Cornell Chatbot

Developer Guide

May 2019

Revision Sheet

Release No.	Date	Revision Description
Rev. 0	04/15/2019	First Draft
Rev. 1	05/10/2019	Adding Content
Rev. 2	05/16/2019	Final Version

Developer Guide
Authorization Memorandum

I have carefully assessed the Developer Guide for The Cornell Chatbot system. This document has been completed in accordance with the requirements of System Development Methodology.

MANAGEMENT CERTIFICATION - Please check the appropriate statement.

_____ The document is accepted.

_____ The document is accepted pending the changes noted.

_____ The document is not accepted.

We fully accept the changes as needed improvements and authorize initiation of work to proceed. Based on our authority and judgment, the continued operation of this system is authorized.

_____	_____
NAME	DATE
Project Academic Supervisor	
_____	_____
NAME	DATE
Project Client	
Wanming Hu	05/16/2019
NAME	DATE
Program Development Team Representative	

Developer Guide

TABLE OF CONTENTS

1.0 GENERAL INFORMATION

1.1 System Overview

An interactive chatbot mobile application to answer Cornell Orientation and other questions:

- Users send questions in text or audio.
- Questions include weather, food, health, study, etc.
- To help first-year Cornell students adapt their college life more quickly and smoothly.

Tool/Environment:

- AWS Service:
    - Cloud formation, API Gateway, Lambda function, DynamoDB
- Language:
    - Swift, Python
- Version Control:
    - Bitbucket
- System Categories:
    - Front-End:
        - Designing interactive interface on iOS platforms, which allows capturing text/ voice questions and reply to the user-side inquiries.
    - Back-End
        - Interfacing with Amazon Lex to process text and audio messages, covering the most frequent questions during orientation events.
        - Implementing a dynamic database with AWS Lambda to collect questions and store answers.
- Operational Status:
    - Partially Operational
    - Under Development
    - Need final reviews from the client (IT @ Cornell)

1.2 Project References

References that were used in the preparation of this document in order of importance to the end user.

Swift

- API calls using Alamofire:
  - https://www.raywenderlich.com/35-alamofire-tutorial-getting-started
- Map API connections:
  - https://developers.google.com/maps/documentation/ios-sdk/intro
- Voice recording & audio playing:
  - https://www.hackingwithswift.com/example-code/media/how-to-record-audio-using-avaudiorecorder
  - https://stackoverflow.com/questions/32036146/how-to-play-a-sound-using-swift
- Swift Voice recognition (possible future work: transcribe audio to text):
  - https://www.raywenderlich.com/2422-building-an-ios-app-like-siri
  - https://medium.com/ios-os-x-development/speech-recognition-with-swift-in-ios-10-50d5f4e59c48

DynamoDB

- Create through templates (AWS CloudFormation):
  - https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-dynamodb-table.html
- Create via Boto3:
  - https://boto3.amazonaws.com/v1/documentation/api/latest/guide/dynamodb.html
- DynamoDB Developer Guide:
  - https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

Amazon Lex

- Getting Start with Amazon Lex:
  - https://aws.amazon.com/lex/getting-started/
- Amazon Lex Developer Guide:
  - https://docs.aws.amazon.com/lex/latest/dg/lex-dg.pdf

API

- Use Generated iOS SDK (Swift) to Call API:
    - Step 1: Generate the iOS SDK of an API using API Gateway Console:
        - https://docs.aws.amazon.com/apigateway/latest/developerguide/generate-ios-sdk-of-an-api.html
    - Step 2: Install AWS Mobile SDK and API Gateway-Generated SDK in a Swift Project
        - https://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-generate-sdk-ios-swift.html
- Build Serverless API:
    - Step 1: Declare serverless resources as provided below:
        - https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-template.html
    - Step 2: Write AWS lambda functions that can be used to “post” or “get” by clients.
    - Step 3: Add lambda functions’ paths to API’s “path” attribute to finish registration of API functions.

1.3 Authorized Use Permission

- Declare API Usage Plan
    - Step 1: Declare Usage Plan in YML for user API keys’ registration as below:
        - https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-usageplan.html
- Register users in run time
    - Step 1: Write a lambda function that creates an API key when a new qualified user accesses the API. The following documentation illustrates how:
        - https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/apigateway.html
    - Step 2: Write Lambda functions that registers the API key to Usage Plan.
    - Step 3: Declare lambda functions above in YML template under API resource so that users have access to those functions via API Gateway.
- Add “security” property to all other functions under serverless resources. This property sets the API key requirement to True and requires users to present their API keys when they try to access those functions.

