Lab 5

Due Apr 17 by 11:59pm **Points** 100 **Submitting** a file upload

CS-554 Lab 5

For this assignment, we're going to apply what we've learned with Redis, React, and now finally GraphQL to make a Location aggregation website similar to FourSquare, called BoreSquare.

You are a developer working at a new startup called BoreSquare who wants to make a **boring** copy of FourSquare, so here you are on your first day…

Data

Locations are the primary pieces of data that we're going to concern ourselves with. A post will be defined in the following format:

```
type Location {

    id: ID!

    image: String!

    name: String!

    address: String

    userPosted: Boolean!

    liked: Boolean!

}
```

In the above schema, `id`, `image`, `name` and `address` will be populated using the **Places API (https://location.foursquare.com/developer/reference/places-api-overview)**

(https://location.foursquare.com/developer/reference/places-api-overview). `id` represents the `fsq_id` parameter in the response from the Places API. `name` represents the name of the location. `liked` will represent whether the user has marked the location as one they liked or not. If a user adds a post to their liked locations, the post is then saved into Redis for later use. `userPosted` represents whether this post came from the Places API (false) or if it was a user-posted BoreSquare Location (true).

The `image` in the above schema will take more effort to implement. This will require you to use the `fsq_id` to access the Get Place Photos route in the REST API and combine the `prefix` and `suffix` parameters in the list's **first** response to create the `image` string.

Do not just dump JSON payloads to the frontend, you will lose major points! (Or get fired from BoreSquare!)

Backend

This assignment's backend will be implemented using **Apollo Server (https://www.apollographql.com/docs/apollo-server/)**

(https://www.apollographql.com/docs/apollo-server/) . In particular, you will be to support the following queries and mutations to make this web app functional:

Queries

1. `locationPosts(pageNum: Int) -> [Location]`: You will map **this route in the Places REST API (https://location.foursquare.com/developer/reference/place-search)**

(https://location.foursquare.com/developer/reference/place-search) and **this route for the images (https://location.foursquare.com/developer/reference/place-photos)**

1. **(https://location.foursquare.com/developer/reference/place-photos)** in this query's resolver function to create Location objects from the results of the REST API.
2. `likedLocations -> [Location]`: You will go to redis to retrieve the Location objects that the user has added to his/her liked list.
3. `userPostedLocations -> [Location]`: You will query all Locations that the user has posted.

Mutations

1. `uploadLocation(image: String!, address: String, name: String) -> Location`: This mutation will create a Location and will be saved in Redis. Outside of the provided values from the "New Location" form, by default, the following values of `Location` should be:
   1. `liked`: false
   2. `userPosted`: true
   3. `id`: a uuid
2. `updateLocation(id: ID!, image: String, name: String, address: String, userPosted: Boolean, liked: Boolean) -> Location`: This mutation will take care of any updates that we want to perform on a particular Location
   1. If this location was not previously in the cache, and the user liked it, then add it to the cache using data from React state.
   2. If a location that came from the Places API and was unliked (liked set to false), you should also remove it from the cache.
3. `deleteLocation(id: ID!) -> Location`: Delete a user-posted Location from the cache.

Frontend Routes

Now that we have written our backend queries and mutation, we can now move onto the frontend!

Here, we need to use **Apollo Client (https://www.apollographql.com/docs/react/)**

(https://www.apollographql.com/docs/react/) to interact with our backend to provide data to our frontend.

