Automating Social Interaction with Bots, Voice Interaction and Speech Recognition

Group #4

Team Website

Aron Riesenfeld

Roderick Montgomery

Gustavo Flores

California State University, Northridge

Spring 2020

ABSTRACT

Our project was to create a software application that would respond to voice commands in order to display posts made by users and to post on behalf of multiple accounts of the user. This was done in two modular phases along with two other prototypes to help us think in the right direction. The first was to create a bot system that worked and responded in real time to perform repetitive actions so the user wouldn't have to. Phase two was to build on the first system and incorporate speech recognition into its abilities so the user would be able to control it with their voice as well as get feedback of their actions.

TABLE OF CONTENTS

INTRODUCTION AND BACKGROUND

STATEMENT OF PROBLEM AREA

The problem we were working with was to create an application to allow users to verbally interact with multiple accounts from a social media website. For example, if a company has multiple Twitter accounts such as a sports, news and entertainment account, all of which had different users following each account. Now, say they wanted to display a certain message to all of their followers. Then instead of having someone log in and tweet the same thing from all of those accounts each time. Then they could use our application to log in once into all of the accounts and then proceed to tweet from them all or from only certain accounts, whichever is needed at that time.

PREVIOUS AND CURRENT WORK, METHODS, AND PROCEDURES

The main approach was to do research on which tools we could use. This led us to Python being the main pillar in our technological stack since we knew Python had the capability of user speech recognition as well as the ability to use a lot of different API's. We tried formulating a clear and achievable plan within the scope of our time frame which we overshot. This is where potential problems with interactivity are attempted to be caught and the rules of user interaction laid out before the design phase.

Previous to our first sprint, we first created a prototype of a simple bot that would tweet out a link to a map pointing to the current location of the International Space Station in Python and hosted our code in Heroku to be executed every 5 to 10 minutes. This helped us understand how we could use Python and it's libraries to post and retrieve data from Twitter. Which gave us the first steps to start working on our project.

Our initial technological stack without voice recognition was steeped in Python, Twitter and Heroku. However for speech recognition we then needed to branch out to include S3, CloudWatch, Alexa Development Console and Lambda and discontinued using Heroku as this would mean we would have more polished tools to use at our disposal as well as the Python libraries we were already using.

This was after we made another prototype that included using speech recognition and voice activation to be able to post from a single Twitter account. This is what gave us our first look into creating and using a Voice User Interface and helped start yet another fundamental base for our project.

BACKGROUND

As mentioned above we used a Twitter bot to get live information from the cloud. We were able to communicate with NASA to find the current location of a specific satellite in terms of the longitude and latitude. We were able to use that information to view the satellite's current location from an aerial perspective. A second achievement was to view the last 6 tweets from Jack Dorsey, the inventor of Twitter. In both cases we demonstrated live and real time information being displayed and read.

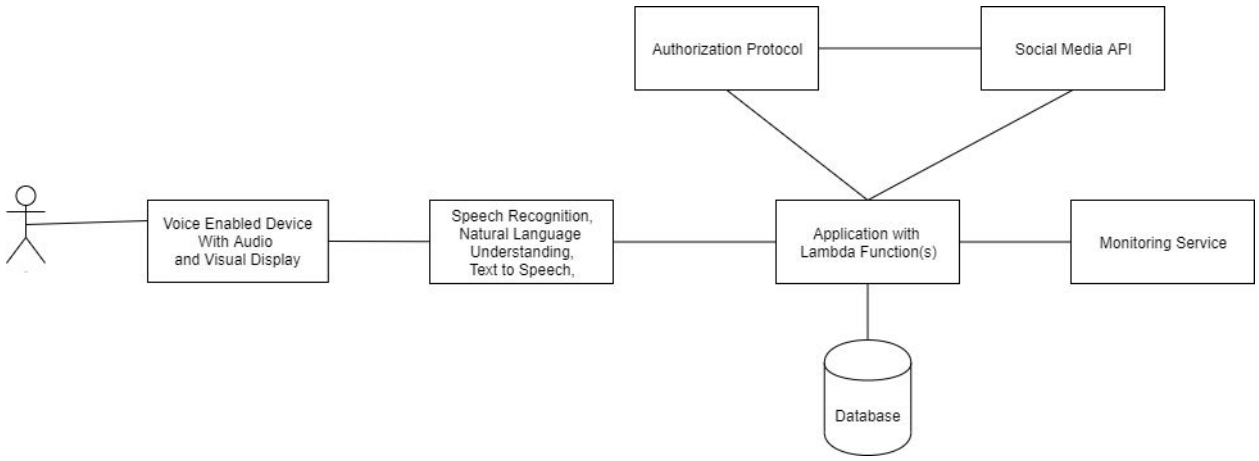
The idea was to build upon this foundation and allow the user more flexibility such as posting from multiple accounts or only certain accounts and also managing their accounts such as signing in or removing accounts. Once that was completed we wanted to incorporate speech recognition into the equation. This was implemented by the result of verbally asking a question such as "What was Jack Dorsey’s latest tweet ". And then a verbal response back with the latest tweet that he posted.

