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[1]. 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

[1]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

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 sucessfully 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

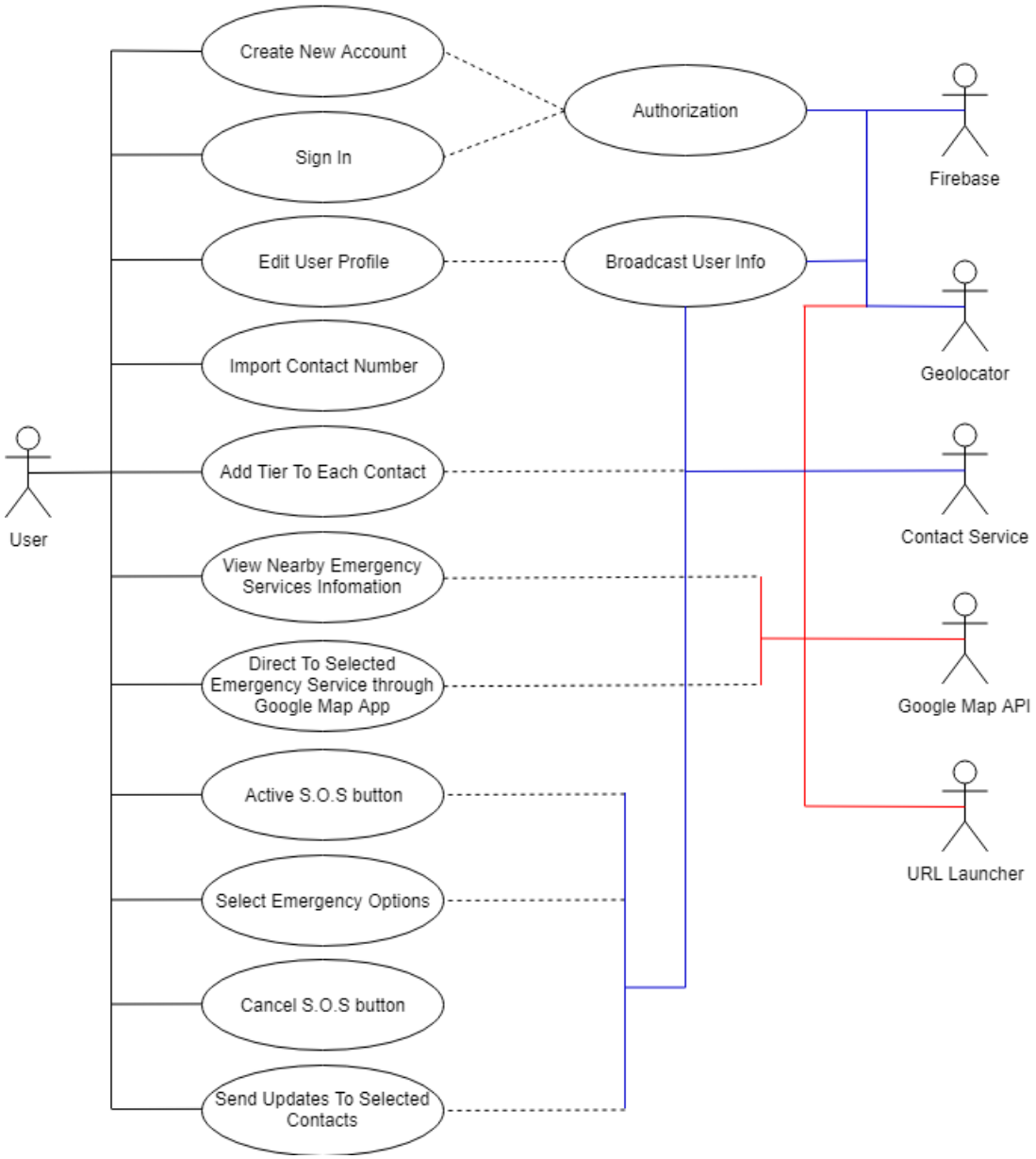

Figure 1: Updated use-case diagram for the EmergenSeek application.

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.

