



EmergenSeek

**CS 4366: Senior Capstone Project**

**Dr. Sunho Lim**

**Texas Tech University**

**Project #5**

**Validation and Full Demo**

**EmergenSeek**

Spring 2019 Final Report

by

**Suhas Bacchu**

**Derek Fritz**

**Kevon Manahan**

**Annie Vo**

**Simon Woldemichael**

## Abstract

In this final report, we describe and detail the validation of EmergenSeek, a multiuse, cross-platform mobile application providing simple access to emergency services and contact connections. As with previous reports, we will structure the report by enumerating on the frontend and backend. Dissimilarly, we will only describe necessary details in the validation of our implementations, to the extent that, any other developer with some programming background will be able to successfully implement our system's components. Only necessary details will be presented to the reader and care should be taken when reading them<sup>1</sup>. Restatements from previous deliverable reports will be made to remind the reader of pre-defined, but important information. Also, a glossary is provided at the end of the report to define terms and jargon that may be confusing or unfamiliar for the reader.

*Keywords:* API Gateway, CI/CD, Cloud, Dart, Flutter, Golang, Lambda, Mobile, Serverless

---

<sup>1</sup>Concepts such as continuous integration, continuous deployment, microservices, serverless computing, NoSQL databases, APIs, and widget-based mobile development will be discussed. These new concepts may be difficult to grasp initially, but sufficient details will be provided.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Walkthrough Steps . . . . .	3
<b>2</b>	<b>Project Overview</b>	<b>3</b>
2.1	Project Definition, and Implemented Features . . . . .	3
2.2	Features In-progress . . . . .	4
2.3	Future Features . . . . .	4
<b>3</b>	<b>UML and Design Documentation</b>	<b>4</b>
3.1	Use-case Diagram . . . . .	5
3.2	Class Diagrams . . . . .	5
3.2.1	Frontend . . . . .	5
3.2.2	Backend . . . . .	10
<b>4</b>	<b>Implementation Details</b>	<b>12</b>
4.1	Flutter Development - Frontend . . . . .	12
4.2	Lambda Development - Backend . . . . .	12
<b>5</b>	<b>Project Development Lifecycle</b>	<b>13</b>
5.1	Go - Setup . . . . .	14
5.2	Informal Testing . . . . .	20
5.3	Formal Testing . . . . .	21
5.4	Deployment . . . . .	22
5.5	Integration . . . . .	25
<b>6</b>	<b>Conclusion</b>	<b>29</b>
6.1	Implementation Challenges . . . . .	29
6.1.1	Lambda Maximum Compute Time Limitation . . . . .	29
6.1.2	Identity Access and Management . . . . .	29
6.2	Group Member Contributions . . . . .	29
<b>7</b>	<b>Glossary</b>	<b>29</b>

# 1 Introduction

The development of EmergenSeek has been an effective software engineering experience for our group. In this final submission, we will discuss our project by (1) restating the idea, project motivations, and our successful implementations. Then, (2) we will discuss the final state of our UML diagrams and architecture, for this Spring semester. After, (3) we will discuss the development cycle for the frontend and backend and briefly touch on the microservices architecture that we follow.

Next, (4) a walkthrough of the discussed development lifecycle, using only necessary details and supplementing those details with additional code comments, snippets, and screenshots. This walkthrough will allow the user to understand the structure of our application, both frontend and backend. The walkthrough will not cover *everything* as the amount of content necessary for a walkthrough that assumes the reader has zero prior knowledge of the systems and concepts used in our project would take an obstructive amount of explanation. Then, (5) we will describe the current state of our serverless application repository; our collection of Lambda functions. To end (6) on our discussion of the validation, we will display Flutter views of the mobile application and their respective functionality.

It is very important to note that our project uses programming languages which are young in maturity and may not be familiar to the reader. But, our backend, written in Go, has syntax very similar to the C and C++ programming languages. Our frontend, written in Dart, has syntax very similar to the Java programming language. As a result of these similarities, it is permissible for the reader to assume what the code does as the syntax is, for the most part, straightforward. To conclude we will detail team member contributions, implementation issues, and steps moving forward with the project.

## 1.1 Walkthrough Steps

The previously mentioned walkthrough will consist of the following, and be enumerated for a single Lambda function and a single Flutter application screen. This section should be referred to in the Project Development Lifecycle section:

1. Feature expectation. Answer the question of “What feature do we need to implement and how do we implement it?”
2. Lambda function: Implement the feature and its expectations around the requirements of using AWS Lambda
3. Informal testing: Get an initial working version of the API Lambda function
4. Formal testing: Write unit tests for the implementation
5. Integration and Infrastructure as Code: Define the deployment
6. Continuous Integration: Do the deployment
7. Integrate Lambda functionality with the frontend
  - 7a. Create feature view. The view is effectively the screen that the user will interact with.
  - 7b. Create feature model. Define how features will be referenced by the state of the application.
  - 7c. Interface with Lambda function. Make calls via an HTTP client to the Lambda function

As all of our project code is open-source, readers are encouraged to reference the code repositories for our project. Repositories also contain additional setup notes and details. They are located at <https://github.com/emergenseek/backend> and <https://github.com/emergenseek/Flutter-Client> for our backend and frontend code, respectively. Readers who would like more insight, or have questions are welcome to open an issue or pull request on either of these GitHub repositories.

## 2 Project Overview

### 2.1 Project Definition, and Implemented Features

EmergenSeek is a mobile application which provides users with multiuse, centralized emergency information and notification services. This application gives friends and family members priority connections in times of emergency or crisis. Our successfully implemented, core feature are as follows:

1. S.O.S. button emergency broadcast — Users are able to utilize the mobile client to invoke an S.O.S. button for automated notification of contacts and emergency services.
2. Emergency service locator — Users are able to utilize the mobile client to search for emergency service (hospitals, and pharmacies). Currently, searching will return results up to a fixed radius of 20 miles.
3. Granular permission definitions for contacts — The user is given full control over what contacts receive what level of information. This is achieved by setting alert tiers on a per-contact basis.
4. 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 respondents.

Additionally, successfully implemented, quality of user-experience features are as follows. While the application is still functional without them, deploying the application for public, functional use, cannot not be possible without them.

1. User registration and login — Users are able to register for an account and track their personal information, necessary in an emergency event.
2. Settings persistence – Users are able to change settings within the application to fit their preference.

## 2.2 Features In-progress

Features which are partially implemented are as follows:

1. Periodic notifications to contacts (location-based polling) — The user shall be able to utilize the mobile client to send periodically send their location information to contacts. While the logic is in-place, we currently do not have a successful implementation of repeating the location polling.
2. Retrieval of international emergency service numbers (ambulance, fire, and police), depending on the user's current location. Connections to the front-end have not been made yet.

## 2.3 Future Features

Lastly, additional quality of user-experience features that have not been, but maybe implemented in the future are:

1. More extensive error handling and reporting (via something like the Catcher, a Flutter package or Sentry.io a multi-language, multi-framework error reporting and management service)
2. More in-depth user preferences, settings and options
3. Alternative sign-up and log-in methods (Google, Facebook, etc.)

## 3 UML and Design Documentation

In this section we will discuss our completed Unified Modeling Language (UML) diagrams and design documents. Items are presented with a figure of the design unit and briefly described.