BRIEF PROJECT DESCRIPTION

We wanted to give user’s control with a simple managerial style, and to minimize code and improve simplicity. We focused on our social media platform to be Twitter and we used Alexa for our voice user interface for our first implementation. However we also included another implementation guide for using Facebook and Azure instead. We included the ability to post from specific or multiple accounts, and retrieving posts about yourself or other users, as well as managing these accounts such as adding new ones or deleted other ones. This enables users to perform simple and repetitive tasks automatically with little effort.

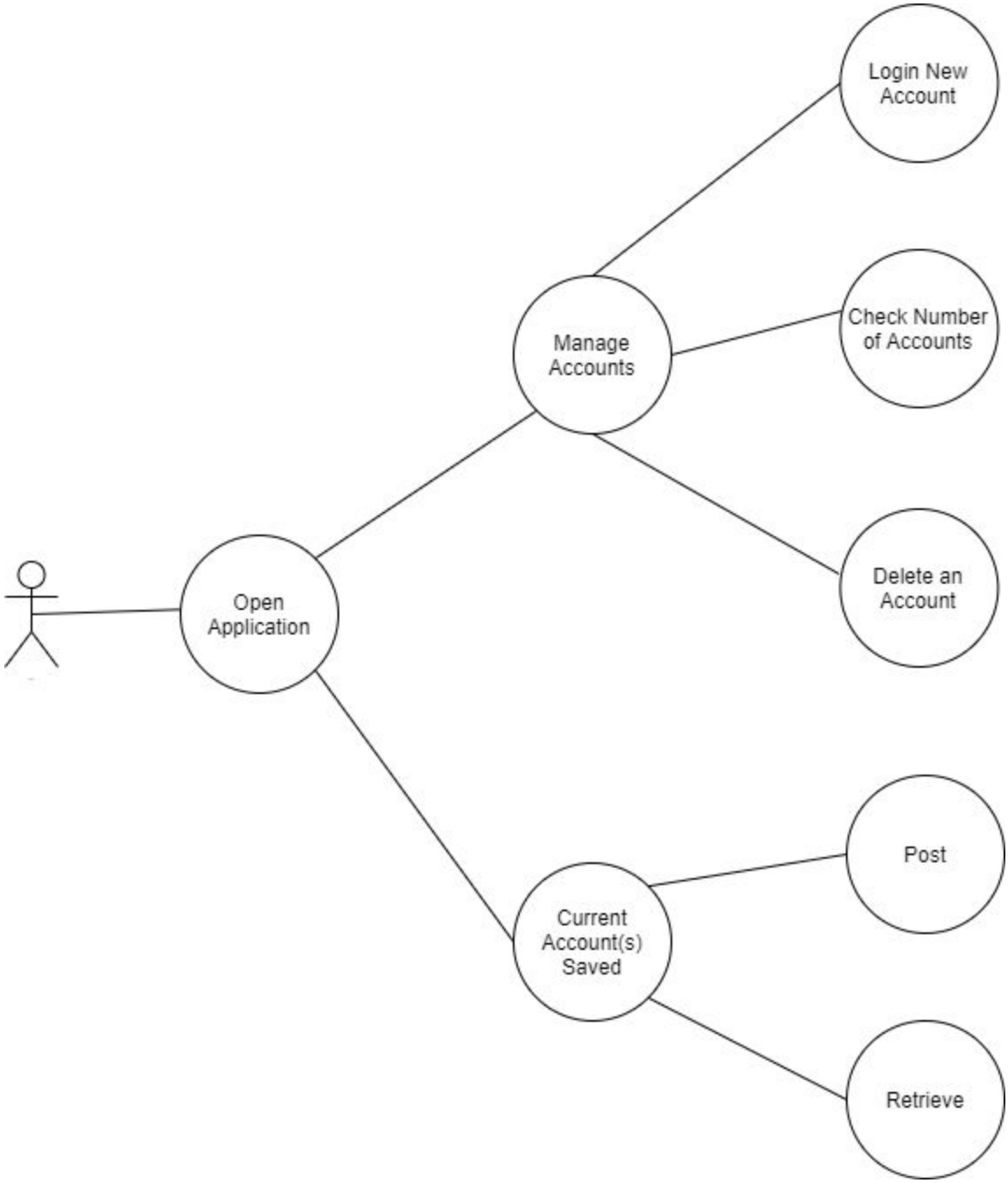
REFERENCE ARCHITECTURE

Architectural Diagram



The architecture for this project is shown above. We would have a user that would interact with a voice enabled device mostly through voice which would however support not only audio but also some sort of visual display feedback to allow the user to log into their social media account. The device should support speech recognition, natural language understanding and text to speech in order to be able to process what the user says as well as provide audio feedback to them. That information should then be handled within the application and taken action depending on what the user wants to do. Since we're dealing with social media accounts, the best way would be to use their API to retrieve or post information which would require an authorization protocol to log in. We could also use a database to store information such as images we might want to post or save as well as information of a