CS 4366: Senior Capstone Project
Dr. Sunho Lim
Project #4 - Implementation and Partial Demo - Project Report
EmergenSeek

Suhas Bacchu Derek Fritz Kevon Manahan Annie Vo Simon Woldemichael

April 1, 2019

Abstract

In this report, we describe and detail the progress that we have made so far in developing the backend and mobile client for the EmergenSeek application. The report will be broken up into sections such that, each section will answer the questions posed in the **"Deliverables"** section of the Project 4 statement file.

1   Project Restatement

As a reminder, EmergenSeek is a mobile application which will provide users with multiuse, centralized emergency information and notification services. This application will provide friends and family members with priority connections in times of emergency or crisis. For the scope of the following two months, within this class, our main function requirements are as follows.

1. S.O.S. button emergency broadcast — The user shall be able to utilize the mobile client to press and hold an S.O.S. button for automated notification of contacts and emergency services.

2. Emergency service locator — The user shall be able to utilize the mobile client to search for emergency service (i.e. hospitals, pharmacies, police boxes)

3. Periodic notifications to contacts (location-based polling) — The user shall be able to utilize the mobile client to periodically send their location information to contacts.

4. Granular permission definitions for contacts — The user shall be given full control over what contacts receive what level of information.

5. Lock screen display of health information for emergency services — In the case of an S.O.S. situation, the user shall have their health information displayed for the convince of first responders.

2   Lambda Functions

This section should be used as a reference when there is any mention of an implemented Lambda function. The functions are prefixed with `ES` (short for EmergenSeek). Please note that every instance of `user_id` is a Universally unique identifier (UUID) generated by Firebase Auth (managed by the Flutter client). This section will detail 6 things for each Lambda function:

1. The HTTP method necessary for interacting with the Lambda function.

2. The API Gateway route associated with the function.

3. The JavaScript Object Notation (JSON) <u>request</u> body required by the function to produce desired output.

4. The JSON <u>response</u> returned by the API for a request.

5. A simple-to-understand description of what the function performs as a result of the request.

6. Services and API's that this function is dependent upon.

a. **ESSendSMSNotification**

  (i) HTTP Method: POST

  (ii) API Gateway Route: /sms

  (iii) JSON Request Body:

```
{
        "user_id": String
        "type": int,
        "message": String
        "last_known_location": double[2]
}
```

  (iv) JSON Response Body (Success):

```
{
        "body": String
}
```

  (v) Description: First, this function will update the user's location in the database. Second, If the user provides 1 as an emergency type (SEVERE) this function will send an SMS message to the user's primary and secondary contacts and send a voice call to emergency services. If the user provides 2 as an emergency type (MILD) this function will only send an SMS message to the user's primary contacts. If the user provides 3 as an emergency type (CHECKIN) this function will only send an SMS message to the user's primary contacts with the contents of the "message" field. The "message" field is only required if "type" is 3. The response body of this function, on success, is simply a string stating that all operations completed successfully.

  (vi) Dependencies: DynamoDB, Twilio

b. **ESSendEmergencyVoiceCall**

  (i) HTTP Method: POST

  (ii) API Gateway Route: /voice

  (iii) JSON Request Body:

```
{
    "user_id": String,
    "last_known_location": double[2]
}
```

  (iv) JSON Response Body (Success):

```
{
        "body": String
}
```

  (v) Description: First, this function will update the user's location in the database. Second, the application will generate an XML document which conforms to Twilio's TwilML (Twilio Markup Language) specification [2]. This document specifies how the programmable voice call is programmed. Third, this application will uplaod the XML document to Amazon's Simple Storage Service so that Twilio may access it for every phonecall that is dispatched as a result of a single request to this Lambda function. Lastly, this function will call the user's primary contacts. Currently, we have not integrated anything which will determine the correct 911 phone number depending on the user's location. The response body of this function, on success, is simply a string stating that all operations completed successfully.

