User Guide

Required Software

- **Rasa 1.6.2** - The chatbot itself, handles conversation flow
- **Java –** Will be required for Logstash
- **Rasa NLU 0.15.1** - Used to interpret the user's messages, extract entities
- **Docker** - Used to run duckling server in it's own container
- **Duckling** - Used by Rasa to extract time entities from user messages
- **MongoDB** - Stores and manages our database
- **Ngrok** - Used to create tunnels to our localhost so the bot can be hosted on telegram
- **Python 3.7.4** - Our language of choice
- **Anaconda3** - Used to manage the environment the bot is installed in
- **Logstash** - Parses and transforms the logs collected by winlogbeats
- **Winlogbeats** - Collects error logs
- **Telegram** - Our chosen interface to message the bot on

Installing Elastic stack

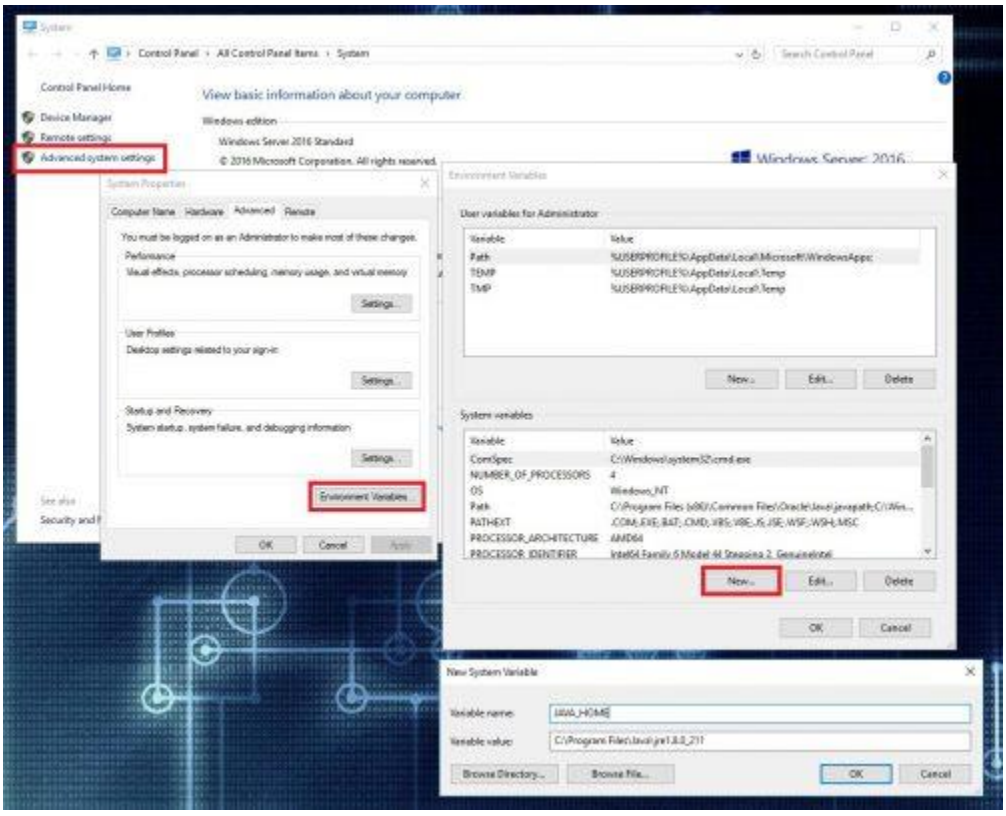
Installing Java

The first step in this process is getting the server prepared for the Elastic services by installing Java and setting up an environmental variable so Elasticsearch can locate Java.

Download and install Java – Make sure to get the x64 version for a 64 bit OS: https://java.com/en/

Take note of the installation path during the install, you'll need to know that for the next step. It should be something like C:\Program Files\Java\jre1.8_xxx:

Create a system variable named JAVA_HOME with a value of the path to the java installation.



Reboot the server.

Installing Elasticsearch

Elasticsearch is the core of the ELK stack and is where all of the data will be stored. Elastic now has a .msi package available for Elasticsearch which makes it a fairly simple install on Windows.

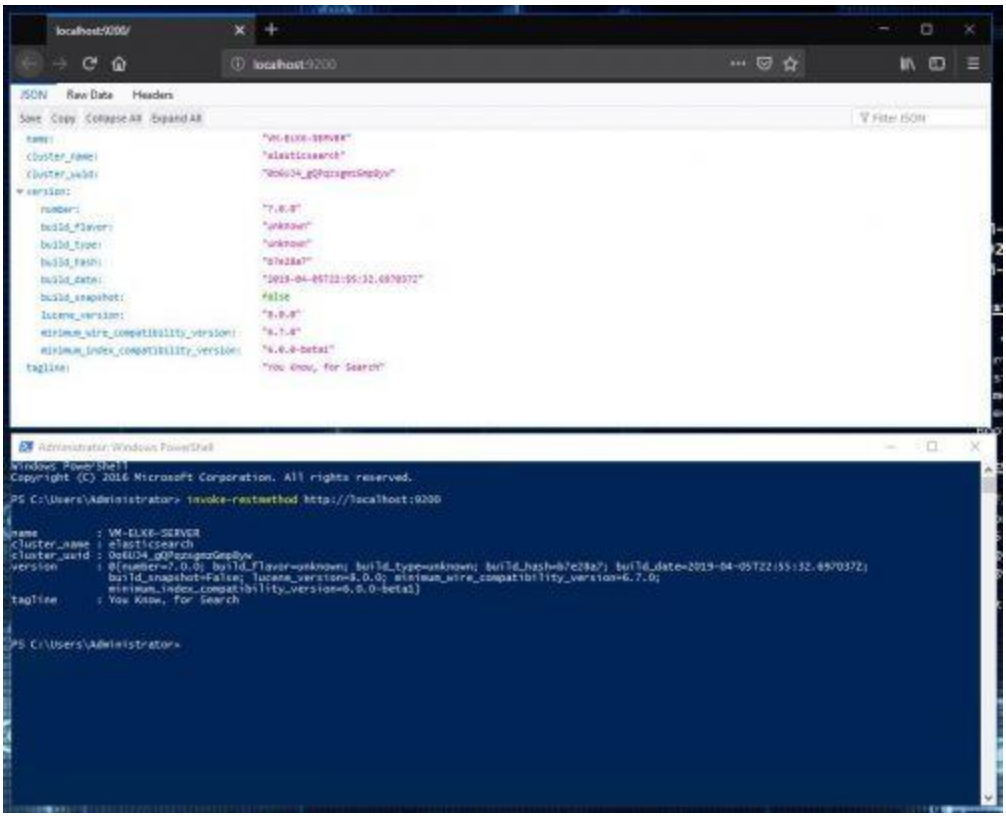
Download the .msi package from Elastic: https://www.elastic.co/downloads/elasticsearch

Double click the package to begin the installation, follow the prompts leaving all the defaults.

Note: You can change the data directory location during the install, which is useful if you were planning on using dedicated drive or separate partition.

Now to test if Elasticsearch is up and running browse to http://localhost:9200 or make a request to the server with PowerShell using the following command:

1	PS C:\> invoke-restmethod http://localhost:9200



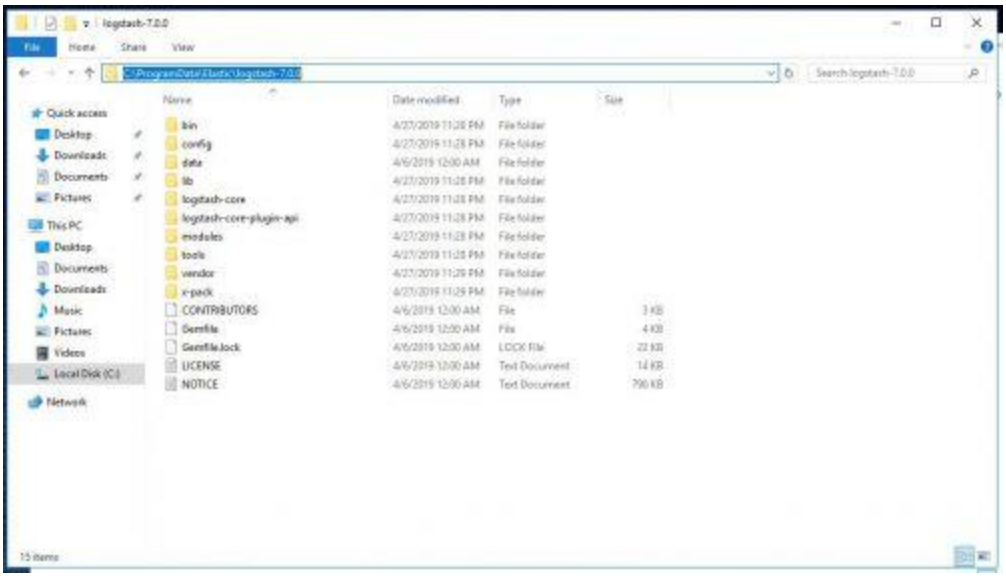
The Elasticsearch install is now complete.

