Smart City Bus

A Project Report

Presented to

The Faculty of the College of

Engineering

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science in Computer Engineering

By

Venkata Adithya Boppana

Varsha Kodali

Sashank Malladi

Ravali Peddi

Graduation: May, 2017

APPROVED

Frank Lin

Digitally signed by Frank Lin
DN: cn=Frank Lin, o=San Jose State University, ou,
email=frank.lin@sjsu.edu, c=US
Date: 2017.05.10 20:49:01 -07'00'

Frank Lin, Project Advisor

Dr. Dan Harkey, Director, MS Computer Engineering

Dr. Xiao Su, Department Chair

ABSTRACT

Smart City Bus

By Venkata Adithya Boppana, Varsha Kodali, Sashank Malladi, Ravali Peddi

Public Transportation is a necessity for many people who cannot drive or who cannot afford private vehicles. According to APTA (American Public Transport Association), number of passengers relying on public transit has increased by 39 percent since 1995. Also, in countries like India and China, public transport serves millions of people. Currently, there are more than 6400 providers of public transport in USA alone. These transit agencies maintain dedicated scheduled services to help passengers in planning their trips.

Currently, transit agencies provide schedule information of their services in various formats. But, the information provided is not accurate as services may not adhere to these static schedules. Schedules are impacted by various factors like traffic congestion, average speed, cha nge in service route etc. Accordingly, there is a need for a solution which provides more reliable information to passengers.

In this project, we propose a solution for transit agencies to maintain more reliable schedule by dynamically tracking their vehicles. Latest Information Technologies involving web applications are used to alert passengers about the exact location of the vehicle and its estimated arrival time. Vehicles are tracked using GPS (Global Positioning System) and estimated arrival time for each stop in a route is calculated considering the impacting factors. As a result, public transit services can be made more reliable means of transport for many people.

Acknowledgments

The authors are deeply indebted to Professor Frank Lin and Professor Dan Harkey for their invaluable comments and assistance in the preparation of this study.

Table of Contents

List of figures

List of tables

1 Project Overview

1.1 Introduction

Public transit services are being increasingly used by people who cannot drive or who cannot afford private vehicles. Transit agencies provide their static schedules in various electronic formats helping passengers in planning their trips. Currently, only a few agencies provide advanced features like estimated arrival time by dynamically tracking their vehicles. Smart City Bus is a smartphone based system which provides real-time transit information of transit services. It encourages more users to ride public transit vehicles which is a positive sign for the growth of a nation's economy.

As part of our project, an android smartphone application was developed for the end users to obtain the estimated arrival time of each bus. This, in turn, minimizes the efforts of passengers to visit agency website to know the arrival time of a vehicle. Instead of a dedicated GPS tracking device, android smartphone placed in each transit vehicle is used for tracking transit vehicles. An application installed on these smartphones sends GPS coordinates of the vehicle to the server at regular intervals. The server then updates the present location of the vehicle in the database. ETA (Estimated Time Arrival) is calculated using an efficient algorithm which is later retrieved when a user makes a request to the server using the end user android application.

1.2 Proposed Areas of Study and Academic Contribution

Our project aim was to create a system which can deliver real-time arrival schedule of public transit buses. These are areas of study for our project.

1.2.1 User Interface Design

User interface is the design of human and machine interaction. It serves the purpose of decision making for a user. UI design is a key feature for every application which provides better understanding of the usage of the application. In our project user interface is an essential feature for three different applications GPS tracking application on transit vehicle, End user application, Web application for transit agency. In our project, we developed two android smart phone applications and a web application. The two smart phone applications were developed as hybrid applications. Hybrid applications combine both traditional native and HTML5 application features. Hybrid applications use the latest HTML5, CSS and JavaScript technologies and are also cross compatible with all platforms (IOS/Android). We have done research on different frameworks available and have decided to use Ionic framework due to its inherent features. Background applications can be developed easily on Ionic by installing the required Cordova plugins. Also, Ionic provides all functionalities required to access native UI elements and provides better performance for hybrid applications.

1.2.2 Backend Development

Backend server applications are required to process data received from frontend and calculate the Estimated Arrival time of the buses using that data. In our project server applications perform three different tasks. One is to update the present location of the vehicle in the database, second to calculate ETA to each stop and other to serve user requests. NodeJS and python was used to develop server supplication in our project. Node JS being event driven, it can handle many concurrent requests with low CPU resources. While to calculate the Estimated Arrival time at bus stops we used Python, Google Distance matrix API's and Open Street Map API's.

1.2.3 Database

Our application is mostly data oriented as it requires transit vehicles data to calculate ETA, also the present location of all the vehicles for an agency. MongoDB and PostgreSQL are used for maintaining nonstructural and structural data respectively. GTFS (General Transit Feed Specification) standard schema by Google is considered for maintaining structural data in PostgreSQL. GTFS is a General Transit Feed Specification which presents a familiar format for public transportation schedules. Transit agencies submit their own transit data in a format specified by GTFS.

2 Background and Motivation

2.1 Literature Survey

In 2014 alone as many as 10.8 billion trips were made on public transport in United States alone. Almost all those passengers do not have the real-time data about the bus arrival time and must wait at the bus stop and unable to plan because of this unpredictability. So, by providing them with exact timing of the bus helps them to plan and notifies the passenger about delays beforehand.

There are different predefined formats for datasets of schedule information for transit agencies. GTFS is a General Transit Feed Specification which presents a common format for Public transportation schedules. Transit agencies submit their transit data in a format specified by GTFS. Developers write applications which consume this transit data provided by the agencies for navigational purposes. Most of the current transit tracking applications are compatible with the GTFS feeds. These GTFS data feeds are also updated by the transit operators to get the most recent updated schedules. GTFS data feeds are static, cannot be impacted by the operational delays of vehicles [5]. Following datasets are used by conventional bus tracking systems [2].

1. Static dataset: Static dataset is static schedules containing routes, stops, trips provided in a specified format like GTFS.
2. Time point dataset: Time point data set is historical data of arrival time of vehicles including stop ID, route ID, bus ID.
3. Real-time transit feeds: Real-time transit data is the data collected from an AVL (Automatic Vehicle Location) system.

Since we used General Transit Feed Specification (GTFS) specifications for the project, our project can be used for any transit agency. Also, other developers can take our idea and improve upon the project. GTFS is widely used by many transit providers when making their data open source also, google static API's use this specification.

2.2 Current State of the Art

Applications that are already existing which are to show arrival times of public transit services does not use real-time transit information. Those applications use static schedule information provided by the transit systems to determine next expected arrival time of the buses. In our project, we have utilized GPS technology to track the bus using a smart phone. As every smart phone comes with built-in GPS, we built an application which can connect with a dedicated server and send the location updates to the server periodically.

Our application includes some noteworthy features over current state-of-the-art. Using GPS enabled smart phone as a tracker will eliminate the cost of using a dedicated GPS tracking device. As the tracking devices come with nice and simple UI for the drivers, the GPS tracking can be turned on only at the starting of the trip. This makes the device more durable and is efficient in terms of data generation and storage. All the operational delays are considered and the algorithm is very efficient in predicting the arrival times. Usage of hybrid technologies and frameworks for building the applications makes them seamless, quick and responsive. There is web portal for transit agencies to monitor their vehicles. The management team can make change to the routes right from their portal. Current state of the art does not include these features. The web application stores the historical data. The data trends can be helpful for the agencies to make decisions in a way to optimize the transit system and make it more accessible to the common people.

3 Requirements and Analysis

3.1 User Requirements

Following Requirements are gathered from a transit user (fictional)

1. I want to access the application from any location so that I can get ETA of nearest approaching routes.
2. I want to see my current location and locations of nearby stops on Google Maps.
3. I want to get information about the nearest stops and all routes stopping by those stops.
4. I want to get the travel directions to each stop when a route is selected.
5. I want to see the list of stops passed and yet to be passed of a selected route.

Following Requirements are gathered from a transit agency (fictional)

1. I want to see the list of current operating routes and their vehicle timestamps.
2. I want to see the path of a route when selected, and retrieve live updates about its current location.
3. I want to see location of both delayed trips and right on schedule trips for all routes on Maps.

3.2 Functional Requirements

1. User Application shall retrieve nearest stops and ETA of all routes stopping by those stops based on user's current location.
2. User Application shall display the current location of user and three nearby stop locations on Google Map.
3. User Application shall display the travel directions to the stop for a selected route.
4. GPS Tracking Application on each bus shall send the GPS coordinates to the server at regular Intervals, also works in background.
5. Web Application for transit agency will display all the currently running trips with their current location and status corresponding to their assigned schedule.

3.3 Technical Requirements

Hardware Requirements

1. Desktop/Laptop
2. Smart Phone (Android 5.0 or above)
3. Active Internet Connection

Software/Infrastructure Requirements

1. Ionic Framework-Framework used to build hybrid application.
2. Cordova Build Tool
3. Android SDK
4. AngularJS, Python, NodeJs, MongoDB
5. Amazon EC2
6. Google distanceMatrix API

4 Project Architecture

4.1 Introduction

Smart City Bus is an Android application which gives users access to most reliable bus schedules. The application uses GPS technology to track the transit buses. Thus, the real-time information provided by this application can be used to plan the trips efficiently with less waiting at the bus stops. The implemented project involved design and development of three modules separately. Bus- side application, server application, user-end application altogether constitutes the proposed system. The overall project architecture is as shown below in the below figure.

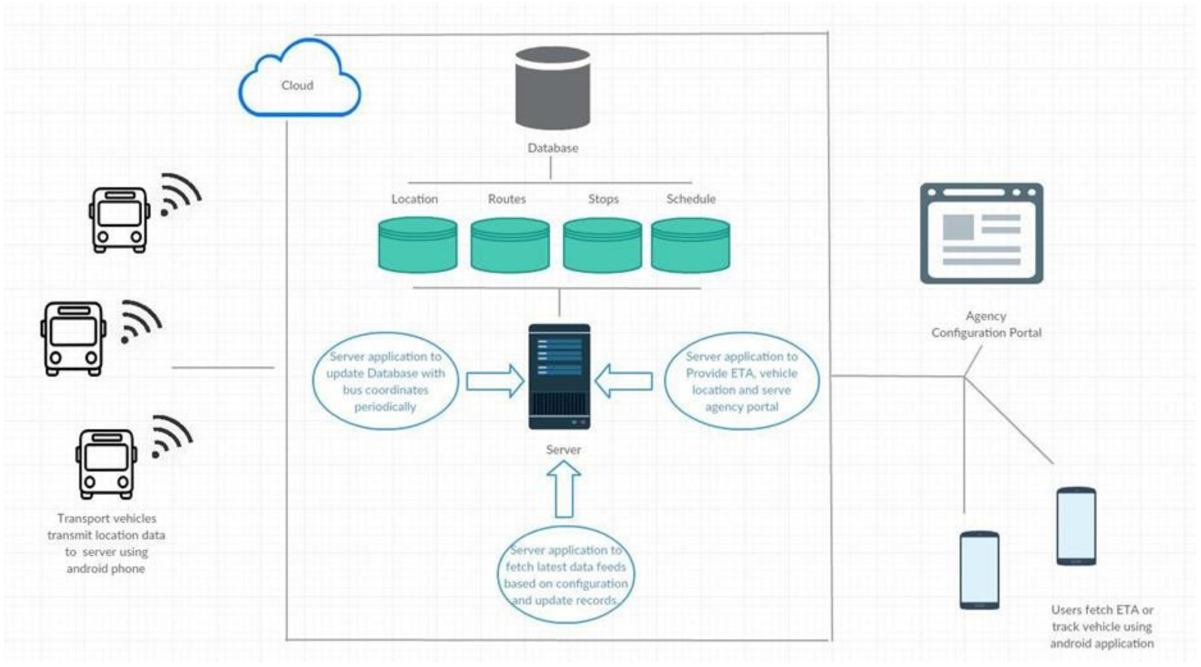

Cloud
Database
Location
Routes
Stops
Schedule
Server application to update Database with bus coordinates periodically
Server
Server application to Provide ETA, vehicle location and serve agency portal
Server application to fetch latest data feeds based on configuration and update records
Agency Configuration Portal
Transport vehicles transmit location data to server using android phone
Users fetch ETA or track vehicle using android application

Figure 1 Project Architecture

4.2 Architecture Subsystems

4.2.1 Client Applications

1. ***GPS Tracking Application for Transit Vehicles***

An android device with bus-side application running in the background will be installed in the bus. Once the trip starts, the application starts tracking the bus and will periodically send latitude and longitude updates of the bus to the server hosted in cloud platform. The application was built on hybrid platform using Ionic framework, HTML5, CSS and JavaScript. The application interacts with the server by sending JSON data (GPS Coordinates).

2. ***End User Application***

This application serves user's requests by providing the list of vehicles in the nearest bus stops based on user location. User location is sent as JSON data to the server. The server queries the database and sends the requested information. The application will interpret the data and displays the live location of the selected bus on Google maps along with its estimated arrival time. This application is also a hybrid application built in Ionic framework using HTML5, CSS and JavaScript.

3. ***Web Application for Transit Agency***

The web portal for transit agencies can be used by supervising team of the transit agency to view the real-time location of all their vehicles. This also provides additional status information on each trip of selected routes. The web application was developed using Angular 2 framework.

