# Design

Based upon our previous section detailing the specific problems we would like to address; this section will contain the design and implementation phases of the project based upon our previously identified objectives. The design phase will focus on detailing the individual modules and how they can interconnect to form a working solution. The implementation phase will look at the final product, as well as any issues encountered along the way that may have differed from the plan.

The general design ethic for the project is for it to be modular. This allows for the replacement of entire services with others as long as the data output remains consistent with other modules. For example, replacing Microsoft’s LUIS natural language processing service with Facebook’s Wit.ai would not require an entire system rewrite, but would only require the replacement of the Natural Language Processing module.

As an addition to the modular design ethic, the project is intended to be a framework that can be refined, worked on and added to over time. It is not intended to be a complete monitoring system for a network – rather it is a framework with a various proof of concept and example uses pre-programmed.

It is intended that the application be semi-autonomous in nature, able to be extremely low maintenance once configured. The application will have two operational modes:

**Interactive mode** is a state where the user leads the conversation. A user will query the app, which will cause the app to produce a result and serve it back to the user.

**Monitor mode** is the applications default state, where it is analysing incoming information from sources such as servers and comparing it to known thresholds. If a threshold is met, a user responsible for that threshold is informed – The application is then in interactive mode once a user replies.

### Data Sources

For our testing we will only be taking data from a production Linux Ubuntu operating system. This will include the following data sources:

**Apache HTTP Server Access Logs** show records of pages served and files loaded by the webserver. This can be valuable information if formatted correctly. This information will be used to generate statistics.

**Apache HTTP Server Error Logs** show records of all error conditions reported by the server, which in some cases will need urgent attention.

**Authorization Logs** track usage of authorization systems such as *sudo* and remote logins over SSH. These can be used for both statistical purposes and for showing login attempts.

**Login Failures Log** is designed to be non-human-readable, and contains all login failures. This may be better for machine parsing than the authorization logs.

**Last Logins Log** is also non-human-readable, and shows the last login of users.[1]

However, there is no reason why extra modules could be added with more commands and to accept more sources.

### Other Commands

Remote administration means being away from tools, so adding tools that are often in an administrators’ arsenal to the application can be very useful in troubleshooting. Having remote access to a machine inside the network means that certain commands can be piped to the user:

**Ping** can be used to ping both internal and external resources, telling the user if that resource is online and what its connection is like.

**Traceroute** shows the user the route packets are taking to a server.

**Nslookup** allows a user to look the DNS information of a resource either internal or external as reported by their DNS server.

**SSH** allows an interactive shell to another computer. This may or may not be possible with our planned setup and will have to be tested during the implementation.

## Security and Other Considerations

Security is a major point in this project, as this application may have access to logs or passwords for servers. To address this issue, we will be using the security offered by the various chat services by associating access to the bot with access to instant messaging services.

In our case, BotBuilder provides a user ID along with every message. We can compare this user ID to a hardcoded whitelist that allows a user access to functions in the bot if they are whitelisted. A user should take steps such as two-factor authentication on their IM account in order to protect it, and by extension, any data the project may be able to give out.

It would be possible to add more security considerations to the modules such as passwords, but this did not seem necessary for this example framework.

## Modules

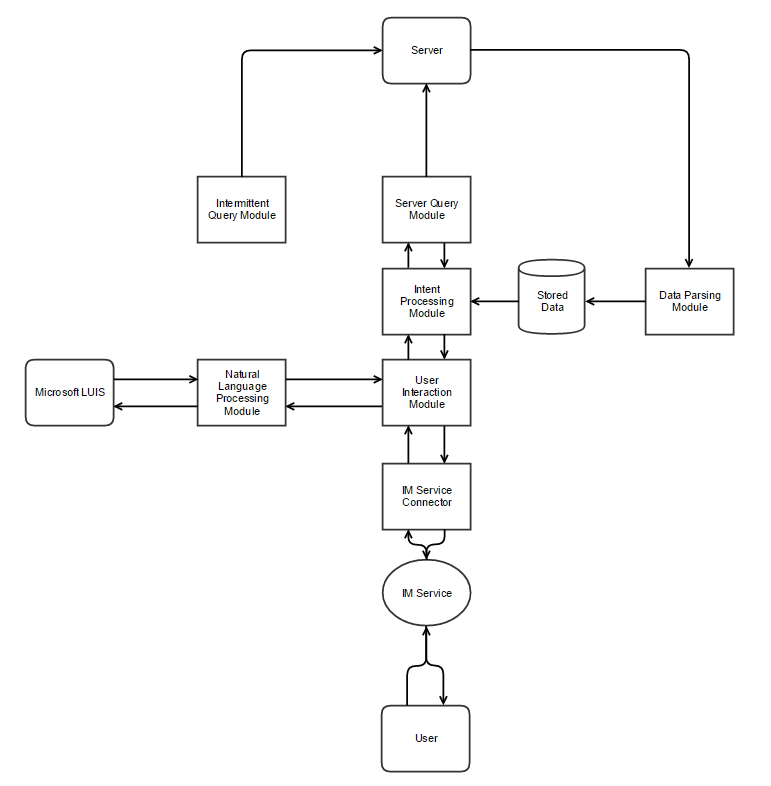


Figure 1 - Technical Prototype Flow Chart

Our initial plan is shown in Figure 1 - Technical Prototype Flow Chart, and shows each individual module. The following section will go into detail of each module and discuss its uses, as well as how they will interact or be swapped out as necessary.

### IM Service Connector

This module connects the User Interaction module to the IM service. This simply manages the connection to the instant messaging server, be it IRC, Jabber, Telegram or any other. As we already know the Microsoft Bot Builder[2] acts as a proxy between many IM services and our application, some of the work here is done for us. However, in the case that a client wishes to use a different service, our User Interaction Module must be fed data in the same format.

Bot Builder does the work of connecting to the IM service and then forwarding user data from the IM to our API endpoint. Following Bot Builder conventions, we will use their library to open a Restify[3] web service on our server, which will receive the data and handle replies to the Bot Builder server. We can expect the data to be very similar irrelevant of what IM service the user is using. This data is handed directly off to the NLP service without modification, and contains the session information such as the user’s ID, what conversation it is part of, other data required for a reply, and the users message.

Bot Builder was chosen due to it’s also modular nature, it allows for different NLP back-ends to be replaced with relative ease as well as not using one entirely. It also allows for many different IM services to use the same bot back-end, so as not to limit the user in their choices and it can be easily integrated into existing company instant messaging systems.