user to avoid having to sign in each time they use the application. Also having some type of monitoring service when working with a voice user interface is very helpful since you will be able to view your logs and other metrics using these types of tools.

Case Diagram

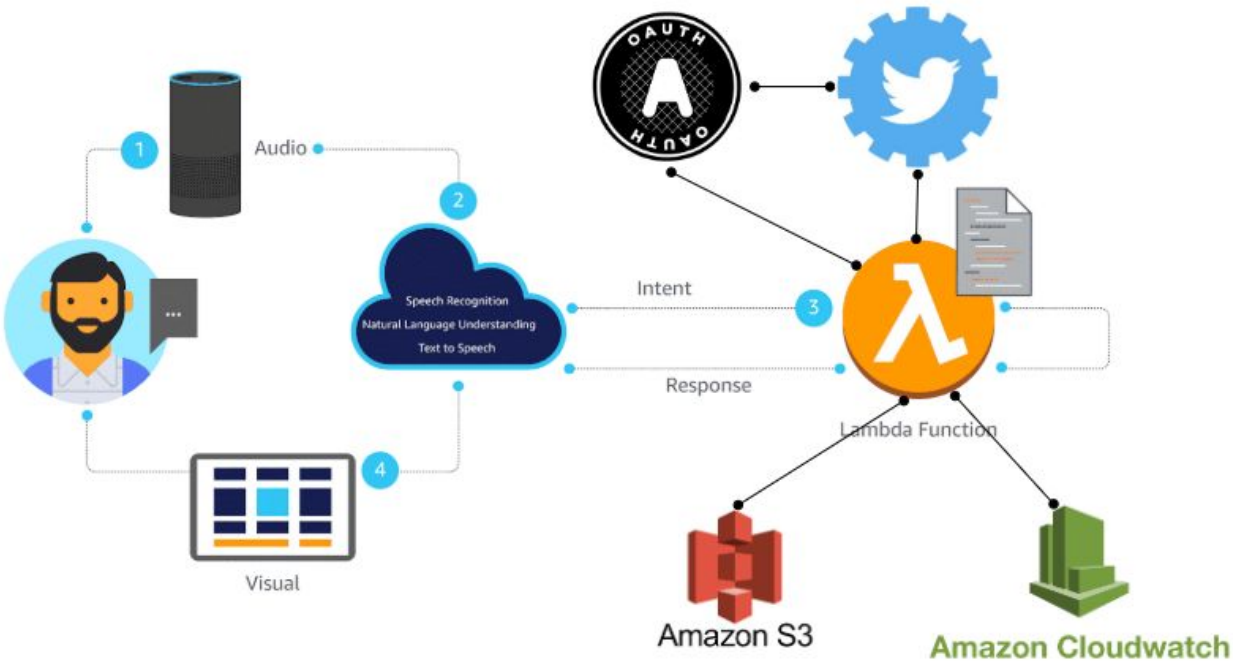


Above is a simple case diagram that shows what this type of project should handle at the bare minimum. A user should be able to open the application then check what accounts they have, add a new account or remove an account. They must also be able to post from all of the accounts saved as well as retrieve posts for the user.

IMPLEMENTATION GUIDES

IMPLEMENTATION GUIDE DETAILS #1

For the first implementation we will create a voice activation system that uses Alexa in order to tweet from multiple accounts using the Tweepy Python library and the Alexa Developer Console. The Alexa Developer Console comes with other Amazon Web Services such as AWS Lambda, Amazon S3, AWS CodeCommit repository for managing your code and Amazon CloudWatch as well as many self-service API's. The editor will be used to write directly to our Lambda file which will be used to execute our code. We can use S3 to store objects and we will use CloudWatch to monitor and manage our skills (skills is just another word for applications when talking about Alexa) by looking at the logs and other metrics. The user will be able to sign in and also remove as many accounts as they want from the skill. They will also be able to retrieve the most recent tweet from any user. Our architecture for this implementation will look like the picture below.



