Criterion C: Development

This is a documentation of algorithms, data structures, and general strategies approached to build complex and sophisticated methods to address challenges.

Table of Contents

1. Modified R-Trees

Summary / Problem	Maps dedicated to showing spatial information with markers often end up like **Figure 1**. What is desired is an interface like **Figure 2** (final implemention) where the markers are not an unintelligible collection of data that users can't comprehend. However, even with a solution like **Figure 2**, it doesn't gaurantee cleanliness because of what comes out of zooming out of a map **(Figure 3)**.
Innovation / Ingeunity	The solution was to use a hierarchial data structure called R-Trees and modify it for this need. It works by first determining levels of **"zoom-levels"** which determine the hierarchy of each data point. The position of the data markers in each zoom levels are chosen particularly so that never in its zoom level in the U.I. will the map interface be as clustered as **Figure 1**. Essentially, the cleaner set of markers in **Figure 2** can be thought of as each **representing** its **own cluster** of numerous markers. **Seciton 2** explains the innovative way in which this data structure is built and **Section 3** explains why this astronomically increases searching speeds. The ingenuity of this approach is that it eliminates the problem faced by Figure 1, and presents a significantly clearner interface in Figure 2; this is all achieved by this invention of a data structure that inherently presents data more effectively.
Requested Feature / Change from Client	The client was aware of other map-based crime web applications and was disheartened that all of them looked like **Figure 1**. This app needed to be much cleaner than that.
How It Resolves the Need	The data structure is inherently able to present data in a much more efficient way than Figure 1. The items in the hierarchy will never be as clustered because of methods defined in Section 3 to build this data structure.
Feedback Recevied	He was very satisfied that the user interface became like Figure 2 in the end. He said the problem was more or less resolved, especially if these zoom_level definitions were made more precisely.

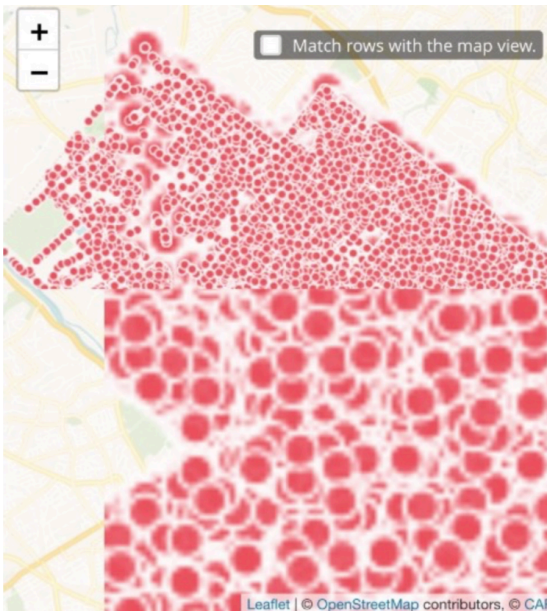


Figure 1

Figure 2

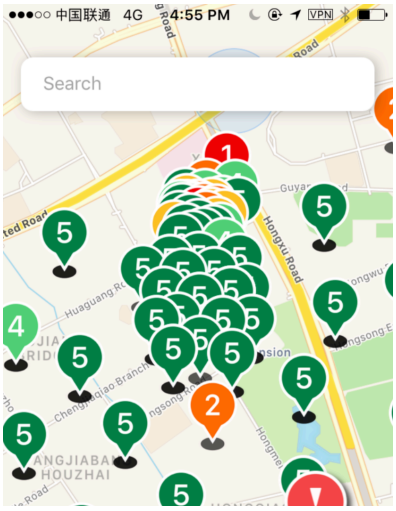
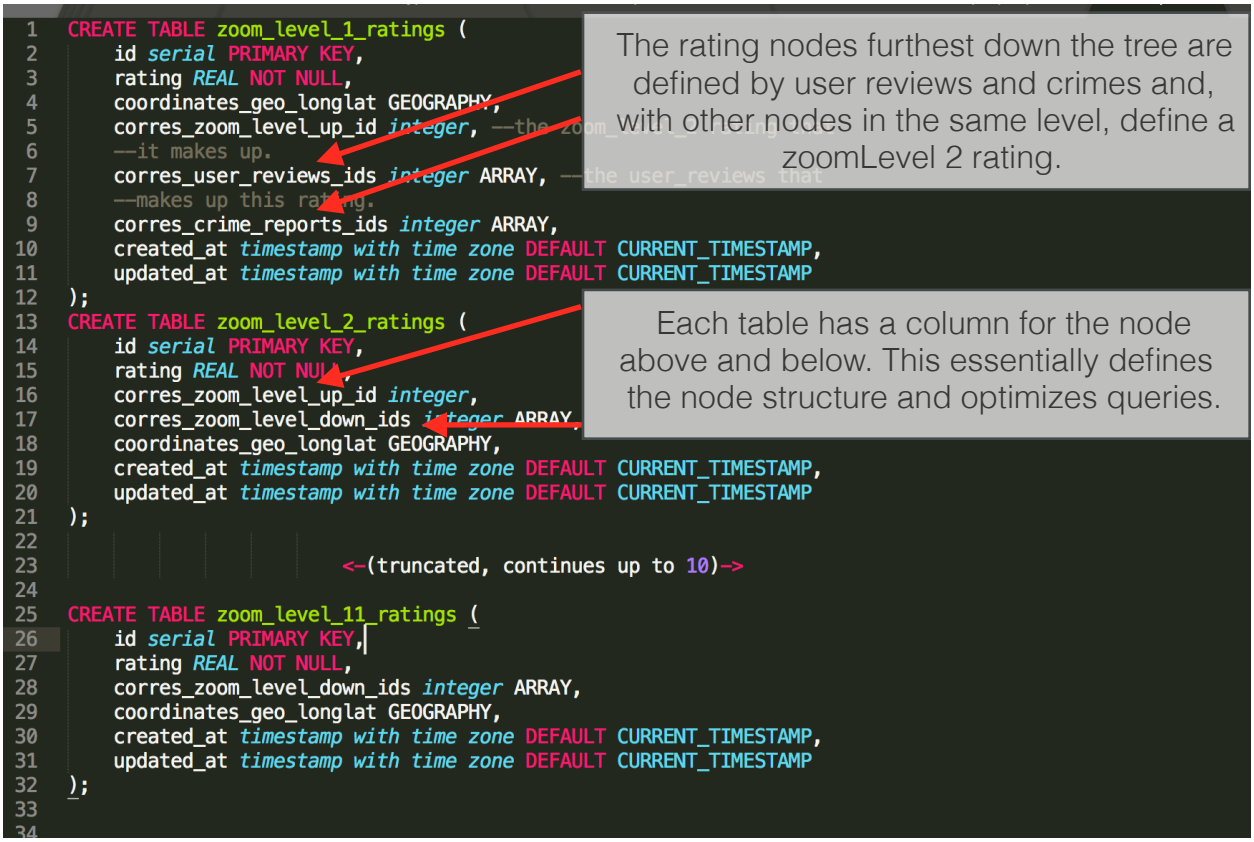


Figure 3 (problem also arises when the user zooms out of the map)

Code Snippet:

PostgreSQL Definition of The Modified R-Tree:

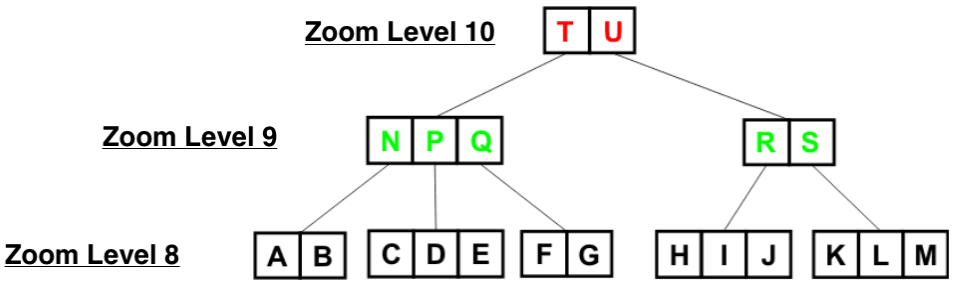



```
CREATE TABLE zoom_level_1_ratings (
    id serial PRIMARY KEY,
    rating REAL NOT NULL,
    coordinates_geo_longlat GEOGRAPHY,
    corres_zoom_level_up_id integer, --the zoom_
    --it makes up.
    corres_user_reviews_ids integer ARRAY, --the user_reviews that
    --makes up this rating.
    corres_crime_reports_ids integer ARRAY,
    created_at timestamp with time zone DEFAULT CURRENT_TIMESTAMP,
    updated_at timestamp with time zone DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE zoom_level_2_ratings (
    id serial PRIMARY KEY,
    rating REAL NOT NULL,
    corres_zoom_level_up_id integer,
    corres_zoom_level_down_ids integer ARRAY,
    coordinates_geo_longlat GEOGRAPHY,
    created_at timestamp with time zone DEFAULT CURRENT_TIMESTAMP,
    updated_at timestamp with time zone DEFAULT CURRENT_TIMESTAMP
);

                    <--(truncated, continues up to 10)-->

CREATE TABLE zoom_level_11_ratings (
    id serial PRIMARY KEY,
    rating REAL NOT NULL,
    corres_zoom_level_down_ids integer ARRAY,
    coordinates_geo_longlat GEOGRAPHY,
    created_at timestamp with time zone DEFAULT CURRENT_TIMESTAMP,
    updated_at timestamp with time zone DEFAULT CURRENT_TIMESTAMP
);
```

2. Optimizing Zoom-In Operation: Modified R-Trees Node Traversing Algorithm

Problem:

Given an example like below:



When the user opens the app at ***zoom_level_10*** and sees ***T, U*** markers, and zoom in within the same window until the user reaches zoom level 8, it would be inefficient to see which of the ***A, B, C, D, E, F, G, H, I, J, K, L, M*** nodes are within the new viewing window of the user, especially if it was ***zoom_level_1***, where there could easily be 100K+ of rows in the table.

Solution:

Instead, the user should traverse down the R-Tree.

Inspiration	Instead of comparing all possible zoom_level_8 nodes, ***A, B, C, D, E, F, G, H, I, J, K, L, M,*** to find out that ***A, B, C, D, E, F, G*** are inside the new viewing window after zooming in (which would especially problematic if it was 100K+ data points instead of just A~M), it traverses the R-Tree.

Tree Traversal Algorithm	(Example where the user zooms in from zoom_level_10 into a zoom_level_8 window, a smaller window where the "T" node is in) 1. Retrieve the serial ID of ***T*** node. 2. Use the ***corres_zoom_level_down_ids*** of the ***T*** node, which is an array that contains the serial IDs of the ***N***, ***P***, ***Q*** nodes (its leaves). 3. Use the ***corres_zoom_level_down_ids*** of each of ***N***, ***P***, and ***Q*** nodes to find that ***A, B*** are children of ***N — C, D, E*** children of ***P — F, G*** children of ***Q***. 4. *Return* ***{A, B} + {C, D, E} + {F, G} = {A,B,C,D,E,F,G}***
Ingenuity	Though this barely makes a difference in the particular small-scale example above, a vast majority of the users will stay in zoom_level_1 and zoom_level_2, which will contain 100K+ nodes. This implies 100Kx4 = ***400K+ iterations*** (x4 for comparisions that determine if a marker is within a window) ***every single time the user zooms in***, unlike what it takes to traverse down the R-Tree. Performance costs are ***significantly*** reduced. It's reduced from thousands of iterations to almost 10. This is possible because of the hierarchial structure inherent to the proposed R-Tree, and the efficient way it traverses the tree.

Code Snippets:

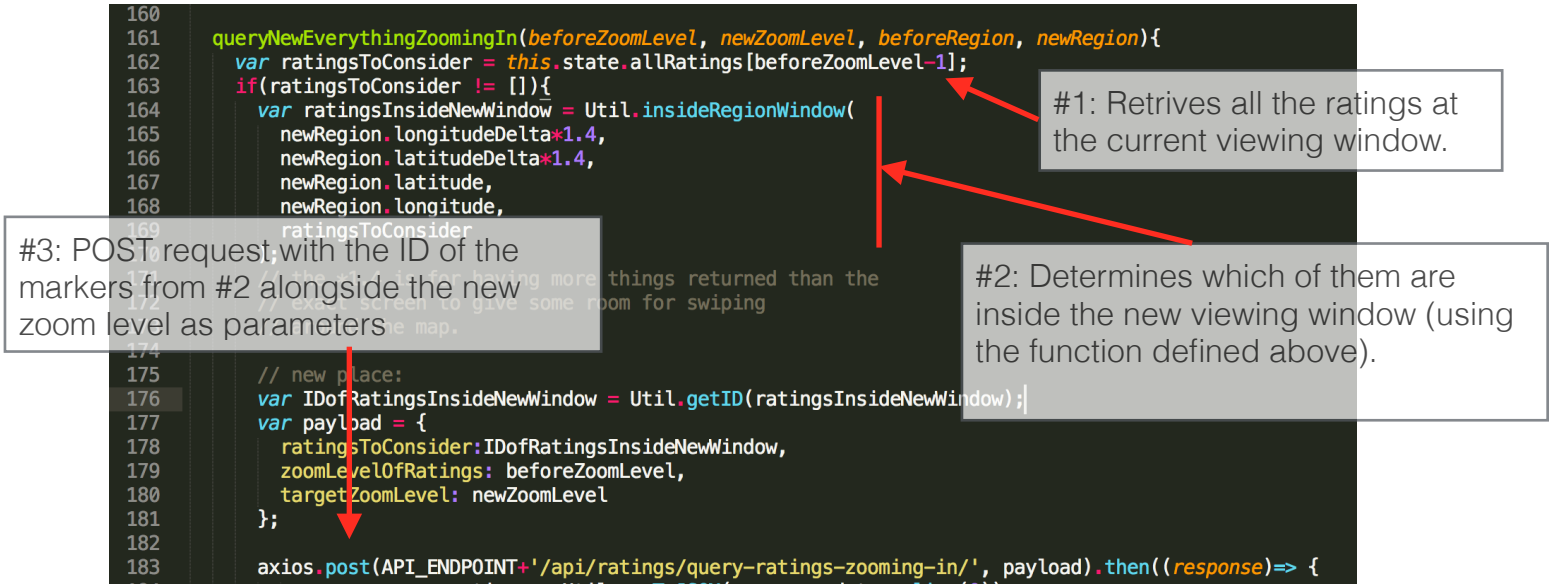
Client Side:
Util.js

```
56
57  function insideRegionWindow(longdelta, latdelta, latitude, longitude, pointArray){
58      var latMax = (latdelta/2)+latitude;
59      var latMin = (latdelta/2)-latitude;
60      var longMax = (longdelta/2)+longitude;
61      var longMin = (longdelta/2)-longitude;
62
63      var array = [];
64
65      for(var ele = 0; ele<pointArray.length; ele++){
66          var element = pointArray[ele];
67          if(element.coordinates_lat_long.latitude < latMax){
68              if(element.coordinates_lat_long.latitude > latMin){
69                  if(element.coordinates_lat_long.longitude < longMax){
70                      if(element.coordinates_lat_long.longitude > longMin){
71                          array.push(element);
72                      }
73                  }
74              }
75          }
76
77      }
78
79      return array;
80  }
81
```

