Software Design Specification

for

PDQ Routing Subsystem

Version 1.5

Antonio Workman 190907
Joshua Alkins 190908

1/3/2021

Table of Contents

1. Introduction

1.1. Purpose of Document

This document contains the design decisions for the Pizza Delivered Quickly Routing Subsystem, prepared for the stakeholders. The technical specification has been drafted following the meeting carried out between the development team and the stakeholders. The project will design and implement a map navigation system that allows the user to travel from the designated factory to the customer along the fastest route available.

1.2. Scope of Product

The purpose of this requirement specification is to:

- Serve as a repository of the requirements governing the Routing Subsystem. These requirements were discussed and determined by the team after discussing with the project sponsor. These requirements will be used throughout the completion of the project as a base for validation and verification of the project management documents. The intended audience of this document includes the clients and the team members of the PDQ project.
- This document may be updated or revised throughout the duration of the project to accommodate any changes to the project requirements and scope.
- The requirements specified in this document represents the objectives our team plans to implement in the creation of Routing Subsystem.

1.3. Overview of Document

PDQ is creating a routing application to direct delivery divers between pizza factories, where pizzas are made, and customers' delivery addresses. This project aims to increase average delivery speed and decrease variation in delivery times. The goal is to deliver ready to cook pizza in 30-minutes and pre-cooked pizzas in 45-minutes from order entry. This is all in order to compete with competing delivery services, who offers a 45-minute or less delivery policy. This routing application will take the form of a GPS system that will be installed in all delivery driver's vehicles.

This document will describe the architecture of the system, major components of the system, and the design decisions made during the project and the rationale behind them.

2. System Architecture Description

The PDQ order entry and delivery system consists of 6 subsystems that interact with each other and customers, pizza factories, delivery drivers and administrators. The Routing subsystem interacts with the Logistics subsystem, which sends orders to be delivered to the routing subsystem. The routing subsystem also interacts with drivers to show directions and administrators to add new drivers to the system.

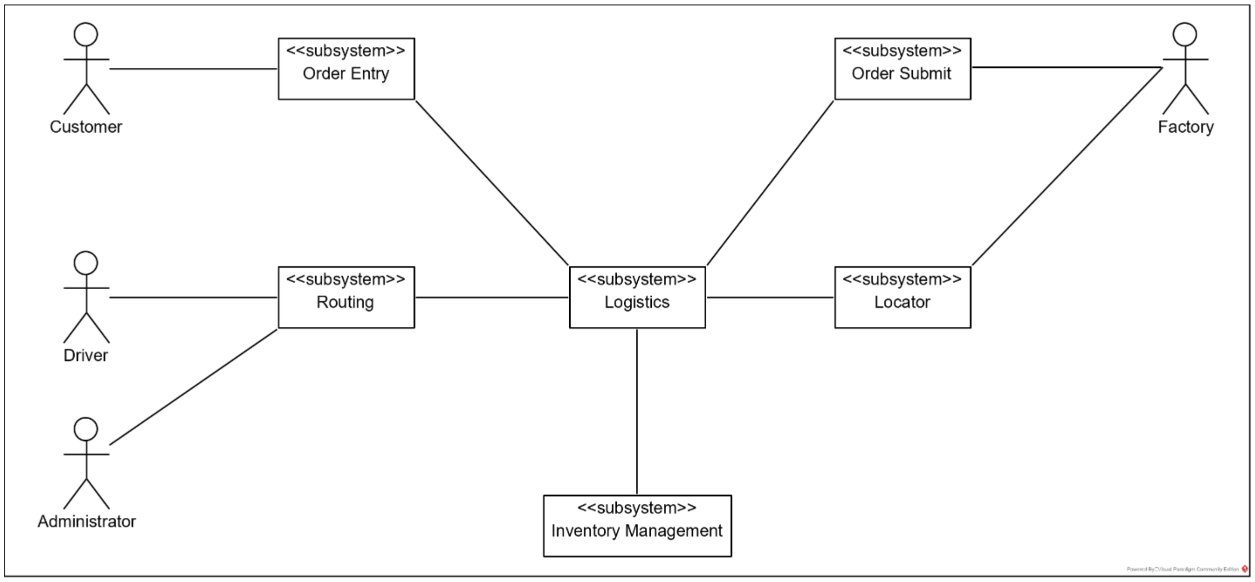


Figure 2.1. PDQ order entry and delivery management system subsystems and interactions.

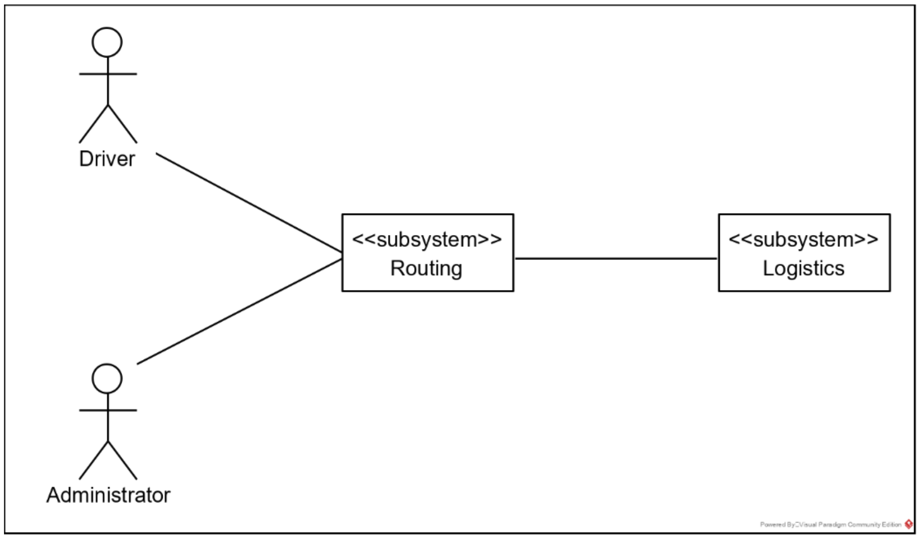


Figure 2.2. Routing subsystem context.

2.1. Overview of Modules/Components

The routing subsystem consist of 3 major components: the routing app, the routing server, and a database. The app will run on delivery driver's mobile devices and is responsible for interacting with drivers and displaying directions. The server is responsible for distributing information to other components and allowing administrators to access driver data. The database is responsible for storing data on administrators, drivers, and deliveries.

The app consists of 3 major components:

- **Login & Security –** verifies driver credentials and grants access to the rest of the system.
- **Server Interface –** responsible for sending and receiving information from the server.
- **Map Service –** responsible for displaying local area map and plotting directions on to the map.