2.0 SYSTEM SUMMARY

2.1 Lex Bot Configuration - Amazon Lex console

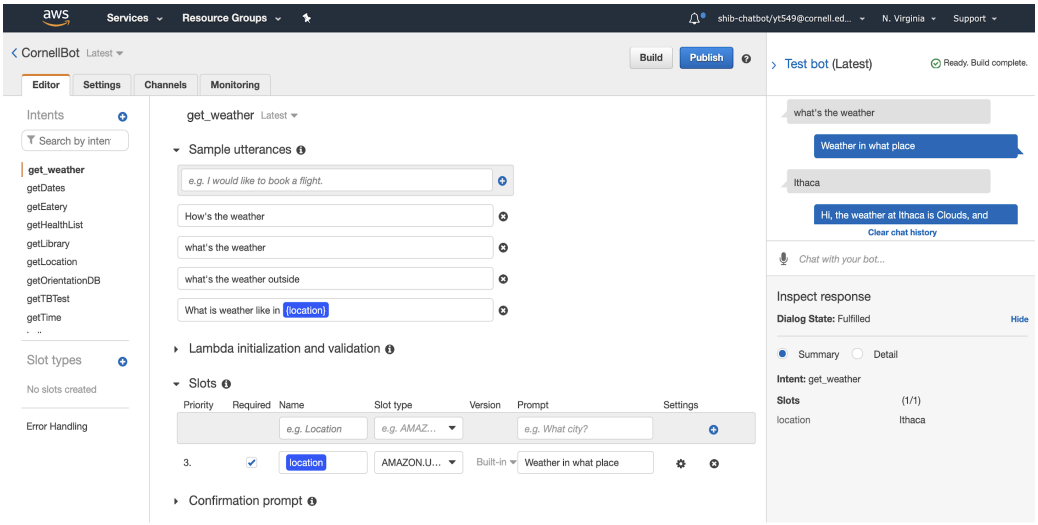
Introduction

Any developer can easily build a conversational environment with Amazon Lex console. Deep learning is subtly built into the Lex console and you only need to specify the basic conversation flow. Furthermore, you can add this console interface on web applications like Facebook Messenger easily.

How to build and test in the Amazon Lex console

In this case, we take the intent of getWeather() in CornellBot as an example. The functionality is realized in the backend of the lambda function, “cornell - chatbot - dev - GetWeather”. The console is pre-configured as follows:

1. **Intent** – get_weather
2. **Slot types** – One custom slot type called locations
3. **Slots** – The intent requires the following information (that is, slots) before the bot can fulfill the intent.
    a. location (AMAZON.US_CITY built-in type)
4. **Utterance** – The below sample utterances show the user's intent, which is real-world language use. This is an important approach to train the Cornell chatbot.
    a. "How's the weather"
    b. “What's the weather”
    c. “What's the weather outside”
    d. “What is the weather like in {location}”
5. **Prompts** – After the Cornell chatbot recognized the intent, it uses the example prompts to fill the slots:
    a. Prompt for the location slot – "Weather in what place?"