Windows are defined by a center point (latitude, longitude) alongside the distance ± to all four sides (e.g. longitude delta).

After zooming in, this function determines which of the markers in its initial zoom_level will be in the new zoom_level. It does so by using left, right, top, bottom comparisons.

Main.js




```
queryNewEverythingZoomingIn(beforeZoomLevel, newZoomLevel, beforeRegion, newRegion){
  var ratingsToConsider = this.state.allRatings[beforeZoomLevel-1];
  if(ratingsToConsider != []){
    var ratingsInsideNewWindow = Util.insideRegionWindow(
      newRegion.longitudeDelta*1.4,
      newRegion.latitudeDelta*1.4,
      newRegion.latitude,
      newRegion.longitude,
      ratingsToConsider
    );
    // the window for having more things returned than the
    // exact screen to give some room for swiping
    // of the map.

    // new place:
    var IDofRatingsInsideNewWindow = Util.getID(ratingsInsideNewWindow);
    var payload = {
      ratingsToConsider:IDofRatingsInsideNewWindow,
      zoomLevelOfRatings: beforeZoomLevel,
      targetZoomLevel: newZoomLevel
    };

    axios.post(API_ENDPOINT+'/api/ratings/query-ratings-zooming-in/', payload).then((response)=> {
```

The payload from Main.js screenshot is sent via a POST request to the Node.js server which executes a PostgreSQL function (defined in **query.sql**):

Database Side (query.sql):

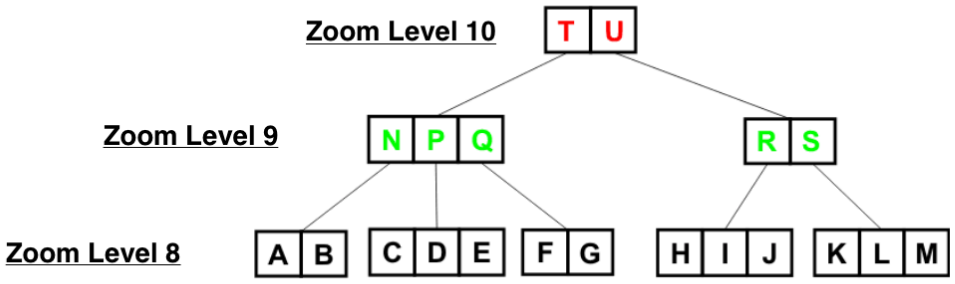
Parameters	
id_of_ratings_considered: int[]	With the graphic example earlier, this contains the uppermose nodes that starts the algorithms: **T, U** nodes.
zoom_level_of_ratings_considered: int	The zoom level of the nodes in id_of_ratings_considered. For the example it's 10.
new_zoom_level: int	The targetted zoom level from which we wan't the nodes. In the example it's 8.

```
32
33  CREATE OR REPLACE FUNCTION new_rating_markers_after_zooming_in (
34      id_of_ratings_considered integer ARRAY,
35      zoom_level_of_ratings_considered integer,
36      new_zoom_level integer
37  )
38  RETURNS TABLE (
39      id integer,
40      rating REAL,
41      coordinates_geo_longlat text
42  ) AS
43  $$
44  DECLARE
45      new_children integer ARRAY := id_of_ratings_considered;
46      iteration integer;
47  BEGIN
48      FOR iteration IN REVERSE zoom_level_of_ratings_considered .. new_zoom_level
49      LOOP
50          IF iteration = new_zoom_level THEN
51              /*
52                  zoom level of ratings considered is just one level
53                  above the target zoom level
54              */
55              RETURN QUERY EXECUTE 'SELECT id, rating, ST_AsText(coordinates_geo_longlat) FROM zoom_level_'||iteration||
56              '_ratings WHERE id=ANY(($1)::integer[])' USING new_children;
57
58          ELSE
59              /*
60                  zoom level of ratings considered is multiple zoom
61                  levels above the targetted. We want to recursively
62                  bring this down.
63
64              */
65              EXECUTE 'SELECT corres_zoom_level_down_ids FROM zoom_level_'||iteration||
66              '_ratings WHERE id=ANY(($1)::integer[])' USING new_children INTO new_children;
67
68          END IF;
69      END LOOP;
70
71  END;
72  $$ LANGUAGE 'plpgsql';
73
```

Output: retrives all the nodes in the desired zoom level (new_zoom_level) whose **parent nodes (or grandparents, or more)** are those with the ID inside id_of_ratings_considered.

#2: Pseudo-recursively ("pseudo" because it uses a linear loop) traverse the hierachy by making the id_of_ratings_considered equal to its leaves, which are parent nodes to the nodes of the layers below to retrieve, until it reaches the desired final lowest layer.

To elaborate on #2, with the example of the alphabet nodes from above:



The **serial IDs** of **T** and **U** nodes are inputted as parameters to the function, as well as their zoom level (**zoom_level_ratings_considered = 10**). It finds the children nodes of them—**N, P, Q, R, S**—and essentially runs the same code as if they were freshly inputted to the function while saying the **zoom_level_ratings_considered = 9** in the new iteration. After retrieving the children of those nodes, then it returns all the desired A~M nodes.