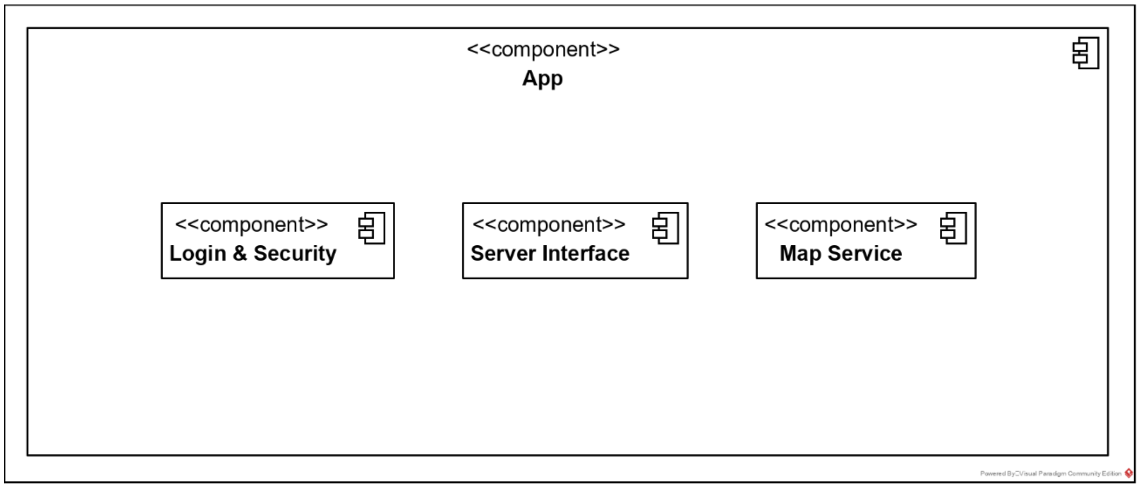


Figure 2.1.1. App components.

The Server has 5 major components:

- **Login and Security –** responsible functions for ensuring the API and admin access is not accessed by unauthorized parties.
- **Database Access –** responsible for all interactions with the database.
- **Admin Webpages –** contains the HTM, JavaScript, CSS, and routing scripts.
- **Delivery API –** contains all routing functions for the API handling delivery data.
- **Driver API –** contains all routing functions for the API handling driver data.

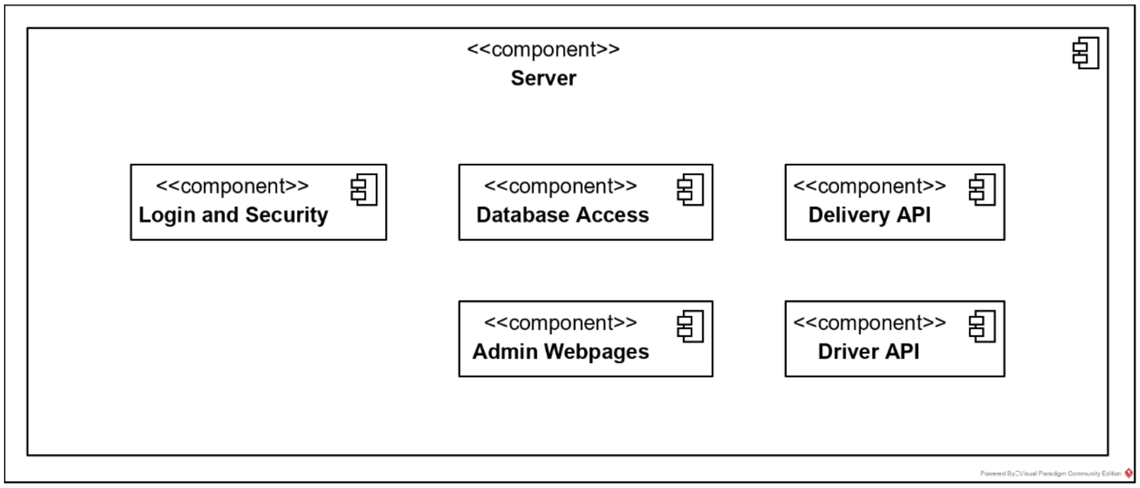


Figure 2.1.2. Server components.

The app is intended to run on android mobile devices, the server is hosted using Heroku's hosting service, and the MongoDB database is hosted on MongoDB Atlas.