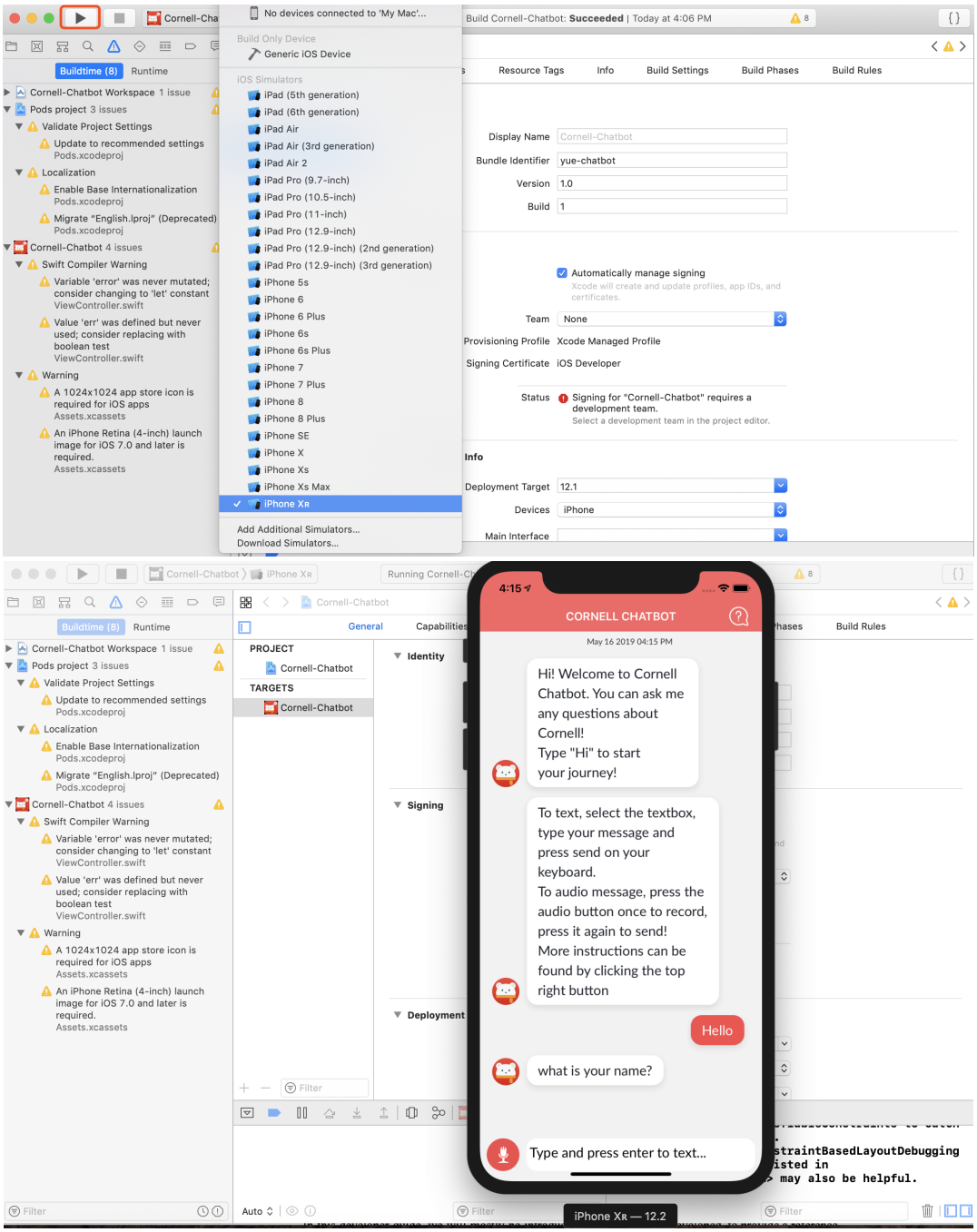
Addition

Please notice that if you need to get access to this “CornellBot” on AWS Lex Console, you must get authorization from IT@Cornell.

3.0 GETTING STARTED

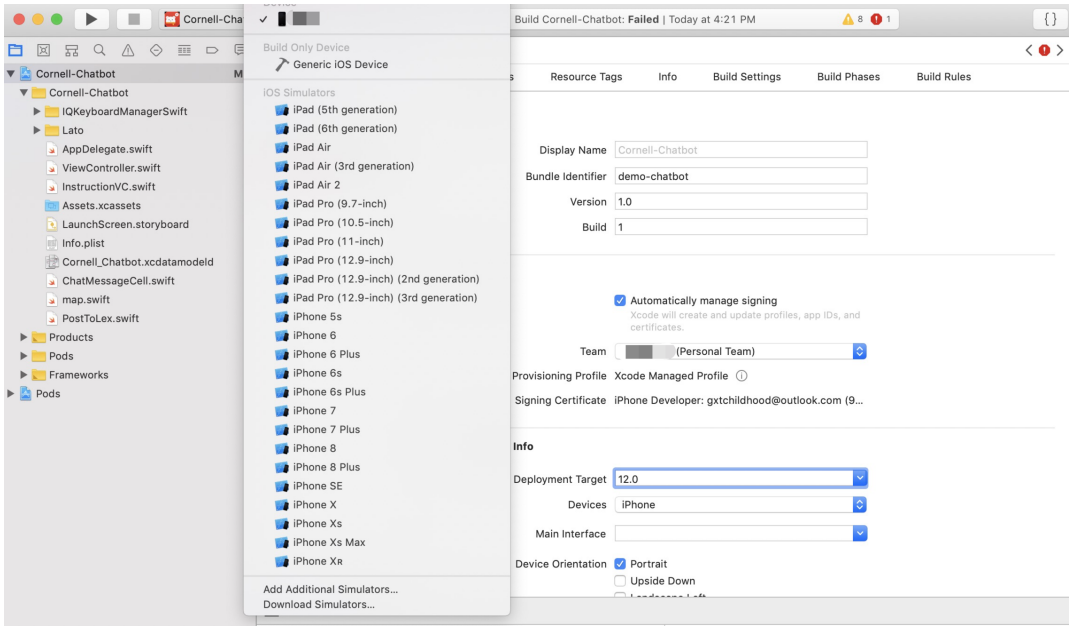
3.1 Launch the App

Once the whole package is downloaded ready, you can open the “chatbot-frontend” file using XCode, and then click the “run” button to test the app. You can select using both virtual devices or real phone devices. For the example below, the app is running on the iOS simulator “iPhone XR”.

