

BOT Platform – Overview

Release V 2.04
Feb-2021

Revision History

Version -2.0 Jan'21	- Websocket support to enable duplex data flow – Dec-2020
CR_2.01	Jan'21
CR_2.02	Feb'21
CR_2.03	Feb'21
CR_2.04	Feb'21
	- Support for scalable Kafka deployment
	- Now supports JDK15 and Jetty 9 and Jersey 2.33
	- DDF file structure has been changed (ITOs to Entities)
	- integrated rasa NLU into NLP engine

Overview

The BOT platform is client-server architecture, client being browser , mobile App or messenger tools like FB, Slack, Skype etc.

The BOT platform support REST as well as WebSocket using Jetty Server as a API gateway.

There are three major components of BOT engines -

- **DIALOG Engine (DE)** – This is core module, and it maintains the state of dialog with the end user
- **NLP Engine (NE)**– This module identifies intent and entities for given user utterances using advanced NLP algorithms
- **BROKER Engine (BE)**– This is communication module and handles messages between DE and NE and uses Kafka based messaging system.

The architecture is highly scalable and can be configured to handle large messaging payloads –

- One can configure multiple Kafka broker servers forming a multi-node cluster for handling large set of message volumes
- Configure number of partitions , replicas so as to distribute the TOPIC messages to different broker server
- Configure different consumerIDs within the groupID so as to distribute DE messages to different NLP Engine instances. (parallel processing)

Technology Stack

Dialog Engine – Open JDK 15, Jetty Server 9.4, Jersey 2.33, Kafka 2.6

NLP Engine – Python 3.7, (rasa, nltk, scikit_learn, kafka, flask)

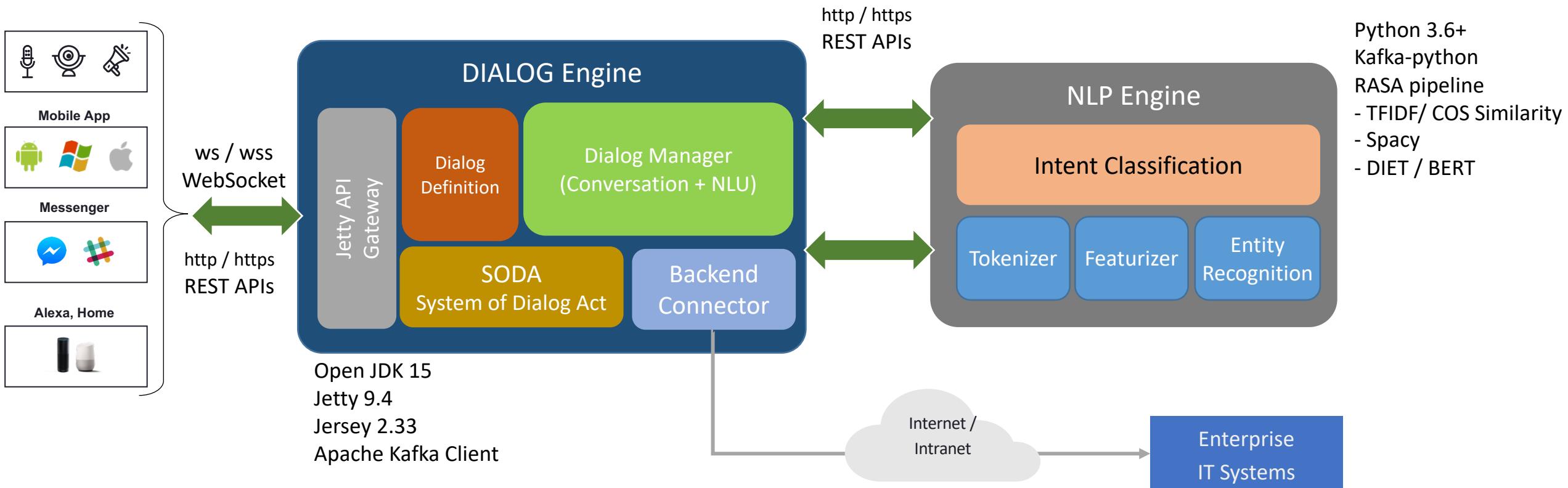
Broker Engine – Apache Kafka 2.6 (single or multimode cluster)

The details on how to download and deploy is available at

https://github.com/hmi-digital/Conversational_UI

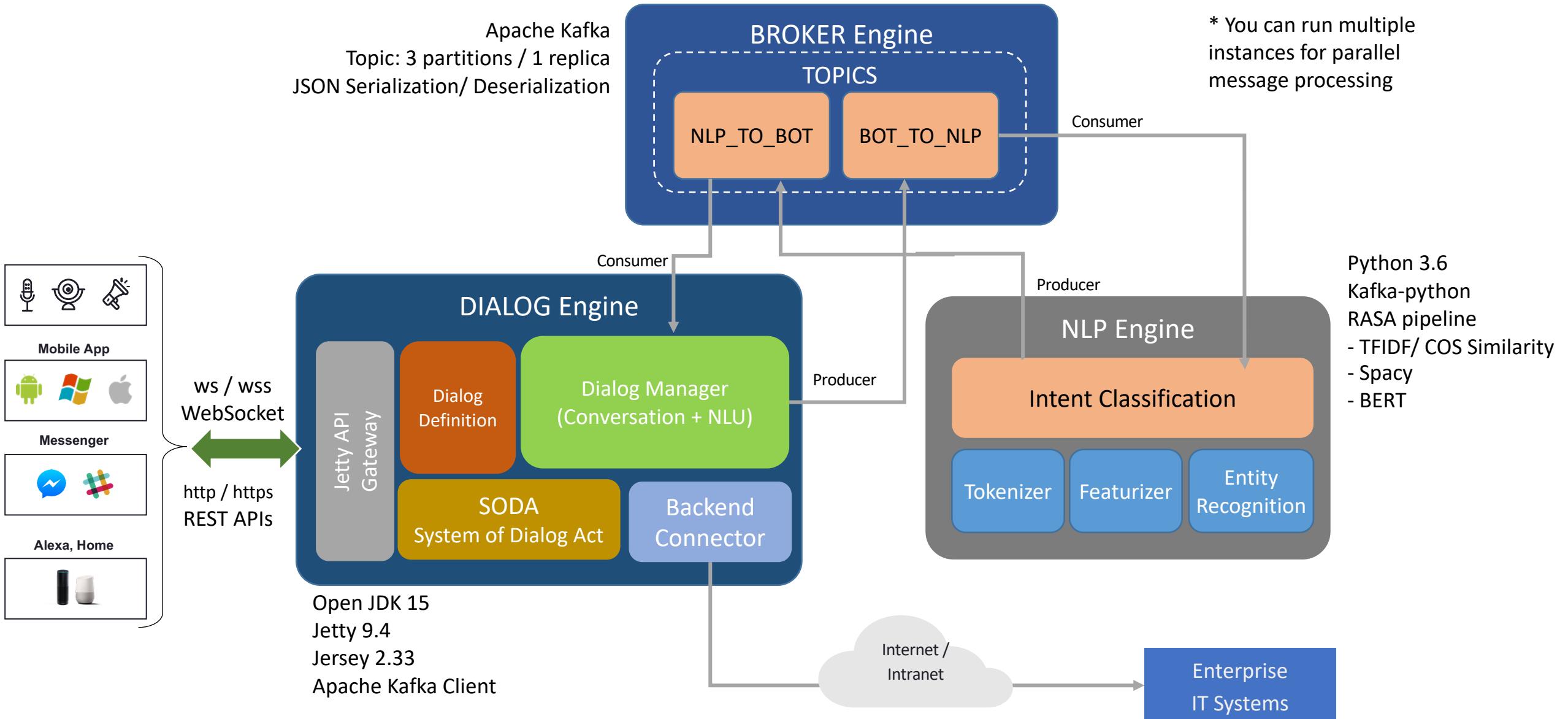
High Level Architecture

OPTION -1 REST API based communication
(recommended for development and testing purpose)



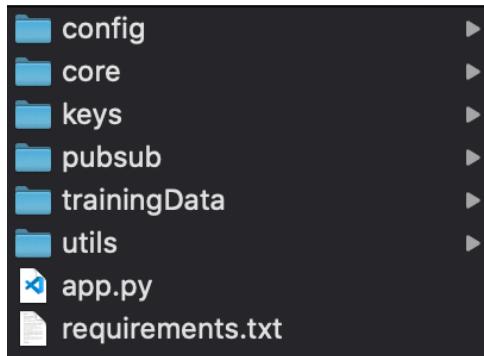
High Level Architecture

OPTION -2 Broker based messages for communication
(recommended for production)



Getting Started – NLP Engine

1. Important - Check all the prerequisite software packages are installed
2. Download the binaries from git repo and unzip. Locate NLPEngine folder



- Contains NLP engine configuration properties file (*nlp.properties*)
- NLP Engine train and predict algorithms are present in this folder
- SSL related private key, certificates and keystore file are stored here
- Source related to communication with Broker Service is stored here
- Training related data (intents and entities files)
- Utility python scripts
- This script runs NLP Engine service
- List of python modules required for installation of NLP Engine

PREREQUISITE –

- Java OpenJDK 15 and above
- Python 3.6 and above (RASA does not support 3.9)

3. Install all the required python modules from requirements.txt

```
$ python -m pip install -r ./requirements.txt
```

4. Load spacy models

```
$ python -m spacy download en_core_web_md  
$ python -m spacy link en_core_web_md en => Ignore - DeprecationWarning  
$ python -m spacy validate
```

4. Load nltk Data by executing commands as shown here.

```
$python  
>> import nltk  
>>nltk.download('punkt')  
>>nltk.download('stopwords')  
>>quit()
```

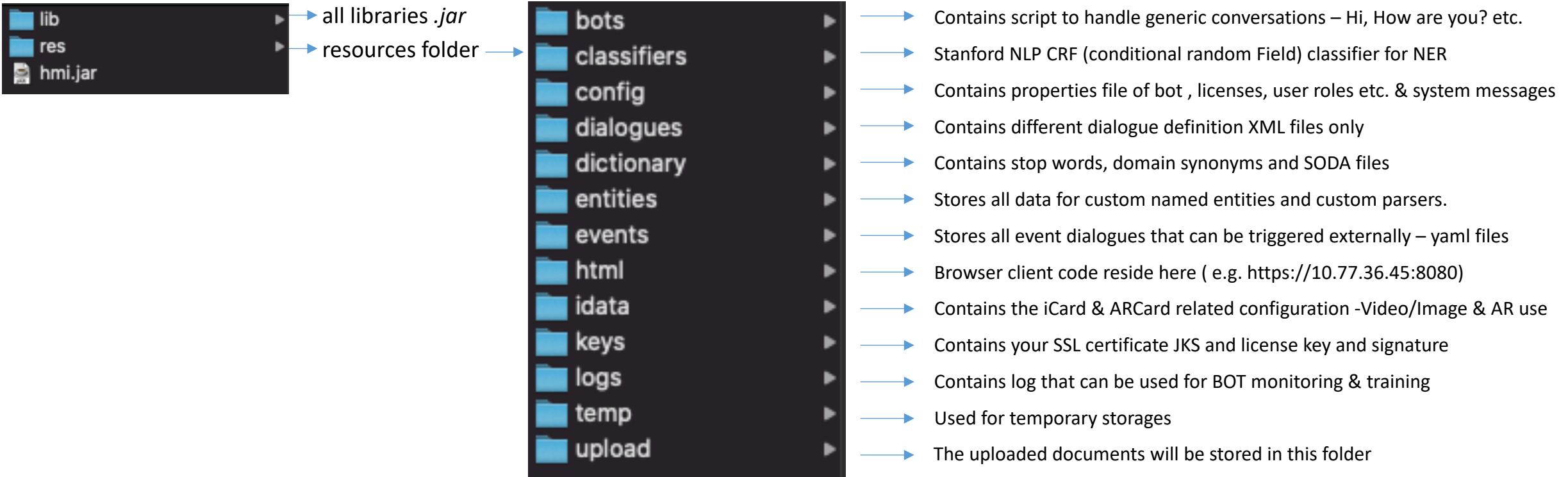
5. Run NLPEngine services

```
$ python app.py
```

kafka-python >= 2.0.2
rasa >= 2.3.1
spacy >= 3.0.3
nltk >= 3.5
flask >= 1.1.2

Getting Started – Dialog Engine

1. Locate DialogEngine folder. The folder structure should look like this -



2. Run the DialogEngine server either as option

- console as a standalone (for development and testing) or
- REST based Server (for production deployment)

\$ java –jar hmi.jar –i console –r trip → -i console => for CONSOLE options and –r trip is for running dialogue trip.xml

\$ java –jar hmi.jar –i rest –r trip –p 8080 → -i rest => for REST API options , –r trip is for running dialogue trip.xml and –p port (default is 8080 if not specified)

Getting Started – Dialog Engine

1. Test bot by running sample use case (trip booking bot) in rest option mode. (e.g. java –jar hmi.jar –i rest –r trip –p 8080)

```
$ java -jar hmi.jar -i rest -r trip_en
INFO (checkServerStatus): NLP_ENGINE up and running.      :|
Loading classifier from /Users/purushottam_d/Demo/DialogEngine//res/classifiers/english.all.3class.distsim.
INFO (ConvEngineProcessor): NER classifier loaded
INFO (ConvEngineProcessor): stopword processor instance created
No AIMLIF Files found. Looking for AIML
INFO (ConvEngineProcessor): AIML chat session instance created
[main] INFO org.eclipse.jetty.util.log - Logging initialized @2592ms to org.eclipse.jetty.util.log.Slf4jLog
WARNING (RESTInterface): Cross Origin Resource Sharing(CORS) is set to true. Potential security risk.
[main] INFO org.eclipse.jetty.server.Server - jetty-9.4.35.v20201120; built: 2020-11-20T21:17:03.964Z; git:
b; jvm 15.0.2+7-27
[main] INFO org.eclipse.jetty.server.session - DefaultSessionIdManager workerName=node0
[main] INFO org.eclipse.jetty.server.session - No SessionScavenger set, using defaults
[main] INFO org.eclipse.jetty.server.session - node0 Scavenging every 600000ms
Feb 01, 2021 10:14:44 AM org.glassfish.jersey.server.wadl.WadlFeature configure
WARNING: JAXBContext implementation could not be found. WADL feature is disabled.
[main] INFO org.eclipse.jetty.handler.ContextHandler - Started o.e.j.s.ServletContextHandler@33f98231
[main] INFO org.eclipse.util.ssl.SslContextFactory - x509=X509@1023150a(1,h=[],w=[])
for Server@3ece1e
/purushottam_d/Demo/DialogEngine/res/keys/hmi.jks,trustStore=null
[main] INFO org.eclipse.jetty.server.AbstractConnector - Started ServerConnector@799f10e1{SSL, (ssl, http/1,
[main] INFO org.eclipse.jetty.server.Server - Started @3754ms
INFO (RESTInterface): REST interface started on https://127.0.0.1:8080/
```

Note:- You can configure it to run as HTTP or HTTPS by changing bot.properties (refer – slide 27)

It will show the url and port details of jetty server on console

<http://192.168.0.106:8080/> - (as an example)

Now you can interact with bot either using

1- REST API

2- Web Socket (recommended as it provides a bidirectional asynchronous communication)

Getting Started - calling BOT server APIs using REST API

...1/3

1. To create a bot instance

Enter below details to your REST Client tool (postman or soapUI etc.)

Method – POST

URL - <https://10.88.246.15:8080/api/v2/bot>

Param - url-encoded-body parameter – key => user value =>John

key=> role value=>user ..(optional) if no role is given by default it is taken as "admin"

key => authToken value => <Token> (optional) if no authToken is given it is empty

POST https://{{BOT_SERVER_IP}}:{{BOT_SERVER_PORT}}/api/v2/bot

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> user	John	
<input checked="" type="checkbox"/> role	admin	
<input checked="" type="checkbox"/> authToken	yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQ...	

Body Cookies Headers (9) Test Results

Status: 201 Created Time: 998 ms Size: 1.37 Kt

KEY	VALUE
Date	Mon, 01 Feb 2021 04:47:08 GMT
Location	https://127.0.0.1:8080/api/v2/bot/d1-F5B4SZ4PWT71
Access-Control-Allow-Origin	*

Once this request is sent , BOT will create a unique URL for every user and returns it in HTTP header response.

This url provides the sessionId and used for following calls
<https://10.88.246.15:8080/api/v2/bot/d1-XX0S37ATGV75>

#curl -k -i -X POST --data-urlencode "user=John"
<https://10.88.246.15:8080/api/v2/bot>

Calling BOT server APIs using REST API

...2/3

2. Now that bot instance is created, you can initiate the conversation

Using the URL that was obtained in step -1 , call following method to initiate the conversation

Method – POST
URL - <https://10.88.246.15:8080/api/v2/bot/d1-XRHHZM5X6JR4>

Param - url-encoded-body parameter – key => userUtterance value => I want to book a ticket

Once this request is sent , Bot will process the userUtternace and will send JSON response with a reply -> “OK, Where do you want to go?”
speech -> JSONObject is provisioned for Alexa like devices or message that needs to be sent to TTS engine
It also sends status on entities and action

Meta-Information such as sessionId, user, source is used to log the details.
If bot has any backend notifications , it will show the number of messages. (See Appendix section for this)

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> userUtterance	I want to book a ticket	

```
1 {"timeStamp": "2021-02-01 10:20:29",  
2 "result": {  
3     "currentEntity": {  
4         "name": "getDestinationCity",  
5         "label": "City Name",  
6         "type": "sys.location.city",  
7         "value": ""  
8     },  
9     "speech": "where do you want to go?",  
10    "entities": [  
11        {  
12            "name": "getDestinationCity",  
13            "label": "City Name",  
14            "type": "sys.location.city",  
15        }  
16    ]  
17 }
```

Calling BOT server APIs using *REST API*

...3/3

3. Continue your conversation -

You can continue your conversation by sending the user utterance in “userUtterance” parameter

POST https://{{BOT_SERVER_IP}}:{{BOT_SERVER_PORT}}/api/v2/bot/{{BOT_SESSION_ID}}

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

<input checked="" type="checkbox"/> userUtterance	I want to go to London City	
Key	Value	Descrip

Body Cookies Headers (8) Test Results Status: 200 OK

Pretty Raw Preview Visualize JSON

```
1
2   "timeStamp": "2021-02-01 10:34:55",
3   "result": {
4     "currentEntity": {
5       "name": "getNumberOfPersons",
6       "label": "No of persons",
7       "type": "custom.slider_1",
8       "value": ""
9     },
10    "speech": "for how many persons?",
11    "entities": [
12      {
13        "name": "getNumberOfPersons",
14        "label": "No of persons",
15        "type": "custom.slider_1",
16        "value": ""
17      }
18    ]
19  }
```

Once this request is sent , Bot will process the userUtternace and will send JSON response with a reply

B: Good evening, How may I help you?
U: Good evening
B: Good evening, Friend, hope you are doing good today. How may I help you?
U: I want to book a ticket
B: Please tell me what city do you want to travel?
U: I want to start from Atlanta
B: And for how many persons?
U: How is weather in Atlanata?
B: The temperature in Atlanta is 21.54 degrees. for how many persons?
U: for 4 persons
B: When do you want to leave?
U: I want to leave on Sunday
B: which class do you want to travel?
U: Business
B: Hey John, this trip to Atlanta costs 255 Dollars. What can I do for you?