### 3.1 Use-case Diagram

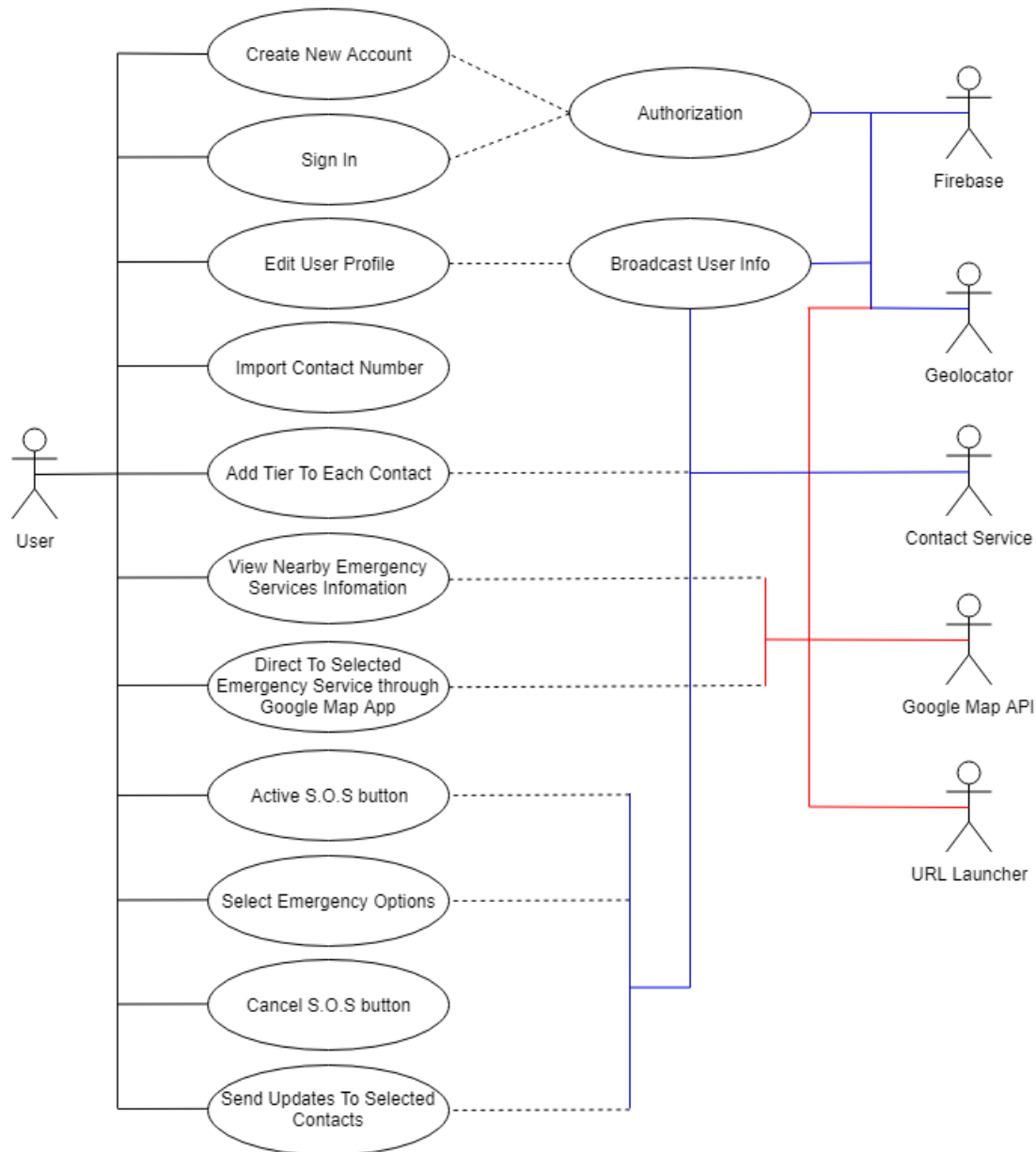


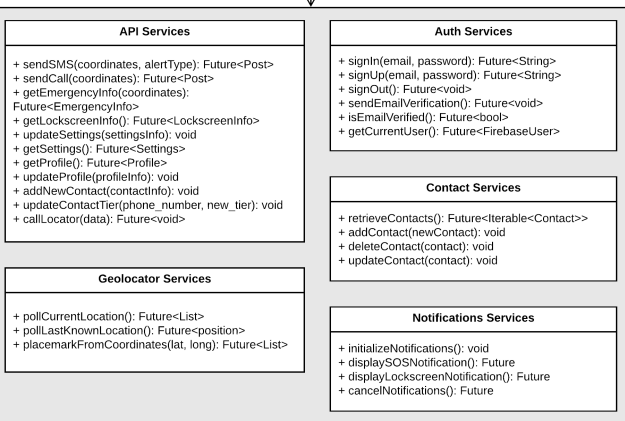
Figure 1: Updated use-case diagram for the EmergenSeek application.

In this use-case diagram we detail external dependencies and their inclusion to each of the use cases. Authorization is dependent on Google's Firebase service. The broadcasting of user info is depending on using MapQuest's Geolocator API. The contact service allows us to read contacts directly from the user's phone. It is important to note that this has only been tested on Android devices. The Google Maps API coupled with the URL launcher allows us to connect our service locator with the user's native Google Maps application. Whenever the user selects a location, they will be directed to the location by tapping on the marker associated with it. Creating new accounts and signing the user in extend the authorization item. All S.O.S. button and emergency related use-cases extend the broadcasting of user info to the backend.

### 3.2 Class Diagrams

#### 3.2.1 Frontend

For frontend class diagrams, we will first show the full models, then discuss the models, broken up into 3 smaller components, Views, Models, and Services.



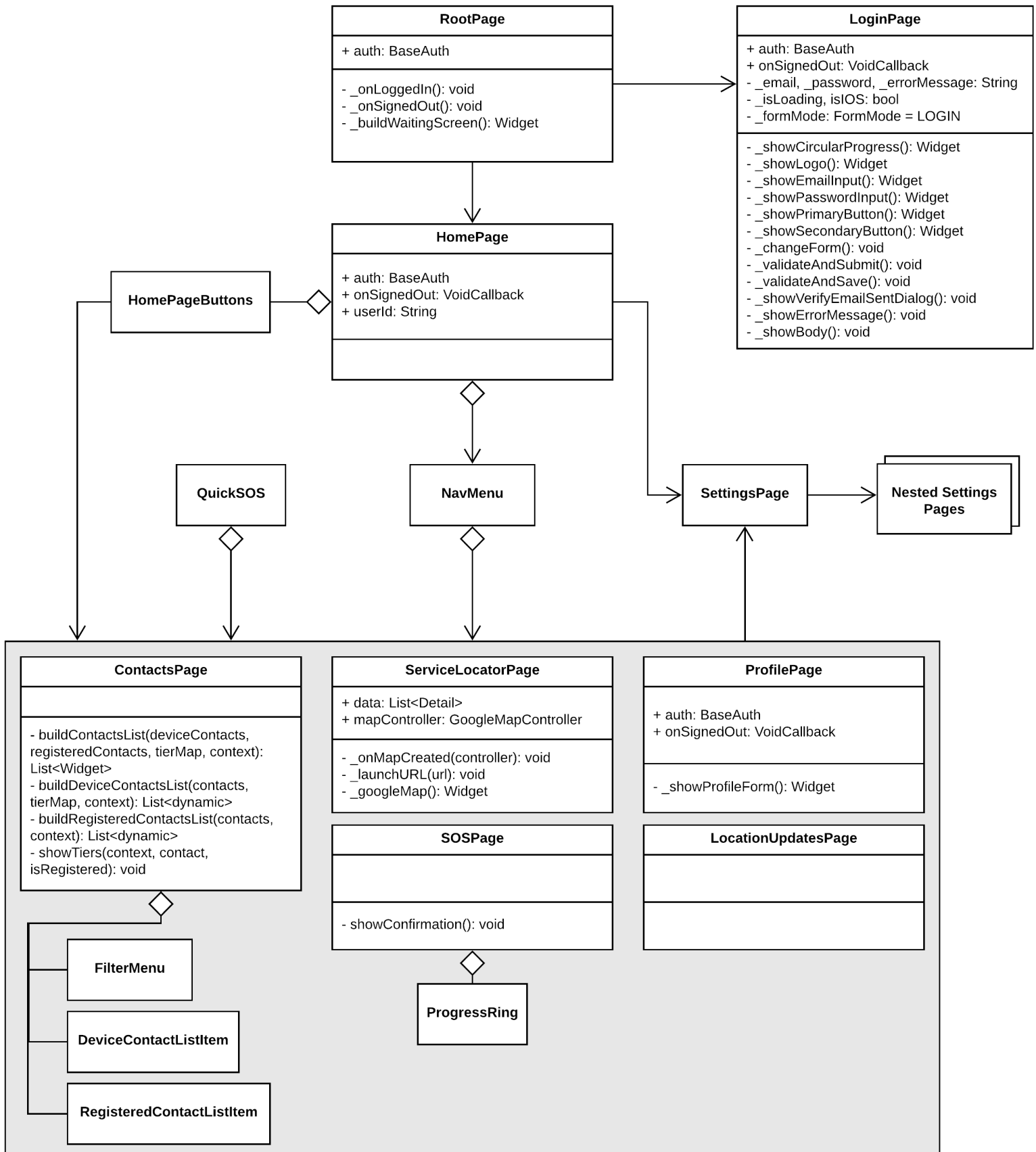


Figure 3: View UML class diagram for the frontend.



The application's views begins via a RootPage, this root page can be thought of a session manager keeping track of the user through an `auth` object. This object tracks actions that occur on certain events; if the user is logged in, signed out, or waiting for some communication with the backend to complete. This session state is managed via the LoginPage. From the RootPage we have a HomePage. This homepage is displayed once the base authentication, encapsulated as an attribute of the RootPage class, has received a successful response from Firebase and tells the application that the user is authorized. From the home page, we have buttons that connect the user to other views. These views, ContactsPage, ServiceLocatorPage, ProfilePage, LocationUpdatesPage, menus and navigations are displayed at the end of our report.

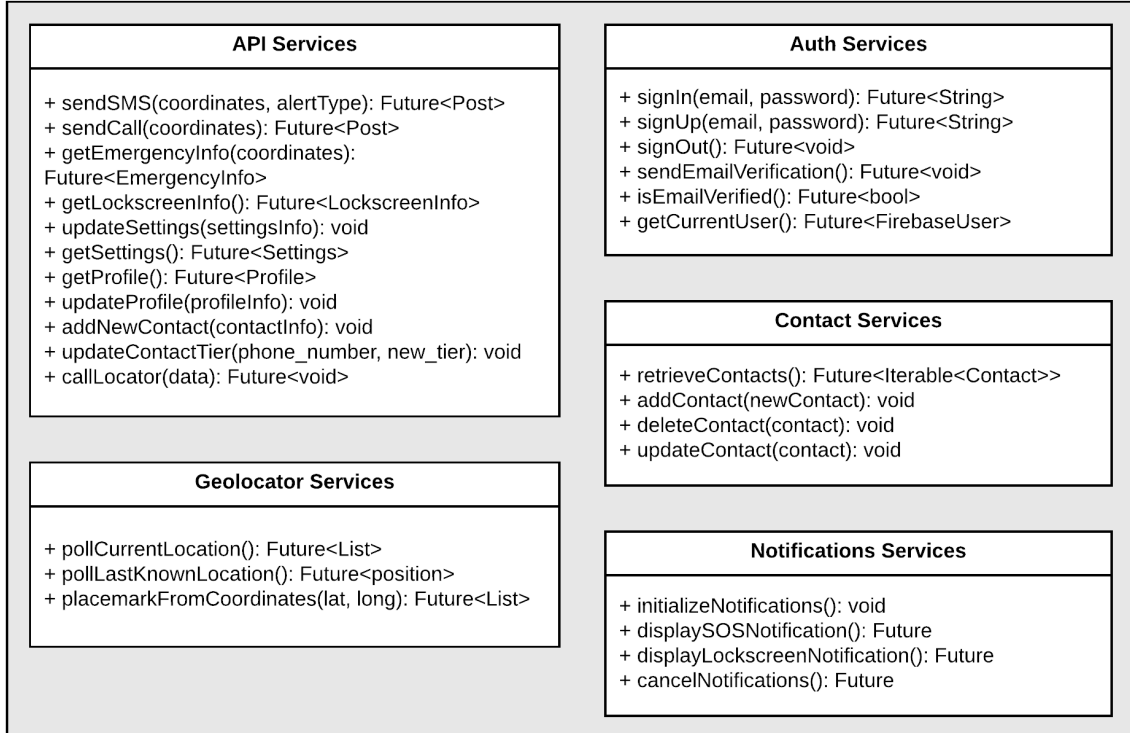


Figure 4: Service UML class diagram for the frontend.