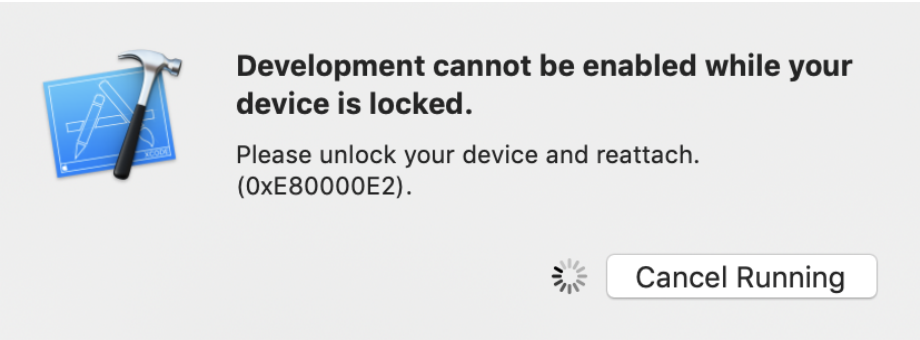


If you are trying to connect to a real device, you need to follow the steps below:

1. Connect your device to the laptop and “trust” your computer
2. Rename “Bundle Identifier”, for example, “demo-chatbot”, and it must be unique
3. Select a development team, you might need to sign in with your own iOS account
4. Choose the “Deployment Target” to match your iOS version, in this case, “12.0”
5. Select your device and click the “run” button



Please make sure to trust your laptop otherwise the following issue will be shown:



For more guide and instruction on how to use this app, please refer to “User Guide”.

4.0 USING THE SYSTEM

4.1 Import the package and setup

To future develop the client side of our system, you will need to have XCode and swift installed on your computer. To properly run our functions, you also need to install the Alamofire and AWSAPIGateway Cocoapods.

If you wish to develop user client sides on other platforms, please install the associated software for them. In this developer guide, we will mostly be introducing the iOS app we developed, to provide a reference to future iOS app developers and/or inspire developers using other platforms.

4.2 Front-End System Menu

The goal of Front-End design is to implement a mobile application based on the iOS operating system. By using this application, users can send text and audio questions and receive corresponding answers from our backend. There are three views inside the application: welcome page, which will appear when users open this application every time; main page, which users can enter questions and receive answers; instruction page, which gives simple ideas about how to use this application to users.

4.2.1 InstructionVC

The *InstructionVC* file is for displaying instruction content to the user. On the top of the page, there is a imageview that works as a decoration. Under the imageview is instructions to inform the user what kind of questions they can ask. The text is scrollable in order to fit in different screen size phone.

4.2.2 ChatMessageCell

The *ChatMessageCell* file is the holder for the message come from both the user and the bot. It consists of a message label to hold the message content and a bubble background view which is the dialog box. It is set to be different colors and align on different sides of the page to differentiate whether it is a message from the user or from the bot.

4.2.3 ViewController

The *ViewController* file contains functions for various different purposes. Please see below for detailed information for each segment of code.

Tableview of messages

The major component of the viewcontroller consists of a tableview containing all *ChatMessageCell*s as explained above. Since users are able to click on certain messages to get map directions or listen to old recordings of voice messages, the tableview is able to react if a user clicks on a certain cell.

We also implemented a *ScrollToBottom* method and call it every time a new message is added to a tableview and it gets updated. Basically, it ensures that the tableview is also scrolled to display the newest chat history between the user and the bot to ease the chat history lookup process for users.

AudioRecorder & Audio Player

Audio chatting with the lex bot is an important feature of our app. Thus, we included functions for audio recording and audio reply in the ViewController class. We used the AVFoundation SDK to initiate recording sessions, record messages and store the voice messages on the device. Then, if a user hits a cell that contains a voice message, we use the AVAudioPlayer to replay the voice message.