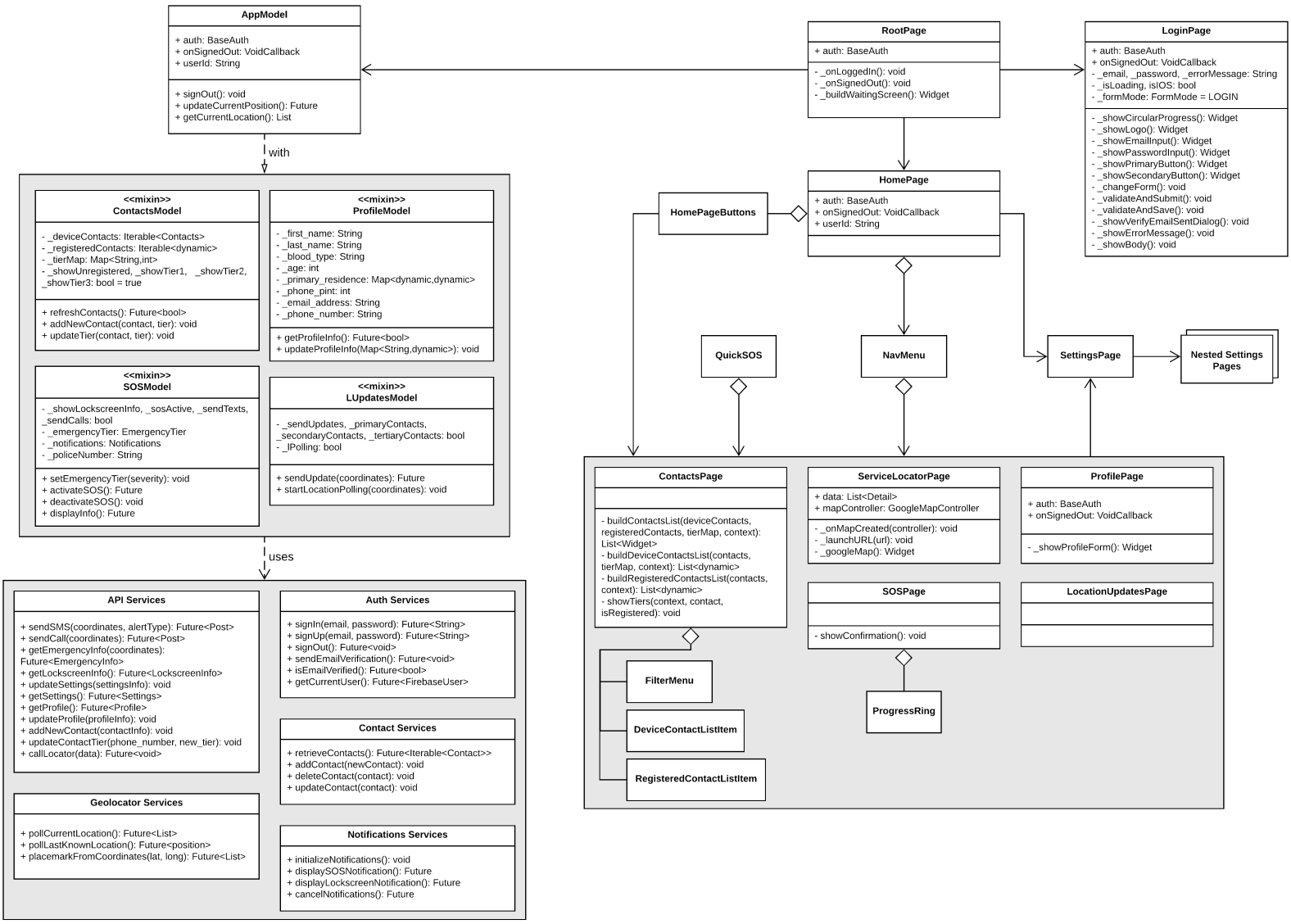

Figure 2: Full class diagram for the frontend.