Before we get started with Alexa we will need to create a Twitter Developer Account. If you already have a Twitter account then you won't have to create a new one. After your request has been accepted you will need to create an application which will then grant you authentication credentials in order to use the Twitter API within our skill. Here is some information on how to get started.

First you will need to create an Amazon Developer Account. If we choose to host our Alexa skill on the developer console then we will have access to an S3 bucket, Cloudwatch, as well as a

code editor and we will also have the ability to use the simulator to test our skill. This is useful if you don't own an Alexa-enabled device as you will still be able to build, edit, test, and publish a skill all from the developer console.

Once you create an account, log in and click on the dashboard followed by Alexa Skill Kit. Here you will be able to see and manage all of the skills you create. You can then create a new skill and in this case choose a custom skill followed by an Alexa-hosted option of either Python (which is what we used) or Node.js. However, If you don't necessarily want an alexa hosted skill then you can create a lambda function endpoint which gives you the option of using Node.js, Java, Python, C#, or GO or you can create a web service in any language as long as it fulfills the requirements presented by Alexa such as being accessible over the internet, accepting HTTPS requests on port 443, supporting HTTP over SSL (Secure Socket Layers)/TLS (Transport Security Layer) etc. After you've chosen a language and a method of hosting you can then choose a template. You can choose some of the more advanced skills to see all the tools that you can use or you can choose the most basic skill "Hello World Skill" to be able to delete some of its contents easier. Once you've chosen a template you can then create your skill.

Now you should be in the Alexa developer console of the skill you just created. So let me provide some information on some of the wordings, if you'd prefer reading about it in more detail you can view the documents provided by Amazon. Also if you'd like to skip this information and move onto the actual implementation you can click here.

A *skill invocation name* is used to begin an interaction with a particular custom skill. It's pretty much how you start your skill. Ex. "Alexa Open Area of Shapes"

An *Intent* represents an action that fulfills a user's spoken request. You map spoken inputs to the intents your cloud-based service can handle. It comes with default intents such as the HelpIntent which you can modify to help guide your user through your skill.

Utterances are a set of likely spoken phrases which are mapped to the intents. Typically you want to have as many Utterances for each intent as you can, incase the user doesn't say the exact word you were thinking of such as for example if you want to invoke the CalculateAreaIntent you might have utterances such as "Find the area of a ___", "What's the area of a ___", "What will be the area of a ___"

Slots are like variables that hold a user's response. For example if your intent needs an input from the user such as "Find the area of a {Shape}" here the {Shape} is the variable that we will store in order to use with our intent. There are also custom types for slots for example if you want the user to input a phone number then you can set the type of this slot to be a phone number which will convert the numbers or words that represent a phone number into a string format without punctuation which will make it easier to use in your code.

The first steps to configuring your skill would be to go to the “build” tab and give your skill an invocation name, this can be whatever you’d like however naming it something to do with what your skill does is always more informative. Please note, whenever making changes in your model remember to save it or build it (which will also save it) in order to test it and which will make sure there aren’t any utterance conflicts or other errors.

Optional but highly recommended would be to enable the use of displays, which can be done by going to the build tab then interface option on the bottom left side followed by turning on Display Interface as well as Alexa Presentation Language (APL). This will allow you to present information to the user through voice AND through a display. I would also suggest enabling the Auto Delegation setting in order to simplify your code by being able to collect and confirm multiple slots at a time if a user fails to input one of them.

Optional if you would like to see how the simulator for testing works now would be a good time before deleting anything. Simply go to the “test” tab at the top, change the “off” option to “deployment” and here you can either type what you would have said or you can hold the mic button and speak as you would normally speak to an Alexa-enabled device. The only difference is you won't have to say the wake word “Alexa” each time you want to grab its attention. So you would just have to say something like “Open” followed by the invocation name of your skill. You can also use your own Alexa-enabled device for testing by first setting your device up with the same account information as your Alexa developer account or by resetting its settings to then match the same account information.

The second step would be to add your intents (as well as remove the hello world intent), the most basic and essential ones we added was an intent to tweet from every account, an intent to tweet from a specific account, an intent to log in, an intent to parse the pin after using the login intent, an intent to tell the user what accounts we currently have logged in and an intent to remove accounts. Of course you can edit these as you see fit as well as add other ones, for example we also then added the ability to retrieve the most recent tweet from any user.

Once you’ve added your intents don’t forget to add the utterances that would likely trigger those events along with whatever slots you think you’ll need. For example if a user wants to tweet something you’ll have to have an utterance of what they’ll say to trigger that intent along with a slot to store what they want to tweet. For example “Tweet out {TweetGoesHere} from all of my accounts”. As mentioned before don’t forget to save your model or simply build it (if you’re going to test it) in order for changes to take effect. You can also view your model versions in case you want to revert back to one of them.