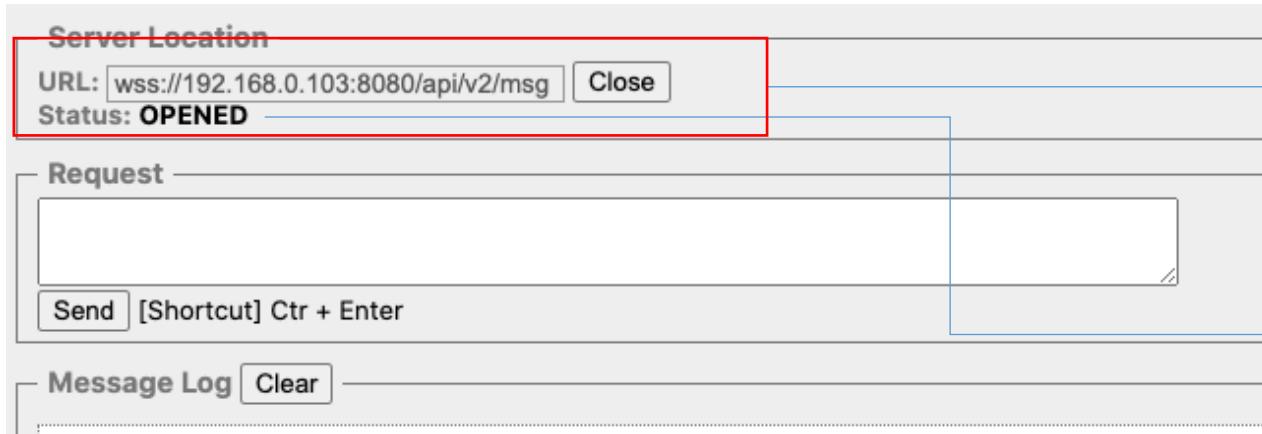
If it has executed any task it will populate the details (intent name and entities) in action JSON object.

Calling BOT server using WebSocket

...1/2

1. To create a bot instance we need to OPEN the web socket first

Enter below details to your WebSocket Client tool (Simple WebSocket Client extension in Chrome Browser)

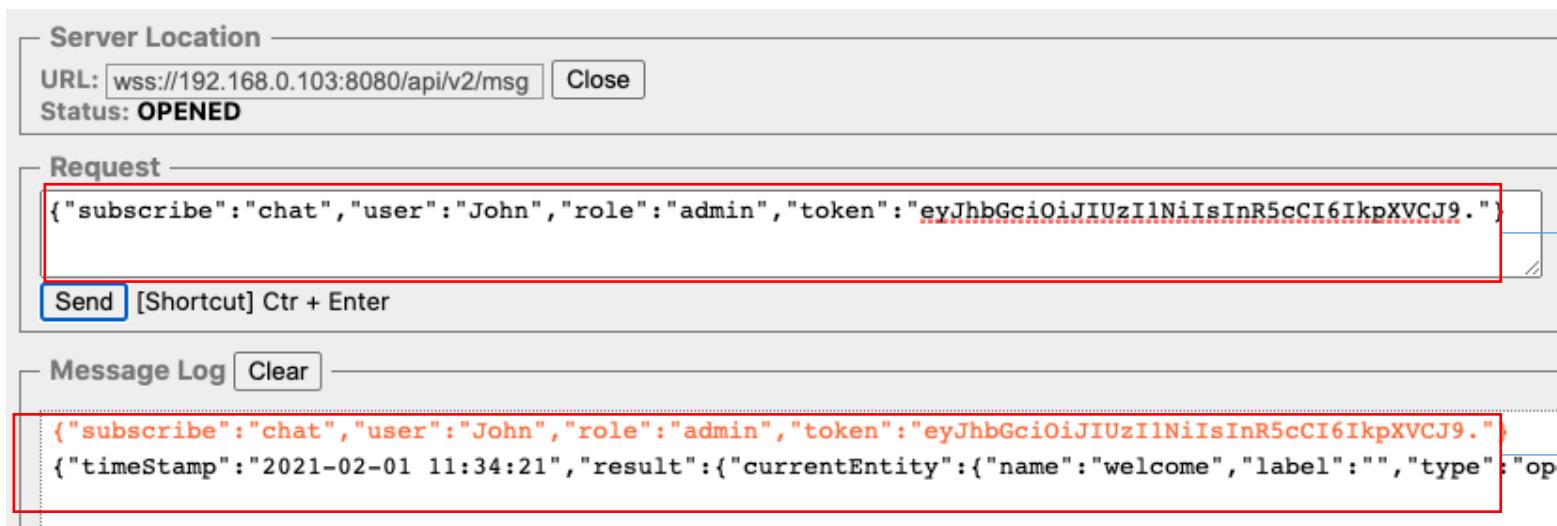


Use secure websocket
wss://192.168.0.101:8080/msg -> for HTTPS
bot server

ws://192.168.0.101:8080/msg -> for HTTP
bot server

Status should show “OPENED”

NOTE – if you are using self signed certificate
please accept the warning of untrusted certificate
in browser before opening the wss socket
connection.



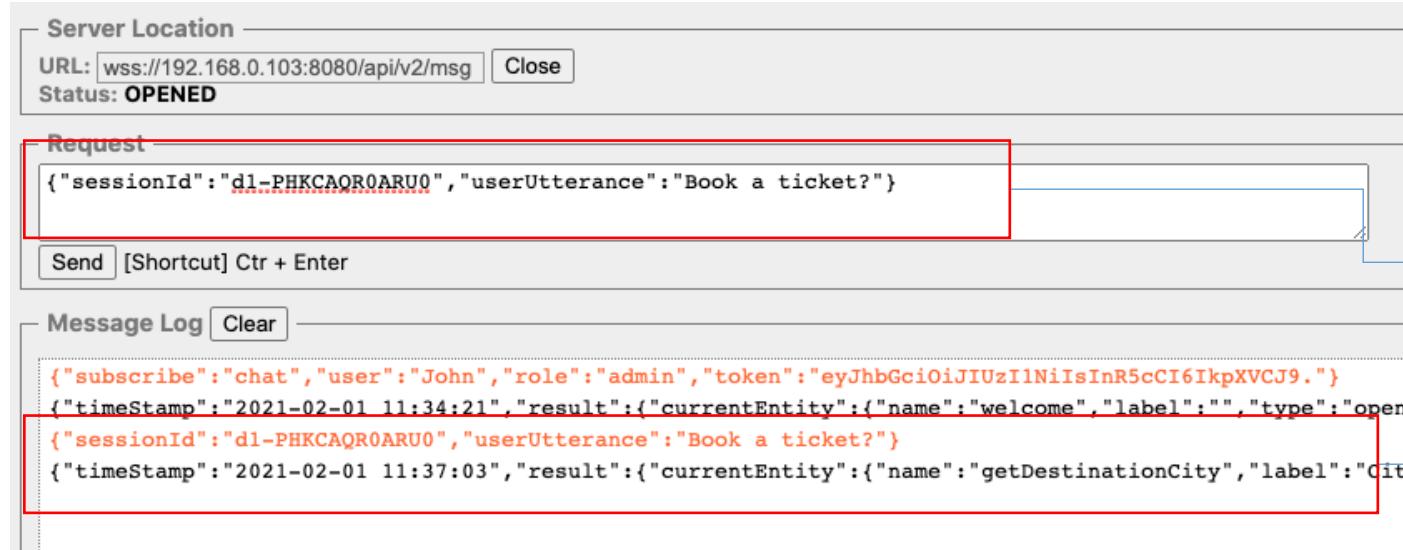
Message Request with user, role and token–
{"subscribe": "chat", "user": "John", "role": "admin",
"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9."}

Bot would respond with a message that gives the
details of Bot instance created . Note the unique
sessionId that has been created.
sessionId -> d1-PHKCAQR0ARU0

Calling BOT server using WebSocket

...2/2

3. To interact with Bot , send the message request with sessionId (see step-2) in the format show below –
- ```
{"sessionId":"<sessionId>","userUtterance":"<userUtterance>"}
```

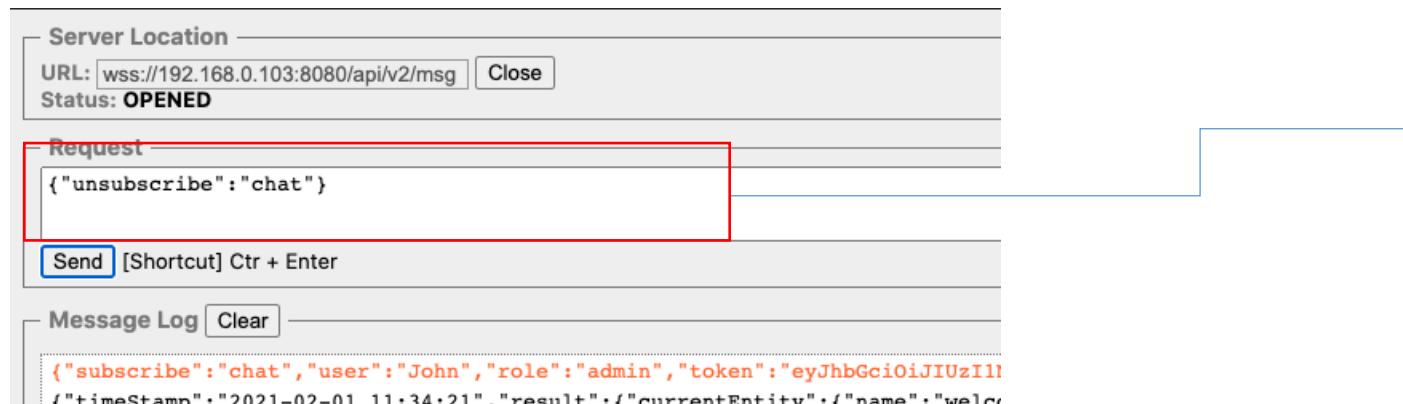


To interact with Bot send request as per the format e.g.

```
{"sessionId": "d1-PHKCAQR0ARU0", "userUtterance": "Book a ticket?"}
```

Bot would respond with a reply that you can process at client side.

4. To close the bot session, send the message request in the format shown below –
- ```
{"unsubscribe": "chat"}
```



This will disconnect the web socket and bot instance will be removed.

Dialog Definition file (DDF)

```
<?xml version="1.0" encoding="UTF-8"?>
<n:dialog xsi:schemaLocation="http://cto.net/hmi schema1.xsd" xmlns:n="http://cto.net/hmi/1.0"
instance" name="trip" company="ABCCorp" version="1.0">
  <startTaskName>start</startTaskName>
  <globalLanguage>en</globalLanguage>
  <useSODA>true</useSODA>
  <allowSwitchTasks>true</allowSwitchTasks>
  <allowOverAnswering>true</allowOverAnswering>
  <allowDifferentQuestion>true</allowDifferentQuestion>
  <allowCorrection>false</allowCorrection>
  <failureAttempts>2</failureAttempts>
  <tasks>
    <!--#DO NOT CHANGE System Task. Acts as a starting task for all the conversations -->
    <task name="start" label="Initial Task"> ... </task>
    <task name="getTripInformation" label="Book ticket"> ... </task>
    <task name="getWeatherInformation" label="Weather information"> ... </task>
    <task name="getWikipediaCityInfo" label="City information"> ... </task>
    <!--#DO NOT CHANGE System Task. Invoked when user explicitly wants to cancel the task
    <task name="cancelTask" label="Cancel"> ... </task>
    <!--#DO NOT CHANGE System Task. Invoked whenever there are two consecutive failed attempts
    <task name="helpTask"> ... </task>
    <!--#DO NOT CHANGE System Task. Invoked whenever number of failed attempts exceeds ALL
    <task name="handoverTask"> ... </task>
    <!--#DO NOT CHANGE System Task. Invoked whenever user says bye , bye bye or exit at start
    <task name="exitTask"> ... </task>
  </tasks>
</n:dialog>
```

Domain or Dialog name , Company name , dialog version

Global Flags for defining Dialog behavior

Different Tasks that BOT needs to perform

"start", "cancelTask", "helpTask", "handoverTask" and "exitTask" are reserved tasks used for

- Cancelling the current task (user explicitly want to cancel the task)
- Providing help to user in case there are consecutive failed attempts in answering entity as configured in <failureAttempts>
- Coming out of current task and to handover the chat from bot to human respectively
- If user says (bye, exit or bye bye) this task will get final confirmation from user before logout.

DDF - Global Flags

```
<startTaskName>start</startTaskName>
<globalLanguage>en</globalLanguage>
<useSODA>true</useSODA>
<allowSwitchTasks>true</allowSwitchTasks>
<allowOverAnswering>true</allowOverAnswering>
<allowDifferentQuestion>true</allowDifferentQuestion>
<allowCorrection>false</allowCorrection>
<failureAttempts>2</failureAttempts>
```

```
resulted in: seek[0.0428] action[0.0101] prov[0.9471] -> prov
```

startTaskName – This specify the initial task to be executed by BOT. Initial task may be other than "start" but after execution the initial task BOT will return to start task.

globalLanguage – This specifies the BOT language. This has to be in ISO language code. (e.g. English – en, Hindi – hi, Danish-da, Dutch-nl, Swedish-sv Arabic-ar Spanish – es etc.)

useSODA (System Of Dialogue Act) – Each utterance will be classified using Max Entropy Classifier to find whether user is => seeking the information **or** => Providing information **or** => issuing the defined command (e.g. switch ON, Switch OFF)

If set to true, dialog manager will check if the user is seeking the information if yes, will switch the task for its fulfillment.

allowSwitchTasks – if set to true it will allow sub-dialogues from other tasks , if set to false only sub-dialogues of same task will be executed. (e.g. for trip.xml DDF use case - User can only answer questions specific to getTripInformation and will not allowed to switch to getWeatherInformation task)

allowOverAnswering - if set to true the user is allowed to provide more than the information that has been asked for (but at least the current question) (e.g. for trip.xml – user while in getTripInformation can provide information like “want to go to London for four people” which will fill entities for getDestinationCity and getNumberOfPersons)

allowDifferentQuestion - if set to true the user can ignore current question and answer a different unanswered question (e.g. for trip.xml – user while in getTripInformation task can say “I want to leave tomorrow” when Bot is actually asking about “where do you want to go?”)

allowCorrection - if set to true user can change a value of an already asked question (e.g. for trip.xml – user while in getTripInformation can say “I want to go to Paris” when Bot is asking “for how many persons”. This will refill the getDestinationCity with new value “Paris”)

failureAttempts – is an integer number which is number of consecutive attempts that will be allowed to user to answer specific entity. In case he/she fails to answer, he/she will be redirected to “helpTask”

DDF – Task definition

```
<task name="getWeatherInformation" label="Weather information">  
  <entities>  
    <entity name="getDestinationCity" label="Weather information">  
      <answerType>sys.location.city</answerType>  
      <fallbackQuestion>for which city do you want to know the weather?</fallbackQuestion>  
      <clarifyQuestion>please provide me with correct city name?</clarifyQuestion>  
      <required>true</required>  
      <useContext>true</useContext>  
      <clearContext>true</clearContext>  
    </entity>  
  </entities>  
  <action>  
    <httpAction>  
      <returnAnswer>true</returnAnswer>  
      <utteranceTemplate>The temperature in %getDestinationCity is #result degrees</utteranceTemplate>  
      <method>get</method>  
      <params>q=%getDestinationCity&mode=xml&units=metric&APPID=706120eb0f11f2ebf8584dadb9bad3d6</params>  
      <url>http://api.openweathermap.org/data/2.5/weather</url>  
      <xpath>/current/temperature/@value</xpath>  
      <jpath></jpath>  
    </httpAction>  
  </action>  
  <followup>  
    <entity name="anotherOne" label="Another City">  
      <answerType>sys.decision</answerType>  
      <fallbackQuestion>do you want to know about other cities?</fallbackQuestion>  
      <required>true</required>  
    </entity>  
    <answerMapping>  
      <item key="YES">getWeatherInformation</item>  
    </answerMapping>  
  </followup>  
</task>
```

Unique task name (intent)

These are multiple entity within entities tag
The slots are filled based on the "answerType"

This is a fallback question asked to user if slot is not filled
and if "required" flag is true.
You can give multiple options by using "|" separator. You
can also use entity by preceding the entity name with %.
e.g. for which city do you want to know the weather? |
Hey, %loginUser_ what city would you want
temperature?

(optional) This is clarifying question which bot will ask in
case user fails to provide correct answer. (if not provided,
bot will reply - Sorry, I did not understand that. Please try again.
for which city do you want to know the weather?)

(optional) If useContext flag is set to true , it will fill this
entity automatically , if it has been already answered in
any of earlier conversation. (refer Appendix 6)

(optional) this is follow-up section and will be triggered
after task is executed. You can provide redirection to
different tasks based on value of follow-up entity.

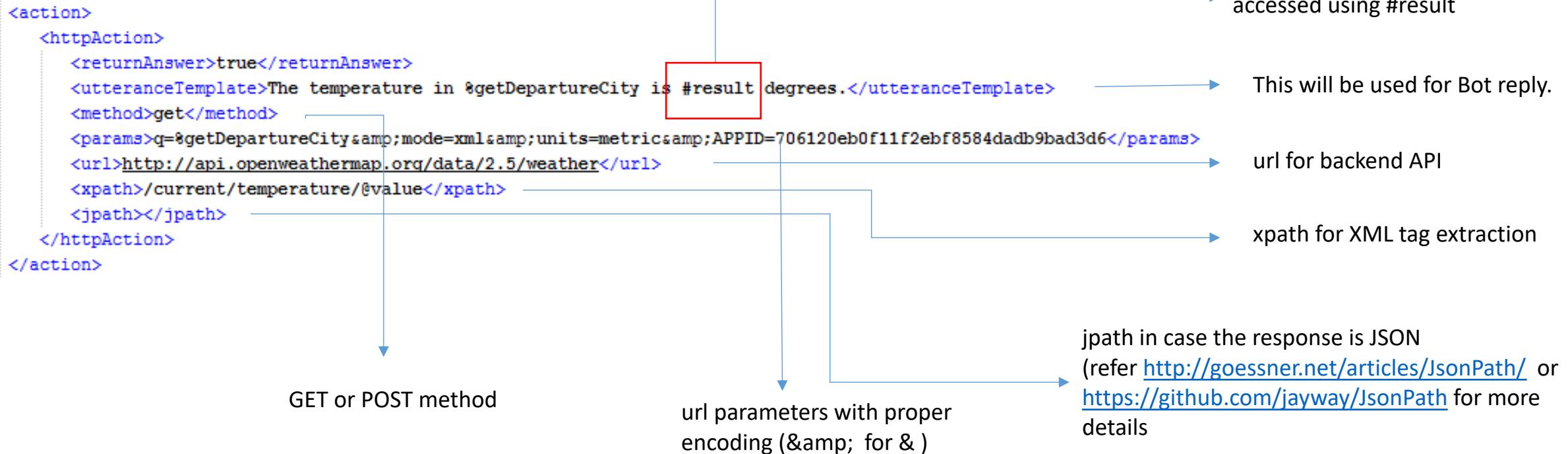
Global ENTITIES

If entity name ends with "_" then this is treated as global entity and will be passed to every task.

