INDEX

1. Project Description

The world is growing very fast and so as the metropolitan cities which is leading to rise in infrastructure and as well as use of vehicles. Technology is becoming vital as it can automate many mundane jobs to be easy and efficient which could end in peaceful and happy life. The cities which are adopting technology as part of their growth in sectors like public transport, street, lights system and parking etc. are becoming Smart Cities.

As population is growing, traffic congestion increases and finding a parking space that too in rush ours becomes difficult for drivers. Car parking is becoming major problem with increasing vehicle size and compact parking spaces. For many people, searching parking spaces becomes a daily headache and even if they know, many vehicles may follow very limited parking spaces to cause serious traffic congestion.

To minimize the traffic congestion and parking problems, Smart parking is an obvious option to get over it. It helps to find parking spots in real time by optimizing the parking spots usage with the help of technology. To achieve this, a Smart Parking system controls and supervise a network of low-cost sensors obtains information about parking available spaces. When deployed as a system, smart parking helps to reduce car emissions in metropolitan cities which in turn avoids unnecessary search for parking.

The main purpose of the project, which involves combination of technologies, is to provide the user complete information about available free parking spots. A large number of sensors are installed near different parking spots and the data which is received by these sensors needs to be stored and processed in real time to gain effective and precise insights about available parking spots. This creates a requirement for a high computational and storage platform that can withhold large number of requests that are being sent. To achieve this high demand needs, Cloud technologies is the most apt step which can provide us virtually unlimited resources that can be used in giving unerring outputs.

With all the information provided by users, our system can optimize the parking system facilitating citizens to find a parking spot in the least amount of time. It can have a great impact on traffic congestion, reducing accidents and saving time for productive tasks.

2. Requirements

User Requirements:

The user of our application would like to park his car at the nearest parking spot available to him. User must allow location sharing and enter the range(in miles as the search radius).

Administrator Requirements:

The administrator would want to see the status of the various sensors and other admin users in the system. The administrator would want to update, edit and delete the sensors and users in the system.

System Requirements:

Mobile	Android 4.4 or above GPS Network connection
Back-end server	Ubuntu 14.10 (Glassfish 4.1) Java EE (EJB 3.0, JPA 2.1) MySQL Restful web services(JAX-RS)
Sensor simulator	HTML CSS JS

3. Mobile UI Design Principles – Storyboard, Wireframes

End-User application UI Wireframes:

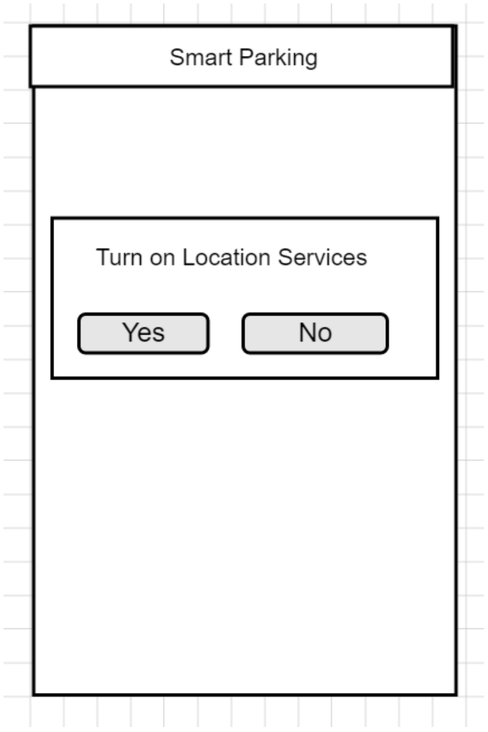

Figure 3.1: Location Permission

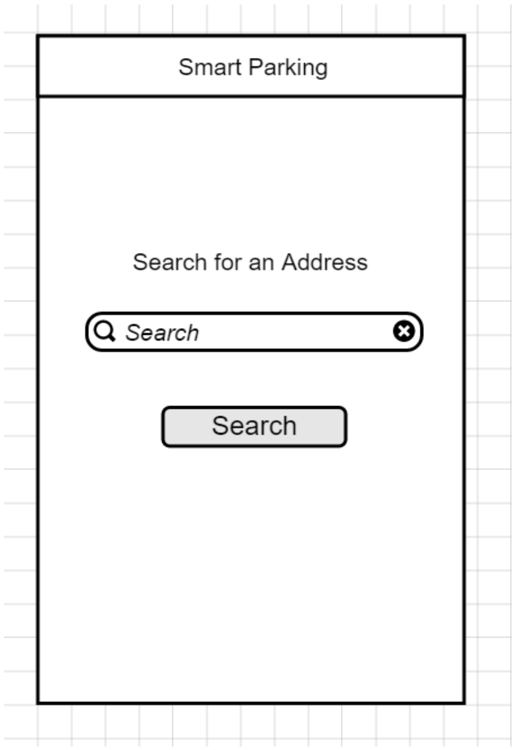

Figure 3.2: Search location

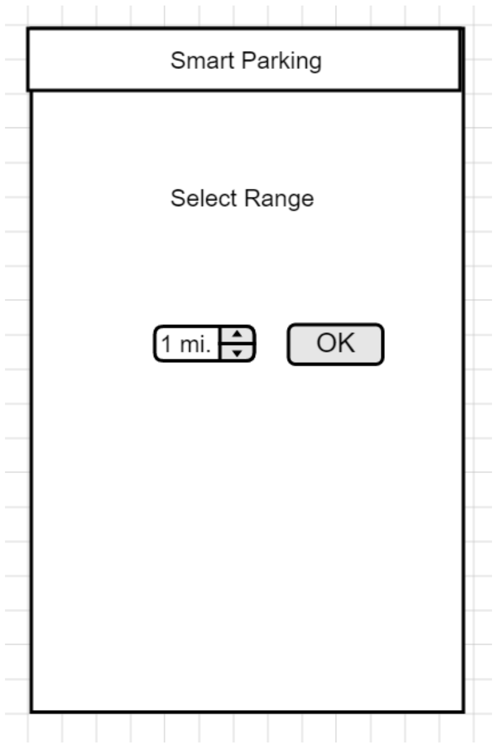

Figure 3.3: Select Range

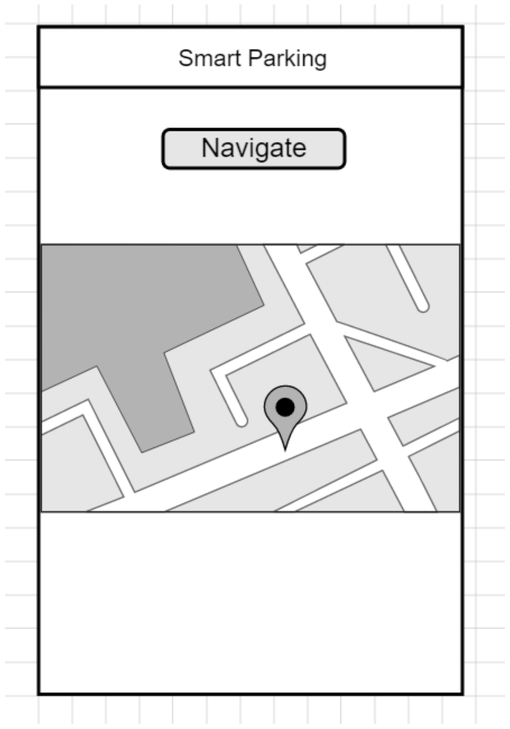

Figure 3.4: Search result on Map

Admin-User application UI Wireframes:

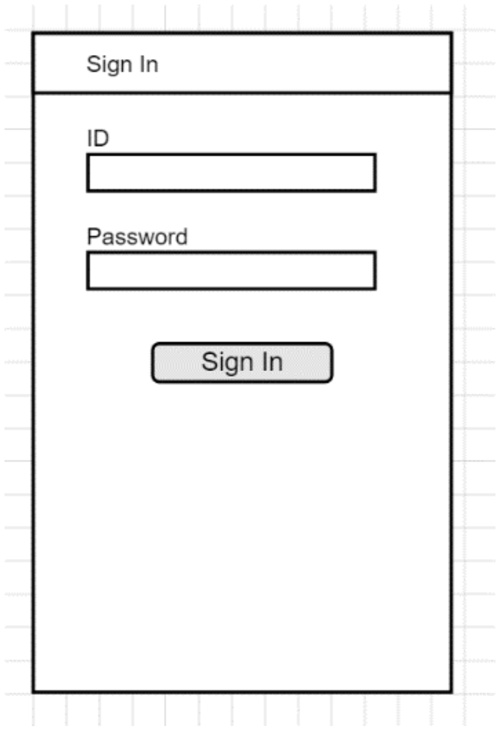

Figure 3.5: Admin user login

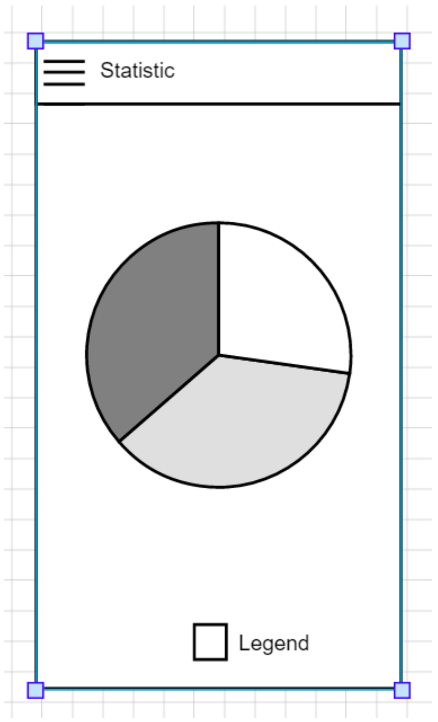

Figure 3.6: Parking spaces statistics

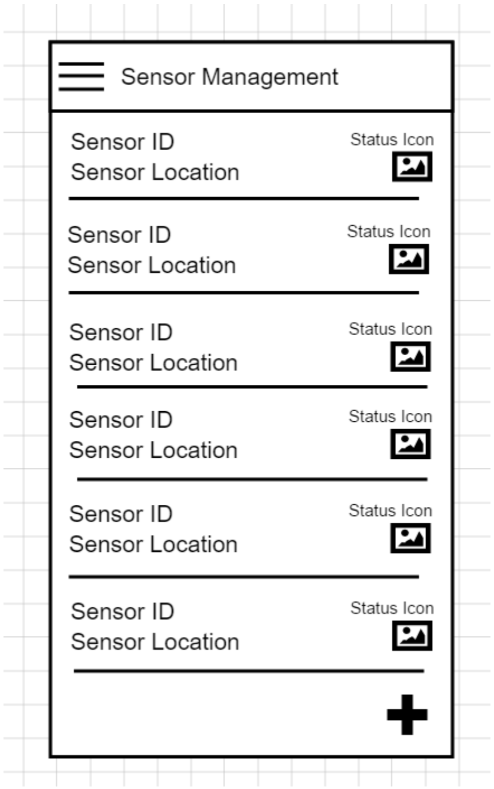

Figure 3.7: Sensor management

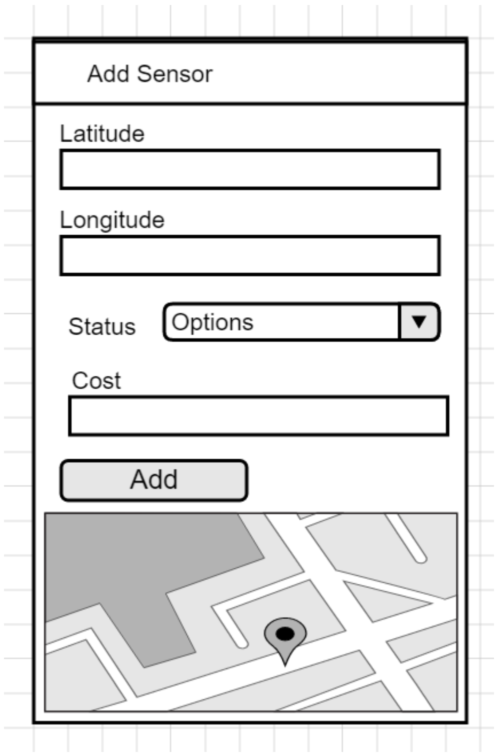

Figure 3.8: Add Sensor

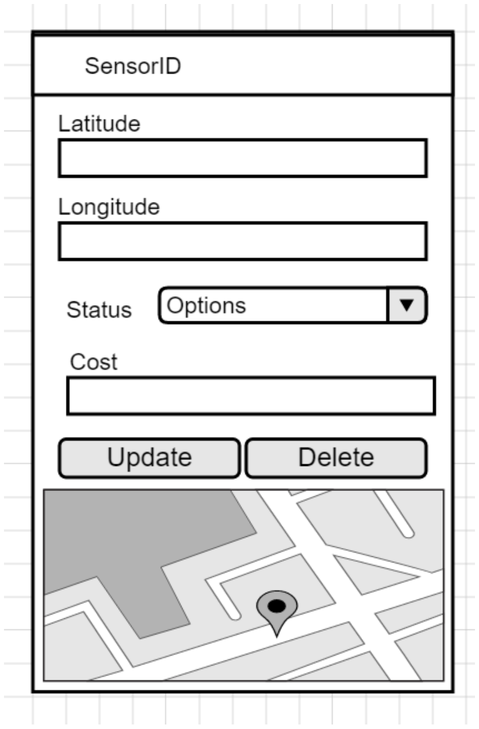

Figure 3.9: Update sensor

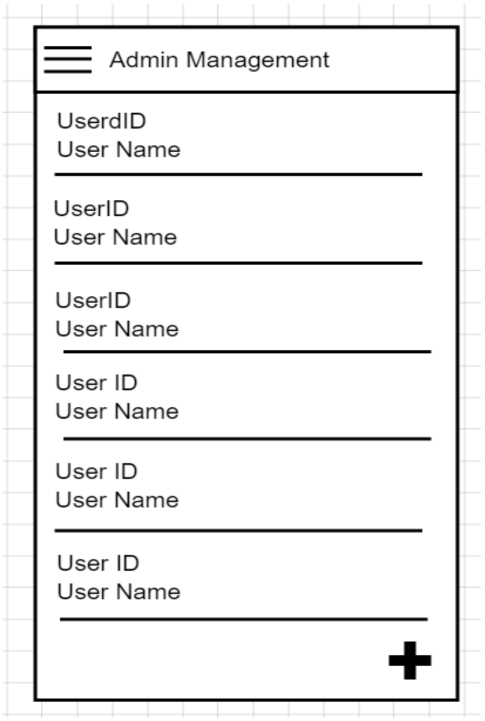

Figure 3.10: User management

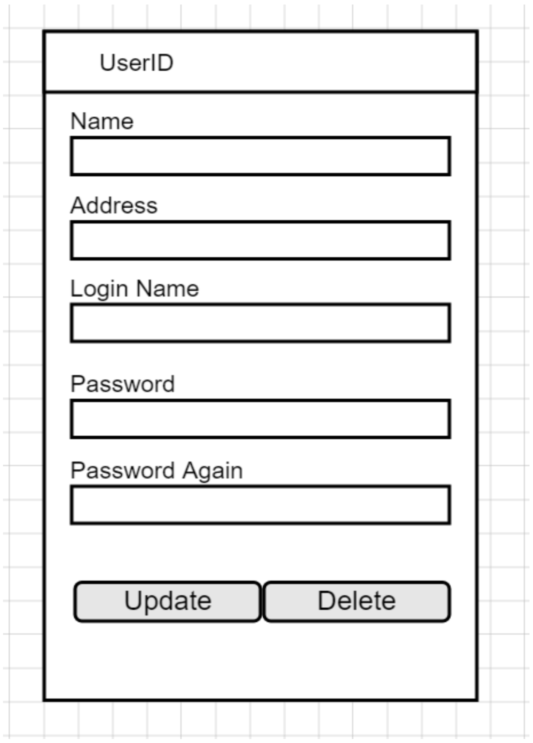

Figure 3.11: Update user

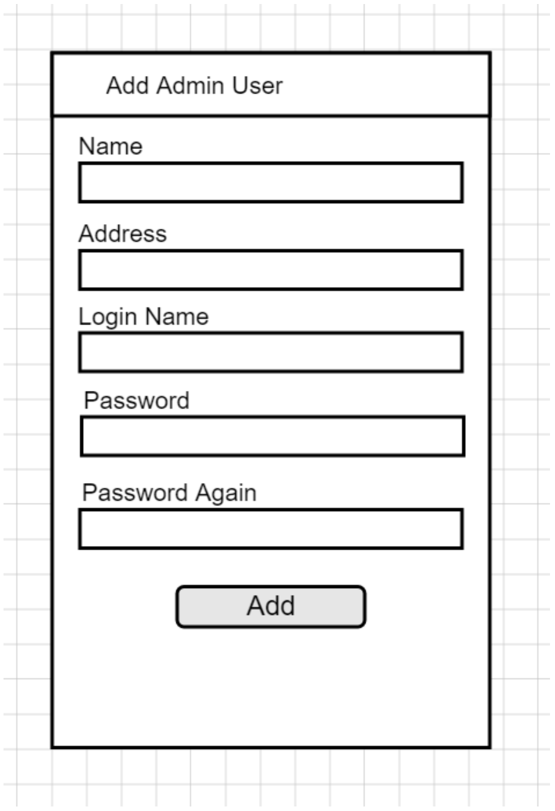

Figure 3.12: Add admin user

4. High Level Architecture Design

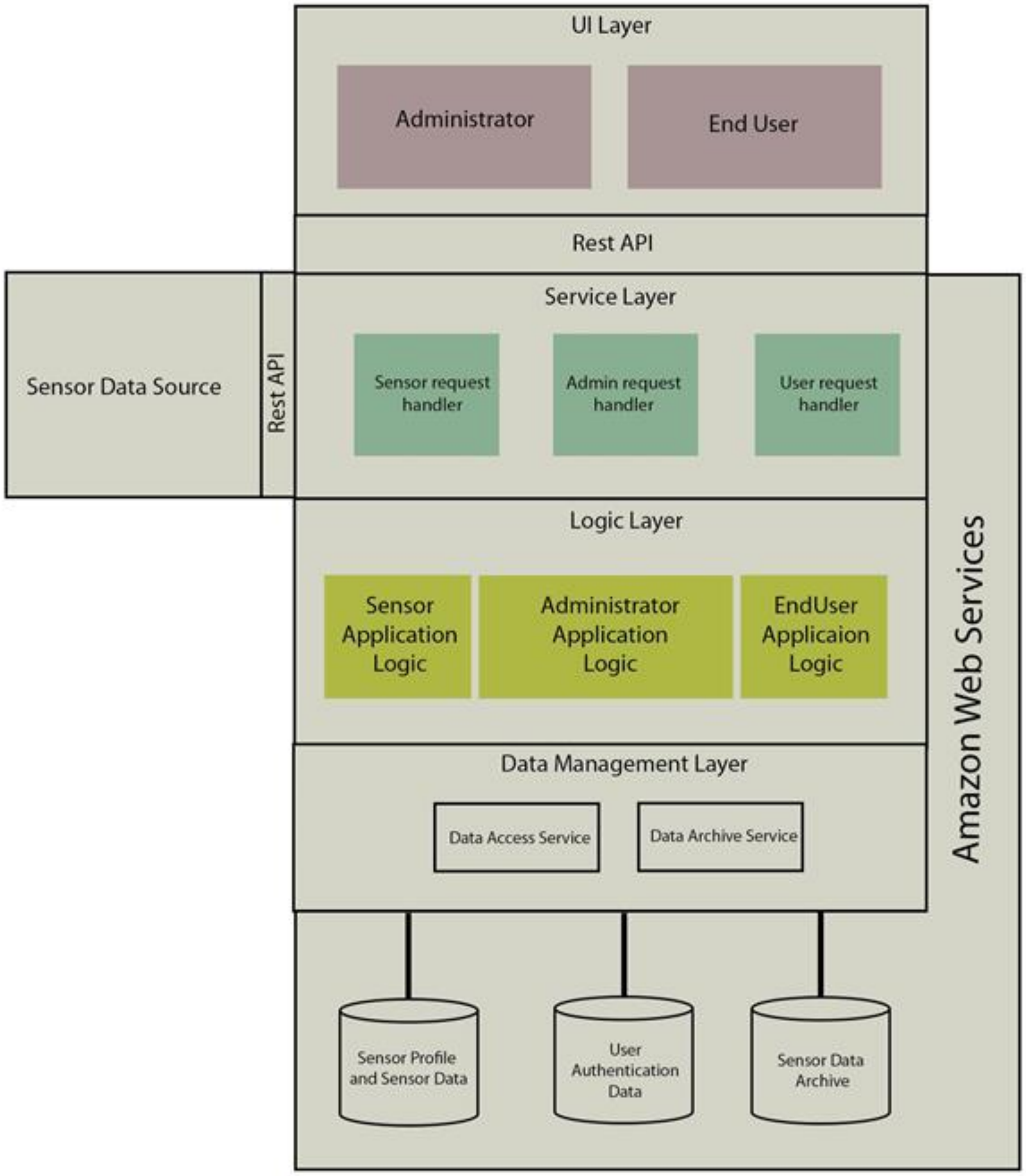

Figure 4.1: Architecture for Smart Parking System

Smart parking system’s architecture is composed of four layers.

a) User Interface Layer (UI Layer):

The UI layer encapsulates the interfaces for the various users of the system. In our case, the users are Government Administrators and the End Users. Both the users use the android applications specific to them to communicate with the system using the REST API.

b) Service Layer:

The service layer contains the various REST API request handlers which accept the RESTful requests from the UI layer and forward the request to the appropriate application logic.

c) Logic Layer:

This layer contains the actual application logic, where the processing occurs. The request received is processed at this layer by the applications responsible for that type of request.

d) Data Management Layer:

Logic layer uses the Data management layer to connect to the databases for retrieving and storing the data required and data received from users respectively during the processing of the data.