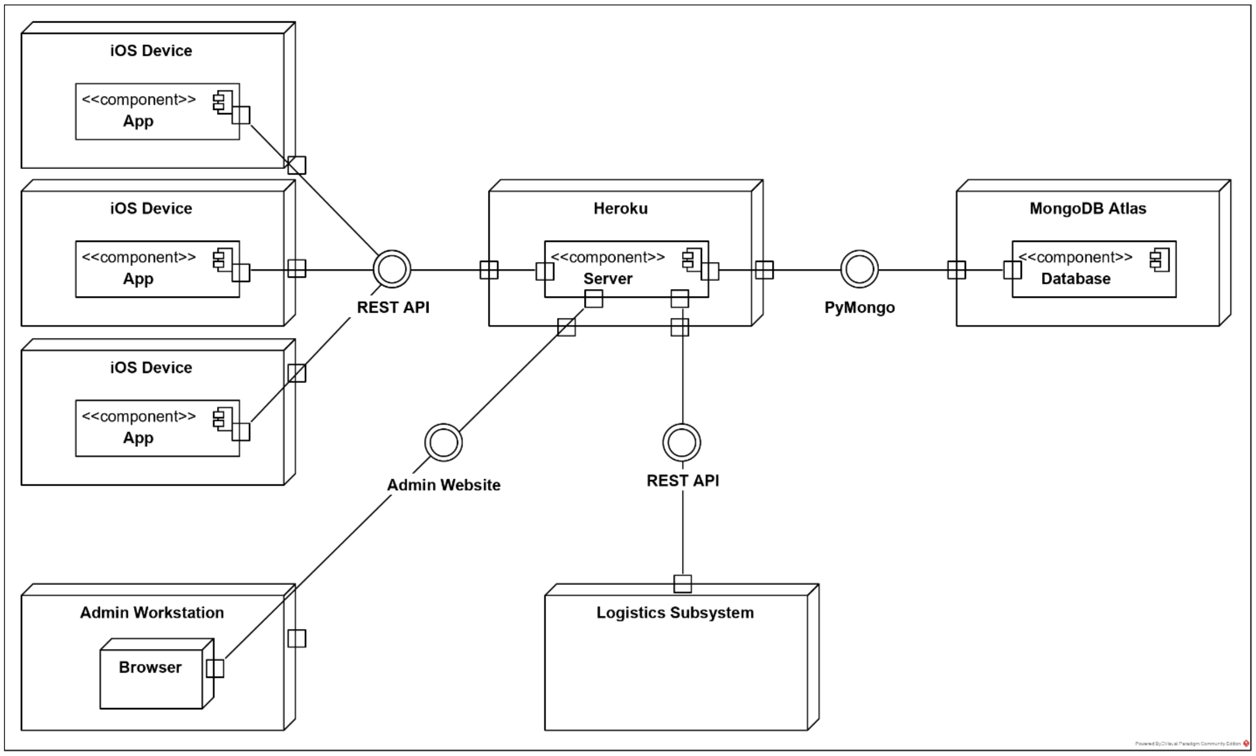
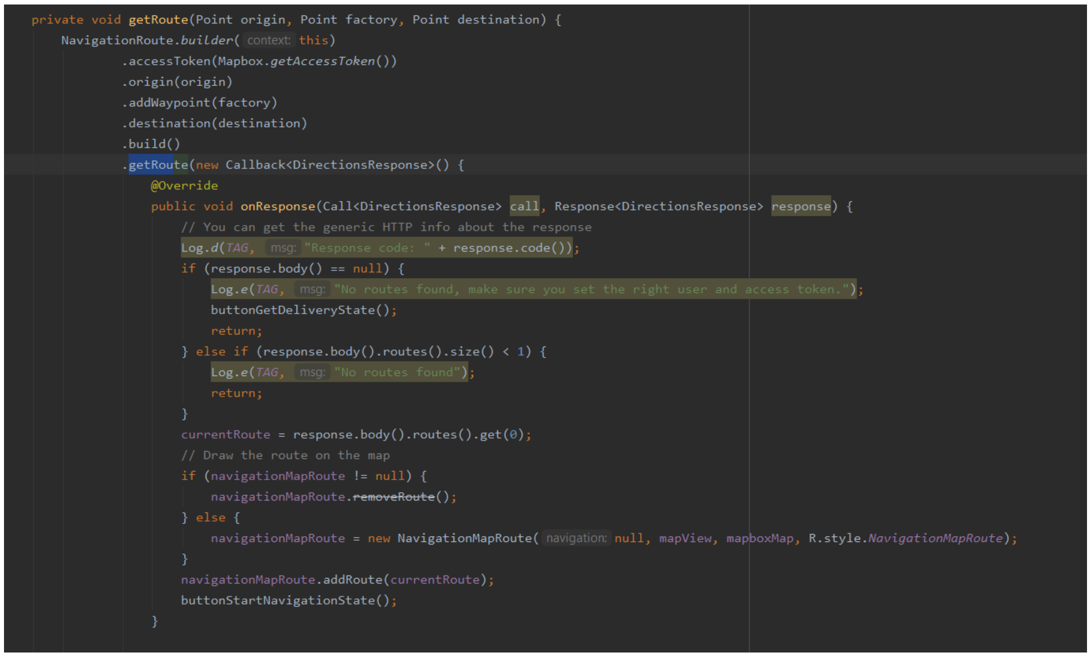


Figure 2.1.3. Routing subsystem deployment.

3. Detailed Description of Components

3.1. Routing App

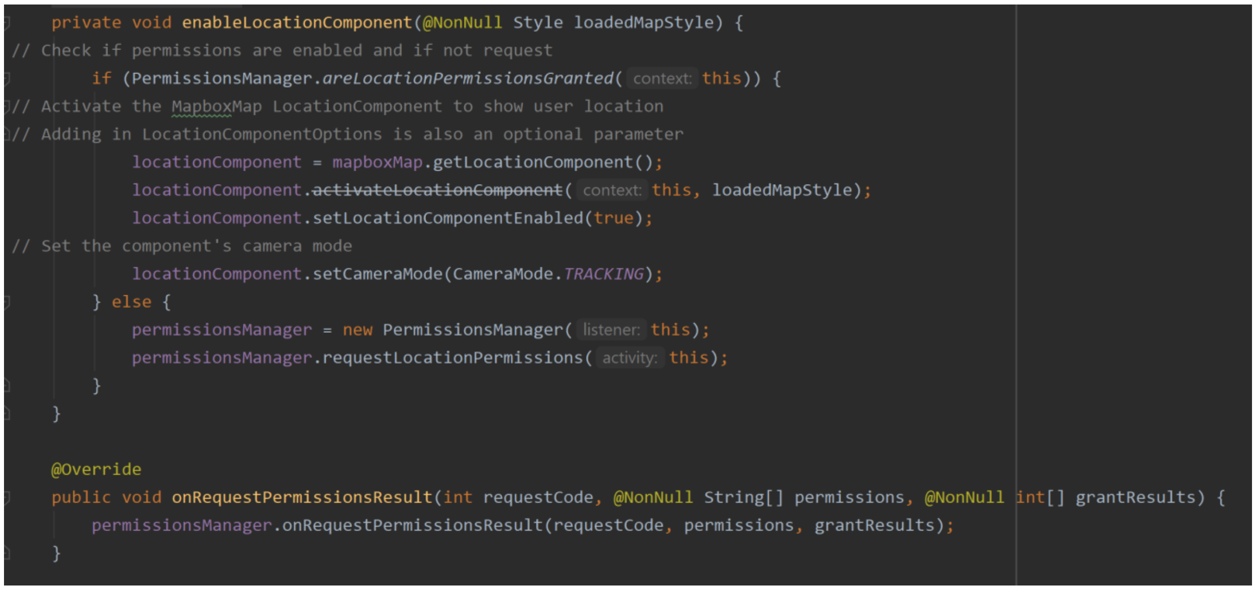
3.1.1. Mapbox API



```
private void getRoute(Point origin, Point factory, Point destination) {
    NavigationRoute.builder( context: this)
            .accessToken(Mapbox.getAccessToken())
            .origin(origin)
            .addWaypoint(factory)
            .destination(destination)
            .build()
            .getRoute(new Callback<DirectionsResponse>() {
                @Override
                public void onResponse(Call<DirectionsResponse> call, Response<DirectionsResponse> response) {
                    // You can get the generic HTTP info about the response
                    Log.d(TAG,  msg: "Response code: " + response.code());
                    if (response.body() == null) {
                        Log.e(TAG,  msg: "No routes found, make sure you set the right user and access token.");
                        buttonGetDeliveryState();
                        return;
                    } else if (response.body().routes().size() < 1) {
                        Log.e(TAG,  msg: "No routes found");
                        return;
                    }
                    currentRoute = response.body().routes().get(0);
                    // Draw the route on the map
                    if (navigationMapRoute != null) {
                        navigationMapRoute.~~removeRoute~~();
                    } else {
                        navigationMapRoute = new NavigationMapRoute( navigation: null, mapView, mapboxMap, R.style.NavigationMapRoute);
                    }
                    navigationMapRoute.addRoute(currentRoute);
                    buttonStartNavigationState();
                }
```

This code above is a snippet of the map code using the mapbox API. This function is responsible for mapping the route from the driver’s location to the customer’s address pulled from the database. The marker is added to the map and a line is placed on the fastest route between the two points.