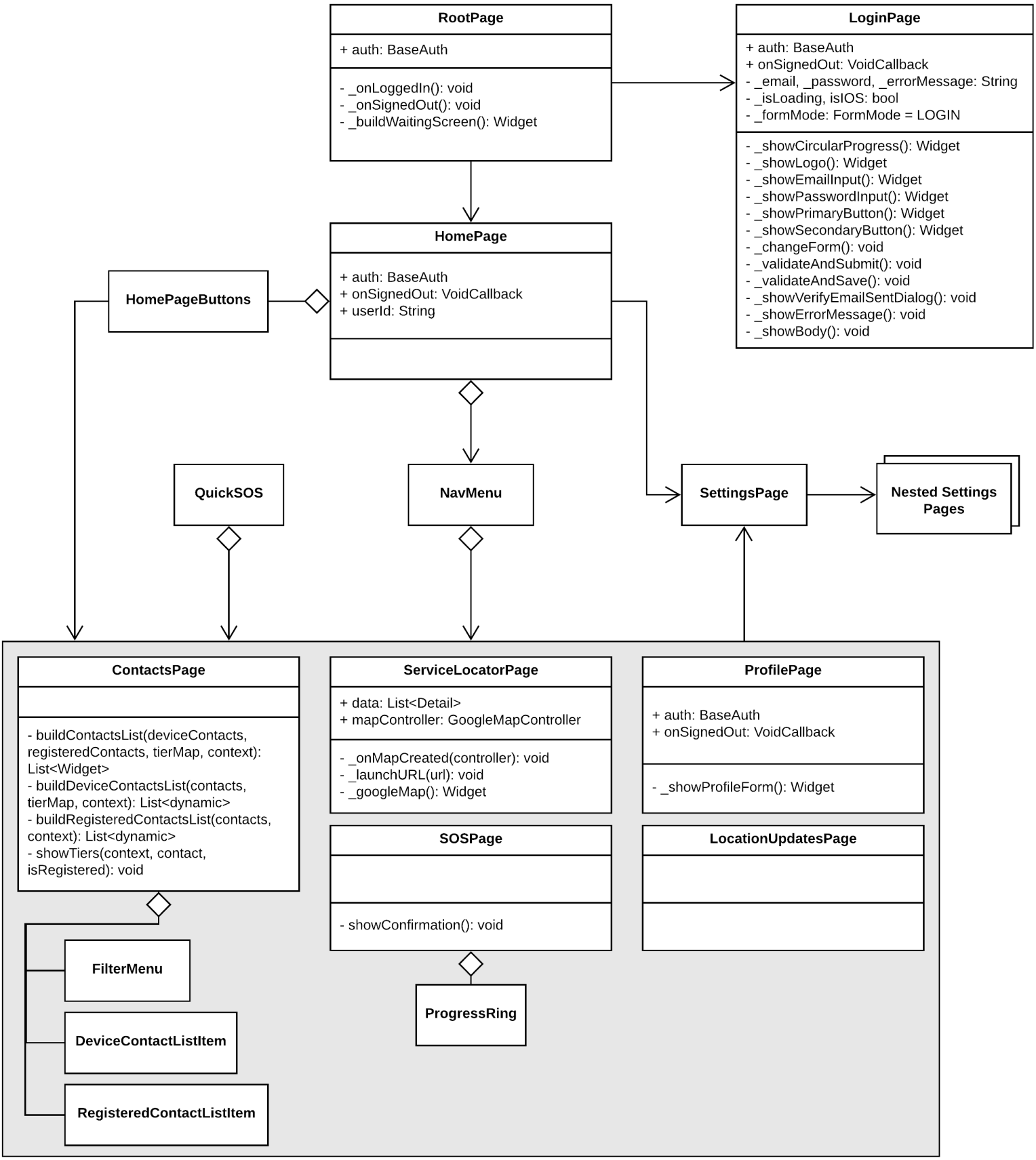

Figure 3: View UML class diagram for the frontend.

The application's views begins via a RootPage, this root page can be thought of a session manager keeping track of the user though 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.