The application's services are necessary for managing external API's, both Firebase and our Lambda functions. The API services class encapsulates many, general API endpoints proxied by AWS API Gateway. The Auth Services class communicates with Firebase to register, login, and logout users. The Geolocator Services class entails methods necessary for the Google Maps integrate map we use for our service locator. The Contact Services class communicates with the native contacts API on the device's host operating system to retrieve the user's contacts. The Notifications Services class encapsulate methods necessary for displaying notifications on the device, both within the notifications center and on the user's lock screen.

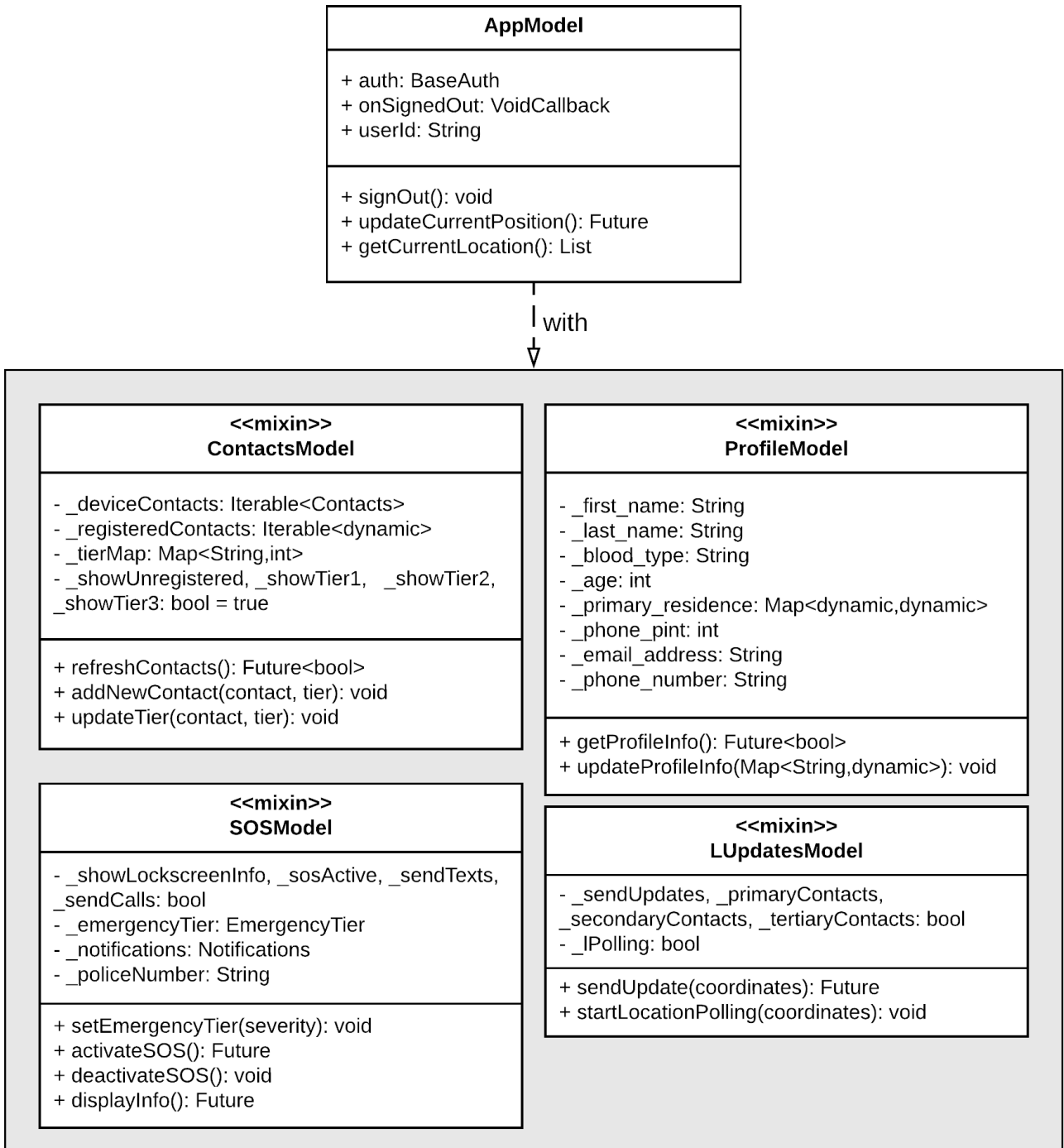


Figure 5: Model UML class diagram for the frontend.

The application's models and global state are tracked in an **AppModel** instance, this **AppModel** is how variables and application flow is tracked, regardless of what view the user is interacting with. The **AppModel** then has several **mixin** objects. A **mixin**, similar to inheritance, is a class which requires functionality from other base classes, but does not need to become the parent of that class. Though mixins, multiple inheritance is possible and allows our application to utilize the scoped models, discussed below.

### 3.2.2 Backend

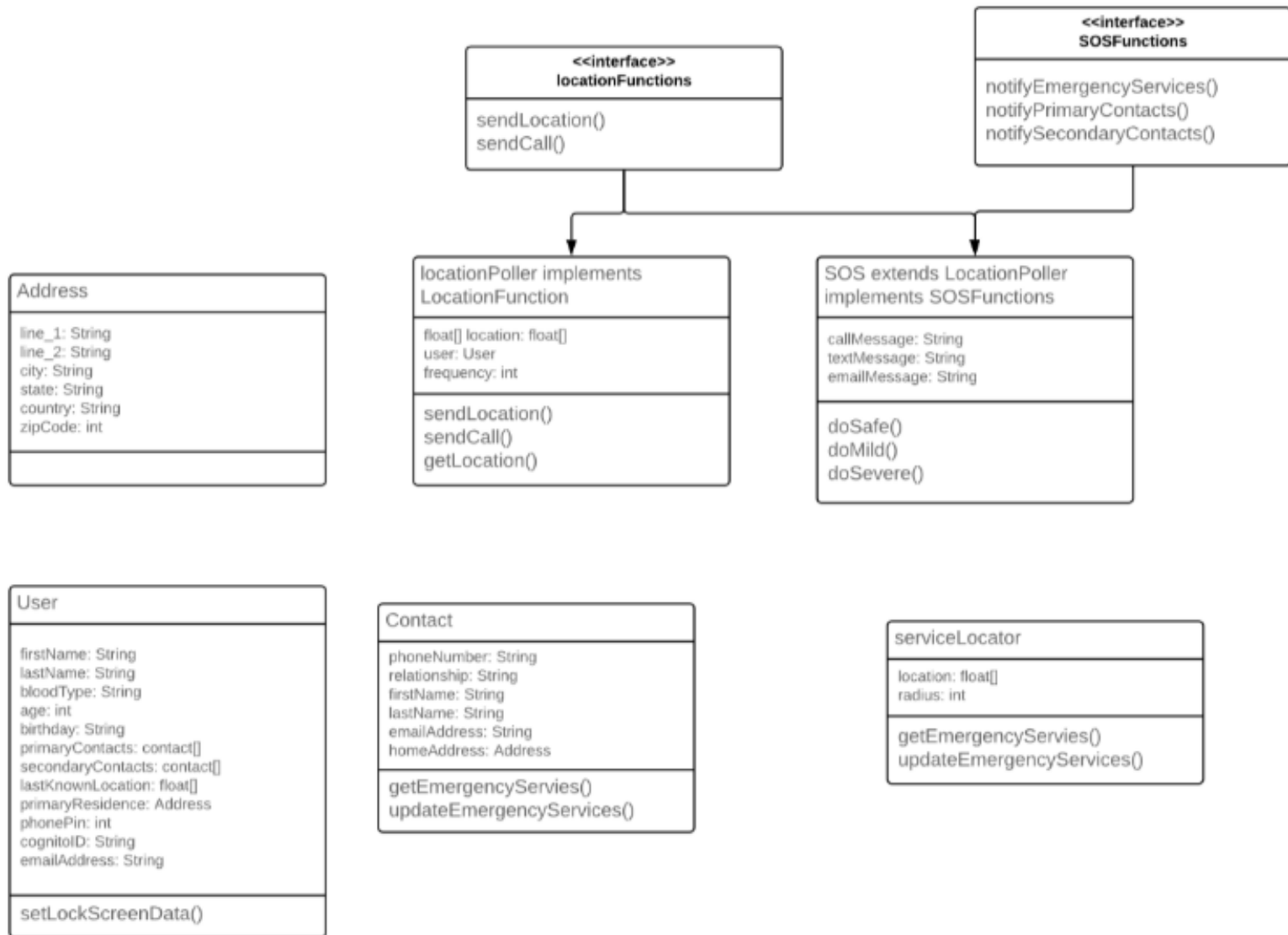


Figure 6: Class diagram for the EmergenSeek backend.

