4 Conditional routing and rendering of the components

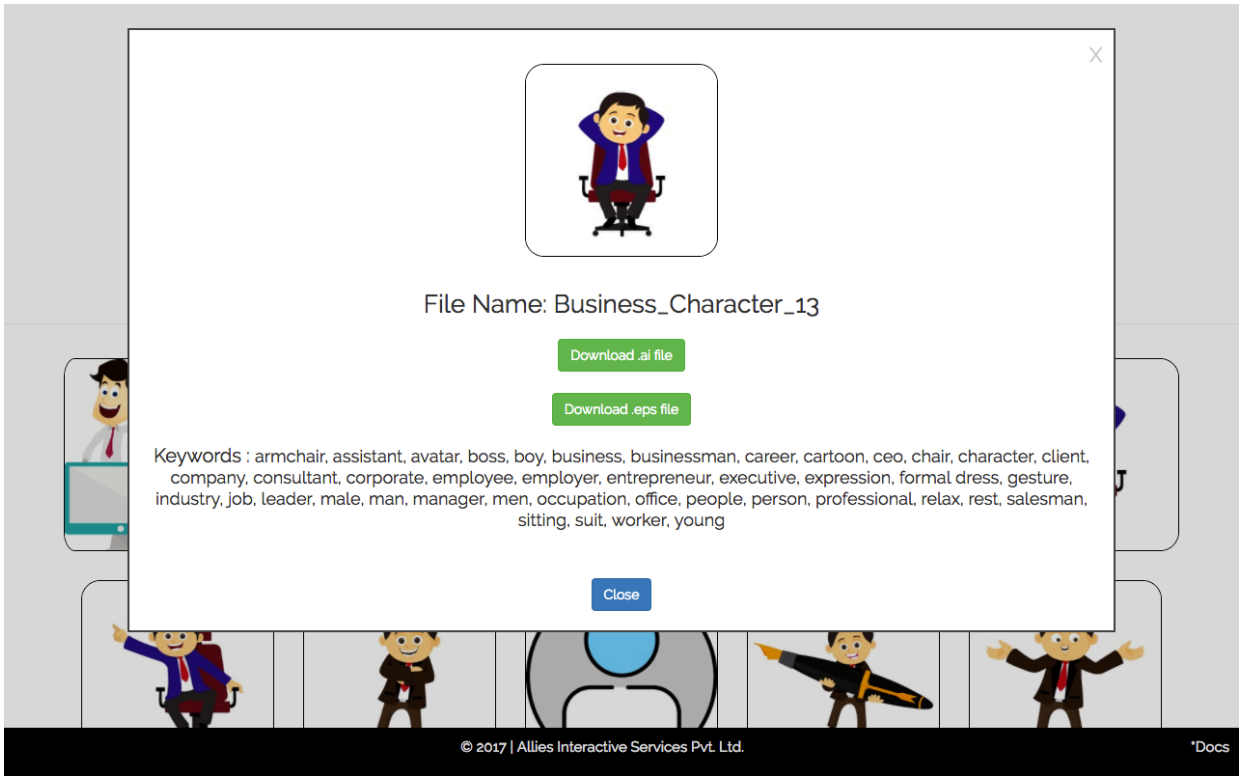
To display the component according to our condition, i.e. to display component in modal, when accessed from a search results and display in a new page, when directly accessed from

the URL. To do that we need to configure Routes from our ***routes.js***. i.e.:

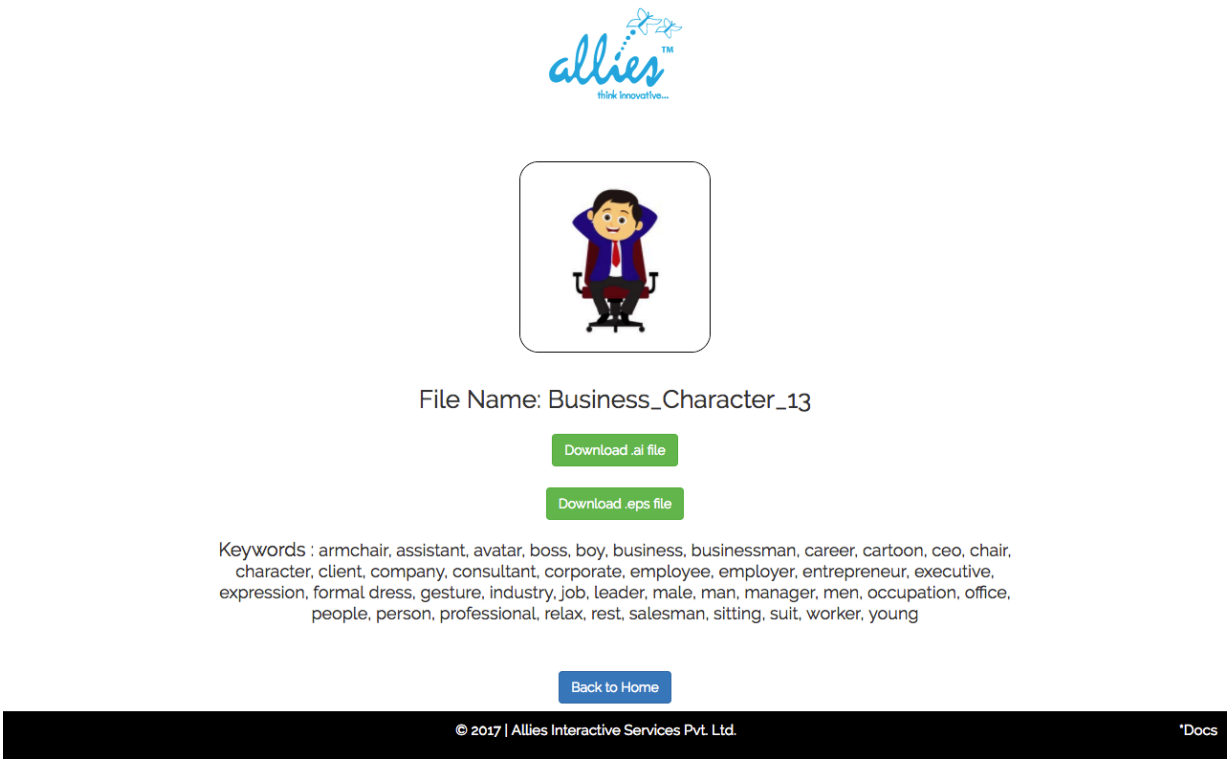
```
<Switch location={isModal ? this.previousLocation : location}>
       <Route exact path = "/" component={App}/>
       <Route path = "/photo/:id/:keyw" component={Individual}/>
       <Route path = "/*" component={NoMatch} />
</Switch>
{isModal ? <Route path='/photo/:id/:keyw' component={Single} /> : null}
```

In Routes, we have used switch with location being previousLocation, if on the search results screen, and location, if on any other location. This will trigger isModal to be true or false. If isModal is true, Single Component (the one that will display in modal) will be executed, else, Individual Component (the one that will display in separate page). Will be executed.

When the component is open in a modal:



When the component is opened directly from the URL:



Single component will display **Close** button that will bring back to search results, whereas Individual component will display **Back to Home** button that will take to the homepage of the application.