  (vi) Dependencies: DynamoDB, Twilio, Simple Storage Service

c. **ESPollLocation**

  (i) HTTP Method: POST

  (ii) API Gateway Route: /poll

  (iii) JSON Request Body:

```
{
    "user_id": String,
    "last_known_location": double[2]
}
```

(iv) JSON Response Body (Success):

```
{
    "body": String
}
```

(v) Description: First, this function will update the user's location in the database. Next, this function will utilize MapQuest's Geocoding API to translate the provided latitude and longitude into a readable address. Lastly, this function will send this updated location, in readable address format, to the user's primary and secondary contacts. The decision to use MapQuest instead of Google's Geocoding API was made as a result of there being a very friendly Golang library to take care of the all of the client code (`https://github.com/jasonwinn/geocoder`). The response body of this function, on success, is simply a string stating that all operations completed successfully.

(vi) Dependencies: DynamoDB, Twilio, MapQuest

d. **ESGetLockScreenInfo**

(i) HTTP Method: POST

(ii) API Gateway Route: /lock

(iii) JSON Request Body:

```
{
    "user_id": String,
}
```

(iv) JSON Response Body (Success):

```
{
    "first_name": String,
    "last_name":  String,
    "blood_type": String,
    "age": int,
    "primary_residence": {
      "line1": String,
      "line2": String,
      "city": String,
      "state": String,
      "country": String,
      "zip_code": String
    },
    "phone_pin": int,
    "email_address": String,
    "phone_number": String
}
```

(v) Description: First, this function will find a user with the matching user id in the database. Next the function will prepare a JSON object containing important user information for the user's lockscreen to be displayed on the client whenever the S.O.S. button is invoked. It is the responsibility of the client to invoke this function anytime the S.O.S. button is interacted with. It should be noted that while the functionality of this Lambda may also be accomplished through an HTTP GET request, using a POST request was simpler when defining the CloudFormation resource necessary for deploying this function from the CodePipeline of AWS CodeStar. More information on CodeStar may be found in the Continuous Integration / Continuous Deployment in section 4.

(vi) Dependencies: DynamoDB

e. **ESServiceLocator**

(i) HTTP Method: POST

(ii) API Gateway Route: /locate

(iii) JSON Request Body:

```
{
    "current_location": double[2]
}
```

(iv) JSON Response Body (Success):

```
[
  {
    "location": {
      "lat": double,
      "lng": double
    },
    "name": String,
    "icon": String,
    "open": bool
  }
]
```

(v) Description: First, this function will query the Google Places API to retrieve pharmacies within a 20 mile radius of the latitude and longitude provided. Next, it will do the same for hospitals and merge the results of these two queries, after extracting only the latitude and longitude of each location of interest, the locations name, its open status, and an icon to identify the location on the client's map.

(vi) Dependencies: DynamoDB, Google Places API

2.1 Additional Functions

Since beginning implementation (specifically since creating our SRS document), we have found the need for three additional Lambda functions; ESCreateUser, ESGetUser, and ESUpdateOnLoad. These are necessary for creating new users of the application, retrieving existing users, and updating the user's location everytime the application is reopened, respectively. The only function which we have known a need for, but have not implemented as a result of it having a low priority is ESUpdateContactPermissions. This function is responsible for controlling which contacts allows how precise of a result whenever a daily, polled SMS message is sent.

2.2 Client Code

An invocation of any of these functions from the client side will look similar to the following body of Dart code:

```dart
import "dart:convert";
import 'package:http/http.dart' as http;

void sendSMS() async {

  var url = "https://tzuvifn7ng.execute-api.us-east-2.amazonaws.com/Prod/sms";
  var headers = {"content-type": "application/json"};
  var body = jsonEncode({
    "user_id": "e78e0c86-f9ba-4375-9554-6dc1426f5605",
    "type": 3,
    "message": "Hello from Lambda, Dart"
  });

  final response = await http.post(url, headers: headers, body: body);
  print(response.body.toString());
}
```

Where https://tzuvifn7ng.execute-api.us-east-2.amazonaws.com/Prod is the direct URL of our API Gateway's production stage.

3 Project Implementation

In this section we will describe how we implemented the project and give an overview of how the backend and mobile client are communicating, as well as the role of all of the technologies involved.

As a whole, and general rule when following a client-server architecture, the backend is an abstraction of all of the operations necessary for the mobile applications functionality. An end-user or a developer on the front-end team will never need to know of the serverless architectural pattern [1] that our backend follows. Beginning from deriving our system features and requirements, we determined what Flutter components would be necessary to expose this functionality to the user and additionally determined the respective AWS Lambda function, written in Go, that would provide that functionality via an API call. That being said, the following subsections will describe precisely how this was achieved for each of the functionalities referenced in section 1. A reminder; details for any reference to a Lambda function prefixed with `ES` can be found in section 2

3.1 1 - S.O.S. button

This client feature depends on three backend Lambda functions. On the client side, the user will press and hold an S.O.S. button. After a set amount of time, visualized via a loading bar, the user will have the option to select an emergency severity. At the moment the client only implements a "severe" and "mild" emergency option.