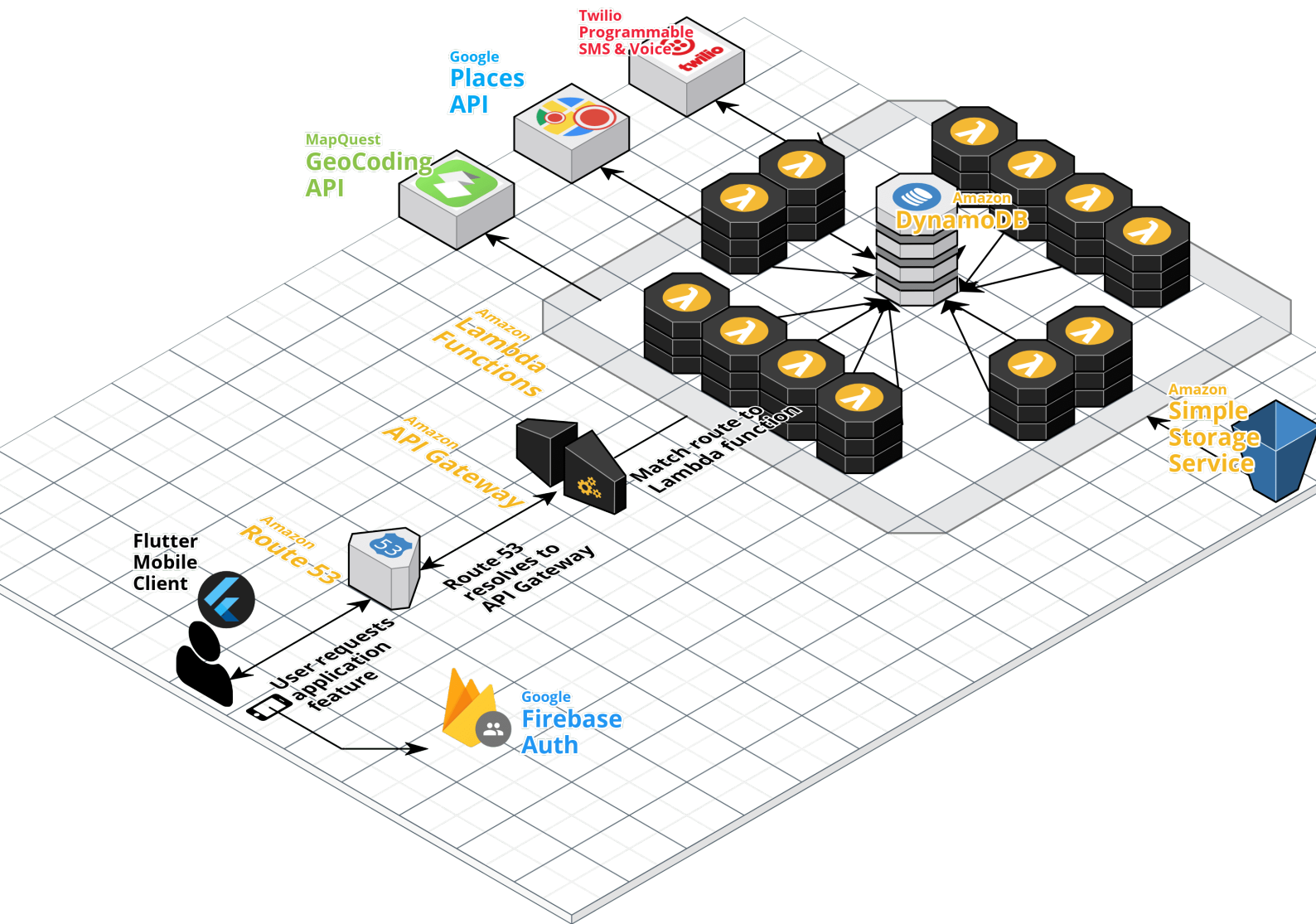


Figure 7: Updated Cloudcraft diagram of the AWS resources, external APIs encapsulated by the backend, and frontend connection abstractions.

No major changes have been made to this document since previous report previsions. This class diagram should be used as a loose reference and not a literal representation of how our backend is structured for reasons related to how Go programs are written. In this class diagram for our backend we have:

1. Address — Encapsulates the data necessary for representing an address.
2. User — Encapsulates the data necessary for representing a user. Has 1 method which retrieves the user's information using a Lambda function for display on the lock screen.
3. Contact — Encapsulates the data necessary for representing a user's contact. Has 2 methods which get and update the contact on the emergency services map so that the user may see which of their contacts are nearby.
4. ServiceLocator — Encapsulates the data necessary for performing location based functions. Has two methods which will get and update emergency service locations for the map on the client.
- 5 LocationFunctions — (Interface) This interface contains the polymorphic methods necessary for performing location related functions. This interface is implemented by the LocationPoller instance.
6. SOSFunctions — (Interface) This interface contains the polymorphic methods necessary for providing functionality to the S.O.S. button. This interface is implmeneted by the SOS instance.

7. LocationPoller — Encapsulates the data necessary for providing functionality to the location polling feature. This instance implements the LocationFunctions interface.
8. SOS — Encapsulates the data necessary for defining the SOS buttons functionality. This instance inherits the LocationPoller instance and implements the SOSFunctions interface. It should be noted that the `emailMessage` attribute does exist because for the scope of this class we focused on and implemented SMS and voice notifications.

## 4 Implementation Details

The implementation of EmergenSeek may be broken of into two core components, Amazon Web Services' (AWS) Lambda and the cross-platform mobile development framework Flutter. These two components also have their own respective, external dependencies and services. Our Lambda functions were written using the Go programming language and the Flutter application depends on the Dart programming language. The following subsections should be read as a preface for the walkthrough of our development lifecycle.

### 4.1 Flutter Development - Frontend

For the implementation of a new feature within the client, several aspects must first be considered. Nearly every feature is broken down into three components: the feature view, the feature model, and the interfaces required to generate the content of the feature. During our development, we would typically begin a feature's implementation with the feature view. By beginning a feature's development with the feature view, we were able to prioritize the user-experience and develop a clearer goal of our features based on what would be most intuitive as a user. The feature view essentially consists of a widget representing a page within the app, along with any sub-widgets representing individual graphical components within the page.

Our top-level page widgets generally started off as stateless widgets, with certain widgets being refactored into stateful widgets as their requirements and responsibilities became better defined. This allowed us to minimize each widget's internal state management as much as possible, instead delegating state management to our feature models. After developing a general prototype of our feature's view, we would move onto implementing the feature model. While our stateful widgets manage small sets of internal state data, a more accessible and persistent state model was necessary for our service. To address this, we utilized Flutter's scoped models [1]. Nearly every key feature possesses a feature model which contains important state data relevant to that feature. These models also provide an interface for managing the feature states from anywhere within the app. Once a feature model is created, it is included as a reference within our app-wide model which is passed into the widget hierarchy at the root of our application.

After creating both the feature view and model, we would move onto considering how the feature would interface with our backend or third-party services. After determining which backend Lambda functions would be relevant to a specific feature, the respective API requests would be written as modular functions within an API service class. Some features required additional Flutter plugins to generate their content as well, which would be imported and utilized within a wrapper class. After implementing our feature services, we would then integrate the appropriate calls within the feature model. We would then add our model references within the feature view and call the various model functions for accessing and managing the feature's state data. One of our main considerations in implementing new features was ensuring a responsive user-experience. Many of our features rely on retrieving and updating data from our backend, which led to challenges in rendering our views quickly. Our solution to this was utilizing Flutter's future builder widget, which allowed us to display placeholders while our API requests were sent and the responses were received and processed.

### 4.2 Lambda Development - Backend

Before discussing our implementation, we will give some background on why we chose to use AWS Lambda. Lambda is a serverless compute service, canonically referred to as a Functions-as-a-Service (FaaS) offering. This means that we do not need to manage any servers to deploy our code. Additionally, this managed compute is scaled up and down as necessary. If our application were to, overnight, receive thousands and thousands of users, we would be able to handle that load as a result of Lambda's auto-scaling functionality. Now that we have some background of Lambda, on to the implementation details.

To start, we create a specialized Go function that conforms to the expectations of the Lambda software development utilities provided by Amazon. This function receives an event, which is simply

an HTTP request with or without a body, and returns an event; an HTTP response with a body. The logic that occurs to process the request and return a desired response is where backend logic comes in. Though communication of various APIs (Google, MapQuest, and Twilio) and our DynamoDB database tables, we can produce these results to fulfill the requirements and expectations desired by our frontend users. These users do not have to think about how these actions occur because they are abstracted away. Instead they only need to know the expected request parameters, and the expected response results. Additionally, because our backend is, as a whole, a REST API, we can create multiple clients on multiple platforms (i.e. web in addition to mobile). While these are not entirely all of the components necessary for developing a fully functioning, long-term and highly scaleable application programming interface, they more than suffice for the scope of this course.

After implementing, formally, and informally testing the Lambda function on our local machines, it is ready for deployment. This deployment happens in three stages, all managed by AWS CodeStar. The first stage happens whenever a push to our `master` Git branch is done. AWS CodeStar will trigger a build via the CodeBuild service. When the build is triggered, the second stage will begin. CodeBuild will read the `buildspec.yml` file in the root of our repository and build binaries out of all of the Go code responsible for each function. The binaries will then be put into a Simple Storage Service bucket so that CloudFormation may retrieve them when creating each Lambda function and associating an API Gateway endpoint to it. CloudFormation is a key part of step three, the deployment. After the binaries have been build and put in an accessible place, CloudFormation will generate updates to our infrastructure by referencing a `template.yml` file, also in the root of our repository. This ever important, `template.yml` is referred to as Infrastructure as Code. This YAML definition will be enumerated on in the next section.

### *Microservices vs. Monoliths*

As students, a monolithic projects are something that we are very familiar with. A monolith is a centralized codebase that contains all of the components and assets of a project or product. Creating a monolith is definitely a lot simpler because everything relating to the project has the same locality. They typically run on the same server, use the same resources (CPU/RAM), are sometimes distributed as executable binaries. The real issues arise when changes need to be made and components need to be scaled. Developers of the monolith must wait on the components and features that *their* assigned components and features are dependent upon before they may start working. Until then, they cannot begin development on a particular part of the product. This can be mitigated with additional planning and project management techniques, but is definitely a large negative that monoliths pose. Additionally, the core features of the monolith must all be written in 1 language, otherwise, additional and often convoluted implementations must be done for multiple languages to communicate with each other despite sharing the same locality. For example, running each language as a process and performing some form of interprocess communication using threads or internal web servers. Monoliths are very tightly-coupled with many strong dependencies.

---

Microservices, on the other hand, are decentralized, independent components of a larger product that each do one thing well for the sake of the application as a whole. This is why we chose to follow this style of development. This is why we chose to follow a microservices architecture. We did not need to wait on one another to fulfill the implementation of a Lambda function. These loosely-coupled components bring up many more challenges than a monolithic product, but their modularity and distributable attributes make them much easier and much faster to develop and deploy [2].