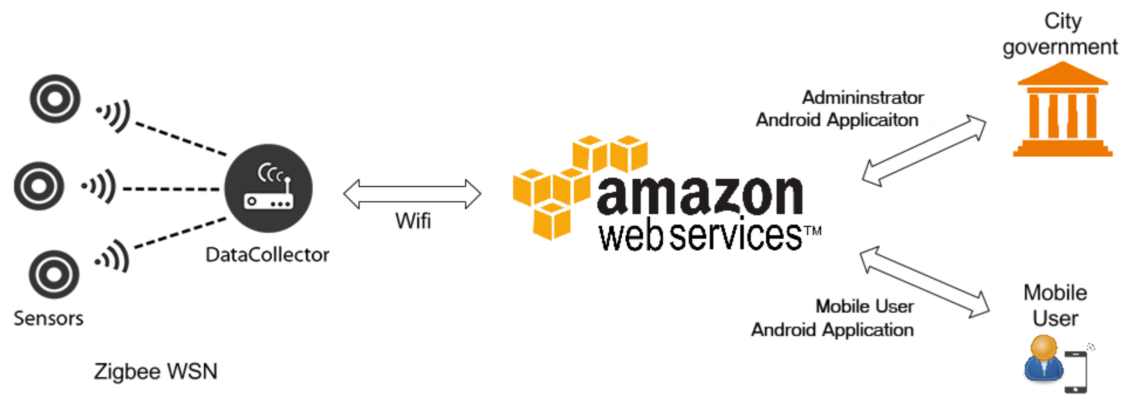

Figure 4.2: Infrastructure of the Smart Parking System

5. Component Level Design

Request Free Parking Spots

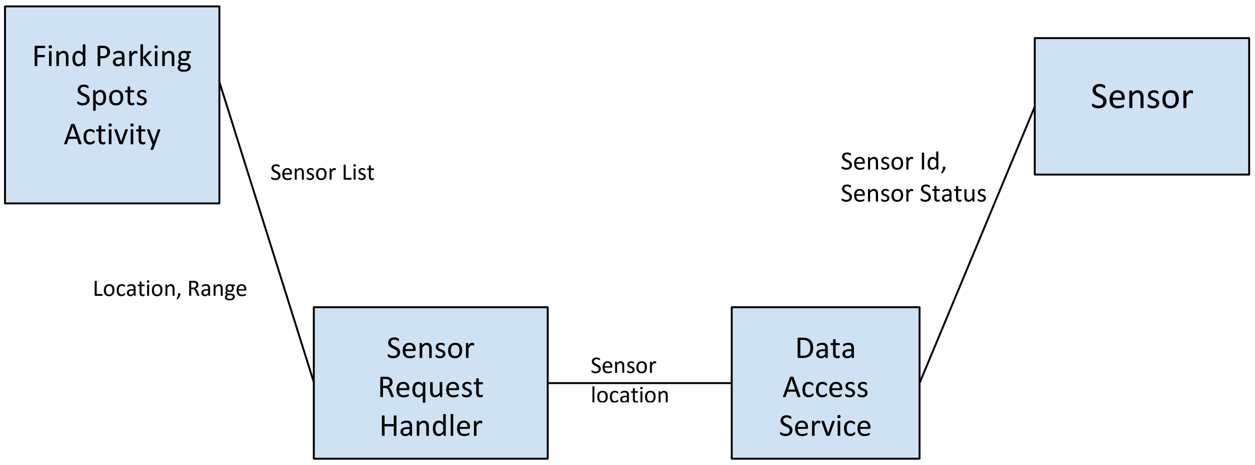

Figure 5.1: Component diagram for requesting free parking spots

Update Sensor Data

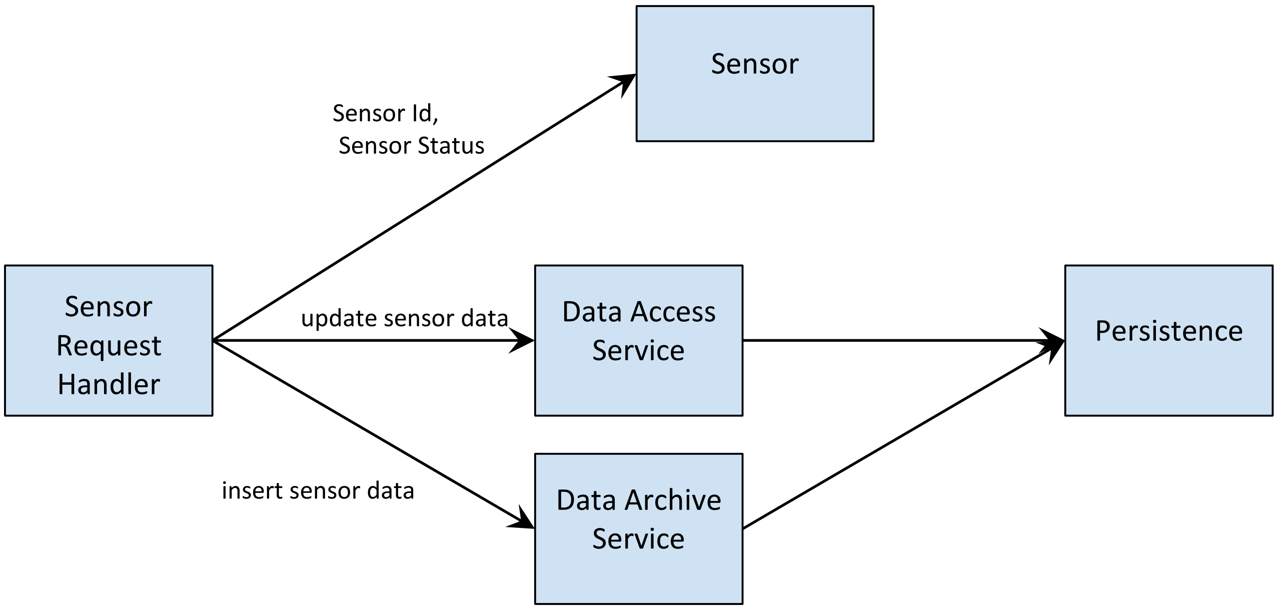

Figure 5.2: Component diagram for updating sensor data

Add New Sensor

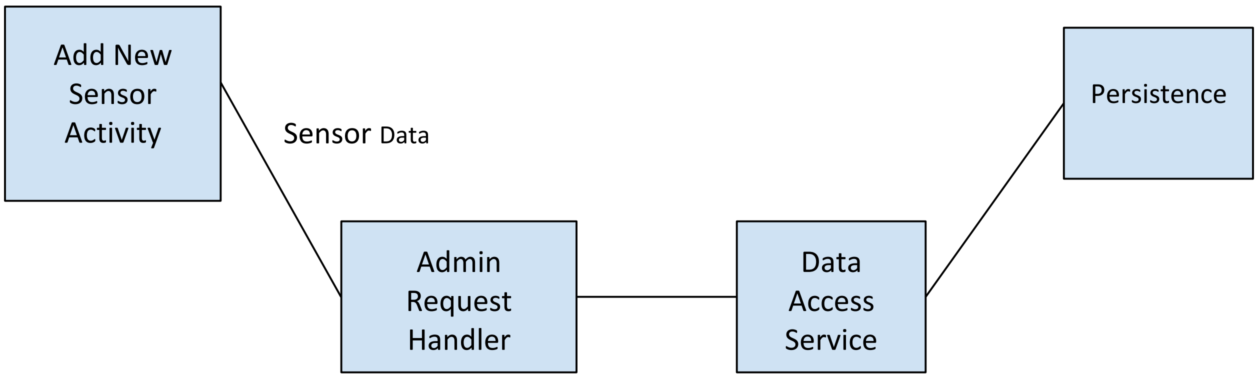

Figure 5.3: Add New Sensor Component diagram

Add New User

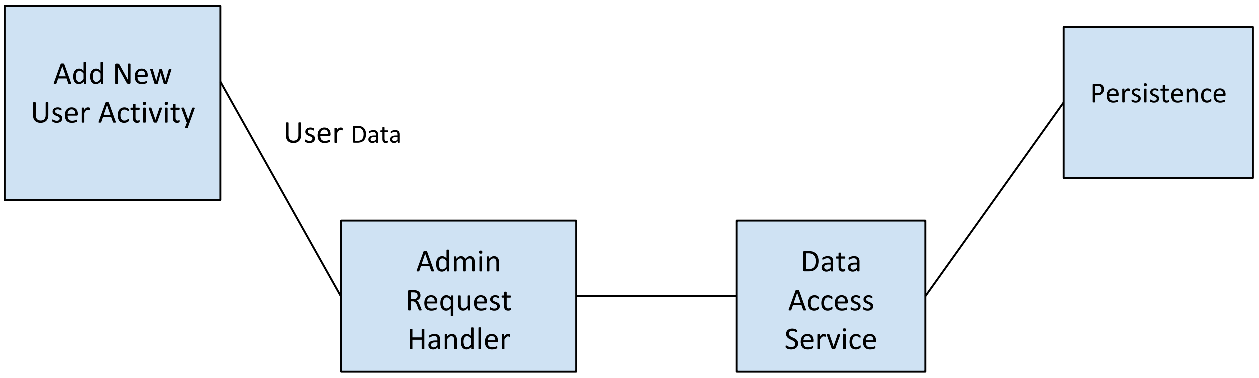

Figure 5.4: Add New User Component diagram

6. Sequence or Workflow

Sequence Diagrams for viewing free parking spots:

The User of our application sends his location and range values to the application logic using the android application. The application logic in the cloud backend finds the nearest parking spots and send the list of these parking spots to the user application. Now the user application accepts this list and plots the spots on the map using Google Maps API.

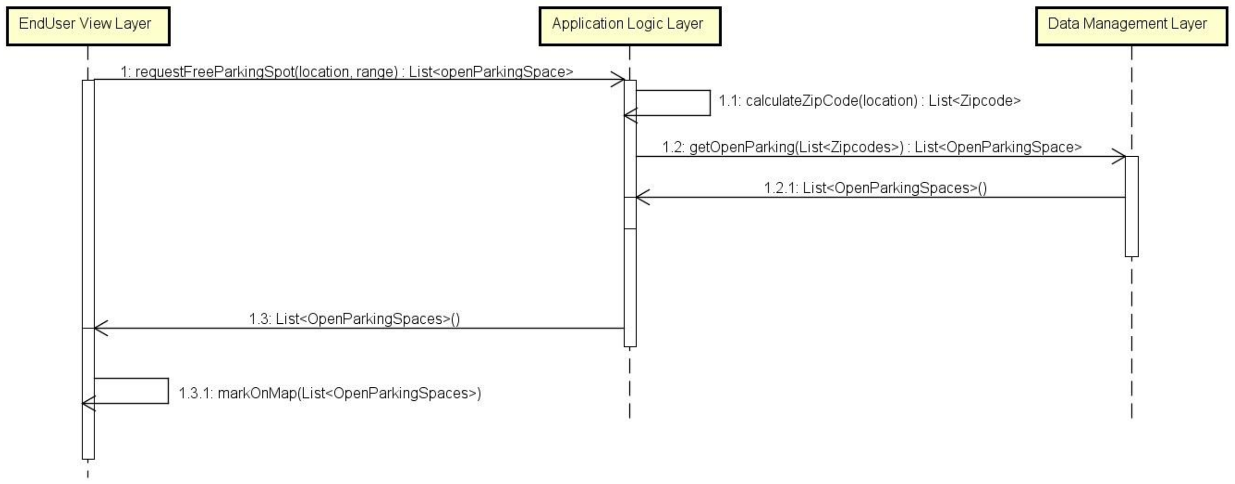
EndUser View Layer
Application Logic Layer
Data Management Layer
1: requestFreeParkingSpot(location, range) : List<openParkingSpace>
1.1: calculateZipCode(location) : List<Zipcode>
1.2: getOpenParking(List<Zipcodes>) : List<OpenParkingSpace>
1.2.1: List<OpenParkingSpaces>()
1.3: List<OpenParkingSpaces>()
1.3.1: markOnMap(List<OpenParkingSpaces>)

Figure 6.1: Request a free parking spot Sequence diagram

Sequence Diagrams for updating the parking sensor data.