The system also provides loginUser_, loginRole_, authToken_, sessionId_, lastAccessedTask_ and lastAccessedEntity_ as default
global entities and any task can use these by referring it as %loginUser_, %loginRole_, %authToken_, %sessionId_,
,%lastAccessedTask_, %lastAccessedEntity_ respectively

DDF – Action – http Action

1. Action using REST based API



DDF – Action - groovy Action with Result Mapping

2. Supports Groovy Action with result mapping

```
<action>
  <groovyAction>
    <resultMappings>
      <resultMapping>
        <message/>
        <redirectToTask>homeLoanEligibility</redirectToTask>
        <resultValue>1</resultValue>
        <resultVarName>action</resultVarName>
      </resultMapping>
      <resultMapping>
        <message/>
        <redirectToTask>homeLoanMenu</redirectToTask>
        <resultValue>2</resultValue>
        <resultVarName>action</resultVarName>
      </resultMapping>
    </resultMappings>
    <returnAnswer>true</returnAnswer>
    <utteranceTemplate/>
    <code>
      <![CDATA[
        Integer res = new Integer(2);
        String type=new String(frame.get("getOkInfo"));
        if (type.matches("(?i)^.*?\b(ok|OK|Okay)\b.*?"))
          res = 1;
        else if (type.matches("(?i)^.*?\b(info|Info|INFO)\b.*?"))
          res = 2;
        executionResults.put("action",res.toString());
        executionResults.put("getOKinfo","processing");
      ]]]
    &lt;/code&gt;
  &lt;/groovyAction&gt;
&lt;/action&gt;</pre>
```

Task redirection based on the result of groovy action
Or could be http-based action.

The #action value will decide the new task to be created.(API
result is always stored in result variable)

NOTE- you can access the variable using # in
utteranceTemplate tag. (e.g. #message)
Do not use result as variable as it is internally used in code.

One can use frame.get("<Entity Name>") method to read the
entity value in Groovy action.

You can modify the entity value in groovy action by using
executionResults.put ("<Entity Name>","<value>")
Note- The changed value will be reflected in subsequent task if
referred.

**You can define a globalEntity here that can be accessed in any
other task. (note – Entity name should end with '_')
e.g. executionResults.put("loanOffered_","Yes");**

DDF – groovy Action → sending predefined response

Best use of Groovy action is whenever you want to send the standard responses with no back end API call.
You will have to write groovy action if you need to make an API call.

```
<action>
  <groovyAction>
    <returnAnswer>true</returnAnswer>
    <utteranceTemplate>Anthony C Lopez - MD FACC is the closest among all which is at 173 North Morrison Ave, San
    Jose, CA 95126, Phone number (408) 293-1088.</utteranceTemplate>
    <code></code>
  </groovyAction>
</action>
</task>
```

Predefined answer with no
backend call.

OR

```
<fallback_question>additional info that you want to provide ?</fallback_question>
<required>true</required>
</ito>
</itos>
<action>
  <groovyAction>
    <returnAnswer>true</returnAnswer>
    <utteranceTemplate>Hey %loginUser_ , this trip to %getDestinationCity costs #price Dollars.
    info %getInfo</utteranceTemplate>
    <code><![CDATA[executionResults.put("price","255")]]></code>
  </groovyAction>
</action>
</task>
```

The code block `<![CDATA[executionResults.put("price","255")]]>` is highlighted with a red oval.

DDF - action with follow up question

Supports Follow up question post executing the action

```
<action>
  <httpAction>
    <returnAnswer>true</returnAnswer>
    <utteranceTemplate>#result</utteranceTemplate>
    <method>get</method>
    <params>format=json&&action=query&&prop=extracts&&explaintext&&exsentences=1&&titles=%getWikiCity</params>
    <url>http://en.wikipedia.org/w/api.php</url>
    <xpath></xpath>
    <jpath>$.extract</jpath>
  </httpAction>
</action>
<followup>
  <entity name="anotherOne" label="Another City">
    <answerType>sys.decision</answerType>
    <fallbackQuestion>do you want to know about other cities?</fallbackQuestion>
    <required>true</required>
  </entity>
  <answerMapping>
    <item key="YES">getWikipediaCityInfo</item>
  </answerMapping>
</followup>
</task>
```

Follow-up question will be asked immediately after executing the task.

Use Uppercase. This KEY will be matched with entity value. (sys.decision)

DDF – prebuilt entities

...1/2

List of entity types available in the platform

City	=> sys.location.city
Time	=> sys.temporal.time => 1:00 AM, morning, evening etc.
Date	=> sys.temporal.date
Person	=> sys.person
Mail	=> sys.mail
Contact	=> sys.contact => (+1 234-567-890)
First Name	=> sys.person.firstname
Last Name	=> sys.person.lastname
Password	=> sys.password => (<i>Min 8 chars, at least 1 uppercase letter, 1 lowercase letter, 1 number and 1 special Char</i>)
Dummy	=> dummy => captures all free text and passes it to entity
Organization	=> sys.organization
QA Parser	=> sys.corpus.qa (for checking the answer in domain corpus)
Yes/No Parser	=> sys.decision => results in YES or NO
ON/OFF Parser	=> sys.onoff => results in ON or OFF
Custom Buttons	=> custom.button_X => (X is prefix used to match the file in /res/entities folder)
Custom List	=> custom.item_X => (X is prefix used to match the file in /res/entities folder)
Custom List	=> custom.urlList_X => (X is prefix used to match the file in /res/entities folder) (in case of multiple selection items use below ones)
Custom List	=> custom.multiItem_X =>=> (X is prefix used to match the file in /res/entities folder)
Custom List	=> custom.multiUrlList_X =>=> (X is prefix used to match the file in /res/entities folder)
Custom Parser	=> custom.pattern_X (X is prefix used to match the file based on REGEX pattern)
Custom Classifier	=> custom.classifier_Y (Y is domain name) => This is ML based text classifier
Menu List	=> custom.menu_X => (X is prefix used to match the file in /res/entities folder)
Open Text	=> sys.opentext => capturing free text .. (<i>used in conjunction with interactive widgets, to be used in conjunction with other entity</i>)

This is used to fill the entity in case the domain corpus has answer to the user question. This will use bot.properties file located at /res/config

If USE_QACHECK=True it will call QA_URL => which should return True or False based on if it has answer to the question

[Important –

Method => POST

url => QA_URL

Parameters => userUtterance="what is ATM"?

Response =>

{"response": "true"} or {"response": "false"}

]

If answer is “YES” this entity will be filled and you can use <Action> to call the appropriate URL.

QA Parser is treated at entity level as against

USE_DOMAIN flag which is treated at intent level (see slide – 27)

It will look for corresponding file in /res/entities folder.
e.g. custom.item_accountType -> will be associated with item.accountType.txt file for parsing

Used to provide list of menu options for user to click.

See slide - 44

DDF – prebuilt entities

...2/2

Generic Number
Scaling Number
Floating Number

=> sys.number
=> sys.number.scale
=> sys.number.float

Slider Number

=> custom.slider_X -> X (from 1 onwards)
This is used to take range-based number input (float or int) from user. The range is defined in /res/entities/slider_1.txt file.

Note – the sequence of entering number has to be MIN,MAX,STEP and DEFAULT

```
##DO NOT REMOVE THIS LINE - SLIDER #MIN #MAX #STEP #DEFAULT
9.0
15.0
0.1
10.0
```

Important –

For better slot filling, following files are available in /res/entities –

firstName.txt	=> sys.person.firstname
lastName.txt	=> sys.person.lastname
organization.txt	=> sys.organization
location.txt	=> sys.location.city

One two three	=> 123
Forty One Hundred	=> 4100
1,239	=> 1239
23545	=> 23545
1 hundred	=> INVALID
Hundred thousand	=> INVALID
1.2 L	=> 120000
1.2 Lac	=> 120000
10 K	=> 10000
1,239	=> 1239
23545	=> 23545
.2L	=> INVALID
1 hundred	=> INVALID
Hundred thousand	=> INVALID
1.256	=> 1.256
123	=> 123
.234	=> INVALID
10 K	=> INVALID
1,239	=> INVALID
1 hundred	=> INVALID
Hundred thousand	=> INVALID

There is often a need to fill the slot that are business specific and are not available as part of standard NER. This can be achieved using custom named entities. Platform supports different ways of creating custom entities

=> If the single items need to be filled –

- *custom_item_1* or *custom_urlList_1* or *custom_pattern_1*

⇒ If multiple Items needs to be filled –

- *custom_multitem_1* or *custom_multiUrlList_1*

1. Custom List

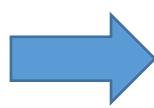
Create *item_1.txt* file in /res/entities folder

```
<entity name="getClass" label="Class">
    <answerType>custom.item_1</answerType>
    <fallbackQuestion>which class do you want to travel?</fallbackQuestion>
    <clarifyQuestion>Is it business , economic or first?</clarifyQuestion>
    <required>true</required>
</entity>
</entities>
```



```
##DO NOT REMOVE THIS LINE - to check
Economy
First
Business
```

If the file has name value pair (item=category) as shown, the entity will be filled with category when item appears in utterance.



```
##DO NOT REMOVE THIS LINE - list of a
truck=auto
car=auto
honda=auto
ford=auto
chevrolet=auto
mercedes=auto
chrysler=auto
genral motors=auto
audi=auto
```

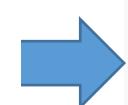
2. Custom Url List – In this we get the list of items fetched from external API. Following steps are taken to support this method

- Create *urlList_X.txt* in */res/entities* folder. This file specifies parameters specific to API
- The API need to responds with the list of items in csv format. e.g.

```
{"type":"userIDs",
"list":"johnd, jeevanK,mathew"
}
```

```
<entity name="selectUser" label="User Name">
<answerType>custom.urlList_1</answerType>
<fallbackQuestion>what is user name?</fallbackQuestion>
<required>true</required>
</entity>
```

Create *urlList_1.txt* file in */res/entities* folder



```

1 #The item list will be populated from below URL properties
2 URL_METHOD=GET
3 URL=http://api.openweathermap.org/data/2.5/weather
4 PARAMS=q=%getDestinationCity&mode=json&APPID=706120eb0f11f2eb
5 JPATH=$..list → json object that contains list
6 CACHE_DAYS=0 → No of days for which data can cached. If cache is
                     not required set it to "0"

```

(Optional) Contains url parameters(if any). You can refer to entity value by using %<EntityName> (e.g. %getDestinationCity). It will replace with entity value ,if already filled in. (this is not yet implemented)

3. Custom Button List – This is like list view but has been categorized as button list to show it as a button selection for better UI. (as against drop down list selection)

- Create *button_1.txt* in */res/entities* folder.

```
<entity name="getClass" label="Class">
  <answerType>custom.urlList_1</answerType>
  <fallbackQuestion>what class would you like to travel?</fallbackQuestion>
  <required>true</required>
</entity>
```

Create *button_1.txt* file in */res/entities* folder

```
1 ##DO NOT REMOVE THIS LINE - to check if it is new or list
2 Economy
3 First
4 Business|
```

IMPORTANT -

Custom Menu List – There is similar option of creating button menu using *custom.menu_X Entity*.

The difference being “button” entity is part of form and “menu” entity is only for selection of sub menu items explained on slide -44

4. Custom Parser

You can provide list of REGEX pattern that will be used to extract entities from user utterance

```
<entity name="getTime" label="Claim Date">
    <answerType>custom.pattern_1</answerType>
    <fallbackQuestion>what date you applied for your claim?</fallbackQuestion>
    <required>true</required>
</entity>
```

Create pattern_1.txt file in /res/entities folder



```
#DO NOT REMOVE THIS LINE - This pattern will find date in DD Month YYYY
\b[0-9]+\s[a-zA-Z0-9-]+\s[0-9-]{4}+\b
```

DDF - Creating custom entities - multiple items (item & urlList based)

...5/6

At times , we need the entity that need to be filled with multiple items.

e.g. what devices are owned by user – entity can fill in one or more items from list (e.g., Mobile, Desktop, VOIP, Laptop etc.)

You can use *custom.multipleItem_X* or *custom.multipleUrlList_X* to extract entities from user utterance

```
<entity name="getMenu" label="Menu">
  <answerType>custom.multiItem_1</answerType>
  <fallbackQuestion>food items that you want to order?</f
  <required>true</required>
</entity>
```



Create *multiItem_1.txt* file in */res/entities* folder

```
##DO NOT REMOVE THIS LINE
Breakfast = BF
Lunch = LCH
Dinner = DNR
Desert = DST
```

```
<entity name="getDevices" label="Devices">
  <answerType>custom.multiUrlList_1</answerType>
  <fallbackQuestion>list of devices registered for you?</fall
  <required>true</required>
</entity>
```



Create *multiUrlItem_1.txt* file in */res/entities* folder

```
1 #The item list will be populated from below URL properties
2 URL_METHOD=GET
3 URL=http://www.mocky.io/v2/5b49de1131000072138bc086
4 ➤ PARAMS=type=%getType&lang=en
5 JPATH=$..list
6 CACHE_DAYS=0
```

(Optional) Contains url parameters. It
will replace it with entity value ,if already
filled in.

JSON response of this API -
{ "type": "deviceTypes", "list": "mobile,desktop,pc,laptop,voip" }

DDF - Creating custom entities - ML based text classifier (only for ENGLISH) ...6/6

At times , we need entity capable of classifying user text to one of the classes.

e.g. if one has to classify nature of ticket when user is asked to describe the problem. As an example “I am facing logging into mail outlook” can be classified as “login” class.

Follow below steps to create ML based classifier entity.

1. Define your classes in json format as shown.
2. Save your file with proper naming convention in
`/res/entities/classifier/data/classifier_<domain name>.json` (e.g. `classifier_ticketType.json`)
3. Set your threshold score that would qualify the class in
`/res/entities/classifier/config/classifier.properties` #stores configuration
`THRESHOLD_SCORE=0.6`
4. Now you can use this classifier in your DDF file as shown. Whenever the user utterance matches with any of the class (with confidence score \geq threshold score) the entity will get filled.
5. Please note the entity value will contain “Class:<name> Utterance:<user utterance>” e.g.
“Class:**software** Utterance:Assignment of ticket to user not working”.
6. **Important:** please delete all the models whenever you update the classes in
`classifier_<domain name>.json` file.

Keep “allowDifferentQuestion” and “allowCorrection” flag to false to keep minimal collision with other entities

```
<useSODA>true</useSODA>
<allowSwitchTasks>true</allowSwitchTasks>
<allowOverAnswering>true</allowOverAnswering>
<allowDifferentQuestion>false</allowDifferentQuestion>
<allowCorrection>false</allowCorrection>
```

```
{
  "domain": "ticketType",
  "classes": [
    {
      "name": "login",
      "utterances": ["I am not able to login into system",
        "user and password is not working on outlook",
        "authentication failed for outlook",
        "can not login into the system",
        "user id and password are not accepted",
        "login issue"]
    },
    {
      "name": "software",
      "utterances": ["Not able to generate report in remedy",
        "The service ticket is not appearing",
        "Ticket assignment is not working",
        "fail to export the file to local folder"]
    },
    {
      "name": "hardware",
      "utterances": ["Printer is not working",
        "Paper jam in printer",
        "keyboard is malfunctioning",
        "Monitor is flickering",
        "Desktop is powering up"]
    }
  ]
}
```

```
<entity name="getIssue" label="Describe Issue">
  <answerType>custom.classifier_ticketType</answerType>
  <fallbackQuestion>describe the nature of problem?</fallbackQuestion>
  <required>true</required>
</entity>
```

NLP Engine - training the BOT

..1/3

NLP Engine(NE) provides ways to define different Intents and train it for possible utterances. NE uses state-of-the-art NLP algorithms. One can configure `/config/nlp.properties` to support following algorithms –

TFIDF – (supports **JSON or .md** format only)

NLU - (RASA based algorithms supports **.md and .yml** format only)

The training data is stored in `/trainingData/intents` folder and uses different formats (**.md or .yml** format is recommended)

1- JSON Format (only for TFIDF)

In order to train BOT follow below steps.

- create `<domain>_<language>.json` file (e.g., `trip_en.json`) with following JSON structure

- here is the basic structure of NLU training data

```
{ "domain":<domain>,
  "tasks":[
    { "name":<taskName>,
      "utterances":[]
    }
  ]
}
```

- copy this file to `/trainingData/intents` folder

- Ensure you use same intent (task name) and entities as defined in your DDF.

In case you add any new utterance to JSON file, you need to retrain the NLP Engine by restarting the DialogEngine. (it will train the model every time you start it)

→ "en" for ENGLISH
"hi" for HINDI
"mr" for MARATHI

```
{
  "domain": "trip",
  "tasks": [
    { "name": "getTripInformation",
      "utterances": ["I want to book a tick",
        "I want to travel",
        "Can you book me on plane",
        "Need a reservation of ticket",
        "Can you do a reservation for me",
        "Need to plan a trip"]
    },
    { "name": "getWeatherInformation",
      "utterances": ["How is weather",
        "Tell me the temperature",
        "Please tell me temperature in city",
        "Will it be cold in city"]
    },
    { "name": "getWikipediaCityInfo",
      "utterances": ["Can you tell me more",
        "Need to know wiki details of city",
        "Please tell me more about",
        "Tell me more about",
        "can you tell me background details o"]
    },
    { "name": "cancelTask",
      "utterances": ["cancel this task",
        "I want to exit this",
        "Get me out of this task",
        "cancel this process",
        "I want to end this process",
        "get me out of this process"]
    }
  ]
}
```

2- Markdown Format (.md) – (supports both TFIDF and NLU)

- create `<domain>_<language>.md` file (e.g., `trip_en.md`) with following JSON structure
- here is the basic structure of NLU training data
 - `## intent:getTripInformation`
 - *I want to book a ticket <!-- no entity -->*
 - *Book me a ticket to [London](destinationCity) <!-- entity "destinationCity" has value "Mumbai" -->*
 - *Please reserve me a ticket for [next day]{ "entity": "startDate", "value": "tomorrow" } <!-- synonyms, method 1-->*
 - *Please book me a ticket in (business) [getClass]class? <!-- entity matched by lookup table -->*
 - *Please reserve me a ticket for [NYC](destinationCity)*
 - `## synonym:New York`
 - *NYC*
 - *NewYork*
 - `## lookup:getClass`
 - *business*
 - *economy*
 - *first*
 - `## regex:zipcode`
 - *[0-9]{5}*
 - `## lookup:currencies <!-- specify lookup tables in an external file -->`
`path/to/currencies.txt`
 - copy this file to `/trainingData/intents` folder
 - Ensure you use same intent (task name) and entities as defined in your DDF.