## 5 Project Development Lifecycle

In this section we discuss our actual implementation details, beginning with the backend and moving on to the frontend. For the benefit of the reader, we will discuss our most complex Lambda function, `ESSendEmergencyVoiceCall`, which provides the frontend the ability to make phone calls via Twilio's API. The programming styles, and patterns used can be generalized to our other functions. Because of this, we will only cover the full lifecycle of one function in this section. Again, it is recommend that readers reference our GitHub repositories for more details and additional notes of reference: (<https://github.com/emergenseek/backend> and <https://github.com/emergenseek/Flutter-Client>). Details such as obtaining Twilio and MapQuest API keys, and configuring AWS are left up to the reader.

## 5.1 Go - Setup

First, we begin with our backend repository, in this repository we have 2 key folders, **common** and **functions**. **Common** encapsulates, four packages, **package database**, **package driver**, **package models**, and **package notification**. The names of these packages are self explanatory. Database encapsulates database functions, driver encapsulates important backend logic functions, models encapsulates data models, and notification encapsulates functions related to Twilio Voice and SMS communication. The **const.go** file defines constants used through the application and belongs to the parent package, **package common**. The **functions** folder encapsulates our serverless application repository, for this walk through we will only focus on **ESSendEmergencyVoiceCall**. Within the **ESSendEmergencyVoiceCall** we have 6 key files.

1. **event.json** – A test request sent to the function
2. **main\_test.go** – Unit tests for the function
3. **main.go** – The function definition (Lambda handler)
4. **Makefile** – For convince, to make local testing easier
5. **request.go** – Defines the expected request body or parameters which will be sent to the function.
6. **template.yaml** – Used for local testing of Lambda and API Gateway using the Serverless Application Model command-line interface and Docker

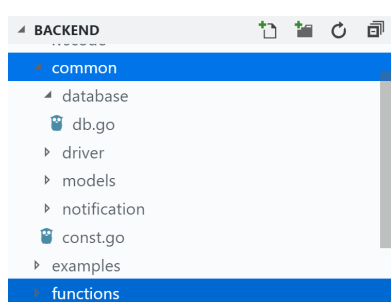


Figure 8: Screenshot of the functions folder and the contents of the common folder in the backend repository.

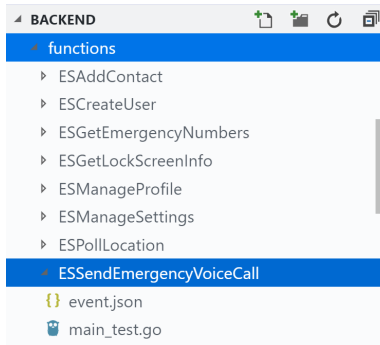


Figure 9: Screenshot of the contents of the functions folder in the backend repository.

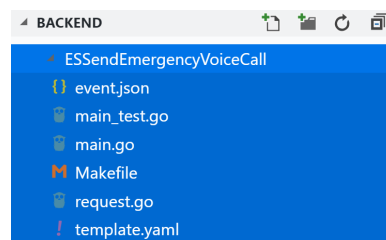


Figure 10: Screenshot of the contents of the ESSendEmergencyVoiceCall repository within the functions folder.

We will begin with **main.go**. **main.go** defined within the **package main** namespace, houses the **func main** function. This is a requirement of the Go programming language, the entrypoint should be called **func main** and must exist within **package main**. Within **func main** was have a call to **lambda.Start** and pass in **Handler**. **Start** is a function within the **lambda** package exposed by the "[github.com/aws/aws-lambda-go/lambda](https://github.com/aws/aws-lambda-go)" import and **Handler** is a function defined above **func main**. Anytime the **ESSendEmergencyVoiceCall** function is called, the code within the **Handler** is executed. The function passed to **lambda.Start** does not need to be called **Lambda**, it must simply be a function with has the expected signature; 1 input parameter of type **events.APIGatewayProxyRequest** and two return parameters of type **events.APIGatewayProxyResponse** and **error**, in that exact order.

As we see in the screenshot, the first duty of the **Handler** is to verify the request. This mean to match the expectation of the API against the request body provided by the user. To do this, we use a helper function that we defined, called **verifyRequest**. If any expectations are failed, we return an error to the user.

```
func main() {
    // AWS Lambda function entrypoint
    lambda.Start(Handler)
}
```

Figure 11: Screenshot of `func main` within package `main`

```
3 import (
4     "encoding/json"
5     "fmt"
6     "net/http"
7
8     retry "github.com/avast/retry-go"
9     "github.com/aws/aws-lambda-go/events"
10    "github.com/aws/aws-lambda-go/lambda"
11    "github.com/emergenseek/backend/common"
12    "github.com/emergenseek/backend/common/driver"
13 )
```

Figure 12: Screenshot of imports defined for the function. Their accessors are either defined as an alias to the left of the import name (like `retry`), or as the package name defined by the dependency. Standard library packages are defined above external, third-party dependencies.

```
33 // Handler is the Lambda handler for ESSendEmergencyVoiceCall
34 func Handler(request events.APIGatewayProxyRequest) (events.APIGatewayProxyResponse, error) {
35     // Verify the request
36     req, status, err := verifyRequest(request)
37     if err != nil {
38         return driver.ErrorResponse(status, err), nil
39     }
40 }
```

Figure 13: Screenshot of the first few lines of the `Handler` definition

```
15 func verifyRequest(request events.APIGatewayProxyRequest) (*Request, int, error) {
16     // Create a new request object and unmarshal the request body into it
17     req := new(Request)
18     err := json.Unmarshal([]byte(request.Body), req)
19     if err != nil {
20         return nil, http.StatusUnprocessableEntity, err
21     }
22
23     // Make sure all of the necessary parameters are present
24     err = req.Validate()
25     if err != nil {
26         return nil, http.StatusBadRequest, err
27     }
28
29     // All checks passed, return req struct for use. http.StatusOK is ignored
30     return req, http.StatusOK, nil
31 }
```

Figure 14: Screenshot of the definition for `verifyRequest`

The request checked against in Figure 14, line 17, is defined in `request.go`. In the Go programming language, data within the same package may be referenced globally. This is known as the package scope (within the same folder, ES. The importance of struct tags (`'json:"user_id"'`), should be noted as these define the expectation of the request sent by the user.



```

7 | // Request defines the expected body parameters of ESSendEmergencyVoiceCall
8 | type Request struct {
9 |     // The ID of the user making the request
10 |     UserID string `json:"user_id"`
11 |
12 |     // When requests are made to this Lambda function,
13 |     // it is assumed that the emergency type is 1 (SEVERE)
14 |     // Type common.EmergencyType `json:"type"`
15 |
16 |     // The user's location at the time of the request
17 |     Location []float64 `json:"last_known_location"`
18 | }

```

Figure 15: Screenshot of Request type.

```

20 | // Validate validates a request to the ESSendSMSNotification Lambda function
21 | func (r *Request) Validate() error {
22 |     // Check if the UserID is present
23 |     if r.UserID == "" {
24 |         return fmt.Errorf("user_id field is required")
25 |     }
26 |
27 |     // Check if the Location is present
28 |     if r.Location == nil {
29 |         return fmt.Errorf("last_known_location field is required")
30 |     }
31 |
32 |     return nil
33 | }

```

Figure 16: Screenshot of Validate, pointer-receiver method bound to the Request type.

Now that the request is validated, we can continue the function execution, the flow is as follows, annotated with code line numbers present in the next six figures, and an additional two.

```

41 | // Initialize drivers
42 | db, twilio, sess, mapKey := driver.CreateAll()
43 |
44 | // Retrieve user from database
45 | user, err := db.GetUser(req.UserID)
46 | if err != nil {
47 |     return driver.ErrorResponse(http.StatusBadRequest, err), nil
48 | }
49 |
50 | // Update the user's last known location
51 | err = db.UpdateLocation(user.UserID, req.Location)
52 |
53 | // Convert the user's last known location to a human readable address
54 | address, err := driver.GetAddress(req.Location, mapKey, false, 0)
55 | if err != nil {
56 |     return driver.ErrorResponse(http.StatusInternalServerError, err), nil
57 | }
58 |
59 | // Create the Twilio Markup Language necessary for the voice call
60 | twilML, err := driver.CreateTwilMLXML(user, address)
61 | if err != nil {
62 |     return driver.ErrorResponse(http.StatusInternalServerError, err), nil
63 | }

```

Figure 17: Screenshot of lines 42 through 63 of func Handler

```

54 // CreateAll initializes the necessary API providers for Lambda handlers
55 func CreateAll() (*database.DynamoConn, *notification.TwilioHandler, *session.Session, string) {
56     // Create a shared session
57     sess := session.Must(session.NewSession(&aws.Config{Region: aws.String(common.Region)}))
58
59     // Initialize database
60     db := &database.DynamoConn{Region: common.Region}
61     err := db.Create(sess)
62     if err != nil {
63         panic(err)
64     }
65
66     // Get MapQuest credentials
67     mapsKey := db.MustGetMapsKey()
68
69     // Get Twilio client credentials using database
70     twilio := &notification.TwilioHandler{}
71     err = twilio.GetCredentials(db)
72     if err != nil {
73         panic(err)
74     }
75     // Authenticate using credentials
76     err = twilio.Authenticate()
77     if err != nil {
78         panic(err)
79     }

```

Figure 18: Screenshot of func CreateAll within package driver

```

35 // GetUser will retrieve a user from the database
36 func (d *DynamoConn) GetUser(uid string) (*models.User, error) {
37     // Create user struct to be searched for using provided uid
38     userKey := &models.User{
39         UserID: uid,
40     }
41     key, err := dynamodbattribute.MarshalMap(userKey)
42     if err != nil {
43         return nil, err
44     }
45     input := &dynamodb.GetItemInput{
46         Key: key,
47         TableName: aws.String(common.UsersTableName),
48     }
49
50     // Search for user matching uid in table
51     result, err := d.Client.GetItem(input)
52     if err != nil {
53         return nil, err
54     }
55
56     // Unmarshal user into struct
57     user := &models.User{}
58     err = dynamodbattribute.UnmarshalMap(result.Item, &user)
59     if err != nil {
60         return nil, err
61     }
62
63     // Cheap check for item not found
64     if user.FirstName == "" {
65         return nil, errors.New("user not found")
66     }
67
68     return user, nil