3.1.2. User Permissions

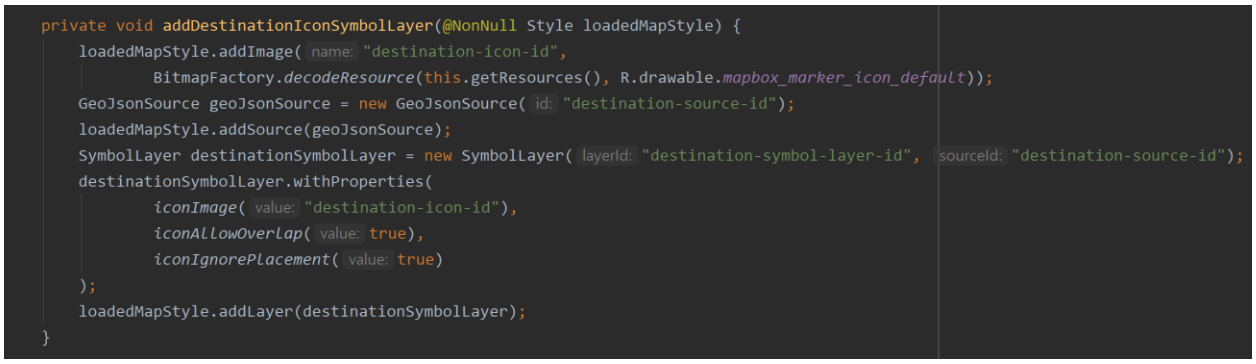


```
    private void enableLocationComponent(@NonNull Style loadedMapStyle) {
// Check if permissions are enabled and if not request
        if (PermissionsManager.areLocationPermissionsGranted( context: this)) {
// Activate the MapboxMap LocationComponent to show user location
// Adding in LocationComponentOptions is also an optional parameter
            locationComponent = mapboxMap.getLocationComponent();
            locationComponent.~~activateLocationComponent~~( context: this, loadedMapStyle);
            locationComponent.setLocationComponentEnabled(true);
// Set the component's camera mode
            locationComponent.setCameraMode(CameraMode.TRACKING);
        } else {
            permissionsManager = new PermissionsManager( listener: this);
            permissionsManager.requestLocationPermissions( activity: this);
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {
        permissionsManager.onRequestPermissionsResult(requestCode, permissions, grantResults);
    }
```

The code snippet above is responsible for getting the user’s permission for using the mobile device’s location services and displaying the user’s location on the map.