Now it’s time to get into the actual code. However since this can be done in many different languages and many different ways we will stick to being as abstract but also as helpful as possible. Go to the “code” tab at the top of the page, here you should be presented with a folder that contains all the current files. One of them being the lambda function which, depending on what language you picked, will be that type of file as well. Here is where the main functions of your skill are currently located since we chose a template. There is also a requirements file

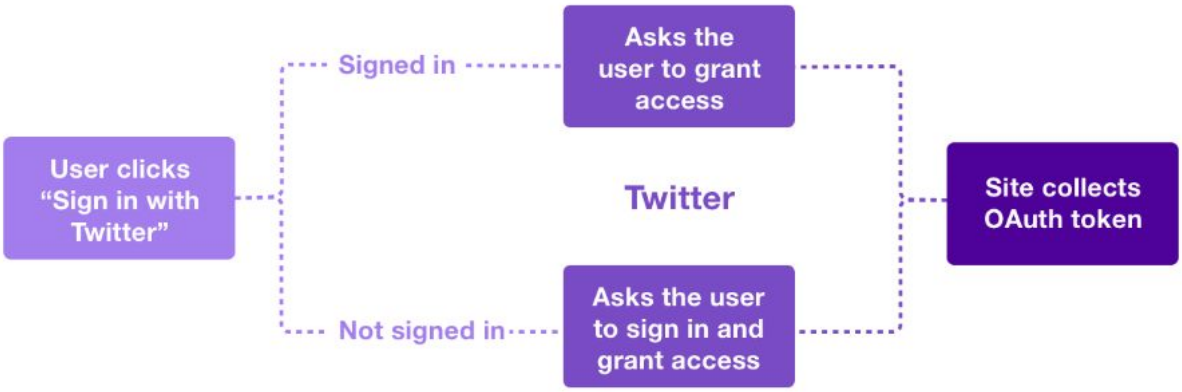
which is where you can specify versions of outside tools or libraries that you plan to use. Here we imported the latest version of the Python library Tweepy to ensure easier access to the Twitter API. We also imported Pyqrcode to create a QR code from a link (which we will later discuss). The other file we currently have is the utilities file which, if hosting on Alexa services, has a function to allow us access to our own AWS S3 Bucket that we can use. Also if we look at the bottom left, all the way under our file folders, we will have links to our Cloudwatch to view our metrics and logs, our S3 bucket where we can store information such as keys, images or any other objects and helpful documentations as well to guide us through creating our skill.

You will want to create a class handler for each intent created in the “build” section, this handles what your intent does after it is called by the user. For example you can have an intent that displays to the user how many accounts they currently have logged in so in this case you would need to create an intent in the “build” section as well as a new class in the main lambda file for it. This will contain the code to collect the names of the accounts we have stored as well as present them to the user through voice and/or the display.

It is up to the developer on how they want to store their information as well as what libraries to use and how they want to present everything to the user. However, we do have some recommendations that we would like to mention. One of our recommendations would be to create a handler file for each account that uses threading in order to be able to handle tweeting out from multiple users at a time.

Also as mentioned earlier you will need to have signed up for a Twitter developer account and create an application in order to receive credentials to use the Twitter API by itself or through Tweepy. The Twitter API uses both OAuth 1.0a and OAuth 2.0 but since we are going to be writing and not just reading we need to use OAuth 1.0a . You will need to pass the Twitter API the consumer key as well as the consumer secret key of your twitter developer account to be able to use the Twitter API.

Another recommendation is that since you will need to provide a way for the user to log in to their Twitter account, typically through a link, there will have to be some sort of display feature while using your skill since displaying a link verbally and expecting the user to write it down is not really feasible especially the ones generated by the API. This can be solved by providing the user with a link possibly through their phone or as mentioned before, using a library to turn the link into a QR code and then displaying that through the Alexa-enabled device. This must be done since Alexa doesn’t currently support clickable links to be displayed to the user from a developer standpoint (it is supported in Alexa however it is not available to developers to use). Once they log in we will then be able to verify using the pin provided and confirm the pin within our code which will then give us that account’s OAuth tokens which we can then store to avoid asking the user to log in again. Provided below is a reference to how the user would interact with the Twitter API whether they are currently logged in through their device that opened the link or if they are not currently logged in.



You can beta test your skill by making it available to only a limited group of testers that you have personally selected that way you can get some feedback about your skill without being overwhelmed by releasing it to the public. This can help reduce costs by catching and making fixes before you release the skill to the general public. You can also use the beta testing tool to test changes to an existing skill, while still keeping the currently live version of the skill available for the general public.

Once you are ready to deploy your skill to the public you must make sure that your skill fulfills the requirements of the certification checklist as well as run the validation and functional tests. Once these requirements have been met you can then submit your skill for certification.