```

Figure 19: Screenshot of func GetUser within package database

```

111 // UpdateLocation updates the location of a user when a location poll is invoked
112 func (d *DynamoConn) UpdateLocation(userID string, location []float64) error {
113     var LocationUpdate struct {
114         LastKnownLocation []float64 `json":1"`
115     }
116
117     // Marshal the update expression struct for DynamoDB
118     LocationUpdate.LastKnownLocation = location
119     expr, err := dynamodbattribute.MarshalMap(LocationUpdate)
120     if err != nil {
121         return err
122     }
123
124     // Define table schema's key
125     key := map[string]*dynamodb.AttributeValue{
126         "user_id": {
127             S: aws.String(userID),
128         },
129     },
130 }
131
132 // Use marshalled map for UpdateItemInput
133 item := &dynamodb.UpdateItemInput{
134     ExpressionAttributeValues: expr,
135     TableName:                 aws.String(common.UsersTableName),
136     Key:                       key,
137     ReturnValues:              aws.String("UPDATED_NEW"),
138     UpdateExpression:          aws.String("set last_known_location = :1"),
139 }
140
141 _, err = d.Client.UpdateItem(item)
142 if err != nil {
143     return err
144 }

```

Figure 20: Screenshot of func UpdateLocation within package database

```

106 func GetAddress(latlng []float64, key string, useTier bool, tier common.AlertTier) (string, error) {
107     // Retrieve location from MapQuest Geocoding API
108     geocoder.SetAPIKey(key)
109     a, err := geocoder.ReverseGeocode(latlng[0], latlng[1])
110     if err != nil {
111         return "", err
112     }
113     // Filter data returned depending on provided AlertTier
114     var address string
115     if a.Street != "" && tier == common.FIRST {
116         address = fmt.Sprintf("%v, ", a.Street)
117     }
118
119     if a.City != "" && (tier == common.FIRST || tier == common.SECOND) {
120         address = address + fmt.Sprintf("%v, ", a.City)
121     }
122     if a.State != "" {
123         address = address + fmt.Sprintf("%v, ", a.State)
124     }
125     if a.PostalCode != "" {
126         address = address + fmt.Sprintf("%v, ", a.PostalCode)
127     }
128     if a.CountryCode != "" {
129         address = address + fmt.Sprintf("%v", a.CountryCode)
130     }
131
132     return address, nil
133 }

```

Figure 21: Screenshot of func GetAddress within package driver

```

135 // CreateTwilMLXML creates the XML necessary for the gotwilio.NewCallbackParameters invocation in notification.SendVoiceCal
136 func CreateTwilMLXML(user *models.User, lastLocation string) ([]byte, error) {
137     // Split phone number so Twilio voice doesn't read it numerically
138     splitPhoneNumber := fmt.Sprintf("%v", strings.Split(user.PhoneNumber, ""))
139
140     // Create message using address and user's information
141     name := user.FormattedName()
142     message := fmt.Sprintf("This is an automated emergency call from EmergenSeek on behalf of %v. ", name)
143     message = message + fmt.Sprintf("They are in need of emergency assistance. Please send help to %v. ", lastLocation)
144     message = message + fmt.Sprintf("Please attempt to call %v at %v. Thank you.", name, splitPhoneNumber)
145
146     // Create, format, and return the XML document
147     doc := etree.NewDocument()
148     doc.CreateProcInst("xml", `version="1.0" encoding="UTF-8"`)
149     response := doc.CreateElement("Response")
150     say := response.CreateElement("Say")
151     say.CreateAttr("voice", common.TwilioVoice)
152     say.CreateAttr("loop", "2")
153     say.SetText(message)
154     doc.Indent(2)
155     twilML, err := doc.WriteToBytes()
156     if err != nil {
157         return nil, err
158     }
159     return twilML, nil
160 }

```

Figure 22: Screenshot of func CreateTwilMLXML within package driver

First, Figure 17 line 42, we call CreateAll (Figure 18) in the driver package. This will return 4 items, an instance of our database, a connection to our Twilio functions within the notification package, an AWS session that can be used to authenticate with different services, and mapKey used for making requests to MapQuest. Then, Figure 17 line 45, we call GetUser (Figure 19) and return an error if the user does not exist or the request to DynamoDB fails. Continuing, Figure 17 line 51 we attempt to update the user's location by calling UpdateLocation (Figure 20), but do not terminate the flow of the program if an error occurs, because this is a secondary functionality of the function. We try to update the user's location as they interact with various API endpoints, but cannot always guarantee the most update to date location.

Then, Figure 17 line 54 we call GetAddress (Figure 21) to translate the user's current location into a human readable street, city, country code and zipcode, this utilizes the MapQuest API. The last two parameters, false and 0, are workarounds for controlling whether or not we should consider alert tier, because this function is called using other Lambda functions which have this requirement. After, Figure 17 line 60, we create a TwilML (Twilio Markup Language) definition that will be passed to Twilio by calling CreateTwilMLXML (Figure 22). This is how Twilio's voice service is programmed by developers. The language is an XML document, that contains some directives for defining what voice will be used, how many times the content will be said, what content will be said.

```

65 // Upload the TwilML to S3 so Twilio can access it
66 callbackURL, err := driver.UploadTwilMLXML(twilML, sess)
67 if err != nil {
68     return driver.ErrorResponse(http.StatusInternalServerError, err), nil
69 }
70
71 // Call all tier 1 contacts
72 for _, contact := range user.Contacts {
73     if contact.Tier == common.FIRST {
74         retry.Do(
75             func() error { return twilio.SendVoiceCall(contact.PhoneNumber, callbackURL) },
76             retry.Attempts(3),
77         )
78     }
79 }
80
81 bodyContent := fmt.Sprintf("Successfully sent emergency call to emergency services and contacts of user %v %v (%v)", u
82 return driver.SuccessfulResponse(bodyContent), nil
83 }

```

Figure 23: Screenshot of lines 66 through 82 of func Handler

```

162 // UploadTwilMLXML uploads the XML generated in CreateTwilMLXML and returns the URL for the object
163 func UploadTwilMLXML(twilML []byte, sess *session.Session) (string, error) {
164     // Create a unique id for the object
165     id, _ := uuid.NewRandom()
166     objectKey := string([]rune(id.String())[0:7]) + ".xml"
167
168     // Upload to S3 bucket
169     _, err := s3.New(sess).PutObject(&s3.PutObjectInput{
170         Bucket:      aws.String(common.S3BucketName),
171         Key:          aws.String(objectKey),
172         ACL:          aws.String("public-read"),
173         Body:         bytes.NewReader(twilML),
174         ContentLength: aws.Int64(int64(len(twilML))),
175         ContentType:  aws.String("text/xml"),
176     })
177     if err != nil {
178         return "", err
179     }
180     return common.S3BucketLocation + objectKey, nil
181 }

```

Figure 24: Screenshot of func UploadTwilMLXML within package driver

To make this document accessible to Twilio's API, we upload it to Amazon's Simple Storage Service (S3) at Figure 23 line 66 by calling UploadTwilMLXML (Figure 24). At the end of func Handler in Figure 23, lines 81 and 82 entail responding to the requesting client with a HTTP 200 OK response and a generic, identifiable message.

## 5.2 Informal Testing

Now that we have written the Lambda function, we can informally test it. To do this we use will use the Serverless Application Model command-line interface (SAM CLI) [3]. Installation and setup details are available in the README.md file of our backend repository. The SAM CLI allows us to test our application in a Docker container, similar to the environment provided by AWS, in the cloud. Instead of deploying the application and later discovering that we have issues, we can catch those issues much quicker by doing the testing and debugging in our local environment. Doing this is as follows.

1. Define a SAM template.yml definition. This YAML definition will define our Lambda function and API Gateway Endpoint.
2. Start Docker
3. Run `make sam` to run a series of commands that build, package, and start the local API
4. Run `make test` to test the endpoint using the sample request defined in `event.json`

```

Resources:
  ESSendEmergencyVoiceCallFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: main.zip
      Handler: main
      Runtime: go1.x
      Tracing: Active
      Events:
        CatchAll:
          Type: Api
          Properties:
            Path: /voice
            Method: POST

```

Figure 25: Screenshot of the local template.yml used for SAM.

```
{
  "user_id": "4a35788a-e3fa-4eb4-b1c2-b9a3be8a58c9",
  "last_known_location": [40.7648, -73.9808]
}
```

Figure 26: Screenshot of the event.json used for locally testing this function.

```
# Test the function locally using the SAM CLI tool
sam:
# Building Go binary
GOOS=linux go build -o main

# Put the binary into a zip folder
${GOBIN}/build-lambda-zip.exe -o main.zip main

# Test
sam local start-api

test:
curl -i -X POST http://127.0.0.1:3000/voice -d @event.json --header "Content-Type: application/json"
```

Figure 27: Screenshot of the Makefile rules used to make testing more convenient.