### Natural Language Processing Module

This module takes the formatted IM messages and sends them to an NLP service, and formats the information to be understood by the User Interaction Module. For the sake of our testing, this module will be tuned to Microsoft LUIS[4] running on an Azure instance.

Intents are what the user ‘intends’ to do. These intents, as well as what triggers them are defined in the LUIS NLP website, and will be planned later.

At program startup, the NLP module queries the User Interaction(UI) Module to get a list of known intents. The NLP module then generates templates to allow it to format the information in a way the UIM understands. No matter what service the NLP module connects to, it will format the information the same way. This generation of functions means that all data coming to the UI module will be in a constant format irrelevant of the intent, and as long as the NLP module sends that data in the same way it is irrelevant which NLP service is in use. We preferred this as it meant significantly less hard-coding of intent functions within the NLP module, making it easier to keep consistency in the event that the NLP module is replaced with one for a different service.

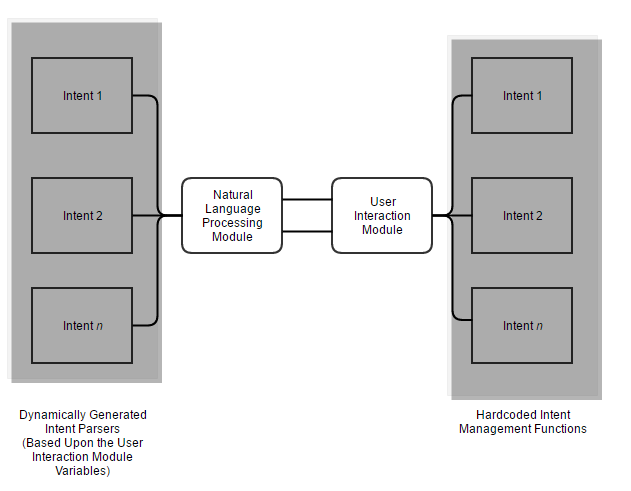


Figure 2 - Showing the relationship between the NLP Module and UI Module

Shown in Figure 2, the intent management functions in the UI Module are hard-coded. The NLP Module then dynamically generates the functions required to handle these incoming intents. The functions here also handle the reply to the user, so it is important to make good use of callbacks between the two modules so the session data is not lost.

Upon receiving a message, the NLP Module will receive the data and immediately send it to our NLP Service (LUIS) to be analysed, which will then be returned and associated with a function related to the intent that the NLP Service deemed the user was trying to activate. The function will then strip the arguments from the formatted message provided by the NLP Service into a more generic format, shown in Figure 3.

This generic format will contain the values of all arguments in the users message. For example, if the user were to type “Show me the log files for Apache on the Ubuntu server”. The intent would be ‘log’, and the two arguments would be ‘Apache’ and ‘Ubuntu’. The NLP Module sends this data, along with the session data in case it is needed later, plus a callback to the UI Module to be handled. The callback, when triggered by the UI module, will accept three arguments. The first argument will be either a string or array – If a string is sent, it is sent directly to the user. If an array is sent, it will be looped and each part of the array will be sent as a separate message to the user. The second argument specifies to the function that the dialog should be handed off to another function that handles multi-message dialogs.

{

**"application1"**:"value1",

**"argument1"**:"value2",

**"server1"**:"value3",

**"argumentN"**:"value4"

}

Figure 3 - Generic argument format

The function that handles multi-message dialogs will be hardcoded, but still will call the intent handler in the UI module for instructions with a specified keyword ‘Convo’, which states that the conversation has entered a multi-message dialog state. This function will loop, receiving data based on how many times it has looped from the UI module with what to send to the user. The UI module may call back with a variable dictating the type of message to send the user, either a string, a prompt or an exit. A string will simply send a message to the user and loop again. An exit will exit the dialog.

However, a prompt is a special situation. It first prompts the user with a question, and then hands off to another handler function using Bot Builder’s Waterfall method, which allows for conversations to be handled differently depending on the flow of a conversation. This final function then adds the users response to an array, and loops back to the original dialog function in order to call and send that data to the UI Module for continued processing.

### User Interaction(UI) Module

This module transports incoming information from the Natural Language Processing module and forwards formatted data of the users’ intent to the Intent Processing Module. This module also formats data intended for the user. In some cases, Intent Processing isn’t necessary and a simple reply can be managed by the User Interaction Module. This module is entirely based on the ‘Interactive Mode’ idea, where it is only in use when the user is directly using the application.

This module can be left static between changes of the NLP and IM modules, as it relies on those modules handling formatting and simply has a list of known intents and how it reacts to them.

The UI Module acts as the middleware between the raw data of the NLP Module and the Intent Processing module. It contains a hard-coded list of intents which should be identical to those specified and trained on the NLP service website, which will act as the basis for the NLP modules functions. As by this point the intent has already been figured out, and the arguments have been correctly parsed, the UI module is where the real human-written responses start to take place.

In the UI module will be an intent handler, containing a function for each known intent plus conversation function for any intents requiring question and answer type conversations. Each function will accept 2-3 arguments, the first being the callback to the NLP Module that sends messages to the user, the second being any arguments that came from the user and formatted by the NLP Module, and finally the session variable which contains all the user data. These functions will then either then call the callback directly to reply to the user with a predefined variable with minor processing such as picking out the user ID and sending it back, or will pass on data to the Intent Processing module with another callback to return back when finished. The flow of data is shown in Figure 4.

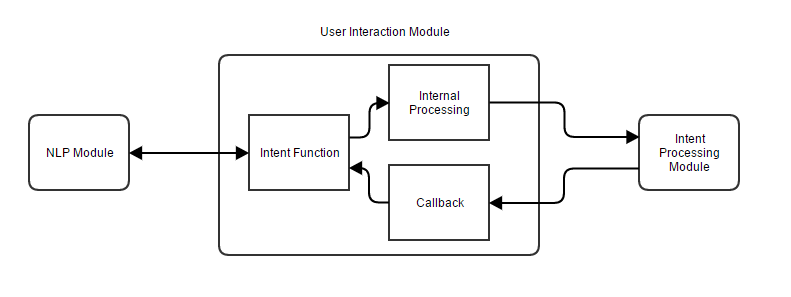


Figure 4 - User interaction module data flow

As an example of a simple intent, such as the ‘help’ intent, the UI module would ignore any arguments and simply call the callback with a simple, hardcoded reply. In the case of an intent requiring responses such as the ‘SSH’ intent, the function would immediately call back to the NLP module requesting a change to the conversation handling function. The conversation handler then loops, incrementing a counter that calls the SSHConvo intent, getting a different prompt or text response each time until all the required information is obtained. This seems like a reliable method for dynamically generating conversations without a complex hardcoding system.