Device Registration

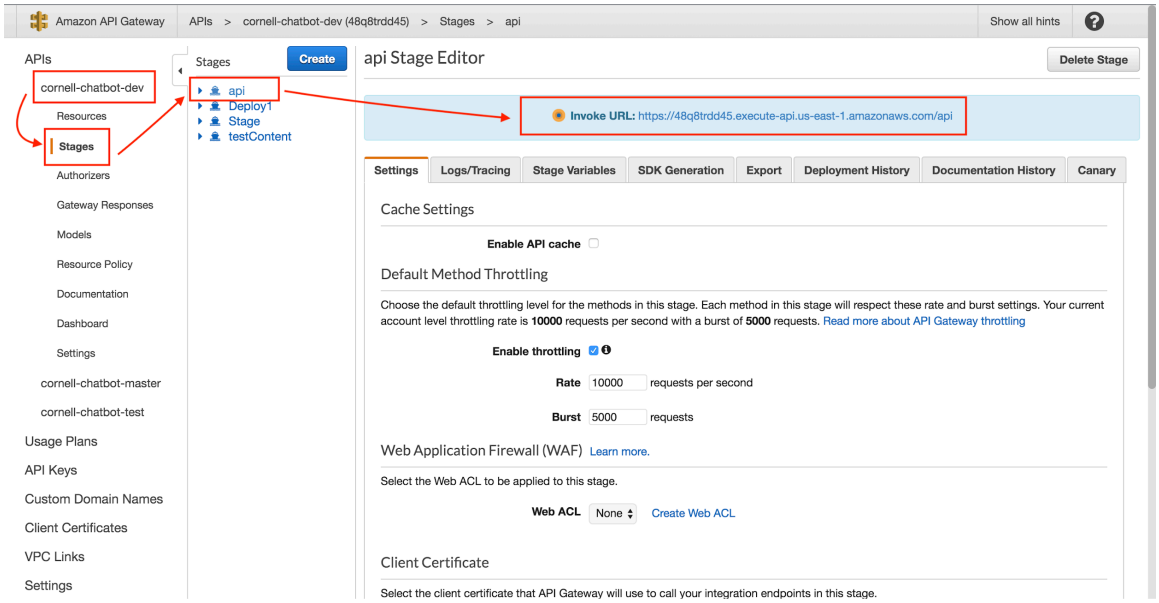
Besides initializing all components in the view, the *viewDidLoad* function goes through a data storage process that attempts to register the user's device and get access permission from our backend. Basically, it does a lookup into the storage of *UserDefaults* and to check if a permission key is already stored. If so, it uses this key in functions of the *PostToLex* class to get access to our bot. Otherwise, it calls the *postID* function and registers the device with the device's unique *identifierForVendor* and stores the returned permission key in *UserDefaults* dictionary for future uses.

Dates

We also implemented a feature of date inside our app, which can let users check the date and time information regarding their previous input. Thus, we used a data structure timer to record time information and a function fireDate to refresh time information after the pre-setting interval.

4.2.4 PostToLex

In order to connect our frontend client side to our backend APIs, we wrote three functions in *PostToLex* to handle different API calls. We used Alamofire to make all the API calls and handle API responses. But other ways are also possible. In order to find the URL for the API calls, first, go to Amazon API Gateway (https://console.aws.amazon.com/apigateway/home?region=us-east-1#/apis), and follow the steps in the screenshot below to find the URL. Note that the API name could differ, given how API StageName and paths are defined in template.yml (more configuration detail provided in 1.2 Project References/Build Serverless API).



The following three functions in PostToLex are methods handling our API calls.

PostID

As mentioned in the ViewController section, *PostID* function is used to register our device by passing the device's unique machine ID to our backend. If succeeded, the POST API call receives a permission key string, which then gets stored by the device. If failed, the API call returns an error with an error message, meaning the device cannot be registered. In that case, we will store a customized "no permission" key so that when the device attempts to communicate with the bot, our backend API automatically rejects the call and returns an error, which the frontend handles by displaying an error message to the user.

PostTextToLex

As explained by the function name, the *PostTextToLex* function makes an API call to our backend by passing the user's text message. Besides the message, we also pass in headers that

contain the user's permission key and an extra field that contains the user's id so our backend can decide if the user is allowed to have access to our chatbot. If so, our backend processes the input message and returns a JSON containing lex's response, which we parse and use to update our *tableview*. If our backend returns an unsuccessful response, either due to permission errors or server errors, the function catches the errors and updates the *tableview* with an error message.
Besides handling the text response from our backend, this function also parses the response JSON to distinguish the user's question intent. If the user was asking for a location related questions such as "where is carpenter hall," our function passes that information back to the main *ViewController,* which uses the intent and the return message to the location in a map application through invoking the *goToMap* function explained below.

PostAudioToLex

Similar to *PostTextToLex,* this function makes an API call to our backend containing the user's message. It passes all the permission key and user id information and does all the error handling similar to the above. The major difference is that this function processes the user's audio recording and convert it to the appropriate PCM format in base64 encoding format before sending the information to our backend. And if there was an error with the audio message conversion, this function returns an error message to the user and prompts the user to try again.