### 5.3 Formal Testing

Next, now that we're confident, that our function behaves as expected we can write formal unit tests for it. These tests are defined in a functions main\_test.go file. The file defines one function for unit testing the handler, and a testing object. This testing object is apart of the Go standard library and provides many useful tools for writing tests. The tests for our application, although not all have unit tests, follow the table-driven testing (TDD) pattern. Tests are given some expected paramters and are looped over to cover different edge cases. For this test, we only have one test in the table. This test also uses a pre-defined user id, located in a DynamoDB table.

```

run test | debug test
func TestHandler(t *testing.T) {
    successfulResponse := "{\"body\": \"Successfully sent emergency call to emergency services and contacts of user EmergenSeek User (4a35788a-e3fa-4eb4-b1c2-b9a3be8a58c9)\"}"
    tests := []struct {
        name          string
        UserID         string
        LastKnownLocation []float64
        ExpectedBody    string
    }{
        {
            "Simple Request",
            "4a35788a-e3fa-4eb4-b1c2-b9a3be8a58c9",
            []float64{40.7648, -73.9888},
            successfulResponse,
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            t.Logf("Running test: %v", tt.name)
            // Create request body
            r := Request{
                UserID: tt.UserID,
                Location: tt.LastKnownLocation,
            }

            // Seralize request body
            b, _ := json.Marshal(r)

            // Create request event
            request := events.APIGatewayProxyRequest{
                Path: "/voice",
                HTTPMethod: "POST",
                Headers: map[string]string{"Content-Type": "application/json"},
                Body: string(b),
            }

            // Create expected response
            expectedResponse := events.APIGatewayProxyResponse{
                StatusCode: 200,
                Headers: map[string]string{
                    "Content-Type": "application/json",
                },
                Body: tt.ExpectedBody,
            }

            // Invoke handler using request
            response, err := Handler(request)
            assert.Equal(t, response.Headers, expectedResponse.Headers)
            assert.Contains(t, response.Body, expectedResponse.Body)
        })
    }
}

```

Figure 28: Screenshot of the test function located in main\_test.go for testing func Handler of ESSendEmergencyVoiceCall.

## 5.4 Deployment

Continuing, now that we have defined a formal test, we can proceed to pushing our code to the master branch of our repository and begin the deployment. To do this, we update our buildspec.yml and template.yml in the root of the repository. Changes to the template.yml are similar to the initial creation of the file we made for local SAM CLI for testing.

1. Update buildspec pre.build stage with:

```

pre_build:
  commands:
    # Run pre-build checks for ESSendEmergencyVoiceCall
    # Make sure we're in the project directory within our GOPATH
    - cd "/go/src/handler/functions/ESSendEmergencyVoiceCall"

    # Fetch all dependencies
    - go get ./...

    # Ensure code passes all lint tests
    - golint -set_exit_status

    # Check for common Go application problems with 'go tool vet'
    - go tool vet .

    # Run all tests included with our application
    - go test .

    # =====

```

2. Update buildspec build stage with :

```

build:
  commands:
    # Run build ESSendEmergencyVoiceCall
    # Change directory into ESSendEmergencyVoiceCall
    - cd "/go/src/handler/functions/ESSendEmergencyVoiceCall"

    # Build our go application
    - go build -o main

    # =====

```

3. Add new resource and matching route/request method to template.yml:

```

Resources:
  ESSendEmergencyVoiceCall:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: functions/ESSendEmergencyVoiceCall
      Handler: main
      Runtime: go1.x
      Tracing: Active
      Role:
        Fn::GetAtt:
          - LambdaExecutionRole
          - Arn
      Events:
        PostEvent:
          Type: Api
          Properties:
            Path: /voice
            Method: post
  ConfigLambdaPermissionVoice:
    Type: AWS::Lambda::Permission
    DependsOn:
      - ESSendEmergencyVoiceCall
    Properties:
      Action: lambda:InvokeFunction
      FunctionName: !Ref ESSendEmergencyVoiceCall
      Principal: apigateway.amazonaws.com

```



After adding, committing, and pushing to the master branch of our GitHub repository, AWS CodeStar will take over and handle our continuous integration and continuous deployment by using the definitions in the `buildspec.yml` and `template.yml`.

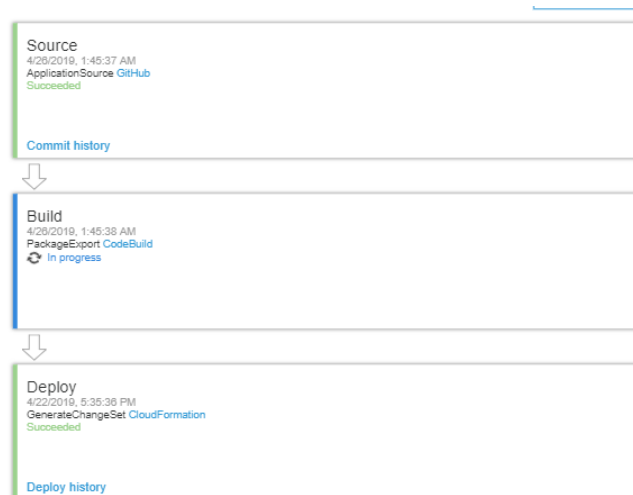


Figure 29: AWS CodeStar screenshot of a build in progress, after a Git Push to our code repository.

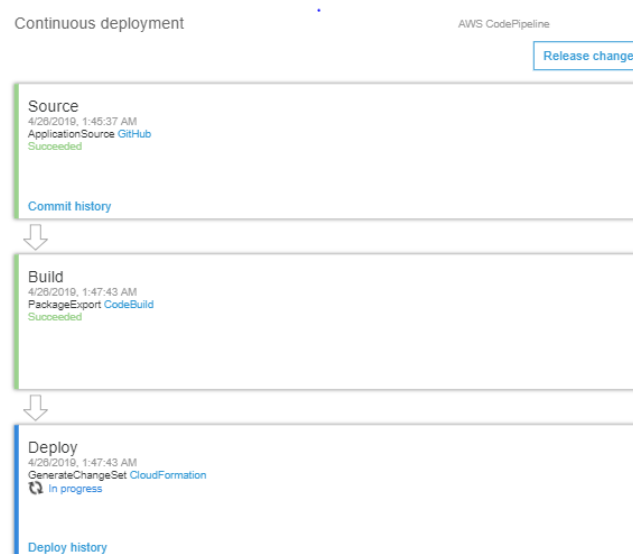


Figure 30: AWS CodeStar screenshot of a deploy in progress, after a successful build.

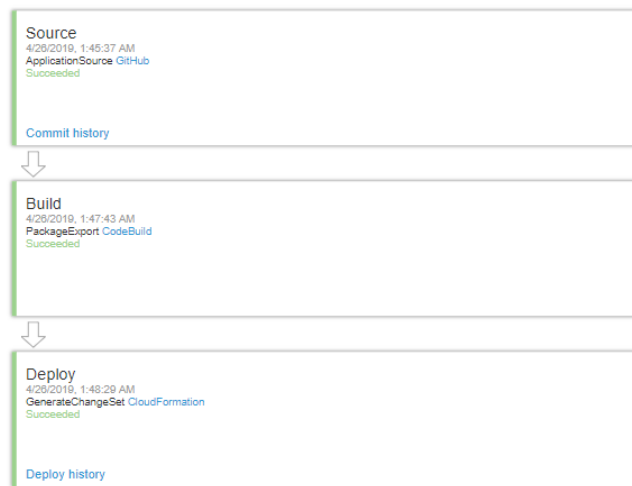


Figure 31: AWS CodeStar screenshot a successful CI/CD pipeline; deploy complete.

## 5.5 Integration

Next, now that the function is deployed, we can begin connecting the frontend. First, we'll need the endpoint at which the function lives. To find this, we can navigate to API Gateway in the AWS console.



Figure 32: AWS API Gateway screenshot of all endpoints we have available for access.



Invoke URL: <https://tzuwifn7ng.execute-api.us-east-2.amazonaws.com/Prod>

Figure 33: AWS API Gateway screenshot of our production stage, Invoke URL

First we create the feature view. This is defined in `sos.dart` within the `screens` folder, encapsulated in the `lib` folder. The feature view is made up of several widgets, each acting as a container, or higher-level parent to perform some function. The `SOSPage` encapsulates the `SOSButton` as its parent. The `SOSButton` encapsulates the `ProgressRing StatefulWidget` which will call the `activateSOS` method. The `ProgressRing` has a method called `showConfirmation` which will allow the user to select their emergency tier or cancel.

```
8 // SOS Screen itself, contains an app bar and the SOS button
9 class SOSPage extends StatelessWidget {
10   @override
11   Widget build(BuildContext context) {
12     return Scaffold(
13       backgroundColor: Colors.blueGrey[400],
14       drawer: NavMenu(),
15       appBar: AppBar(
16         title: Text("S.O.S."),
17         actions: <Widget>[
18           IconButton(
19             icon: Icon(Icons.settings),
20             tooltip: 'Settings',
21             onPressed: () {
22               Navigator.push(context,
23                 MaterialPageRoute(builder: (context) => SettingsSOS()));
24             },
25           ) // IconButton
26         ], // <Widget>[]
27       ), // AppBar
28       body: Center(child: SOSButton()); // Scaffold
29     }
30   }
31 }
```

Figure 34: Screenshot of the `SOSPage` widget's Dart class

```

33 class SOSButton extends StatelessWidget {
34   @override
35   Widget build(BuildContext context) {
36     return ScopedModelDescendant<AppModel>(
37       builder: (context, child, model) => Column(
38         mainAxisAlignment: MainAxisAlignment.center,
39         crossAxisAlignment: CrossAxisAlignment.center,
40         children: <Widget>[
41           model.getSOSStatus() == true
42             ? ShowUp(
43               child: Text("S.O.S. Alert Broadcasted \n ${model.getPoliceNumber()} Has Been Cont
44                 textAlign: TextAlign.center,
45                 style: TextStyle(
46                   fontSize: 25.0, fontWeight: FontWeight.w300)), // TextStyle // Text
47               delay: 500,
48             ) // ShowUp
49             : ShowUp(
50               child: Text("Hold To Activate S.O.S.",
51                 textAlign: TextAlign.center,
52                 style: TextStyle(
53                   fontSize: 25.0, fontWeight: FontWeight.w300)), // TextStyle // Text
54               delay: 500,
55             ), // ShowUp
56           Padding(
57             padding: EdgeInsets.all(32.0),
58           ), // Padding
59           Stack(
60             alignment: Alignment.center,
61             children: <Widget>[
62               RawMaterialButton(
63                 padding: EdgeInsets.all(15.0),
64                 elevation: 10.0,
65                 shape: new CircleBorder(),
66                 fillColor: model.getSOSStatus() == false
67                   ? Colors.white
68                   : Colors.black,
69               child: new Icon(
70                 Icons.error_outline,
71                 color: model.getSOSStatus() == false
72                   ? Colors.blue[200]
73                   : Colors.red,
74                 size: 150.0,
75               ), // Icon
76               onPressed: () {}), // RawMaterialButton
77             ], // <Widget>[]
78           ), // Stack
79         ], // <Widget>[]
80       )); // Column // ScopedModelDescendant