4.2.2 Server Applications

The server application performs three different tasks. It interacts with bus-side application, user application and feed data base. One server application will keep track of current location of all the buses by updating the database based on the data received from all the tracking applications. The second server computes the distance and estimated arrival times from current location of the bus to all its stops along the route and updates the database. These computations are performed every time the bus co-ordinates are updated. The third server application processes the requests from the user applications. It queries the database for nearest three bus stops based on the user location. It returns the list of all the buses that have stops in those bus stops by sub-querying based on the bus stops. Node.JS was used to build the server applications which interacts with the client applications. Python was used to build the server which computes the estimated arrival time due to its numerous supporting libraries.

Approach for ETA Calculation:

Each route gets its own process with slight differences in functionality and data. As each route has different bus stops, every route process has different GPS locations in its Global 'Bus stop location array'. At the start of the route process, a master thread is created which checks in fixed time intervals for new buses on the route. The master thread does this by checking the database using the transit id key. Once the master thread finds a new bus it creates a new slave thread which start calculating the arrival time of the bus to the stops. The steps followed by the slave thread corresponding to bus perform following steps.

- Query and get the latest GPS location for a bus (using transit id key).
- Calculate the next bus stop that is going to be reached by the bus, by comparing this GPS location to the GPS locations of the bus stops of that route.
- Pass the GPS locations of the bus and the nearest bus stop to Google traffic API and store the returned travel time value.
- Update the new arrival time to the bus stop in the database.
- Also, update the estimated arrival time of next bus stops by calculating the difference between new arrival time and old arrival time of previous bus stop.
- Repeat these steps for every 30 seconds. This is achieved by built-in timer in the server application.

Once the bus reaches its destination master thread notices it and kills the slave thread.

4.2.3 Database

Initially our application fetches static schedule of transit agency and an algorithm is applied on top of it to generate dynamic arrival time based on real-time location of the bus.

This implementation requires us to store the static schedule undisturbed and maintain few additional fields which can be easily accessible by server applications. A good database schema should be designed for making the app very much responsive.

Google has a standard feed file format (GTFS- General Transit Feed Specification) which is accepted by many transit agencies across the globe to share their schedules. According to GTFS, public transport companies provide csv files with data interrelated between the files. With GTFS being designed under a specific schema, it is always appropriate to maintain them in similar manner for easy synchronization. So as part of the application built, we maintained all the GTFS feeds in a relational database (PostgreSQL). This gives very much flexibility to apply complex queries (involving joins) to obtain desired datasets unlike NOSQL databases.

As per the implementation, we had to store and query real-time location of the vehicle continuously. These being smaller tasks involving less computation but purely data centric, NOSQL Database is utilized to serve client request making application more responsive. NOSQL being schema less design, the entities are not fixed. And so, data structure can be modified at any point of time to handle request more efficiently.

Following is the block diagram for functional overview

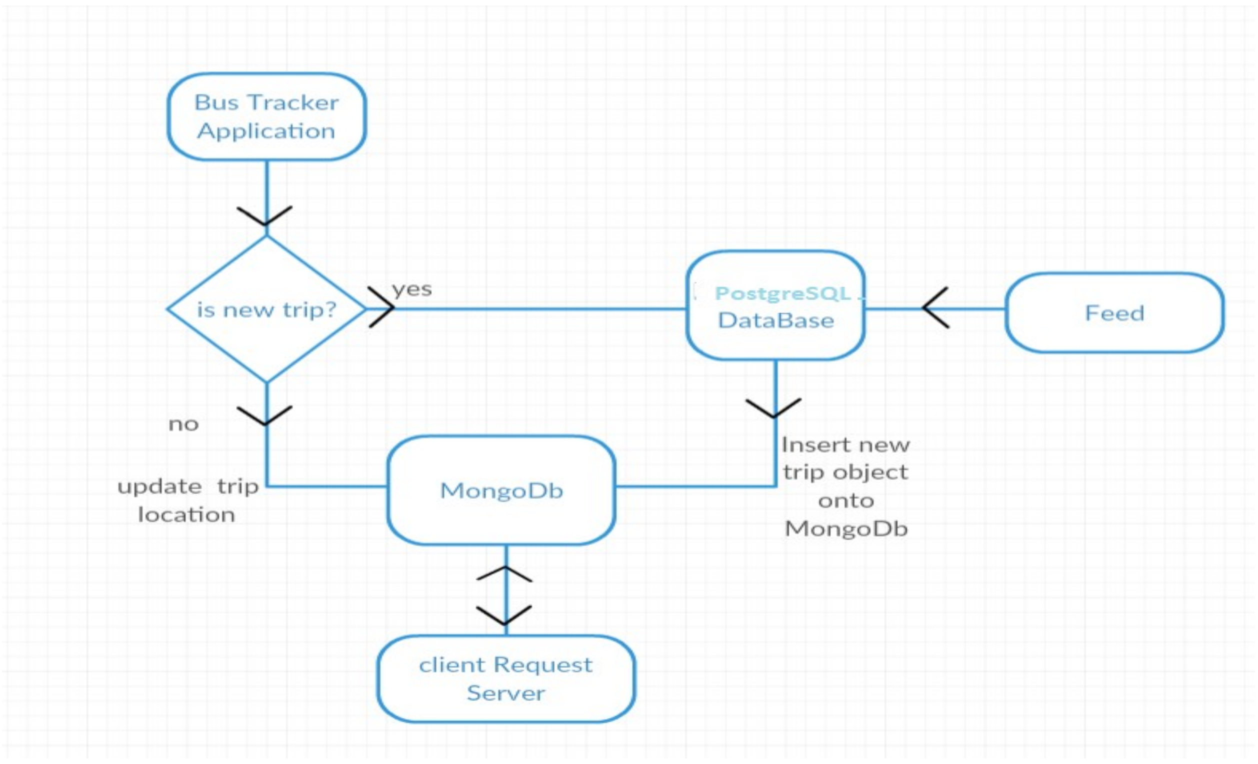


Figure 2 Data access Architecture

5 Technology Descriptions

5.1 Client Technologies

5.1.1 Ionic Framework

Ionic is an open source front-end HTML framework for developing hybrid mobile applications. It is built on top of Cordova and AngularJs. Ionic uses AngularJs MVC architecture for developing optimized mobile applications. Cordova is used to provide access to the native device functionalities using JavaScript. NodeJs environment is used for starting and emulating ionic applications. In our project, GPS tracking application on transit vehicle should run even in the background. Background applications can be developed easily on Ionic by installing the required Cordova plugins. Also, Ionic provides all functionalities required to access native UI elements and provides better performance for hybrid applications.

5.1.2 AngularJS

AngularJS is JavaScript framework which provides advanced features for framework application development. Following features makes Angular extensively used in web application developments.

1. Data Binding: Data values are automatically updated in the view whenever its value is changed in the model.
2. Directives: Angular provides a way to initiate new HTML syntax for the application.
3. Controller: Controllers perform operations in response to the DOM action sequences.
4. Server Communication: Supports many third-party libraries to build XHR request to communicate with the server.
5. Reusable Components: Angular 2.0 provides object oriented features like classes and objects.

In our Project web application for transit agency was built using AngularJs version 2 framework to continuously update the location of the vehicles without refreshing/reloading the page. End User Application also uses Angular 2.0 so that ETA can be updated on the page for every 2 minutes without refreshing/reloading page.

5.1.3 Google API

Google provides API's for multitude of services provided by google maps to be used in third party application. These are paid API services and we are charged by number of API calls. Google Maps Distance Matrix API are used when calculating the estimated arrival time of the buses to their respective stops. Google Maps API is used to render a map on end user application. Google Maps Marker function is used to display user location marker and the nearby stop locations markers. Google Map Directions API is used to get the travel directions from user location to the stop of a selected route.

5.2 Middle-Tier Technologies

5.2.1 Amazon Web Services

AWS is Infrastructure as a service provider which provides us with reliable, scalable, and inexpensive cloud computing services . Of the services provided by AWS we used Elastic compute cloud (EC2), Elastic block storage (EBS), and Amazon Relationship database service (RDS).

Elastic Compute Cloud service (EC2): EC2 provides scalable, elastic and on-demand compute resources to users. In our project, we used EC2 to host all our server applications and for running the mongo dB application.

Relationship Database Service (RDS): RDS is the AWS service which provides different relationship databases for the users. We are using Post-gre SQL database in RDS where the GTFS data is stored.

5.2.2 Node JS

Node.js is an open source server-side JavaScript Platform. It has rich set of JavaScript modules, uses event driven, non-blocking I/O making it more suitable for real-time applications. The following features of Node suits well for building our server application

1. Asynchronous I/O and Event Driven: Node Server doesn't wait for until API calls return the data. Instead it moves on to the next call without blocking in the current event. This makes Node.js incredibly fast in executing the code.

2. Single Threaded: Unlike traditional Apache Http Server, Node.js handles concurrent requests using single Event Loop thread and executed in the background by different workers. Callback

will be invoked to return the result.

In our project, the Node Server application is developed using Express Framework. It is usually a standard server side framework for Node.js. With Express framework, it is convenient to handle client-server communication using RESTful services. This server application is responsible for handling requests from user application and from Web Application for Transit Agencies, developed for monitoring purposes. Server will establish connection to MongoDB database using MongoDB-Node.js driver

5.2.3 Python

Python is an open source programming language which is licensed under OSI model and currently taken care by the python software foundation. We chose python for the design of the Estimated Time calculation server for following reasons.

1. Multi-threading: Unlike node.js we can create threads and allocate resources for those threads in python. There by ensuring the completion of the task assigned to the thread. Also by using thread joins we can determine to execution flow of the threads.
2. Third party libraries: We can import third party libraries into python run time with very limited overload. Also, PYPI library helps us to import those libraries with ease. This allows us to use third party code aggressively without reinventing the wheel.
3. Coding style: Python has dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles.

5.3 Data-Tier Technologies

5.3.1 Amazon RDS - PostgreSQL

PostgreSQL is one of the most advanced and open source relational database management system which adopt SQL standards with some revisions. It completely supports ACID (Atomicity, Consistency, Isolation, and Durability) features. It also supports custom functions to query JSON data directly from the flat tables that are most useful for our application.

Amazon Relational Database Service (Amazon RDS) provides an easy set up for a relational database in the cloud. Its cost-efficient and resizable solution allows its users to focus more on developing their applications.

5.3.2 MongoDB

MongoDB is an open source cross-platform document-oriented database model. It uses collections and documents which are equivalent to tables in RDBMS. The main feature of MongoDB is it supports dynamic schema design. A collection is group of documents and each document will store data as key-value pairs. All the documents in the same collection need not have same set of fields and need not store same or related data thus enforcing dynamic schema. The schema is designed based on user requirements.

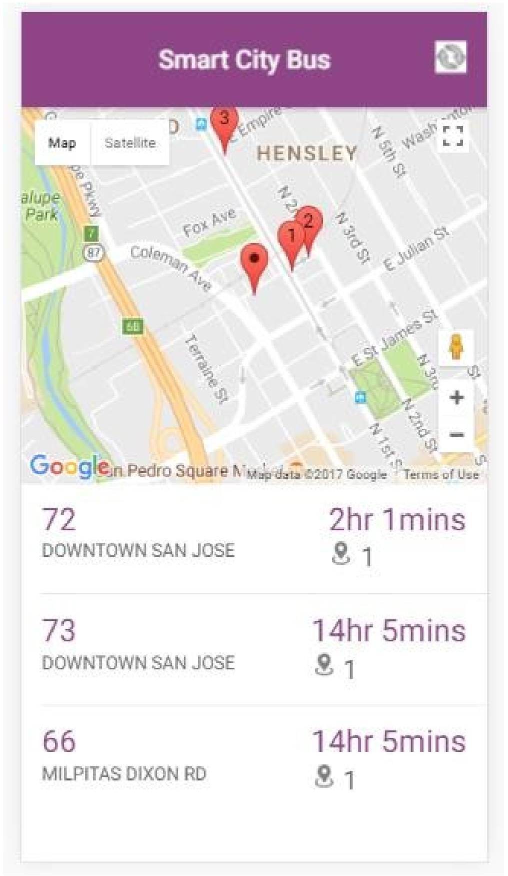
For our project, we have stored locations of all the stops in one collection, so that whenever the server receives user location, this collection is accessed to find near-by stops. Once the near-by stops are known, a different collection with all GTFS feed data is accessed to get related trips.

6 Design and Implementation

6.1 Client Design and Implementation

6.1.1 End User Application

End User Application is a hybrid mobile application developed using Ionic framework. It provides real time transit information of nearby approaching routes based on user location. When the user opens the application, Google Map is rendered on the Home page showing a marker for user's location. Also, HTTP request is made to the server to send the GPS coordinates. Server then responds with the transit information of all routes stopping by user's nearby stops. Stop locations are also displayed on the Google map. ETA of the routes stopping at those stops is displayed on the Home page of our application. Once the response is received from the server ETA of the routes is updated for every two minutes locally without hitting the server. A refresh button is also provided for user convenience to make a new HTTP request with the new user location. Figures 3 and 4 show the home screen and the 2nd screen of the user application.

