# ParkWizard: Street Parking Android App

Siddharth Shah, Kunal Baweja, Anand Naik, Dhruv Shekhawat

## TABLE OF CONTENTS

**1 Introduction** **2**

1.1 System Introduction 2

1.2 Motivation 2

1.3 Objectives 3

**2 System Design and Implementation** **3**

2.1 Technology Stack 3

2.2 Architecture 5

2.3 Algorithm 7

**3 Salient Features** **11**

**4 Challenges** **12**

**5 Conclusion** **12**

# 1. Introduction

## 1.1 System Introduction

ParkWizard is an Android application that helps users find available street parking locations. It works on a crowd-sourced, point-based system where users earn points by reporting street parking locations and are able to use those points later in order to obtain available parking spots. This point-based system works well towards maintaining a smooth information flow between users(finders) who want to find available parking spots and those(informers) who have information about them. At a point, any user can act as a finder or an informer. Users are authenticated using Amazon Cognito which assigns temporary security credentials for accessing our backend AWS resources. The app displays all the available parking spots which are within 500 meters of radius from the location where the user wants to find parking. Users are initially granted 100 free points on creating an account with our app. Each time, users lose 5 points on searching for a parking and gain 10 points on reporting available parking spots.

## 1.2 Motivation

Finding open street parking spots in metropolitan cities such as New York has always been a distressing task. Delay in finding street parking always leads to increased traffic due to the congestion caused by slow moving vehicles looking for parking. In the worst case, we try to park our vehicles at places which are not official parking spots or are too crammed up to accommodate our vehicles. If we could find a way to search parking through the click of a button it would not only decongest traffic but would also save fuel and precious time. Moreover, it would help release the enormous stress associated with finding parking each time. Various apps have been developed in the past but with an intention of helping users book garage parking instead of street parking. Very few attempts have been made to guide users to the nearest available street parking locations. To mitigate this issue, we have created ParkWizard which employs cloud services to update users with real-time street parking information.

### 1.3 Objectives

The application tries to fulfill several objectives through the course of the app development process. Few of these are as follows -

- Develop a score-based, crowd sourced approach that helps users find open parking spots at the cost of some points allotted to them.
- Allow users to log into our app using their Facebook accounts.
- Guide users to the nearest available parking locations which are within 500 meter radius of the user's search location.
- Get users to report available parking spots by giving them an incentive to earn points which they can later use to find available parking spots.
- Ensure data reliability by penalizing users whom we suspect of feeding wrong data about parking spots.
- Utilize AWS SQS to efficiently manage parking information on the server side.
- Grant users access to our backend AWS resources through AWS Cognito.
- Deploy the app on Elastic Beanstalk.

# 2. System Design and Implementation

## 2.1 Technology Stack

We employ Google applications, Amazon web services and Facebook OAuth 2.0 to make our app more scalable, reliable and secure. The description of these services are as follows -

- AWS Cognito

  Amazon Cognito enables us to authenticate users through social identity providers such as Google, Facebook or Amazon and create temporary security credentials for them to access our backend AWS cloud services.

- AWS Elastic Beanstalk

  AWS Elastic Beanstalk is a service for deploying and scaling web applications and services developed with languages such as Java, and Python. This works well for us as we have built an Android app with a backend written in Python. We simply upload our code and Elastic Beanstalk automatically handles the deployment, from capacity provisioning, load balancing, auto-scaling to application health monitoring.

- AWS Elasticsearch
  Amazon Elasticsearch makes it easy to deploy, manage, and scale search features for log analytics, full text search, application monitoring. Elasticsearch's easy-to-use APIs and real-time availability, scalability, and security benefits us when we store and fetch parking information from the backend in real-time.

- AWS SQS
  Amazon Simple Queue Service is a fast, reliable and scalable message queuing service. It is simple and cost-effective and helps in decoupling the components of a cloud application. We can use Amazon SQS to transmit high volumes of data, without losing data. In a data-critical application like ours, which relies only on the data being reported by users, SQS functions well by guaranteeing data integrity and availability. Moreover it becomes easier for our backend to dequeue the data one at a time and handle it consistently i.e SQS ensures efficient handling of load on the server side.

- Google Cloud Messaging
  Google Cloud Messaging (GCM) service lets our Android application notify a user of an event, even when the user is not using our app. We need to register our application and the user's device with GCM to obtain a token which we use to notify the user's device on events such as reporting and updating parking locations. Google-provided GCM Connection Servers gets notifications from our Django server and sends them to the users' GCM-enabled Android app running on their device.