Sensor sends the data whenever its detects the status change in the parking spot. The sensor data storage in the cloud backend accepts this update request and then stores the updated data in the backend.

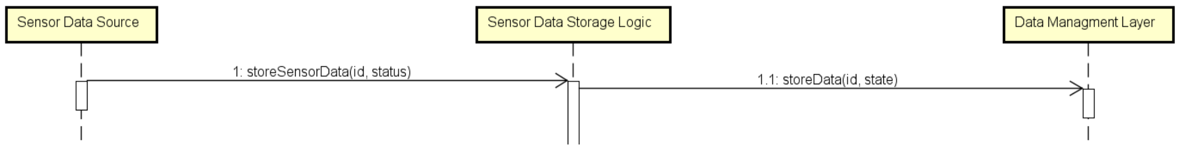
Sensor Data Source
Sensor Data Storage Logic
Data Managment Layer
1: storeSensorData(id, status)
1.1: storeData(id, state)

Figure 6.2: Sequence diagram for saving sensor data

Sequence Diagrams for Adding a new parking sensor:

The Administrator logs into the system and enters the details like ID, Location, Address and Cost for the new sensor. This data is sent to the cloud backend, which in turn saves in the Database. Adding another admin user is similar process.

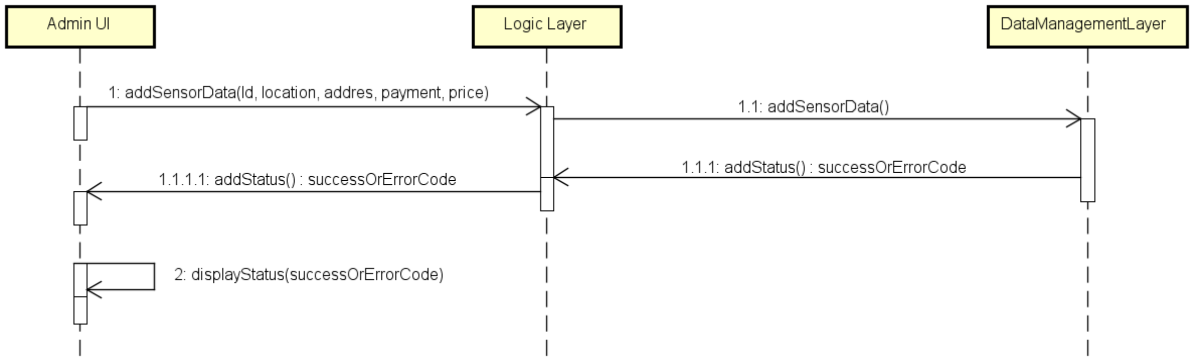
Admin UI
Logic Layer
DataManagementLayer
1: addSensorData(Id, location, addres, payment, price)
1.1: addSensorData()
1.1.1: addStatus() : successOrErrorCode
1.1.1.1: addStatus() : successOrErrorCode
2: displayStatus(successOrErrorCode)

Figure 6.3: Sequence diagram for Adding a new sensor

7. Mobile & Cloud Technologies Used & Descriptions

7.1 Mobile Technologies

We are using android applications as a main component of various user interactions in our application. There are two android applications one for User to find the nearest parking spots and another for Administrator to manage the parking sensors.

Location Services:

We use the location services to obtain the current user location to find nearest available parking spots.

Google Maps API:

This API is used for mapping the location co-ordinates of the free parking spots on the map and also used for navigating the user to the selected parking spot.

MP Android Charts API:

This API is used in the Administrator's application to view the statistical data.

Both the android applications use RESTful API to communicate with the backend running on the Amazon Web Services Cloud.

7.2 Cloud Technologies

We are using Amazon Web Services as main component of the infrastructure for our application.

EC2 Instances:

Most of the processing of our application is done in Elastic compute cloud, hence an efficient solution for the mobile applications. The Glassfish server runs on the EC2 instance which accepts the requests from the sensors and then stores the sensor data in RDS. When the user requests for a parking spot, the EC2 processes the data on RDS and sends the response back to the client device.

Amazon RDS:

Currently we are using the amazon RDS to store the sensor data and user data. Sensor Data has the location coordinates, unique Identifier, status and cost of that parking spot associated with that sensor. User data has the login details of the users who are the administrators. In the future, we plan to use the Big Data storage.

Auto Scaling and Load Balancing:

Using auto scaling we are able to launch new instances whenever there is lot of load on the system. We set the rule to launch a news ec2 instance with the saved AMI whenever the CPU usage goes beyond 80%. Load balancing will forward the new requests to the new instance instead of the previous instance thus increasing the performance of the system.

8. Interfaces – RESTful & Server Side Design

RESTful API for user app

API name	**Get all sensors by range**
URL	http://54.68.124.173:8080/SmartParking/api/user/getAllSensorsByRange
Return code	StatusOK : success StatusBadRequest : fail
Usage	Send user location and desired search range for all parking lot information
Input JSON format	{ “location”: “user latitude, user longitude” “range”: “desired range in miles” }
Input JSON example	{ “latitude”: “37.3351874, -121.8810715” “range”: “2” }
Output JSON format	{ "sensorList":[{ "idSensor":[integer ID], "location":[string "sensor latitude, sensor longitude"], "status":[integer [0:'Undefine',1:'Free', 2:'Occupied', 3:'Unavailable', 4:'Inactive']], "zipcode":[integer zipcode], "cost":[double cost], "sensorHistoryList":[reserved]}, "sensorDataWrapperList":[reserved], "userList": [reserved] }
Output JSON example	{ "sensorList":[{ "idSensor":1, "location":"37.3361255, -121.885707555", "status":1,"zipcode":1,"cost":5.0,"sensorHistoryList":[]}, {"idSensor":2, "location":"37.336658, -121.884452", "status":2,"zipcode":1,"cost":2.25,"sensorHistoryList":[]}], "sensorDataWrapperList":null, "userList":null }

RESTful API for administrator app

For sensors:

API name	**Get all sensor data**
URL	http://54.68.124.173:8080/SmartParking/api/sensor/getAllSensors
Return code	StatusOK : success StatusBadRequest : fail
Usage	Get all sensor information
Input JSON format	N/A
Input JSON example	N/A