3. Defining/Propogating The Modified R-Tree Seamlessly With New Data

Context	
The Need (Context)	Perhaps the greatest challenge in building this data structure. The entire system must react to every new review or crime report that is inputted by a user; each new review/report must add to **some** node of the tree and those changes must propogate to the rest of the leaves below. This is because the ratings of the parents depend on the ratings of its leaves. Contributions of leaf nodes to a parent node want to be relevant. A marker in Brazil shouldn't affect a marker in the United States. How is this determined seamlessly?
Issues/Questions	How to know when to create a new upper node? How to create a new upper node in the existing tree? How to propogate the change?
Algorithm	1. A user_review is inputted. It does a search to find the closest **existing zoom_level_1 node** that it can contribute to. 2. **IF (THE NODE <u>IS NOT</u> CLOSE ENOUGH):** ***create*** a **new** node in zoom_level_1 at the position and rating of the new review. Do a search to find the closest zoom_level_2 node that this new zoom_level_1 rating can contribute to. Repeat this up to level 10, until there is an iteration where there is a sufficiently close node at the layer above. 3. **ELSE (THE NODE <u>IS</u> CLOSE ENOUGH):** (still using the example of a user_review contributing to zoom_level_1) The identified zoom_level_1 node that is close enough is updated according to the new user_review. Update the rating in the zoom_level_1 node according to the new user_review and propogate the change in ratings upwards the tree.
Client's request / Feedback from client's request	This algorithm is only supplementary to the rest of the program, and does not alone meet a specific need of the client.

```
ELSE

    /*
    The closest zoom_level_1 rating is too far away from the
    user review. I put down a review for a street in downtown Shanghai
    and it contributes to a rating for a suburb in SuZhou...

        1. Put down a user_review at that coordinate.
        2. Create a rating at zoom_level_1 at that coordinate.
            Where all it makes up is the rating in the user_review.
        3. Populate to all the upper zoom levels.


    */

    INSERT INTO user_reviews (
        rating,
        coordinates_geo_longlat,
        comments,
        from_user_id,
        country,
        corres_zoom_level_1_id
    ) VALUES (
        rating_in,
        coordinates_geo_longlat_in,
        comments_in,
        from_user_id_in,
        country_in,
        0
    ) RETURNING id INTO new_below_node_id_after_insert;

    UPDATE user_infos
        SET reviews_id_arr = reviews_id_arr || new_user_review_id
        WHERE id=from_user_id_in;

    INSERT INTO zoom_level_1_ratings (
        rating,
        coordinates_geo_longlat,
        corres_user_reviews_ids,
        corres_crime_reports_ids,
        corres_zoom_level_up_id
    ) VALUES (
        rating_in,
        coordinates_geo_longlat_in,
        ('{'||new_below_node_id_after_insert||'}')::integer ARRAY,
        '{}',
        1000
    ) RETURNING id INTO new_upper_node_id_after_insert;

    /*
    Putting the ID of the new row in zoom_level_1 to the
    corresponding row in user_reviews
    */
    UPDATE user_reviews as this_review
        SET corres_zoom_level_1_id=new_upper_node_id_after_insert
        WHERE this_review.id=new_below_node_id_after_insert;

    new_below_node_id_after_insert:=new_upper_node_id_after_insert;



    /*
        Now populating all the upper nodes...
        Same considerations of if there is a close enough point
        nearby still applies.
    */

    SELECT id, coordinates_geo_longlat
        INTO id_to_upper_node, closest_potential_note_coordinates_lat_long
        FROM zoom_level_2_ratings as rate
        ORDER BY rate.coordinates_geo_longlat <-> coordinates_geo_longlat_in LIMIT 1;


    /* If there isnt a marker in zoom_level_2 that is close enough. */
    IF closest_potential_note_coordinates_lat_long IS NULL OR ST_DISTANCE(coordinates_geo_longlat_in, closest_potential_no
        INSERT INTO zoom_level_2_ratings (
            rating,
            coordinates_geo_longlat,
            corres_zoom_level_down_ids,
            corres_zoom_level_up_id
        ) VALUES (
```

This is the **ELSE** to the **IF** statement from above, where there **ISN'T** a zoom_level_1 node that is close enough (imagine a user review contributing to a rating marker that is 100km away; it should just **create** a new node at that point). **INSERT INTO** user_reviews is standard & updates the user profile to update the list of reviews.

Then the code **INSERTS** into **zoom_level_1_ratings** the **coordinates** of the new user_review. This is where it is **creating** the marker at the new place that is too far away from existing nodes.

The **INSERTed** user_review needs to know which **zoom_level_1** it corresponds to. This is why it **stores** the **new** node **id** and **UPDATE** user_reviews.

What if, after creating a new zoom_level_1 node because the **new** user review is **too far** away from nearby existing node, but the new **zoom_level_1** node is ***close enough*** to a nearby **zoom_level_2** rating node?

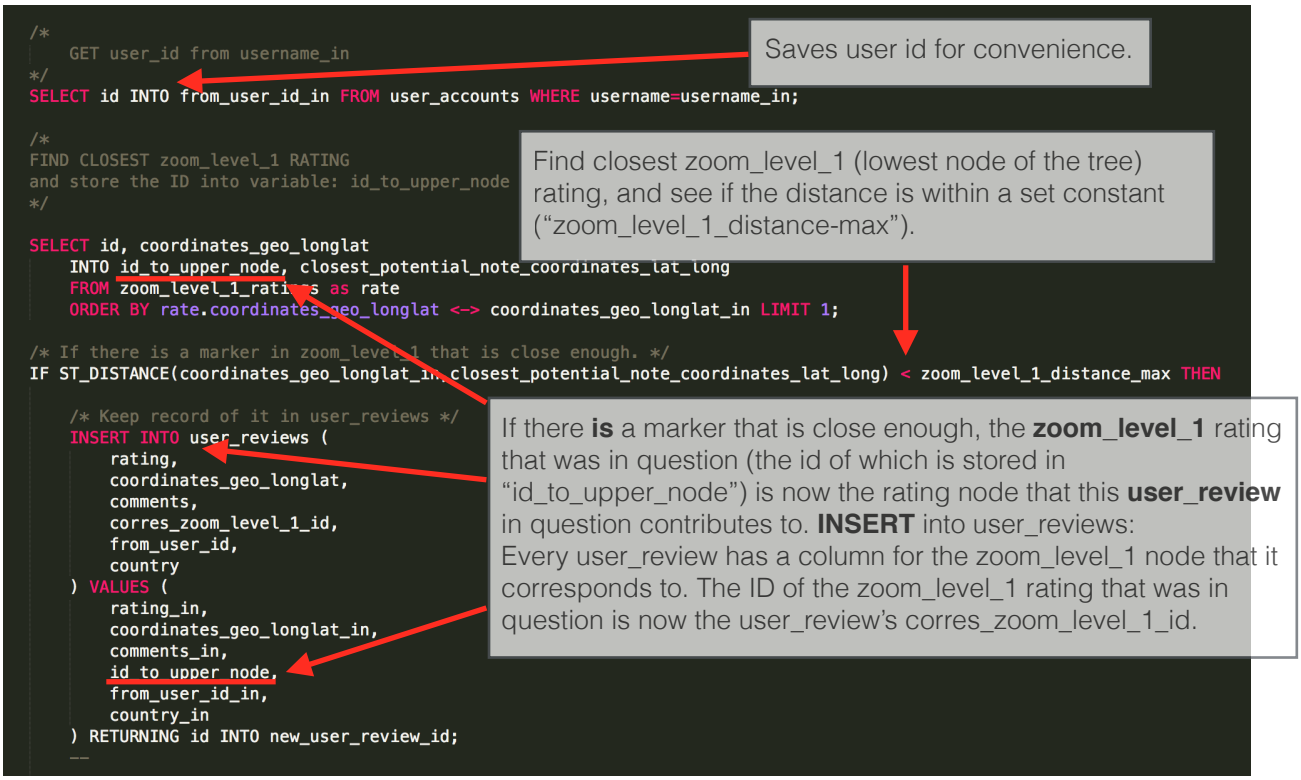
This is the reason behind the 2nd IF statement below, looking for a node in **zoom_level_2** that is close enough. The process iterates itself again.