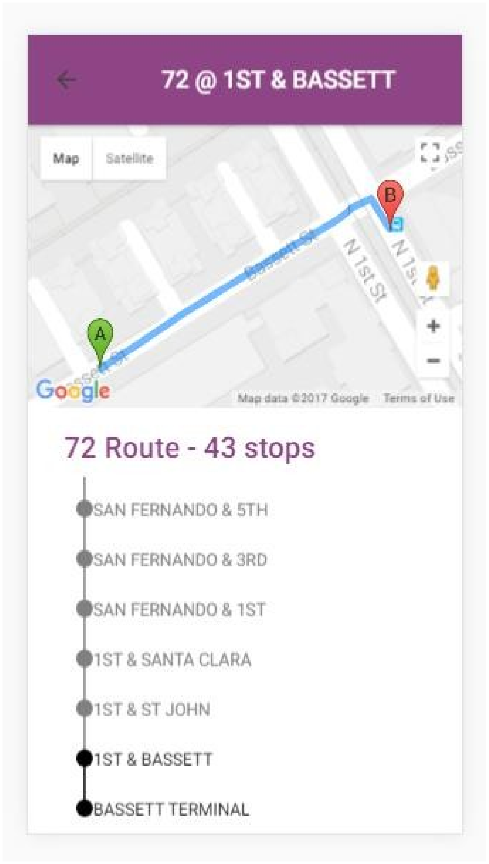


72 DOWNTOWN SAN JOSE 2hr 1mins 1

73 DOWNTOWN SAN JOSE 14hr 5mins 1

66 MILPITAS DIXON RD 14hr 5mins 1

Figure 3 Home screen of user application



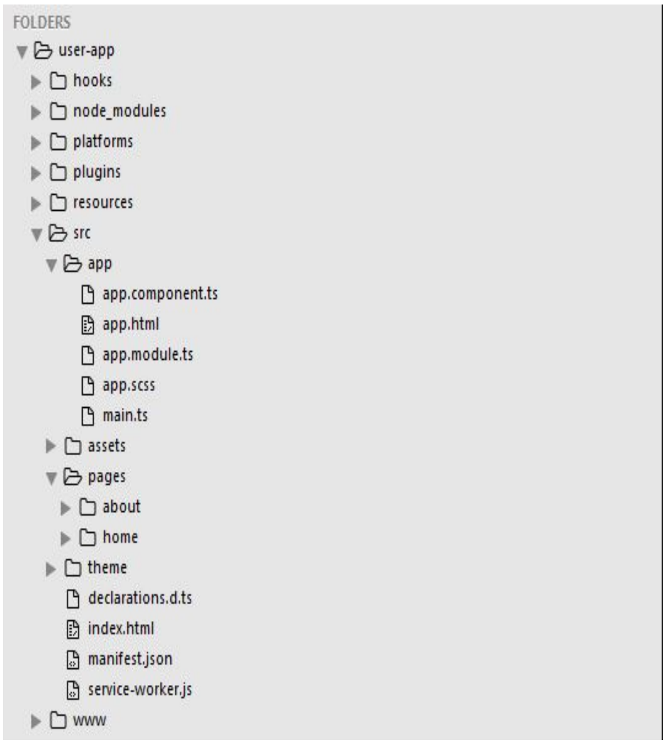
72 Route - 43 stops

- SAN FERNANDO & 5TH
- SAN FERNANDO & 3RD
- SAN FERNANDO & 1ST
- 1ST & SANTA CLARA
- 1ST & ST JOHN
- 1ST & BASSETT
- BASSETT TERMINAL

Figure 4 2nd screen of user application

Once the user selects a route, he/she will be navigated to the 2nd page. About page contains the travel directions from the user's location to the stop of the selected route. Also, it displays the list of stops passed and yet to be passed along the route's path.

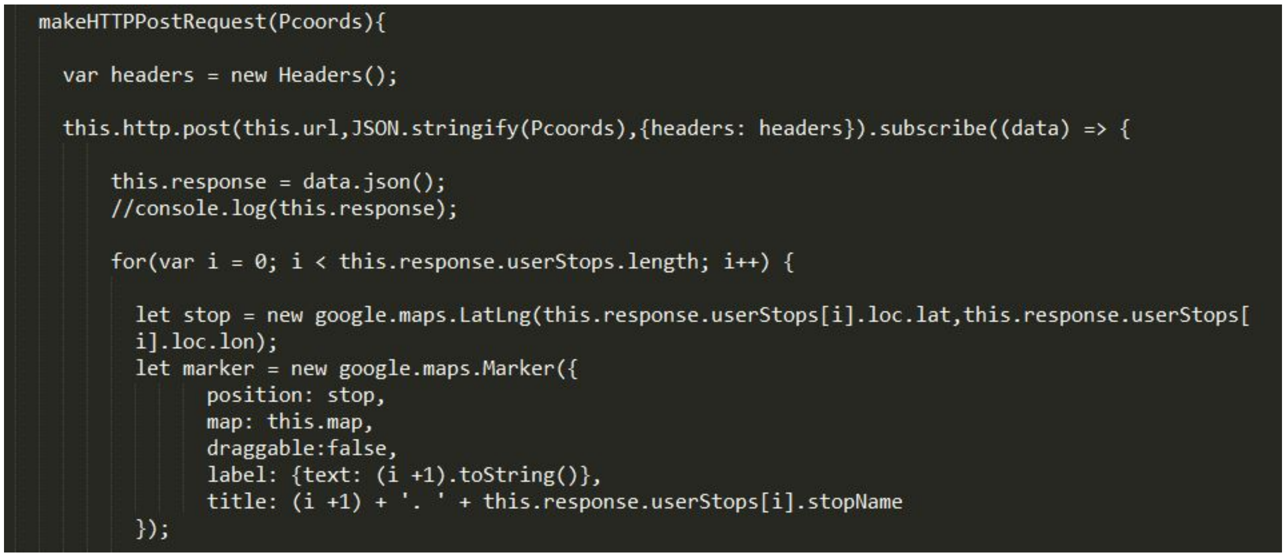
The following snippet shows the folder structure for the user application.



```
FOLDERS
▼ user-app
  ▶ hooks
  ▶ node_modules
  ▶ platforms
  ▶ plugins
  ▶ resources
  ▼ src
    ▼ app
        app.component.ts
        app.html
        app.module.ts
        app.scss
        main.ts
    ▶ assets
    ▼ pages
      ▶ about
      ▶ home
    ▶ theme
      declarations.d.ts
      index.html
      manifest.json
      service-worker.js
  ▶ www
```

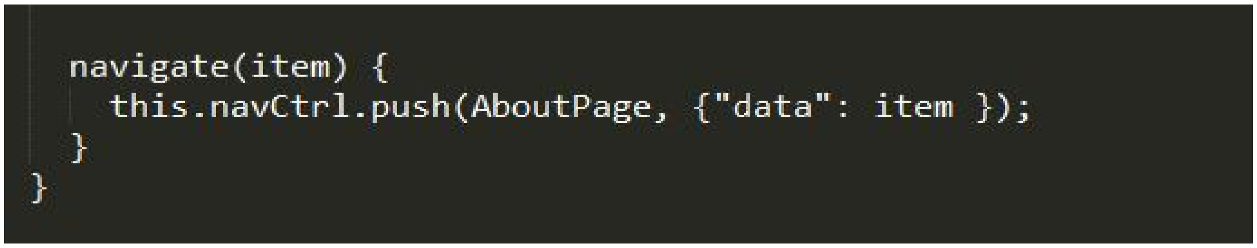
App folder contains the source code of our application. app.module.ts declares the ng-module directive, available providers and entry components. Home page is set as the root component, so once the app loads, home page view will be rendered.

In the Home component, as soon the view loads, http POST request is made to the server with geo location coordinates of the user as input. Once the response is received from the server, Google Map is rendered in the home page with user's location and three nearby stop location pins on the map. Also, route schedules are displayed as shown in the above figures.



```
makeHTTPPostRequest(Pcoords){

  var headers = new Headers();

  this.http.post(this.url,JSON.stringify(Pcoords),{headers: headers}).subscribe((data) => {

      this.response = data.json();
      //console.log(this.response);

      for(var i = 0; i < this.response.userStops.length; i++) {

        let stop = new google.maps.LatLng(this.response.userStops[i].loc.lat,this.response.userStops[
        i].loc.lon);
        let marker = new google.maps.Marker({
              position: stop,
              map: this.map,
              draggable:false,
              label: {text: (i +1).toString()},
              title: (i +1) + '. ' + this.response.userStops[i].stopName
        });
```

ETA for the routes is updated for every two minutes locally without making a server call. When the user selects a route, a data object is built for the selected routes and navigated to the about page using navController methods.



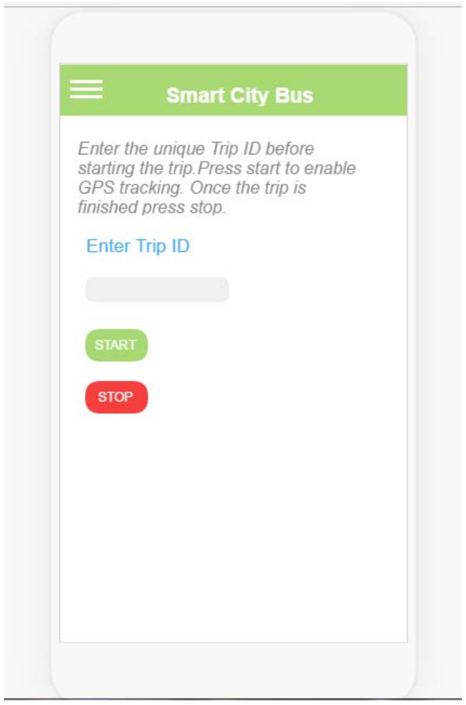
```
  navigate(item) {
    this.navCtrl.push(AboutPage, {"data": item });
  }
}
```

About page displays the directions from the user location to the nearby stops. Also, it displays the list of stops for that route, stops are differentiated with two different colors as shown in the above figure.

6.1.2 GPS Tracking Application for Transit Vehicles

Android phone placed in transit vehicle to record the vehicle's current location once the trip is started. GPS tracking application is installed on the android device to enable tracking. Hybrid mobile application is developed which sends GPS coordinates of the vehicle to the server at regular intervals even when the user suspends/closes the application. Background tracking is implemented to continuously update the location.

When a new trip is started, the driver riding the vehicle enters the unique TripID and presses the start button to start tracking the vehicle. The driver must press the stop button once the trip has finished. Fig. 5 shows the screen of the GPS tracking application.



Enter the unique Trip ID before starting the trip.Press start to enable GPS tracking. Once the trip is finished press stop.

Figure 5 UI Screen for GPS Tracking Application

Following Cordova plugin is installed to enable background tracking in the application.

```
$ionic plugin add cordova-plugin-background-mode
```

Once the plugin is installed, background mode is enabled in the ionic application. Background mode is supported in Android, IOS and Windows platforms.

```
import { Background Mode } from 'ionic-native';
BackgroundMode.enable();
```

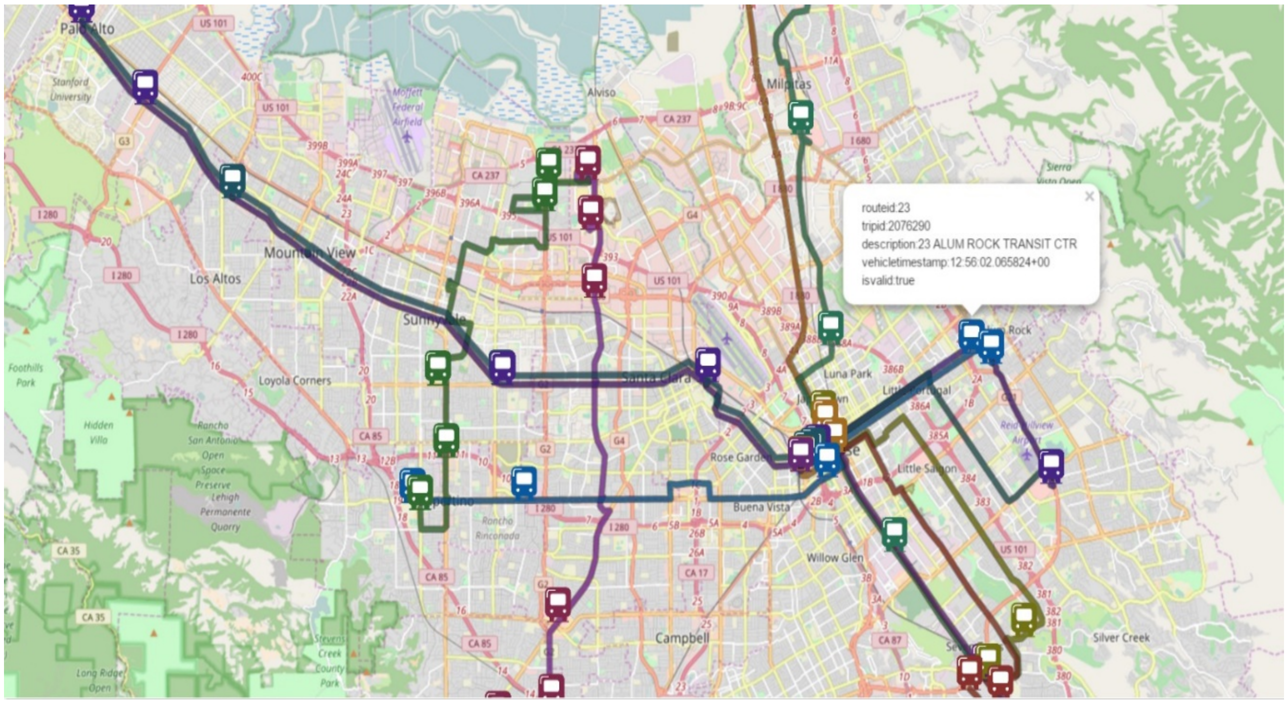
To access the GPS location services of the device, following permissions must be added to the android manifest file.