- Google Maps API
  We have used the Android Google Maps API to implement the location finding functionalities. The Maps API provides the location i.e. latitude and longitude of a location while user searches and reports the parking locations. The user's own

location is displayed on the map. An API key is obtained to access the Google Maps which enables us to monitor our application's Maps API usage, and ensures that Google can contact us about your application if necessary.

- Facebook OAuth 2.0

  Facebook OAuth allows us to authenticate users on our app and let them login through their Facebook accounts. This allows us to delegate the login functionality to Facebook and decouple it from our application.

## 2.2 Architecture

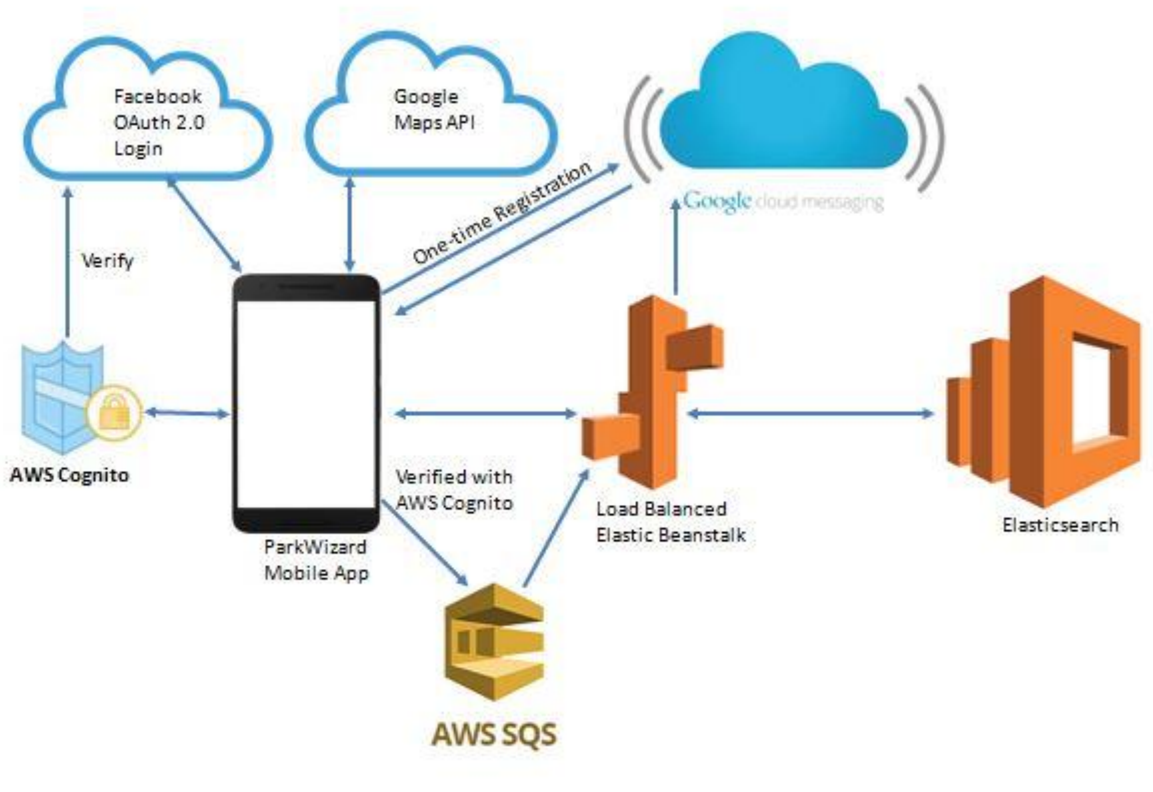

Fig 1. App Architecture

To facilitate user log in through Facebook account, our app communicates with Facebook OAuth 2.0 with the user information. At the time of login we also utilize AWS Cognito to provide users with temporary security credentials to access our app's backend resources. We have built our backend with Django web-server that communicates with AWS Elasticsearch to store and retrieve real-time parking data. When users report parking, the information is first sent to AWS SQS standard queue from where it moves forward to the Django server. This ensures reliable data delivery and efficient handling of load on the server side. As a map is required to find parking, our app uses the Google Maps API using an API key. On receiving the reported parking information, Django server streams it to Elasticsearch which stores the data in the (latitude, longitude, #available spots, #total spots) format. The user information is also stored on ES in the format (userid, points). When searching for parking, Django server retrieves the data from ES based on (latitude, longitude) provided by user and sends it back to the app. We deploy our Django server on a load balanced AWS Elastic Beanstalk.