Context

Ingenuity

1. Solves the issue where a zoom_level_1 rating is determined by a review that is too far away for it to be relevant. It catches for this by rather making a new rating in Brazil instead of contributing to the rating of an irrelevant area.
2. Speed: This algorithm takes far less than a second to run; with geo-spatial indexing, it's O(40).
3. It builds the hierarchy robustly and accurately; with this comes the powerful techniques explained in #1 and #2.

NOTE: The PostgreSQL function that implements this algorithm is over 700 lines long. The following show explanations of the crucial pieces that constitute the crux of the algorithm.



```
/* UPDATE USER_ACCOUNTS to add the new review into their profile. */
UPDATE user_infos
    SET reviews_id_arr = reviews_id_arr || new_user_review_id
    WHERE id=from_user_id_in;


/*

Update zoom_level_1 ratings
-- Two things:
-- Update the user_reviews_ids array to add user reviews
-- Change the rating.

*/

UPDATE zoom_level_1_ratings AS this_node
    SET corres_user_reviews_ids = this_node.corres_user_reviews_ids || new_user_review_id,
    rating = calculate_rating_from_user_reviews_and_crime_reports(corres_user_reviews_ids ||
        new_user_review_id, corres_crime_reports_ids),
    updated_at = current_timestamp
    WHERE id=id_to_upper_node
    RETURNING corres_zoom_level_up_id, rating
        INTO id_to_upper_node, rating_of_current_node;


/* Then populate the changes to all the nodes above. */

PERFORM propagate_updates_upwards(rating_of_current_node, 2, id_to_upper_node);
```

User profiles have columns that is an array of all the reviews

After determining the right parent node, it updates the parent node with a new rating (based on the addition of the new node) and the list of leave nodes of the parent node.

```
722
723  CREATE OR REPLACE FUNCTION propagate_updates_upwards(
724      rating_in integer,
725      first_table_zoom_level integer,
726      id_to_upper_node_in integer
727  ) RETURNS void AS
728  $$
729  DECLARE
730      id_to_upper_node integer := id_to_upper_node_in;
731      rating_of_current_node integer := rating_in;
732  BEGIN
733      FOR iteration IN first_table_zoom_level .. 11
734      LOOP
735          IF iteration != 11 THEN
736              EXECUTE 'UPDATE zoom_level_' || iteration || '_ratings AS this_table' ||
737                          ' SET rating=calculate_rating_for_higher_zoom_levels(this_table.rating,
738                          array_length(this_table.corres_zoom_level_down_ids, 1), $1), updated_at=current_timestamp' ||
739                          ' WHERE id=$2 RETURNING corres_zoom_level_up_id, rating'
740                          USING rating_of_current_node, id_to_upper_node_in
741                          INTO id_to_upper_node, rating_of_current_node;
742          ELSE
743              EXECUTE 'UPDATE zoom_level_11_ratings AS this_table' ||
744                  ' SET rating=calculate_rating_for_higher_zoom_levels(this_table.rating,
745                  array_length(this_table.corres_zoom_level_down_ids, 1), $1), updated_at=current_timestamp' ||
746                  ' WHERE id=$2' USING rating_of_current_node, id_to_upper_node;
747          END IF;
748      END LOOP;
749  END;
750  $$ LANGUAGE plpgsql;
751
```

This function works for any node at any level; whatever node it is in whatever level, it uses a loop to define the operation to propogate the change.

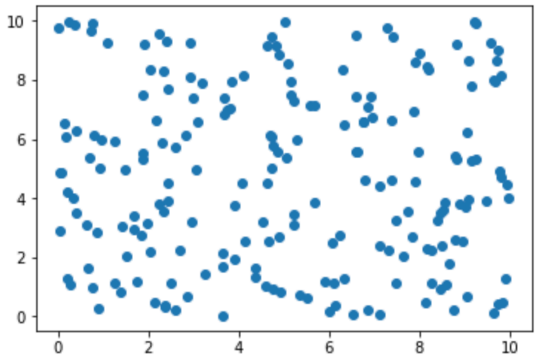
A rating of a node is determined by the ratings of its children nodes. So this loop retrieves the leaves, appends the new rating into that list, and calculates the rating altogether.

The loop terminates at updating the rating at the uppermost node, the corresponding node at zoom_level_11.

The remaining part of the algorithm will not be specified further.

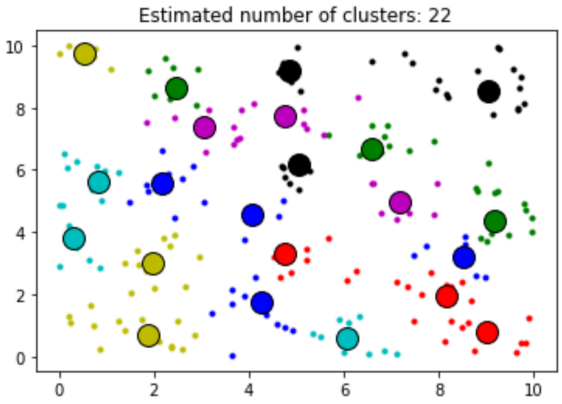
4. Making Current Data More Presentable: Mean-Shift Clustering Algorithm

Given a set of data like the following:



With the current method, we may have some upper nodes prematurely defined. But these may not be the best clusters to represent the set of data.

Ultimately, the objective is to “clean up” the geospatial database periodically to make it so that the algorithmically generated ratings marker positions that are pre-maturely defined are more presentable and better representative of the data. So we want to end up with:



Which is fairly reasonable.

This uses the Mean-Shift algorithm:

The algorithm considers the feature space following a probability density function. For instance, given an input (set of data), it considers them as part of the distribution. The dense regions of the space, the desired clusters, they correspond to the "mode" (local maxima) of the distribution.

For each data point:

Mean Shift Process
1. Define a window around each data point (selected randomly initially). 2. Compute the mean of the data points within the window. 3. The next data point considered is at the mean discovered at the previous step.

Kernal Density Estimation is used to estimate the aforementioned density function:

$$\hat{f}(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

Where K is the kernel, and the Gaussian kernel was used:

$$\phi(x) = e^{-\frac{x^2}{2\sigma^2}}$$

The maxima of any function is found (mode of the density function in this case) when the gradient is zero. Setting the gradient to zero, we obtain:

$$\mathbf{f}(\mathbf{x}) = \sum_{n=1}^{N} \frac{K'\left(\left\|\frac{\mathbf{x}-\mathbf{x}_n}{\sigma}\right\|^2\right)}{\sum_{n'=1}^{N} K'\left(\left\|\frac{\mathbf{x}-\mathbf{x}_{n'}}{\sigma}\right\|^2\right)} \mathbf{x}_n$$

And the vector f(x)-x is the "mean-shift".

The algorithm sets x to f(x) until f(x) converges (i.e. the gradient is zero).