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses- permissionandroid:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS"/>
```

Now the GPS location of the device can be retrieved using the utility function getCurrentPosition. GPS coordinates are sent to the Node server at regular intervals. Geolocation is update is run inside the Angular Zone. HTTP request is made to the server whenever a new geolocation is recorded in the background.

6.1.3 Web Application for Transit Agency

A web application is developed as part of monitoring screen to the management of transit agency. This application is developed using Angular2 framework and typescript. Let us understand the design and implementation from following description. This webpage gives following view to the transit agency supervisor.

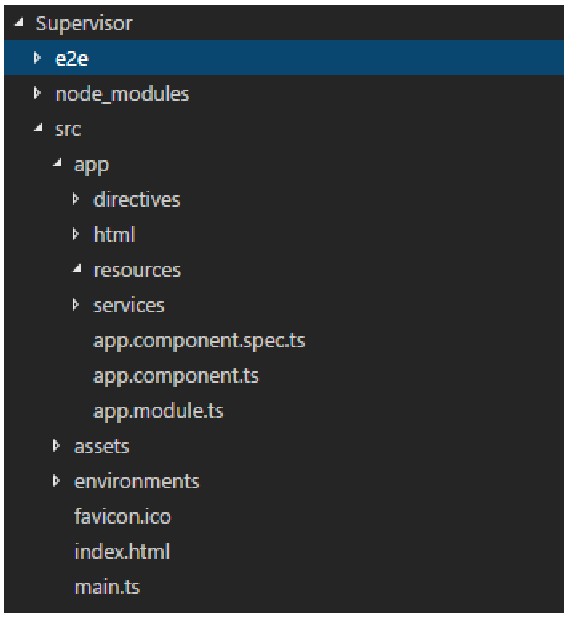


routeid:23
tripid:2076290
description:23 ALUM ROCK TRANSIT CTR
vehicletimestamp:12:56:02.065824+00
isvalid:true

Figure 6 Web portal for Supervision

In the above figure, lines display the routes served by a Transit agency. With bus icon displaying current trip in service.

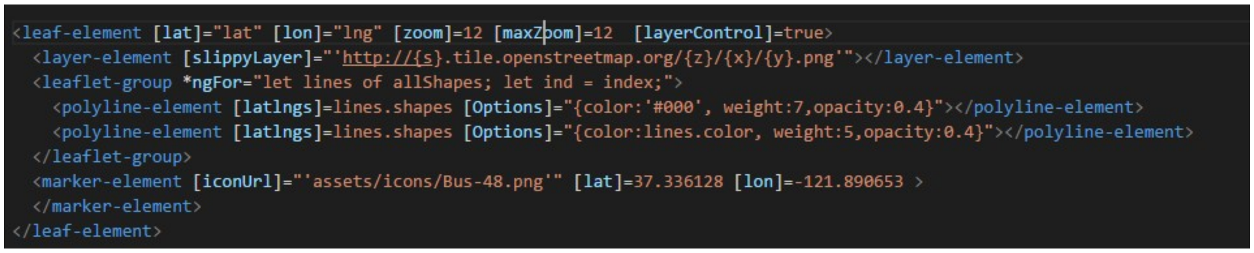
Angular CLI was used as command line interface to start building this application. With its support, the following folder structure is maintained throughout the project.



```
Supervisor
  e2e
  node_modules
  src
    app
      directives
      html
      resources
      services
      app.component.spec.ts
      app.component.ts
      app.module.ts
    assets
    environments
    favicon.ico
    index.html
    main.ts
```

Let us understand each component developed to display required information to the user. MVC pattern is being followed for designing front end of this application.

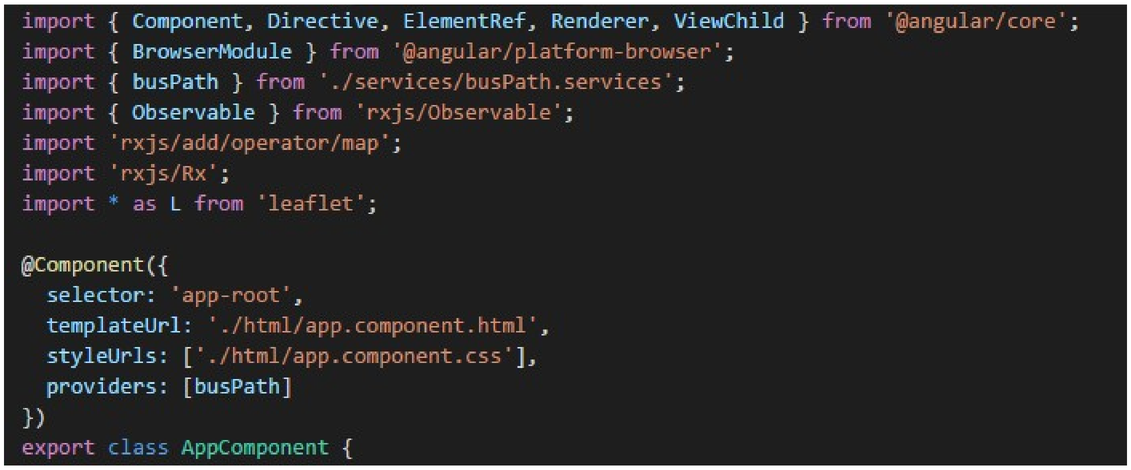
As part of View, leaflet components were utilized to show contents on the map. Open street map from OSM was used as layer on top of leaflet to display map.



```
<leaf-element [lat]="lat" [lon]="lng" [zoom]=12 [maxZoom]=12  [layerControl]=true>
  <layer-element [slippyLayer]="'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png'"></layer-element>
  <leaflet-group *ngFor="let lines of allShapes; let ind = index;">
    <polyline-element [latlngs]=lines.shapes [Options]="{color:'#000', weight:7,opacity:0.4}"></polyline-element>
    <polyline-element [latlngs]=lines.shapes [Options]="{color:lines.color, weight:5,opacity:0.4}"></polyline-element>
  </leaflet-group>
  <marker-element [iconUrl]="'assets/icons/Bus-48.png'" [lat]=37.336128 [lon]=-121.890653 >
  </marker-element>
</leaf-element>
```

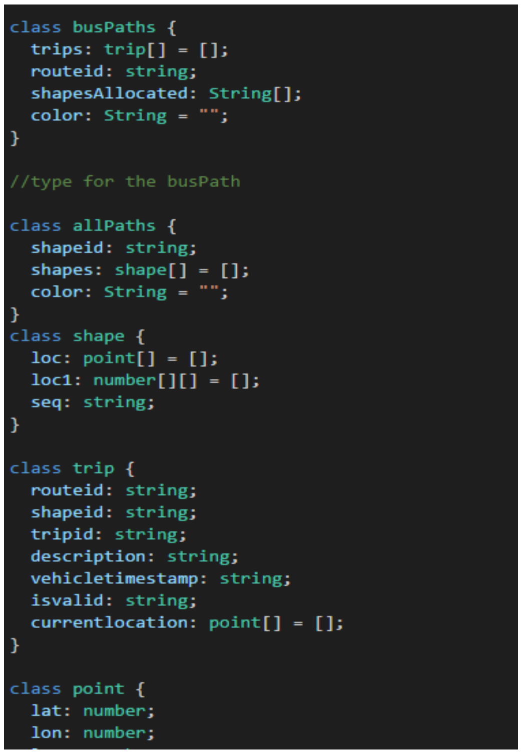
From the above code snippet it is very clear that map is being loaded from leaflet element tag with layer being taken from Open Street map url. And polyline tags are being used to draw lines on the map to display different routes operated by VTA. Figure 6 shows clearly how different were displayed to user with clear segregation using different colors.

Now let us understand the component implementation. The controller implementation can be found in app.component.ts file. The component classes are defined as follows



```
import { Component, Directive, ElementRef, Renderer, ViewChild } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { busPath } from './services/busPath.services';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import 'rxjs/Rx';
import * as L from 'leaflet';

@Component({
  selector: 'app-root',
  templateUrl: './html/app.component.html',
  styleUrls: ['./html/app.component.css'],
  providers: [busPath]
})
export class AppComponent {
```

Component annotation specified above AppComponent class is binding data from provider specified to the template url mentioned. An instance of the provider specified is created and data is fetched from the model layer. Following are the user types defined to maintain custom data from the model layer.



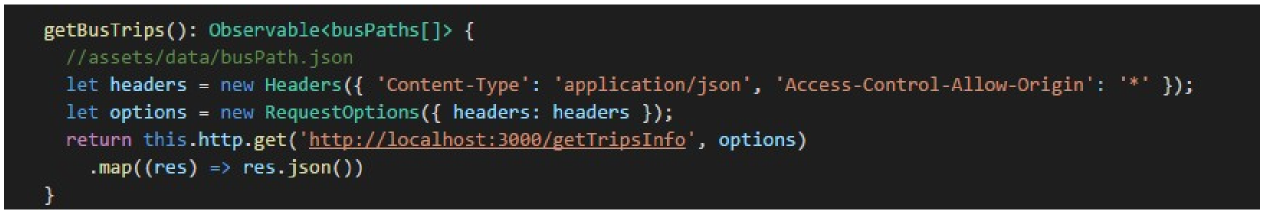
```
class busPaths {
  trips: trip[] = [];
  routeid: string;
  shapesAllocated: String[];
  color: String = "";
}

//type for the busPath

class allPaths {
  shapeid: string;
  shapes: shape[] = [];
  color: String = "";
}
class shape {
  loc: point[] = [];
  loc1: number[][] = [];
  seq: string;
}

class trip {
  routeid: string;
  shapeid: string;
  tripid: string;
  description: string;
  vehicletimestamp: string;
  isvalid: string;
  currentlocation: point[] = [];
}

class point {
  lat: number;
  lon: number;
```

“busPath” is the parent structure that holds all the trips aggregated with the route id. As part of implementation we have selected 10 routes and hence all the current trips under those bus routes are maintains in an array of “busPath” type.

“allPaths” structure maintains all the shapes corresponding routes. It maintains locations in a sequential manner of which the bus follows throughout its journey.

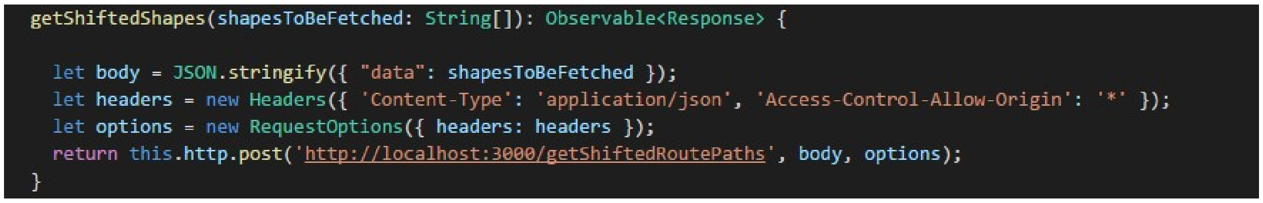
And each bus information is specific to the trip it is assigned under. So “trip” structure maintains all the information about the vehicle like current location, description, timestamp etc.

To bind data to the view, the controller fetch required information from following routines specified in service layer. Following are few important routines to be noted.



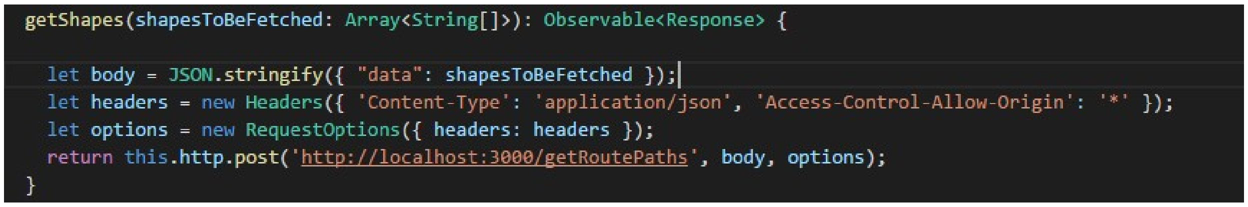
```
getBusTrips(): Observable<busPaths[]> {
  //assets/data/busPath.json
  let headers = new Headers({ 'Content-Type': 'application/json', 'Access-Control-Allow-Origin': '*' });
  let options = new RequestOptions({ headers: headers });
  return this.http.get('http://localhost:3000/getTripsInfo', options)
    .map((res) => res.json())
}
```

The above routine fetches all the information on the current running trips which is later aggregated corresponding to routeId.