We have used Amazon SQS when a user reports a new parking spot or updates number of available parking spots. Each such request is queued in SQS and processed by our web server in a FIFO manner. It unburdened us from writing code that would have required us to handle multiple concurrent requests in an efficient manner, instead the same is taken care of by SQS and our application receives messages(requests) from SQS in a FIFO manner.

To make a new message entry in SQS, one must possess the credentials to access it. Hence, while reporting parking spots or updating number of spots the users of the app are required to have credentials that allows them to access SQS. Means to this end is Amazon Cognito, which generates temporary credentials for users to allow them to access SQS.

For the purpose of storing parking spots and their details, we have used no-sql data storage provided by Amazon Elasticsearch services. You can create multiple indices in Elasticsearch, which allowed us to create indices for Parking Locations and Users in our domain. Also Elasticsearch provides a very rich JSON-style domain-specific language 'Query DSL', that we used to execute queries on our domain.

And last but not the least, we deployed our web server on Elastic Beanstalk which provides support for multiple platforms and inherent load balancing.

We have used Google Cloud Messaging to send notifications to users regarding point updation. We chose to use GCM over SNS to send push notifications, as even after choosing SNS we would have had to send a notification to GCM only; which in turn will notify the client app.

Biggest advantage of using all these cloud components is that it provided us with scalability and alleviated us from performing administrative tasks.