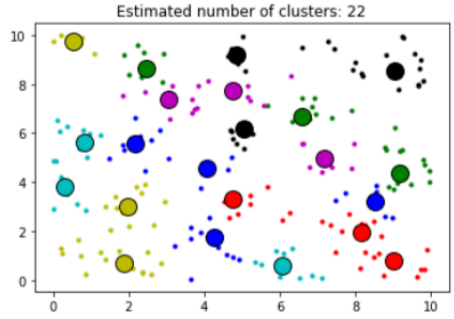
In [49]:
```
from sklearn.cluster import MeanShift, estimate_bandwidth

X = []
for iter in range(0,len(data_x)):
    X.append([data_x[iter],data_y[iter]])

X = np.asarray(X)

bandwidth = estimate_bandwidth(X, quantile=0.1, n_samples=20)


ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
ms.fit(X)
labels = ms.labels_
cluster_centers = ms.cluster_centers_

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)

print("number of estimated clusters : %d" % n_clusters_)

# ##############################################################################
# Plot result
import matplotlib.pyplot as plt
from itertools import cycle

plt.figure(1)
plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    my_members = labels == k
    cluster_center = cluster_centers[k]
    plt.plot(X[my_members, 0], X[my_members, 1], col + '.')
    plt.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=14)
plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

Generate Fake Data

Define Bandwidth, quantile, number of samples (number of clusters).

Visualize clusters with appropriate coloring.

```
number of estimated clusters : 22
```



5. Modularity Using React.js Components for Iteratively Generated Interfaces

Summary / Problem	User interface features such as Figure 4 require repitition; however, the code for the class for the **list** of feed items become unorganized and harder to maintain when the code for designing everything for a feed item is defined in the same list class.
Innovation / Ingeunity	The solution was to use components; a feed item is defined as its own component, and the feed list class simply creates new feed item components. It needs to only concern itself with the providing it with the right data in the right format, nothing about what it takes to create one feed item. This abstraction improves maintainence as all feed items are defined separately from the feed list class, so I could focus my attention in one thing at a time. In the future, if I wanted to change how a feed item design, I don't need to change the entire class that deals with feeds, I need only to alter the feed item class.

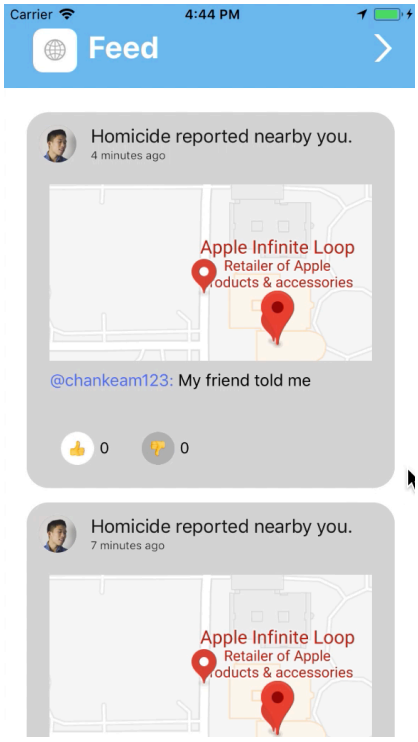


Figure 4

Code Snippets:

NotificationView.js

Figure 4

```
componentDidMount(){
    this.retrieveItem('userObj').then((obj) => {
        if(obj && obj.accessToken){
            axios.post(API_ENDPOINT+'/api/feed/query-feed-first', {
                username: obj.username,
                accessToken: obj.accessToken
            }).then((response) => {
                this.setState({data:response.data});
            }).catch((err) => {
                if(err.response.data.error=='TokenExpiredError'){
                    this.props.accessTokenExpired();
                }
            });
        }else{
            this.props.noAccessToken();

        }
    }).catch((err) => {
        console.log(err);
    });
}
```

Retrieves the data from the server.

```
renderData(){
    if(this.state.data){
        return (
            <FlatList
                data={this.state.data}
                style={styles.listContainer}
                renderItem={({item}) => <OneNotification data={item}/>}
                refreshing={this.state.refreshing}
                onRefresh={() => this.handleRefresh()}
            />
        );
    }
}
```

"Data" is an array, and the renderItem returns a list item with each element in the data array.

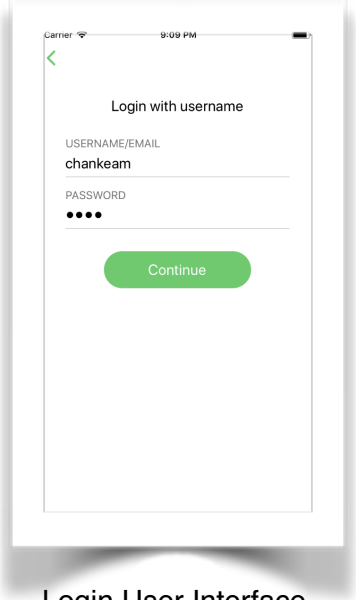
This is all this class does to generate each list item. It concerns itself with none of the logistics of defining the UI and functionalities. It just calls the component.

This imports the OneNotification component. It handles all operations for each feed item.

```
import OneNotification from './OneNotification/OneNotification';
```

6. WebToken Authorization Protocol

Summary / Problem	How do we know if a user is "logged in"? How do we know if each user-specific API calls are truly from a particular user? ***NOTE:*** This protocol makes use of an NPM library called **jsonwebtoken**.
Innovation / Ingeunity	1. The server generates a JWT Token with every successful login attempt. 2. This JWT Token is passed along with the user's username with every user-specific API call (like submitting a rating). It's saved in local storage for the client, and the client retrieves it from local storage at every API call. 3. After 30 days the JWT Token expires. In the client-side, it will log out and ask the user to login again. This way, it is able to securely identify that the request coming from a certain username is indeed coming from the user it claims to be. The experiation date prevents malicious attempts perpetrated by using another user's JWT token while knowing their username. It also simplifies the process of putting out the API as a publically available service.



Login User Interface

LoginEmail.js (client-side app code)

```
}else{
    //this is username
    this.setState({checkingTrue:true});
    axios.post(API_ENDPOINT + '/api/auth/login/username', {
        username:this.state.username,
        password: this.state.password
    })
    .then((response) => {
        if(response.data.errorCode == 'u1'){
            this.setState({accountWrong:true});
        }else{
            if(response.data.accessToken){
                console.log("has accesstoken");
                this.setState({checkingTrue:false});

                this.storeItem("userObj", {
                    accessToken: response.data.accessToken,
                    username: this.state.username,
                    firstname: response.data.firstname,
                    lastname: response.data.lastname,
                    email: response.data.email,
                    phoneNumber: response.data.phoneNumber
                }).then(() => {
                    const {navigate} = this.props.navigation;
                    navigate('Main');
                });

            }else{
                this.setState({accountWrong:true, checkingTrue:false});
            }

        }
    })
```

Makes an API call to the server to request login.

Error handling.

If login is successful, the client will receive an web access token. It calls a function that handles it.

AuthController.js (server-side Node.js code)