In case you add any new utterance to JSON file, you need to retrain the NLP Engine by restarting the DialogEngine. (it will train the model every time you start it)

You can convert the training data from one format to other using available tool in rasa.

`$ rasa data convert nlu --data data/nlu.md --out data/nlu.json -f json`

3- YML Format (.yml) → (supports only NLU)

- create `<domain>_<language>.yml` file (e.g., `trip_en.md`) with following JSON structure
- here is the basic structure of NLU training data

```
version: "2.0"
nlu:
  - intent: getTripInformation
    examples: |
      - I want to book a ticket <!-- no entity -->
      - Book me a ticket to [London](departureCity) <!-- entity "deaprtureCity" has value "Mumbai" -->
      - Please reserve me a ticket for [NYC>{"entity": "departureCity", "value": "New York"} <!-- synonyms, method 1-->
      - Please book me a ticket in (business) [getClass]class? <!-- entity matched by lookup table -->
  -synonym: book
    examples: |
      - reserve
      - booking
  -lookup: getClass
    examples: |
      - business
      - economy
      - first
```

- copy this file to `/trainingData/intents` folder
- Ensure you use same intent (task name) and entities as defined in your DDF.

In case you add any new utterance to JSON file, you need to retrain the NLP Engine by restarting the DialogEngine. (it will train the model every time you start it)

More information on training format is available at - <https://rasa.com/docs/rasa/training-data-format/>

DialogEngine - intent classification

NLP Engine provides intent classification as a JSON structure with some confidence score.

In order to provide proper intent classification based on the output of NLP Engine, DialogEngine uses following two parameters. IE_THRESHOLD_SCORE and IE_SIMILARITY_INDEX configured in *bot.properties*

These parameters should be fine tuned such that DialogEngine is able to classify the unique intent properly. In case the intents are too similar it will prompt user for clarification.

IE_THRESHOLD_SCORE=0.45
IE_SIMILARITY_INDEX=0.5
LOG_DIALOG=true
SHOW_INTERACTIVE_CARDS=true
ALLOWED_FAILURE_ATTEMPTS=3

This is used by Intent Engine – If it finds more than one intent with score exceeding threshold level, this parameter will be used to qualify the second intent.(if score difference is < similarity index then this will be set to INTENT_CLARIFICATION where user will be prompted to confirm

This is used by Intent Engine – this is threshold confidence score , if exceeded intent will be considered as INTENT_FOUND

Important –
There is provision to call the task directly using #<taskname>

e.g., if userUtterance is #getTripInformation and if it is present in DDF , the BOT engine will directly call this task bypassing the NLPEngine.

```
{  
  "intent": {  
    "name": "getTripInformation",  
    "confidence": "1.00"  
  },  
  "entities": [],  
  "intent_ranking": [{  
    "name": "getTripInformation",  
    "confidence": "1.00",  
    "utterance": "Need a reserv  
  }, {  
    "name": "getFAQ",  
    "confidence": "0.20",  
    "utterance": "How do I know  
  }]
```

Training, activity and Dialog Logs

1- The failed conversation as well as tasks that are flagged by user as “liked” and “disliked” are logged into a log file `<mmm>_<dd>_failures.log` , `<mmm>_<dd>_likes.log` and `<mmm>_<dd>_dislikes.log` respectively. (mmm is current month and dd is current date) located in folder `/res/logs/training/<MMM>`

```
Fri Jun 08 14:26:52 IST 2018 domain:trip_en user: sessionId: clientIP: task:getTripInformation utterance:S: for how many persons to Pune ? U: ?
```

This log will help in training the BOT - for intents that seems to be correct but not defined in DDF file , user not satisfied by bot answer etc.

2- The logs are captured whenever action is executed in a file `<mmm>_<dd>_activity.log` (mmm is current month and dd is current date) located in folder `/res/logs/activity/<MMM>`

```
Sat May 13 23:37:32 IST 2017 domain:trip user:John task:getFAQ TYPE:GroovyAction@4f5de9f8
Sat May 13 23:38:09 IST 2017 domain:trip user:John task:getTripInformation TYPE:GroovyAction@647e46e1
Sat May 13 23:39:42 IST 2017 domain:trip user:John task:getWeatherInformation TYPE:HTTPAction@2bf4b0ef
```

3- The dialog conversations are logged if `LOG_DIALOG` flag in `bot.properties` is set to true (`LOG_DIALOG=true`). They are stores in a file `<mmm>_<dd>_dialog.log` (mmm is current month and dd is current date) located in folder `/res/logs/dialog/<MMM>`

```
Sun May 14 16:34:39 IST 2017 domain:trip client_ip:192.168.0.109 user:John id:d1-5SABSB4MOVIF dialog:"S":"How may I help you?","U":"Hi","S":"Hello t
Sun May 14 16:37:52 IST 2017 domain:trip client_ip:192.168.0.109 user:John id:d2-VDQMVNCDSFYB dialog:"S":"How may I help you?","U":"How is weather",
```

Due to GDPR compliance requirement all the important data provided by user is masked (replaced with ??) while storing in LOG file.

Dialog Engine configuration

... 1/2

Dialog Engine configuration file *bot.properties* is located in *DialogEngine/res/config* folder and stores some global properties parameter.

#Bot SSL Params

PROTOCOL = HTTPS

If set to HTTPS the Jetty server will run as secure HTTPS else HTTP (default). For HTTPS the keystore password will be from KSPASSWORD. (it will accordingly enable WebSocket to ws:// or wss://)

#Bot Params

USE_NLG = false

If set true, will fill natural language responses . Tell me now, OK, etc.

USE_GREETINGS = true

If set true, will answer greeting related generic questions . Good Morning etc.

IGNORE_PREV_TASK = false

If TRUE (and allowSwitchTask -> TRUE) the prev task will not be retained in context.

ALLOWED_FAILURE_ATTEMPTS = 5

After these many failed attempts Bot will invoke “handoverTask” to initiate transfer from bot to human.

SHOW_INTERACTIVE_FORM = true

This will create the interactive card response (see appendix 5 for more details)

#FAQ service params

USE_DOMAIN = false

This is for answering domain questions that are build into main conversation itself.

DOMAIN_URL_METHOD = get

URL method for querying domain Qs. (e.g. FAQs that are part of user conversations)

DOMAIN_URL = http://localhost:5000/faq

URL => should respond to Qs directly as text (e.g. ATM is cash withdrawal machine)

GET method -> http://localhost:5010/faq?userUtterance=“What is ATM”

Response will be - {“response” : “ATM is cash with drawl Machine”} or {“response” : “NA”}

#Check QA as ITO

USE_QACHECK = false

This is used in conjunction with “sys.corpus.qa” entity. If true, it will call QA_URL ,which should return True or False based on if it has answer to the question. [Method => POST, url => QA_URL]

QA_URL = http://mydomainqa.com/v2

Parameters => userUtterance=“what is ATM”? Response => {“response”：“true”} or {“response”：“false”}]

If answer is “YES” this entity will be filled and you can use <Action> to call the appropriate URL

#Intent classifier params

IE_THRESHOLD_SCORE = 0.42

Used by Intent Engine – if confidence score is > threshold, intent will be considered as INTENT_FOUND

IE_SIMILARITY_INDEX = 0.3

If more than one intent is found with score > threshold level, this parameter will be used to qualify the second intent. If score difference is < similarity index, then user will be prompted for INTENT_CLARIFICATION

Dialog Engine configuration

... 2/2

#Log params

```
MASK_DATA = false → If true, data will be masked during logging the information  
LOG_DIALOG = true → If true , it will log dialog logs to /res/log/dialog folder
```

#CORS Params

```
ENABLE_CORS = true → If true it will send following parameters in its headers. (Need to check security policies)  
ACCESS_CONTROL_ALLOW_ORIGIN = *  
ACCESS_CONTROL_ALLOW_METHODS = GET,PUT,POST,DELETE,OPTIONS  
ACCESS_CONTROL_ALLOW_HEADERS = X-Requested-With,Authorization,Content-Type,Accept,Origin  
ACCESS_CONTROL_EXPOSE_HEADERS = Location
```

#kafka broker Param

```
NLP_ENGINE_URL = https://localhost:5050 → NLP Engine URL and port if it is used with REST based option  
NLP_KS_PASSWORD = naturalDialog → Keystore password for SSL based communication  
USE_BROKER = true → If true it will use Broker service for communication between CORE else uses REST based APIs  
MAX_WAIT_TIME = 100 → This is maximum time in Sec, Dialog engine will wait for NLP Engine to respond via Broker.
```

NLP Engine configuration

NLP Engine configuration file *nlp.properties* is located in *NLP Engine/config* folder of NLP Engine and stores NLP properties parameter.

```
[ENGINE_PARAMETERS]
USE_BROKER = true → If true, Broker Engine will be used for communication between DE and NE
HTTPS = true → If true , NE will use Secures HTTPS protocol for communication between DE and NE
PORT = 5050 → Port on which NE will listen for requests

[BROKER_SERVICE]
#enter csv for multiple server e.g. 192.67.1.45:9092,192.6
KAFKA_BROKERS = 192.168.0.101:9092 → Kafka Broker URL. It can be multiple URLs for production set up.
PARTITIONS = 3 → Number of partitions should be same as that of Broker properties
MAX_POLL_RECORDS = 1
OFFSET_RESET = latest
GROUP_ID = nlpEngine_1
CLIENT_ID = nlp_1
TOPIC_BOT_TO_NLP = bot_to_nlp → This topic is responsible for sending messages from Dialog Engine to NLP Engine
TOPIC_NLP_TO_BOT = nlp_to_bot → This topic is responsible for sending messages from NLP Engine to Dialog Engine

#Data format of intnet definition - json,md
[DATAFORMAT]
FORMAT = md → format in which the intents have been defined in /traininData/intnets/<domain>_<locale>.<format>

#Algorithms TFIDF,NLU
[NLP_ALGORITHM]
ALGORITHM = TFIDF → Type of algorithm –
TFIDF → uses [TFIDF] parameters to classify intents using TFIDF/Cosine Similarity
NLU → uses rasa based [NLU] configuration to run NLP algorithm

#decide your config file based on SPACY, BERT, DIET
[RASA]
CONFIG_FILE = config_diet.yaml → Defines rasa based NLU pipeline – tokenizer, featuriser, classifier, entity extractor

#Desired dimensionality and #Number of iterations
[TFIDF]
VECTOR_DIMENSION = 200 → Word space dimension for TFIDF-Cosine similarity algorithm
ITERATION_NUMBER = 15 → No of iterations
```

Broker Engine configuration

Broker Engine configuration file *broker.properties* is located in *DialogEngine/res/config* folder and stores broker parameter.

```
#Zookeeper Params
ZOOKEEPER_HOST = localhost:2181 → Zookeeper URL and port

#Kafka Server Params
# if multiple brokers use comma seperation e.g. localhost:9092
KAFKA_BROKERS = localhost:9092 → Kafka Broker server URLs. For production, it can be multiple servers
MAX_POLL_RECORDS = 1 → Maximum records to be polled – 1 as it is 1 to 1 communication between DE and NE
#either latest or earliest
OFFSET_RESET = latest → Only latest message to be received when consumer starts

#Topic and Partition Params
GROUP_ID = dialogEngine_1 → Group ID
CLIENT_ID = dialog_1 → Client ID

PARTITIONS = 3
REPLICATION = 1
TOPIC_BOT_TO_NLP = bot_to_nlp → This topic is responsible for sending messages from Dialog Engine to NLP Engine
TOPIC_NLP_TO_BOT = nlp_to_bot → This topic is responsible for sending messages from NLP Engine to Dialog Engine
```

BOT roles configuration

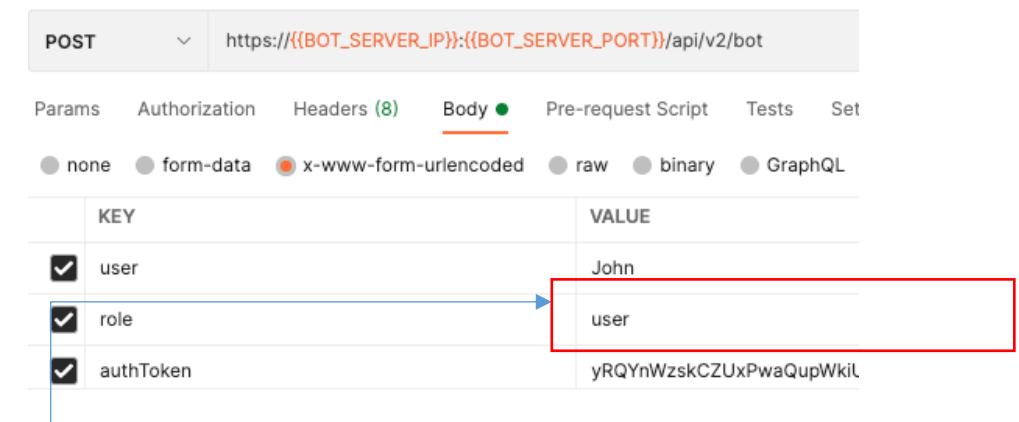
Bot configuration file *roles.properties* is located in *DialogEngine/res/config* folder and stores different roles and their access levels.

```
#Update your roles and its hierarchy, 0 being highest hierarchy
LEVEL_0=SUPERUSER
LEVEL_1=ADMIN
LEVEL_2=MANAGER,HEAD
LEVEL_3=USER,CUSTOMER
LEVEL_4=
LEVEL_5=
```

Important –

- User role is defined at the time of creating bot instance (see slide 6). If no role is provided it is treated as “admin” role
- One can define the specific role to task (both xml and yml) so that it can be accessed to only that user and anyone having higher level of access.

Different roles and their hierarchies can be defined in this file. Level 0 is treated as highest priority. So user with “superuser” role can access all the tasks.



Since John has role of “user” (level->3) he can not access the “getTripInformation” task which is meant for “admin” (level ->2), bot will respond with message -> Sorry, I am not allowed to process this,

Triggering Event Dialogs

...(1/4)

Platform provides mechanism to trigger dialog events that are outside main DDF. The event specific dialogs are not part of main DDF and are written separately in “/res/events” folder. For ease of use , it is written in yaml format with file extension .yml.

- All DDF features are available as listed below

- use of custom entity
- support of groovy and http action
- resultMapping feature to trigger task based on action (http or groovy) result
- followup task

- The events are triggered by sending “userUtterance” parameter as “@<eventName>” in POST request

For example “@loanProduct” will look for event dialog file “loanProduct.yml” file in /res/events folder

The screenshot shows the Postman interface with a POST request configuration. The URL is set to `https://{{BOT_SERVER_IP}}:{{BOT_SERVER_PORT}}/api/v2/bot/{{BOT_SESSION_ID}}`. The 'Body' tab is selected, showing the following form-data:

KEY	VALUE
<input checked="" type="checkbox"/> userUtterance	@loanProduct
Key	Value

- The event dialog contains set of tasks that the BOT engine will execute as if they are part of main dialog. The tasks of event dialog once executed will be removed automatically as they are not part of main dialog defined in DDF.
- All global parameters that are set in main DDF dialog files will automatically apply for event triggered dialogues.
- The other advantage of using yaml , there is no need to restart Dialog engine whenever you add new event triggered dialogue in event folder.

Triggering Event Dialogs

...(2/4)

Following are few samples of event triggered dialog file -

```
# To avoid Compilation Error Exception – do not add comment  
lines, use only double quotes, add line feed after every if-then  
or while-loop's {} statement closure
```

Single Task with groovy Action – *testSample1.yml*

```
---
```

```
tasks :  
  - task :  
    name : EVT_getFeedback  
    label : Feedback  
    entities :  
      - entity :  
        name : getConfirmation  
        label : Confirmation  
        required : true  
        answerType: sys.decision  
        fallbackQuestion: "if BOT was able to answer to your query?"  
      - entity :  
        name : getRating  
        label : Rating  
        required : true  
        answerType: sys.number  
        fallbackQuestion: "how would you rate this on scale of 0-5?"  
  action :  
    type : groovyAction  
    returnAnswer : true  
    utteranceTemplate : "#msg Please visit again."  
    code : 'executionResults.put("msg","Thank you for your feedback.")'
```

Please note that all the task name must start with "EVT_"

Use variable other than "result"

Single Task with groovy Action with followup – *testSample2.yml*

```
---
```

```
tasks :  
  - task :  
    name : EVT_getFeedback  
    label : Feedback  
    entities :  
      - entity :  
        name : getConfirmation  
        label : Confirmation  
        required : true  
        useContext : true  
        answerType: sys.decision  
        fallbackQuestion: "if BOT was able to answer to your query?"  
      - entity :  
        name : getRating  
        label : Rating  
        required : true  
        answerType: sys.number  
        fallbackQuestion: "how would you rate this on scale of 0-5?"  
  action :  
    type : groovyAction  
    returnAnswer : true  
    utteranceTemplate : "#msg"  
    code : 'executionResults.put("msg","Thank you for your feedback.")'  
  followup :  
    entity :  
      name : anotherOne  
      label : 'More feedback'  
      required : true  
      answerType : sys.decision  
      fallbackQuestion: "do you want to give another feedback?"  
  answerMapping:  
    - map :  
      # use quotes for key  
      key : 'YES'  
      value : EVT_getFeedback  
    - map :  
      # use quotes for key  
      key : 'NO'  
      value : start
```

...