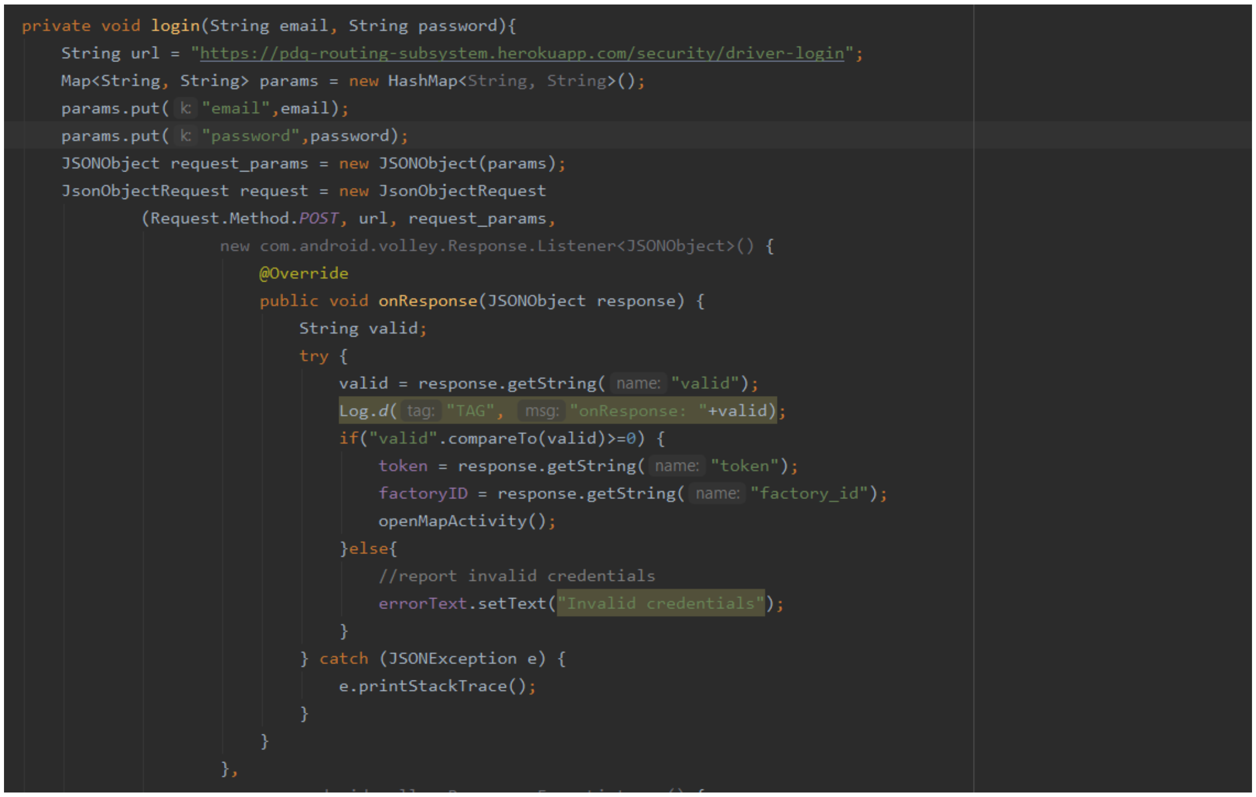
3.1.3. Add Destination



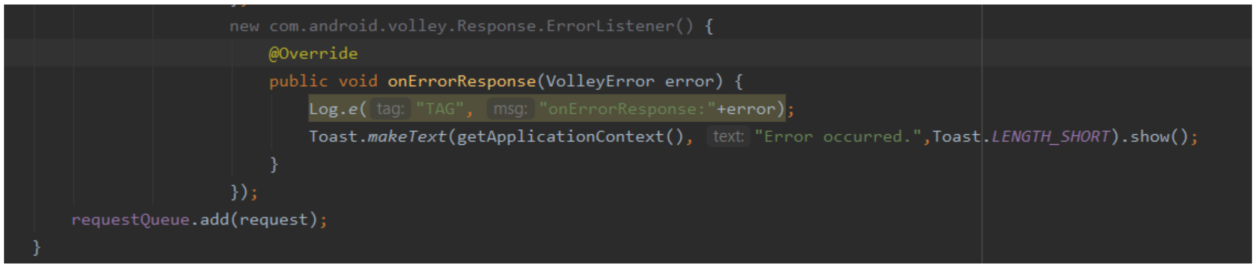
```
private void addDestinationIconSymbolLayer(@NonNull Style loadedMapStyle) {
    loadedMapStyle.addImage( name: "destination-icon-id",
            BitmapFactory.decodeResource(this.getResources(), R.drawable.mapbox_marker_icon_default));
    GeoJsonSource geoJsonSource = new GeoJsonSource( id: "destination-source-id");
    loadedMapStyle.addSource(geoJsonSource);
    SymbolLayer destinationSymbolLayer = new SymbolLayer( layerId: "destination-symbol-layer-id",  sourceId: "destination-source-id");
    destinationSymbolLayer.withProperties(
            iconImage( value: "destination-icon-id"),
            iconAllowOverlap( value: true),
            iconIgnorePlacement( value: true)
    );
    loadedMapStyle.addLayer(destinationSymbolLayer);
}
```

The code snippet above is used for adding the destination placemark on the map after being pulled from the database.

3.1.4. Login with Volley



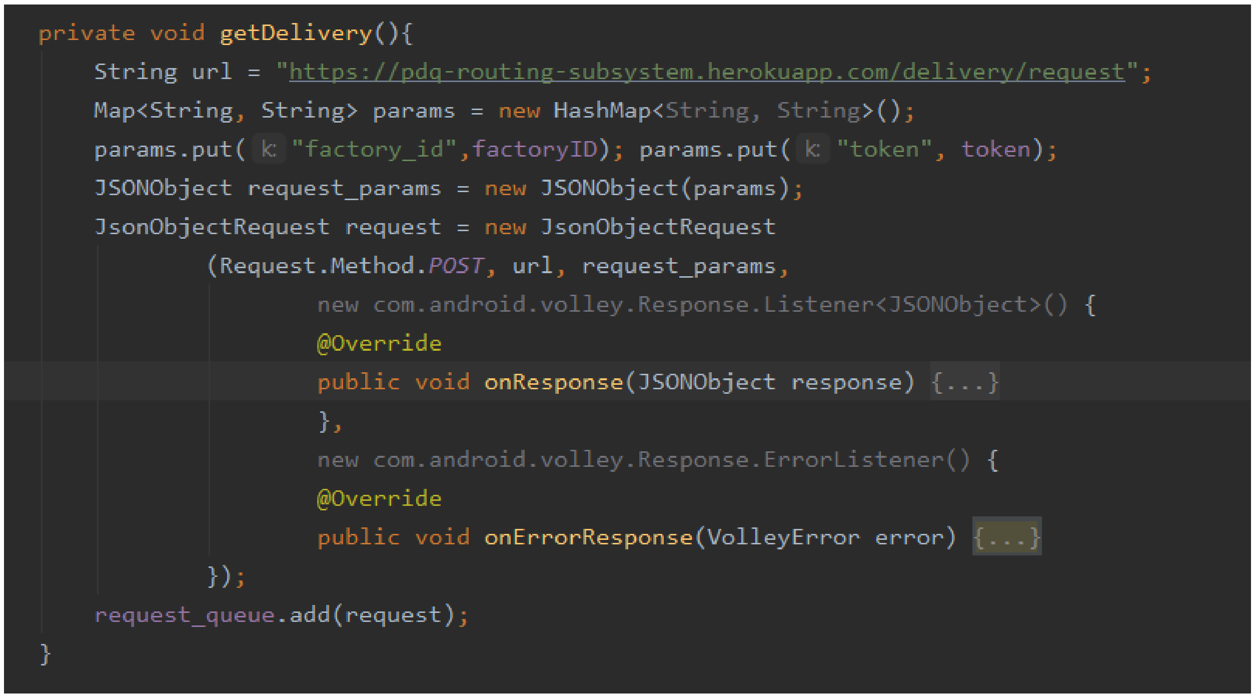
```
private void login(String email, String password){
    String url = "https://pdq-routing-subsystem.herokuapp.com/security/driver-login";
    Map<String, String> params = new HashMap<String, String>();
    params.put( k: "email",email);
    params.put( k: "password",password);
    JSONObject request_params = new JSONObject(params);
    JsonObjectRequest request = new JsonObjectRequest
            (Request.Method.POST, url, request_params,
                    new com.android.volley.Response.Listener<JSONObject>() {
                        @Override
                        public void onResponse(JSONObject response) {
                            String valid;
                            try {
                                valid = response.getString( name: "valid");
                                Log.d( tag: "TAG",  msg: "onResponse: "+valid);
                                if("valid".compareTo(valid)>=0) {
                                    token = response.getString( name: "token");
                                    factoryID = response.getString( name: "factory_id");
                                    openMapActivity();
                                }else{
                                    //report invalid credentials
                                    errorText.setText("Invalid credentials");
                                }
                            } catch (JSONException e) {
                                e.printStackTrace();
                            }
                        }
                    },
```



```
                    new com.android.volley.Response.ErrorListener() {
                        @Override
                        public void onErrorResponse(VolleyError error) {
                            Log.e( tag: "TAG",  msg: "onErrorResponse:"+error);
                            Toast.makeText(getApplicationContext(),  text: "Error occurred.",Toast.LENGTH_SHORT).show();
                        }
                    });
    requestQueue.add(request);
}
```