To prove this framework is viable, we will implement several simple responses and 1-2 complex responses. The list of planned user interactions, and therefore intents, are:

* None: Will respond that no intent has been detected.
* Version: Will respond with the systems version.
* Help: Wil respond with a hard-coded ‘help’ text.
* Identify: Will respond with identifying features of the user and the current conversation, such as user ID and platform in use.
* Ping: Will tell the user if a given IP address or endpoint is online and replying to pings.
* Traceroute: Will perform a traceroute to a given endpoint, and reply with individual messages to the user with each hop.
* SSH: Will open up a live shell session with a given server, asking for username, password, and port and then forwarding messages to and from the server.
* Logs: Will connect to a hard-coded server and allow the user to choose (with natural language) between several different logs, before tailing the chosen log to the user.
* Associate: Will associate this conversation with a hardcoded query, which operates on a hardcoded timer. If an ‘alert’ is triggered by this query, the user will be informed and the alert data will be sent to them.

It is believed that these intents are sufficient examples of what the framework can do and how it can be worked with.

### Intent Processing(IP) Module

The IP module manages all complex processing of information, and is called upon by several modules. At its core, the IP module receives data to the required function, processes information and then returns that information via callback to whichever module requested it. Not all intents require extended processing, but it makes sense to separate the majority of the processing that happens away from direct user responses.

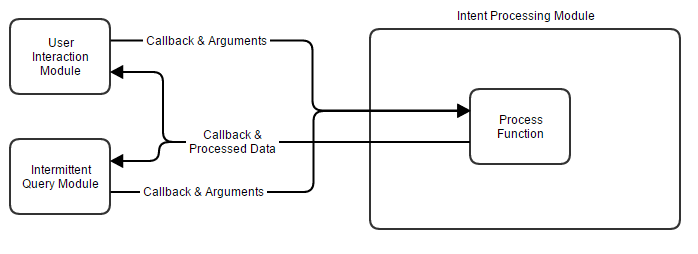


Figure 5 - Intent Processing Data Flow Example

Figure 5 shows an example of the flow of data from two modules to the Intent Processing module. In this case, either module can request a function from the IP module as long as it has the correct arguments and provides a callback, the IP module will then reply with the processed data.

Referring back to our list in the UI module planning, only the Ping, Traceroute, SSH and Associate intents will require processing in the IP module. Additionally, there will be a ‘helper’ file downloading function, which will download data from at least SSH servers for log retrieval. Each function will require specific data from the UI module, and will return specific processed data.

Ping can expect simply an address and a callback. It will send a request to the given address and call back whether the address is alive or not. This is our simple test to show that the framework is able to take a command, process it, and return information to the user. Traceroute, similarly, can expect only an address and a callback, returning the call for every single hop. This shows that the application has the ability to return multiple times to send the user better formatted data. Associate will expect three arguments: Some data referring to which query the user intends to associate the conversation with, the session variable which allows the function to save it against the query, and the callback in order to inform the user of the success or failure. Finally, the SSH function will be able to take the previously compiled connection data to connect to a server, and receive new data once a connection is established to forward messages from the user to the server, as well as reply with server messages through the callback. SSH will be a complex example of multi-stage conversation and will be a challenge to see if it is even possible with the planned configuration.

### Data Parsing Module

This module parses incoming data from servers into data more easily readable by other modules, and attempts to keep data consistent in the case of module replacement. The Parsing module will have functions added as required, and will primarily use Regular Expressions and replacement functions to make data more consistent. For example, the Server Query module may receive log data from a server containing line by line date and login time data, it could then send this data to the Data Parsing module to format this data into an object with a more Javascript friendly date system plus having the data predictably located in a variable as part of the object tree. However, this function would have to be hardcoded in as part of the Data Parsing module.

This module will be used when we deem necessary while developing, rather than being rigorously planned at this stage. An example data flow is shown in Figure 6, where various modules send unformatted data to the Data Parsing module, and the module returns back formatted data. This makes it significantly easier to maintain cleaner code when it comes to managing data and text by keeping all major text manipulation in one place.

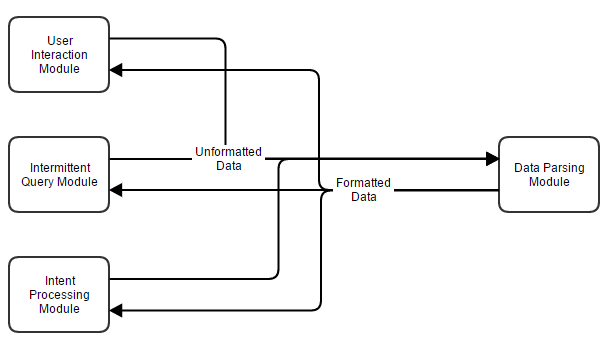


Figure 6 - Data Parsing Module example data flow

### Server Query Module

The Server Query module manages queries to and from servers. It will have a function for each major file transfer protocol, such as SCP and FTP, which will both accept connection information and file locations to be downloaded. These functions will regularly be called by the Intermittent Query Module in order to make a request to a server, and this data will parse through the Data Parsing module to enter a consistent format.

In our example intents, the only functions that actually do regular queries to server are the ‘log’ intent and the Intermittent Query Module, which contains a hardcoded list of queries the application does regularly. The Server Query module will be quite simple in practice, just being functions able to download files using data provided by other modules.

An example function for SCP would import an SCP downloading Node module, and expect an object containing the server address, file location and a local location where to save that file, and would call back to the function that called it with the downloaded file location. This function can then retrieve the data and send it to the Data Parsing module if required.

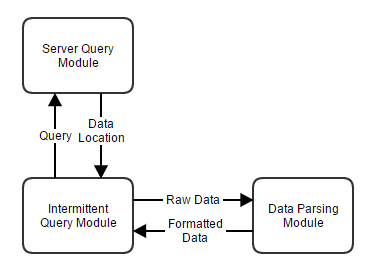


Figure 7 - Intermittent Query, Server Query & Data Parsing Interaction Example

As shown in Figure 7, the Intermittent Query module will sent a query to the Server Query module containing required data, such as credentials and address information, and the Server Query module will call back with the location of the data. The Intermittent Query module can then read that data and send it to the Data Parsing module if it needs to.