4.2.5 Map

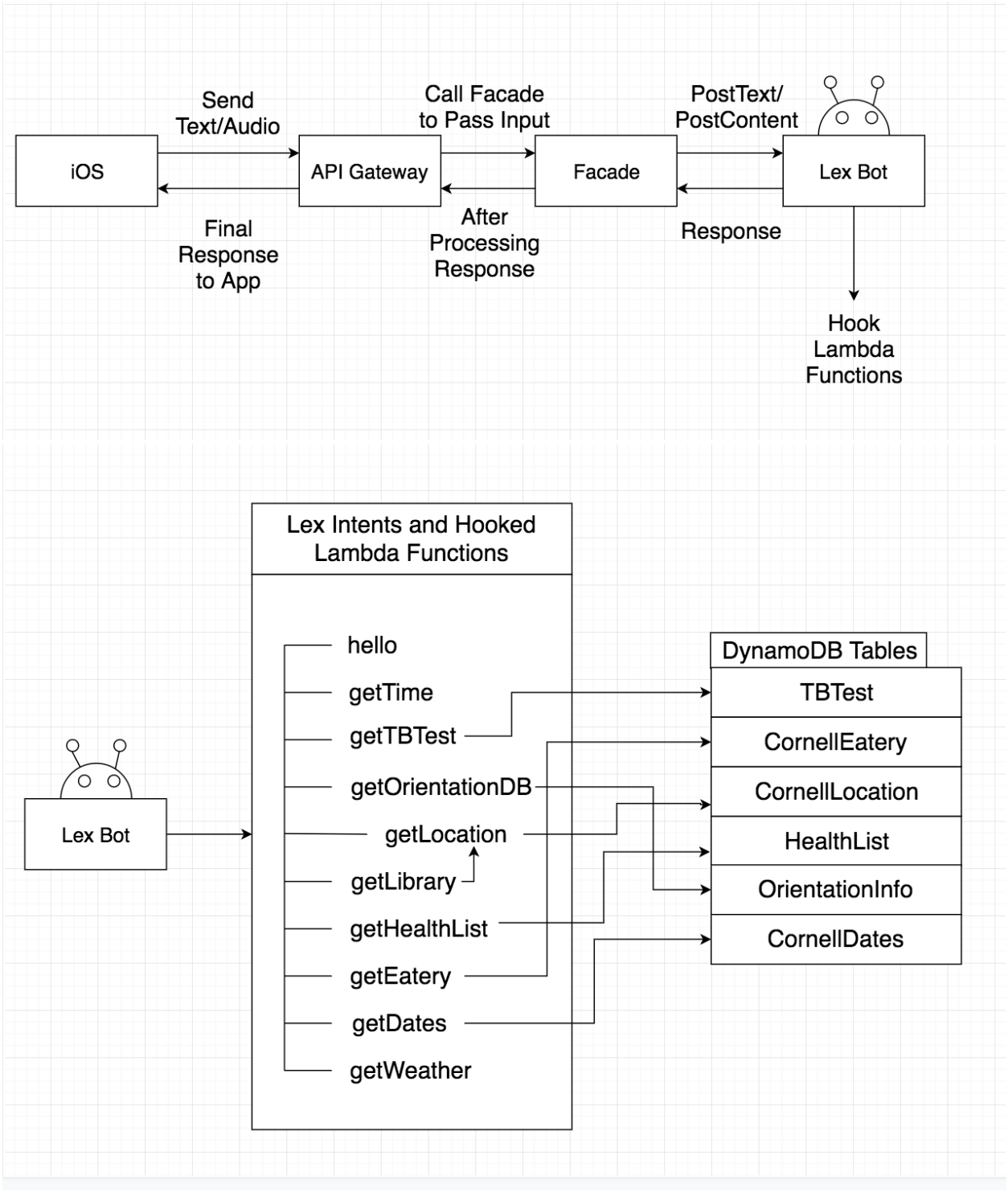
goToMap

map.swift includes function *goToMap* which is used to invoke local map applications on the client side if the user asked lex a question about direction. Since Google maps typically provides the best accuracy, the function attempts to first look for google map, then apple map then safari's map if none of the previous ones work on the user's phone. To ensure that the function only tries to invoke an application if possible, it first checks each of the URL of these applications by attempt URL init. If initialization returns a valid value, then the function proceeds by passing the address to the map application and open it up on the user's phone.

4.2.6 Info.plist

To enable the iOS app to access the phone's location and make recordings, please make sure to edit the info.plist file to grant permission. Also, since the Cornell Chatbot app might attempt to open other map applications, we also need to grant permissions.