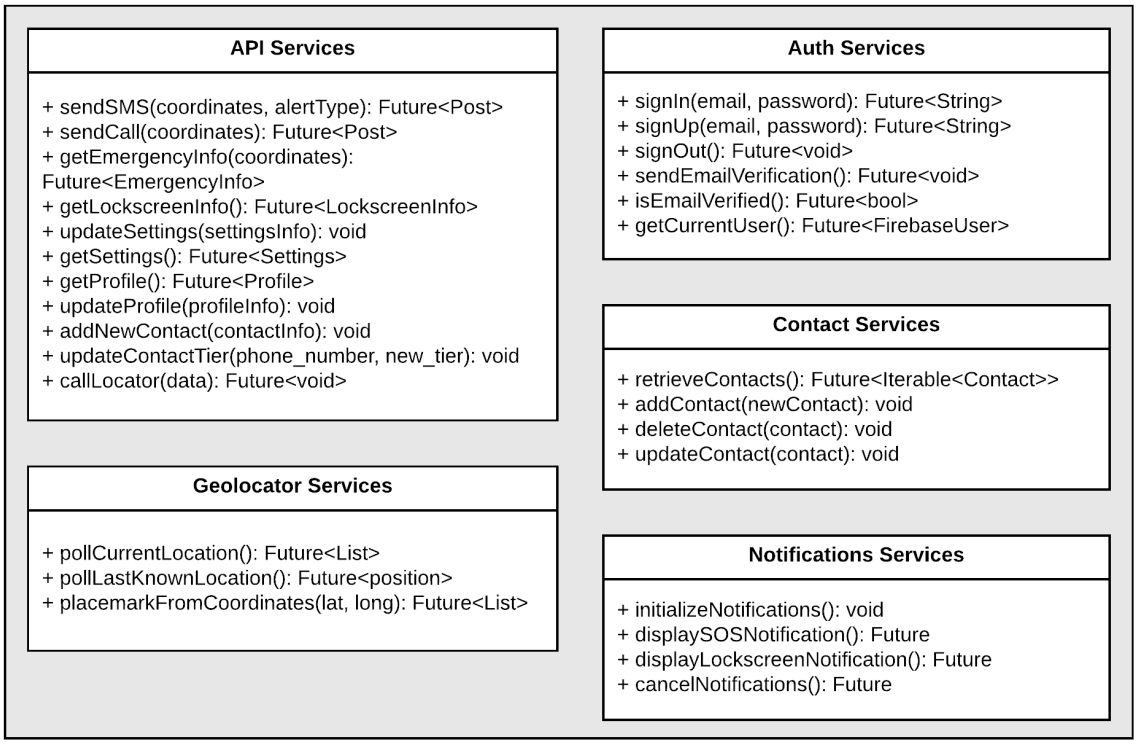

Figure 4: Service UML class diagram for the frontend.

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 clas 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 notifcations on the device, both within the notifications center and on the user's lock screen.

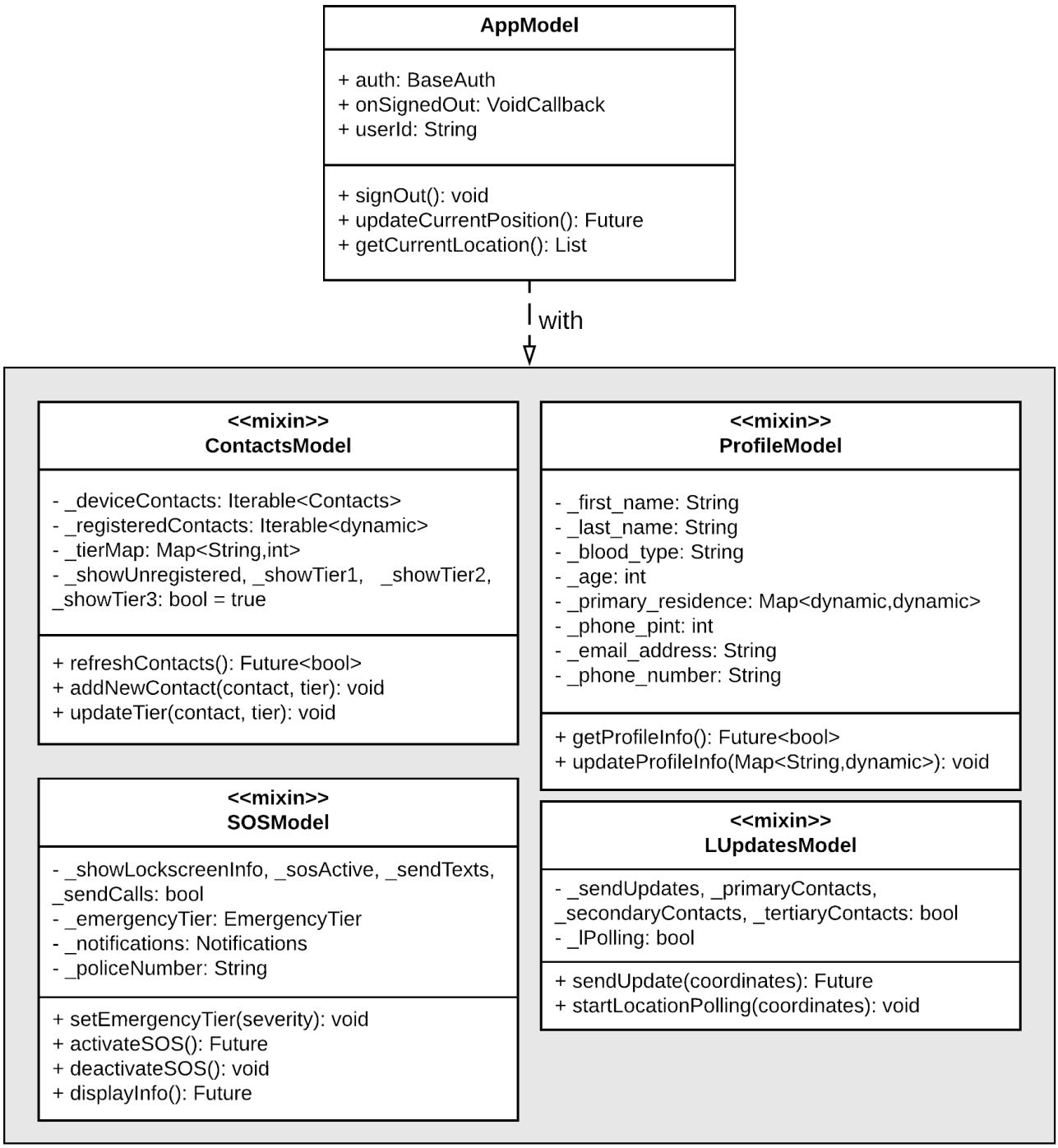

Figure 5: Model UML class diagram for the frontend.