- If the user selects `severe`, the application will make a request to ESSendSMSNotification and ESSendEmergencyVoiceCall via two, asynchronous HTTP POST requests.

- If the user selects `mild`, the application will make a request to only ESSendSMSNotification via a single asynchronous HTTP POST request.

- In the case of a `severe` selection, the application will make a request to ESGetLockScreenInfo to retrieve necessary personal information related to the user who pressed the S.O.S. button.
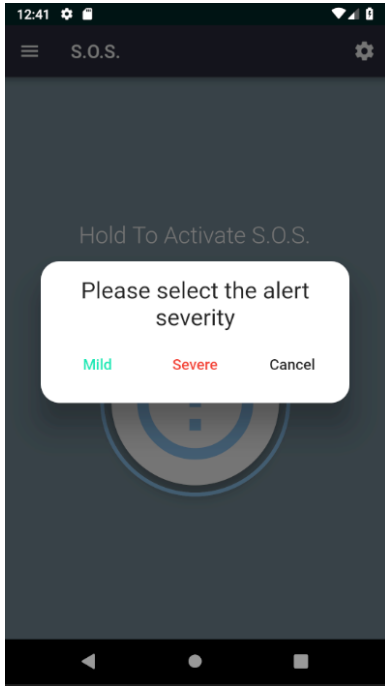

Figure 1: Client-side prompt requesting that the user select the emergency severity at the end of holding the S.O.S. button.
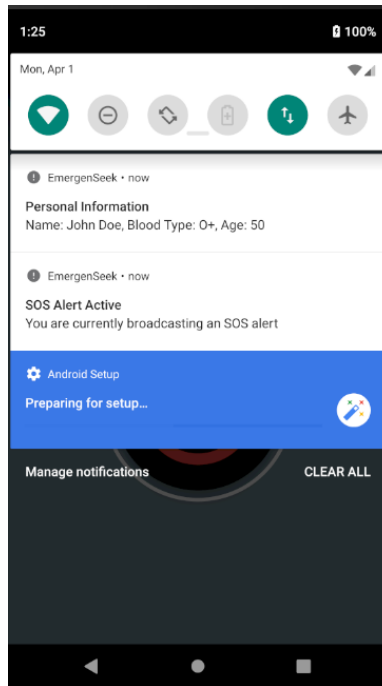
Figure 2: Example lockscreen data returned from the API as a result of invoking the S.O.S. button.
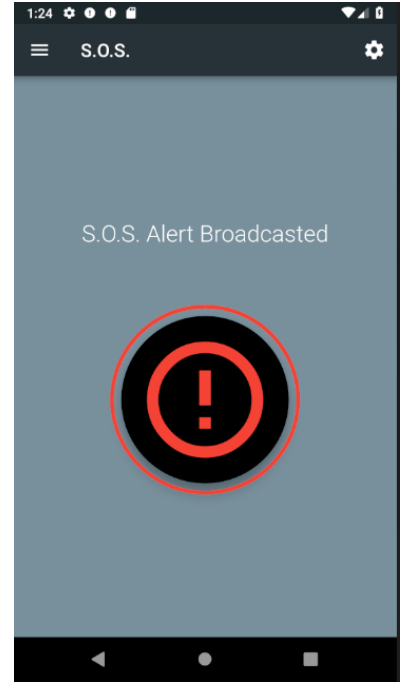
Figure 3: The state of the S.O.S. button after being invokved. As long as the button is in an invoked state, lockscreen information will be displayed.