Triggering Event Dialogs

...(3/4)

Following are few samples of event triggered dialog file -

Single Task with resultMapping Action – *testSample3.yml*

```
---
```

```
tasks :  
  - task :  
    name : EVT_getFeedback  
    label : Feedback  
    entities :  
      - entity :  
        name : getConfirmation  
        label : Confirmation  
        required : true  
        answerType: sys.decision  
        fallbackQuestion: "if BOT was able to answer to your query?"  
      - entity :  
        name : getRating  
        label : Rating  
        required : true  
        answerType: sys.number  
        fallbackQuestion: "how would you rate this on scale of 0-5?"  
    action :  
      type : groovyAction  
      resultMappings :  
        - map :  
          message : "Thank you for your feedback. Please visit again."  
          redirectToTask : start  
          resultVarName : action  
          # use quotes for key  
          resultValue : '1'  
      returnAnswer : true  
      utteranceTemplate : null  
      code : 'executionResults.put("action","1")'  
...  
...
```

Single Task with httpAction – *testSample4.yml*

```
---
```

```
tasks :  
  - task :  
    name : EVT_getTemprature  
    label : Temprature  
    entities :  
      - entity :  
        name : getCityName  
        label : 'City Name'  
        required : true  
        answerType: sys.location.city  
        fallbackQuestion: "for which city do you want to know the weather?"  
    action :  
      type : httpAction  
      returnAnswer : true  
      utteranceTemplate : "The temperature in %getCityName is #result degrees."  
      method : GET  
      params : q=%getCityName&mode=xml&units=metric&APPID=706120eb0f11f2ebf8584dadb9bad3d6  
      url : http://api.openweathermap.org/data/2.5/weather  
      xpath : /current/temperature/@value  
      jpath : null
```

Triggering Event Dialogs

...(4/4)

Following are few samples of event triggered dialog file -

Multiple Task with resultMapping Action – *testSample5.yml*

```
---
```

```
tasks :  
  - task :  
      name : EVT_getBooking  
      label : 'Ticket Booking'  
      entities :  
        - entity :  
            name : getDepartureCity  
            label : 'Departure city'  
            required : true  
            answerType: sys.location.city  
            fallbackQuestion: "where do you want to go?"  
        - entity :  
            name : getDate  
            label : 'Departure Date'  
            required : true  
            answerType: sys.temporal.date  
            fallbackQuestion: "when do you want to leave?"  
        - entity :  
            name : getInfo  
            label : 'Addition Info'  
            required : true  
            answerType: sys.opentext  
            fallbackQuestion: "any additional information that you may have?"  
      action :  
        type : groovyAction  
        resultMappings :  
          - map :  
              message : "Your ticket for %getDepartureCity city is booked. The additional info is %getInfo"  
              redirectToTask : EVT_getTemprature  
              resultVarName : action  
              # use quotes for key  
              resultValue : '1'  
        returnAnswer : true  
        utteranceTemplate : null  
        code : 'executionResults.put("action","1")'  
  - task :  
      name : EVT_getTemprature  
      label : 'City Temprature'  
      entities :  
        - entity :  
            name : getCityName  
            label : 'City Name'  
            required : true  
            answerType: sys.location.city  
            fallbackQuestion: "For which city do you want to know the weather?"  
      action :  
        type : httpAction  
        returnAnswer : true  
        utteranceTemplate : "The temperature in %getCityName is #result degrees."  
        method : GET  
        params : q=%getCityName&mode=xml&units=metric&APPID=706120eb0f11f2ebf8584dadb9bad3d6  
        url : http://api.openweathermap.org/data/2.5/weather  
        xpath : /current/temperature/@value  
        jpath : null
```

To avoid Compilation Error Exception – do not add comment lines, use only double quotes, add line feed after every if-then or while-loop's {} statement closure

API for messenger integration

Bot can be used to integrate with Messaging platform like Facebook or slack by calling appropriate API's listed below.

1. To create the BOT instance for specific user ID (e.g. if a unique user is 1000345667)

Method => GET

URL => [http://192.168.3.19:8080/api/v2/msgbot?user= 1000345667](http://192.168.3.19:8080/api/v2/msgbot?user=1000345667)

Return =>

if successful creates the BOT instance

if unsuccessful returns

- if no userID provided returns {"response":"Error: need user ID for BOT creation"}

- initialization failed if instance is already running

2. To send the user message to instance that has been created in STEP-1

Method => GET

URL => [http://192.168.3.19:8080/api/v2/msgbot/1000345667?userMessage= "Hi"](http://192.168.3.19:8080/api/v2/msgbot/1000345667?userMessage=Hi)

-> use URLEncoder.encode(userUtterance, "UTF-8") for sending the user Utterance

Return => JSON response with response in result.reply JSONObject

3. To check if instance is already available

Method => GET

URL => <http://192.168.3.19:8080/api/v2/msgbot/hasInstance?userID=1000345667>

Returns=> {"response":"false"} or {"response":"true"} based on whether such instance is available.

Appendix -1 Managing interactive response through HTTP action

One can build the interactive chat by building the API that can return JSON response that has message object as shown below.

e.g., To show a gluco meter image with video , if you build an API that can return a JSON response like this -

Note – Use single quotes to specify the resources, if any. (the use of double quote will make the JSON an invalid JSON

```
"message": {  
    "data": {  
        "info": {  
            "image": "https://13.126.162.96:8002/img/glucometer.jpg",  
            "id": "GLUCO_ICARD_2",  
            "text": "Best deals on Bayer CONTOUR glucose meter. 20% OFF at all stores in your city",  
            "video": "https://www.youtube.com/embed/FRthv8Jv3IY"  
        }  
    },  
    "chat": "

Please find more details below on supplier near your location.</p><a href='http://'  
    "in/':>Click here for further details.</a>",  
    "type": "ICardTextImageVideo"  
}


```

Here is the BOT response that will have appended JSON object at the bottom as received by API

```
{  
    "distiles": "...",  
    "type": "entityList",  
    "entities": [  
        {  
            "id": "1",  
            "name": "getDIYOption",  
            "label": "DIY option",  
            "entityType": "sys.decision",  
            "type": "radio",  
            "elements": "yes,no",  
            "value": "",  
            "isVisible": "true",  
            "isActive": "true"  
        }  
    ],  
    "message": {  
        "data": {  
            "info": {  
                "image": "https://13.126.162.96:8002/img/glucometer.jpg",  
                "id": "GLUCO_ICARD_2",  
                "text": "Best deals on Bayer CONTOUR glucose meter. 20% OFF at all stores in your city",  
                "video": "https://www.youtube.com/embed/FRthv8Jv3IY"  
            }  
        },  
        "chat": "

Please find more details below on supplier near your location.</p><a href='http://'  
        "in/':>Click here for further details.</a>",  
        "type": "ICardTextImageVideo"  
    }  
}


```

```
<action>  
    <httpAction>  
        <returnAnswer>true</returnAnswer>  
        <utteranceTemplate>#result</utteranceTemplate>  
        <method>get</method>  
        <params>id=%getCardType</params>  
        <url>http://10.44.22.76:8090/lima/serlayer/getDuplicateInsuranceCard</url>  
        <xpath></xpath>  
        <jpath>$..message</jpath>  
    </httpAction>  
</action>  
</task>
```

Here is task to read the same

- specify jpath as JSON object \$..message
- **Use only #result** in utteranceTemplate – this will put chat object here if present. (this is required irrespective of you have chat object or not)

Note this is a JSON object and not key

Appendix -1 Managing interactive response through Groovy action ... (1/3)

One can build similar experience using Groovy action (refer earlier slide for understanding the interactive action using HTTP)

The BOT JSON response will append the JSON body that you can use on client for creating interactive card.

Build API that has below response. (Needs to have chat object for returning the BOT reply)

```
"message": {  
    "data": {  
        "error": "",  
        "info": {  
            "image": "<div><img src='img/ticket.jpg' alt='ticket.jpeg'> </div>",  
            "video": "<div class='video-container'><iframe src='https://www.youtube.com/embed/1234567890123456789012345678901234567890'></div>",  
            "audio": "",  
            "text": "Your ticket is booked. Please see the details in attached file."  
        },  
        "chat": "<p>Your ticket is booked successfully for 600 Dollars. Wish you a safe journey!</p>  
        <a href='https://www.irctc.co.in/eticketing/loginHome.jsf'>Click here for login</a>"  
    }  
},  
"chat": "<p>Your ticket is booked successfully for 600 Dollars. Wish you a safe journey!</p>  
        <a href='https://www.irctc.co.in/eticketing/loginHome.jsf'>Click here for login</a>"
```

Here is the BOT response that will have appended JSON object at the bottom

```
        "lu": "+",  
        "name": "cancelTask",  
        "label": "Cancel"  
    },  
],  
}  
,  
"message": {  
    "data": {  
        "error": "",  
        "info": {  
            "image": "<div><img src='img/ticket.jpg' alt='ticket.jpeg'> </div>",  
            "video": "<div class='video-container'><iframe src='https://www.youtube.com/embed/1234567890123456789012345678901234567890'></div>",  
            "audio": "",  
            "text": "Your ticket is booked. Please see the details in attached file."  
        },  
        "chat": "<p>Your ticket is booked successfully for 600 Dollars. Wish you a safe journey!</p>  
        <a href='https://www.irctc.co.in/eticketing/loginHome.jsf'>Click here for login</a>"  
    }  
},  
"chat": "<p>Your ticket is booked successfully for 600 Dollars. Wish you a safe journey!</p>  
        <a href='https://www.irctc.co.in/eticketing/loginHome.jsf'>Click here for login</a>"
```

```
<action>  
    <groovyAction>  
        <resultMappings>  
            <resultMapping>  
                <message></message>  
                <redirectToTask>start</redirectToTask>  
                <resultCode>1</resultCode>  
                <resultVarName>action</resultVarName>  
            </resultMapping>  
        </resultMappings>  
        <returnAnswer>true</returnAnswer>  
        <utteranceTemplate>#result</utteranceTemplate>  
        <code>  
            <![CDATA[  
                String type=new String(frame.getAttribute("getTicketType"));  
                Integer action = new Integer(1);  
                import org.json.JSONException;  
                import org.json.JSONObject;  
                def url = new URL('http://www.mocky.io/v2/58dfb92c1000007d03cc15ca');  
                def connection = url.openConnection();  
                connection.requestMethod = 'GET';  
                if(connection.responseCode == 200) {  
                    body = connection.content.text;  
                    executionResults.put("body",body);  
                }  
                executionResults.put("action",action.toString());  
            ]]>  
        </code>  
    </groovyAction>  
</action>
```

- Populate the JSON response in "body" variable
- **use only #result** in utteranceTemplate this will put chat object here if present.
- You can as well populate a variable for redirection , if required. (action = 1 in this case)

Appendix -2 -Managing interactive response through Groovy

...(2/3)

One can manage the API integration completely through Groovy.

```
<resultMapping>
  <message></message>
  <redirectToTask>getWeatherInformation</redirectToTask>
  <resultValue>1</resultValue>
  <resultVarName>action</resultVarName>
</resultMapping>
</resultMappings>
<returnAnswer>true</returnAnswer>
<utteranceTemplate>#chat</utteranceTemplate>
<code>
<![CDATA[
import org.json.JSONException;
import org.json.JSONObject;
import java.util.logging.Logger;
import java.net.URLEncoder;
Logger logger = Logger.getLogger("");
String body = new String ("");
String chat = new String ("");
def url =new URL('http://10.44.22.76:8090/lima/serlayer/getDuplicateInsuranceCard');
def connection = url.openConnection();
if (connection.responseCode == 200) {
body = connection.content.text;
JSONObject json = new JSONObject(body);
if (json.getJSONObject("message").has("chat"))
chat = json.getJSONObject("message").getString("chat");
logger.info ("chat output ====="+chat);
}
executionResults.put("chat",chat);
executionResults.put("body",body);
//for result mapping
executionResults.put("action","1");
]]>
</code>
</groovyAction>
<action>
```

1. Please populate body as a total JSON response. It will look for message (see JSON response on Appendix-1 slide) and if present will process it.
2. Parse which ever object you want in your utterance (#chat in this case)

Option-1 – if you use #chat then it will only give utterance and will not append it in BOT processed response. But you can customize your response.

Option-2 – If you use ONLY #result then it will be automatically processed for chat object for response and additionally append JSON response at the end of BOT processed response.

You can manage #action in case you need to redirect the task based on the result in conjunction with resultMapping

Appendix -2 -Managing interactive response through Groovy

...(3/3)

Instead of relying on backend API , one can create the custom JSON response in Groovy itself

Option -1

```
<action>
  <groovyAction>
    <returnAnswer>true</returnAnswer>
    <utteranceTemplate>#result</utteranceTemplate>
    <code><![CDATA[
      import org.json.JSONException;
      import org.json.JSONObject;
      String body = new String ("");
      String account = new String(frame.get("getAccountNumber"));
      String chat = "";
      chat = "<p>Here is list of transactions for account ending with "+ac
      JSONObject resultJson = new JSONObject();
      JSONObject messageJson = new JSONObject();
      JSONObject dataJson = new JSONObject();
      JSONObject infoJson = new JSONObject();
      infoJson.put("image", "<img src='img/holiday.jpg'>");
      infoJson.put("video", "<div class='video-container'><iframe width='100%' height='100%' src='http://127.0.0.1/img/store_pune.png'></iframe></div>");
      infoJson.put("text", "Searching for week end deals. Save upto \$200");
      dataJson.put("info", infoJson);
      messageJson.put("chat", chat);
      messageJson.put("type", "iCardTextImageVideo");
      messageJson.put("data", dataJson);
      resultJson.put("message", messageJson);
      executionResults.put("body", resultJson.toString());
    ]]></code>
  </groovyAction>
</action>
</task>
```

Option -2 – In this option you can create iCard of type **iCardTextImageVideo** by configuring your data in csv file. Load your csv file in following format and upload at **/res/idata** (e.g. **/res/idata/gmeter_iCard.csv**)

#ID,Chat,Text,Image,Video
GLUCO_ICARD_1, "See below ongoing Deals","Medtronic 20 % ongoing sale. Hurry Up!!!",
http://127.0.0.1/img/store_pune.png, <https://www.youtube.com/embed/FRthv8Jv3IY>

Use following groovy action to load this data and create JSON response

```
<action>
  <groovyAction>
    <returnAnswer>true</returnAnswer>
    <utteranceTemplate>#result</utteranceTemplate>
    <code>
    <![CDATA[
      import cto.hmi.idatautil.ProcessCardData;

      String itemFile = "gmeter_iCard.csv";
      String id = "GLUCO_ICARD_1";
      String body = ProcessCardData.GetJSONString(itemFile,id);
      executionResults.put("body",body);
    ]]>
  </code>
</groovyAction>
```

OR

You can pass the information directly to ProcessCardData class function -
ProcessCardData.GetJSONString
(chat,text,imgURL,videoURL);

```
<utteranceTemplate>#result</utteranceTemplate>
<code>
<![CDATA[
  import cto.hmi.idatautil.ProcessCardData;
  String chat = "See below for detailed instruction on how to install Invit
  String text = "1.Turn off the power supply before plugging adapters into
  String imgURL = "content/adapterInfo.png";
  String videoURL = "https://www.youtube.com/embed/zlJy-t8qfwc";
  String body = ProcessCardData.GetJSONString(chat,text,imgURL,videoURL);
  executionResults.put("body",body);
]]>
</code>
```

Appendix -3 – Managing both Transactional and FAQ conv. using intent engine

In order to manage both Transactional and FAQ conversation in one dialogue one can follow below approach

- Use Intent Engine to handle all your transactional scenarios. (No need to add FAQ specific utterances)
- Create “getFAQ” task to handle all FAQ specific questions
..(Important – task name has to be “getFAQ” and “overanswering” flag to be set to true)
- Set USE_DOMAIN parameter in bot.properties to “false” as we do not want 2 sources for finding FAQ answers
- In DDF add getFAQ task as shown here (use groovy for better control on chat response) note – use answerType as “dummy”

```
#Sat Oct 29 15:04:24 IST 2016
USE_NLG=true
USE_GREETINGS=true
USE_DOMAIN=false
DOMAIN_URL_METHOD=get
DOMAIN_URL=http://localhost:5000/faq
```

```
<task name="getFAQ" label="Find FAQ">
  <entities>
    <entity name="getQuestion">
      <answerType>dummy</answerType>
      <fallbackQuestion>OK</fallbackQuestion>
      <required>true</required>
    </entity>
  </entities>
  <action>
    <groovyAction>
      <returnAnswer>true</returnAnswer>
      <utteranceTemplate>#chat</utteranceTemplate>
      <code>
        <![CDATA[
import org.json.JSONException;
import org.json.JSONObject;
String question=new String(frame.get("getQuestion"));
String body = new String ("");
String chat = new String ("");
String api = "http://www.mocky.io/v2/590ae3b82900004b0523d934";
String urlParam = "?userUtterance="+ URLDecoder.encode(question, "UTF-8");
def url = new URL(api+urlParam);
def connection = url.openConnection();
connection.requestMethod = 'GET';
if (connection.responseCode == 200) {
body = connection.content.text;
JSONObject json = new JSONObject(body);
if (json.has("response"))
chat = json.getString("response");
if (chat.equals("NA"))
chat="Sorry, I did not understand that.";
}
executionResults.put("body",body);
executionResults.put("chat",chat);
]]>
      </code>
    </groovyAction>
  </action>
</task>
```

Appendix -4 – Building Interactive Form using UI Widgets

...1/4

The JSON provides 3 types of iForm object (if enabled in bot.properties file).

1- The iForm object with type “taskList” for user to select the task with a click of button. (use #getTripInformation in userUtterance urlencoded body parameter to call getTripInformation task if user clicks on it)

```
"iForm": {  
  "type": "taskList",  
  "feedback": {  
    "isVisible": "false",  
    "likes": "0",  
    "dislikes": "0"  
  },  
  "tasks": [  
    {  
      "id": "1",  
      "name": "getTripInformation",  
      "label": "Book ticket",  
      "role": "user"  
    },  
    {  
      "id": "2",  
      "name": "getWeatherInformation",  
      "label": "Weather information",  
      "role": ""  
    },  
    {  
      "id": "3",  
      "name": "getWikipediaCityInfo",  
      "label": "City information",  
      "role": ""  
    },  
    {  
      "id": "4",  
      "name": "getHotelInformation",  
      "label": "Hotel information",  
      "role": ""  
    }  
  ]  
}
```

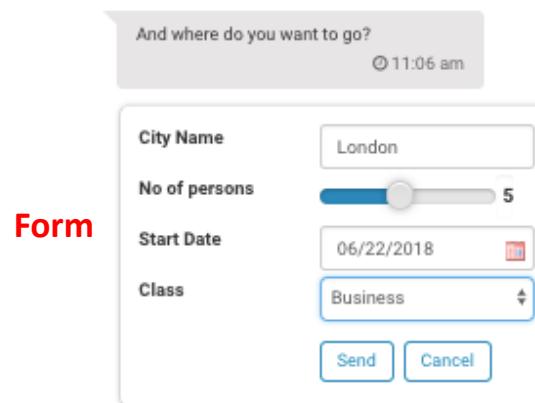
This is count of likes and dislikes given by the user

Type of UI
e.g. text, number, button, list, multiSelectionList, radio, date, time , slider etc.

For UI type list, multitemplist, slider and radio this will show the elements.