Separating Server Queries from the other module makes sense at this stage as it can be highly customised. It is likely that server queries will be written from scratch depending on differing systems and are unlikely to be shared between installations, unless a large number of generic queries can be created. It is important therefore that the Server Query module be very generic in its structured responses, so it can be easily replicated when creating more queries. Figure 8 contains a snippet of the proposed structure of data output from the module. Very generic, containing just the name of the query (for confirmation), the data requested and any other information just in case it is relevant.

{

**"name"**:"serverQuery1",

**"data"**:"values",

**"other"**:"OtherInfo"

}

Figure 8 - Server Query Example Output

### Intermittent Query Module

This module queries data from servers based on a timer, and is related primarily to the “monitor” mode of the program. Server queries will be stored as objects inside of an array, and hard coded into the program.

An example would be an intermittent query that checks, every minute, the ‘auth’ log from a Linux server. In this case, we would need to SCP to the server every minute and download the auth log, then parse it for information we cared about such as failed or successful logins, being careful to not include information of our own connection to get data, and then inform the user if it meets certain criteria. This also means the user must associate their conversation with the query using the ‘associate’ intent before receiving alerts.

{

name: 'LinuxAuthLogs',

server: 'linux', *//Defined in the servers file and looked up later*

type: 'logs',

application: 'system',

specific: 'auth’,

delay: 60000, *//How often between queries*

run: **function**(callback) {

*//Call the server query module to get the file*

serverQuery.scp(**this**.server, **this**.type, **this**.file, **function**(fileLocation) {

*//Call back with file location and read the file*

readfile(fileLocation, **function**(file) {

*//Call back with data and parse it*

**var** parsedFile = dataParse(file);

*//Inform the user using their saved session of the alert and its details*

informUser(parsedFile, userSavedSession);

})

});

}

}

Figure 9 - Example layout of an intermittent query

Figure 9 shows an example layout of an intermittent query in pseudocode, detailing all information related to the server aside from credentials, which is saved in the servers file shown later. It contains a function that can be called, which handles the gathering of data using other modules and then finally responding to the user. The name is used to associate users, where a user can simply type ‘associate LinuxAuthLogs’ to receive alerts for that query. The function itself will be called by a separate part of the module, which loops through queries on start-up and configures the intervals at which the queries will be run automatically based upon the definitions above.

### Stored Data

Data that may be required to be stored between sessions will be stored on disk in the form of JSON. Data that may be relevant to store include conversational sessions so users do not have to re-associate themselves with a query, and server data such as log history. If data needs to be stored, we can simply call JSON.stringify() on it and save it to disk, then load it to a variable on startup. This will be done as needed and is not a directly planned occurrence.

### Server File

The server file is simply a file containing objects of known servers, their logins and known paths to logs. These objects can be navigated by the Server Query module to obtain information required to retrieve logs and other data.

{

"linuxServer": {

address: 'linux.local',

protocol: 'scp',

port: 22,

username: 'root',

password: 'toor',

logs: {

system: {

auth: {

path: "/var/log/",

file: "auth.log"

}

}

}

}

}

Figure 10 - Server object example

Figure 10 shows an example of a simple server layout. A correctly configured Intermittent Query would be able to access the full path of the log file, as well as the address, protocol, port, and login by simply following the object. For example, *servers[this.server][this.type][this.application][this.specific].path* when referencing from the Intermittent Query module would give the module access to the path for that server.

### Setup File

The setup file will not be included in the final project paper as it contains private information such as passwords to servers, private keys for the LUIS application and lists of whitelisted IP addresses. This seemed necessary to address as the mysterious ‘setup’ variable is reference often in the code appendices but never makes an appearance itself. It is simply an object containing information not directly relevant to the code.

### LUIS

The choice of using the LUIS NLP service between others was a tough one at first, but it was decided that it in fact did not really matter which NLP service was chosen. As the project is deemed to be a framework rather than a fully defined software application, LUIS can theoretically be replaced with any one of many NLP services as long as the NLP module is tuned to output the same information.

However, LUIS ended up being the chosen service for testing due to the project lead having prior experience with it, and it seemed fairly fast to train and export learning data to a remote Azure server. The choice of Azure was easy, as it naturally connects to LUIS and Bot Builder, allowing for a much-simplified approach to the project.

The plan for training LUIS is to come up with a base phrase, such as “ping {address}” and then working off of that manually to add extra training phrases, like “Please ping the address {address}” and “Ping the IP address {address}”. Training will be continuous as the project is being developed until testing starts.

# Implementation

This section follows on from our design section to show the actual development of the application

## Forwarding Ports

The first step in producing the application was met with the issue of forwarding ports. Because Bot Framework expects you to have a HTTP endpoint for it to send information to, and one had not yet been set up on a production server, we have to open ports locally. This is a problem, as our test environment does not allow for port forwarding.

To fix this we used an application called Ngrok[5]. Ngrok allows us to expose local servers to the internet from behind a NAT when we are not able to port forward. Using this tunnel allowed us to continue our test deployment. At this point, we have an internet-connected chat bot that we can test with the Bot Framework Emulator[6] Shown in Figure 11 is a basic ‘Hello World’ response.

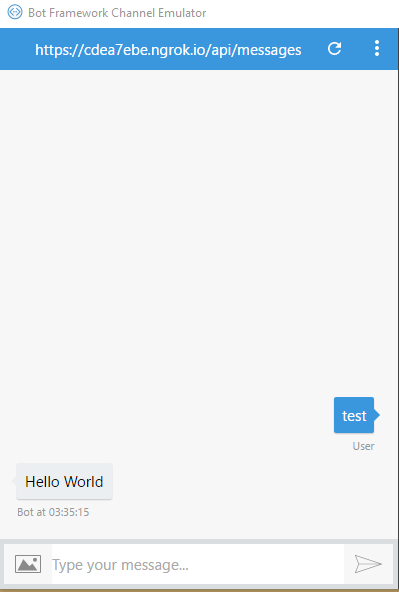


Figure 11 - Successful remote call and response

## IM Connection

Next we need to connect our application to an IM service. In this case Telegram, as it has an abundance of formatting styles and rich messaging capability that can be used to make data look better for the user, if required. This is as simple as registering a bot with the Telegram service by messaging the ‘BotFather’ user and following its instructions. Entering the given HTTP API tokens into Microsoft’s API allows the bot to connect directly to the Telegram service, so we can access our bots commands from within Telegram, shown in Figure 12.