## 2.3 Algorithm

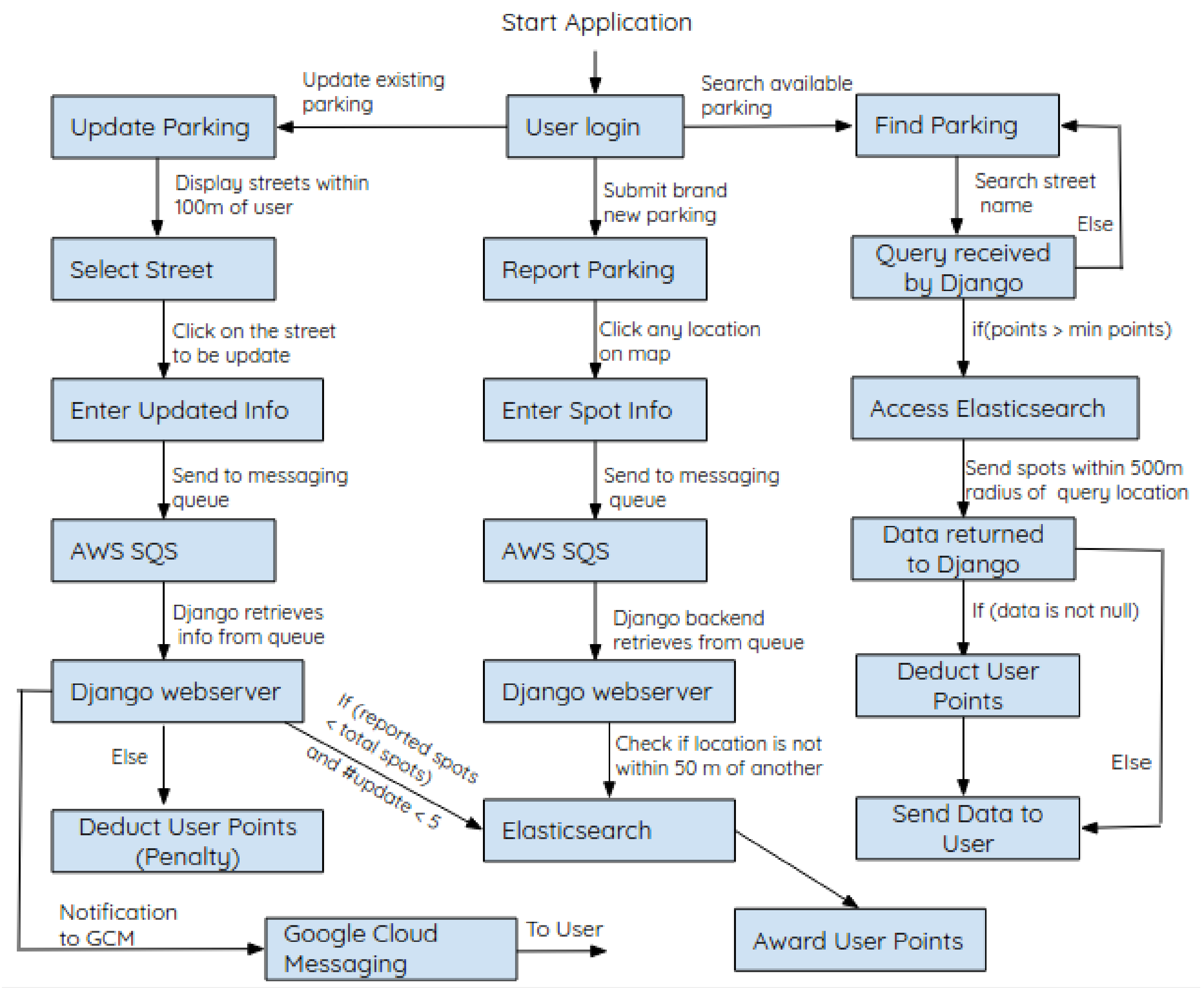

Fig 2. App State Transitions

Users start out with logging into our app using their Facebook account. During the signup we utilize AWS Cognito to assign temporary security credentials to users by authenticating them with their Facebook accounts. These temporary credentials now allow a user to access our AWS backend resources. At this time the application also interacts with Google Cloud Messaging Service which we discuss later. After signing in for the first time a user is allotted hundred free points which he can use for finding available parking locations. The (user's id, points) tuple is stored in Elasticsearch when the user logs in for the first time. After every subsequent log in, we just query elasticsearch to check if the user has an already existing account with our app. Every time users search for parking and a non-null result is returned, points are deducted by 5. Users require a minimum of 5 points  to be able to look for parking. If they run short of these points, they need to start reporting or updating open parking spot information. We have separated the logic for 'reporting' and 'updating' a parking location.

## Reporting

A parking is 'reported' for the first time, provided it is not within 50 meters of an already existing parking location. This is done to ensure that we do not have duplicate values for the same street parking spot in the Elasticsearch. The idea is that with time, we would be able to gather more and more data about the streets in the world as they would be reported for the very first time by users. When reporting the first time, we also store the total spots that can be available for parking on a street. This information would be useful for us later to catch fraudulent users who supply the number of available parking locations greater than total spots. If a user is successfully able to report a parking i.e one that is not within 50 meter distance from another, then he is awarded ten points.

## Updating

A parking is updated if it is an already existing parking in Elasticsearch which means that that parking has been reported by users before. When a user wants to earn points by updating a parking spot he can click the update parking button after which a map is displayed with the user location and all the existing parking spots which are within 100 meter radius of that user. The user can only update these spots since all the other spots outside the 100 meter radius are not displayed. The idea is to ensure that a user reports what he sees. We nullify the ability for letting users update parking spots which they aren't close to, assuming that they cannot know about spots which they can't witness in real-time. This feature is implemented to ensure data credibility and accuracy. On a successful parking update the user earns ten points just as he would when reporting a new parking.

## SQS Queue

When users report or update a parking location they basically need to tap on the map where they want to report and provide details which specifies the street name, number of available spots and the total number of permissible spots. This data is enqueued into AWS SQS with a type specified for every data entry. The type can be either 'report' or 'update' based on what operation the user performed. The data is queued into SQS in a FIFO manner which would ensure that no data is lost and that it is processed in the correct order of operations on the backend.

## Django Web Server

The Django web server(backend) extracts the parking data from the queue in a FIFO order and checks the type of the data entry. Based on the type of the entry the required action is performed i.e. updateAction or reportedAction. If the operation is 'report parking' then the backend extracts the (latitude, longitude) of the reported location and checks elasticsearch for a parking within 50 meter radius of the reported parking. If such a parking does not exist then we add the reported parking to Elasticsearch or else we reject it as we consider this to be a duplicate entry. Also, if the parking is successfully added then we award the user with ten points i.e. increment his points by ten and store it back into elasticsearch.