Value of entity, if filled

Entity currently active, if true



Form

2- The iForm object with type “entityList” for user to enter various entity data as part of form element and submit the same to Bot

```
"iForm": {  
  "feedback": {  
    "isVisible": "false",  
    "likes": "0",  
    "dislikes": "0"  
  },  
  "type": "entityList",  
  "entities": [  
    {  
      "id": "1",  
      "name": "getDestinationCity",  
      "label": "City Name",  
      "entityType": "sys.location.city",  
      "type": "text",  
      "elements": "",  
      "value": "",  
      "isVisible": "true",  
      "isActive": "true"  
    },  
    {  
      "id": "2",  
      "name": "getNumberOfPersons",  
      "label": "No of persons",  
      "entityType": "customSlider",  
      "type": "slider",  
      "elements": "1,10,1,5",  
      "value": "",  
      "isVisible": "true",  
      "isActive": "false"  
    },  
    {  
      "id": "3",  
      "name": "getStartDate",  
      "label": "Start Date",  
      "entityType": "sys.temporal.date",  
      "type": "date",  
      "elements": "",  
      "value": "",  
      "isVisible": "true",  
      "isActive": "false"  
    }  
  ]  
}
```

Show theUI element only if this flag is set to “true”.
This is done to handle entities like “opentext” as they take free text and needs to be shown as a single separate item in Form

Appendix -4 – Building Interactive Form using UI Widgets

...2/4

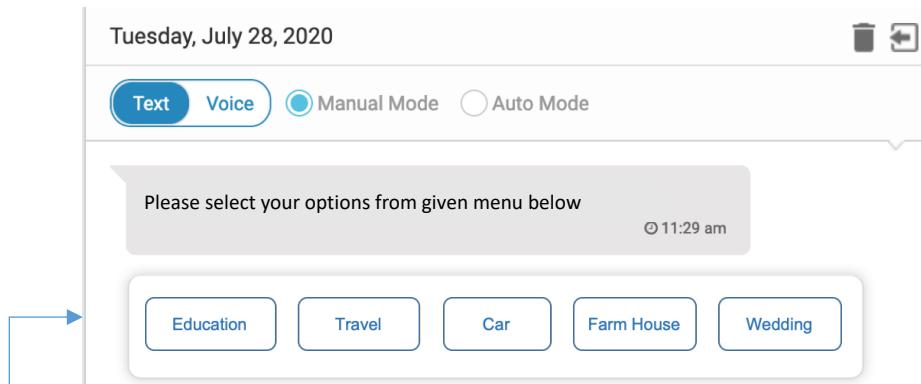
3- The iForm object with type “menuList” for user to select the clickable menu items. This will give user better options to maneuver to different tasks through proper hierarchy. (This will avoid cluttering of all the task list in taskList options - 1)

```
        "feedback": {  
            "isVisible": "false",  
            "likes": "0",  
            "dislikes": "0"  
        },  
        "type": "menuList",  
        "entities": [  
            {  
                "id": "1",  
                "name": "menuItems",  
                "label": "",  
                "entityType": "customItems",  
                "type": "list",  
                "elements": "Education,Travel,Car,Farm House,Wedding",  
                "value": "",  
                "isVisible": "true",  
                "isActive": "true"  
            }  
        ]  
    }  
}
```

Education

Entity => custom.menu_loans
File => menu_loans.txt at
/res/entities folder

##DO NOT REMOVE THIS LINE – to check type
Education
Travel
Car
Farm House
Wedding



Important – Use entity type custom.menu in your task to populate the list of subMenu options. Do not use this Entity type as part of FULL FORM element , you can use custom.button instead.

Appendix -4 – Interactive chat by embedding HTML tags

...3/4

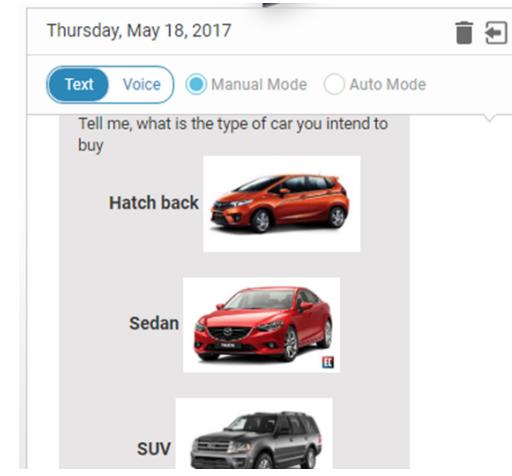
One can use [CDATA] wrapper in utterance tags like <fallback_question>, <utteranceTemplate> or <message> to build more structured response including images.

Of course, this needs to be inferred and handled appropriately on client side – Web, Mobile , Unity UI etc.

```
<task name="bookCar" label="Book Car">
  <entities>
    <entity name="getCarType" label ="Car Type">
      <answerType>custom_item_!</answerType>
      <fallbackQuestion><![CDATA[<p>what is the type of car you intend to buy?</p>
<br><center><b>Hatch back</b><img src='img/hatchback.jpg' height='50%' width='%50'></center><br>
<br><center><b>Sedan</b><img src='img/sedan.jpg' height='50%' width='%50'></center><br>
<br><center><b>SUV</b><img src='img/suv.jpg' height='50%' width='%50'></center><br>]]></fallbackQuestion>
      <required>true</required>
    </entity>
  </entities>
  ..
```

The images can be kept at /res/html/img folder

This is custom parser =>
HatchBack, Sedan, SUV



Important – The key message that BOT needs to speak through (TTS) needs to be embedded in <p> tag.
The message is automatically parsed and sent to “speech” JSONObject in its REST response

Appendix -4 – Processing utterance filled through i-Forms

...4/4

If interactive cards are created for user to enter the data , in order to process entities seamlessly following provision has been made that require special treatment -

City	=> sys.location.city	=> city:<data>;
Person	=> sys.person	=> person:<data>;
First Name	=> sys.person.firstname	=> firstName:<data>;
Last Name	=> sys.person.lastname	=> lastName:<data>;
Organization	=> sys.organization	=> organization:<data>;

This logic needs to be implemented on client side.

e.g. For trip booking the filled in data by user in form for entities (getDestinationCity, getNumberOfPerson, getStartDate, getEndDate) could be passed like

city:Mumbai;<space>4;<space>7/3/17;<space>7/14/17 => this is processed on Bot side in filling entities slots sequentially.

Appendix -5 – Use of context to fill entities automatically

When entity is being defined in DDF there are two flags that can be effectively used to fill them based on the context. The entity will be filled in automatically if it uses the same name as defined earlier in other task and <useContext> flag is set.

There is another xml TAG that can be used to clear the context in case you are sure that the intended use of entity is done and now needs to be cleared from history. For example

```
<task name="getTripInformation" label="Book ticket" role="user">
  <entities>
    <entity name="getDestinationCity" label="City Name">
      <answerType>sys.location.city</answerType>
      <fallbackQuestion>where do you want to go?|what city do you want to travel?</fallbackQuestion>
      <clarifyQuestion>Please provide me with the correct city name?</clarifyQuestion>
      <required>true</required>
    </entity>
```

TASK-1

In this task , we defined entity “getDestinationCity” which takes city name

```
<task name="getWeatherInformation" label="Weather information">
  <entities>
    <entity name="getDestinationCity" label="Weather information">
      <answerType>sys.location.city</answerType>
      <fallbackQuestion>for which city do you want to know the weather?</fallbackQuestion>
      <clarifyQuestion>please provide me with correct city name?</clarifyQuestion>
      <required>true</required>
      <useContext>true</useContext>
      <clearContext>true</clearContext>
    </entity>
  </entities>
```

TASK-2

In this task , we again defined entity “getDestinationCity” but with a two XML tags <useContext> and <clearContext> to true.

With this configuration, if the entity is used in earlier conversation e.g. London, the user will not be asked again. The entity will be filled in with London. Also since the “clearContext” has been set to true , it will also cleared from history as task gets completed.. So next time user asks the Weather Information , Bot will prompt user with fallback question “for which city do you want to know the weather?”

Appendix -6 – Storing and passing the information at Task level

For certain scenarios we need a feature where we need to capture the information in one task (captured in dummy entity) and pass it to another task to fill different entities.

For such scenario there is a provision of **storeCache** , **useCache** and **clearCache** flags –

Here are the steps to follow

- set **<storeCache>** to true for entity of which you want to store the information (e.g. dummy or opentext etc.)
- Use XML attribute **useCache** and set it to true while defining the task which wants to use this info
- You can clear the CACHE by setting **<clearCache>** to true in entity that you think is appropriate.

Here is the sequence of events –

User -> I want weather information => will switch to getWeatherInformation

Bot-> for which city you want to know weather

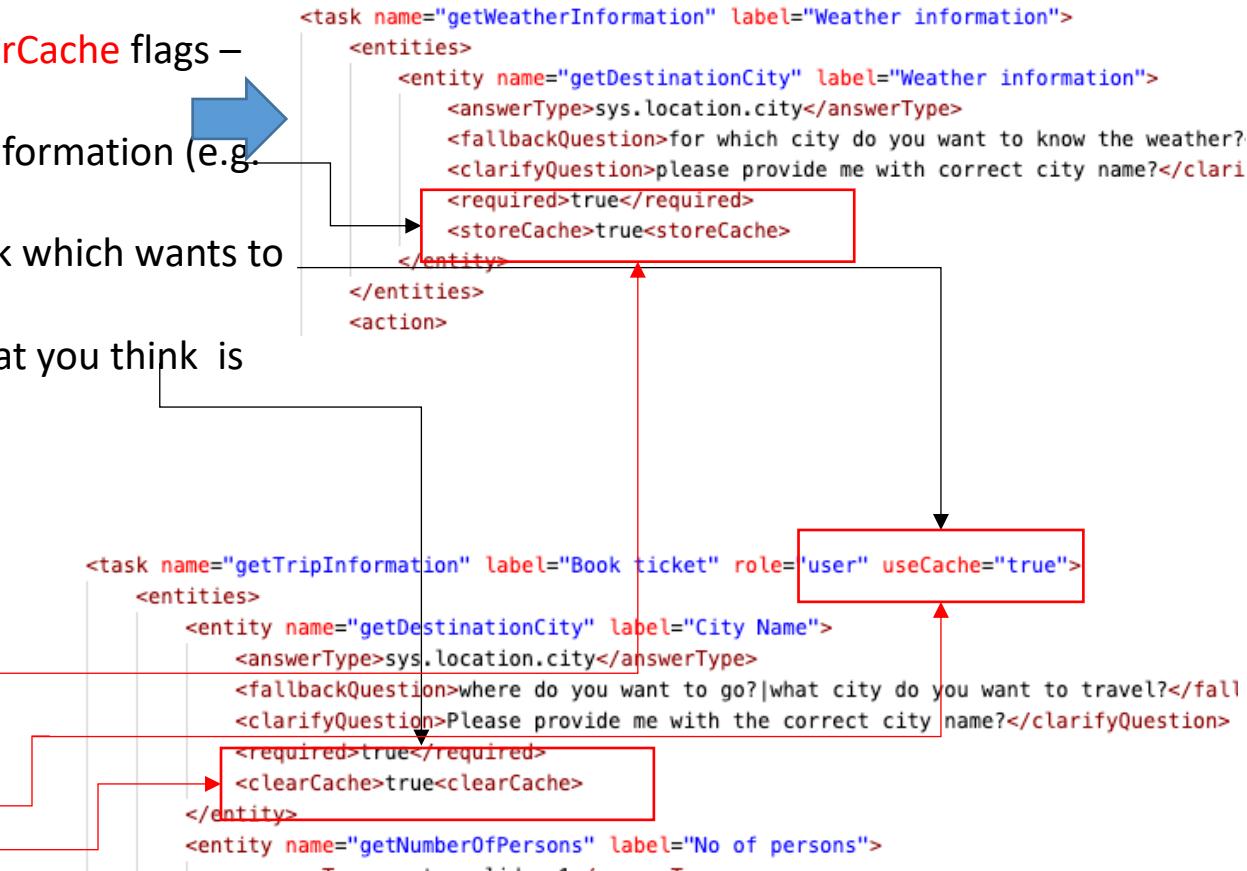
User -> London => (London will be stored as a String in CACHE)

Bot -> Temp is London is 12 degree. How may I help you?

User -> I want to Book a ticket => (here along with this utterance , London will also be passed as useCache attribute is set to TRUE)

Bot -> For How many persons => (The getDestinationCity will automatically filled with London and will also clear it from CACHE)

– note-“allowOverAnswering” flag is set to true



Appendix -7 – License

Three types of license are available

- 1- LIFE_TIME -> does not come with expiry date
- 2- SINGLT_TIME -> One time license with expiry date
- 3-TRIAL -Trial for experimentation with expiry date

- All the license keys and signatures are stored in /res/keys folder (viz. *license.key* and *license.sig*)
- All the license properties are stored in /res/config/license.properties folder (**CAUTION** - Do not tamper with this file)

```
#Please do not update this file
EMAIL=xyz@gmail.com
COMPANY=xyz
LICENSE_TYPE=life_time
EXPIRATION=0000-00-00
VERSION=1.2
```

Appendix -8 – APIs to get DDF and current task stack

...1/7

In order to get the Dialog definition following API is available

Method -> **POST**

URL -> *http://<IP>/api/v2/bot/{instanceId}/getDDF*

e.g. <http://192.168.0.103:8080/api/v2/bot/d1-YDEHE06UPRMM/getDDF>

Here is a sample response -

```
{  
  "allowCorrection": true,  
  "name": "trip",  
  "company": "xyz",  
  "useIntentEngine": true,  
  "allowDifferentQuestion": true,  
  "useSODA": true,  
  "allowSwitchTasks": true,  
  "allowOverAnswering": true,  
  "tasks": [  
    {  
      "name": "start",  
      "label": "",  
      "itos": [  
        {  
          "name": "welcome",  
          "label": "",  
          "type": "open-ended"  
        }  
      ]  
    },  
    {  
      "name": "getTripInformation",  
      "label": "Book ticket",  
      "itos": [  
        {  
          "name": "getDestinationCity",  
          "label": "To city",  
          "type": "sys.location.city",  
          "value": null  
        },  
        {  
          "name": "getNumberOfPersons",  
          "label": "No of persons",  
          "type": "sys.number",  
          "value": null  
        },  
        {  
          "name": "getStartDate",  
          "label": "Start Date",  
          "type": "sys.temporal.date",  
          "value": null  
        },  
        {  
          "name": "getEndDate",  
          "label": "End Date",  
          "type": "sys.temporal.date",  
          "value": null  
        }  
      ]  
    }  
  ]  
}
```

In order to get the active tasks on stack following API is available

Method -> **POST**

URL -> *http://<IP>/api/v2/bot/{instanceId}/tasks*

e.g. <http://192.168.0.103:8080/api/v2/bot/d1-YDEHE06UPRMM/tasks>

Here is sample response -

```
{  
  "tasks": [  
    {  
      "name": "getTripInformation",  
      "label": "Book ticket",  
      "entities": [  
        {  
          "name": "getDestinationCity",  
          "label": "To city",  
          "type": "sys.location.city",  
          "value": null  
        },  
        {  
          "name": "getNumberOfPersons",  
          "label": "No of persons",  
          "type": "sys.number",  
          "value": null  
        },  
        {  
          "name": "getStartDate",  
          "label": "Start Date",  
          "type": "sys.temporal.date",  
          "value": null  
        },  
        {  
          "name": "getEndDate",  
          "label": "End Date",  
          "type": "sys.temporal.date",  
          "value": null  
        }  
      ]  
    }  
  ]  
}
```

Appendix -8 – APIs to get dialogs, clear dialog & terminate instance ...2/7

In order to get past conversation between user and bot following API is available

Method -> **POST**

URL -> <http://<IP>/api/v2/bot/{instanceId}/getDialog>

e.g. <http://192.168.0.103:8080/api/v2/bot/d1-YDEHE06UPRMM/getDialog>

Here is sample response-

```
{
  "user": "John",
  "userAgent": "[Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36]",
  "clientIP": "10.88.250.101",
  "instanceID": "d1-FZP1O9PXOKGG",
  "timeStamp": "2017-05-19 18:42:27",
  "dialog": [
    {
      "S": "How may I help you?"
    }
  ]
}
```

Clear Dialog History

In order to clear past dialog history between user and bot following API is available

Method -> **POST**

URL -> <http://<IP>/api/v2/bot/{instanceId}/clearDialog>

e.g. <http://192.168.0.103:8080/api/v2/bot/d1-YDEHE06UPRMM/clearDialog>

In order to terminate the dialog instance following API is available

Method -> **POST**

URL -> <http://<IP>/api/v2/bot/{instanceId}/kill>

e.g. <http://192.168.0.103:8080/api/v2/bot/d1-YDEHE06UPRMM/kill>

Here is a sample response

```
{
  "response": "INFO:Removed instance d1-3BXY78BBPJ0M"
}
```

Get Dialog Failure Info

In order to get dialog failure count in dialog (bot failed to answer and user likes and dislikes) following API is available

Method -> **POST**

URL -> <http://<IP>/api/v2/bot/{instanceId}/failureInfo>

<http://192.168.0.103:8080/api/v2/bot/d1-NDMMAMPK0XFS/failureInfo>

Here is a sample response

```
{
  "sessionId": "d1-JUHIHLFWCBRM",
  "failures": "1",
  "likes": "2",
  "dislikes": "0"
}
```

Appendix -8 – Bot engine status and custom entities

...3/7

In order to get bot engine's current status following API is available (it gives id and current failures in responding to user)

Method -> **POST**

URL -> *http://<IP>/api/v2/status*

e.g. <http://192.168.0.103:8080/api/v2/status>

Here is sample response-

```
{  
  "status": {  
    "domain": "trip",  
    "currentSessions": "2",  
    "uiType": "RESTInterface",  
    "company": "abc corporation",  
    "URI": "https://192.168.0.102:8080/",  
    "startedOn": "Fri Jun 08 23:31:45 IST 2018",  
    "sessionIDs": [  
      {  
        "failures": "3",  
        "dislikes": "12",  
        "id": "d1-IUXAHITAEUIT",  
        "user": "John",  
        "likes": "8",  
        "lastAcess": "2018-06-08 22:32:25"  
      },  
      {  
        "failures": "3",  
        "dislikes": "4",  
        "id": "d2-4J6VU00SP3XB",  
        "user": "Steve",  
        "likes": "17",  
        "lastAcess": "2018-06-08 23:32:40"  
      }  
    ]  
  }  
}
```

In order to obtain custom entities from engine to make BOT client more interactive by providing the bot user option to select , one can use this API

Method -> **POST**

URL -> *http://<IP>/api/v2/bot/{instanceId}/getEntityData*

e.g. <http://192.168.0.103:8080/api/v2/bot/d1-YDEHE06UPRMM/getEntityData>

Here is sample response-

```
{  
  "name": "trip",  
  "version": 1.1,  
  "items": [  
    {  
      "values": [  
        "Hatch back",  
        "Sedan"  
      ],  
      "name": "item_1"  
    },  
    {  
      "values": [  
        "all",  
        "broker",  
        "claim",  
        "support",  
        "individual"  
      ],  
      "name": "item_5"  
    }  
  ]  
}
```

Appendix -8 – APIs to get autocomplete data ...*under test*

...4/7

In order to provide the user with recommended words while typing on chat window , bot engine provides list of words based on the users current utterance. Following steps are required to be taken for using this feature

- 1- Create the corpus (contains all the possible utterances and FAQ Qs that user may type) in /res/autocomplete/data folder with the file name as *corpus.txt*
- 2- Call below API from BOT client as user types in (ONCHANGE event of JavaScript in text input field)
- 3- Get the recommend list of words from bot engine by calling following API

Method -> **POST**

URL -> *http://<IP>/api/v2/autocomplete*

URL encoded body parameter -> userUtterance="I want "

e.g. <http://10.88.250.233:8080/api/v2/autocomplete>

NOTE: if you have uploaded new corpus to BOT , please delete the Pickle file (.pkl) file in model folder.