```
if(result.rows[0]){
    bcrypt.compare(req.body.password, result.rows[0].password, (err, res_bool) => {
        if(err){
            res.status(500).json({
                errorCode:'b1',
                message:'Error in bcrypt?'
            });
        }else{
            if(res_bool){
                // success! Issue accessToken

                var token = jwt.sign({
                    username: req.body.username
                }, JWTSECRET, {
                  expiresIn: 86400 * 30 // expires in 24 * 30 hours (30 days)
                }, (err, token) => {
                    if (err) {
                        res.status(500).json({
                            errorCode: 'b1',
                            message:'jwt failure',
                            message: err
                        });
                    }
                    // result.rows[0].password
                //  firstname: response.data.firstname,
                // lastname: response.data.lastname,
                // email: response.data.email,
                // phoneNumber: response.data.phoneNumber
                    res.status(202).json({
                        accessToken: token,
                        firstname: result.rows[0].firstname,
                        lastname: result.rows[0].lastname,
                        email: result.rows[0].email,

                    });
                });
            }else{
```

Compares the hash of the inputted login password to the hash stored in the database.

Signs a new json web token with the correct credentials.

Returns to the client a valid accesstoken with 202 status.

Sample use of JWT AccessToken—submitting a crime report:

RatingModal.js (client-side app code, submits crime report). *Please note that crime reporting is a user specific process; each crime has an identifiable user that reported it.*

```
89      }
90      crimeSubmitted(){
91
92          //      req.body.username,
93          // req.body.accessToken,
94          // req.body.TOC,
95          // req.body.coordißates_lat_long,
96          // req.body.comments,
97          // req.body.countryIn ----- THE FRONT END PROVIDES THE COUNTRY!!
98
99          console.log(API_ENDPOINT+'/api/crimes/user/report-crime');
100         this.setState({showLoading:true});
101         this.retrieveItem('userObj').then((obj) => {
102             if(obj.accessToken){
103                 axios.post(API_ENDPOINT+'/api/crimes/user/report-crime', {
104                     username: obj.username,
105                     accessToken: obj.accessToken,
106                     TOC:this.state.selectedTOC,
107                     coordinates_lat_long: this.state.coordinates_lat_long,
108                     comments: this.state.comment,
109                     countryIn: this.state.country
110                 }).then((response) => {
111                     console.log(response);
112                     this.props.ratingSubmitComplete(); //closes the modal
113                     this.setState({showLoading:false}); // show success message!
114                 }).catch((err) => {
115                     console.log("ERROR IN ERR");
116                     console.log(err.response);
117                     if(err.response.data.error=='TokenExpiredError'){
118                         this.props.accessTokenExpired();
119                     }
120                 });
121             }else{
```

Retrieves the accesstoken from local storage to use it in the API call to the server.

(Error handling) Checks if the accesstoken exists.

Makes the user-specific API call to report crime at a location. It passes in "username" to claim that it's from that user. The accesstoken verifies this claim.

middlewares.js (server-side code that verifies a JWT token)

```
/*
req.body.accessToken
req.body.username
*/
jwt.verify(req.body.accessToken, JWTSECRET, (err, result) => {
    if(err){

        console.log(err.name);
        console.log(err);
        res.status(403).json({
            message:'access invalid',
            error:err.name
        });
    }else{
        var decodedUsername = result.username;
        if(decodedUsername != req.body.username){
            res.status(403).json({
                message: 'username invalid',
                receivedUsername:req.body.username,
            });
        }else{
            next();
        }
    }

});
```

Uses a Node.js API to verify if the inputted accesstoken is valid, according to the long secret string called **JWTSECRET**.

Identifies that the accesstoken belongs to the username of the person the API call is claiming to be. The JWT token contains the username so it can compare as such.

Allows the rest of the code for submitting a crime report to continue, now knowing that the user is the valid user (specified in the subsequent **Section 7**).

7. Node.js Middlewares

Summary / Problem	Many user-specific API calls to the Node.js servers should never do anything before authenticating the user. This requires a barrier from running the rest of the code before authentication, where this authentication process needs to be universal across the server-side code.
Innovation / Ingeunity	The solution is to use **middlewares**, which are defined in a separate file, separate from all the other server code. One of these middlewares is the authentication protocol and it does not let the rest of the code operation for the particular API endpoint continue without authenticating first. This improves **maintanence** and increases development time by upgrading **modularity**; **none** of the API endpoints need to **concern itself** with authentication procols and definitions of it is not copied and scattered across the server code. If the authentication protocol needs to be changed, I only need to change one file instead of changing the entire system. It also solves the issue of not allowing the rest of the operation to continue. There's a "**next()**" comment that allows the server code to continue, which gives me control to specify when NOT to continue (e.g. authenetication fails).
Request from client / Feedback from client	This is simply supplementary to other componets of the system and does not directly address a request from the client. It contributes to solving the problem of needing a secure authentication protocol for the user.

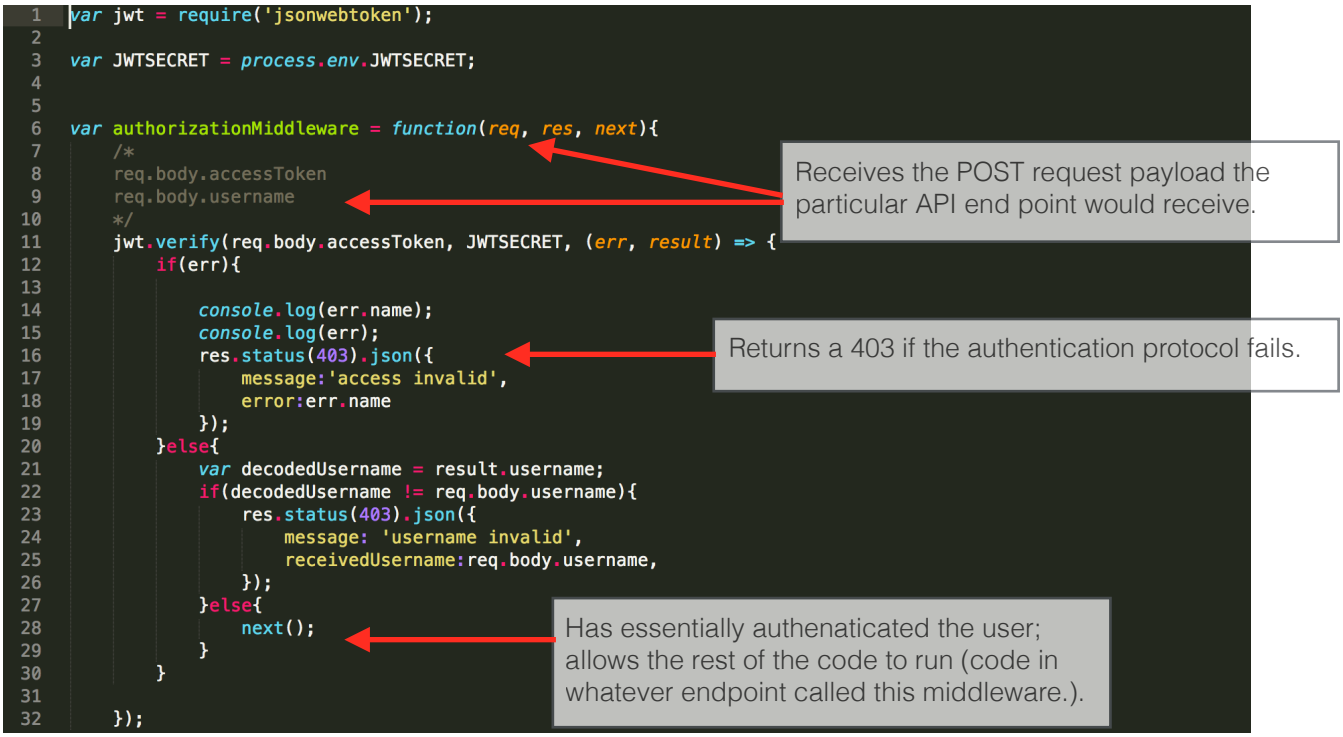
Code Snippets:

Ratings.js (server-side code handles rating).