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

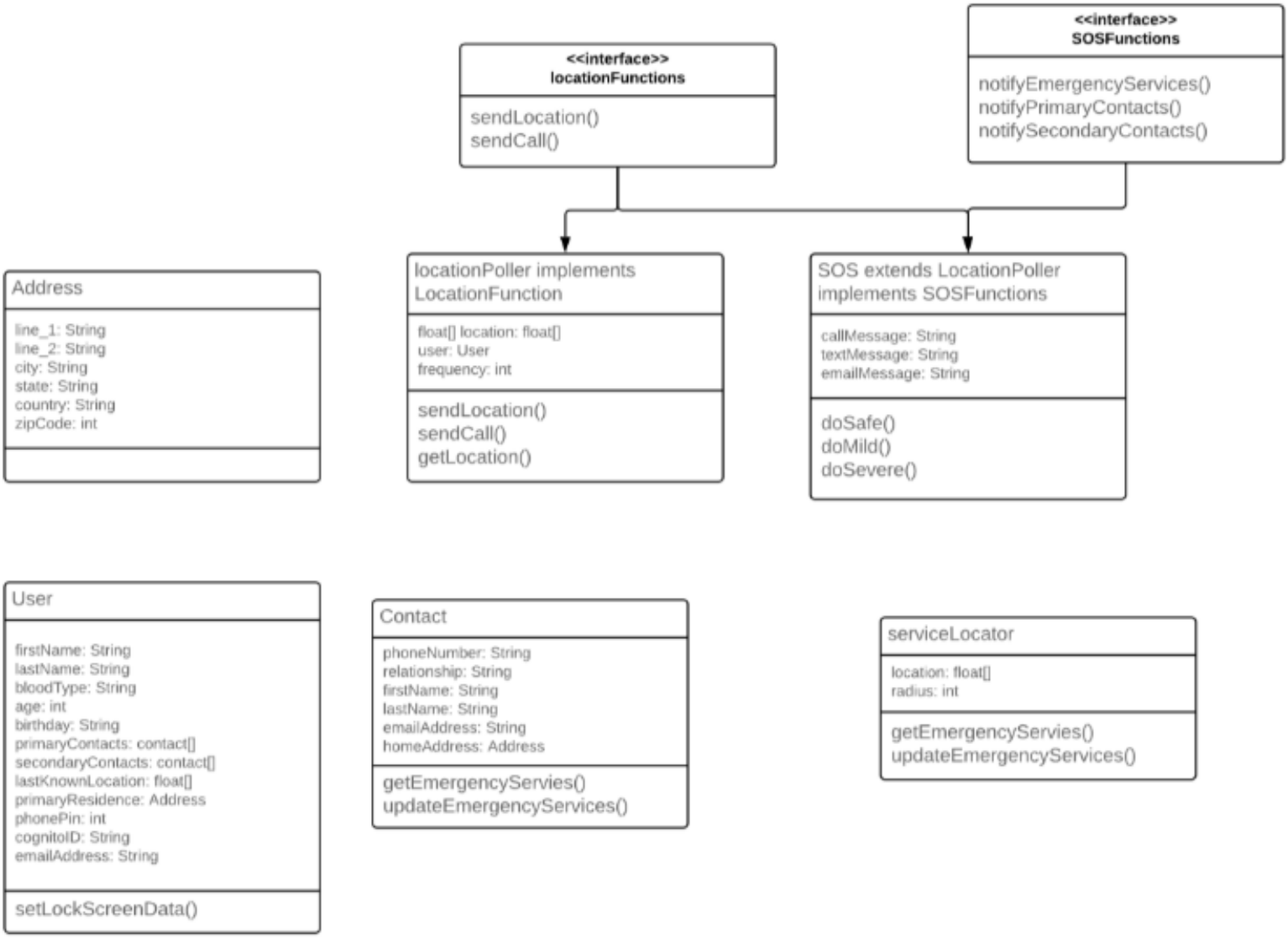

Figure 6: Class diagram for the EmergenSeek backend.