Here is sample response-

The screenshot shows the Postman application interface. At the top, there's a header bar with the URL `http://10.88.250.176:8`. Below it, the main window has a 'POST' dropdown and the full URL `http://10.88.250.176:8080/autocomplete`. The 'Body' tab is active, showing the following form-data entry:

Key	Value
<input checked="" type="checkbox"/> userUtterance	I want
New key	Value

Below the body, the 'Body' tab is again highlighted, and the response pane displays the JSON output:

```
{"list": ["to", "the"]}
```

Appendix -8 – APIs to capture user feedback (like and dislike)

...5/7

In order to capture user feedback on whether he/she has got satisfied answer or not from BOT , after every task completion , user will be provided with like and dislike option. The count of likes and dislikes is maintained and can be seen in status API call. (see slide 51) Here is flow for taking user feedback -

- 1- User is presented with Like and Dislike icon (using iCard json) after every task completion
2. User clicks Like or Dislike button (optional for user to click or not)
- 2- APIs are called upon clicking the icons and count of the same is maintained
- 3- The task information is logged into a log file <mmm>_<dd>_likes.log and <mmm>_<dd>_dislikes.log respectively in /res/logs/training folder. (see slide 26)

Following APIs are provided to take user feedback.

API

LIKE

Method -> **POST**

URL -> <http://<IP>/api/v2/bot/{instanceId}/like>

e.g. <https://10.88.246.15:8080/api/v2/bot/d2-L2IDCOQCFFCJ/like>

Response -> { "response": "Info:registered like response"}

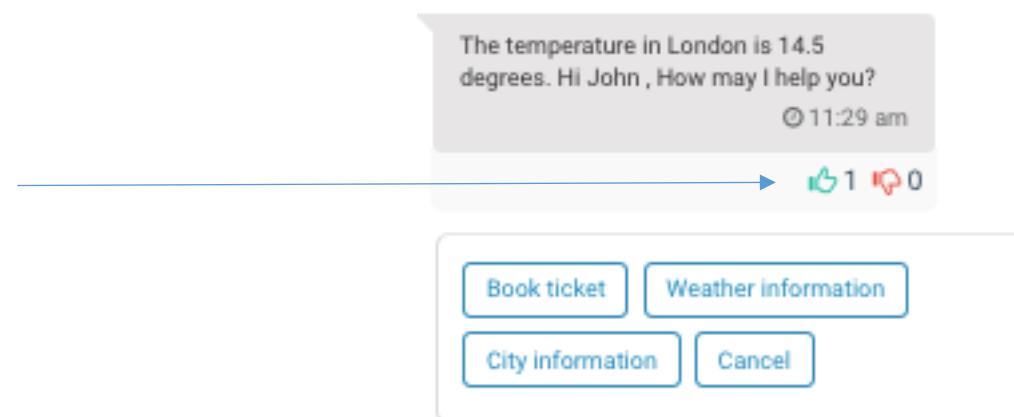
DISLIKE

Method -> **POST**

URL -> <http://<IP>/api/v2/bot/{instanceId}/dislike>

e.g. <https://10.88.246.15:8080/api/v2/bot/d2-L2IDCOQCFFCJ/dislike>

Response -> { "response": "Info:registered dislike response"}



Appendix -8 – Manage push notifications – for REST based Bot

...6/7

This feature supports sending notifications to bot user that are generated by external systems. (next best conversation)

This can also be useful for scenarios where backend workflows take long time to process the request. In this case, follow below steps

1. Send 201 OK HTTP response so that user can continue to interact with Bot without interruption.
2. Trigger the backend workflow
3. Call sendNotification API with proper message post completion of task
4. Bot JSON response will show notification =1 in its response
5. Client then can call getNotification API to show the response to user

sendNotification

In order to send notification to user following API is available

Method -> **POST**

URL -> <http://<IP>/api/v2/bot/{instanceId}/sendNotification> or
<http://<IP>/api/v2/bot/{user}/sendNotification>

Param - url-encoded-body parameter –

key => **message** value => **Your Application is processed ID-65457**

e.g. <http://192.168.0.103:8080/api/v2/bot/d1-YDEHE06UPRMM/sendNotification>

Note - You can send notification to all running bots by using instanceId as ALL

e.g. <http://192.168.0.103:8080/api/v2/bot/ALL/sendNotification>

getNotification

In order to get notification following API is available

Method -> **POST**

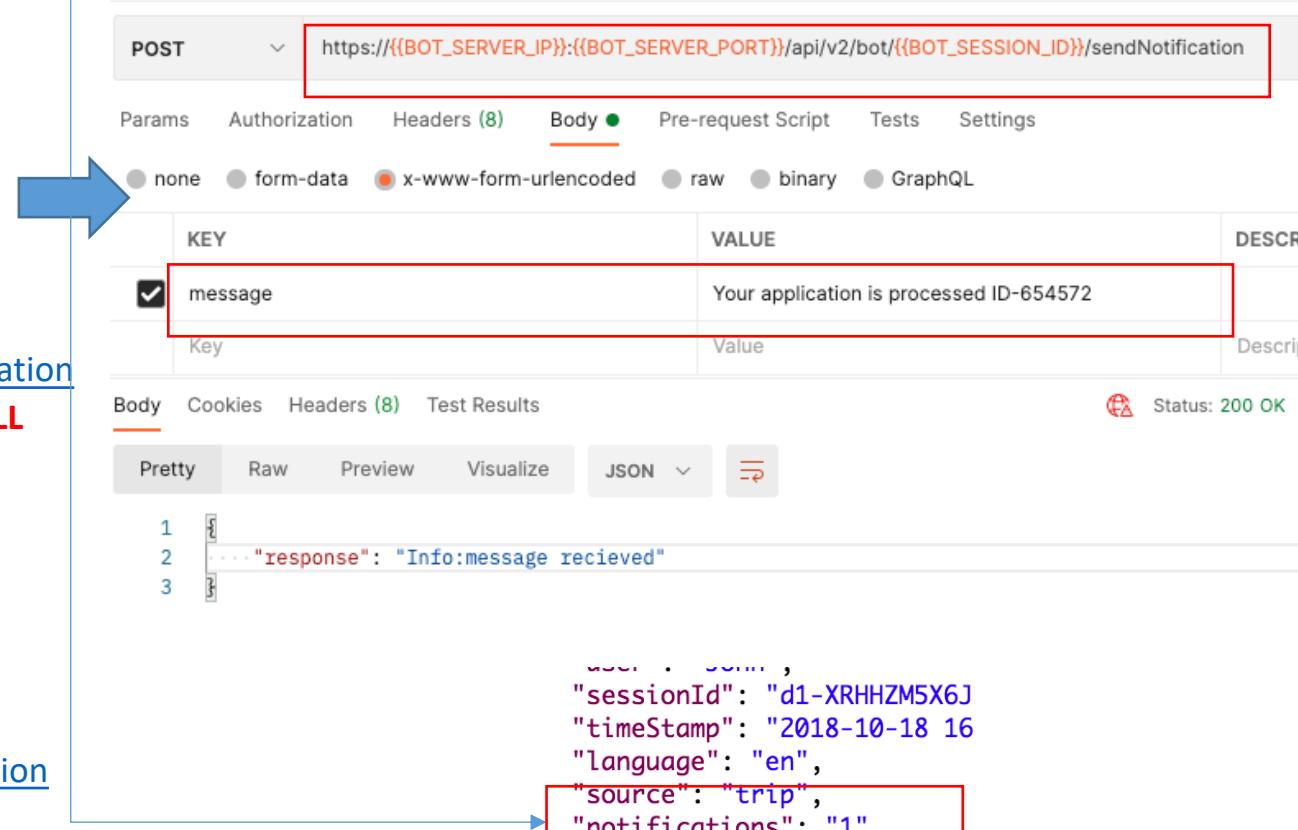
URL -> <http://<IP>/api/v2/bot/{instanceId}/getNotification> or
<http://<IP>/api/v2/bot/{user}/getNotification>

e.g. <http://192.168.0.103:8080/api/v2/bot/d1-YDEHE06UPRMM/getNotification>

Sample Response -

```
"notifications": [  
    {  
        "id": "d1-XRHHZM5X6JR4",  
        "message": "Your Application is processed. ID-65457"  
    }  
]
```

Note – if message starts with “@<eventfilename>” then you should process it on client side by passing it as userUtterance (e.g. @offerCard will trigger the event by passing userUtterance as “@offerCard”)



The screenshot shows a Postman interface with a POST request to `https://{{BOT_SERVER_IP}}:{{BOT_SERVER_PORT}}/api/v2/bot/{{BOT_SESSION_ID}}/sendNotification`. The 'Body' tab is selected, showing a key-value pair: 'message' with the value 'Your application is processed ID-65457'. The response section shows a status of 200 OK with a JSON body containing a single notification entry:

```
1  {  
2      "response": "Info:message received"  
3  }
```

```
...  
"sessionId": "d1-XRHHZM5X6J",  
"timeStamp": "2018-10-18 16",  
"language": "en",  
"source": "trip",  
"notifications": "1",  
"result": {  
    "query": "A"
```

Appendix -8 – Manage push notifications - for WebSocket based Bot

...7/7

This feature supports sending notifications to bot user in real time that are generated by external systems. (next best conversation)
For WebSocket based Bot the messages will be delivered in real time due to bidirectional data flow.

sendNotification

In order to send notification to user following API is available

Method -> **POST**

URL -> *http://<IP>/api/v2/bot/{instanceId}/sendNotification*

Param - url-encoded-body parameter –

key => **message** value => **Your Application is processed ID-65457**

e.g. <http://192.168.0.103:8080/api/v2/bot/d1-YDEHE06UPRMM/sendNotification>

Note - You can send notification to all running bots by using instanceId as ALL

e.g. <http://192.168.0.103:8080/api/v2/bot/ALL/sendNotification>

The messages will be delivered in real time to intended Bot users with standard JSON response

POST https://{{BOT_SERVER_IP}}:{{BOT_SERVER_PORT}}/api/v2/bot/{{BOT_SESSION_ID}}/sendNotification

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Body (x-www-form-urlencoded)

KEY	VALUE
message	Your application is processed ID-654572

Status: 200 OK

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON

```
1 "response": "Info:message received"
```

If you need to initiate any conversation through back end you can send the message that starts with @<eventfilename> e.g. @offerCard
Please note that this will process the dialog on server side and will deliver the dialog directly to Bot.

Appendix –9 – APIs to manage next best conversation

This feature supports sending notifications to bot user that can trigger the next best conversation. This can be generated from backend systems.

This will be useful for scenarios where backend wants to trigger the conversation with user through event dialog. (slide – 29)

This uses the sendNotification API with a difference that your message parameter contains the “@<eventName>” . The eventName is your event file name. (yaml file)

sendNotification

In order to send notification to user following API is available

Method -> **POST**

URL -> *http://<IP>/api/v2/bot/{instance_id}/sendNotification*

Param - url-encoded-body parameter –

key => **message** value => **@loanProduct**

e.g. <http://192.168.0.103:8080/api/v2/bot/d1-YDEHE06UPRMM/sendNotification>

The screenshot shows a Postman interface with a red box highlighting the URL field: `https://{{BOT_SERVER_IP}}:{{BOT_SERVER_PORT}}/api/v2/bot/{{BOT_SESSION_ID}}/sendNotification`. A blue arrow points from the text "In order to send notification to user following API is available" to this URL. Below the URL, the "Body" tab is selected, showing a key-value pair: `message: @loanProduct`. The response body is displayed as JSON: `{ \"response\": \"Info:message received\" }`.

Important –

The client can now poll the server for any new notifications or next best conversation by sending “**userUtterance**” parameters as “**polling-query**”

Bot server will process this request purely as a polling request without taking any action.

Method – **POST**

URL - <https://10.88.246.15:8080/api/v2/bot/d1-XRHHZM5X6JR4>

Param - url-encoded-body parameter – key => **userUtterance** value => **polling-query**

Appendix -10 – APIs to upload the content to server

In order to provide the user to upload the file content to BOT server following API is available

Method -> **POST**

URL -> *http://<IP>/api/v2/bot/{instanceId}/upload*

Form Data body parameter -> file=<Selected File>

e.g. <https://192.168.0.103:8080/api/v2/bot/d1-DVRQ10Z6AXD6/upload>

1- On client user will select the file to be uploaded.

2- Once uploaded the file will be stored on server in /res/upload folder with following prefix - <User>_<Time>_<FileName>

Example => John_Fri_Jan_29_18_03_37_IST_2021_uwg.jpg

The screenshot shows the Postman application interface. A red box highlights the URL field which contains `https://{{BOT_SERVER_IP}}:{{BOT_SERVER_PORT}}/api/v2/bot/{{BOT_SESSION_ID}}/upload`. Below the URL, the 'Body' tab is selected, indicated by a green dot. Under the 'Body' tab, the 'form-data' option is selected, shown by a red circle. A table below shows a single key-value pair: 'file' with the value 'uwg.jpg'. At the bottom right, the status is shown as 'Status: 200 OK'.

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> file	uwg.jpg	

```
1 {"response": "Info:successfully uploaded the file: John_Mon_Feb_01_11_25_46_IST_2021_uwg.jpg "}
```

Appendix -11 – Creating Hindi Bot

One can create Hindi Bot by following below mentioned steps -

1- Create the `<domain>.hi.json` file that contains all the possible user utterances in Hindi (e.g. in this case “`trip_hi.json`” file to be stored in `/CoreNLP/data` folder of NLP Engine)

2 – Add domain specific synonyms , if any to “`synonyms_hi.txt`” file located in `/CoreNLP/dictionary` folder

3 – Create the DDF file `<domain>.xml` that contains all hindi specific messages. Ensure `<globalLangauge>` set to “hi”. (e.g. in this case `trip.xml` to be stored in `/res/dialogues` folder)

4 – The utterances in Hindi are processed in English so the HINDI is transliterated into ENGLISH during utterance processing by BOT engine

e.g. मैं इकॉनमी क्लास से जाऊंगा => mai ikonami klasa se ja'uniga

So all the entities need to be accordingly created. Example below shows Custom.item_1 entity for getting class of travel -

```
##DO NOT REMOVE THIS LINE
Economy=Economy
ikonami=Economy
First=First
Business=Business
Business=Business
```

To accommodate the transliteration issues

```
{
  "domain": "trip",
  "tasks": [
    {
      "name": "getTripInformation",
      "utterances": ["मैं टिकट बुक करना चाहता हूँ",
        "मुझे टिकट बुक करना है",
        "मुझे रिजर्वेशन करना है",
        "मुझे रिजर्वेशन करना है",
        "मैं यात्रा करना चाहता हूँ",
        "मैं टिकट आरप्पिग करना चाहता हूँ",
        "क्या आप कृपया मेरे लिए टिकट बुक कर सकते हैं"]
    },
    {
      "name": "getWeatherInformation",
      "utterances": ["मौसम कैसा है?"]
    }
  ]
}
```

```
<startTaskName>getWeatherInformation</startTaskName>
<globalLanguage>hi</globalLanguage>
<useSODA>true</useSODA>
<allowSwitchTasks>true</allowSwitchTasks>
<allowOverAnswering>true</allowOverAnswering>
<allowDifferentQuestion>true</allowDifferentQuestion>
<allowCorrection>true</allowCorrection>
<tasks>
  <task name="start"> ... </task>
  <task name="getTripInformation" label="टिकट बुकिंग"> ...
  <task name="getWeatherInformation" label="मौसम की जानकारी">

```

In order to know the transliterated words that needs to go in entity follow below steps –

- create file `utternce.dat` in `/res` folder that contains all possible utterances

- go to `hmi.jar` and run below command

- `>java -cp hmi.jar cto.hmi.bot.util.Transliteration -f /res/utterance.dat -t DEV_TO_ENG ..(for European use EU_TO_ENG , for arabic ARB_TO_ENG or ENG_TO_DEV, ENG_TO_ARB)`

- It will process file and will generate the output file `utterance_out.dat` with transliterated words that can be used in entity. OR

`>java -cp hmi.jar cto.hmi.bot.util.Transliteration -u "viajar con clase ejecutiva" -t EU_TO_ENG -process utterance directly`

मैं business क्लास से जाऊंगा |mai business klasa se ja'uniga
मैं इकॉनमी क्लास से जाऊंगा |mai ikonami klasa se ja'uniga

..(for European use EU_TO_ENG , for

`>java -cp hmi.jar cto.hmi.bot.util.Transliteration -u "viajar con clase ejecutiva" -t EU_TO_ENG -process utterance directly`

Appendix -12 – Creating AR Step Instructions Guide Application

...1/4

In order to create the augmented reality application, one need to take following approach.

- Create the dynamic reference Image library in CSV file (e.g. *refImageData.csv*)

```
#ID,ReferenceImage URL,width in centimeter  
TFPowerSupply,"https://192.168.0.103:8080/content/TFPowerSupply.png","20.0"  
Glucometer,"https://192.168.0.103:8080/content/gmeter.png","5.8"
```

- Upload this csv file so that it can be accessed using URL. Enter this URL in Setting option provided in ARIA Android App.
(e.g. <https://13.126.162.96:8002/content/refImageData.csv>)
- Enter the steps that user has to follow in a csv file.

```
#ID,Label,Instruction,"Position<x,y,x> in cm",Rotation<r>,Scale<s>,Target<name>,Overlay_ID,ImageOrText<*.png>/<text>,"Position<x,y,z> in cm",Rotation<r>,Scale/Size<s>,"Color<0-255,0-255,0-255,0-255>"  
GLUCO_USE,STEP-1,Please plug in power adapter before taking the reading.,"5.8,9.64,0",0,0.8,Glucometer,ID-1,https://192.168.0.101:8080/content/arrow_small.png,"5.3,2.76,0",90,0.6,,  
GLUCO_USE,STEP-2,Insert a test strip to the bottom of your meter.,"5.8,9.64,0",0,0.8,Glucometer,ID-2,https://192.168.0.101:8080/content/arrow_small.png,"3.02,8.02,0",180,1.16,,  
GLUCO_USE,STEP-3,Screen will display a strip with flashing blood drop to indicate that it is ready.,"5.8,9.64,0",0,0.8,Glucometer,ID-3,https://192.168.0.101:8080/content/arrow_small.png,"2.02,2.18,0",0,1.24,,
```
- Upload this csv file in **/res/idata** folder (e.g. */res/idata/gmeter_ARCard.csv*)
- Now user can navigate these steps by creating event file *gmeterUserGuide.yml* file in */res/events* folder. (see next slide)
- The event file can be called from main dialogue file by using “@gmeterUserGuide.yml” utterance.