```
getShiftedShapes(shapesToBeFetched: String[]): Observable<Response> {

  let body = JSON.stringify({ "data": shapesToBeFetched });
  let headers = new Headers({ 'Content-Type': 'application/json', 'Access-Control-Allow-Origin': '*' });
  let options = new RequestOptions({ headers: headers });
  return this.http.post('http://localhost:3000/getShiftedRoutePaths', body, options);
}
```

The above specified routine fetch the shapes that correspond to currently running trips.



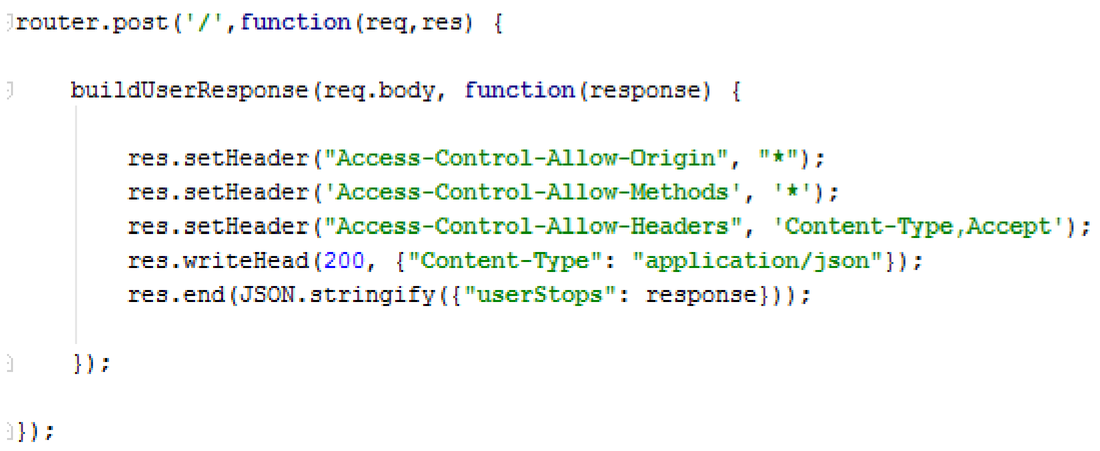
```
getShapes(shapesToBeFetched: Array<String[]>): Observable<Response> {

  let body = JSON.stringify({ "data": shapesToBeFetched });
  let headers = new Headers({ 'Content-Type': 'application/json', 'Access-Control-Allow-Origin': '*' });
  let options = new RequestOptions({ headers: headers });
  return this.http.post('http://localhost:3000/getRoutePaths', body, options);
}
```

To clearly differentiate overlapping of the routes, few co-ordinates were shifted by pixel units on to the map. So, the corresponding shapes with the sequential path locations were fetched from above routine

6.2 Middle-Tier Design and Implementation

6.2.1 RESTful Services

REST is Representational State Transfer which uses Http methods for data communication between front-end applications and the server. The data is a resource which is identified by URI and accessed by clients. To use RESTful API, we should first define the resources that are needed in our project. In our project, the server receives user location JSON object using POST Http method. REST uses Text, JSON, XML representation of data. Based on the format of data being communicated between server and the front-end applications, Http headers are set accordingly. End User application sends its location (latitude and longitude) as JSON data as shown below:



```
router.post('/',function(req,res) {

    buildUserResponse(req.body, function(response) {

        res.setHeader("Access-Control-Allow-Origin", "*");
        res.setHeader('Access-Control-Allow-Methods', '*');
        res.setHeader("Access-Control-Allow-Headers", 'Content-Type,Accept');
        res.writeHead(200, {"Content-Type": "application/json"});
        res.end(JSON.stringify({"userStops": response}));

    });

});
```

```
Headers:

Content-Type    =    Application/Json
Body:
{
       "lon": -121.894975,
       "lat": 37.341253
}
```

Based on the user location, the server sends near-by stops, routes and ETA as response to the POST request. Building the required response involves connecting to MongoDB database to query for near-by stops based on the user location. The latest trips along with their ETA is retrieved from database. JSON Object with the resource to be sent to the user is built and sent from the server to the user as Http Response.

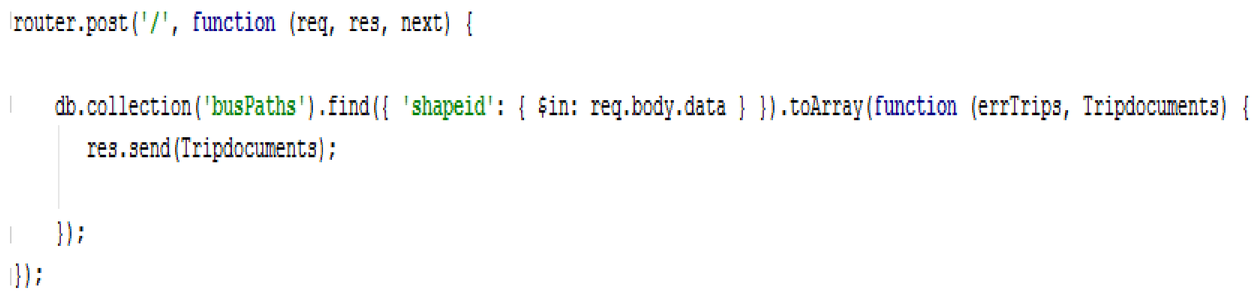
The Server also handles the Web application for Transit Agency. Continuous Http requests are made to update the latest information about all the trips on the Supervisor screen.

Whenever, the supervisor wishes to see the updates about particular trip, a POST request is made to the server with body of the request containing JSON data related to that trip. The server reads the JSON data and queries the database, and returns the response by building the JSON object as required by the front-end application. The following code snippets show, how RESTful services are used for data communication between web application for Transit Agency and Node Server.

Below is the Http Request sent to get paths of all the routes:

```
Headers:
Content-Type    =    Application/Json
Body:
{
       "data":["22_EB04","22_EB02","23_EB04","55_NB01","55_NBSD1","55_NB02","55_NB04"
,"55_NBSD3","57_NB02","57_NB03","73_NB01","73_NB02","66_NB03","66_NB01","68_NB05","68
_NB04","72_NB01","181_NB01","181_NBSD1","522_EB02"]

}
```

The above request is handled by the below POST Method

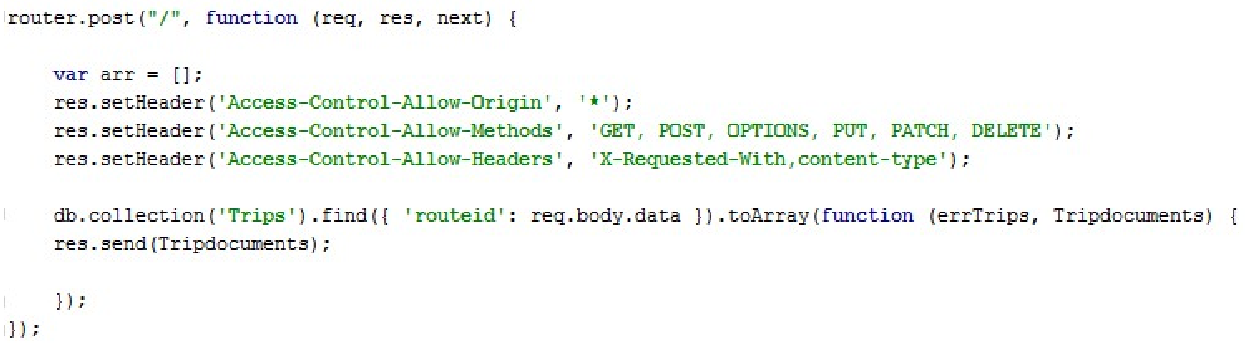


```
router.post('/', function (req, res, next) {

    db.collection('busPaths').find({ 'shapeid': { $in: req.body.data } }).toArray(function (errTrips, Tripdocuments) {
        res.send(Tripdocuments);

    });
});
```

Similarly, to get any trip in real-time

```
Headers:

Content-Type    =    Application/Json
Body:
{
       "data":22
}
```

The above request is handled by below POST method and sends required JSON Object as response

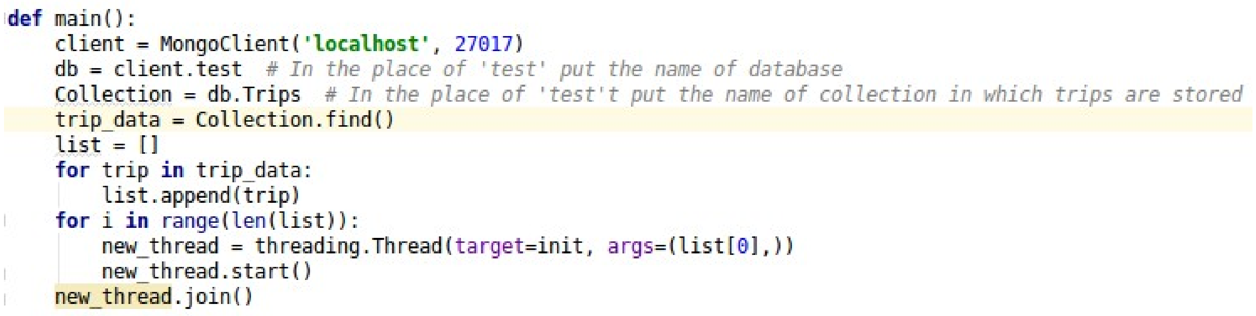


```
router.post("/", function (req, res, next) {

    var arr = [];
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH, DELETE');
    res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type');

    db.collection('Trips').find({ 'routeid': req.body.data }).toArray(function (errTrips, Tripdocuments) {
    res.send(Tripdocuments);

    });
});
```

6.2.2 *ETA Calculation*

ETA calculation server queries the running trips data from the mongo dB database and uses those data to calculate the real-time ETA. After calculating the real-time ETA, it will replace the existing old ETA with the new real-time ETA in the database. A python server application is written to perform this task.

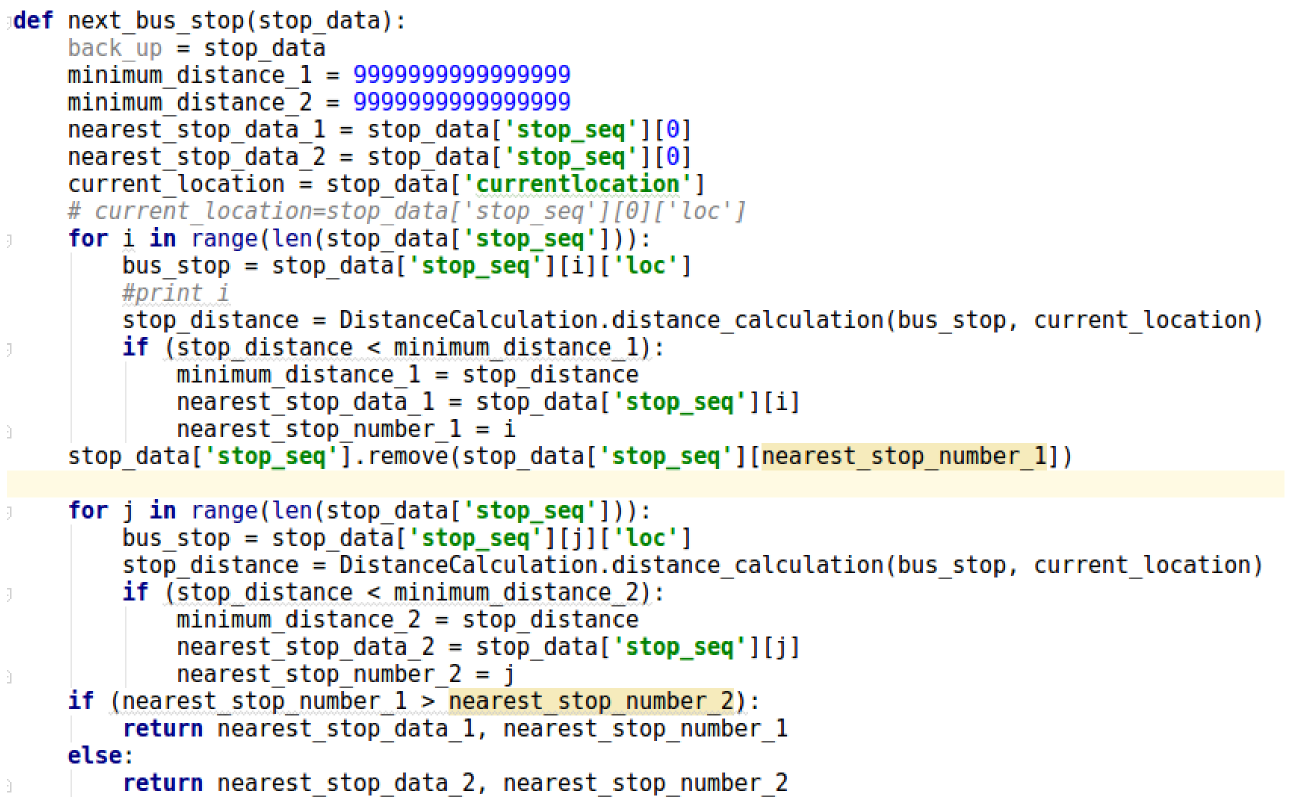
When server which calculates, ETA is started, it first connects to the MongoDB database and then queries all the trips which are running now and puts all those trips in a list. Since we want to allocate similar resources for calculating ETA of every trip, new thread is created for calculating the ETA for every running trip and pass the data which is required to calculate the ETA for that trip.