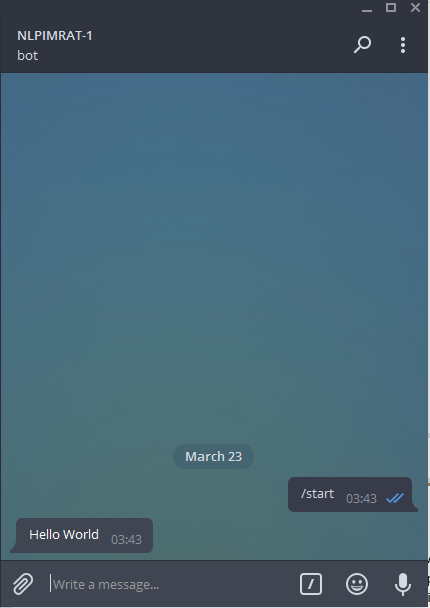


Figure 12 - Call and response from within Telegram

## Implementing Natural Language Processing

NLP must be added early, even prior to training being started. It is important that the module is at least configured so it can be expanded upon easily. All programming for Luis.ai is done via their website, but when finished is hosted on an Azure endpoint. After setting up a basic NLP app and publishing it to Azure, we have access to the ‘intents’ that we configured. Intents are defined as “What the user intended to do based on what was said”, and data is formatted in a way that is understandable in code rather than in natural language. It also separates the users intent from any potential arguments in their message, as well as giving them names, making it much easier to pick out values and arguments from messages. These intents allow us test our connections between Azure, BotBuilder, and Telegram.

In this case, some very simple training had to be added for Luis to understand the ‘ping’ command.

Training is a fairly simple process, but can become very intensive. To add a command first you must break the command down into its most bare components. In this case, the smallest possible rendition of the command is “ping {$address}”. But in natural language, it could be “Could you please ping the address {$address} for me?”. We must think up as many different forms of this as possible to train LUIS into understanding what it receives.

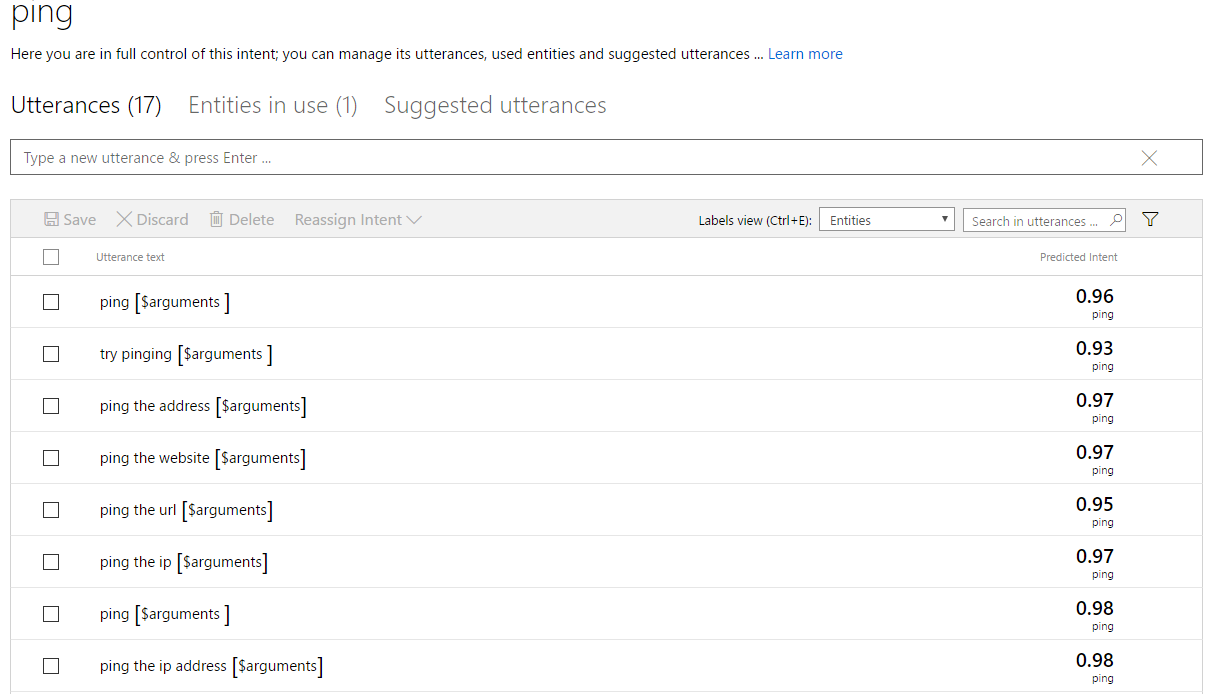


Figure 13 - Training the 'ping' intent

We can quickly come up with some basic ‘utterances’, which are defined as the different ways of saying the command in natural language. Luis then attempts to parse it based on what it knows so far. In Figure 13 we can see on the right hand side that Luis successfully guessed that our utterances were related to the ping intent with a 93-98% accuracy. With regards to the *[$arguments}*, this variable is manually set to begin with, but after further training Luis is able to guess where the arguments will appear in a given utterance.

In Luis, we named our intent “ping”, and we can easily tell if a user’s intent was to ping by the response we receive from LUIS. For example, if it DID match ping, we can tell the user what arguments we managed to pull from their command. Failing that, we can give a default error message. At the same time, we can also program in the intents for querying the version number, and some templates for a help system. At this point the first rendition of the dynamic NLP module started taking shape, an early and simplified example of which is shown in Figure 15, where the known intents are being looped and a simple argument parser is put in place.

To confirm that the NLP Module template generator was working correctly, we created a very simple example of the ‘ping’ command. We know that ‘ping’ is an intent that might be returned by the NLP module, so just for testing we can reply with a response that shows we know the user requested the ping intent, and what address they were looking to ping. This snippet of code is shown in Figure 14.

*//Grabs known intents from user interaction module, creates a template handler for each by looping through them*

**for** (**var** i = 0; i < knownIntents.length; i++) {

**var** match = knownIntents[i];

**var** intentGenerator = `

dialog.matches('*${*match*}*', [

function (session, args) {

console.log("Handling intent: *${*match*}*")

var arguments = [];

//Simplifies arguments for the intent handler

for (var i = 0; i < args.entities.length; i++) {

arguments.push({

name: args.entities[i].type,

value: args.entities[i].entity

});

}

//Sends intent plus data to the UIM for processing, then returns back with the

//text for the user

intentHandler['${match}'](arguments, function(response) {

session.send(response);

}); }

]); `;

eval(intentGenerator);

}

Figure 14 - NLP Module template generator

*//List of known intents*

*//This allows us to have a static list of expected data even in the case that the NLP module changes*

**var** knownIntents = ['ping'];

*//Handling of each intent, NLP module agnostic*

**var** intentHandler = {

ping: **function**(arg) {

console.log(arg);

**var** response = "You wish me to ping " + arg[0].value.replace(/\s/g, "");

**return** response;

}

};

Figure 15 - User Interaction Module(UIM) example code

It turns out that Luis separates punctuation from arguments when extracting them from text, so the address “google.com” becomes “google . com”. Thankfully this can be easily fixed by deleting whitespace before sending the data off for processing, and this becomes a common theme.