If your skill is then certified, Amazon customers will then be able to see your skill in the Alexa App and then choose to use it if they'd like.

IMPLEMENTATION GUIDE DETAILS #2

Our second implementation makes use of Azure and Heroku rather than AWS, and interacts with the Facebook API rather than the Twitter API. All of the speech recognition and playback is handled by Azure Cognitive Services, and all Facebook API calls are handled by a group of Heroku worker dynos. The user of our system will be interacting with a local client built with Python, which records and plays back voice user interface (VUI) audio and coordinates the sending and receiving of data to and from the above cloud services. The component of this project that separates it from other social media clients and bot managers is its VUI, in which user speech is interpreted by the Azure Language Understanding Intelligent Service.

The Azure Language Understanding Intelligent Service (LUIS) operates on a similar set of principles to AWS Alexa, digesting the intents of user input utterances and executing code based on those intents. The concepts described in the LUIS documentation are similar to those found in the Alexa documentation, with some subtle differences. However you can skip these concepts by clicking here.

An *utterance* is a user's natural language input to a LUIS application. These are the user's raw commands that are to be parsed by the LUIS app. As part of a user-friendly experience, it is helpful for the app to be able to capture and understand several different utterances for each action available to the user, taking into account the fact that natural language allows the user to issue the same command in multiple different ways. LUIS ought to be able to understand relevant information and discard irrelevant information by examining utterance length, sentence order, pluralization, and vocabulary choice. For instance, the phrases "Read the comments on my last status," and "Show me the responses I got on my most recent status update," should both result in the same action being taken by the system.

An *entity* is a keyword that is extracted from an utterance. Entities are specified so that LUIS can reliably extract information from a variety of different utterance formations. An important component of LUIS is its incorporation of machine learning to better associate users' commands or requests with an appropriate system response, so entities can be learned from context. However, a developer is free to record specific entities that LUIS can pattern match against the utterances it takes as input. Returning to the two example utterances above, the entities that might be isolated by LUIS are "Read" and "Show me", "comments" and "responses", "my last" and "my most recent", and "status", and "status update."

An *intent* is the task or behavior that is associated with the entities extracted from an utterance. This is the goal that the user intends to achieve. Every possible action in a LUIS application must be described as an intent. Every LUIS application comes with a None intent by default, which is used as a fallback for when a user's utterance does not match any recognized entities. When an utterance leads into the None intent, the client can either request clarification from the user or provide a help menu with valid commands from which the user can choose.

Intent	Entity	Example utterance
CheckWeather	{ "type": "location", "entity": "seattle" } { "type": "builtin.datetimeV2.date","entity": "tomorrow","resolution":"2018-05-23" }	What's the weather like in `Seattle` `tomorrow`?
CheckWeather	{ "type": "date_range", "entity": "this weekend" }	Show me the forecast for `this weekend`

An example of the relationships between utterances, entities, and intents for a weather-checking application

An important distinction between Azure LUIS and AWS Alexa must be noted. While Alexa is specialized for processing a user's verbal input, LUIS takes input as text. Therefore, we need a way to take a user's verbal commands and convert them to text that can be processed by LUIS.

Azure offers a Speech to Text service and an associated SDK that we will be incorporating into our system. To use Speech to Text, we need to create an Azure resource and associate it with a subscription to the Speech service, thereby acquiring the subscription keys needed to access the service securely.

We will also need the ability to record an Audio file from the user's client and send it over to the Speech service. We will use the PyAudio library to record a user's utterance into a .wav file, since the Speech service accepts input in that format.

Recognizing user voice input from an audio file requires using the Speech SDK to communicate with the Speech service. Using Azure's App Service, we can host our Speech application in the cloud. Sending audio input to the service and receiving text back from it works in the following way: Using our subscription keys and region, we create a SpeechConfig object. With the .wav file received from the client, we also create an AudioConfig object. Using these two objects, we can then create a SpeechRecognizer object, which we will use to recognize the language in the utterance given by the .wav file. The SpeechRecognizer will return the text of the utterance wrapped in a SpeechRecognitionResult object. We will then extract the text and return it as a response to the client, and from there, feed it into LUIS.

After the user's voice input has been converted to text, it will be received as a response from the Azure Speech service to our local client. From the client, we will send it to our LUIS application, which will parse entities from the utterance given by the user and activate the appropriate intent.

As with Azure Speech, we will need to create an Azure resource associated with LUIS and a new Azure app. We can access our app from the My Apps section of the Azure portal, and from there we can build out, train, and test our app. Next, we must publish our app to its endpoint. Using the endpoint URL we have created, we will then be able to query our application, sending the text produced by the Speech service as a query parameter to LUIS, which we can send

programmatically via a REST API. Our LUIS app will return to us a JSON object which we will next use as input to our Facebook status scheduling bot and third-party Facebook client code, which will live on a Heroku server.