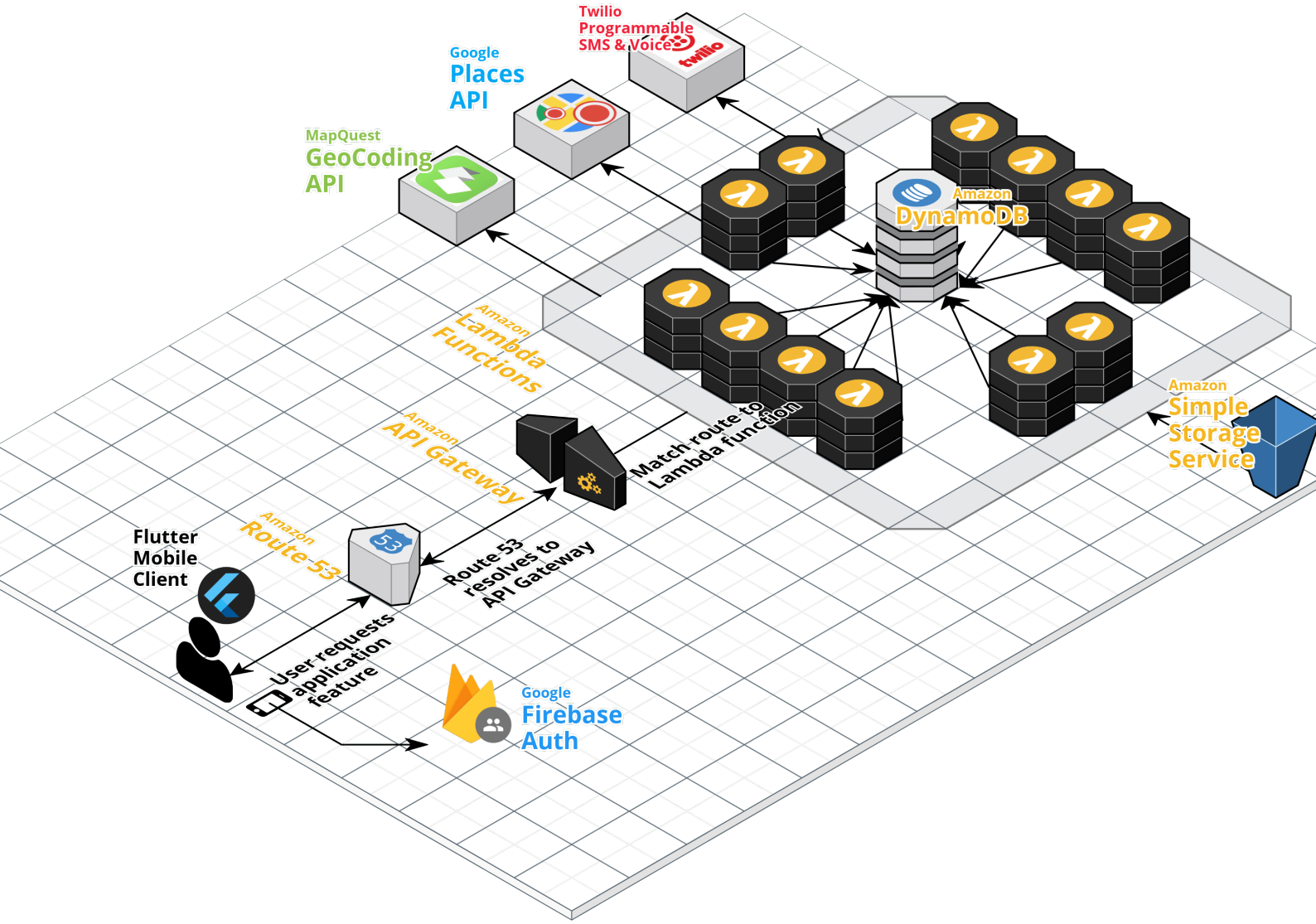

Figure 7: Updated Cloudcraft diagram of the AWS resources, external APIs encapsulated by the backend, and frontend connection abstractions.

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 clas 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 manged 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.

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)

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

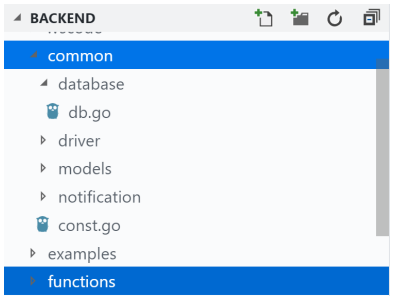

Figure 8: Screenshot of the functions folder and the contents of the common folder in the backend repository.

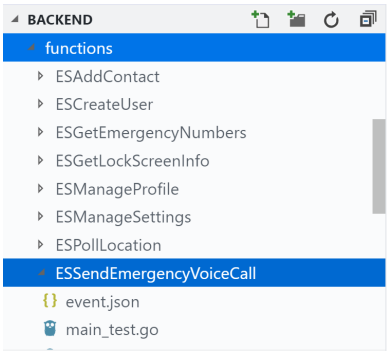

Figure 9: Screenshot of the contents of the functions folder in the backend repository.