```
222
223 router.post('/input-rating', middlewares.authorizationMiddleware, (req, res) => {
224
225
226     var SQL_QUERY = `
227         SELECT input_user_review_and_update_user_profile(
228             ${req.body.rating},
229             ST_MakePoint(${req.body.coordinates_lat_long.longitude}, ${req.body.coordinates
230             '${req.body.comments}',
231             '${req.body.country}',
232             '${req.body.username}')
233     `;
234     pool.query(SQL_QUERY, (err, result) => {
235         if(err){
236             console.log(err);
237             res.status(500).json({
238                 message:'database error'
239             });
240         }else{
241             res.status(202).json({
242                 message:'success'
243             })
244         }
245     });
```

Calls the authorizationMiddleware to authenticate the user before proceeding.

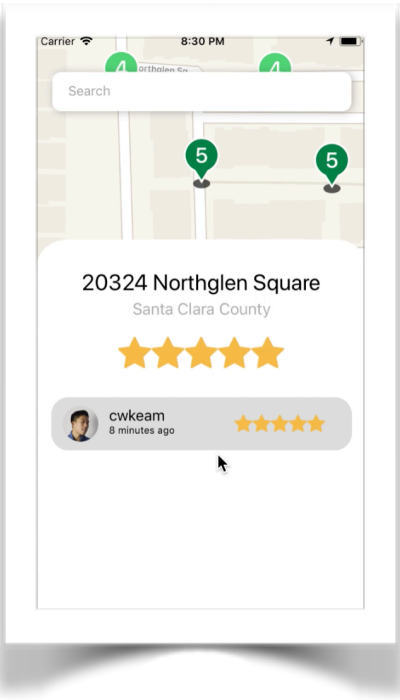
Middlewares.js

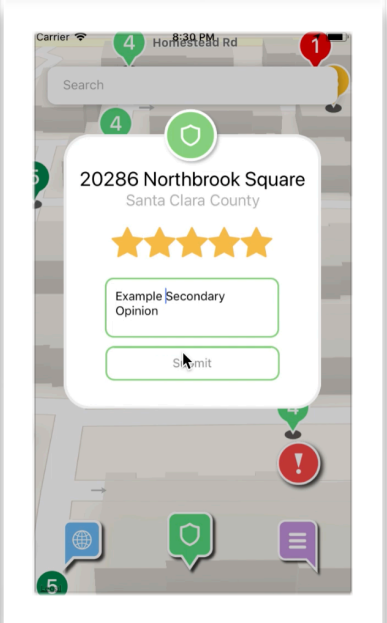


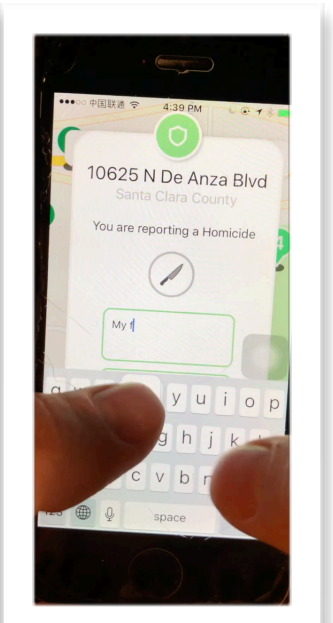
The specified middleware is called before running the rest of the code in the endpoint. Once the middleware reaches next(), the rest of the code is able to run.

8. Proxy API Endpoint with Google Maps

Summary / Problem	Refer to figures below; the app needs to display the country, city, and road address of where a user is submitting a rating to and where a certain rating marker is rating. This is most easily done with Google maps API. However, the implementation had to extend beyond making a simple call to the Google Maps API. 1. Extra formatting is required with the returned data from Google Maps. 2. Google Maps API is blocked in certain countries due to firewalls.
Innovation / Ingeunity	My remote Node.js server makes the call to the Google Maps API instead. The client that requires the Google Maps data makes a call to my server, and the server makes the call to the Google Maps API, processes the data, and returns it to the user. The ingenuity in this approach: 1. Any user from any country will receive a valid response about the locations for my server is not blocked anywhere. 2. My server's connection to the Google Maps API can be optimized; this can't be done with every individual client-end mobile phones. 3. The user-end has minimal requirement to process any information; it directly presents what it receives. We can't ensure that every client-side mobile device will be a high powered computer to deal with extra overhead but we can ensure my remote Node.js server is.
Request from client / Feedback from client	After looking at apps like Waze, the client really wanted to be able to see the exact road address of where a crime was reported and where there is a rating marker; simply looking at a map sometimes does not bring adequate information. This implemention fulfilled this requirement completely. The client was very content and especially pleased because it addressed an issue (the firewall problem) that he hadn't even thought of.







This section makes use of the Google Maps external API: https://maps.google.com/.

Code Snippets:

(Client Side) RatingModal.js

```
componentDidMount(){
    var self = this;
    axios.post(API_ENDPOINT+"/api/util/get-address-from-coordinates-lat-long", {
        coordinates_lat_long:[this.state.coordinates_lat_long.latitude, this.state.coordinates_lat_long.longitude]
    })
    .then((response) => {
        self.setState({
            address:  response.data.address,
            city: response.data.city,
            country: response.data.country
        });
    })
    .catch((err) => {
        console.log(err);
        console.log("ERROR! ", err.response);
    });
}
```

As the component renders, it makes a POST request to my API endpoint (defined by a global constant API_ENDPOINT) with the necessary lat,long data.

Sets the address, country, and city presented in the UI.

(server side – where the endpoint is defined) Util.js

```
29
30  router.post('/get-address-from-coordinates-lat-long', (req, res) => {
31      var link = 'https://maps.googleapis.com/maps/api/geocode/json?latlng='
32      + req.body.coordinates_lat_long[0].toString()
33      + ',' + req.body.coordinates_lat_long[1].toString()
34      + '&key=' + GOOGLE_MAPS_API_KEY;
35      axios.get(link)
36          .then(function (response) {
37              var addressComponents = response.data.results[0].address_components;
38              var address = addressComponents[0].short_name + ' ' + addressComponents[1].short_name;
39              var city = addressComponents[3].short_name;
40              var country = addressComponents[5].short_name;
41              res.send({
42                  address,
43                  city,
44                  country
45              });
46          }).catch((err) => {
47
48              console.log(err);
49          });
50  });
51
```

Makes a GET request to the Google Maps API where parameters are passed in as the URL.

Formats the address, city, and country in a convenient JSON format and returns it to the client.

(Word Count: 752)