3.2   2 - Emergency service locator

This client feature depends on only one backend Lambda function. On the client side, whenever the user opens the service locator application, a single asynchronous HTTP POST request will be made, containing the user's current location extracted from the device using native permissions. ESService-Locator will then response with a list of JSON objects each containing information about a hospital or pharmacy. The client will, using the data returned by ESServiceLocator, create a map markers. The markers will then be placed on a Google-style map for the viewing of the user. The Flutter package responsible for friendly displaying of the returned geo-data is a result of the `google_maps_flutter` plugin (`https://github.com/flutter/plugins/tree/master/packages/google_maps_flutter`).
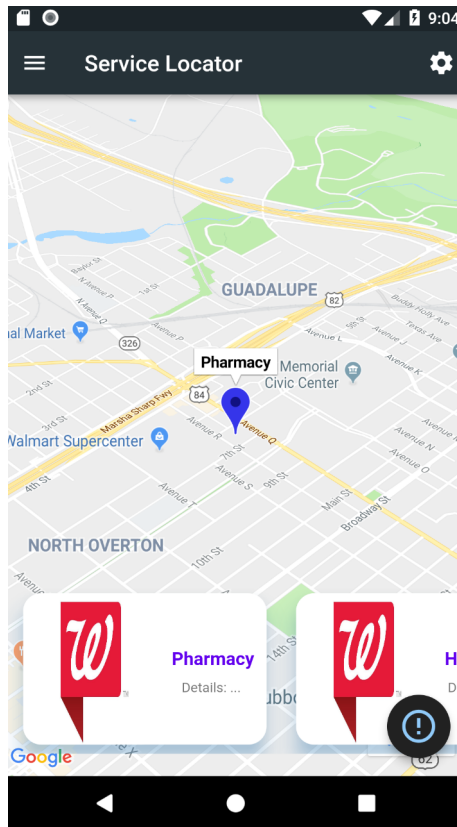

Figure 4: Screenshot of the service locator map containing mock data.

3.3   3 - Periodic notifications to contacts (location-based polling)

A note that this feature has not been implemented, but it's implementation details are something that we have discussed since our SRS document. This client feature depends on only one backend lambda function. On the client side, a primitive `Thread`, provided by the `threading` package, will start whenever the client first registers for the application and enables the location-based polling feature. By default, the thread will sleep for 24 hours. At the end of the sleep, the client will make a single asynchronous HTTP POST request to ESPollLocation containing the user's id, and their location at the time of the API call.

3.4   4 - Granular permission definitions for contacts

This client feature depends on one backend Lambda function. On the client side, user's are able to add and removes users from their native contacts book. Each contact is then assigned an alert tier. Depending on their alert tier, depends on how detailed the location information that they receive is. For example, a user assigned alert tier 1 would be able to see 2500 Broadway, Lubbock, TX 79409, USA, but a user assigned a tier 3 alert tier would only see TX, USA.
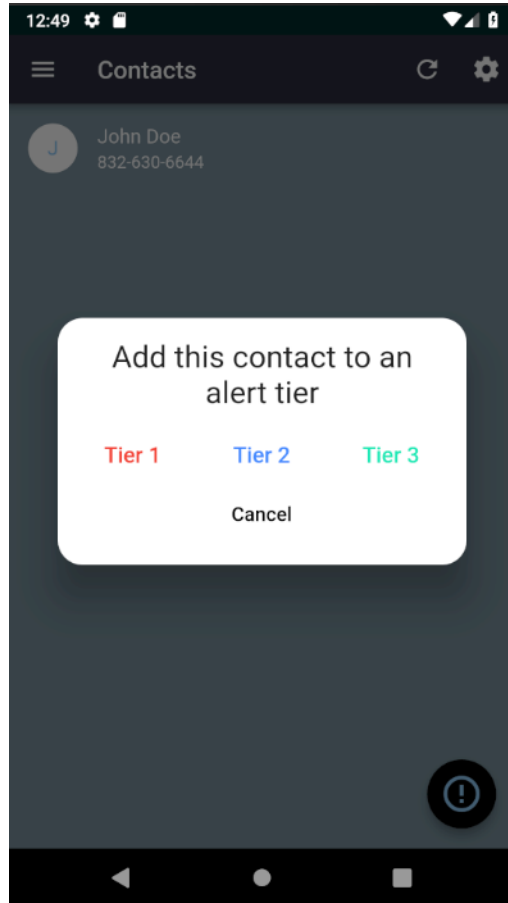
Figure 5: Client-side prompt containing alert tier selection for a single user.

4 Continuous Integration / Continuous Deployment

In this section, we will d

References

[1] What is the AWS Serverless Application Model (AWS SAM)? — `https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html`

[2] TwiML^{TM} for Programmable Voice — https://www.twilio.com/docs/voice/twiml