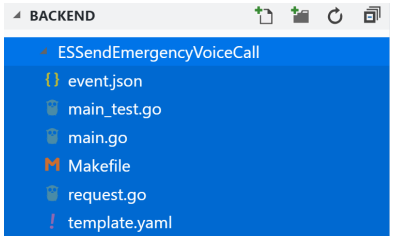

Figure 10: Screenshot of the contents of the ESSendEmergencyVoiceCall repository within the functions folder.

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" 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.

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

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

```go
import (
    "encoding/json"
    "fmt"
    "net/http"

    retry "github.com/avast/retry-go"
    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
    "github.com/emergenseek/backend/common"
    "github.com/emergenseek/backend/common/driver"
)
```

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.

```go
// Handler is the Lambda handler for ESSendEmergencyVoiceCall
func Handler(request events.APIGatewayProxyRequest) (events.APIGatewayProxyResponse, error) {
    // Verify the request
    req, status, err := verifyRequest(request)
    if err != nil {
        return driver.ErrorResponse(status, err), nil
    }

```

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

```go
func verifyRequest(request events.APIGatewayProxyRequest) (*Request, int, error) {
    // Create a new request object and unmarshal the request body into it
    req := new(Request)
    err := json.Unmarshal([]byte(request.Body), req)
    if err != nil {
        return nil, http.StatusUnprocessableEntity, err
    }

    // Make sure all of the necessary parameters are present
    err = req.Validate()
    if err != nil {
        return nil, http.StatusBadRequest, err

    }
    // All checks passed, return req struct for use. http.StatusOK is ignored
    return req, http.StatusOK, nil
}
```

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.

```go
// Request defines the expected body parameters of ESSendEmergencyVoiceCall
type Request struct {
    // The ID of the user making the request
    UserID string `json:"user_id"`

    // When requests are made to this Lambda function,
    // it is assumed that the emergency type is 1 (SEVERE)
    // Type common.EmergencyType `json:"type"`

    // The user's location at the time of the request
    Location []float64 `json:"last_known_location"`
}
```

Figure 15: Screenshot of Request type.

```go
// Validate validates a request to the ESSendSMSNotification Lambda function
func (r *Request) Validate() error {
    // Check if the UserID is present
    if r.UserID == "" {
        return fmt.Errorf("user_id field is required")
    }

    // Check if the Location is present
    if r.Location == nil {
        return fmt.Errorf("last_known_location field is required")
    }

    return nil
}
```

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 fix figures.

```go
    // Initialize drivers
    db, twilio, sess, mapKey := driver.CreateAll()

    // Retrieve user from database
    user, err := db.GetUser(req.UserID)
    if err != nil {
        return driver.ErrorResponse(http.StatusBadRequest, err), nil
    }

    // Update the user's last known location
    err = db.UpdateLocation(user.UserID, req.Location)

    // Convert the user's last known location to a human readable address
    address, err := driver.GetAddress(req.Location, mapKey, false, 0)
    if err != nil {
        return driver.ErrorResponse(http.StatusInternalServerError, err), nil
    }

    // Create the Twilio Markdown Language necessary for the voice call
    twilML, err := driver.CreateTwilMLXML(user, address)
    if err != nil {
        return driver.ErrorResponse(http.StatusInternalServerError, err), nil
    }
```

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

```go
// CreateAll initializes the necessary API providers for Lambda handlers
func CreateAll() (*database.DynamoConn, *notification.TwilioHandler, *session.Session, string) {
    // Create a shared session
    sess := session.Must(session.NewSession(&aws.Config{Region: aws.String(common.Region)}))

    // Initialize database
    db := &database.DynamoConn{Region: common.Region}
    err := db.Create(sess)
    if err != nil {
        panic(err)
    }

    // Get MapQuest credentials
    mapsKey := db.MustGetMapsKey()

    // Get Twilio client credentials using database
    twilio := &notification.TwilioHandler{}
    err = twilio.GetCredentials(db)
    if err != nil {
        panic(err)
    }
    // Authenticate using credentials
    err = twilio.Authenticate()
    if err != nil {
        panic(err)
    }
```