Installing Logstash

Logstash is responsible for receiving the data from the remote clients and then feeding that data to Elasticsearch. Installing Logstash is a little more involved as we will need to manually create the service for it, but it is still a fairly straight forward install.

Download the Logstash package in .zip format: https://www.elastic.co/downloads/logstash

Unzip it to where it is going to be installed to permanently, in this case I am using C:\ProgramData\Elastic\Logstash.



Note: You may run into issues starting the service if the installation path contains a space, ex C:\Program Files\Elastic. To get around this wrap the path in quotes when creating the service in NSSM, ex “C:\Program Files\Elastic”.

Next is the configuration file, which needs saved to the Logstash\config directory. You can download a copy of the one I used here: logstash.conf.zip

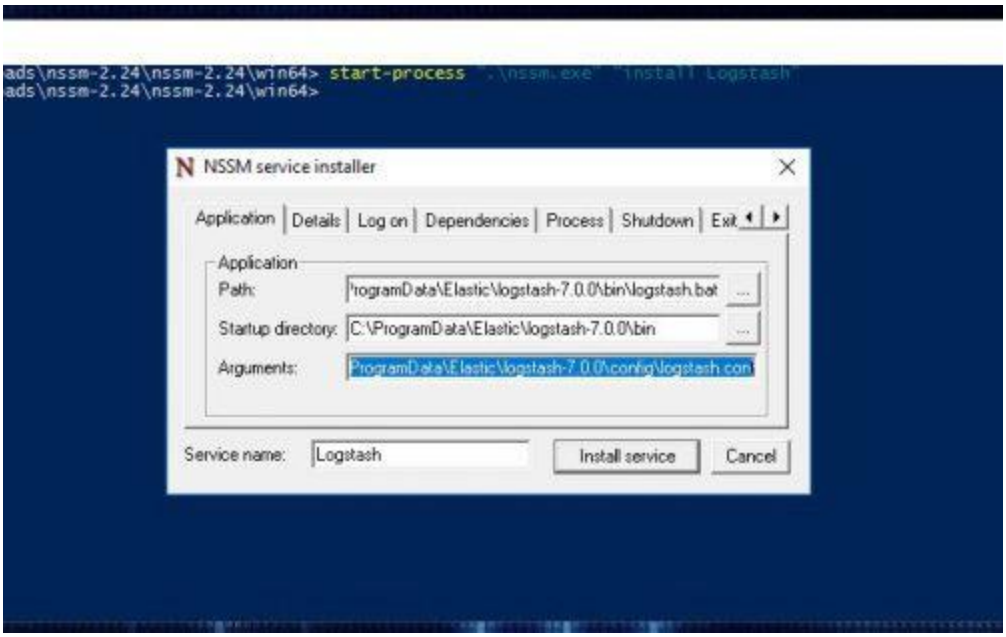
There is also a sample configuration file in the config directory named “logstash-sample.conf” that you can refer to.

NSSM is going to be used to create the service for Logstash. You can download NSSM (Non-Sucking Service Manager) here: https://nssm.cc/download

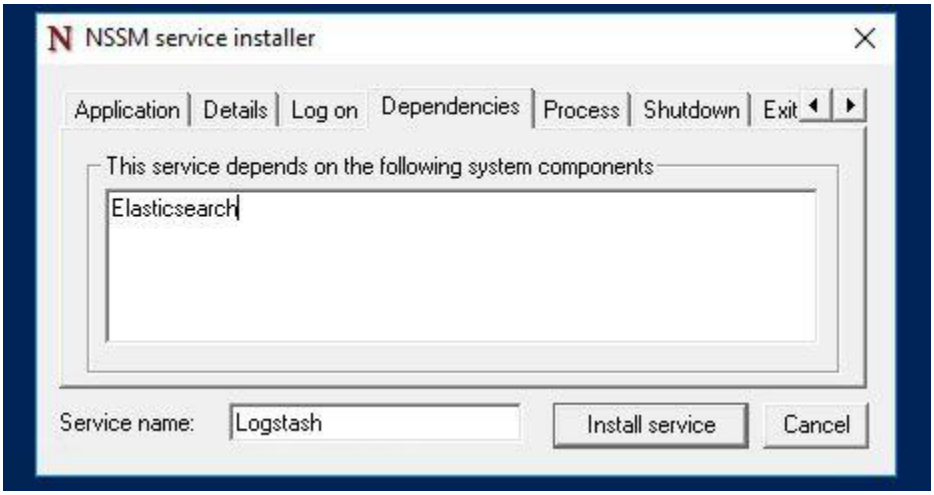
Unzip the NSSM package and then use the following command to create the Logstash service:

1	PS C:\Users\Administrator\Downloads\nssm-2.24\nssm-2.24\win64> start-process ".\nssm.exe" "install Logstash"

Set the following values on the Application tab:
Path: C:\ProgramData\Elastic\logstash-7.0.0\bin\logstash.bat
Arguments: -f C:\ProgramData\Elastic\logstash-7.0.0\config\logstash.conf
Service name: Logstash

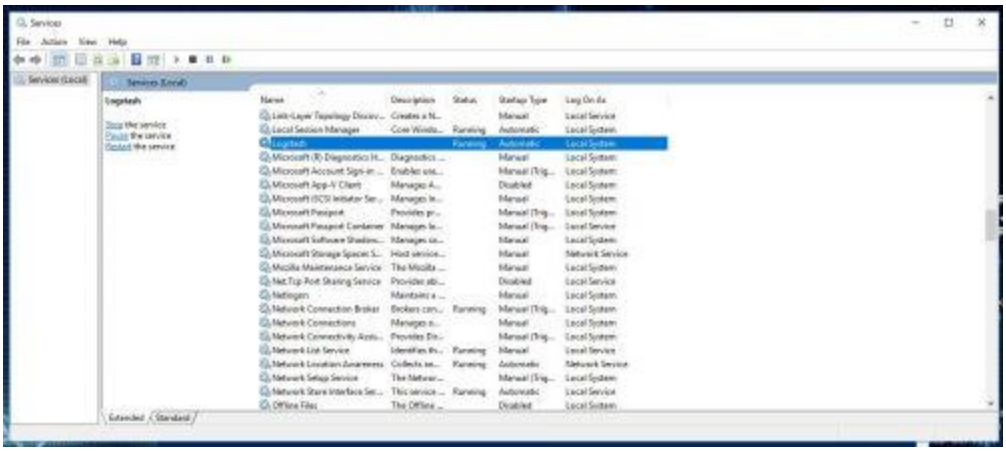


Set a dependency of Elasticsearch:



Click Install service.

Open services.msc and verify the service starts:



Don’t forget to add a firewall rule to allow the inbound connections:

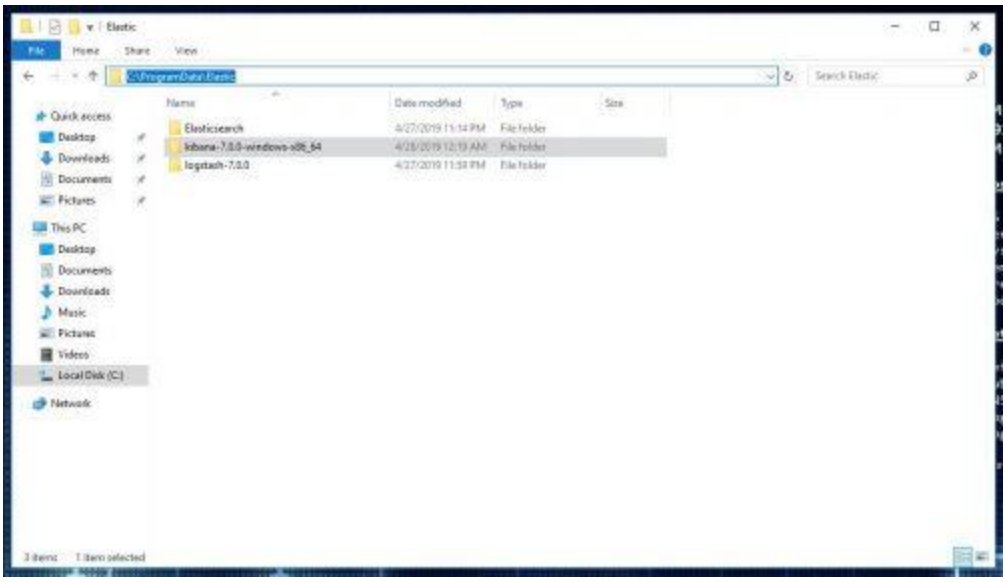
1	PS C:\> netsh advfirewall firewall add rule name=Logstash-Inbound-5044 dir=in action=allow protocol=TCP localport=5044

Installing Kibana

Kibana is the web based front end that will be used to search the data stored in Elasticsearch. The Kibana installation is very similar to the Logstash install, and NSSM will be used again for the service creation.