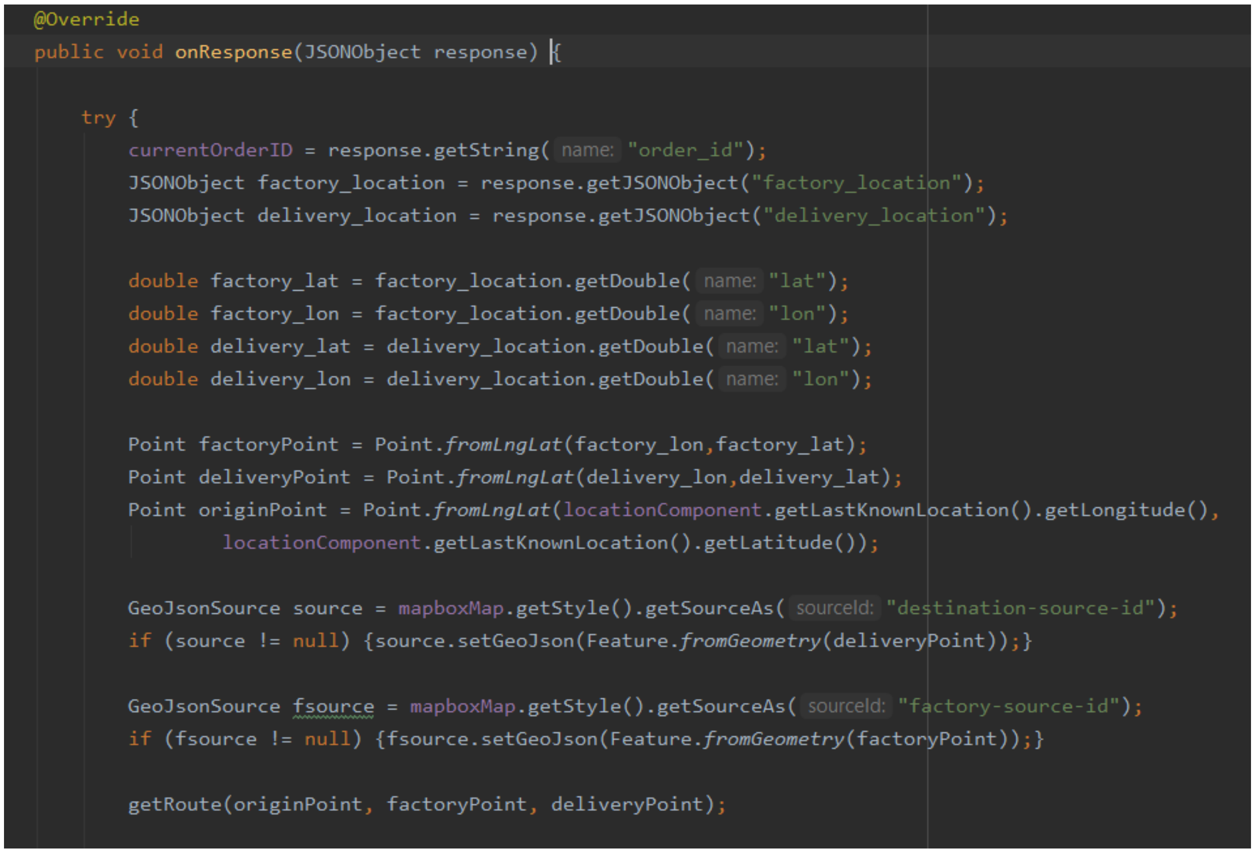
The code above is from the LoginActivity in the routing app and is the method used to verify driver credentials. It takes an email and a password as arguments and attaches them to a JSON object in a request and adds the request to the request queue. If the response from the server indicates the credentials are valid the MapActivity is opened.

3.1.5. Get Delivery with Volley



```
private void getDelivery(){
    String url = "https://pdq-routing-subsystem.herokuapp.com/delivery/request";
    Map<String, String> params = new HashMap<String, String>();
    params.put( k: "factory_id",factoryID); params.put( k: "token", token);
    JSONObject request_params = new JSONObject(params);
    JsonObjectRequest request = new JsonObjectRequest
            (Request.Method.POST, url, request_params,
                    new com.android.volley.Response.Listener<JSONObject>() {
                    @Override
                    public void onResponse(JSONObject response) {...}
                    },
                    new com.android.volley.Response.ErrorListener() {
                    @Override
                    public void onErrorResponse(VolleyError error) {...}
            });
    request_queue.add(request);
}
```

The code above from the MapActivity in the routing app attaches a JSON object with the factory ID assigned to the driver logged in, a requests information on a pending delivery.



```java
@Override
public void onResponse(JSONObject response) {

    try {
        currentOrderID = response.getString( name: "order_id");
        JSONObject factory_location = response.getJSONObject("factory_location");
        JSONObject delivery_location = response.getJSONObject("delivery_location");

        double factory_lat = factory_location.getDouble( name: "lat");
        double factory_lon = factory_location.getDouble( name: "lon");
        double delivery_lat = delivery_location.getDouble( name: "lat");
        double delivery_lon = delivery_location.getDouble( name: "lon");

        Point factoryPoint = Point.fromLngLat(factory_lon,factory_lat);
        Point deliveryPoint = Point.fromLngLat(delivery_lon,delivery_lat);
        Point originPoint = Point.fromLngLat(locationComponent.getLastKnownLocation().getLongitude(),
                locationComponent.getLastKnownLocation().getLatitude());

        GeoJsonSource source = mapboxMap.getStyle().getSourceAs( sourceId: "destination-source-id");
        if (source != null) {source.setGeoJson(Feature.fromGeometry(deliveryPoint));}

        GeoJsonSource fsource = mapboxMap.getStyle().getSourceAs( sourceId: "factory-source-id");
        if (fsource != null) {fsource.setGeoJson(Feature.fromGeometry(factoryPoint));}

        getRoute(originPoint, factoryPoint, deliveryPoint);
```

Once the app gets a response it converts the latetude and longitude of the driver, the factory and the delivery location to points and call the get route function.

3.2. Routing Server

3.2.1. Scripts

3.2.1.1. DriverAPI.py

Contains flask blueprint with all routes used in the Driver API that is used to get, update, or store driver data.

3.2.1.2. DeliveryAPI.py

Contains flask blueprint with all routes and functions used in the Delivery API that is used to get, update, or store delivery data.

3.2.1.3. Security_API.py

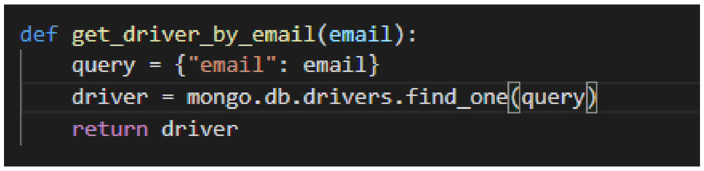
Contains flask blueprint with routes to verify drivers.

3.2.1.4. AdminRoutes.py

Contains flask blueprint with all routes used to direct administrators to correct webpages.

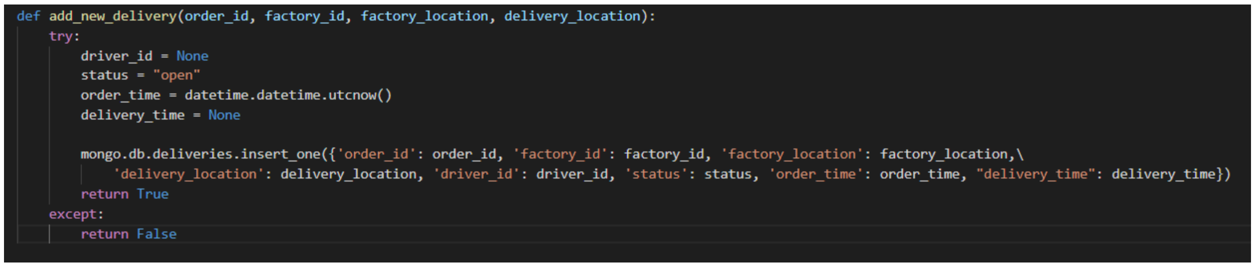
3.2.1.5. DB.py

Contains all functions used to access the database.



```
def get_driver_by_email(email):
    query = {"email": email}
    driver = mongo.db.drivers.find_one(query)
    return driver
```

The code above is an example of how data was retrieved from the MongoDB database using pymongo.



```
def add_new_delivery(order_id, factory_id, factory_location, delivery_location):
    try:
        driver_id = None
        status = "open"
        order_time = datetime.datetime.utcnow()
        delivery_time = None

        mongo.db.deliveries.insert_one({'order_id': order_id, 'factory_id': factory_id, 'factory_location': factory_location,\
            'delivery_location': delivery_location, 'driver_id': driver_id, 'status': status, 'order_time': order_time, "delivery_time": delivery_time})
        return True
    except:
        return False
```

The code above is an example of how documents were inserted into the MongoDB database using pymongo.