Figure 18: Screenshot of `func CreateAll` within `package driver`

```go
// GetUser will retrieve a user from the database
func (d *DynamoConn) GetUser(uid string) (*models.User, error) {
    // Create user struct to be searched for using provided uid
    userKey := &models.User{
        UserID: uid,
    }
    key, err := dynamodbattribute.MarshalMap(userKey)
    if err != nil {
        return nil, err
    }
    input := &dynamodb.GetItemInput{
        Key:       key,
        TableName: aws.String(common.UsersTableName),
    }

    // Search for user matching uid in table
    result, err := d.Client.GetItem(input)
    if err != nil {
        return nil, err
    }

    // Unmarshal user into struct
    user := &models.User{}
    err = dynamodbattribute.UnmarshalMap(result.Item, &user)
    if err != nil {
        return nil, err
    }

    // Cheap check for item not found
    if user.FirstName == "" {
        return nil, errors.New("user not found")
    }

    return user, nil
```

Figure 19: Screenshot of `func GetUser` within `package database`

```go
// UpdateLocation updates the location of a user when a location poll is invoked
func (d *DynamoConn) UpdateLocation(userID string, location []float64) error {
    var LocationUpdate struct {
        LastKnownLocation []float64 `json:":l"`
    }

    // Marshal the update expression struct for DynamoDB
    LocationUpdate.LastKnownLocation = location
    expr, err := dynamodbattribute.MarshalMap(LocationUpdate)
    if err != nil {
        return err

    }

    // Define table schema's key
    key := map[string]*dynamodb.AttributeValue{
        "user_id": {
            S: aws.String(userID),
        },
    }

    // Use marshalled map for UpdateItemInput
    item := &dynamodb.UpdateItemInput{
        ExpressionAttributeValues: expr,
        TableName:                 aws.String(common.UsersTableName),
        Key:                       key,
        ReturnValues:              aws.String("UPDATED_NEW"),
        UpdateExpression:          aws.String("set last_known_location = :l"),
    }

    _, err = d.Client.UpdateItem(item)
    if err != nil {
        return err
    }
```

Figure 20: Screenshot of `func UpdateLocation` within `package database`

```go
func GetAddress(latlng []float64, key string, useTier bool, tier common.AlertTier) (string, error) {
    // Retrieve location from MapQuest Geocoding API
    geocoder.SetAPIKey(key)
    a, err := geocoder.ReverseGeocode(latlng[0], latlng[1])
    if err != nil {
        return "", err
    }
    // Filter data returned depending on provided AlertTier
    var address string
    if a.Street != "" && tier == common.FIRST {
        address = fmt.Sprintf("%v, ", a.Street)
    }

    if a.City != "" && (tier == common.FIRST || tier == common.SECOND) {
        address = address + fmt.Sprintf("%v, ", a.City)
    }
    if a.State != "" {
        address = address + fmt.Sprintf("%v, ", a.State)
    }
    if a.PostalCode != "" {
        address = address + fmt.Sprintf("%v, ", a.PostalCode)
    }
    if a.CountryCode != "" {
        address = address + fmt.Sprintf("%v", a.CountryCode)
    }

    return address, nil
}
```

Figure 21: Screenshot of `func GetAddress` within `package driver`

```go
// CreateTwilMLXML creates the XML necessary for the gotwilio.NewCallbackParameters invocation in notification.SendVoiceCal
func CreateTwilMLXML(user *models.User, lastLocation string) ([]byte, error) {
    // Split phone number so Twilio voice doesn't read it numerically
    splitPhoneNumber := fmt.Sprintf("%v", strings.Split(user.PhoneNumber, ""))

    // Create message using address and user's information
    name := user.FormattedName()
    message := fmt.Sprintf("This is an automated emergency call from EmergenSeek on behalf of %v. ", name)
    message = message + fmt.Sprintf("They are in need of emergency assistance. Please send help to %v. ", lastLocation)
    message = message + fmt.Sprintf("Please attempt to call %v at %v. Thank you.", name, splitPhoneNumber)

    // Create, format, and return the XML document
    doc := etree.NewDocument()
    doc.CreateProcInst("xml", `version="1.0" encoding="UTF-8"`)
    response := doc.CreateElement("Response")
    say := response.CreateElement("Say")
    say.CreateAttr("voice", common.TwilioVoice)
    say.CreateAttr("loop", "2")
    say.SetText(message)
    doc.Indent(2)
    twilML, err := doc.WriteToBytes()
    if err != nil {
        return nil, err
    }
    return twilML, nil
}
```

Figure 22: Screenshot of `func CreateTwilMLXML` within `package driver`

First, 17 line 45, we call CreateAll 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.

6 Conclusion

6.1 Implementation Challenges

6.1.1 Lambda Compute Time

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, per-nanosecond 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.

6.1.2 Identity Access and Management

Text

6.1.3 CloudWatch

Text

6.2 Group Member Contributions

Text

6.3 Closing

Text

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

[2] 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