## Implementing Commands

Now that we have confirmed that Luis can be trained and that the bot can receive not only intent data, but also argument data, we can start to actually implement our intents. Our intents are separated into ‘basic’ and ‘advanced’ commands, basic being defined as not requiring intervention by the Intent Processing(IP) module and advanced being mostly handled by the IP module. At this point we are only really interested in the User Interaction and NLP modules, so basic commands will come first.

Our plan states that we have 9 intents, the basic ones being ‘None’, ‘Version’, ‘Help’, and ‘Identify’. These all came together incredibly quickly, as they are all planned to be functions that respond with either text provided by the user or hardcoded text inside the User Interaction module. First, the NLP Module function generator was modified to send the ‘session’ argument along with arguments and callbacks to the User Interaction module. Then a simple function was made for each, returning their text to the NLP Module for sending to the user. The ‘none’ function simply replied that no intent was detected, ‘version’ took the version number from the package file, ‘help’ replied with a predefined help text, and ‘identify’ took the Channel ID, User ID, Address ID and Username from the session variable and returned them to the user for debugging. Figure 16 shows examples of the help, none and version intents.

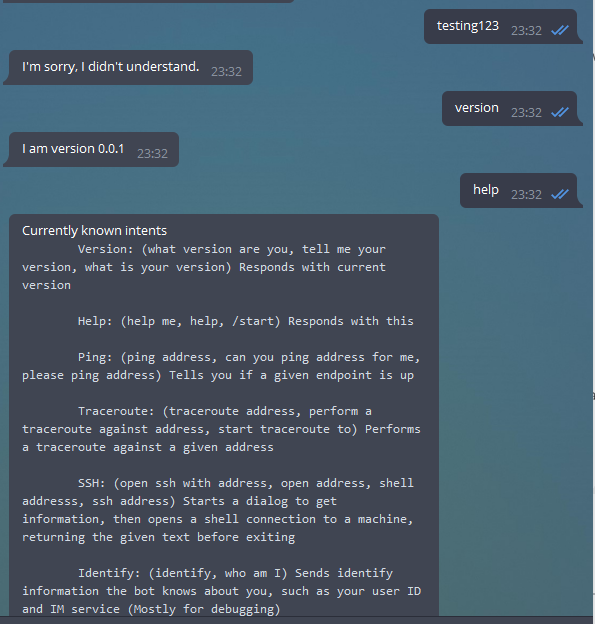


Figure 16 - Initial basic intent processing

The more complex commands required some extra work, and the implementation of the Intent Processing module.

### Ping

Instead of using the systems in-built ping command by piping user input straight to a console, which would make it ripe for command injection, we instead opted to use a powerful but small library simply named ‘ping’[7]. It sanitises inputs and simply checks if a given IP is online or not. Using Node’s in-built DNS tools combined with *ping* allows us to quickly check if a given address is ping-able, as well as using our previously trained NLP service, we can set up a small alive-or-not host checker.

The reason behind doing a DNS lookup prior to pinging is because the ping library, for unknown reasons, does not support the pinging of Windows hosts by computer name. The simple fix was to use DNS lookup to first obtain the IP address of the machine, and then have the ping library check if it is online.

The Intent Processing module was created similarly to the User Interaction module, with an object containing all of the functions that would be called. The ping function accepted the address, did a DNS lookup upon it (in the case that the user provided an IP address, the DNS lookup will just reply with the IP address, so this isn’t an issue), pinged the address, and had a simple response telling the user if the host was alive or not and what IP address the address resolved to. It then calls back down the chain, first to the User Interaction Module.

The User Interaction module function for the Ping intent is simple, it takes the argument from the NLP Module, replaces any spaces, and sends it to the Intent Processing module. Once the IP module calls back, the UI module also calls back to the NLP module, which finally sends the message that was passed down the chain to the user. Figure 17 shows an example of several successful and failed pings, and it is interesting to note that the final phrase, “Please try to ping *x*” was not directly trained into Luis, and it in fact correctly guessed the intent and argument based on its training thus far.

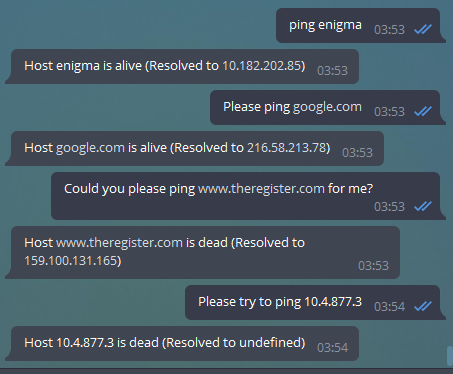


Figure 17 - Simple host PING example

In terms of the actual training, Figure 18 shows the 14 base examples used to train Luis, where ‘*x*’ refers to a random IP address or website address. After even those few examples, Luis was guessing intent with a 95% accuracy, which was considered close enough for testing. At any point, more training can be added to improve the accuracy further.

ping x  
ping the ip address x  
ping the address x  
can you ping the domain x  
can you ping the address x  
please try pinging x  
please ping x  
ping the url x  
ping the ip x  
try pinging x  
can you ping x for me  
can you ping the website x for me  
is x online  
is x up

Figure 18 - Base training examples for the 'ping' intent

### Traceroute

The traceroute intent was achieved with a similar library, simply named ‘traceroute’[8]. This library takes an address as an argument, and returns an array of ‘hops’ containing each IP and three millisecond timings of the ping to that hop.

As with the ping intent, the User Interaction list of known intents was updated with the new intent, and a new function made to handle it, which simply replaces whitespace in the argument and hands it over to the Intent Processing module. The IP module function for the traceroute intent takes the address and calls the traceroute library, which then returns the hops array. We then cycle through this array to clean up the data and also average the milliseconds for the pings to each hop, it then calls back all the way to the NLP Module for each hop, sending a message each time it loops. This function is an example showing that the framework is capable of managing multiple callbacks to the user.

For training LUIS, we used much less training compared to the ping intent, primarily due to the difficulty of finding alternative ways to say ‘traceroute *x*’. We ended up with only 6 examples to train Luis with, but this seemed to be enough for Luis to claim a 97-100% accuracy.