While Azure provides the Azure Bot Service, it is primarily intended for chatbots, and it is not well-suited for simple, schedulable interactions with the Facebook API. Therefore, our tweet scheduler feature and all of our Facebook API calling code will be hosted on Heroku, a PaaS service that is built with these types of interactions in mind.

Once our LUIS application has returned to our client the JSON object that we will use to determine how we proceed with the interaction. If LUIS has returned an object that indicates the user's utterance has led to the None intent, we will re-query the user and provide the user with a list of acceptable commands. Otherwise, we will extract the proper intent from the returned object and use it to call a remotely-stored Heroku worker.

Heroku applications are built into lightweight, isolated containers called *dynos*. Dynos contain all of your application code as well as all the language and framework dependencies needed to run that code. The dynos offered by Heroku are divided into three classifications: web dynos for hosting web applications, one-off dynos for small jobs that you don't intend to repeat, and worker dynos for everything else. Each of our API interactions will be separated into its own worker, and the bot will live in its own worker as well, for which a scheduler can be set.

When the local client receives the JSON response from LUIS containing the intent of the user, the client will call an appropriate worker. If LUIS determines that the user's intention will be to post a new status, for instance, the worker containing the post-status function will be called remotely, with the text of the user's status being submitted as an argument to the worker being called and the function within it, which will send a post-status request to the Facebook API. Likewise, if the user's intent is to retrieve data from Facebook, the worker that contains the appropriate API call will be called, and it will return to the client the requested data. Additionally, if the user's intent is to request that a bot perform Facebook actions automatically, the worker that contains the bot will be called, with the frequency of the behavior being passed in as an argument. The worker will set a scheduler to automate the requested behavior.

When we call a worker, we will receive a text response. This response will contain one of two things. In the event that the user completes an interaction that does not request data from the Facebook API (such as posting a status) a simple success message will be returned. Otherwise, if the user is requesting data from the Facebook API (for instance, requesting all of the comments on their most recent status), the response will contain all of the information that the user requested.

Once the user's client has received this text response, it will need to provide it to the user. Since this system uses a voice user interface, we will need to convert that text into speech that the

user can listen to. To do this, we are returning to Azure Cognitive Services, this time making use of its text to speech service.

Azure's text to speech functionality works very similarly to the previously described speech-to-text process. Once again, we are relying on the Azure Speech Service to handle this work for us. As we did earlier in this process, we will be using our subscription key and region information to build a SpeechConfig object as well as an AudioConfig object to specify the .wav file we will be sending back to the client. Using these, we create a SpeechSynthesizer which takes our text and and, from it, produces a synthesized reading of the text. We will then send the resultant .wav file back to the client, where it will be played by the client using another local audio library.

A notable feature of this implementation is that the client coordinates the activity of all of the services to which it offloads the work of speech recognition, speech transformation, and Facebook API interaction. The client takes user input in the form of speech, which it records as an audio file. It then sends off that audio file to the Azure Speech service, where it is transformed into text. The text returned by the Speech service can then be sent off to the Azure Language Understanding Intelligent Service, which parses that text and extracts the intention of the user, once again returning that information to the client. The client uses that data to determine which Heroku worker it needs to call in order to process the user's request, calls the appropriate worker, and awaits whatever data is to be returned from Heroku. That data, be it a confirmation message of success or failure, or a string of formatted text requested by the user, is then sent off once more to the Azure Speech service, this time converting it into a synthesized speech audio file. That audio file is then played by the client, and the interaction loop is complete.

CONCLUSION

SUMMARY

The focus of our project was on user accessibility and workflow efficiency. The whole time keeping the end user in mind. So speed and management came to the forefront of our design approach. In the end this application can be used for a wide range of needs: business, practical assistant, entertainment or simply general personal use.

Starting with a foundation of Python for voice interaction, we were able to accomplish basic vocal output resulting in a text response in real time. Efficiency was pretty good in terms of verbal output recognition as well as proper text responses thanks to Alexa's out of the package framework. Of the five types of speech recognition options (speaker dependent, speaker independent, discrete, continuous and natural) we went with the most advanced form which is natural language. These types of systems can understand continuous speech at a normal rate of speaking. We also included an intent detection component for when the nature of a user's question or request is unclear or incomplete the system can ask verbally for clarification.

In the end we accomplished our goal of being able to post from multiple accounts while using our voices to control the application. However as mentioned below we encountered many problems some of which still need to be fixed and fine tuned. However with the time given to us and our level of expertise in this matter we are satisfied with the work accomplished so far and are eager to continue working on this application to try and polish it up.