3.2.1.5. Security.py

```
def token_required(f):
    # Creates wrapper to verify tokens for API routes
    @wraps(f)
    def decorated(*args, **kwargs):
        try:
            pass
            token = request.json['token']
            try:
                data = jwt.decode(token, SECRET_KEY,algorithms="HS256")
            except :
                return jsonify({'message': 'token is invalid.'})
        except:
            return jsonify({'message': 'token is missing.'})


        return f(*args, **kwargs)
    return decorated
```

The code above is a decorator in the routing server that ensures that the request has a token that authenticates the source of an API call.

3.2.2. API

The deployed API is host at the URL: https://pdq-routing-subsystem.herokuapp.com/

3.2.2.1. Add Delivery

URL: https://pdq-routing-subsystem.herokuapp.com/delivery/add
Method: POST
Required JSON: {‘order_id’: <order_id>,
‘factory_id’:<factory_id>,
‘factory_location’: {“lat”: <latitude>, ”lon”:longitude},
‘delivery_location’: {“lat”: <latitude>, ”lon”:longitude}}
Success Response:
Code: 200
Content: {‘success’}
Error Response:
Code: 400
{“Unexpected error”}

3.2.2.2. Request Delivery

URL: https://pdq-routing-subsystem.herokuapp.com/delivery/request
Method: POST
Required JSON: {‘factory_id’:<factory_id>}
Success Response:
Code: 200
Content: {“_id”: <delivery_id>,
“order_id”: <order_id>,”
factory_id”: <factory_id>,
“factory_location”: {“lat”: <latitude>, ”lon”:longitude},
“delivery_location”: {“lat”: <latitude>, ”lon”:longitude} ,
“driver_id”: <driver_id>},
“status”: ”open”, “order_time”: <order_time>,
“delivery_time”: <delivery_time> }
Error Response:
Code: 400
{“Unexpected error”}

3.2.2.3. Complete Delivery

URL: https://pdq-routing-subsystem.herokuapp.com/delivery/deliver
Method: PUT
Required JSON: {‘order_id’: <order_id>,
‘driver_id’:<driver_id>}
Success Response:
Code: 200
Content: {‘success’}
Error Response:

Code: 400
Content: {“Unexpected error”}

3.2.2.4 Driver Login
URL: https://pdq-routing-subsystem.herokuapp.com/security/driver-login
Method: POST
Required JSON: {‘email: <email>,
‘password:<password>}
Success Response:
Code: 200
Content: {‘valid’: ’valid’}
{‘valid’: ‘invalid’}
{‘valid’: ‘missing’}

3.2.3. Database

To run the docker container enter the run command in terminal in directory where you wish to store the docker image.

Run Command:
```
docker run -d -p 2717:27017 -v <directory>-mongo:/data/db --name pdq-mongo mongo:latest
```

Example command:
```
docker run -d -p 2717:27017 -v C:\Users\joshu\Desktop\pdq-mongo:/data/db --name pdq-mongo mongo:latest
```

Development database is run on docker, on ports 2717 on local machine to port 27017 in docker container.
The directory is just a location on your local machine, where you want to store data in the database between session.

The database can be accessed at URL: mongodb://127.0.0.1:2717/PDQ

4. Design Decisions and Tradeoffs

Design Decision	Rational	
Android Application	A relatively simple means of making a system that is easily accessible to anyone with an android device. Allows the user experience to be tailored to the device being used as is now standard. Everyone on the development team has had experience with android applications, which allows for quick development.	
	Discarded Alternatives	
	Alternative	Rational
	Website	The application needs to be portable and as such a mobile application was the best bet.
Mapbox API	A very simple map API capable of running on android devices. It is able to be used in China as supposed to the Google API which isn't.	
	Discarded Alternatives	
	Alternative	Rational
	AMaps	This API was, in the opinion of the development team, not as simple to utilize as Mapbox API is.
	Google API	The Google API does not work well in China and as such is not a viable option.
Volley	A library used to easily send HTTP requests and parse responses.	
Python Flask	A simple easy to use python micro-framework. As students we were quite familiar with this framework and wanted to explore the creation of APIs using flask.	
	Discarded Alternatives	
	Alternative	Rational

	Vapor	A simple and effective server framework often used to connect to databases. Since we were not familiar with Vapor and had run into some issues using it, we opted to switch flask which we were more familiar with to meet deadlines.
MongoDB	Currently the most popular NoSQL document-store making it an obvious choice for a document-store DB. It allows for information to be stored as simple easy to use JSON-type formatted documents, that does not require a predefined database schema. This allows for the information collected by the system to be changed relatively easily if the delivery information being tracked changes in the future. NoSQL databases like MongoDB also excel at quickly storing and retrieving large amounts of data, which benefits the system as its main function is to store and track deliveries.	
	Discarded Alternatives	
	Alternative	Rational
	PostgresSQL	The storage of delivery information does not gain much benefit from relational functionality and may restrict changes to the format of delivery information in the future. Storage of admin and driver account data may benefit from a RDBMS and ACID restraints, though this is not needed for the time being. The system could be transitioned to using PostgresSQL to store account information in conjunction with MongoDB for delivery information in the future. Additionally, as students we wanted to focus on gaining more experience with NoSQL databases.
	Firebase Realtime Database	A document store database considered in place of MongoDB. The service provides many features making it easy to use in many frameworks including android and provides consistent data across multiple users. This would make it extremely easy to setup and use in an android app, however since it is a google service it would not function

		without a VPN in china.
	HBase	A NoSQL columnar database that would allow easy changes to the format of data in the future. However, this DB was considered to be more difficult to setup and use with flask or android and the benefits of column families was not seen to be a sufficient reason to use over MongoDB.
PyMongo	Standard python library to access MongoDB. Covers all the needs of the system with regards to accessing the database. No alternatives were discussed.	
	Discarded Alternatives	
	Alternative	Rational
	Mongo Engine	An Object Document Mapper (ODM), useful for accessing MongoDB database and controlling interactions with the database with objects. Was considered but given the simplicity of the current iteration of the system it was deemed not necessary for the time being but should be should transitioned to in the future.
Heroku	Commonly used and well trusted hosting service. The member of the development team responsible for deployment of the demo site was most familiar with this platform.	
MongoDB Atlas	Standard platform for hosting MongoDB databases. The platform is owned by the creators of MongoDB and is therefore well supported. The platform is also supported by the chosen hosting service Heroku.	