```

Figure 35: Screenshot of the SOSButton widget's Dart class

```

void showConfirmation() {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return ScopedModelDescendant<AppModel>(
        builder: (context, child, model) => AlertDialog(
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.all(Radius.circular(20.0))), // RoundedRectangleBorder
          contentPadding: EdgeInsets.all(17.0),
          content: Column(
            mainAxisAlignment: MainAxisAlignment.min,
            children: <Widget>[
              new Text(
                "Please select the alert severity",
                textAlign: TextAlign.center,
                style: TextStyle(fontSize: 25.0),
              ), // Text
              Padding(
                padding: EdgeInsets.fromLTRB(0.0, 0.0, 0.0, 12.0),
              ), // Padding
              Row(
                mainAxisAlignment: MainAxisAlignment.spaceBetween,
                children: <Widget>[
                  new FlatButton(
                    onPressed: () {
                      model.setEmergencyTier("MILD");
                      model.activateSOS();
                      resetProgress();
                      Navigator.of(context).pop();
                    },
                    child: new Text(
                      "Mild",
                      style: TextStyle(
                        fontSize: 16.0,
                        color: Colors.tealAccent[400]), // TextStyle
                    ), // Text // FlatButton
                  new FlatButton(
                    onPressed: () {
                      model.setEmergencyTier("SEVERE");
                      model.activateSOS();
                      resetProgress();
                      Navigator.of(context).pop();
                    },
                    child: new Text(
                      "Severe",
                      style: TextStyle(
                        fontSize: 16.0, color: Colors.red), // TextStyle
                    ), // Text // FlatButton
                  new FlatButton(

```

Figure 36: Screenshot of the showConfirmation method of ProgressRing StatefulWidget.

The activateSOS method is the beginning of the model implementation. The model will encapsulate calls to a service method that will invoke the Lambda function directly

```

54 Future activateSOS() async {
55   _sosActive = true;
56   List coordinates = await pollCurrentLocation();
57
58   // Retrieve contact info for police using location
59   var emergencyInfo = await getEmergencyInfo(coordinates);
60   _policeNumber = emergencyInfo.police;
61
62   // Display local notification
63   _notifications.displaySOSNotification();
64
65   // Send API requests according to severity
66   if(_emergencyTier == EmergencyTier.SEVERE && _sendCalls == true){
67     sendCall(coordinates, _userId);
68   }
69   if(_sendTexts == true){
70     _emergencyTier == EmergencyTier.SEVERE ? sendSMS(coordinates, 1, _userId) : sendSMS(coordinates, 2, _
71   }
72
73   // Display personal info as a notification if user has enabled it
74   if(_showLockscreenInfo == true){
75     displayInfo();
76   }
77   // Inform [Model] to rebuild dependent widgets
78   notifyListeners();
79 }

```

Figure 37: Screenshot of the activateSOS method within the SOSModel class, a scoped model.

The method which invokes the Lambda function directly is implemented as callSOS. Note that the URL and path is the same as our Invoke URL and the path the we defined for the Lambda function. Also note the `http.post` call on the url. This is important because we defined that the application should only accept POST requests.

```

76 Future<Post> sendCall(List coordinates, String uid) async {
77   var url = "https://tzuvifn7ng.execute-api.us-east-2.amazonaws.com/Prod/voice";
78   Map<String, String> headers = {
79     'Content-Type': 'application/json',
80   };
81   var body = jsonEncode({
82     "user_id": uid,
83     "last_known_location": coordinates
84   });
85
86   final response = await http.post(url, headers: headers, body: body);
87   print(json.decode(response.body));
88
89   if (response.statusCode == 200) {
90     // If the call to the server was successful, parse the JSON
91     return Post.fromJson(json.decode(response.body));
92   } else {
93     // If the call was not successful, notify user of error code
94     print("sendCall request failed. Error Code: ${response.statusCode}");
95   }
96 }

```

Figure 38: sendCall service function within the api.dart file of the services package

## 6 Conclusion

### 6.1 Implementation Challenges

#### 6.1.1 Lambda Maximum Compute Time Limitation

Throughout the development of the project, the only limitation that we have found as a result of using Lambda comes from the maximum compute time. Because Lambda instances are derived on a per-request basis (customers are charged for the amount of time their functions are running), there is a maximum cap of 15-minutes at which a single Lambda function can run for a single request. Because of this we have not been able to implement ESPollLoop, which will give our application the ability to send notifications on a periodic, repeating basis.

#### 6.1.2 Identity Access and Management

Similarly to our partial demo and implementation, one of the largest implementation issues we have had thus far is the integration of AWS CloudFormation for automatically provisioning the resources necessary for creating each Lambda function and their respective API Gateway route. Overall, assigning the correct permissions for the service workers responsible for creating these resources is the biggest issue. We currently still need to correct this issue.

### 6.2 Group Member Contributions

## 7 Glossary

- Cross-platform application - software which has a single implementation, but may be executed on different distributions. (i.e. one code base, running on many different operating systems or processor architectures.)
- Serverless application repository - single responsibility application definitions which act as independent entities. The application repository as a whole defines the API for our mobile application. (i.e. each serverless application is a Lambda function, and the lambda function is responsible for a system feature.)
- Application Programming Interface (API) - Definitions and communication protocols used for building the structure of software.
- HyperText Transfer Protocol (HTTP) - An application protocol used heavily for websites and services throughout the Internet.
- REpresentational State Transfer (REST) - A architectural style for APIs that define some specifications for creating web-based services.

## References

- [1] scoped\_model 1.0.1 — Flutter Package — [https://pub.dartlang.org/packages/scoped\\_model](https://pub.dartlang.org/packages/scoped_model)
- [2] Mattihas Jung, Sascha Möllering, Peter Dalbhanjan, Peter Chapman, Christoph Kassen  
*Microservices on AWS; AWS Whitepaper.*  
<https://docs.aws.amazon.com/aws-technical-content/latest/microservices-on-aws/microservices-on-aws.pdf>
- [3] What is the AWS Serverless Application Model (AWS SAM)? — <https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html>

## List of Figures

1	Updated use-case diagram for the EmergenSeek application. . . . .	5
2	Full class diagram for the frontend. . . . .	6
3	View UML class diagram for the frontend. . . . .	7

4	Service UML class diagram for the frontend. . . . .	8
5	Model UML class diagram for the frontend. . . . .	9
6	Class diagram for the EmergenSeek backend. . . . .	10
7	Updated Cloudcraft diagram of the AWS resources, external APIs encapsulated by the backend, and frontend connection abstractions. . . . .	11
8	Screenshot of the functions folder and the contents of the common folder in the backend repository. . . . .	14
9	Screenshot of the contents of the functions folder in the backend repository. . . . .	14
10	Screenshot of the contents of the ESSendEmergencyVoiceCall repository within the functions folder. . . . .	14
11	Screenshot of <code>func main</code> within <code>package main</code> . . . . .	15
12	Screenshot of imports defined for the function. Their accessors are either defined as an alias to the left of the import name (like <code>retry</code> ), or as the package name defined by the dependency. Standard library packages are defined above external, third-party dependencies. . . . .	15
13	Screenshot of the first few lines of the <code>Handler</code> definition . . . . .	15
14	Screenshot of the definition for <code>verifyRequest</code> . . . . .	15
15	Screenshot of <code>Request</code> type. . . . .	16
16	Screenshot of <code>Validate</code> , pointer-receiver method bound to the <code>Request</code> type. . . . .	16
17	Screenshot of lines 42 through 63 of <code>func Handler</code> . . . . .	16
18	Screenshot of <code>func CreateAll</code> within <code>package driver</code> . . . . .	17
19	Screenshot of <code>func GetUser</code> within <code>package database</code> . . . . .	17
20	Screenshot of <code>func UpdateLocation</code> within <code>package database</code> . . . . .	18
21	Screenshot of <code>func GetAddress</code> within <code>package driver</code> . . . . .	18
22	Screenshot of <code>func CreateTwilMLXML</code> within <code>package driver</code> . . . . .	19
23	Screenshot of lines 66 through 82 of <code>func Handler</code> . . . . .	19
24	Screenshot of <code>func UploadTwilMLXML</code> within <code>package driver</code> . . . . .	20
25	Screenshot of the local <code>template.yml</code> used for SAM. . . . .	20
26	Screenshot of the <code>event.json</code> used for locally testing this function. . . . .	21
27	Screenshot of the Makefile rules used to make testing more convenient. . . . .	21
28	Screenshot of the test function located in <code>main_test.go</code> for testing <code>func Handler</code> of <code>ES-SendEmergencyVoiceCall</code> . . . . .	22
29	AWS CodeStar screenshot of a build in progress, after a Git Push to our code repository. . . . .	24
30	AWS CodeStar screenshot of a deploy in progress, after a successful build. . . . .	24
31	AWS CodeStar screenshot a successful CI/CD pipeline; deploy complete. . . . .	25
32	AWS API Gateway screenshot of all endpoints we have available for access. . . . .	25
33	AWS API Gateway screenshot of our production stage, Invoke URL . . . . .	26
34	Screenshot of the <code>SOSPage</code> widget's Dart class . . . . .	26
35	Screenshot of the <code>SOSButton</code> widget's Dart class . . . . .	27
36	Screenshot of screenshot of <code>showConfirmation</code> method of <code>ProgressRing StatefulWidget</code> . . . . .	28
37	<code>sendCall</code> service function within the <code>api.dart</code> file of the <code>services</code> package . . . . .	29