4.3 Back-End System Structure



4.3.1 Back-End design overview

The big picture and the goal of the Back-End design are to connect the functionalities in our Lex Bot with our frontend application using API. Therefore, any frontend inquiries would be sent to our backend handlers and be mapped to the proper answers we prepared in our database. Initially, our API would directly call Lex runtime service to receive a response from our bot. Later, we realized the need for pre-processing the response before sending it back to our app. In order to reduce coupling and dependency between API and the pre-processing procedures, we introduced facade. Consequently, API will call the method implemented in the facade, which will then call Lex runtime service. When facade receives a response from the Lex Bot, it can process the response and send the processed response to API, which then sends the response to our app.

4.3.2 Lambda Functions for Lex Model

hello()
Remembering the user's name, and greeting with a welcoming message.

getTime()
Giving the chatbot user exact local time.

getWeather()
Adopting the Weather API to get current weather information (weather type and temperature interval) based on the user's input location and the default location is “Ithaca”.

getOrientationDB()
Using a DynamoDB table of orientation information to answer the basic questions to the freshman orientations. Relying on unique major slot type, a freshman in different departments can quickly get to know the detail location, orientation event time and emergency contact information.

getLocation()
Answering a location inquiry with an address. It first searches for the address in our Cornell Location database table. If such location is not stored in the database, it will refer to Amazon GeoLocation API to return an address for that location.

getLibrary()
Answers users with a list of six Cornell libraries and their locations.

getDates()
Obtaining add or drop class deadline from DynamoDB table to inform students with important academic dates. Two actions are enabled. If the user asks about “add class”, the follow-up question “what year are you” will be triggered since students in different years have different deadlines. If the user asks “drop class”, only one answer will be provided since the drop class deadline is the same for all students. Drop class deadline includes dates that dropping without any penalty, dropping with a W on the transcript and the absolute drop date with penalties.

getHealthList()
Cornell Health requires international students to take a list of vaccinations before officially matriculated. If the student hasn’t completed the requirement, they may face course hold or other limits. As a result, the health requirement is among the most focused topic for newly-enrolled students.
As an answer to this problem, we set up a DynamoDB list which stores the vaccination requirement list for international students. By asking the question about the general health requirements, the user will get a completed list about what they need, therefore they don’t have to search through the Internet.
For questions on specific vaccination shots, the bot will also answer the specific information on these topics. For example, if the user asking questions on Tdap, the bot will answer how many shots should be taken or shown on this student’s immunization record.

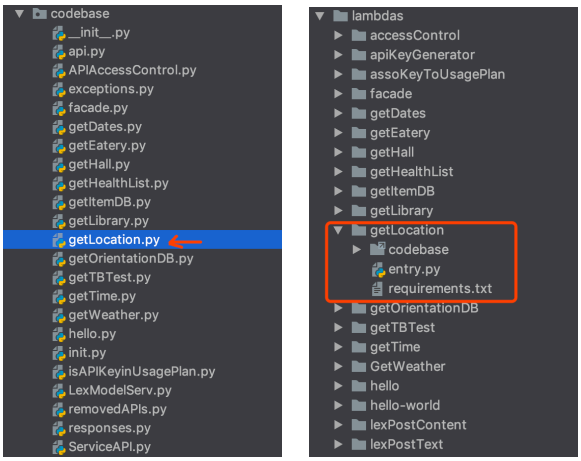
getTBTest()
This is a sub-topic of the general health requirement of Cornell. For the international students from countries identified by the World Health Organization as has “high evidence” of Tuberculosis (TB) existence, they have to take TB test at Cornell Health.
It is a bothersome process for the new students to search through the Internet to check if their countries are on the list. And it is also troublesome to consult whether they have to take the TB test at Cornell. This function will give a prompt answer to the students who have such concerns.

getEatery()
Recommending eateries around Cornell campus, based on where the user is/where the user wants to eat. After the user expresses an interest in finding a place to eat, the chatbot will ask where the user locates. It will search in our CornellEateries DynamoDB table based on the request and fetch the eateries in that specific area. If the user is not sure where s/he is, the chatbot would answer with eateries around Central Campus. If the user does not have a

preference, the chatbot would randomly pick an area for the user and recommend the eateries for the user.

4.3.3 Lambda Function handler

After you created a lambda function under the folder of codebase, you need to specify a handler, which is in the corresponding entry.py under folder of Lambda/[function name], this handler function invoked and triggered the corresponding intent function when AWS services executed. In addition, the handler function specified the exceptions and assert possible errors in the console of AWS Lex.