Output JSON format	{
	"sensorList":[{
	"idSensor":[integer ID],
	"location":[string "sensor latitude, sensor longitude"],
	"status":[integer [0:'Undefine',1:'Free', 2:'Occupied', 3:'Unavailable', 4:'Inactive']],
	"zipcode":[integer zipcode],
	"cost":[double cost],
	"sensorHistoryList":[reserved]},
	"sensorDataWrapperList":[reserved],
	"user List": [reserved]
	}

Output JSON example	{
	"sensorList":[{
	"idSensor":1,
	"location":"37.3361255, -121.885707555",
	"status":1,"zipcode":1,"cost":5.0,"sensorHistoryList":[]},
	{"idSensor":2,
	"location":"37.336658, -121.884452",
	"status":2,"zipcode":1,"cost":2.25,"sensorHistoryList":[]}],
	"sensorDataWrapperList":null,
	"userList":null
	}

API name	**Add sensor data**
URL	http://54.68.124.173:8080/SmartParking/api/admin/addNewSensor
Return code	StatusOK : success StatusBadRequest : fail
Usage	Add new sensor data
Input JSON format	N/A
Input JSON example	N/A

Output JSON format	{
	"sensorList":[{
	"idSensor":[integer ID],
	"location":[string "sensor latitude, sensor longitude"],
	"status":[integer [0:'Undefine',1:'Free', 2:'Occupied', 3:'Unavailable', 4:'Inactive']],
	"zipcode":[integer zipcode],
	"cost":[double cost],
	"sensorHistoryList":[reserved]},
	"sensorDataWrapperList":[reserved],
	"userList": [reserved]
	}

Output JSON example	{
	"sensorList":[{
	"idSensor":1,
	"location":"37.3361255, -121.885707555",
	"status":1,"zipcode":1,"cost":5.0,"sensorHistoryList":[]},
	{"idSensor":2,
	"location":"37.336658, -121.884452",
	"status":2,"zipcode":1,"cost":2.25,"sensorHistoryList":[]}],
	"sensorDataWrapperList":null,
	"userList":null
	}

API name	**Update sensor data**
URL	http://54.68.124.173:8080/SmartParking/api/admin/updateSensor
Return code	StatusOK : success StatusBadRequest : fail
Usage	Update existing sensor data

Input JSON format	{
	"idSensor":[integer ID],
	"location":[string "sensor latitude, sensor longitude"],
	"status":[integer [0:'Undefine',1:'Free', 2:'Occupied', 3:'Unavailable', 4:'Inactive']],
	"zipcode":[integer zipcode],
	"cost":[double cost]
	}

Input JSON example	{
	"idSensor":1,
	"location":"32.1242, -128.22131",
	"status": 1,
	"zipcode": 95234,
	"cost": 20.1
	}

Output JSON format	N/A
Output JSON example	N/A

API name	Delete sensor data
URL	http://54.68.124.173:8080/SmartParking/api/admin/deleteSensor
Return code	StatusOK : success StatusBadRequest : fail
Usage	Delete existing sensor data

Input JSON format	{
	"idSensor":[integer ID]
	}

Input JSON example	{
	"idSensor":1
	}

Output JSON format	N/A
Output JSON example	N/A

For user management:

API name	Get all user data
URL	http://54.68.124.173:8080/SmartParking/api/admin/getAllUsers
Return code	StatusOK : success StatusBadRequest : fail
Usage	Get all user information
Input JSON format	N/A
Input JSON example	N/A

Output JSON format	{
	userList:[
	{“idUser”:[string id]
	“name”:[string name]
	“address”:[string address]
	“loginName”:[string loginname]}]
	}

Output JSON example	{ userList:[{“idUser”:”1” “name”:SJSU “address”:”1 Washington Square”

	“loginName”:”SJSU_admin”}] }

API name	**Add user data**
URL	http://54.68.124.173:8080/SmartParking/api/admin/addNewUser
Return code	StatusOK : success StatusBadRequest : fail
Usage	Add new user data

Input JSON format	{
	userList:[
	{“idUser”:[string id]
	“name”:[string name]
	“address”:[string address]
	“loginName”:[string loginname]
	“password”:[string password]}]
	}

Input JSON example	{
	userList:[
	{“idUser”:”1”
	“name”:SJSU
	“address”:”1 Washington Square”
	“loginName”:”SJSU_admin”
	“password”:”1234567”}]
	}

Output JSON format	N/A
Output JSON example	N/A

API name	**Update user data**
URL	http://54.68.124.173:8080/SmartParking/api/admin/updateUser
Return code	StatusOK : success StatusBadRequest : fail
Usage	Update existing user data

Input JSON format	{
	userList:[
	{“idUser”:[string id]
	“name”:[string name]
	“address”:[string address]
	“loginName”:[string loginname]
	“password”:[string password]}]
	}

Input JSON example	{
	userList:[
	{“idUser”:”1”
	“name”:SJSU
	“address”:”1 Washington Square”
	“loginName”:”SJSU_admin”
	“password”:”1234567”}]
	}

Output JSON format	N/A
Output JSON example	N/A

API name	**Delete user data**
URL	http://54.68.124.173:8080/SmartParking/api/admin/deleteUser
Return code	StatusOK : success StatusBadRequest : fail
Usage	Delete existing user data
Input JSON format	{ “idUser”:[string id] }
Input JSON example	{ “idUser”:”SJSU” }
Output JSON format	N/A
Output JSON example	N/A

RESTful API for sensor simulator

API name	**Update sensor status**
URL	http://54.68.124.173:8080/SmartParking/api/sensor/updateSensorStatus
Return code	StatusOK : success StatusBadRequest : fail
Usage	Update existing user data
Input JSON format	{ "idSensor":[integer ID], "status":[integer [0:'Undefine',1:'Free', 2:'Occupied',3:'Unavailable', 4:'Inactive']] }
Input JSON example	{ "idSensor":123, "status":1 }
Output JSON format	N/A
Output JSON example	N/A

9. Client Side Design

In Smart Parking system, we have two types of applications in which one would be used by admin to have control over the sensors, end-users etc. and other application for the end-users who would be utilizing the Smart Parking System to get information about available free spots for parking.

User App flow

Once user downloads and installs the Smart Parking system user application, either he/she can use his/her current location or enter the zip code/address with the desired range for the system to get the available free parking spaces using RESTful calls to the backend. JSON Response returned from cloud backend is parsed and is used to display the parking spots. The parking spots also have the price for it on the Google maps within the search range provided by the end-user.