```
def main():
    client = MongoClient('localhost', 27017)
    db = client.test  # In the place of 'test' put the name of database
    Collection = db.Trips  # In the place of 'test't put the name of collection in which trips are stored
    trip_data = Collection.find()
    list = []
    for trip in trip_data:
        list.append(trip)
    for i in range(len(list)):
        new_thread = threading.Thread(target=init, args=(list[0],))
        new_thread.start()
    new_thread.join()
```

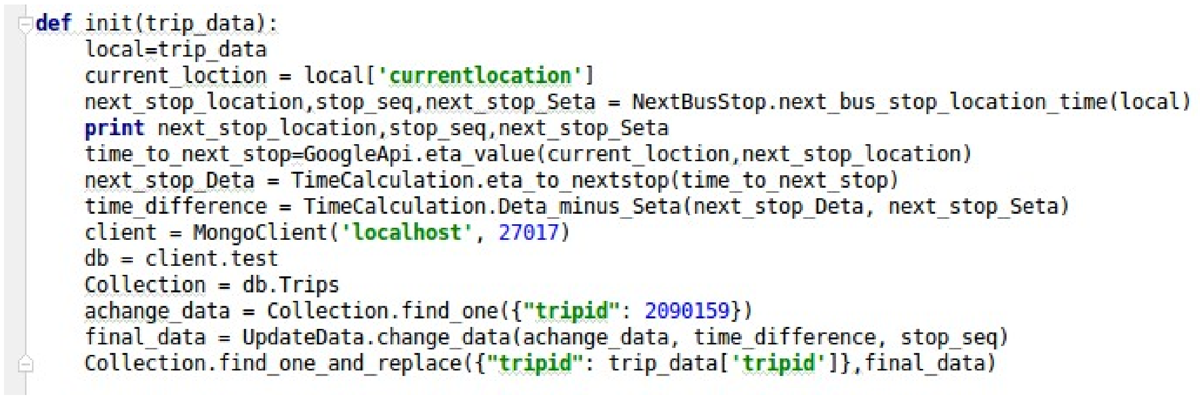
When a new thread is created, it will have the trip data which was passed by the main processes. Trip data contains the following information: current location of the bus, the GPS co-ordinates of all the stops that the trip must make, previous static ETA, Order of the stops, the route to which this trip travels.

We extracted the current location of the bus from the trip data and used this GPS co-ordinate to find out the next bus stop for this trip. We did this by calculating the distance between every stop in the trip to current location of the bus stop. To perform this calculation, we needed an algorithm which can calculate the distance between two points when given the GPS co-ordinates. We have used the ‘Haversine algorithm” for this purpose. The following is a code snippet from next stop calculation.



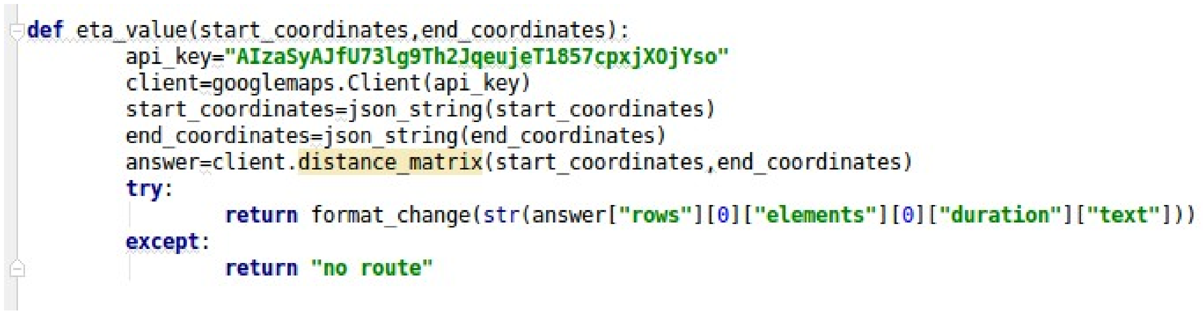
```
def next_bus_stop(stop_data):
    back_up = stop_data
    minimum_distance_1 = 9999999999999999
    minimum_distance_2 = 9999999999999999
    nearest_stop_data_1 = stop_data['stop_seq'][0]
    nearest_stop_data_2 = stop_data['stop_seq'][0]
    current_location = stop_data['currentlocation']
    # current_location=stop_data['stop_seq'][0]['loc']
    for i in range(len(stop_data['stop_seq'])):
        bus_stop = stop_data['stop_seq'][i]['loc']
        #print i
        stop_distance = DistanceCalculation.distance_calculation(bus_stop, current_location)
        if (stop_distance < minimum_distance_1):
            minimum_distance_1 = stop_distance
            nearest_stop_data_1 = stop_data['stop_seq'][i]
            nearest_stop_number_1 = i
    stop_data['stop_seq'].remove(stop_data['stop_seq'][nearest_stop_number_1])

    for j in range(len(stop_data['stop_seq'])):
        bus_stop = stop_data['stop_seq'][j]['loc']
        stop_distance = DistanceCalculation.distance_calculation(bus_stop, current_location)
        if (stop_distance < minimum_distance_2):
            minimum_distance_2 = stop_distance
            nearest_stop_data_2 = stop_data['stop_seq'][j]
            nearest_stop_number_2 = j
    if (nearest_stop_number_1 > nearest_stop_number_2):
        return nearest_stop_data_1, nearest_stop_number_1
    else:
        return nearest_stop_data_2, nearest_stop_number_2
```

We are looping through all the stops in the trip and calculate the nearest bus stop to the current location. Once we find the two nearest stops to the current location. We are using the direction of the bus to find the next bus stop in this code snippet.



```
def init(trip_data):
    local=trip_data
    current_loction = local['currentlocation']
    next_stop_location,stop_seq,next_stop_Seta = NextBusStop.next_bus_stop_location_time(local)
    print next_stop_location,stop_seq,next_stop_Seta
    time_to_next_stop=GoogleApi.eta_value(current_loction,next_stop_location)
    next_stop_Deta = TimeCalculation.eta_to_nextstop(time_to_next_stop)
    time_difference = TimeCalculation.Deta_minus_Seta(next_stop_Deta, next_stop_Seta)
    client = MongoClient('localhost', 27017)
    db = client.test
    Collection = db.Trips
    achange_data = Collection.find_one({"tripid": 2090159})
    final_data = UpdateData.change_data(achange_data, time_difference, stop_seq)
    Collection.find_one_and_replace({"tripid": trip_data['tripid']},final_data)
```

After calculating the next stop for the bus in this trip we use Google API call to get the time it will take for the bus to travel from its current place to its next stop. To do the Google API call we will need the formatted current GPS location of the bus and the formatted GPS location of the next bus stop. We must make sure that the GPS location data is formatted in accordance to the requested Google standard. This will give us time the bus will take to travel from its current location to its next bus stop all the while taking traffic conditions into account.



```
def eta_value(start_coordinates,end_coordinates):
        api_key="AIzaSyAJfU73lg9Th2JqeujeT1857cpxjX0jYso"
        client=googlemaps.Client(api_key)
        start_coordinates=json_string(start_coordinates)
        end_coordinates=json_string(end_coordinates)
        answer=client.distance_matrix(start_coordinates,end_coordinates)
        try:
                return format_change(str(answer["rows"][0]["elements"][0]["duration"]["text"]))
        except:
                return "no route"
```

By using the data received, old ETA is updated for the next stop to the newly calculated ETA. We can perform the same steps for all the stops in the route and update with the new ETA. But, that way we must use many Google API calls for every single updating of real- time ETA. This would be costly. So, we use the time difference between the previous ETA and new ETA for the next stop and use this difference and haversine algorithm to calculate the estimates for the rest of the stops in the route.

Once every stop is updated with the new ETA the thread will connect to the mongo dB database and performs and find and replace operation using the unique trip id field. Thus, completing the ETA calculation and updating for this trip. This will be done to every trip which is running at a time and updating is done every 30 seconds.

6.3 Data-Tier Design and Implementation

As part of the implementation there are two database configurations to be considered

6.3.1 Storing GTFS Feed

The feed database was built based on GTFS standards. It mainly consists of 8 tables filled with corresponding CSV file provided by agency. Let us understand the database configuration by observing tables

Table	Description
Agency	It consists of at least one transit agency our application is serving Example: VTA
Routes	It consists of routes being served by agency. Multiple trips may constitute towards a single route. Example: “22- Palo Alto to Eastridge”
Trips	The trip constitutes an actual bus running on a route. There might be multiple trips associated with route. This can be considered for tracking of the vehicle and calculate estimated Arrival time.
Stop_times	This provides static schedule for a trip at a Stop. It relates trips with the stops.
Stops	This table consists of data related to location of the stop.
Calender_Dates	This provides Trip frequency details on specify day of a week
Fare	This entity provides Fare details based on the routes

Table 1 Feed file data description

Below figure depicts the database schema followed for maintaining GTFS files for our application.

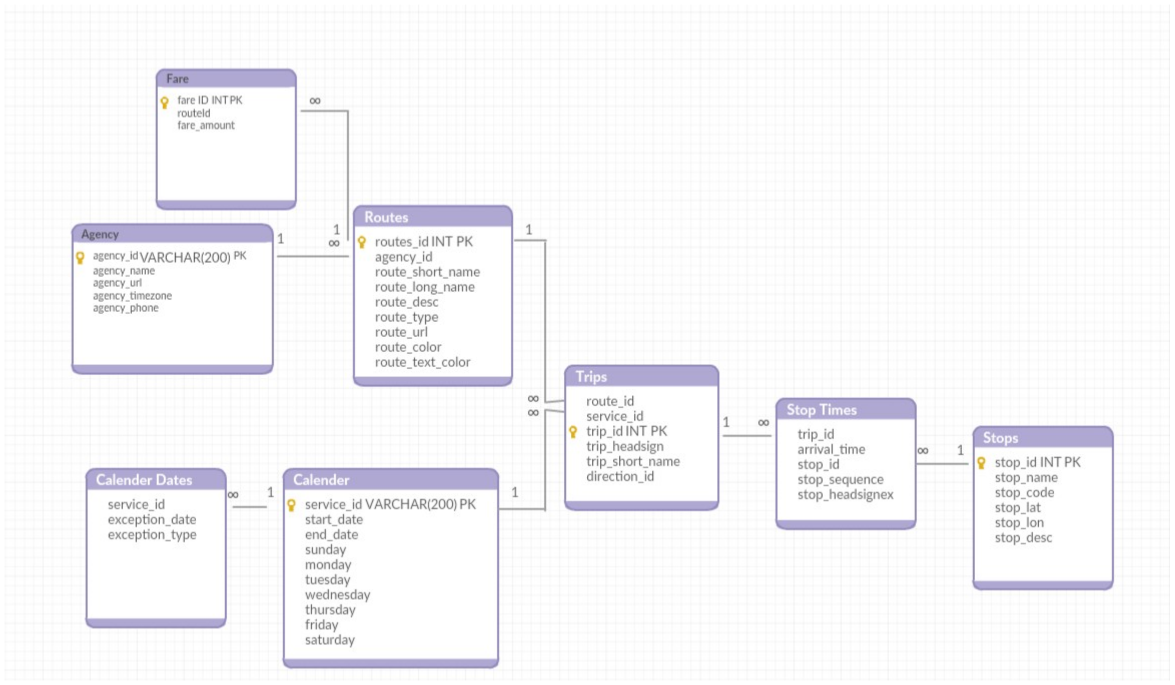


Figure 7 Database Design Schema

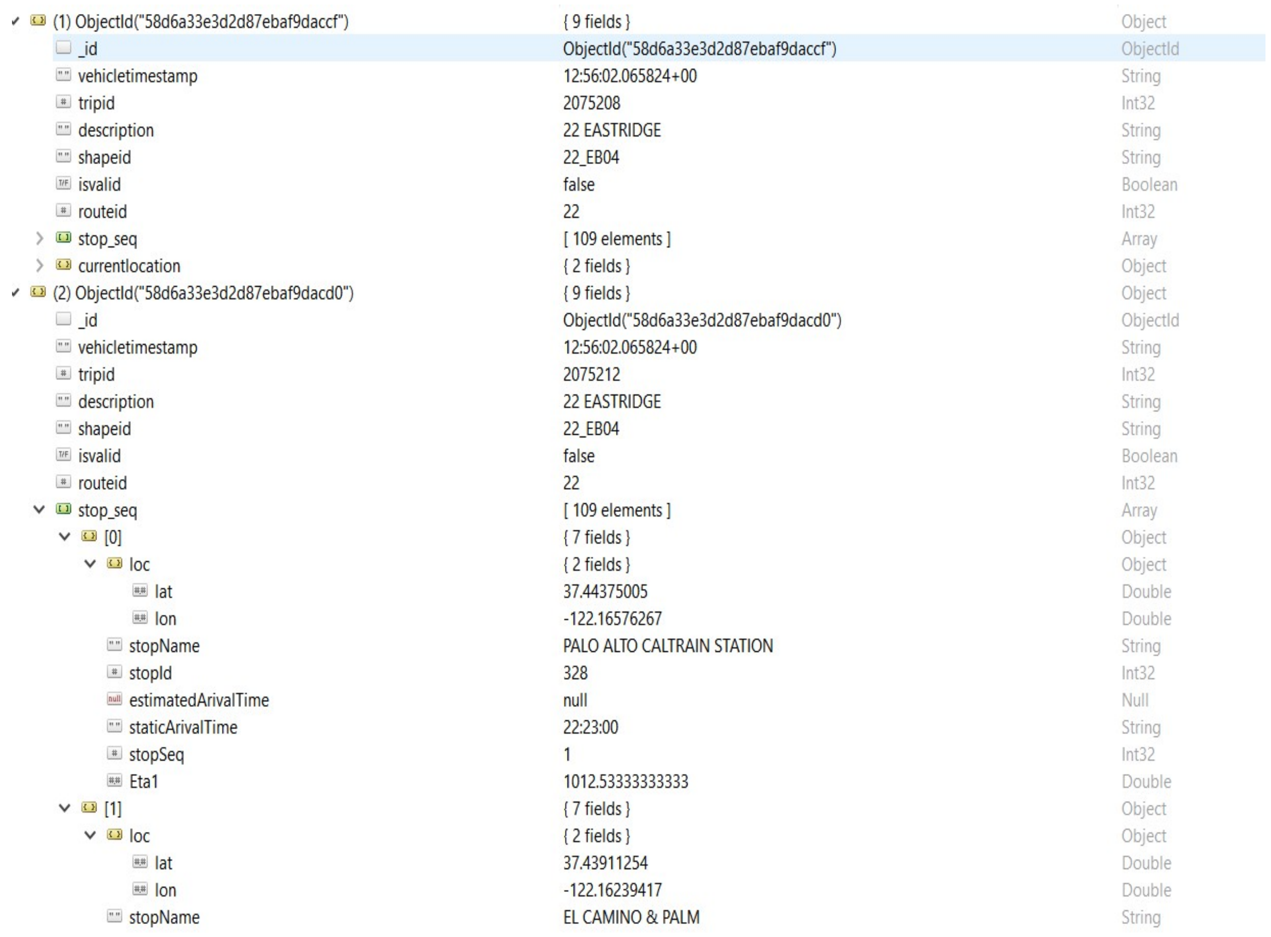
6.3.2 *Maintain Trip Information*

On Trip Creation, an object is queried from the PostgreSQL and inserted on to mongo dB for further handling of the client request.

The tracking server application will be periodically updating the current location of the vehicle to the corresponding trip object.

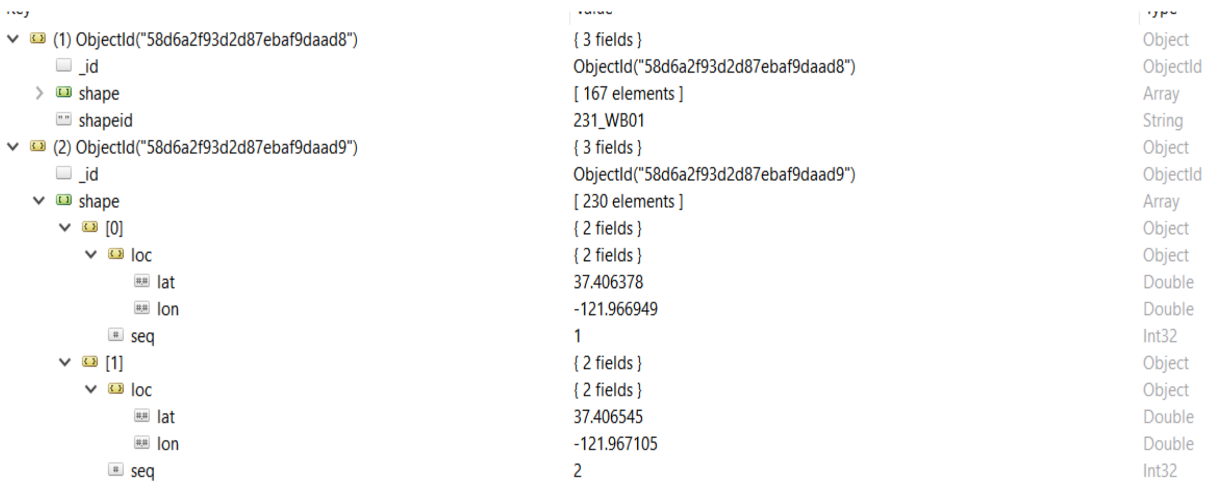
The following entities depicts the data stored in the mongo dB

a. All the trip information: This data includes all the running vehicle information under corresponding route. The below snippet shows trip information corresponding to route 22.