If the operation is an update operation then the backend does not perform any checks on location as we only display the existing parking locations which are within 100 meter radius of the user's location. The user could only have choose one of those parking options to update. Although there are no location checks, but we do need to perform checks on the available spots claimed by the user. We want to guarantee that the data is not false and isn't a fraud attempt by the user to earn points. We check if the value of the available spots is not greater than the total possible number of parking spots. If it is then we deduct points of the user as a penalty for submitting incorrect data. We get this value from the time the parking location was first 'reported' and use it as a standard every time a user updates parking location. Our assumption is that the initial reported total possible spots is correct.

We can work towards assigning our application to beta users who we can trust to feed in the correct data. Another check that we perform on the backend is that we only allow a user to update parking locations a maximum of five times a day. This ensures users cannot write bots to continuously earn points by updating parking. Also a user would not be able to make gain unrestrained points by updating the same location with the same spots repeatedly. He would be disallowed from updating more than 5 times a day. After we have performed our tests, we can store the data to Elasticsearch and award the user with ten points.

## Google Cloud Messaging

After a user logs in, the application communicates with Google Cloud Messaging to obtain a token for registering the user's device. This unique token is sent to the Django server along with the parking data when the user reports or updates a parking spot. After the Django server performs the operations, a notification about the status is sent to Google Cloud messaging along with the user's registered device token which then redirects the notification to the user's device. These notifications include information about points being added or deducted (fraudulent) based on whether the operation was successful or not.

## Searching

For searching, a user is provided with a search bar through which he can find streets where he desires to park. After entering the desired location and clicking search, (latitude, longitude) of the searched location is sent to Django server which checks whether the user has a minimum number of points needed to search a parking. This check is performed by querying elasticsearch with the userid. If the check is successful then the web server extracts all parking spots that are within 500 meter radius of the searched location and sends it to the app. Finally a map is displayed with markers representing the parking spots along with the total number of available spots. If non-zero number of parking spots are returned then we deduct 5 points from the user, else we don't as our app wasn't able to find any open parking locations for the user.

## 3. Salient Features

### Data Reliability

As our application is heavily dependent on the data supplied by users, we need to ensure that the information we store in Elasticsearch is reliable. To accomplish this we confirm that the user only updates parking information for streets which are within 100 meter distance from the user's location. Also, we deduct points for users who proclaim a higher number of available spots than the total number of parking spots itself. Therefore, it's crucial that we maintain the total possible parking spots of streets in our Elasticsearch. Additionally, users are only allowed to report parking spots that are not within 50 meter distance of another existing parking spot. This check is performed on the backend before the reported information is stored into Elasticsearch. This check ensures that we do not store duplicate parking spots for the same location. Another feature that we implement is that we disallow users from updating parking spots more than 5 times a day. This offers security against bots who may try to earn unrestrained points by continuously reporting parking locations. This would also protect against cases where a user may try and earn points by repeatedly updating the same parking spot with the same number.

### Incentive-based Ideology

Our application does not contain any parking data of its own. It relies on its users to report and update parking spots in real-time. Therefore, we need to create some kind of an incentive for users to feed data to the app. A score-based approach works well for us as users require a certain minimum score to be able to find parking. If they fall short of this they are forced to feed available parking information to the app which guarantees an everlasting cycle of information supply and retrieval.

### Notification Support

We have registered our application with Google Cloud Messaging service which allows us to send notifications to users about the status of the operations - reporting and updating parking. Users are updated about the success/failure of the operation on their device. To implement this, a user’s device needs to communicate with GCM to obtain a unique registration token which is sent to the backend along with parking data. This unique token is thereafter used to notify the user via GCM.

## 4. Challenges

We have tried to cover most of the limitations while developing the app. However some challenges still persist which can be dealt with in the future. For example, we rely heavily on the fact that users report the correct parking information each time. However, there could be some users who feed false data in order to earn points. We have implemented a feature of tracking such users if the spots they report more than total available spots on that street. We look forward to coming up with better solutions to track such users and ensure that the wrong data does not get stored in Elasticsearch. Furthermore, log analysis of user data feed using Spark can give us great insights and help us eliminate potentially false information from our database. Furthermore, we would also like to work towards improving our algorithm for storing and retrieving parking locations and ensuring further data accuracy.

## 5. Conclusion

Parkwizard is an android app with a mission of mitigating the inconvenience suffered while finding street parking. It employs a crowd-sourcing, score-based approach where users earn points on reporting free parking spots and use those points to find parking when required. ParkWizard utilizes various cloud services to function as a scalable, reliable and secure application.