PROBLEMS ENCOUNTERED AND SOLVED

Having an only voice user interface at first proved to be very difficult especially when trying to sign in to accounts. Obviously people don't want to say their email and password outloud so we could not implement that type of signing in. This led to the idea of using a hybrid VUI and GUI so we could present the user with an interface that would be more private. We wanted to display a hyperlink that they could click so they could log in, however hyperlinks are currently not available to developers using Alexa even though Alexa is capable of displaying them. Our next thought was to create a QR code and display it to the user so they could scan it on their phones and be redirected to the login page. Creating the QR code was easy enough but trying to display it along with the "buggy" testing console proved to be difficult and quite a huge learning curve that we did not have time to completely overcome.

Another major problem we encountered was that we spent most of our time just moving what we had from Sprint 1 to trying to make it work with Alexa. Since our project was done in python and we were going to use Alexa with Python we thought it wouldn't be too difficult however, the way we received, processed, stored information and performed actions were so different in both

environments that it consumed most of our time to get everything working. We didn't have much experience with implementing VUI's and when we realized we had to have a hybrid, we then saw that we would have to be learning two different types of new skills, not just one. Not being able to meet up with each other and talk about different ideas and solutions really slowed down our project as well.

Minor problems included everyone not having an Alexa enabled device which we then found out we could use the Amazon Developer Console to test our skills. We also had trouble using OAuth to sign in but we quickly figured out how it worked and what we needed. Some delayed problems included having to wait to get a Twitter Developer Account approved followed by having to wait to get an Amazon Developer Account as well.

SUGGESTIONS FOR BETTER APPROACHES TO PROBLEM/PROJECT

In hindsight we feel that the modular approach was not the best approach since we started our project only with Python and then had to refractor it all to make it work with Alexa. Instead of doing a stand-alone twitter application in Python and then attempting to integrate Alexa we should have started right off the bat with Alexa. Therefore our suggestion would be to start in the development environment that you know is best to use for your application that way you won't have to lift and shift or refactor your code somewhere else down the line.

As mentioned before, we didn't have time to implement the visual component of our application to display a QR code to the user so they can log in and that was because we were so focused on trying to create an application that didn't require any visual interaction that we realized too late that we actually had to have some type of visual feedback throughout our project as well. Another suggestion would be to think about what the user is going to be doing with your application as well as how you want to represent that information to the user so you can know what type of tools you will need to use.

SUGGESTIONS FOR FUTURE EXTENSIONS

There are numerous visual challenges that are ripe to be tackled. In its current state the voice activation component is already helpful to people that are visually impaired. However as we mentioned above the visual components still need work since we not only need a Voice User Interface but also a Graphical User Interface

There are also many additions that could be added to the application itself such as being able to display links that people tweet out or play videos that they have tweeted. Another extension

would be to be able to log into different social media accounts such as Facebook or Instagram and perform similar actions as we can do from Twitter.

APPENDICES/REFERENCES

http://docs.tweepy.org/en/latest/

https://developer.twitter.com/en/docs/basics/getting-started

https://developer.amazon.com/en-US/docs/alexa/hosted-skills/build-a-skill-end-to-end-using-an-alexa-hosted-skill.html

https://developer.amazon.com/en-US/docs/alexa/custom-skills/display-interface-reference.html

https://developer.amazon.com/en-US/docs/alexa/alexa-presentation-language/understand-apl.html

https://developer.amazon.com/en-US/docs/alexa/custom-skills/delegate-dialog-to-alexa.html

https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a/obtaining-user-access-tokens

https://pypi.org/project/PyQRCode/

https://developer.amazon.com/en-US/docs/alexa/devconsole/test-your-skill.html#h2_register

https://docs.microsoft.com/en-us/azure/cognitive-services/luis/luis-concept-utterance

https://docs.microsoft.com/en-us/azure/cognitive-services/LUIS/luis-concept-entity-types

https://docs.microsoft.com/en-us/azure/cognitive-services/LUIS/luis-concept-intent

https://docs.microsoft.com/en-us/azure/cognitive-services/LUIS/luis-get-started-create-app

https://docs.microsoft.com/en-us/azure/cognitive-services/LUIS/luis-get-started-get-intent-from-rest

https://pypi.org/project/PyAudio/

https://docs.microsoft.com/en-us/azure/cognitive-services/speech-service/quickstarts/speech-to-text-from-file

https://devcenter.heroku.com/articles/dynos

https://devcenter.heroku.com/articles/clock-processes-python

https://docs.microsoft.com/en-us/azure/cognitive-services/speech-service/quickstarts/text-to-speech-audio-file

https://pypi.org/project/playsound/