Download the Kibana package for Windows in .zip format: https://www.elastic.co/downloads/kibana

Unzip it to where it is going to be installed to permanently, in this case I am using C:\ProgramData\Elastic\Kibana.

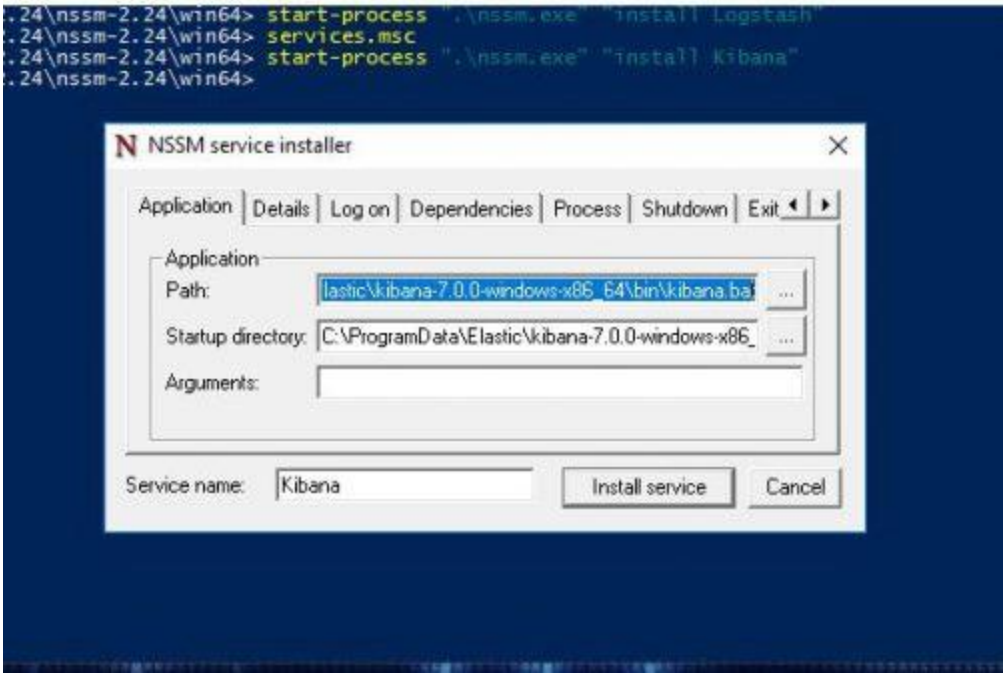


Note: You may run into issues starting the service if the installation path contains a space, ex C:\Program Files\Elastic. To get around this wrap the path in quotes when creating the service in NSSM, ex “C:\Program Files\Elastic”.

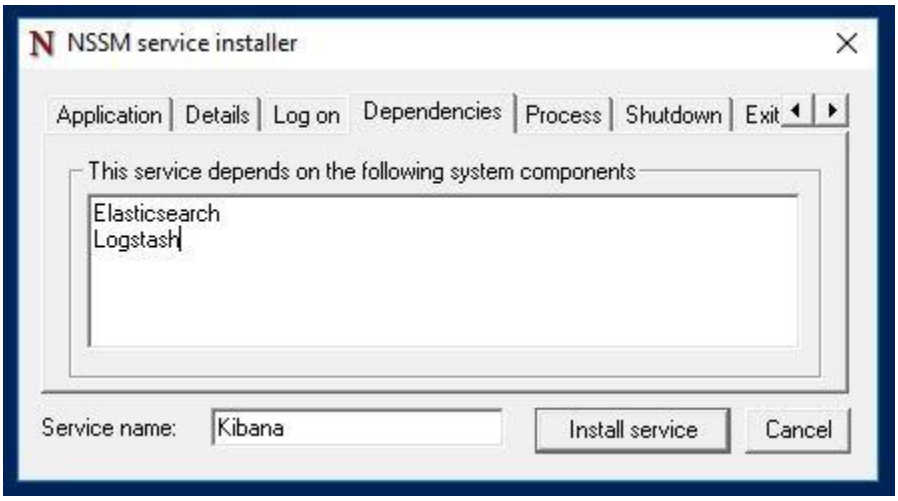
Use the following command to call NSSM and create the Kibana service:

1	PS C:\Users\Administrator\Downloads\nssm-2.24\nssm-2.24\win64> start-process ".\nssm.exe" "install Kibana"

Set the following values on the Application tab:
Path: C:\ProgramData\Elastic\kibana-7.0.0-windows-x86_64\bin\kibana.bat
Service name: Kibana



Set a dependency of Elasticsearch and Logstash:



Click Install service.

Navigate to the Kibana configuration file, found in the config directory, which in this case is C:\ProgramData\Elastic\kibana-7.0.0-windows-x86_64\config\kibana.yml.

Open the file in notepad and uncomment/edit the following lines:

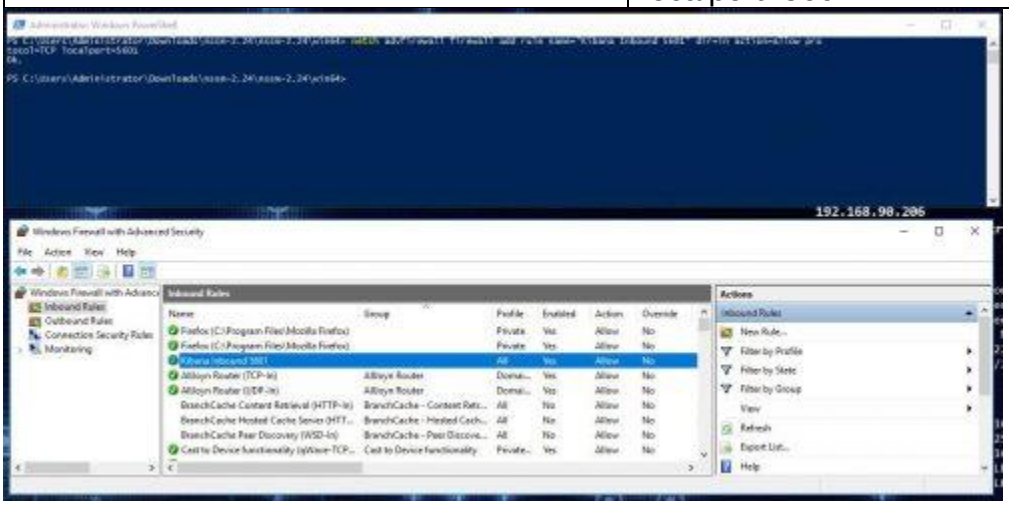
1 2 3 4 5	# Kibana is served by a back end server. This setting specifies the port to use. server.port: 5601

6 7	# Specifies the address to which the Kibana server will bind. IP addresses and host names are both valid values. # The default is 'localhost', which usually means remote machines will not be able to connect. # To allow connections from remote users, set this parameter to a non-loopback address. server.host: "0.0.0.0"

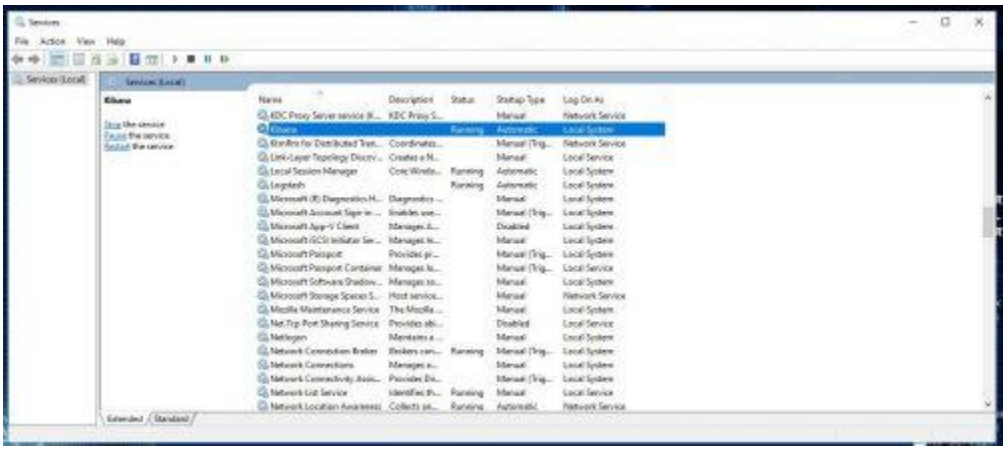
This will bind Kibana on port 5601 and force it to listen to all IP addresses on the host, this will allow remote hosts to access Kibana via http://IPAddress:5601.