```
(1) ObjectId("58d6a33e3d2d87ebaf9daccf")      { 9 fields }                              Object
    _id                                       ObjectId("58d6a33e3d2d87ebaf9daccf")      ObjectId
    vehicletimestamp                          12:56:02.065824+00                        String
    tripid                                    2075208                                   Int32
    description                               22 EASTRIDGE                              String
    shapeid                                   22_EB04                                   String
    isvalid                                   false                                     Boolean
    routeid                                   22                                        Int32
  > stop_seq                                  [ 109 elements ]                          Array
  > currentlocation                           { 2 fields }                              Object
(2) ObjectId("58d6a33e3d2d87ebaf9dacd0")      { 9 fields }                              Object
    _id                                       ObjectId("58d6a33e3d2d87ebaf9dacd0")      ObjectId
    vehicletimestamp                          12:56:02.065824+00                        String
    tripid                                    2075212                                   Int32
    description                               22 EASTRIDGE                              String
    shapeid                                   22_EB04                                   String
    isvalid                                   false                                     Boolean
    routeid                                   22                                        Int32
  v stop_seq                                  [ 109 elements ]                          Array
    v [0]                                     { 7 fields }                              Object
      v loc                                   { 2 fields }                              Object
          lat                                 37.44375005                               Double
          lon                                 -122.16576267                             Double
        stopName                              PALO ALTO CALTRAIN STATION                String
        stopId                                328                                       Int32
        estimatedArivalTime                   null                                      Null
        staticArivalTime                      22:23:00                                  String
        stopSeq                               1                                         Int32
        Eta1                                  1012.53333333333                          Double
    v [1]                                     { 7 fields }                              Object
      v loc                                   { 2 fields }                              Object
          lat                                 37.43911254                               Double
          lon                                 -122.16239417                             Double
        stopName                              EL CAMINO & PALM                          String
```

b. Route Shape Information: This data consists of the sequential locations through which each of the routes form its shape. The shape id field corresponds to shape id attached in the trip information so much like it acts like a foreign key. The shape field is an array of locations in sequential manner.



(1) ObjectId("58d6a2f93d2d87ebaf9daad8")	{ 3 fields }	Object
_id	ObjectId("58d6a2f93d2d87ebaf9daad8")	ObjectId
shape	[167 elements]	Array
shapeid	231_WB01	String
(2) ObjectId("58d6a2f93d2d87ebaf9daad9")	{ 3 fields }	Object
_id	ObjectId("58d6a2f93d2d87ebaf9daad9")	ObjectId
shape	[230 elements]	Array
[0]	{ 2 fields }	Object
loc	{ 2 fields }	Object
lat	37.406378	Double
lon	-121.966949	Double
seq	1	Int32
[1]	{ 2 fields }	Object
loc	{ 2 fields }	Object
lat	37.406545	Double
lon	-121.967105	Double
seq	2	Int32

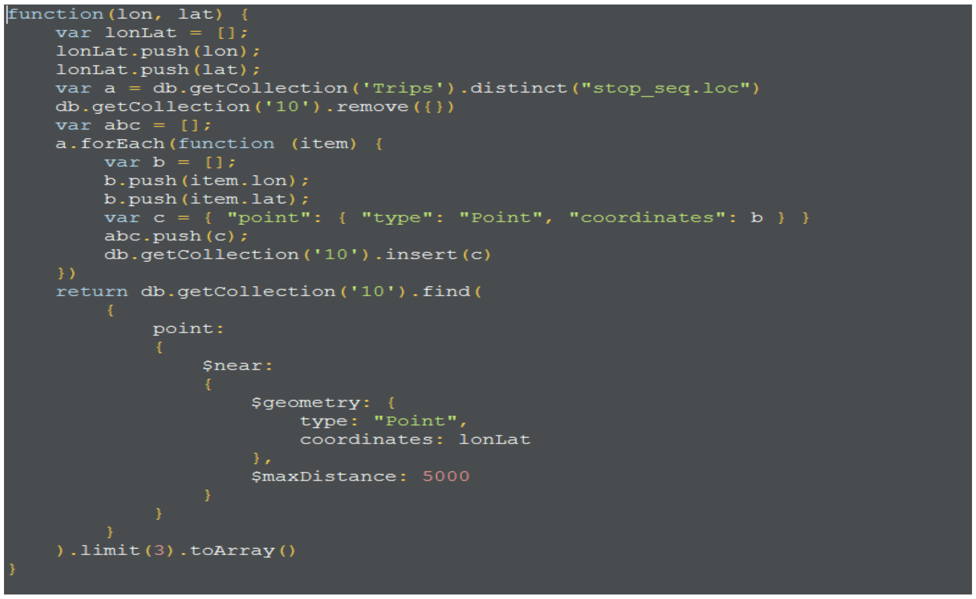
Following are the queries applied on mongo dB to serve the client request based on current location of the user

a. Nearby busses

b. Nearby stops

c. Bus Estimation Time at Nearby Bus stops.

As part of finding nearby stops to a specific location, geo features of mongo dB were utilized. The following function describes finding nearby 3 stop locations to the location passed.



```
function(lon, lat) {
    var lonLat = [];
    lonLat.push(lon);
    lonLat.push(lat);
    var a = db.getCollection('Trips').distinct("stop_seq.loc")
    db.getCollection('10').remove({})
    var abc = [];
    a.forEach(function (item) {
        var b = [];
        b.push(item.lon);
        b.push(item.lat);
        var c = { "point": { "type": "Point", "coordinates": b } }
        abc.push(c);
        db.getCollection('10').insert(c)
    })
    return db.getCollection('10').find(
        {
            point:
            {
                $near:
                {
                    $geometry: {
                        type: "Point",
                        coordinates: lonLat
                    },
                    $maxDistance: 5000
                }
            }
        }
    ).limit(3).toArray()
}
```

7 Testing and Verification

7.1 Testing Strategy

This section discusses the various tests performed on different modules in our project. Different modules in our projects are three front end applications (GPS Tracking, Web application, End User android application) and server applications. Following tests were performed on each component to check if the component meets its requirements described in its design and development chapter.

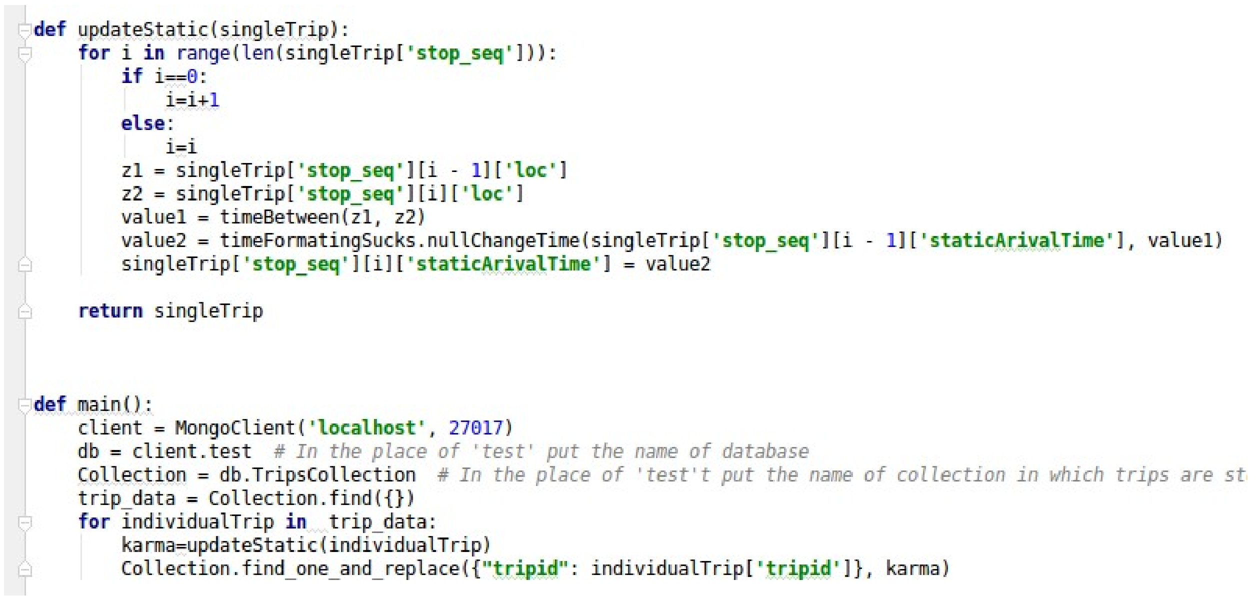
1. Unit Testing: Unit testing was made for both front end and back end applications to test the quality and behavior of the applications. In this testing, services developed in each application are tested for its expected behavior individually.
2. Integration Testing: During the development process, all the components were built in parallel, therefore for those of the components which depend on each other's response, stubs were developed to provide the expected behavior. These stubs were replaced by the developed components and integration testing was performed.

7.1.1 Testing with Multiple Scenarios using Simulated Data

To test the end user application, simulated data is stored to continuously update the location of the bus. This data was used to send response to the user application and web application to display current location of the bus and estimated arrival times.

Testing the application with multiple scenarios is necessary to make sure that the application is working as desired in different situations. This type of testing also reduces the possibility of the application breaking down in the edge cases. But creating these edge cases and different scenarios in real world and testing the application at those time is difficult. To test at the required we should simulate the scenarios with dummy data.

Luckily VTA provides diverse types of data to simulate the edge situations. One way to create multiple scenarios is to update the current location of the bus using scripts. To update the current locations, we utilized the routes GPS co-ordinates provided by VTA. Below is the code snippet of the script to achieve the simulation.