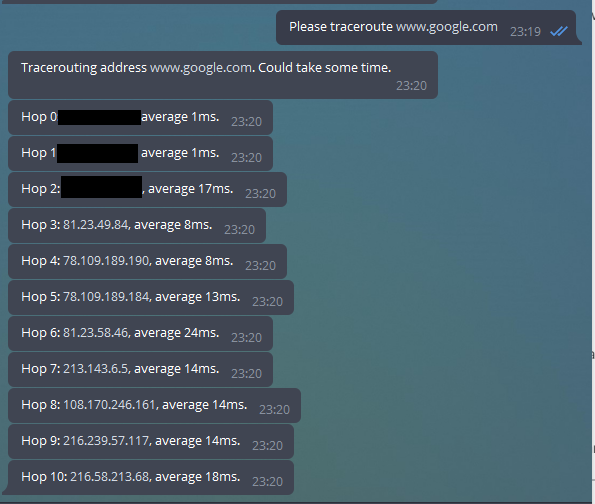


Figure 19 - Successful example of tracerouting

Shown in Figure 19 is the response to a traceroute command, with the calculated average time in milliseconds afterwards the IP address of the hop.

### SSH

The implementation of a live shell session was much more complex than any intents so far, and was the first and only intent to require a conversational dialog with multiple questions and answers. Unfortunately, due to issues with how Bot Builder managed callbacks and data, the implementation did not fully follow the planned design and several key features had to be left out, which will be described later. While it may certainly be possible to do exactly as planned with Bot Builder, it would have required major changes to how the dynamic generation of functions in the NLP Module worked, which would likely have ruined the modular aspect of the framework. We chose to instead reduce functionality of the SSH intent rather than stray from the overall project goals.

To start with the implementation, we first had to modify our existing generator in the NLP module to be able to handle a variable telling it to swap functions, as per our design. We then had to make another function which handled question and answer style dialogs. It was fairly easy to follow the design up until it came to maintaining data while looping, and as global variables were not a clean option and Bot Builder had no easy way to maintain data between dialog changes, it was decided that we would try to include relevant data and saved data when calling the dialog again.

This idea ended up making the code fairly hard to work with as it made it quite messy, but with little other option it continued to be developed on in the User Interaction module, where a function was made containing an *if* statement that would reply differently dependant on what iteration we were on. Lastly, the Intent Processing module gained a function that connected to an SSH server, streamed incoming data to the user, then disconnected.

It was at this point it came to realisation that there was no safe and clean way to pass further variables to the server and receive the response, and so it was decided to not complete this function. However, it still operates fine up to this point, as shown in Figure 20, but unfortunately the ‘next command’ simply exits the connection.

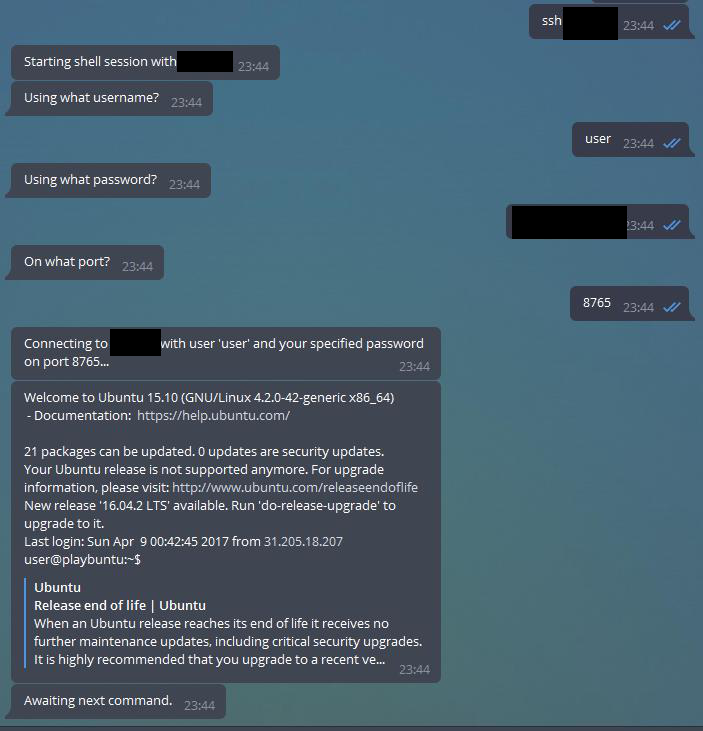


Figure 20 - Dialog-based SSH connection

### Log Handling

Other features such as the alert association features cannot be completed without functions that can gather logs, so this is the obvious next step.

Getting the right combination of words to be able to traverse our planned server object list was quite difficult, but eventually the approximate format “$*application $arguments* logs $*server*” was decided on. This by itself doesn’t make a lot of sense, but when expanded into “Get the System Auth logs for linuxServer” it makes a bit more sense. Using Figure 10 as an example, we can actually traverse this object using the collection of arguments a user sends by extracting the variables from that message and applying it to the servers variable in the format *servers.$server.$arguments.$application*, or using our message as an example *servers.”linuxServer”.logs.”system”.”auth”*, and this returns the path of the log file the user would like.

With this decided, around 10 sentences were trained into Luis following the same format, as it proved difficult to be able to swap variables around (for example *$server*, *$application* then *$arguments*) without Luis starting to assign variables incorrectly. It will be interesting to see during testing to see how badly this may affect understanding.

After training was completed, the User Interaction module was updated with a logs function that used the above object traversal technique to get data such as address, login, port and filenames from the server object, which it then builds into a cleaner object to send to the Intent Processing object. The Intent Processing module then gained it’s getFile function, which accepts the above object and uses it, plus the any-file[9] library, to create a download link for the given log file. The last 20 lines of this logfile is then read and sent back down the line to the user with minor text processing.

It was at this point that it was realised that the plan for the Data Parsing module seemed fairly pointless. All text modification to this point has been fairly minor and in-line, and so it was decided that an extra module just to manage it would be extraneous. There is no reason why it could not be added later as an addition module, but for the sake of our application it did not seem necessary.

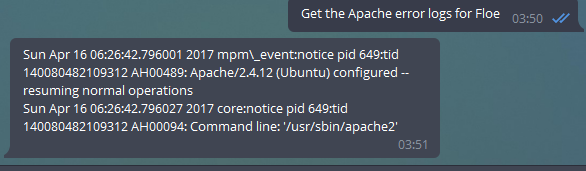


Figure 21 - Example of server object traversal and simple log parsing

Figure 17 shows an example of the log grabbing in action, though with a short error file. The application is able to parse the request into the application, the argument and the server, and traverse the Servers object to figure out which log the user is looking for, then use other information to download and show that log.

### Automation

As log handling is now complete, we are able to re-use those functions and information to create our final module, the Intermittent Query module. The format for intermittent queries is fairly simple, as shown in our design, it effectively contains the same information as a request from the user, an extra few variables with its name plus a delay, and finally two functions that handle the data. The first function, ‘run’ contains the call to the getFile function, as well as gathering any details required for it, and also manages sending users messages when it has the data. The second function, ‘format’ acts much as the Data Parsing module would, formatting the data as required for the user and stripping useless information.

The ‘run’ function is fairly self-explanatory, but the format function has some interesting uses. Firstly it is important to note that our example use of the Intermittent Query module is to check the ‘auth’ logs on a Linux machine, which shows SSH connection attempts and uses of the ‘sudo’ command. The first interesting use is the ‘IP Whitelist’, which allows any IP addresses in the array without informing the user. This is useful, as the getFile process actually uses SSH to connect to the server to download logs, and we are not particularly interested in spamming the user with alerts about the application checking for problems. Secondly, we have a ‘blacklist’ which are text that, when detected in a line of this log file, are deleted. This is just to remove some of the more uninteresting logs, such as removed sessions or disconnected sessions, as we are only interested in new or failed connections from IP addresses that aren’t our own.

The third interesting part of the format function is the regular expression used for splitting text. Due to how Linux outputs linefeeds, it was being incorrectly formatted through messages, so it was decided to use a regular expression to split to new lines based on the date stamp. However, attempting to split on the date stamp caused it to be deleted. This was solved using a negative lookahead expression, which allows us to confirm that something exists within a string without actually matching it. The code used is shown in Figure 22.

text.split(/(?=[A-Z][a-z][a-z] \d\d \d\d:\d\d:\d\d)/g);

Figure 22 - Split text regular expression with negative lookahead

The run function uses the format function to deal with incoming data before sending it to the user. However, data cannot be sent directly to the user, we must have a saved conversation with full address information in order to do so, this is where the ‘associate’ intent comes into play. Firstly though, to start this function running, an automatically running function was placed inside the index file for the application, which looped through all programmed queries, and set them up to run based on their delay. Saved queries will now run automatically irrelevant of whether a user is associated with them, which would be useful in the case of needing to save data or retrieve statistics over time.

The ‘associate’ intent, as described above, is used to attach a user to a query, so they receive all alerts that that query is programmed to give. Similar to the other commands, we first had to teach Luis how to get arguments from them. This was a simple process as we decided on the base command “associate *x*”, where ‘x’ related to the name of a particular Intermittent Query.

This was a simple intent, requiring a function in the User Interaction module that forwarded the users session straight into the Intent Processing module, which first checks if the query exists, and then saves their address to a global variable against the query name. At this point, the user is considered associated and will then be notified once the query they are associated with loops and has an alert.

### Real World Example

During our testing, we left the application online with our example intents running, and our Telegram conversation associated with the Auth alert. After a time, it was noticed that we were getting very persistent messages of failed login attempts which escalated to over a hundred times per minute, and it became obvious quickly that we were attempting to be bruteforced.

Due to it being a test server, only designed to be up for a few days at maximum and shut down when not in use, it was not expected that this machine would be attacked and so security was not deemed top priority when configuring the server. The only reason we were able to identify that an attack was occurring was due to the application sending messages to the project lead via the IM service configured on all their devices, and they were quickly able to reconfigure the server to ban such attempts.

An example message from the program during this time is shown in Figure 20, with IP addresses blanked.

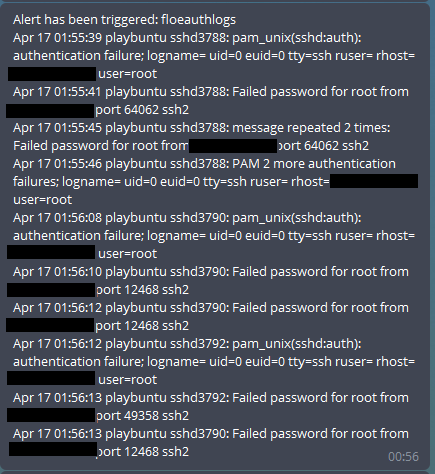


Figure 23 - Real bruteforce attack example output

## Securing the Software

Securing software against use from unauthorised parties is often an important part of any application, but in this case is somewhat easier than expected. The method for securing it as described in the design seemed to work fine, only allowing particular User ID’s access to any commands. This was set as a simple if-else inside the NLP Module, and the whitelist set in the setup file. If the user’s ID is in the setup file, then allow them to continue, else tell them they are not permitted and give them their user ID for debugging purposes, as seen in Figure 24.



Figure 24 - User failing to authenticate with the bot

## Testing

This section will show testing against general use.

# References

[1] Ubuntu, “LinuxLogFiles - Community Help Wiki,” 2015. [Online]. Available: https://help.ubuntu.com/community/LinuxLogFiles. [Accessed: 23-Mar-2017].

[2] Microsoft, “Microsoft Bot Framework.” [Online]. Available: https://dev.botframework.com/. [Accessed: 23-Mar-2017].

[3] Restify, “API Guide | restify,” 2016. [Online]. Available: http://restify.com/. [Accessed: 14-Apr-2017].

[4] LUIS, “LUIS: Homepage,” 2017. [Online]. Available: https://www.luis.ai/home/index. [Accessed: 05-Mar-2017].

[5] A. Shreve, “ngrok - secure introspectable tunnels to localhost,” 2017. [Online]. Available: https://ngrok.com/. [Accessed: 23-Mar-2017].

[6] Microsoft, “Bot Framework Emulator | Documentation | Bot Framework,” 2017. [Online]. Available: https://docs.botframework.com/en-us/tools/bot-framework-emulator/. [Accessed: 23-Mar-2017].

[7] D. Zelisko, “ping - a ping wrapper for nodejs,” 2016. [Online]. Available: https://www.npmjs.com/package/ping. [Accessed: 16-Apr-2017].

[8] W. James, “traceroute - Simple wrapper around the native traceroute command,” 2016. [Online]. Available: https://www.npmjs.com/package/traceroute. [Accessed: 16-Apr-2017].

[9] M. Colomer, “any-file,” 2017. [Online]. Available: https://www.npmjs.com/package/any-file. [Accessed: 17-Apr-2017].