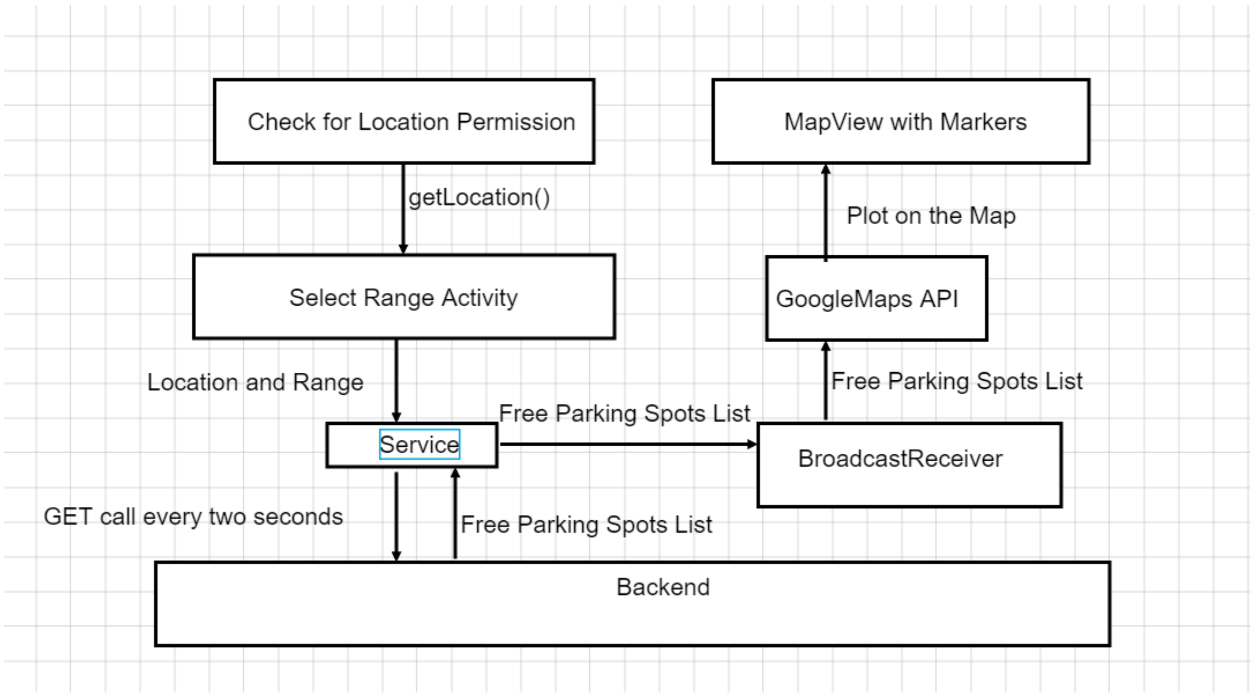

Figure 9.1: User Application flow

Admin App flow

Once the admin users download and installs the Smart Parking system admin application, admin user has to login with the Admin ID and Password to get authenticated.

Administrator can see statistics through donut chart about how many parking spaces have been occupied, how many are free, how many are unavailable and how many are inactive. Admin can also have control over the sensors that are used in Smart Parking system where admin can add, update and delete the sensors. Admin can also add, update and delete other admins in the system through admin management. All the calls to the backend is through RESTful API.

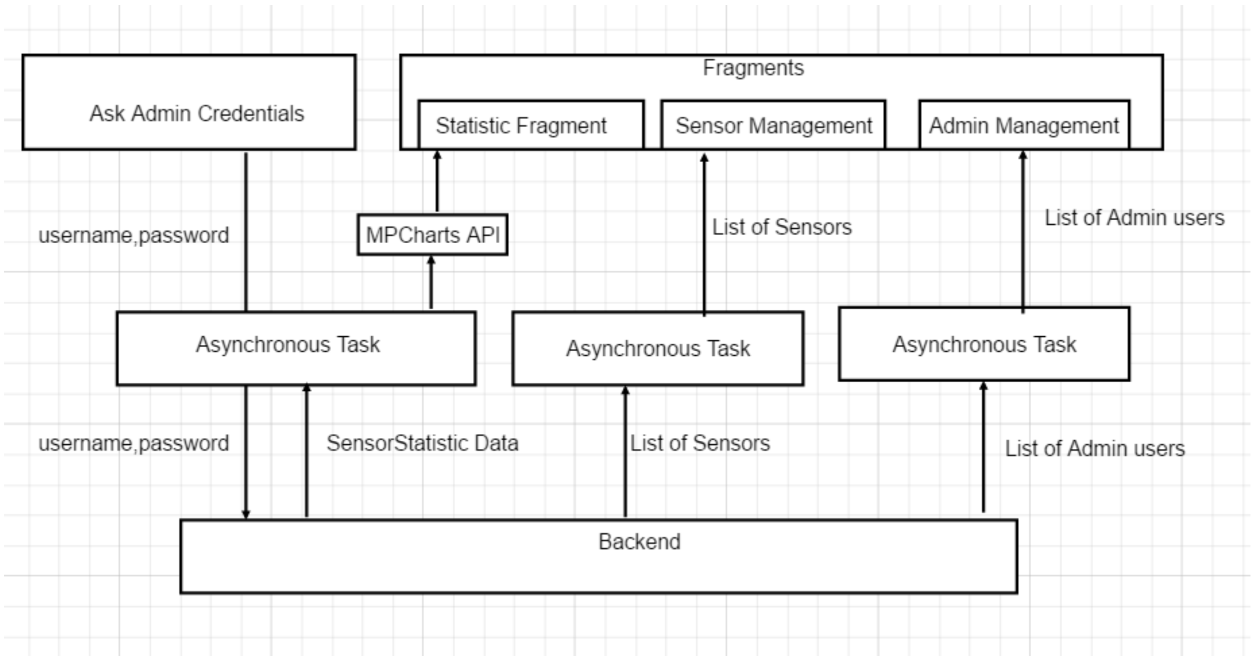

Figure 9.2: Administrator Application flow

12. Design Patterns Used

Design Patterns used in our Mobile Applications (Front End):

Builder Pattern:

Builder pattern is used in our application to build the alert dialogs. Whenever the user adds or modifies the sensors or users on the admin application, the alert dialog is used to confirm the user action. Alert dialog has the same construction process but create different representations.

Observer Patterns:

This pattern is used in our application for the Broadcast Receiver. The Service which checks for the updates in the parking spots, sends the broadcasts to the user view activity to update the google maps.

Design Patterns used in cloud (Back End):

Transfer Object Pattern

We have used the Transfer object pattern to send and receive data across the mobile application and the cloud. The JSON Data is mapped into the instance of a DTO class. The transfer object has methods to set and get the various properties of the object. The object of the class implementing this pattern is serializable and hence makes it easy to transfer over the network.

Data Access Object Pattern (DAO Pattern)

We use this pattern to abstract the database data access logic from the higher levels of the architecture. The object of the DAO class will have the methods that execute SQL queries on the database. It also maps the database records into the objects of the entities.

Facade Pattern

We used this pattern to hide the complexities of the underlying logic to the higher levels of the architecture. We provide an interface which can be used by the applications to access the logic abstracted by this pattern.