Don't forget to add a firewall rule to allow the inbound connections:

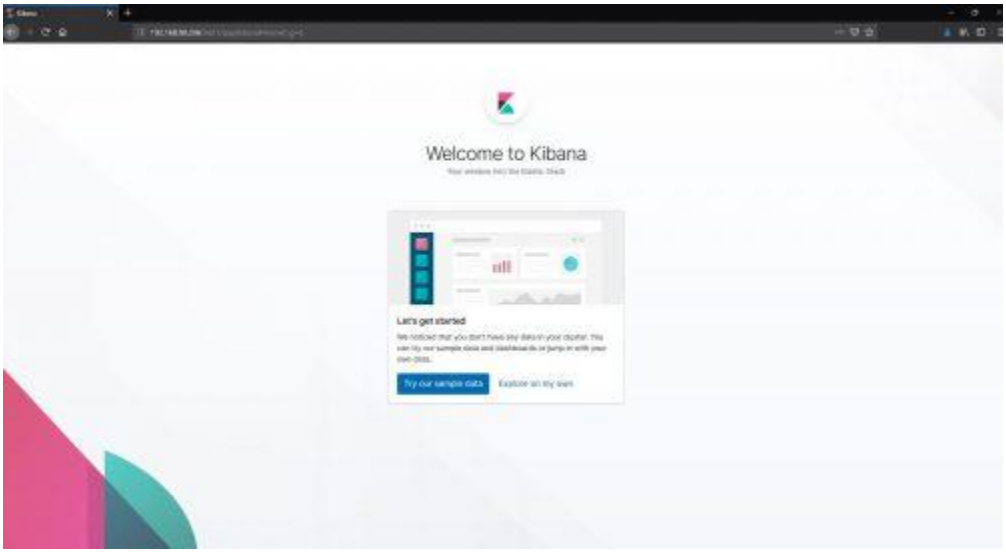
1	PS C:\> netsh advfirewall firewall add rule name=Kibana-Inbound-5601 dir=in action=allow protocol=TCP localport=5601



Open services.msc and verify the service starts.



Verify Kibana is accessible by http://IPAddress:5601:



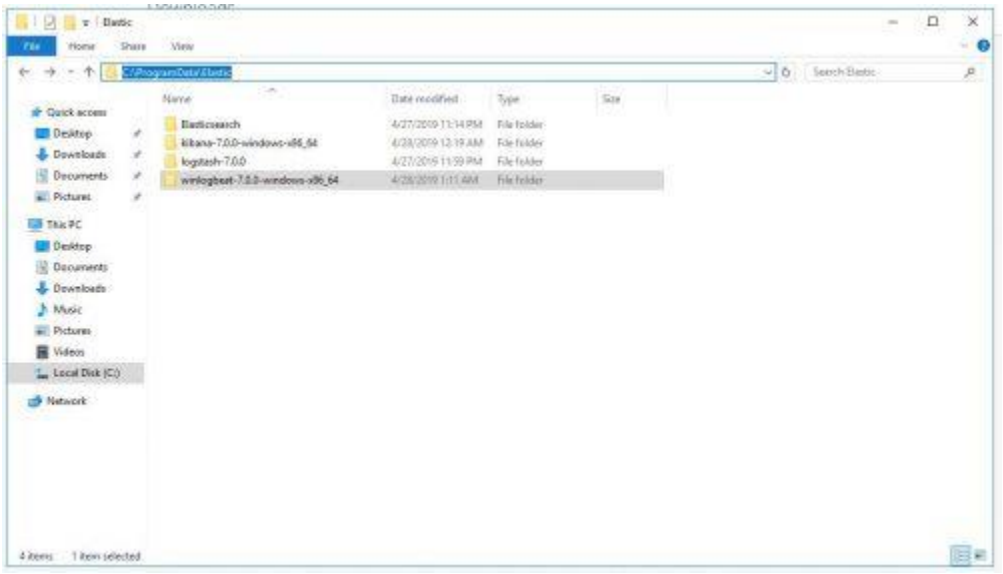
The ELK stack is up and running at this point, now it is time to start ingesting some logs from the local host.

Installing Winlogbeat

Winlogbeat is going to be the “agent” that gets installed on each Windows server/client that will forward logs from the host to the ELK instance. If you have ever worked with Splunk, Winlogbeat is similar in nature to the Universal Forwarder.

Download the Winlogbeat package for Windows in .zip format: https://www.elastic.co/downloads/beats/winlogbeat

Unzip the package to its permanent home, I will be using C:\ProgramData\Elastic\Winlogbeat:

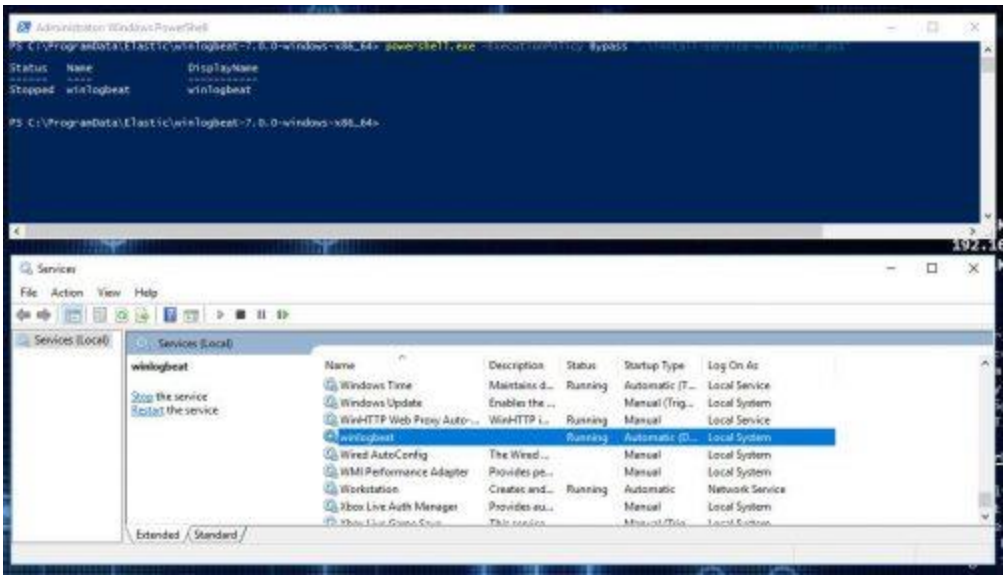


Edit the winlogbeat.yml configuration file, commenting out the Elasticsearch output and uncommenting the Logstash section setting the host to the IP of the ELK server:

```
#==============================
== Outputs
==============================
======

# Configure what output to use when sending the data collected by the beat.

#---------------------------- Elasticsearch output ----------------------------
#output.elasticsearch:
  # Array of hosts to connect to.
  # hosts: ["localhost:9200"]

  # Optional protocol and basic auth credentials.
  #protocol: "https"
  #username: "elastic"
  #password: "changeme"

#----------------------------- Logstash output -----------------------------
output.logstash:
  # The Logstash hosts
  hosts: ["192.168.90.206:5044"]
```

Install Winlogbeat as a service with the PowerShell install script:

```
PS C:\ProgramData\Elastic\winlogbeat-7.0.0-windows-x86_64> powershell.exe -ExecutionPolicy Bypass ".\install-service-winlogbeat.ps1"
```