(Left: Python function, Right: corresponding handler function)

4.3.4 YAML file explanation

To connect the lambda functions and Amazon web services, we need to use a template in YAML to accomplish. Even though the Json template can also achieve this, with similar semantics capabilities, the YAML file provides better in the way of human-readability and support comments. In the Cornell chatbot, we specify our AWS CloudFormation resources and properties in YAML-formatted file.

Create Lambda Function

In the following section, the detailed explanation of the YAML file is introduced with the case of GetLocationFunction.

The type if specified as AWS::Lambda::Function resource which creates a Lambda function. A deployment package and an execution role are both required to build this Lambda function.

In the Properties section and its attributes, FunctionName and CodeUri specified the file location. As for the syntax, intrinsic function Fn::Sub substitutes variables specified the values and corresponding environment settings.

In the Events, the function's tracing configuration are described in Mode and Get tells us that we are retrieving data from AWS services through a REST API.

The Policies attributes restrict the right to the identity-based policies in IAM. When user make a request, the request is denied or allowed strictly obeying the rule of consensus between the engineering team and clients. Misuse of policies would lead to security violations, network abuse, and spam, etc.

In the case of GetLocationFunction, AWSLambdaBasicExecutionRole and AmazonLexFullAccess relatively grant the function to access AWS services and resources and connection read-only access to DynamoDB resources. This is because the function only needs to retrieve data from databases rather than write the databases. In our design, the related NoSQL DynamoDB elements can be added, deleted and edited in the console of DynamoDB on the AWS web services manually.

```
# ==========  get location ===========

GetLocationFunction:
  Type: 'AWS::Serverless::Function'
  Properties:
    FunctionName: !Sub '${Application}-${Environment}-getLocation'
    CodeUri: './build/build/lambdas/getLocation'
    Tags:
      Application: !Sub '${Application}'
      Environment: !Sub '${Environment}'
      Resource: !Sub '${Application}-getLocation'
    Events:
      GetLocationResource:
        Type: 'Api'
        Properties:
          Path: '/getLocation'
          Method: 'GET'
          RestApiId: !Ref 'Api'
    Policies:
      - 'AWSLambdaBasicExecutionRole'
      - 'AmazonDynamoDBReadOnlyAccess'
```

(Screenshot: GetLocationFunction resource in template.yml)

Create DynamoDB Table

In the following section, the detailed explanation of the YAML file is introduced with the case of CornellEateryTable.

The type if specified as AWS::DynamoDB::Table resource which creates a DynamoDB table without changing schema in DynamoDB console. A deployment package and an execution role are both required to build this DynamoDB table.

In the Properties section and its attributes, TableName specified the name of the created table. AttributeDefinitions and KeySchema configures primary key information in the table. In the case of CornellEateryTable, the primary key is the field area, and therefore is specified in the AttributeName.

If needed, more details are provided in 1.2 Project References/DynamoDB/Create through templates (AWS CloudFormation).

```yaml
#======= Cornell Eatery Table Creation ============

CornellEateryTable:
  Type: AWS::DynamoDB::Table
  Properties:
    TableName: CornellEatery
    AttributeDefinitions:
      - AttributeName: area
        AttributeType: S
    KeySchema:
      - AttributeName: area
        KeyType: HASH
    BillingMode: PAY_PER_REQUEST
```

(Screenshot: CornellEateryTable resource in template.yml)

5.0 APPENDIX

5.1 User Testing Instruction

User testing instructions: Sentence in double quotation marks are instructions for users

"You are a student new to Cornell. Today is your first day at school and you want to head out and explore the school, so you open the Chatbot app, browse through the page and feel free to take your first step."

1. **Get the name of the user (Expect user to type "hi" and reply their name)**

"Before go out, you want to check the weather."

2. **Get the weather (Expect user to type "What is the weather like today")**

"You want to first go to the orientation of information science major, but you don't know where it is."

3. **Get the location for orientation (Expect user to type "Where is the orientation of information science")**

"During the orientation, you heard about the MPS lab, so after the orientation, you want to check out the lab, but you don't know where it is."

4. **Get the location for MPS Lab (Expect user to type "Where is MPS Lab")**

"You want to get directions to the MPS Lab"

5. **Go to google map for detail direction (Expect user to long pressed and go to google map)**

"When you arrive at MPS Lab, you find it too crowded, so you want to find another place to study."

6. **Find a library in school (Expect user to type "Where can I study")**

"Sitting in the library you find, you start wondering when you can add classes."

7. **Find out the add class date (Expect user to type "When can I add the class")**

"After spending some time in the library, you are hungry and it's lunchtime, but you don't know where you can eat."

8. **Find out a place to eat (Expect user to type "Where can I eat")**

"Now feel free to play with the app, ask any question you want in the app imagine you are a new student."