Appendix -12 – Creating AR Step Instructions Guide Application

...2/4

This yaml file will help user navigate through different steps.

```
---
```

```
tasks :
```

```
- task :
```

```
  name : EVT_getARReady
```

```
  label : Enable Camera
```

```
  entities :
```

```
    - entity :
```

```
      name : stepCounter
```

```
      label : ''
```

```
      required : true
```

```
      answerType : custom.item_ok
```

```
      fallbackQuestion : 'Focus your camera to your object and S
```

```
      clarifyQuestion : 'Please provide me with correct option, s
```

```
      storeCache : true
```

```
action :
```

```
  type : groovyAction
```

```
  resultMappings :
```

```
    - map :
```

```
      message : ''
```

```
      redirectToTask : EVT_stepProcessor
```

```
      resultVarName : 'action'
```

```
      # use quotes for key
```

```
      resultValue : '1'
```

```
returnAnswer : true
```

```
utteranceTemplate : '#result'
```

```
code :
```

```
import cto.hmi.idatautil.ProcessARData;
```

```
String itemFile = "gmeter_ARCard.csv";
```

```
String id = "GLUCO_USE";
```

```
int stepNumber = 1;
```

```
String body = ProcessARData.GetJSONString(itemFile,id,stepNumber);
```

```
executionResults.put("stepCounter","1");
```

```
executionResults.put("action","1");
```

```
executionResults.put("body",body);
```

```
'
```

The csv file name should be the same as that uploaded in /res/idata folder

```
- task :
```

```
  name : EVT_stepProcessor
```

```
  label : Get Model
```

```
  entities :
```

```
    - entity :
```

```
      name : getCommand
```

```
      label : ''
```

```
      required : true
```

```
      answerType : custom.item_ar_commands
```

```
      fallbackQuestion : ''
```

```
      clarifyQuestion : 'Please provide command like Previous or Next'
```

```
    - entity :
```

```
      name : stepCounter
```

```
      label : ''
```

```
      required : true
```

```
      answerType : custom.item_ok
```

```
      fallbackQuestion : ''
```

```
      useContext : true
```

```
action :
```

```
  type : groovyAction
```

```
  resultMappings :
```

```
    - map :
```

```
      message : ''
```

```
      redirectToTask : start
```

```
      resultVarName : 'action'
```

```
      # use quotes for key
```

```
      resultValue : '2'
```

```
    - map :
```

```
      message : ''
```

```
      redirectToTask : EVT_stepProcessor
```

```
      resultVarName : 'action'
```

```
      # use quotes for key
```

```
      resultValue : '1'
```

```
returnAnswer : true
```

```
utteranceTemplate : '#result'
```

```
code :
```

```
import cto.hmi.idatautil.ProcessARData;
```

```
String itemFile = "gmeter_ARCard.csv";
```

```
String id = "GLUCO_USE";
```

```
String stepNo = frame.get("stepCounter");
```

```
int stepNumber = 1;
```

There are 2 methods available -
ProcessARData.GetJSONString(String fileName, String ID, int record) and
ProcessARData.GetJSONString(String csvData, String cardtype) based on your need

```
try
```

```
{
```

```
  stepNumber = Integer.parseInt(stepNo);
```

```
}
```

```
catch(NumberFormatException e)
```

```
{
```

```
  System.out.println("Error in processing utterance");
```

```
}
```

```
String command = new String(frame.get("getCommand"));
```

```
String body = "";
```

```
int totalRecords = ProcessARData.NoOfARRecords(itemFile,id);
```

```
if (command == "PREV" && stepNumber !=1 )
```

```
{
```

```
  stepNumber = stepNumber - 1
```

```
}
```

```
else if (command == "NEXT" && stepNumber < totalRecords)
```

```
{
```

```
  stepNumber = stepNumber + 1;
```

```
}
```

```
else
```

```
{
```

```
  ;
```

```
}
```

```
if (command != "EXIT")
```

```
{
```

```
  body = ProcessARData.GetJSONString(itemFile,id,stepNumber);
```

```
}
```

```
if (command == "EXIT")
```

```
{
```

```
  executionResults.put("action","2");
```

```
  executionResults.put("body",body);
```

```
}
```

```
else
```

```
{
```

```
  executionResults.put("stepCounter",String.valueOf(stepNumber));
```

```
  executionResults.put("action","1");
```

```
  executionResults.put("body",body);
```

```
}
```

```
'
```

The id should be same as that in csv file.

Appendix -12 – Creating AR Step Guide Application

...3/4

In AR Step Guide application, the JSON response shows all the required attributes that can be parsed to create the augmentation using Unity engine.

You can overlay as many images and texts as you want by appending the information sequentially in csv file

Message type is ARCardStepGuide

```
"message": {
    "data": {
        "overlayItems": [
            {
                "color": "",
                "imageOrText": "https://192.168.0.101:8080/content/arrow_small.png",
                "rotation": "270",
                "scale": "0.8",
                "id": "ID-6",
                "position": "4.1,2.72,0"
            },
            {
                "color": "255,255,0,255",
                "imageOrText": "Your Reading",
                "rotation": "0",
                "scale": "42",
                "id": "ID-5",
                "position": "4.7,3.72,0"
            }
        ],
        "panelInfo": {
            "rotation": "0",
            "scale": "0.8",
            "step": "last",
            "id": "GLUCO_USE",
            "label": "STEP-5",
            "position": "6,6,0",
            "info": "Your blood meter reading will be available now on screen. You may take experts help.",
            "target": "CiscoModem_B"
        }
    },
    "chat": "Your blood meter reading will be available now on screen. You may take experts help.",
    "type": "ARCardStepGuide"
}
```

Image Overlay

Text Overlay

Info Panel position

Appendix -12 – Creating AR Step Instructions Guide Application

...4/4

You need to configure the steps by providing the relevant information in csv file uploaded in **/res/idata** folder (e.g. **/res/idata/gmeter_ARCard.csv**). Here are the steps that you need to follow to capture the proper X,Y coordinates.

Step-1

Resize your image to physical dimension with Resolution of 100 pixel/cm before uploading it to ref Image library. (In this case a glucometer with width of 5.8 cm)

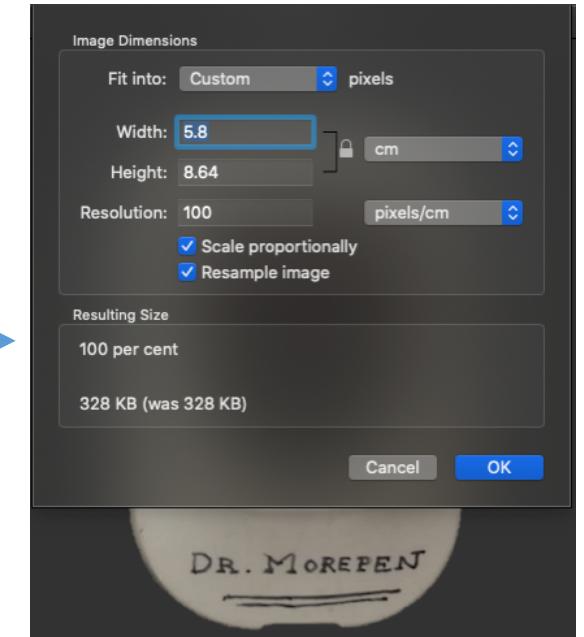
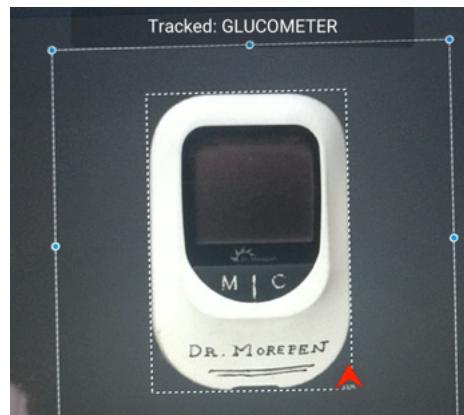
Step-2

Upload csv file so that it can be accessed using URL (*refImageData.csv*)

Step-3

Configure your *ARCard.csv* with image overlay positions in cm's keeping **left bottom corner** as reference point.

As can be seen from below screen shot , *arrow.png* image centre position is aligned to specified position. (for example, 0,8.64 is right bottom corner)



```
#ID,ReferenceImage URL,width in centimeter
Glucometer,"https://192.168.0.101:8080/content/glucoMeter.png","5.8"
#ID,Label,Instruction,"Position<x,y,x> in cm",Rotation<r>,Scale<s>,Target<name>,Overlay_ID,ImageOrText<*.png/><text>,"Position<x,y,z> in cm",R
GLUCO_USE,STEP-1,Please plug in power adapter before taking the reading.,"5.8,9.64,0",0,0.8,Glucometer,ID-1,https://192.168.0.101:8080/content/glucoMeter.png
GLUCO_USE,STEP-2,Insert a test strip to the bottom of your meter.,"5.8,9.64,0",0,0.8,Glucometer,ID-2,https://192.168.0.101:8080/content/arrow.png
GLUCO_USE,STEP-3,Screen will display a strip with flashing blood drop to indicate that it is ready.,"5.8,9.64,0",0,0.8,Glucometer,ID-3,https://192.168.0.101:8080/content/glucoMeter.png
GLUCO_USE,STEP-4,Hold the test strip with your blood drop till the meter beeps.,"5.8,9.64,0",0,0.8,Glucometer,ID-4,https://192.168.0.101:8080/content/glucoMeter.png
GLUCO_USE,SETUP-5,Your blood meter reading will be available now on screen. You may take expert assistance by selecting that option.,"5.8,9.64,
```

Appendix -13 – Supporting new language to BOT

In order to add new language to BOT following steps are required

- For generic conversation add required utterances in AIML format in `/res/bots/aiml/Generic_xx.aiml` file. e.g.,
`<category><pattern>jeg har det godt </pattern> <template>godt at høre, at </template> </category>`
- Add necessary utterances in `/res/bots/aiml/Nlgfiller.aiml` to cover the NLU generation.
- Create the dialog file that is specific to given language. Ensure that XML tag `<globalLanguage>` specifies the language code e.g., `da` for Danish
- Add all the stopwords in `/res/dictionary` folder and name it as `stopwords_xx.txt` (where xx is language code)..DO=dog etc.
- All entities in `/res/entities` needs to be created in transliterated English for engine to identify.
- Create the intents and its utterance variations to train bot using NLP Engine . (`/config/trainingData/intent`) file.
- Create the dialogue and events in `/res/dialogues` and `/res/events` folder respectively

(This section is not for Bot Builder, only for Bot Developer)

- Add necessary utterances to SODA classifier (`/res/dictionary`)to identify whether utterance is of information, command or seek type.
- Add the locale and region combination to `DialogManagerContext` , and transliteration details in `DialogManagerHelper.class`
- Add the standard messages that are part of core conversation i.e. `SORRY_MSG`(Sorry I did not understand that), `GOT_MSG`(I go that) etc. to `/res/config/appMessages_xx.properties` file. (where xx is language code. For example, `da` for danish)

(IMPORTANT – The nonEnglish characters must be entered in Unicode's e.g `OR_MSG=u0905u0925u0935u093E` ,take help of [converter tool](https://r12a.github.io/app-conversion/))

Appendix 14 – Deploying Bot Server with Broker Service

...1/3

In order to deploy Bot, we need following services to be running in the order mentioned below

- Broker Engine
- NLP Engine and
- Dialog Engine

1. BROKER ENGINE

You need to set up Kafka server first for using Broker Service.

Follow this step (as per the [article](#))—

- Download the latest binary stable version of Kafka from [here](#).
- Unzip this file.
- The Kafka instance (Broker) configurations are kept in the config directory

server.properties configuration (keep other parameter to default values)

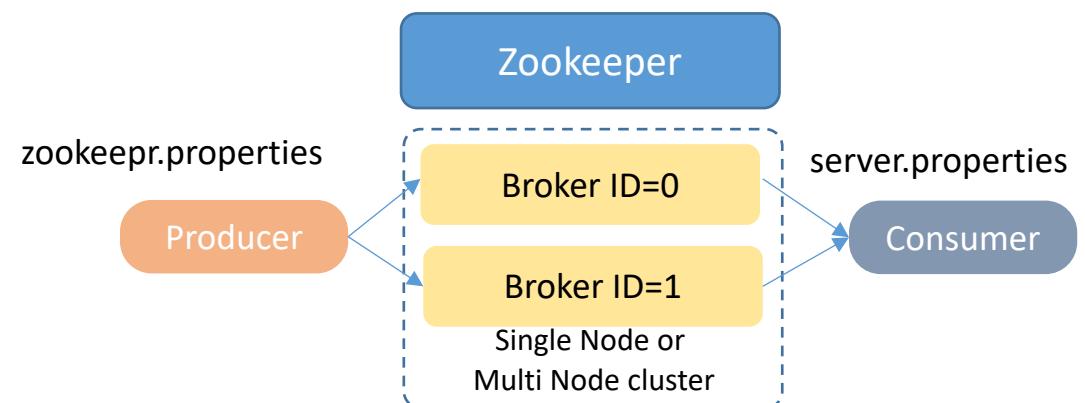
- *listeners=PLAINTEXT://0.0.0.0:9092*
- *Advertised.listener=PLAINTEXT://<BROKER_IP>:9092*
- *broker.id=0*
- *log.dirs to <kafka_home_directory>/data/kafka/broker_0*
- *log.retention.hours=24*
- *zookeeper.connect=<BROKER_IP>:2181*
- Add below parameters at the end
delete.topic.enable=true
auto.create.topics.enable=false

Run Kafka Server

- Go to *bin* folder of your kafka server
- Start the zookeeper Service (for linux)
`$ zookeeper-server-start.sh ..//config/zookeeper.properties`
- Start Kafka broker(for linux)
`$ kafka-server-start.sh ..//config/server.properties`

zookeeper.properties configuration

- *dataDir to <kafka_home_directory>/data/zookeeper*

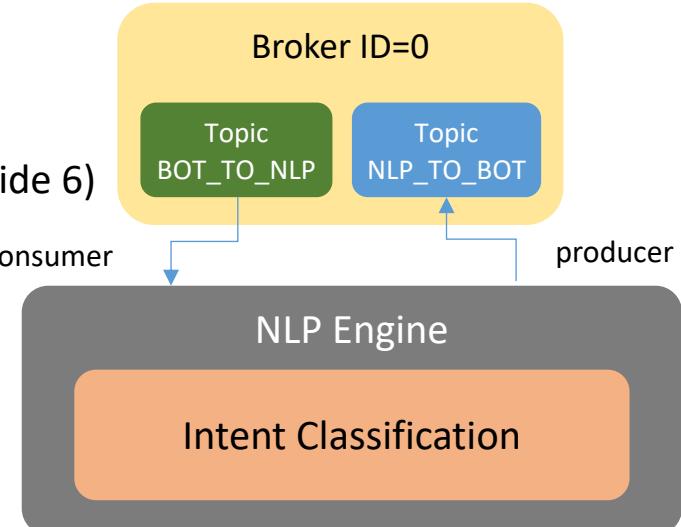


Appendix 14 – Deploying Bot Server with Broker Service

...2/3

2. NLP ENGINE:

- Download and unzip *NLP Engine.zip*
- Add necessary python modules as explained in “Getting started” guide – (slide 6)
- Set *USE_BROKER = true* in *nlp.properties* located in *Core_NLP/config* folder
- Run the NLP Engine service –
`$ python app.py`
- This command will start NLP engine service
- It is now ready to listen to broker messages



3. DIALOG ENGINE:

- Download and unzip *DialogEngine.zip*
- Set *USE_BROKER = true* in *bot.properties* located in */res/config* folder
- Run the Dialog Engine services for sample trip_en dialog –
`$ java -jar hmi.jar -i rest -r trip -p 8080`
- It is now ready for trip booking use case (located at *\res\dialogues* folder)

Now you are ready to run a sample trip booking dialog

- It uses inbuilt jetty server and will listen at port 8080
- Go to chrome browser and enter URL - <https://<Your IP>:8080/en>

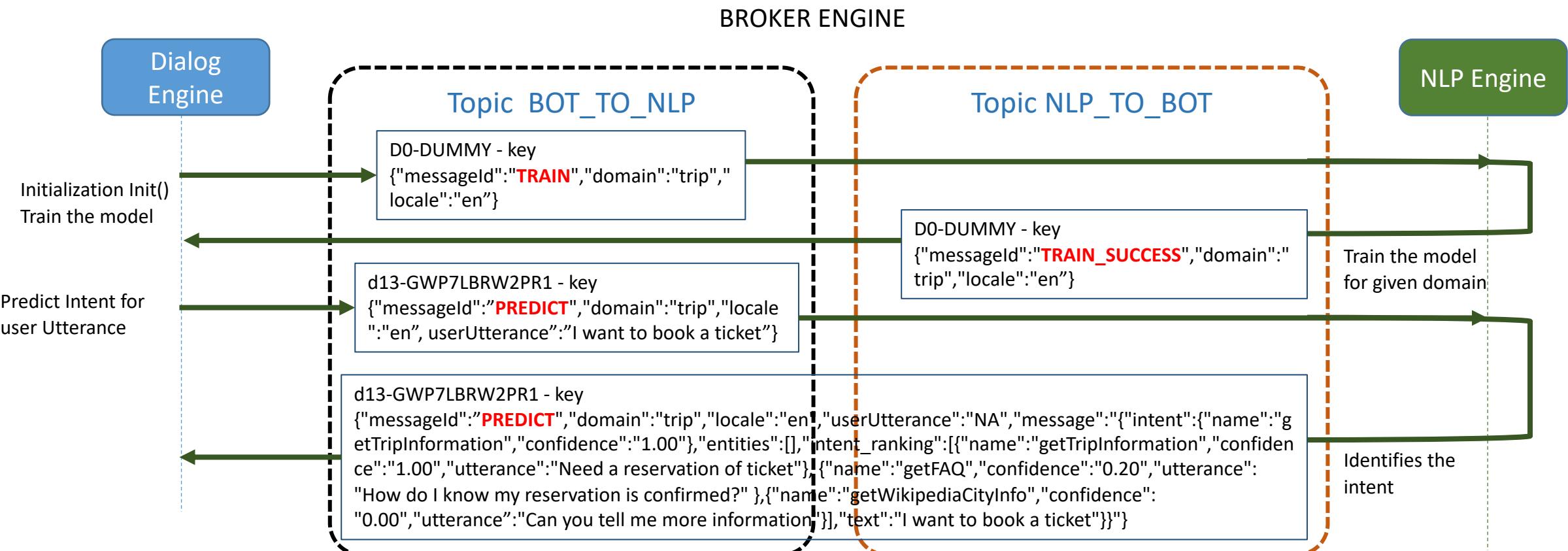
B: Good evening, How may I help you?
U: Good evening
B: Good evening, Friend, hope you are doing good today. How may I help you?
U: I want to book a ticket
B: Please tell me what city do you want to travel?
U: I want to start from Atlanta
B: And for how many persons?
U: How is weather in Atlanta?
B: The temperature in Atlanta is 21.54 degrees. for how many persons?
U: for 4 persons
B: When do you want to leave?
U: I want to leave on Sunday
B: which class do you want to travel?
U: Business
B: Hey John, this trip to Atlanta costs 255 Dollars. What can I do for you?

Appendix 14 – Deploying Bot Server with Broker Service ..broker messages

...3/3

Core Dialog engine has in-built NLP to test BOT during the development and testing. However, for production , one should use the kafka based Broker services for handling large number of messages between BOT and NLP engine.

The high-level message flow is shown below



Thank you