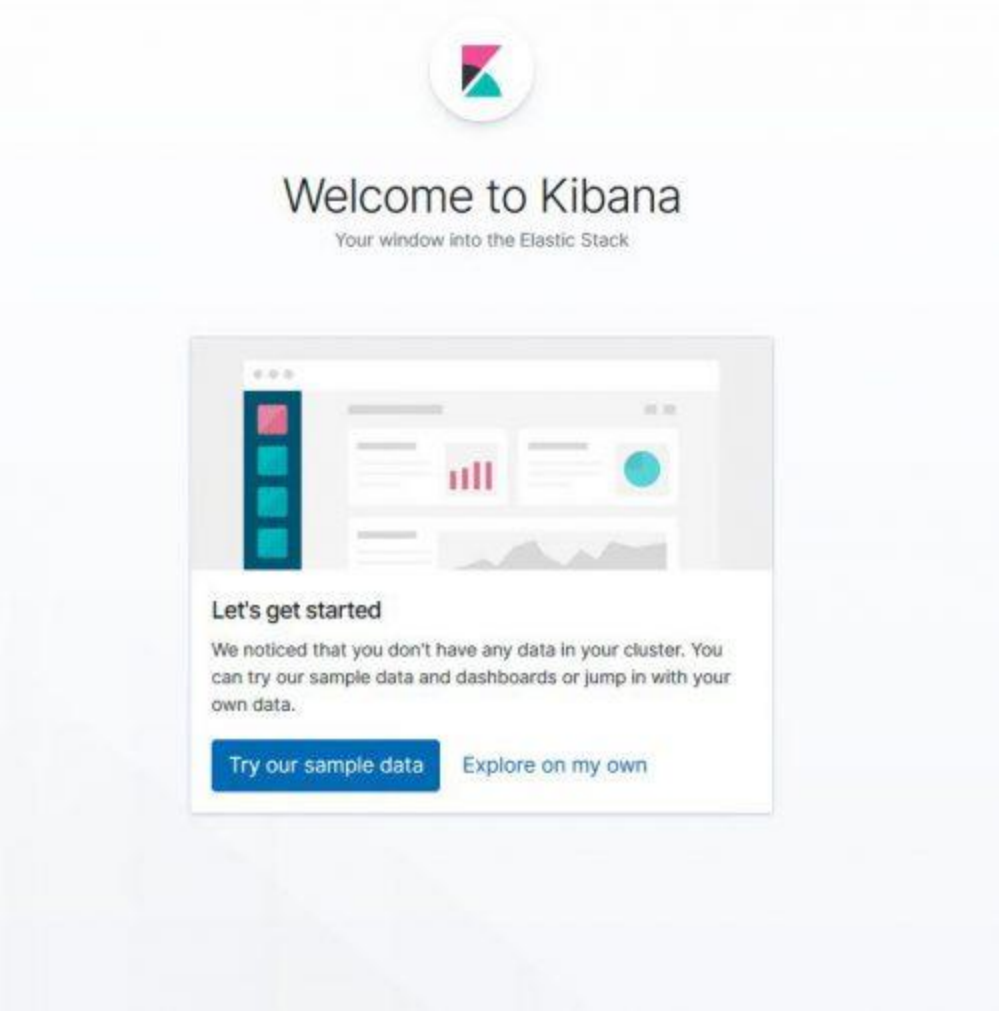


Make sure the service is started.

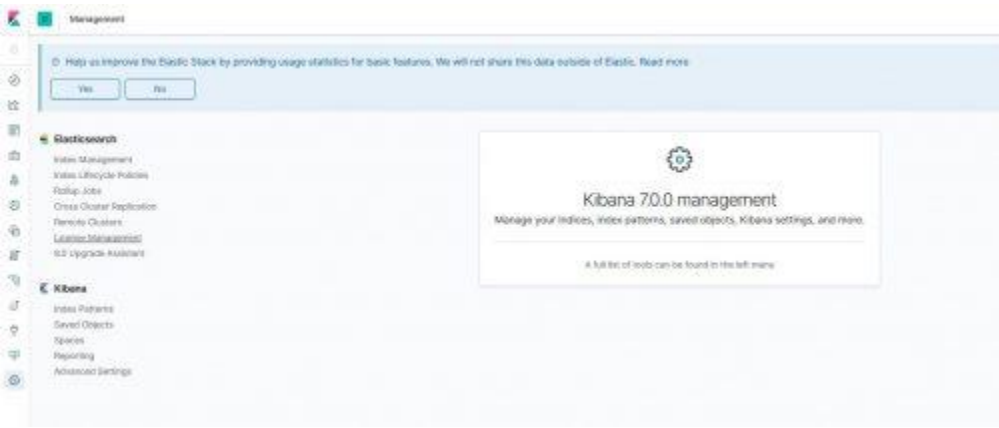
Creating An Index Pattern In Kibana

Index patterns tell Kibana what Elasticsearch indices we want to search, so now that there is Winlogbeat data in Elasticsearch, an index pattern needs to be configured on the Kibana side.

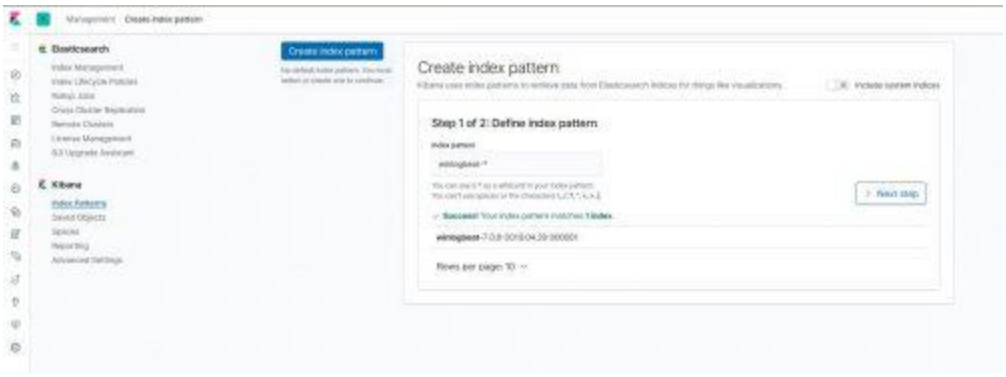
Access Kibana at http://IPAddress:5601 and click on Explore on my own:



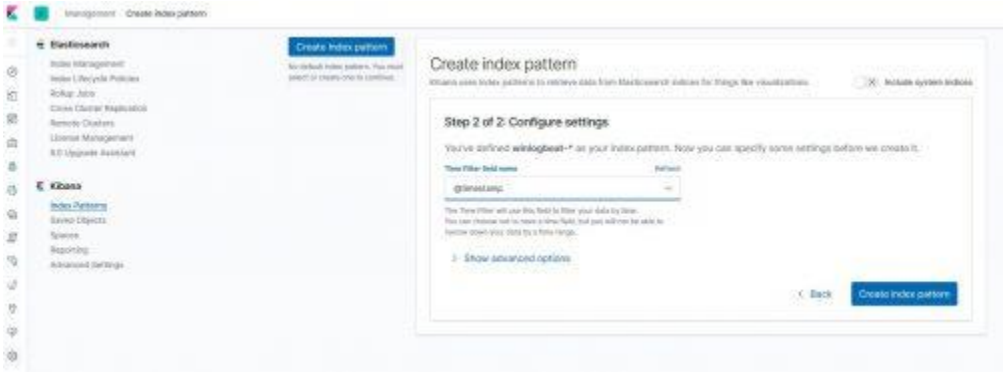
Click on Management (Gear Icon) on the left hand menu:



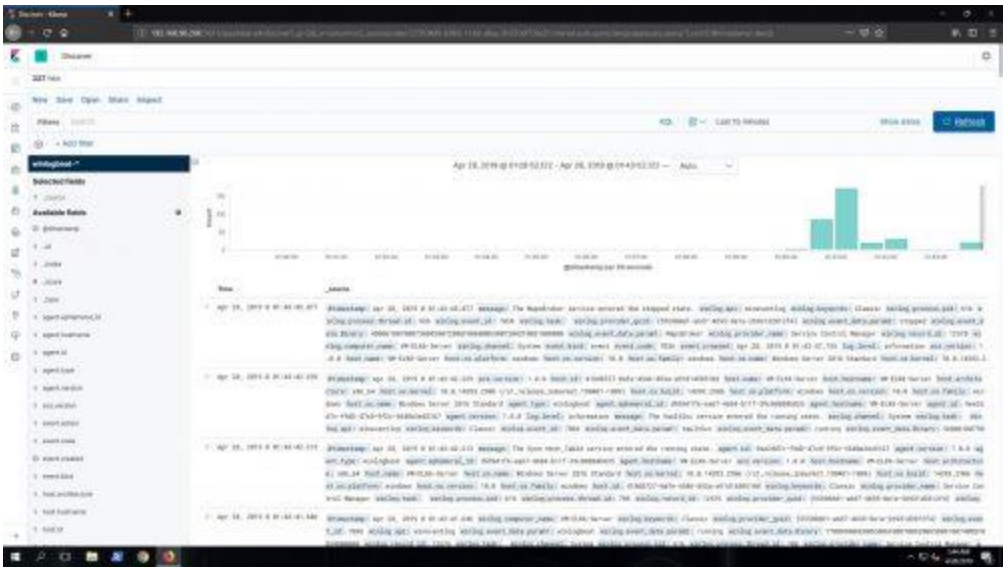
Click on Kibana > Index Patterns and then Create an Index pattern using winlogbeat-* as the pattern:

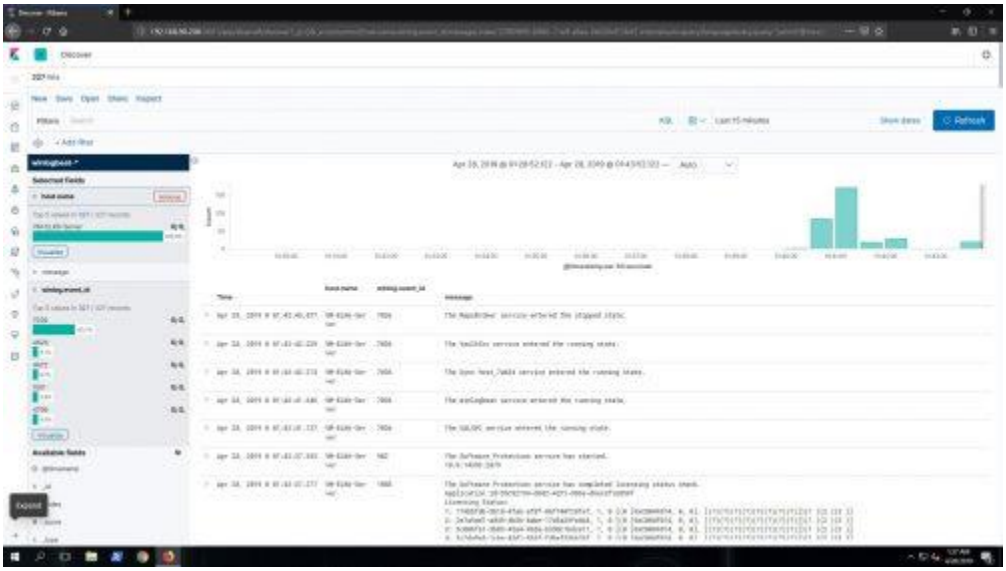


For the Time Filter field name use @timestamp and then click Create index pattern:



The basic Windows Event Logs – Application, Security, and System are now flowing into the ELK stack from the ELK server itself:





Installing Curator *(optional)*

One thing that often seems to be an after thought when it comes to the ELK stack is storage/data management, which is critical to monitor/manage since the server will keep ingesting data until it fills the disk. That is where Curator comes in and provides an automated way of accomplishing this task.

Curator is a tool to help curate, or manage, the Elasticsearch indices. In this case, we will be using it as a scheduled task to clear out the older data after it reaches a specified age.

Download and install Curator, following the defaults: https://www.elastic.co/guide/en/elasticsearch/client/curator/current/windows-msi.html

Note: The default install directory is C:\Program Files\elasticsearch-curator\, we'll need this for later.

To use Curator, two basic configuration files are needed. One is for Curator itself and the other is an “actions” file, that specifies what to delete and the conditions to do so.

You can download a copy of the configuration files I used here: config.zip

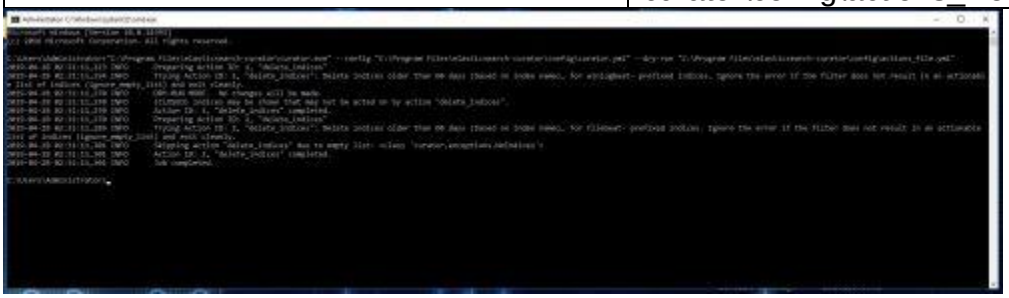
These config files should work for Winlogbeat and Filebeat, clearing data out that is older than 60 days. You can of course tune these configurations to keep as much or little data to suit your needs.

Download and unzip the config folder to C:\Program Files\elasticsearch-curator\.

Note: Make sure the time strings on the Logstash configuration match the Curator “actions_file” configuration, ie timestring: ‘%Y.%W’. In the config files used in this post, I have made sure to match these values already.

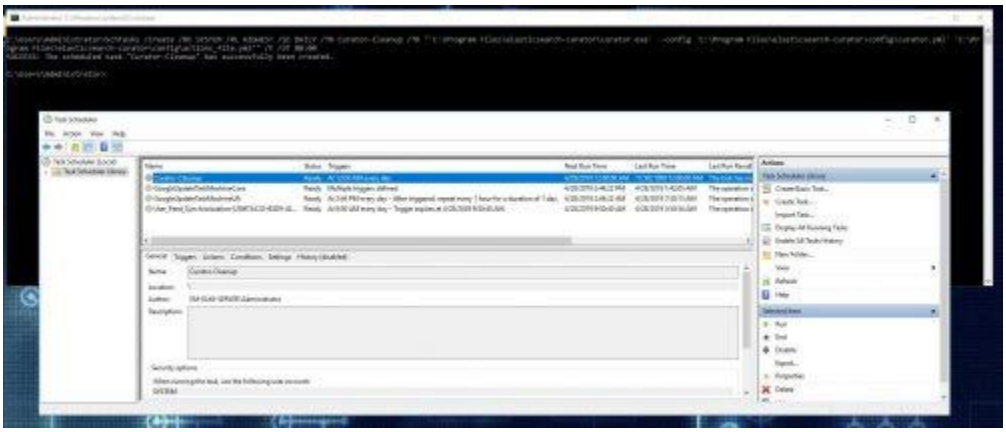
To test Curator and see what all it would do without making any changes aka a “dry run”, the following command can be used at a cmd prompt:

1	C:\> "C:\Program Files\elasticsearch-curator\curator.exe" --config "C:\Program Files\elasticsearch-curator\config\curator.yml" --dry-run "C:\Program Files\elasticsearch-curator\config\actions_file.yml"



Once you have the right command and have verified it works as expected, Curator can be scheduled as a daily task with the following command:

1	C:\> SchTasks /Create /RU SYSTEM /RL HIGHEST /SC DAILY /TN Curator-Cleanup /TR '"C:\Program Files\elasticsearch-curator\curator.exe' --config 'C:\Program Files\elasticsearch-curator\config\curator.yml' 'C:\Program Files\elasticsearch-curator\config\actions_file.yml"' /F /ST 00:00



Bot Installation

To connect to the bot, open telegram and search for the bot using it’s telegram username - this will vary from our own as any new admins will need to create their own telegram bot account to host it.

The use of the bot after connecting is simple, requiring users to simply message the bot and answer any of the questions regarding the initial query the bot may have.

Currently the bot is capable of the following queries:

- Finding any new logs in the database
- Finding logs on a specific date
- Finding logs between two given dates
- Findings logs before or after a date
- Counting the amount of new error logs
- Returning more detailed information on a recently returned log
- Marking the most recently queried logs as having been read by the bot - this is done at the users request, automatically

- Extending the results - the user can specify an amount of results written to file to be output from the bot

For a typical query, the following sequence of events will occur:

1. The user messages the bot asking for error logs from some time period (can be now or a date range)
2. The bot processes the message and assigns an intent where it believes the user wishes to query the database
3. The bot executes its first custom action where it extracts information from the users message and returns a message asking the user to confirm the query and the date (or range) to be searched
4. The user confirms they want to query with that information, if not the bot will then issue a message asking the user to reenter a query and steps 1-5 will repeat as necessary
5. If yes, the bot carries out a database search and returns the results in the form of a message to the user, if more than 3 results are found then the 4th result onward are written to the “currentIDs.txt” file. If no results are returned then a message explaining this is issued

Should the user’s “find” query return more than 3 results then the bot will stop outputting the remaining results and will instead store these results in a file called “currentIDs.txt” which will be in the same directory as the bot and the “actions.py” file. The bot will then message the user informing them of the file’s existence and what it contains. The user can then ask the bot to output a specified number of results from the file, or to output the “top”/”bottom” half of the file, in the form of the usual results message.

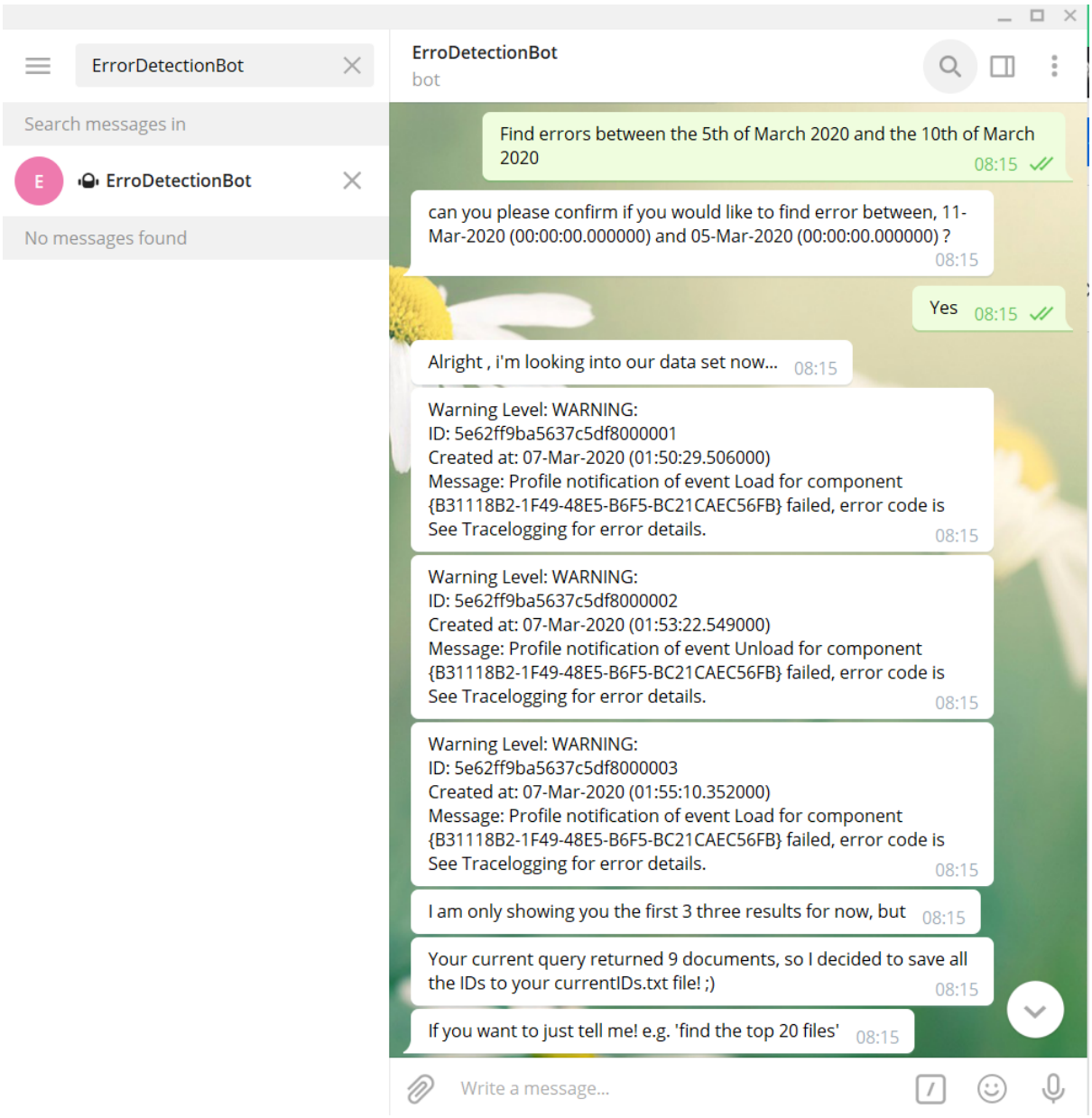


Figure 1: Screenshot of telegram conversation with the after asking it to find errors between the 5/03/20 and 10/03/20

Introduction

The following section contains detailed information on how the various software used for our bot works, as well as short guides on how to add features to the bot or edit existing ones. A lot of this information was found by experimenting with the bot as the

documentation provided is not very detailed (included in the appendices), most of the information comes from our own understanding which was developed through lots of testing and searching online forums for help.

Important Notes:
As most of the required software are python libraries, the bot is set up inside of a conda environment. To install any of the required python libraries you must first use the 'activate *your_environment_name_here'* command and then use the python 'pip' command required to install the libraries to the same environment, or alternatively 'conda' can be used for some libraries hosted on the conda-forge.

In our action's file we use Pymongo to connect to and query the database, however, if your Mongo database is not ran locally then find any instances of "MongoClient('localhost', 27017)" in the file and replace the content in the brackets with your own connection string. After this has been done, change the db variable, currently "client.newRasaDB", to "client." followed by your rasa database name. Change any 'col' variables from "db.rasaCollection" to "db." followed by your Mongo collection name.

To ensure the bot is running correctly, the actions server must be started using the command 'rasa run actions'.

Docker must also be used with the command 'docker run -p 8000:8000 rasa/duckling' which will install duckling (if it is not installed) and then will run duckling inside of a docker container so that your bot may connect to it. If duckling is run some other way or you have your own duckling server then change the duckling ip address in the 'Rasa/config.yml' file to match, or the bot will be unable to recognise time entities.

Rasa

The user side of rasa was explained in the user guide, but you will need to know how to modify our messages, or change and create your own intents and custom actions. Each key bit of rasa is explained in more detail in the following section, but I have created a short little guide on how to create your own action, detect when it should be called and when to execute it.

Step 1: Open the actions.py file in the rasa folder, create a new action class, define a name function so it returns a unique name and then write the code you want it to execute in a method called run.

Step 2: Create your own intent related to that action in the ‘Rasa/data/nlu.md’ file, add training data below that intent so the bot knows what to look for. Mark any entities you are searching for. Edit the ‘Rasa/domain.yml’ file and add your action name to the actions section, intent name to the intents section and the entity (if it’s new) in the entity section.

Step 3: Create a story in the ‘Rasa/data/stories.md’ file - for most stories your custom action will immediately follow your intent being detected by the bot.

Step 4: Finally, train your model. This can be done by running the ‘rasa train’ command in your bot’s anaconda environment.

By the end of this, you will have a custom action which is run when the intent you have written is found by the bot.

To modify any existing actions, intents, entities or stories just edit each individual file where you wish, not every file needs to be changed if only a small modification is being made e.g. changing what message an action will send back to the user.

Intents

Intents are examples of user messages which help the bot to understand what a user is saying or asking of the bot when it receives a message from them.

To add more intents to the bot fopen the nlu.md file in the ‘Rasa/data’ directory with some kind of text editor. Write ‘## intent:’ and then the name of your new intent. Below this, write your examples with each new example starting with a dash, e.g.

Figure 2: Greeting intent as seen within the Brackets editor

Any new intents must be declared in the domains file, under the intents section, and for any messages or actions to follow your intent being interpreted then a story must be written in the ‘stories.md’ file in the same directory.

Custom Actions

Custom actions are functions or other bits of code to execute, usually in response to the bot inferring a specific intent from the user.

They are written as python code under the ‘actions.py’ file, and as such are quite flexible as you can import any python library to do what you require. Our main example being pymongo which we use to query the mongodb database, format the results and then output them using the ‘dispatcher’.

Actions must be in the form of their own class in actions.py, with a ‘name’ method to tell the bot the actions name, and a ‘run’ method which is the code to be executed.

Actions are capable of reading the user’s last message or any slots from the program, so if the user requests a date to query for example it can be read from the message and substituted in the mongo query or the bots reply, making the bot very dynamic.

Entities

Entities are found from the user's message by the bot, they can be marked in an intent using the format: [entity text](entity_name) or found using a library such as SpaCy.

To define an entity for cancelling something, you could define:

```
-[stop](cancel)
-[stop the request](cancel)
-[cancel this](cancel)
```

The bot would search for the text in the square brackets and store the entity with the name 'cancel' - custom actions could then search for the 'cancel' entity and cease what they were doing if found, for example.

Entity's also help the bot to more correctly interpret the user's intent.

Any entities you wish to find should be declared in the entity section in the domain file.

Stories

Stories tell the bot what to do when a specific intent is interpreted by the bot, functioning like a series of 'if' 'else' statements almost. They are conversations with the rasa bot boiled down to intents and custom actions only.

Stories can tell the bot to utter a predefined message in response to an intent, but can also be used to respond with multiple messages and custom actions to a user's message. They can be simple, e.g. being only one user message and one bot response long, but they can be defined as the length of conversations.

Stories are found in the stories.md file in the 'Rasa/data' directory, and can be edited with a text editor. The '*' means this is an intent from a user's message, with '-' being the bot's response, either an 'utter' which is a predefined message, or a custom action.

Figure 3. Basic story where user says hello to the bot and it would greet them back

```
## happy path
* greet
  - utter_greet
* mood_great
  - utter_happy
```

```
## happy path + find_error_simple
* greet
   - utter_greet
* find_errors_simple
   - action_infoCheck
* affirm
   - action_search_database
* thank
   - utter_goodbye
```

Figure 4. Story user and bot greet other, user queries the bot, bot confirms info given, user approves and the search is executed

Slots

Slots are used like variables to store information as the bot runs, like entities or intents they must be declared in the domains file with their name and type - if the type isn't known they can be declared as 'Unfeaturized'. Custom actions can write values to the slots, such as extracted entities or times.

In our program an action is run when a user wants to query the database, which extracts the conditions e.g. a date range to search within, and then a message is sent to the user to confirm this. The extracted times are added to a dictionary which is stored in a slot called time. If the user confirms the search then his original message containing the times is no longer the latest message visible to the bot, so our search actions can not extract the same times - this is where the slot is used, as the extracted conditions are not lost so the search can still be done. Other slots are also used, for example a time pointer storing ‘before’ or ‘after’ to determine if a date query should use ‘less than’ or ‘greater than’ the given date.

The ‘extracted_time’ ‘IDCount’ slots should not be removed as this will stop the bot from being able to query using times or from accurately reporting how many results were added to the file storing result counts higher than three.

Domain

Contains the names of custom actions, intents, slots and entities - should be updated with any new or removed ones so the bot can function - any custom actions not in the domains file could not be found during the running of the bot for example and so would do nothing when called.

Endpoints

Contains the connection details of the bot e.g. port its running on locally, more importantly it connects to the actions server so your bot may run custom actions.

Config

The config file, found in ‘Rasa/config.yml’ contains import configuration details for your bot, such as the language the bot speaks, the pipeline and policies.

The pipeline contains libraries and modules to process the user’s message, detect entities and classify the user’s intent from the message. Our bot mainly uses the CRFEntityExtractor to detect entities like ‘errors’ from a message so the bot knows we wish to query the database, along with DucklingHTTPExtractor to extract times and dates. Another key pipeline feature is the EntitySynonymMapper which is used to detect

synonyms of our entities, for example we have a ‘before’ entity which gets assigned to ‘before’ and synonyms like ‘previously’ or ‘prior to’.

Policies are used to help the bot predict the next appropriate action to take, guided by the stories file.

It is recommended not to remove existing pipeline elements or policies as this may break the bot, but if you find any new features you wish to add to the pipeline then it should be okay to add them, same with policies.

Python

Libraries

The bot only uses a select few libraries, listed below with how they are utilised in our script.

- Pymongo - Used to connect to the Mongo database and query it. Key variables are ‘client’ which is the connection to Mongo itself, ‘db’ is the database and ‘col’ is the collection containing the log files. Change these as necessary with your own Mongo details.
- Rasa SDK - Used to manipulate the bot, reads the message text and extracted information and can also be used to output information to the user in the form of a message. The tracker is used to get data stored in slots as well as message text, whereas the dispatcher is used to output messages to the user. Custom actions can store extracted information in the slots which can be utilised by other custom actions e.g. our ‘action_info_check’ stores a dictionary containing extracted times which are then accessed by our ‘find’ custom actions, which query the database using the times from the dictionary.
- Dateutil.parser - Used to convert extracted times into datetime objects so they can be used in pymongo date queries. Pymongo date queries must use datetime objects.
- Bson.ObjectId - Used to parseid’s extracted in string format into ObjectIds so they can be used in pymongo ID queries.

Actions and Essential Data Structures

Dict (stored in extracted_times slot) - Stores the time values extracted from the message with an identifier like ‘to’ or ‘from’ so the right values are found by other actions, also stores ‘number_of_times’ so the actions know if they are searching for a single date or date range.

If you view actions.py in an IDE like Visual Studio code it will say there’s an error and that Dict has been used before assignment, ignore this, as the bot will give it a value before the action is called.

action_infoCheck

Takes the user’s message and extracted time entities, stores the extracted times in a dictionary and stores this in the ‘extracted_times’ slot, along with the number of extracted times so the find actions know whether they are querying a single date or date range.Then returns a message with the time entities in, and asks them to confirm if the extracted times are correct. If the answer is yes, depending on the intent, one of the ‘action_find’ actions will be executed.

Following a confirmation, If there are two dates found and none are null, then the next action will be ‘action_search_database’ - however if one of the times returned is null then the words ‘before’ or ‘after’ a single date have been used and ‘action_find_one’ will occur. These actions are dictated by the stories file, not the custom action itself, but are reliant on this action occurring an ‘affirm’ intent following given by the user.

action_out_of_scope

Returns a message to the user if they have asked something of the bot it is currently incapable of doing.

action_find_one

If the user has asked for logs ‘before’ or ‘after’ a single date this action will occur - first it checks the user’s message for the word ‘before’ or ‘after’; if the word ‘before’ is used then a query of less than the given date will occur - if the word ‘after’ is used then a query of greater than the given date will be used.

Results will be iterated through in a for loop, dispatching messages until the count is greater than 3, and the excess will be iterated through and added to the ‘currentIds.txt’ file.

action_find_new_error
Carries out a simple query where any logs which returns results where the field “bot_has_checked” is marked as false.

Results will be iterated through in a for loop, dispatching messages until the count is greater than 3, and the excess will be iterated through and added to the ‘currentIds.txt’ file.

action_count_errors
Carries out a simple query which counts the number of logs which have the field ‘bot_has_checked’ marked as false, and returns the count in a message to the user.

action_more_details
Checks the file containing recent results, if there are any results the it queries using the id of the first item and returns more fields, like if the log has been checked by the bot already, the hostname of the computer where the error occurred and the OS the computer is running. If none returns a message saying there were no recent results to provide more detailed information of.

action_mark_read
Checks the file containing recent results and carries out a mongo query where it sets “bot_has_checked” field to “true”, after each update checking if the update went through and counting successful vs unsuccessful cases, reporting both back to the user if there were any.

action_search_database
The action first checks the ‘number_of_times’ value from the dictionary in the ‘extracted_times’ slot - if there is only time then a query searching for a specific date or time will occur. If there are two times present, the query will look for logs between the dates, however if the word ‘now’ is found in the user’s message it carries out the same query but looks logs between the largest and smallest of the two dates, rather than the order they are given like in the original query. This is because someone could give a query asking for ‘between the 5th of March and now’ or ‘between now and the 5th of March’ - only one would give a result even though they are lexically asking for the same thing, just not logically the same.

Results will be iterated through in a for loop, dispatching messages until the count is greater than 3, and the excess will be iterated through and added to the ‘currentIds.txt’ file.

action_extend_result

Occurs when a user asks for a number of results from the ‘currentIDs.txt’ file, can also work if a ‘top’ or ‘bottom’ position is given in the message. Then opens the file and finds result logs based on the parameters given, iterates through them and outputs them in the usual message format without the 3 message limit. Storing Id numbers in a text file was done to increase performance and not cluther the Bot memory, as in a real world scenario there would most likely be hundreds of files if not more.

actionFindBeforeAfter

Retrieves error logs before or after a specified time stamp using a spaceTimePointer entity which will help us determine whether we are before or after a specified period of time. Using the synonym function the machine learning model then mapps the words synonym of before to the value ‘before’ and does the same with ‘after’, according to the provided data in the nlu.md file.

Telegram

To connect the bot to telegram, ensure that you have ngrok and telegram installed. Ngrok is only essential if you are running rasa on your localhost and not on a server.

Right click and run ngrok.exe in administrator mode, then enter the command ‘ngrok.exe http localhost:5005’ - this will give an ip address to your localhost port which will be running the rasa bot - if your port is different change the command to reflect this.

Next, open another window of ngrok and use the command ‘ngrok.exe http 192.168.99.100:800’ - this will give your duckling server in the docker container an ip address. Change the ip in the command to reflect your duckling server if run locally. After this, edit your config.yml file and change your duckling server’s url to the one given by ngrok e.g. “https://482bbec8.ngrok.io”

Then follow the instructions given at: https://rasa.com/docs/rasa/user-guide/connectors/telegram/

And change the url to reflect the url given to you for your rasa bot, it should look a little like this (https://bacdc53a.ngrok.io was the temporary ip we were given):

```
telegram:
  access_token: "1110592419:AAGsGl_npDH9Vh6LWwO_JZqXzAxkbCbRjBQ"
  verify: "LogCheckerinobot"
  webhook_url: "https://bacdc53a.ngrok.io/webhooks/telegram/webhook"
```

Note: ngrok addresses only last for 8 hours, if you were planning on running the bot as a business solution you should host it online with the other components and change the duckling url and credentials url to reflect their new urls. This would be a more permanent solution.

Bot Limitations

The bot is not perfect and will not be able to correctly identify a user's intent every time, it is limited by a small amount of training data and in a business scenario would need more training data and examples to learn from. Sometimes entities are not extracted perfectly even if the intent is correct, so queries will not be executed or return no results even when they should be valid based on the information provided. The bot is also limited in it's capability, if the user asks questions it has not been programmed to answer it will not know what to do and will most likely send a reply that does not make sense in the context of the situation.

For logs to be recorded every machine that you wish to monitor must run winlogbeats in conjunction with logstash so errors can be sent to the mongo database to be checked, else the bot will fail as there will be no error logs to check.

The bot is also limited by our own machines as we could only run it locally with no server infrastructure, a business with server space could run the bot and it's modules easily but would need to change the urls in the config.yml and credentials.yml to match the hosting locations of duckling and the bot itself.