```
def updateStatic(singleTrip):
    for i in range(len(singleTrip['stop_seq'])):
        if i==0:
            i=i+1
        else:
            i=i
        z1 = singleTrip['stop_seq'][i - 1]['loc']
        z2 = singleTrip['stop_seq'][i]['loc']
        value1 = timeBetween(z1, z2)
        value2 = timeFormatingSucks.nullChangeTime(singleTrip['stop_seq'][i - 1]['staticArivalTime'], value1)
        singleTrip['stop_seq'][i]['staticArivalTime'] = value2

    return singleTrip


def main():
    client = MongoClient('localhost', 27017)
    db = client.test  # In the place of 'test' put the name of database
    Collection = db.TripsCollection  # In the place of 'test't put the name of collection in which trips are st
    trip_data = Collection.find({})
    for individualTrip in  trip_data:
        karma=updateStatic(individualTrip)
        Collection.find_one_and_replace({"tripid": individualTrip['tripid']}, karma)
```

Using this script, we created scenarios like:

- Bus which starts later crossing the earlier started bus
- Bus breaks down and stops.
- GPS co-ordinates of the bus cannot be sent to server due to poor signal quality, device breakdown or other reason
- Bus takes a wrong turn and takes an incorrect route
- A bus taking detour due to unavoidable circumstances.
- Two buses in the same route crossing each other.

User Application Test Cases

- When the user opens the application, map should load with the user location and nearby stop locations.
- Routes with their stop number and ETA are displayed.
- When the refresh button is clicked, new response from the server must be loaded.
- ETA of each should be updated for every two minutes.
- When the user selects a route, about page should be loaded with directions to the stop.

- Stops passed by and yet to be passed must be differentiated based on bus current location.

Portal Test Scenarios

- Open the portal at provided URL.
- Upon loading, leaflet map should load based on given center as location.
- All the shapes of the selected bus routes should be fetched from the database.
- Upon fetched overlapping coordinates are shifted based on bearing provided to clearly differentiate the routes.
- Load the shapes onto the map by drawing the polylines.
- Then trip information should be fetched for all the corresponding routes displayed on the map.
- Load the markers onto the map at corresponding current locations of the VTA vehicles running.
- Maintain trip data on to session at the server.
- For every 15 seconds, update the session with the new current locations from the database.
- Fetch newly updated current locations from the session and relate it to the markers displayed on the map.
- This process repeats and all currently running busses will be displayed on the map in the form of markers.
- Additional information on the corresponding trip can be viewed by hovering mouse on to the marker.

7.2 Performance Evaluation

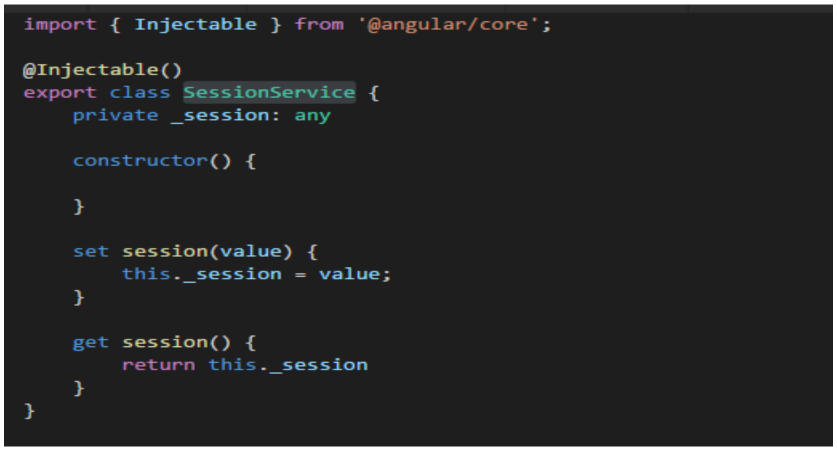
7.2.1 Analysis on Increased Load

The main concern for Server application is to be able to handle large set of clients seamlessly without any performance setbacks. We kept track of response time for single client while developing the application and tried to improve it by great extent by efficiently managing number of requests made and by getting required data by making fewer connections to database.

7.2.2 Cache Management for the portal

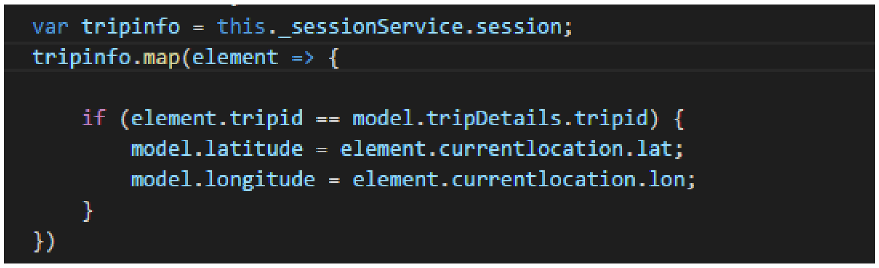
For higher responsiveness, we have utilized Angular2 component design to maximum extent. Each marker on the screen is an individual component. So each marker frequently requires updated location to reallocate itself. To avoid huge number of server calls, all data is maintained at one location and it is updated at specified frequency with just one server call. And all the markers retrieve current location from maintained cache data avoid their respective server call.

The following code snippet shows component designed to store session data.



```
import { Injectable } from '@angular/core';

@Injectable()
export class SessionService {
    private _session: any

    constructor() {

    }

    set session(value) {
        this._session = value;
    }

    get session() {
        return this._session
    }
}
```

And the below snippet depicts marker component fetching trip information from the session data.



```
var tripinfo = this._sessionService.session;
tripinfo.map(element => {

    if (element.tripid == model.tripDetails.tripid) {
        model.latitude = element.currentlocation.lat;
        model.longitude = element.currentlocation.lon;
    }
})
```

8 Deployment and Maintenance

8.1 Deliverables

The final deliverables are two android applications 1) android application for user 2) android application to collect bus GPS coordinates

1) Android application for user: This is the application which user will use to see the time of arrival of selected bus to user's location. It will show the arrival time and the current location of the bus on a map.

2) Android application for bus: This application will be running on the android phone which will be located on the bus. It collects the GPS coordinates of the bus regularly. The collected data is sent to the server which will be hosted on the cloud.

3) Agency web portal: A web portal for transit agencies to update their schedules. It also provides historical data of bus arrival time at each stop along a route. This data helps agencies in monitoring their services.

8.2 Deployment

Deployment and maintenance plays an important role for an application to be fully functional. The following describes deployment process of this application on Amazon AWS.

As part of this application, we are mainly using two services provided by Amazon AWS.

a. Amazon EC2
b. Amazon RDS

As part of Amazon RDS, we configured PostgreSQL as its database instance. From the open docs provided by transit agency, the data base tables should be filled with bulk import functionality. Staging tables are configured for any data correction to be made before filling main tables.

As part of server, we utilized two Amazon Ec2 instances. Both run on Linux platform. One being utilized only for the mongo dB and other for running server applications.

We have set security groups of Mongo Ec2 instance such that its ports are only accessible by the server applications on another instance. This is done as part of security measure to avoid opening ports to public. Instance with server applications to be deployed should be installed with following software

a. Python
b. Node.js

Utilized FileZilla manager to transfer files of the server applications to root folder of the Ec2 instance through SSH.

Connection to these ec2 instances were established using Putty to start the server applications with data source pointed correctly. Server applications mainly include, node server providing restful services, python server for ETA calculation and HTTP server hosting web portal.

8.3 Challenges

1. VTA open sources lot of important data for developers to use them in their application. This data is released in GTFS Format. Unfortunately, most of the data released by VTA is riddled with incomplete entries and formatting errors. This means the application developer should make sure he must sanitize the data before utilizing them in their application. Making sure thousands of fields of data is corrected took us considerable amount of time from our part. This is one of the challenges in doing the project.
2. Finding nearest bus stop based on user location from huge database of bus stops.
3. Finding the latest active trip from all available trips for single route depending on the time the user is using the application

9 Summary and Recommendations

9.1 Summary

The aim of this project was to provide daily commuters with reliable bus schedule and transit agency supervisors with a monitoring system. With advancement of technology, this has now become more achievable goal by making use of web apps and cloud technologies. Android being used by many users this solution will benefit most of the commuters. And supervisors will be easily monitoring all the vehicles easily helping them plan schedules.

9.2 Future Enhancements

1. Extending to include multiple transit agencies

Currently our project scope is to only use the data from VTA and design an application which can deliver a real-time ETA for the user who is going to use VTA services. But the application canbe extended to include the many other transit agencies which use the GTFS format for publishing their data**.**

2. Include trip planner feature in user application

The User application now only shows the Estimated Arrival Time for the public transportation to their respective stops. New features can be added to the User application which can proactively collect the travel history patters of the user. This can be used to suggest, notify the user with personalized suggestions. The application also can be extended as a travel assistant. Given the start, end and time. The application should be able to give a list of busses and all the possible routes one can take using the public transportation.

3. Provide analytics to transit agency

To use our application user should let us know his location. This leaves us in a position where we can collect the GPS locations of the users when they are planning to take a public transit bus. Which gives us insight into the patterns of passenger traffic and the timing of the peak and low passenger traffic. Using this data along with the passenger count which can be collected from the buses, we can increase or decrease the number of buses that start during a certain time. This kind of dynamic bus scheduling helps in reducing the crowded buses at peak times.

10 Glossary

Term	Definition
Android	An open source operating system from Google used for smartphones and tablets.
Concurrency	Server application should be able to handle multiple clients at any given point of time.
Cordova	Apache Cordova is a mobile application development frame work. Developers make use of web technologies lie HTMl, CSS, Javascript to build smartphone applications.
CSS	Cascading style sheet defines used defining styles to the HTML elements.
Event Driven	The flow of program is defined by events such as user actions or messages.
GPS	Location of users can be traced using latitude and longitude data provided by Global Positioning System (GPS).
ETA	Estimated time of arrival for the buses.
GTFS	General Transit Feed Specification (GTFS) defines a common format for public transportation schedules.
HTML	Hypertext Markup Language is a standardized format defined for developing browser pages.

Python	Programming language used for backend code and test scripting.
Hybrid Application	Application built using both native and web application elements.
Ionic Framework	It is a SDK to build smartphone hybrid applications.
JavaScript	Scripting language supported by most of the browsers.
AWS	Amazon Web Services, a comprehensive Cloud Computing Platform provided by Amazon.com

11 References

[1] J. Biagioni, T. Gerlich, T. Merrifield and J. Eriksson, "EasyTracker: Automatic Transit Tracking, Mapping, and Arrival Time Prediction Using Smartphones*", in 9th ACM Conference on Embedded Networked Sensor Systems,* 2011, pp.68-81.

[2] F. Sun, Y. Pan, J. White and A. Dubey, "Real-time and Predictive Analytics for Smart Public Transportation Decision Support System", *in IEEE International Conference Smart Computing* (SMARTCOMP), 2016, pp. 1-8.

[3] J. Gong, M. Liu, and S. Zhang, "Hybrid dynamic prediction model of bus arrival time based on weighted of historical and real-time GPS Data", *in 25th Chinese Control and Decision Conference* (CCDC), 2013, pp. 972-976.

[4] R.P.S. Padmanaban, L. Vanajakshi, and S.C. Subramanian, "Estimation of bus travel time incorporating dwell time for APTS applications", *IEEE* Conference *Intelligent Vehicles Symposium,* 2009, pp. 955-959.

[5] P. Zhou, Y. Zheng and M. Li, "How long to wait: predicting bus arrival time with mobile phone based participatory sensing" *10th international conference on Mobile systems, applications, and services,*2012, pp.379-392.

[6] C.-H. Wu, J.-M. Ho, and D.-T. Lee, "Travel-time prediction with support vector regression,"

Intelligent Transportation Systems, IEEE Transactions on, vol. 5, 2004, pp. 276–281.

[7] D. Ingle, N. Navi, Mumbai, "Experimental Estimates of Low-Cost Bus Tracking System Using Area-Trace Algorithm", *Fifth International Conference on Communication Systems and Network Technologies*, 2015, pp.525-529.

[8] C.I.J. Steven, Y. Ding and C.Wei, "Dynamic bus arrival time prediction with artificial, Natural Networks", *Journal of Transportation Engineering*, 2002, 128(5):429-438.