5. Appendix

A – References:

References:
Example SDS's used as reference for the creation of this document:
https://www2.rivier.edu/faculty/vriabov/CS552_SW_Design_Specification_Example.pdf
https://yuan-lu.weebly.com/uploads/3/9/3/7/39371931/designdoc-2.pdf
http://csis.pace.edu/~marchese/SE616/WebDocs/soft-des-spec-tmp.htm
http://www.cs.iit.edu/~oaldawud/CS487/project/software_design_specification.htm

Deployed PDQ admin page demo: https://pdq-routing-subsystem.herokuapp.com

PDQ routing app GitHub: https://github.com/AntonioWorkman/RoutingApp
PDQ routing server GitHub: https://github.com/joshua-alkins/PDQ_routing_server

B – UI Pictures:

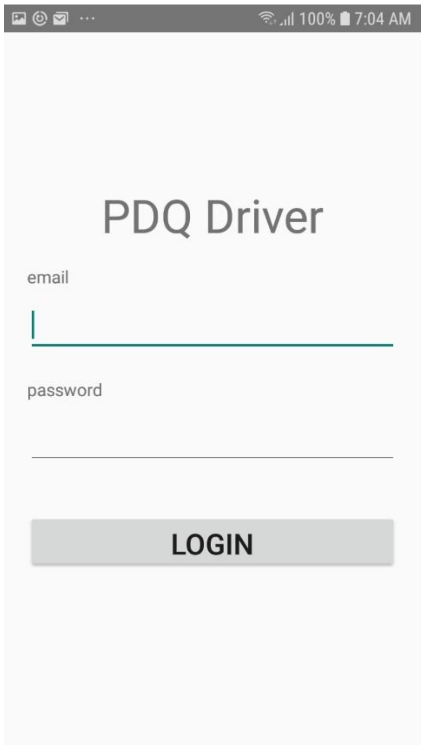


Figure B.1. App login screen.

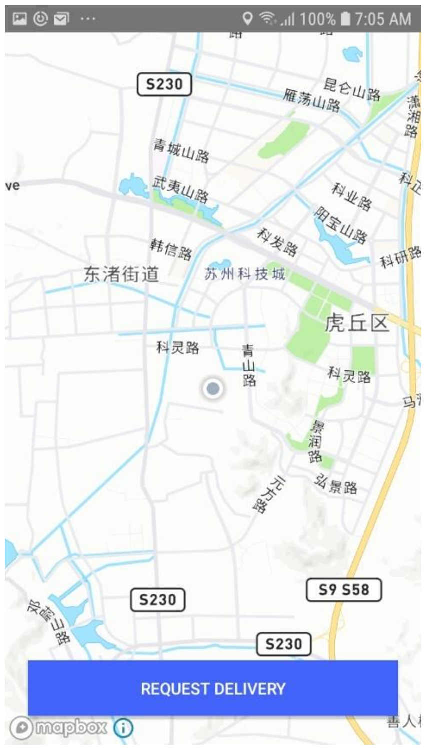


Figure B.2. Map screen without active delivery

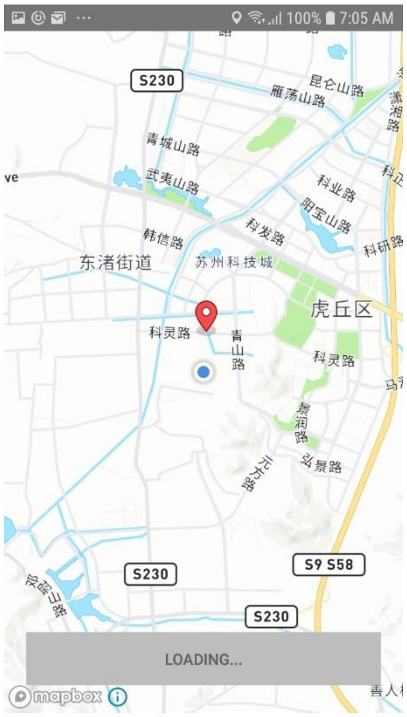


Figure B.3. Map screen waiting on response from server.

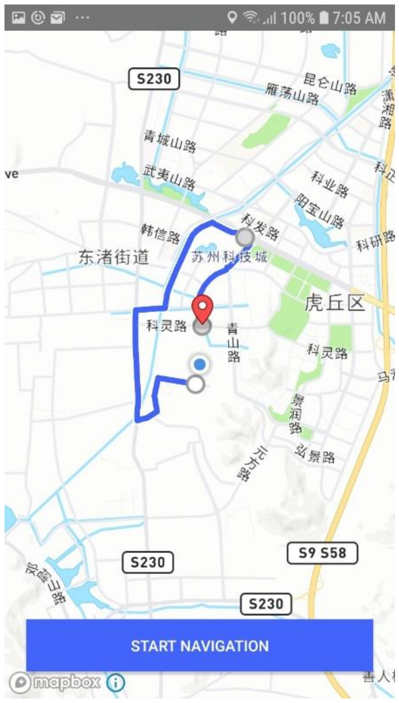


Figure B.4. Map screen displaying route.

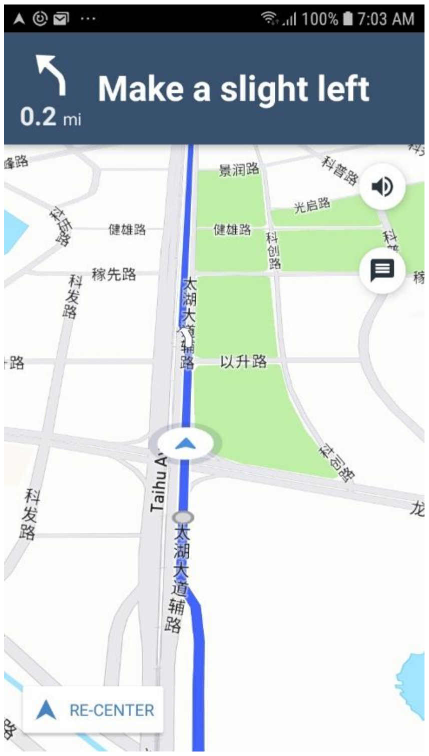


Figure B.5. Navigation screen on route.