- `/`
  1. Once a post has been added to the user's bin, the "Like" button should be toggled to "Remove like"
  - Should display a list of location results from the Places API.
  - A user should be able to click a "Get More" button in order to perform another query to get more Locations from the Places API.
  - A user should be able to click on a "Like" button found on a particular image post to "like" a location.
  - Each location post should contain the location name , image, address, and the like/dislike button.
- `/my-likes`
  - Should display a list of locations that the user has previously liked.
  - This page should have similar functionality as the / page.
- `/my-locations`
  1. However, the user should be able to delete their locations from this page.
  2. Users can also only upload new locations from here as well.
  - Should display a list of locations that the user has posted.
  - This page should have similar functionality as the / page.
- `/new-location`
  1. Image URL (we only allow image urls that are already on the internet)
  2. Address
  3. Location name
  - Should render a form that has fields for:

The Senior Developer tells you that since the `/`, `my-likes`, and `my-locations` pages look very similar, he hints that maybe you can create one common React component instead of three to handle list logic and then use wrappers around this component to provide page-specific functionality.

Special Behavior

1. Liking Locations: when a user adds a post to their liked locations, if the post came from the Places API, then you should save the location to the cache. However, if the location is user-posted, then even if the user doesn't add the post to their likes, it should stay in the cache.
2. Any failures, like the failure to like an image, or to delete should fail gracefully. (ie: if it fails, display a message)

Strategy for Completing this Assignment On Time

This assignment is not a trivial assignment, so please start early! Project/Time management is crucial in any company, and BoreSquare is no different!

The following are suggested milestones that you can complete in order to complete this assignment in a timely fashion:

Each section will be estimated in terms of how long each part will take.

A block of time is an arbitrary amount of time, but will serve as an estimate for how long each part will take to complete compared to the others.

1. Figure out how Apollo Server works: 1 block
    - Go to the docs and do their mini tutorial. It should take no more than 10 minutes
2. Finish Backend routes: 3-6 Blocks
3. Finish Frontend without any Apollo Client logic (skip or use dummy data): 6-8 blocks
4. Figure out how Apollo Client works: 1 block
    - Go to the docs and do their mini tutorial. It should take no more than 10 minutes
5. Implement Apollo Client into your React Frontend: 2-3 blocks

Extra Credit!!

So far, we've been using redis as we would perhaps use another other database, so why don't we use a cool feature that redis provides? Back in the BoreSquare offices…

Once you finish the MVP (do not attempt without finishing basic requirements!), your boss asks you if you can implement a new feature for him in return for a one-time bonus, so if you are eager, then read on…

Let's imagine a scene at a feature planning meeting at BoreSquare…

Your boss wants to grow a sense of community in the BoreSquare user base, so he wants to add a feature that allows a user to see how close their liked locations are to their current location. He wants to allow users to see if they are a traveler or not. Therefore, your boss wants you to add a new feature that arranges the content that the user has liked in order of distance.

Here are the changes that you need to make outlined:

Data

The GraphQL type, `Location` will change slightly. We will add a new field called `distance`, which represents the distance from your current location. This field will require you to use the `geocodes` field from the Places API, the Geolocation Web API (or some npm package that uses it), and some algebra to take the coordinates of the location, the coordinates of the device's current location, and convert those two into a measurable distance.

```
type Location {

    id: ID!

    image: String!

    name: String!

    address: String

    userPosted: Boolean!

    liked: Boolean!

    distance: Int!

}
```

Backend

Now, you need to implement a new query function:

- `getTopTenClosestLocations() -> [Location]`: This is the query that will give us the new functionality that our boss wants. This query will look through our liked posts in Redis and return the top 10 posts in terms of smallest distance. Look at the **Z family of functions in Redis (https://redis.io/commands/zrevrange)**
- **(https://redis.io/commands/zrevrange)** to see how to perform a query to get the top and most popular posts

Frontend

To make the life of your QA (Quality Assurance) engineers easier, instead of implementing this new feature as a "Sort by…" UI element, you are asked to implement this feature as a new page at the `/distance` route.

- `/distance`
  - <50 miles (or 80.46 km): Local
  - >=50 miles (or 80.46 km): Traveler
  - Should display the user's top 10 closest locations from his/her bin.
  - Should have the same features as in `/` and `/my-likes`
  - At the top of the page, it should say if the user is a different category if the total of the top 10 image posts is: