AskAmex-CDMS

About CDMS

Code Based Dialog Management System(CDMS) is code based approach which automates the bot flow conversations with user or code-based approach for creating and maintaining bot responses with user.User can have many queries and each query may have different bot flow conversations.So firstly the conversation starts at Mainbot and based on the card-member request,the control is passed to appropriate microbot based on the user query.As we have discussed each query may have different bot-flows,so each microbot conversational flows are defined relevent to the requests satisfied by microbot.

With the goal of making the card member experience a seamless and easy one when chatting with one of our bots, the CDMS implements several

different shared services such as NLP, APIs, and machine learning.

General Concepts

What is a Microbot?

A bot(robot) is a software program which performs automated predefined tasks.Bots are of many types such as chatbots,google bots etc.Here we will be using only chat bots.

Chatbots imitate human conversation.So when user comes and ask a chatbot to perform certain task.The chatbot processes the user input without human intervence and performs the task and outputs the user in chatform or voice form.Here user feels like as if he is talking to a real human.

The bots we use in cdms are classified into mainbot and microbot.Firstly when the user initiates the conversation in the chatbot,the control of chatbot is controlled by mainbot.This input is processsed by NLP and intents(we will discuss about intents in the later part of this documentation) are derived from the utterance of the user.Then this mainbot maps the intents to appropriate microbot.Each microbot is configured to execute certain intents.So based on the Mainbot maps the intent to the relevant Microbot.

Intents

Suppose user utters input such as "I want to check my card balance".Intent is generally the intention behind the utterance of the user or the goal motive the user wants tho achieve.The intent for the user input "I want to check my card balance" is "get_card_balance".So the intent classification from user utterance is done by NLP(natural language processing) .

Entities

Entities are the specific details provided by the user to the bot.These details make the bot understand to perform functions on the entity value specified by user.These can also be defined as the details required by the bot to complete the goal of the user.Entities are objects that are extracted from user utterance.NLP processes user utterance and extracts entities from the utterance.For example if the user utterance is I want to know the balance of my blue card.Here the entity is the card_type and the value of entity is blue.

In this cdms, in every action file, we have entities object defined corresponding to each action.The attributes of entity object are:-

- state-this state attribute is used to update the entity in bot memory.The entity will be updated in bot memory as cdms(memory)[state]=entity. Example:-cdms(memory)[card_selector]=entity.Override variable checks whether to update the bot memory or not.
- the type of entity wanting to be capture (i.e. CARD_TYPE).
- This function checks whether the entities we get from the request are actually matching the user account details.

Actions

Actions are tasks executed or to be exeeuted by the chatbot.The actions drive the bot from one point to another point of conversation and helps in achieving the goal of user based on relevant inputs.

Each Microbot consists of action folder.This actions folder consists of the tasks required to be performed.The 'Actions' package will contain many modules (.py files) with a corresponding YAML file. Each of these module/YAML file pairs represents a certain action taken based on the conditions found in the bot memory.

Memory

Certain inputs the user makes need to be saved, so the bot has the requisite information to assist them properly. Requests cannot do this; anything we send in a request is immediately forgotten the moment we send a new one. The memory is a **dynamically generated JSON**, which means that **thr oughout the bot's flow we will add attributes to it**. For example, if the user reports a fraudulent charge on a card, the memory will maintain which one of their cards it is. It will also maintain which action to take next, so that on the next user utterance, we haven't lost our progress in the flow. Memory is the parameter used by the bot to keep in track of the conversation flow.

Installation & Setup

Environment (E0) Setup

Make sure to use Python Version 3.8. Version 2.7 is no longer compatible with c3ip CDMS.

Step 1. Setup ENV Variable PIP_CONFIF_FILE which holds pip.conf file. This is needed for installing dependencies.

1. Create file path: $HOME/.config/pip/pip.conf
2. Insert contents into pip.conf file:
   pip.conf

```
[global]
index = https://ci-repo.aexp.com/repository/pypi/simple/
index-url = https://ci-repo.aexp.com/repository/pypi/simple/
```

Steps 2 & 3 requires adding secrets, private keys, and certificates locally. For security reasons, we cannot publish them here. Please reach out to an AAA team member to be sent the details.

Step 2. Open Terminal:

1. Create directory and secrets file:

```
cd /opt/epaas/vault/secrets
touch secrets
```

2. Paste in secrets

```
aaa.functions.get.account.transaction.sor.v2.hmac.client.secret= XXXXXXXXXXXXXXX
aaa.functions.get.account.transaction.sor.v2.cas.api.id=XXXXXXXXXXXXXXX
aaa.functions.cbis.get.a2a.token.api.client.id=XXXXXXXXXXXXXXX
aaa.functions.cbis.get.a2a.token.api.client.secret=XXXXXXXXXXXXXXX
askamex.cdms.nlp.api.key=XXXXXXXXXXXXXXX
```

Step 3. Create the following directories and files locally:

- `/opt/epaas/certs/key.key`
- `/opt/epaas/certs/certs.cer`

Add in the private key and certificate once received from a team member.

Step 4. Clone down the askamex-cdms repository:

https://github.aexp.com/amex-eng/c3ip_askamex-cdms

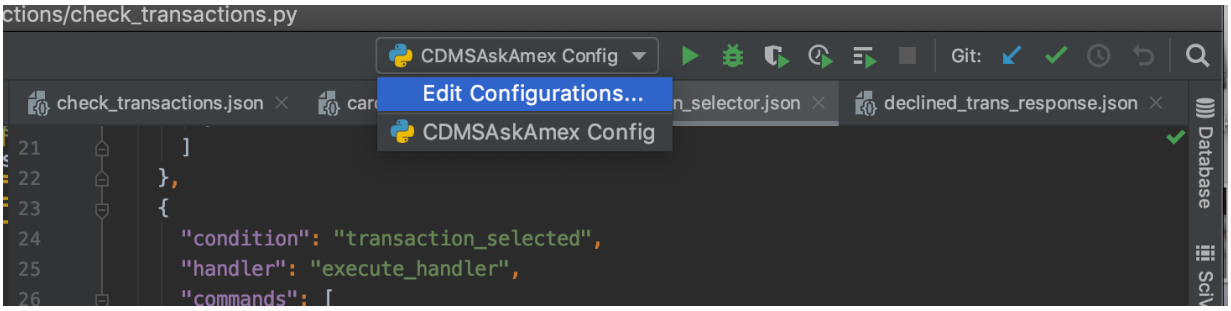
Step 5. We'll need to make a copy of the pem files from your local askamex-cdms repo (../askamex-cdms/.s2i) to a local folder under /opt:

```
mkdir -p /opt/app-root/src/.s2i


# copy the pem files to above .s2i folder
cp <folder of askamex-cdms repo>/.s2i/c3ip_kafka_* /opt/app-root/src/.s2i/
```

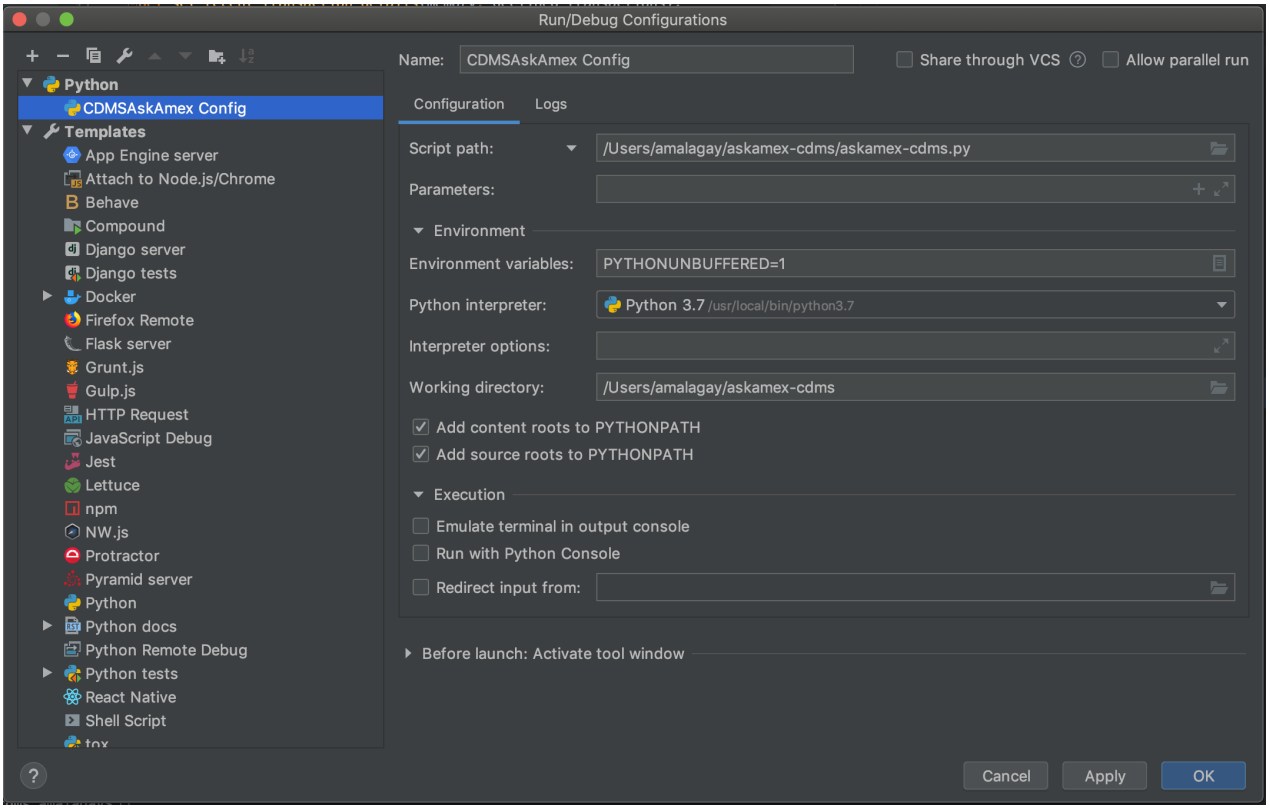
Step 6. Open PyCham to set up environment.

Click 'Edit Configurations' in the upper right tool bar.

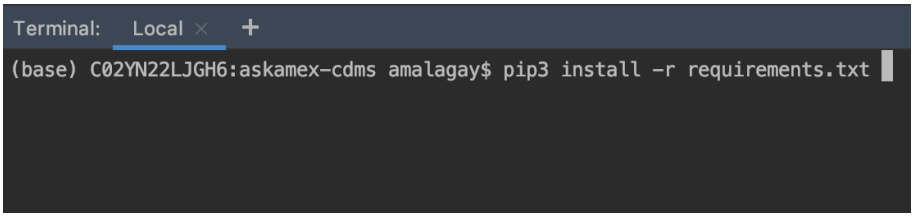


Select Python Template

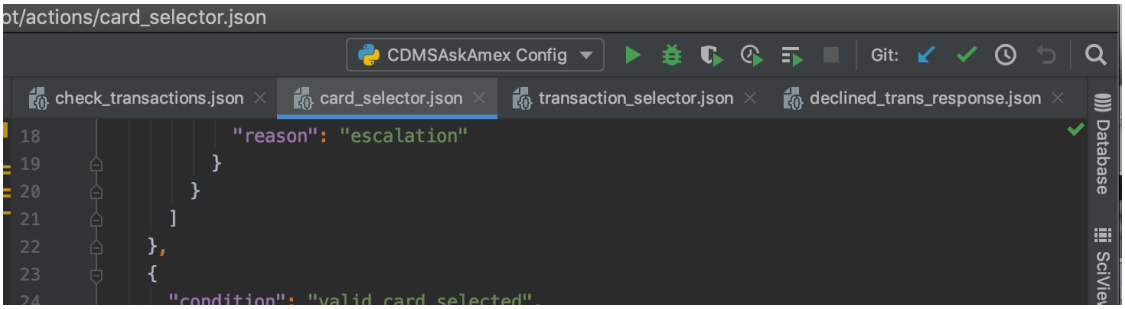
- Set *Script path* to location of **askamex-cdms.py** in your local **askamex-cdms** repo.
- Check that Python Interpreter is set to **Python >3.7** (in this case Python 3.8).
- Set *Working directory* to location of **askamex-cdms** repo.
- Click 'Apply' and 'Ok' when done



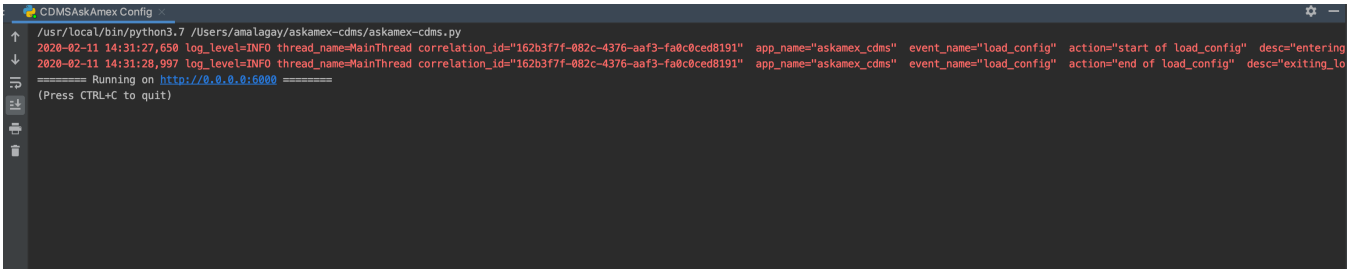
Navigate to **askamex-cdms** repo and install requirements.



Step 7. Click the green 'play' button in the upper right tool bar to run the configuration.



You'll see the logs in the Run window of PyCharm.

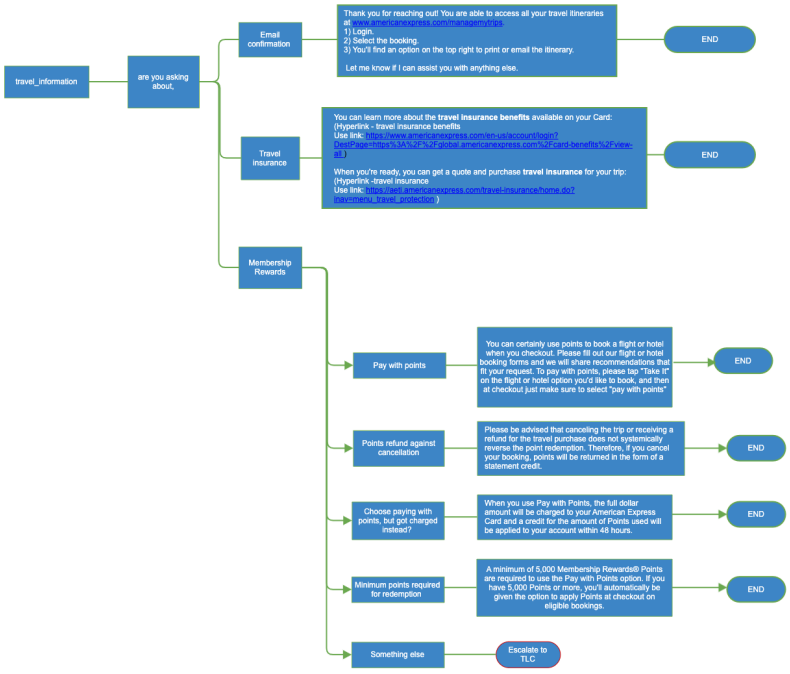


Now it's all set up and ready to run!

Build a Bot

Before taking a deep dive into parts of CDMS, let's take a high level look at how a developer would jump right into bot building. You can reference this recording of a previous workshop that goes over this process.

Starting out, we'll use this flow as an example of how we want our bot to behave:



The purpose of this bot is to provide the card member information about travel and membership rewards based on the **multi-choice options** the card member selects. We'll name this microbot **travel_bot**.

First let's take a look at the basic structure of how we're going to set up the travel_bot.

```
askamex-cdms
 bots
    travel_bot
       actions
       constants
       util
```

You'll place your microbot under the **bots** directory of the CDMS repository - this is were we keep all the microbots.

Under the travel_bot, we create three directories: actions, constants, and util.

- Actions - contains the python and YAML files of the bot. These files dictates what conversation "turn" the bot should take based on different inputs. Inputs could be from the user response, api data, or any combination of the two.
- Constants - variables referenced throughout the bot journey which remain unchanged,
- Util - can contain various user-defined functions that support the bot logic that is executed in actions.

Starting at the very beginning of the flow we see ***travel_information.*** As previously mentioned this is the intent that is resolved from a user utterance. For example, the utterance "I need help with travel" will be resolved by NLP to the intent ***travel_information*** which will direct the card member to the travel_bot.

In order for the card member to be taken to the travel_bot, the entry point action file **intent_received.py** and **intent_recieved.yaml** is required. An **inte nt_received** python and yaml file are required for all bots.

intent_received.py

```
import os

from cdms import capture_config_based_requirements, execute_action_v1, execute_handler
from cdms.util import cdms

from bots.travel_bot.util.bot_util import bot_config

ACTION_CONFIG_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'intent_received.yaml')

def capture_requirements(memory, request):
    capture_config_based_requirements(memory, request, ACTION_CONFIG_PATH)


def execute(memory, request):
    actions = {
        is_travel_info_intent: execute_handler
    }
    return execute_action_v1(memory, request, ACTION_CONFIG_PATH, bot_config(), actions)


def is_travel_info_intent(param):
    return cdms(param.memory).get('intent_name') == 'travel_information'
```

In the python file, provide the function definition **capture_requirements** - this is a standard function definition that can be reused across all bots and is used to capture intents, user selections, and functions. More on this later! Next, list the set of conditions that can be met within the actions of the execute definition with its handler. In this case, there is a single action *is_travel_info_intent* using a default handler.
Next, define the criteria in which the condition is met by writing the function definition of the action. It's important to note that only one action definition should be evaluated to be true if there is more than one action.

The below code block defines that the intent_name captured matches the string *travel_information*. If this condition is found to be true, then a set of commands defined in the yaml file is executed.

Action definition

```
def is_travel_info_intent(param):
    return cdms(param.memory).get('intent_name') == 'travel_information'
```

Moving onto the corresponding .yaml file - define the name to match that of the python file (intent_received). Because the bot needs to capture the intent of the utterance given by the card member - define the kind of requirements captured. In this case, the capture type is *intent* and the intents that will match to the travel bot are listed under the *in_scope_intents*.

Under events are the condition names defined in the python file. The commands underneath the condition are what are executed if the condition is found to be true. The next action or turn that the bot should take is defined with the operation next_action with the file name that contains those actions (i.e. execute definition) - in this case the next python file will be named travel_info_options.

The next operation is update_memory which will update the bot memory. The last operation is to send a message back to the user. The template that contains the words and options are configured in Dialogue Management Services (DMS). In the use case, the bot sends a message asking for the user to select from 3 options: Email Confirmation, Travel Information, Membership Rewards. You'll see how these options come into play in the next_action: travel_info_options.

intent_received.yaml

```
name: intent_received
requirements:
  action: intent_received
  capture_type: intent
  in_scope_intents:
    intents:
    - travel_information
    threshold: 0.3
events:
- condition: is_travel_info_intent
  handler: execute_handler
  commands:
  - operation: next_action
    inputs:
      action: travel_info_options
  - operation: update_memory
  - operation: send_message
    inputs:
      template_name: travel_info_multi_choice
```

Moving onto the next action file (travel_info_options), you'll see that its similarly formatted in that it contains a definition for capture_requirements, execute, and action conditional definitions. Most action files will follow this structure. The purpose of this action file is to execute the commands based on the user selection from the multi-choice options they were presented.

Starting out, there is still a capture_requirements definition because we need to capture the choice selected by the user. Next, define the actions - sinc e there were three options to select from, there are three possible actions that can be taken. These actions are named accordingly based on the options presented to the user.

Finally, define the conditionals in which the actions will be evaluated to be true. It's important to note that in bot memory, *the choice that the user selects will be saved in memory under the action file name that it was captured in* (i.e. travel_info_options). You can access the user selection utterance by calling it from memory like so: **cdms(param.memory)['travel_info_options']**.

For example, if the user selects *Travel Insurance* from the options list, we can validate that to be true by comparing what is stored in memory to the string *Travel Insurance.* Repeat this logic for the rest of the options.

travel_info_options

```
import os

from cdms import capture_config_based_requirements, execute_action_v1, execute_handler
from cdms.util import cdms

from bots.travel_bot.util.bot_util import bot_config

ACTION_CONFIG_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'travel_info_options.yaml')


def capture_requirements(memory, request):
    capture_config_based_requirements(memory, request, ACTION_CONFIG_PATH)


def execute(memory, request):
    actions = {
        is_email_confirmation_selected: execute_handler,
        is_travel_insurance_selected: execute_handler,
        is_membership_rewards_selected: execute_handler
    }
    return execute_action_v1(memory, request, ACTION_CONFIG_PATH, bot_config(), actions)


def is_email_confirmation_selected(param):
    return cdms(param.memory)['travel_info_options'] == "Itinerary Email Confirmation"


def is_travel_insurance_selected(param):
    return cdms(param.memory)['travel_info_options'] == "Travel Insurance"


def is_membership_rewards_selected(param):
    return cdms(param.memory)['travel_info_options'] == "Membership Rewards"
```

Moving onto the corresponding .yaml file, you'll see again that it follows the same structure as the intent_received.yaml file. However, there are differences in the requirements.

For multi-choice options, we define the **capture_type** to be a **list** since it came from a list of options. We also define the **in_scope_utterances** to be those options that were provided to the user. These in_scope_utterances should be the exact string that is in the template that was presented to the user.

Going back to the flow, you see that 2 of the 3 conversations end after a message is sent: Email Confirmation and Travel Insurance. To handle cases like these, the flow of operations would be to send the message and relinquish control back to the main bot. This is achieved by the operation **send_message** and provide the template name that contains the message (again, configured in DMS). The send_message is followed up by a **relinquish_control** which basically ends the conversation and hands the user back to the main bot, where they can begin a conversation again.

For the last option that provides another multi-choice list to the user, you'll see that it follows the exact same order of commands as the previous condition in intent_recieved that sent a multi-choice option message to the user. Let's move onto the last actions file required to complete this conversation journey.

travel_info_options.yaml

```
name: travel_info_options
requirements:
  action: travel_info_options
  capture_type: list
  in_scope_utterances:
  - Itinerary Email Confirmation
  - Travel Insurance
  - Membership Rewards
events:
- condition: is_email_confirmation_selected
  handler: execute_handler
  commands:
  - operation: send_message
    inputs:
      template_name: email_confirmation_message
  - operation: relinquish_control
    inputs:
      action: relinquish_control
      template: User selected email confirmation, travel info flow completed
      reason: ''
- condition: is_travel_insurance_selected
  handler: execute_handler
  commands:
  - operation: send_message
    inputs:
      template_name: travel_insurance_message
  - operation: relinquish_control
    inputs:
      action: relinquish_control
      template: User selected travel insurance, travel info flow completed
      reason: ''
- condition: is_membership_rewards_selected
  handler: execute_handler
  commands:
  - operation: next_action
    inputs:
      action: membership_rewards_options
  - operation: update_memory
  - operation: send_message
    inputs:
      template_name: membership_rewards_multi_choice
```

For the last part of the journey where the user selects *Membership Rewards*, you should notice that it follows the exact same structure and logic as travel_info_options.

membership_rewards_options.py

```
import os

from cdms import capture_config_based_requirements, execute_action_v1, execute_handler
from cdms.util import cdms

from bots.travel_bot.util.bot_util import bot_config


ACTION_CONFIG_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'membership_rewards_options.
yaml')


def capture_requirements(memory, request):
    capture_config_based_requirements(memory, request, ACTION_CONFIG_PATH)


def execute(memory, request):
    actions = {
        is_pay_with_points_selected: execute_handler,
        is_refund_points_selected: execute_handler,
        is_pay_with_points_charges_selected: execute_handler,
        is_minimum_redeem_points_selected: execute_handler,
        is_something_else_selected: execute_handler,

    }
    return execute_action_v1(memory, request, ACTION_CONFIG_PATH, bot_config(), actions)


def is_pay_with_points_selected(param):
    return cdms(param.memory)['membership_rewards_options'] == "Pay with Points"


def is_refund_points_selected(param):
    return cdms(param.memory)['membership_rewards_options'] == "Refund Points"


def is_pay_with_points_charges_selected(param):
    return cdms(param.memory)['membership_rewards_options'] == "Pay with Points Charges"


def is_minimum_redeem_points_selected(param):
    return cdms(param.memory)['membership_rewards_options'] == "Minimum redeem Points"


def is_something_else_selected(param):
    return cdms(param.memory)['membership_rewards_options'] == "Something else"
```

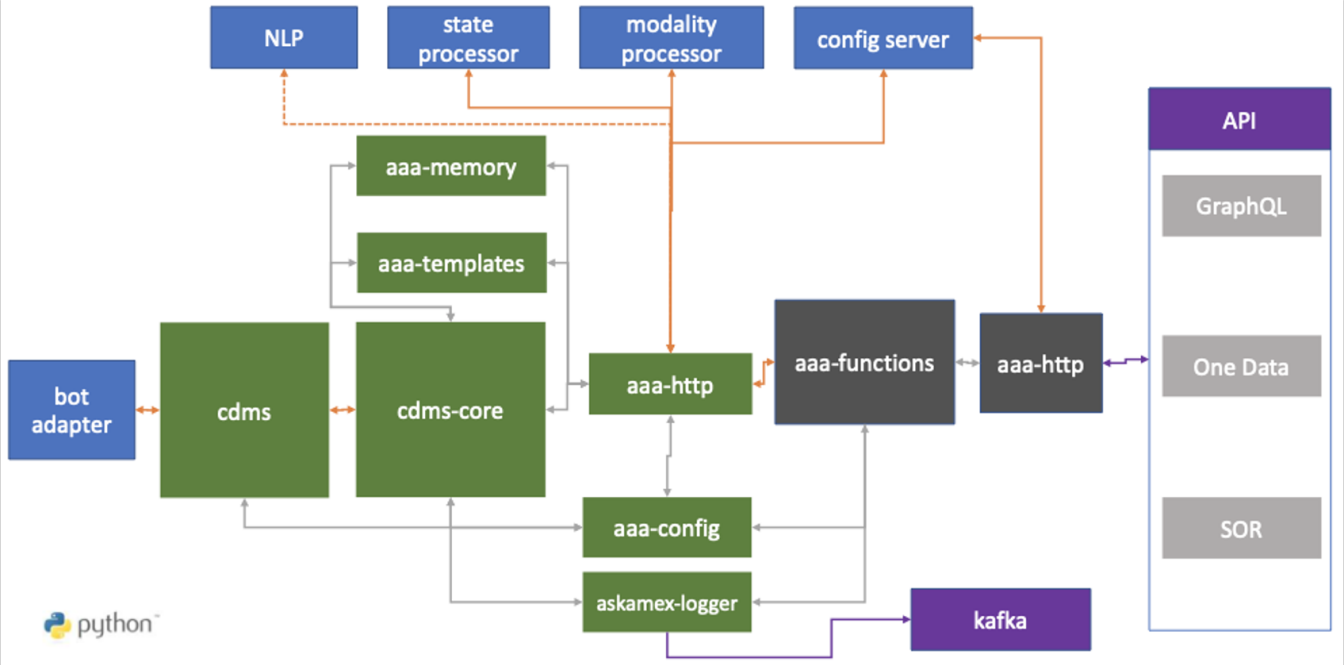
The same logic can be applied to the corresponding .yaml file as well. For the option *Something Else* in the flow, we see that when the user selects it, they should be escalated to a CCP (Customer Care Professional). In order to specify this, we still apply the **relinquish_control** operation and action, but provide the reason as **escalation** instead of leaving it empty like the previous relinquish_control operations. This tells the bot to take the user to a CCP.

membership_rewards_options.yaml

```
name: membership_rewards_options
requirements:
  action: membership_rewards_options
  capture_type: list
  in_scope_utterances:
  - Pay with Points
  - Refund Points
  - Pay with Points Charges
  - Minimum redeem Points
  - Something else
events:
- condition: is_pay_with_points_selected
  handler: execute_handler
  commands:
  - operation: send_message
    inputs:
      template_name: pay_with_points_message
  - operation: relinquish_control
    inputs:
      action: relinquish_control
      template: User selected pay with points, travel info flow completed
      reason: ''
- condition: is_refund_points_selected
  handler: execute_handler
  commands:
  - operation: send_message
    inputs:
      template_name: refund_points_message
  - operation: relinquish_control
    inputs:
      action: relinquish_control
      template: User selected refund points, travel info flow completed
      reason: ''
- condition: is_pay_with_points_charges_selected
  handler: execute_handler
  commands:
  - operation: send_message
    inputs:
      template_name: pay_with_points_charges_message
  - operation: relinquish_control
    inputs:
      action: relinquish_control
      template: User selected pay with points charges, travel info flow completed
      reason: ''
- condition: is_minimum_redeem_points_selected
  handler: execute_handler
  commands:
  - operation: send_message
    inputs:
      template_name: minimum_redeem_points_message
  - operation: relinquish_control
    inputs:
      action: relinquish_control
      template: User selected minimum redeem points, travel info flow completed
      reason: ''
- condition: is_something_else_selected
  handler: execute_handler
  commands:
  - operation: send_message
    inputs:
      template_name: escalate_message
  - operation: relinquish_control
    inputs:
      action: relinquish_control
      template: User selected something else in membership rewards
      reason: 'escalation'
```

CDMS Framework

CDMS ARCHITECTURE



File Structure

```
askamex-cdms
 bots
    my_bot
       actions
       constants
       entities
       functions
       sub_intents
       util
```

ACTIONS

This 'Actions' folder will contain (.py files) with a corresponding YAML file. Python file have function definitions of what to perform on the request data.YAML file is a configuration file which consists of configurations for entities,requirements,events (operations that need to executed if the condition is satisfied).**capture_type** is assigned either list, function or intent based on the requirement that needs to be captured.

First function that to be executed in action folder is **capture_requirements.**

capture requirements -There are 3 types of capturing requirements.

- Intent capture
- List capture
- Function capture

capture_requirements function in .py file

```
def capture_requirements(memory, request):
    capture_config_based_requirements(memory, request, ACTION_CONFIG_PATH)
```

This is the way how **requirements** are configured in YAML file

Intent capture-Suppose the user utterance is "I want to get a loan".So the intent to this utterance maybe "get_loan".So based on this intent the Mainbot gives the control to loan boat.This capture requirements function checks whether get_loan intent is in scope of loan bot or not.Under **in_scope_intents** section, we mention all the intents the bot satisfies.This function verifies whether the intent driven from the user response is in scope of intents configured to Microbot. It also compares the confidence level and threshold of the intent and executes the next action if and only confidence is greater than or equal to threshold.If the user utterance is "I want to cancel a loan". Intent to this utterance is "cancel_loan".So it will be mapped to loan bot.But the intents that are configured to loan are "get_loan","extend_loan".cancel loan is not in scope of loan bot intents.So loanbot tells mainbot that this intent is not in my scope I can't handle this request.

If the intent that is captured is in scope of the bot and confidence is greater than theshold,then the bot memory is updated with intent_name, confidence_score.The captured_intent will be assigned to "**intent_name"** variable in memory.**Example:-cdms(memory)['intent_name'] ='card_declined'**

List capture-when the user is provided with a list of options to choose from, it checks whether the option selected by the user in scope of defined options.

If the option that is selected is in scope utterances of the bot ,then the bot memory is updated with user selected utterance,The selected option will be assigned to "action variable of requirement**"** variable in memory.**Example:-cdms(memory)['intent_received']='card_declined'**

Function capture-This requirement is used when the user is provided with a list of options and these list of options are dynamic.This function checks whether the user has selected from the options that is provided to user.This is a user-defined function and not part of core-cdms.

In the YAML file the **action** variable is the name of the file. The selected variable is stored is assigned to action varaible in memory.Example-user sends a request with message as blue card and action is card_selector in YAML file,then card_selector is assigned with value as "blue card".**cdms(memory)['card_selector']="Blue card".**

Second function that executes in the action folder is **capture_entities**.

capture_entities-

Entities are the relevant information that we get from user utterance and the task that required to be performed on object such as card type.This function ensures that memory variable of the microbot is updated with the entities we get from the user utterance.

In the entities section, we have different variable such as state, type and post processor.

state:-As we need to update the bot memory with entities, the entities are assigned to state variable in the bot memory.For example, cdms(memory)['card_selector'] =entity_values.

type:-Type variable indicates entity type we need to extract from the user utterance of the request data.type consists of name and threshold.name indicates the variable name of the entity that the bot needs to be captured.threshold is the minimum level that the needs to be satisfied.

overrride-this is boolean variable.It decides whehther to update the entity_state of memory variable or not.

```
requirements:
  action: intent_received
  capture_type: intent
  in_scope_intents:
    intents:
    - card_declined
    threshold: 0.2
```

capture_entities function in .py file

```
def capture_entities(memory, request):
    extract_entities(memory, request, ACTION_CONFIG_PATH)
```

This is the way how **entities** are configured in YAML file

```
entities:
- state: card_selector
  override: true
  types:
  - name: CARD_TYPE
    threshold: 0.3
  post_processor: card_type_entity_processor
```

execute function in .py file

```
async def execute(memory, request):
    print("INTENT RECEIVED")
    actions = {
        is_single_card: execute_handler,
        is_multi_card_with_card_selected: execute_handler,
        is_multi_card_with_no_card_selected: handle_is_multi_card_with_no_card_selected
    }
    return await execute_action_v1(memory, request, ACTION_CONFIG_PATH, bot_config(), actions)
```

This is the way how **events** are configured in YAML file

post processor:-This function checks whether the entities we get from the request are actually matching the user account details.

Next,**execute** function is executed

execute-

This function contains the actions variable in dictionary form with action name as key and handler for that action as value.By default the handler for every action is execute handler.But we can also define custom handlers.

Corresponding YAML contains events which consists of conditions.Condiitons are basically the keys defined in the actions variable of python file.

This function internally calls **execute_action_v1** function. This function calls get_action_config which loads .yaml file and if any sub-intents are present in the request then the execute_intent function is called with sub-intents.If sub-intents are not present in the user request,then execute_intent function with empty sub-intents variable.

execute_intent:-

This functions executes the handler based on the condition satisfied.

handler

The execute handler executes all the operations that are mentioned in the events category of the condition satisfied in .yaml file.

We can also define our own custom handlers.custom handlers are defined in case where the user should be provided with dynamic list of options.Then these options are sent as attachment to execute handler.this execute handler will be called in custom handler with some variables modified.

Events:-Events is the list of actions that can be performed in the bot flow.Events consists **conditions**.

Condition-each condition is action that can be performed in the bot flow.Each condiiton consists of handler and commands that need to per performed if the condition is satisfied.

handler-With the help of handler we can execute the commands that are mentioned.

commands-Commands are set of operations.

Here are the list of operations that can be performed when the condition is true:-

Next_action:-Here we update the memory variable with the next action. This action requires inputs which is the action name that is to be executed next when the user made a request.

execute_action:-This operation takes the action name as input and calls the execute function of the action that is given as input

update_memory:-update_memory will update the actual database so the point of the flow the bot is in doesn't remain local

send_message:-This operation sends a message to the user. There are various message_types that can be we can send to user:

```yaml
events:
- condition: is_single_card
  handler: execute_handler
  commands:
  - operation: execute_action
    inputs:
      action: get_transactions
- condition: is_multi_card_with_card_selected
  handler: execute_handler
  commands:
  - operation: execute_action
    inputs:
      action: get_transactions
- condition: is_multi_card_with_no_card_selected
  handler: handle_is_multi_card_with_no_card_selected
  commands:
  - operation: next_action
    inputs:
      action: card_selector
  - operation: update_memory
  - operation: send_message
    inputs:
      template: 'Which Card are you asking about:'
      message_type: single_choice
```

This is **execute** function of card_type_entity_processor.py file cd_bot

```python
def execute(memory, request):
    if cdms(memory).get('card_selector'):
        accounts = ast.literal_eval(cdms(memory).get('accounts'))
        card_type_entity = cdms(memory).get('card_selector').split(' ')[0].strip()
        matching_card = [account['account_number_display_text']
                         for account in accounts
                         if card_type_entity.lower() in account['account_number_display_text'].lower()]
        cdms(memory)['card_selector'] = matching_card[0] if len(matching_card) == 1 else None
        if cdms(memory)['card_selector']:
            cdms(memory)['selected_card_account'] = str(selected_card_account(accounts, cdms(memory)['card_selector']))
```

constants class

```python
class Constants:
    CONTENT_TYPE_KEY = 'Content-Type'
    CONTENT_TYPE_VALUE = 'application/json'
    CORRELATION_ID_KEY = 'X-AXP-CorrelationId'
    AUTHORIZATION_KEY = 'Authorization'
    USER_AUTHORIZATION = 'X-AXP-User-Authorization'
    APP_NAME = 'askamex_cdms'
    ACTION_START = 'action-start'
    ACTION_END = 'action-end'
    LOG_SUCCESS_REASON = 'SUCCESS'
    LOG_SUCCESS_RESULT = 'success'
    LOG_FAILURE_RESULT = 'failure'
```

bot_config function of bot_util.py file

```python
def bot_config():
    return {
        'notify.sleep.seconds': config.get('askamex.cdms.card.declined.notify.sleep.seconds'),
        'notify.retires': config.get('askamex.cdms.card.declined.notify.retries'),
        'master.botname': config.get('askamex.cdms.card.declined.master.botname'),
        'botname': config.get('askamex.cdms.card.declined.botname'),
        'destination.botname': config.get('askamex.cdms.card.declined.destination.botname'),
        'message_id.prefix': config.get('askamex.cdms.card.declined.message_id.prefix'),
        'bot.user_id': config.get('askamex.cdms.card.declined.bot.user_id'),
        'journey': 'card_declined'
    }
```

text: use when you don't want to prompt the user for further inputs

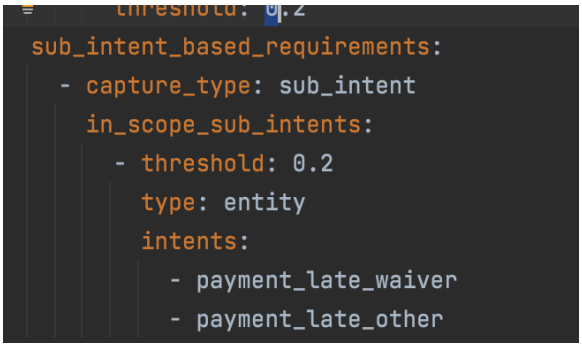
single_choice: This appears when the user doesn't have free inputs (they must pick from one in a list)

richtext: This appears when the output includes a deeplink like a hyperlink or a phone number

Based on the message we have sent to user,user reacts to it based on the goal he want to achieve.The control goes to the next action that is specified in the cdms(memory) and executes the condiitonal event that is satisfied based on the user input.

Relinquish_control:- This gives the control to customer care professional.

delegate_control-This operation is performed withinthe bot when the bot consists of sub intents and when the intent captured is sub-intent.To pass control from the bot to sub-intent module,this operation is used.



ENTITIES

In the above section,we have seen how entities are captures and updated in the memory.

In this entities folders contains a file for each captured entity and validates whether the entity that is captured is associated with the user or not.

Suppose if we have extracted card_type as blue card from the user utterance.Then this function verifies whether user consists of a blue card or not.

FUNCTIONS

We use this functions module to make API calls to ask-amex functions.For example,if in the card_declined,we need to know the transactions history.So we make an API call to get_acocunt_transactions API.

Functions are explained clearly here askamex-functions.

CONSTANTS

constants class contains all the constants that can be used many times.We can add our own constants based on the needs of the bot-flow.

UTIL

Util consists of bit_util.py file which consists of bot_config function. bot_config functions defines basic configurations of the bot like mainbotname,bot.user_id and so.

bot_util.py also consists of helper functions that bot may need in the process of bot_flow.

These are the examples of some helper functions present in bot_util.py file of cd_bot.

```
def get_auth_headers(header):
    return header['X-AXP-User-Authorization'].split(" ")[1] if header.get('X-AXP-User-Authorization') else ""


def selected_card_account(member_accounts, selected_card=None):
    return member_accounts[0] if len(member_accounts) == 1 else \
        next((account for account in member_accounts if account['account_number_display_text'] == selected_card), {})


def selected_card_account_token(member_accounts, selected_card):
    return selected_card_account(member_accounts, selected_card).get('account_token_decrypted')
```

SUB-INTENTS

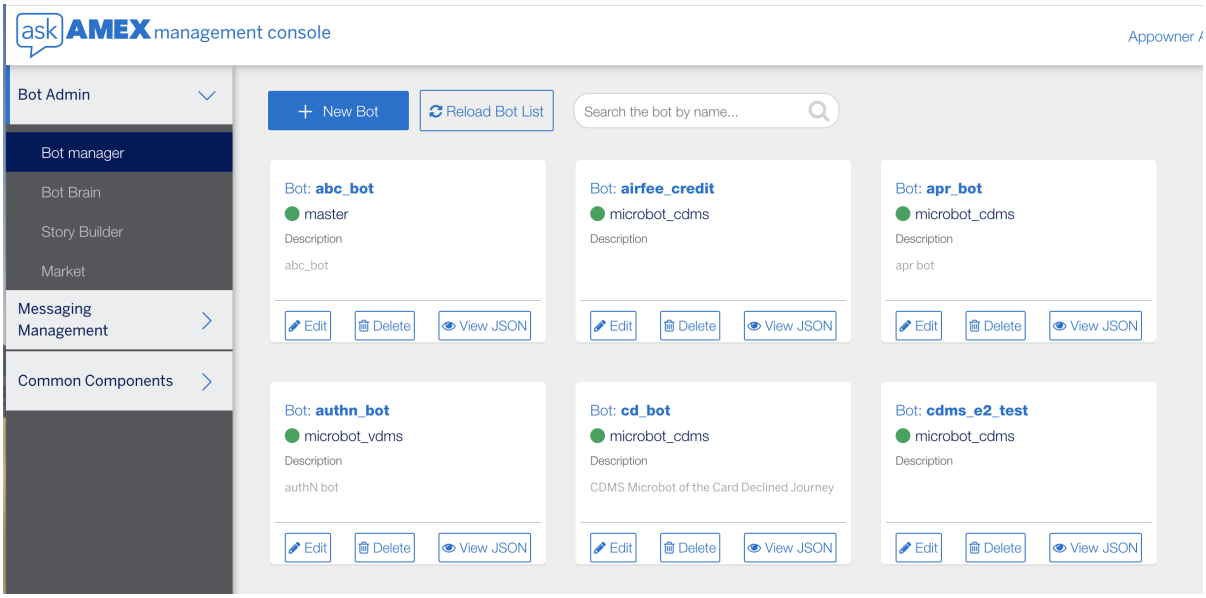
You can see an example of bot consisting of sub-intents in pl (payment late) bot.This is similar to the case of main bot passing control to the individual bots.When user utters aa utterance regarding late payment,then the intent is extracted from this response.It checks whether the intent in the scope of main intents or sub-intents.If the intent is in the scope if sub-intents,the the control is passed to the sub-intent module which consists of file-structure similar to bot like actions,entities,util,functions.

As we saw earlier that the captured intent is saved in the cdms memory as intent_name.Similarly,captured sub-intent is assigned to cdms memory variable as **sub_intent_name**.

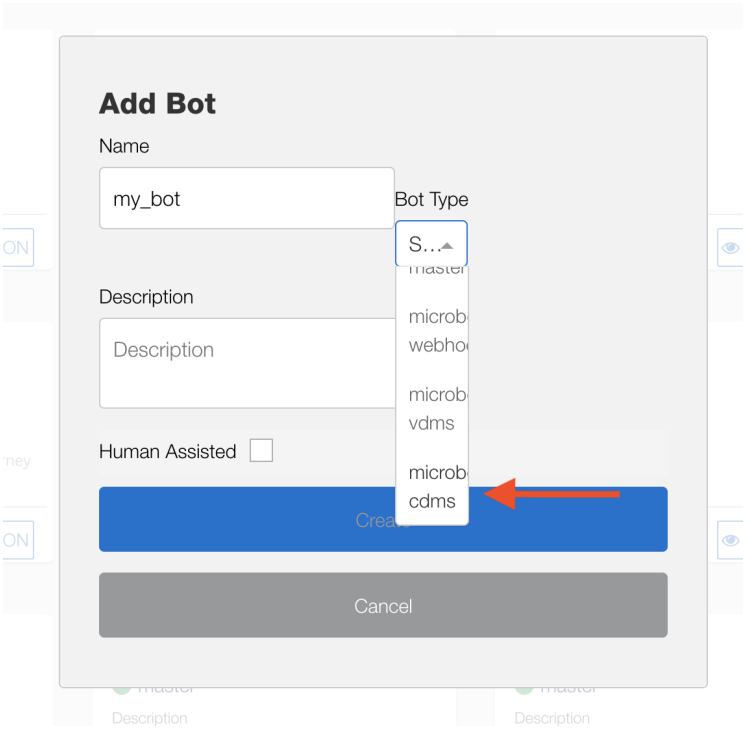
TEMPLATES

Templates are used to format messages that we want to reply with back to the user. Templates are implemented when we use the send_message operation in an event, as mentioned here. Templates can be created/updated in the askAmex bot management services.

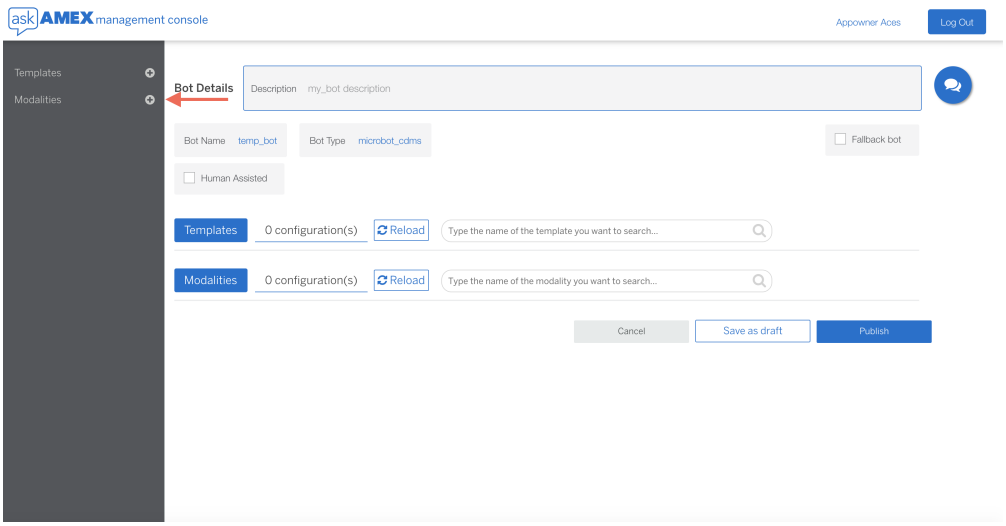
Step 1. Navigate to the **Bot Manager** section under **Bot Admin** and select ' + New Bot ' button.



Step 2. Enter your bot name and description of your bot. Select '**microbot_cdms**' as the bot type and click 'Create'.



Step 3. Create your modalities.



There are three different modalities you need create:

Modalities

Provide the user a list of defined options.

Parameters

KeyName	Value
Name	**listModality**
Channel	**mobile**
Type	**static**

JSON output	{"content":{"desc":"PLACE_HOLDER_TEXT"},"update_time":"2018-05-07T06:55:58.276660","userid":"7d5eb07f-de20-48cb-91ac-97fc9ff284a3","conversationid":"c_b1cb4690-eca3-421c-a662-f763b4da5a65","create_time":"2018-05-07T06:55:24.054300","attachment":{"question_id":"1234","options":[{"option_id":"1","option_text":"Yes"}]},"messageid":"m_messageid1","type":"single_choice"}

Attributes: Add in the following two attributes

KeyName	Value
Name	**desc_text**
Access	**content.desc**
Mandatory	**false**

KeyName	Value
Name	**options_list_list**
Access	**attachment.options**
Mandatory	**false**

Output a message that includes a deeplink, hyperlink, or phone number

Parameters

KeyName	Value
Name	**richTextModality**
Channel	**mobile**
Type	**static**
JSON output	{"content":{"desc":"PLACE_HOLDER_DESC_TEXT"},"update_time":"2019-02-04T19:16:37.804180","userid":"7d5eb07f-de20-48cb-91ac-97fc9ff284a3","conversationid":"c_b1cb4690-eca3-421c-a662-f763b4da5a65","create_time":"2019-02-04T19:16:37.804150","messageid":"m_ab347f83-71cf-4cce-9e46-74c8e9eee523","type":"richtext","attachment":{"text_keys":{"KEY_1":{"text":"PLACE_HOLDER_HYPERLINK_TEXT","text_type":"hyperlink","uri":"PLACE_HOLDER_HYPERLINK_URI"},"KEY_2":{"text":"PLACE_HOLDER_PHONE_NO_TEXT","text_type":"phone_number","uri":"PLACE_HOLDER_PHONE_NO_URI"}}}}

Attributes: Add in the following two attributes

KeyName	Value
Name	**options_list_hashmap**
Access	**attachment.text_keys**
Mandatory	**false**

KeyName	Value
Name	**planText_1_text**
Access	**content.desc**
Mandatory	**false**

Output a message to the user

Parameters

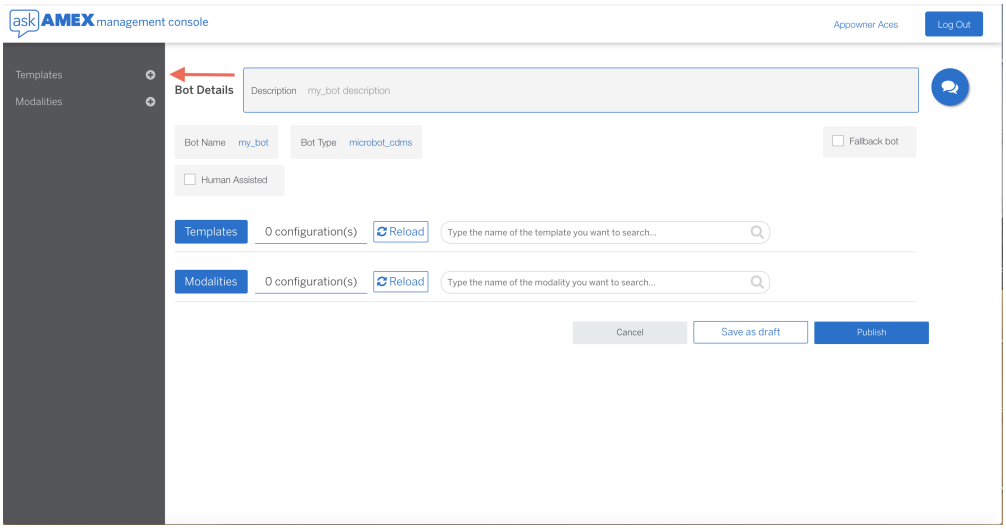
KeyName	Value
Name	**textModality**
Channel	**mobile**
Type	**static**

JSON output	{"content":{"desc":"PLACE_HOLDER_TEXT"},"update_time":"2018-05-07T06:55:58.276660","userid":"7d5eb07f-de20-48cb-91ac-97fc9ff284a3","conversationid":"c_b1cb4690-eca3-421c-a662-f763b4da5a65","create_time":"2018-05-07T06:55:24.054300","messageid":"m_messageid1","type":"text"}

Attributes: Add in the attribute

KeyName	Value
Name	**planText_1_text**
Access	**content.desc**
Mandatory	**false**

Step 4. Create your templates. The template set-up depends on the type of modality you use.

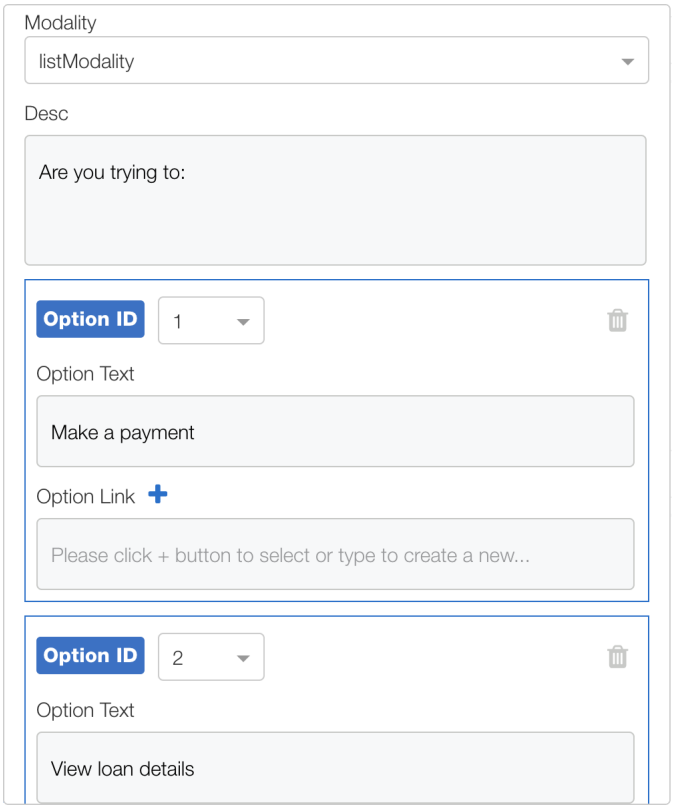


Templates

KeyName	Value
Name	***Your template name. Needs to match 'template_name' referenced in JSON file.***
Channel	**mobile**
Locale	**en-US**
Modality	***Select from modalities you created***

Depending on the modality selected, you will need to provide in the appropriate info:

Modality	Info

listModality	Desc: *Text to display before options list* Option Text: *Text of option* **Continue to add options as needed* 
richTextModality	Text: *Text to display to user, can include deeplinks, hyperlinks, phone numbers.* Text Type: *Select text type and add in text and URI*

Modality

richTextModality

Text

There's a dedicated Personal Loans team to assist with this. You can reach them at:
{{KEY_0}}

Text Type phone_number KEY_0

Text

1-844-273-1384

Uri

tel://8442731384

+ Add New Option

Modality

richTextModality

Text

While you're not currently eligible to apply for an American Express® Personal Loan, you can check out some of our other lending products, features, and services.

Text Type hyperlink KEY_0

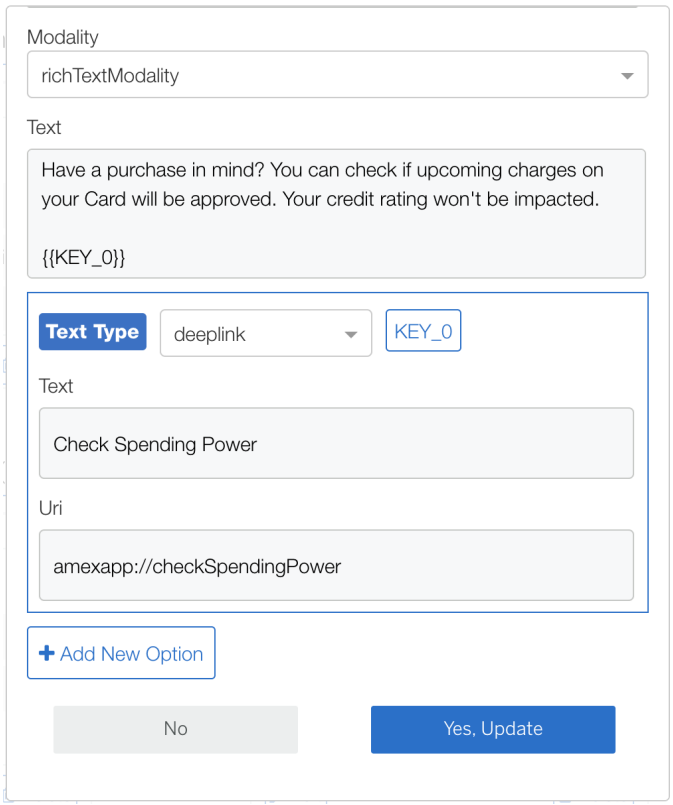
Text

View Options

Uri

https://global.americanexpress.com/lending/lending-options

+ Add New Option

<table>
<tr><td></td><td> 
</td></tr>
<tr><td>textModality</td><td>Response Text: Text to output to user</td></tr>
</table>

Step 5. Publish your templates!

Usage

Now that we have the actual templates set up, we need to link it to the CDMS. In the action JSON files with a '**send_message**' operation, you'll need to configure the '**inputs**' based on the type of message and template. Below you can find examples of how each modality is used.

Single Choice, List Modality

```
...
{
  "condition": "is_manage_your_loan",
  "handler": "execute_handler",
  "commands": [
    {
      "operation": "next_action", <-- Ensure that the next request takes us to next action, which is to
select from options given in the template
      "inputs": {
        "action": "manage_loan_options" <-- Next action to be taken which allows the user to pick from
options listed
      }
    },
    {
      "operation": "update_memory"
    },
    {
      "operation": "send_message",
      "inputs": {
        "template_name": "manage_your_loan", <-- template name in the askAmex bot management service
        "message_type": "single_choice" <-- message_type is "single_choice" because the user is picking from
a list
      }
    }
  ]
}
...
```

richText Modality

```
...
"commands": [
  {
    "operation": "send_message",
    "inputs": {
      "template_name": "not_pre_approved",
      "message_type": "richtext",
      "template_type": "dynamic"
    }
  },
...
```

text Modality

```
...
"commands": [
  {
    "operation": "send_message",
    "inputs": {
      "template_name": "card_not_authorized",
      "message_type": "text"
    }
  },
...
```

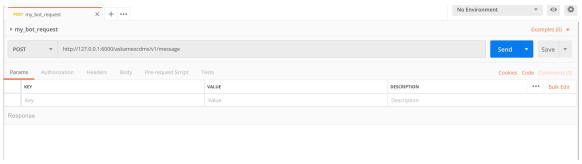
TESTING

GET Memory

POST REQUEST

To test our bot, we can send **POST** requests from Postman to see if the dialogue is branching properly.

Step 1. Open Postman and create a new **POST** request by changing the request type. Set the request URL to: http://127.0.0.1:6000/askamexcdms/v1/message



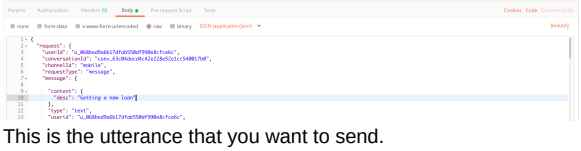
Step 2. Enter the following headers in Headers section.

Key	Value
Content-Type	application/json
bbv	{{bbv}}
gatekeeper	{{gatekeeper}}
accountToken	{{accountToken}}
X-AXP-User-Legacy-Authorization	blueboxvalues={{bbv}};gatekeeper={{gatekeeper}}
X-AXP-CorrelationId	12212122
X-AXP-Locale	en-US
Authorization	AppId 661ef271-3037-4ae8-a3f0-916fa4f19b8d
X-AXP-User-Authorization	AppId 661ef271-3037-4ae8-a3f0-916fa4f19b8d

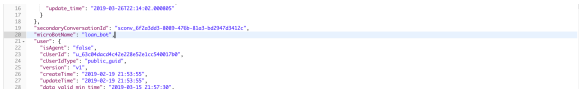
Step 3. Add **raw JSON(application/json)** body in Body section. To save the trouble of having to add in a JSON file from scratch, here is a pre-written one to copy over: JSON Body file.

This JSON file may not have all the metadata that you need for your bot (such as multiple cards). Please reference other JSON files in the Postman Collection or add in the required fields.

Before proceeding, you will need to change some of the parameters so that it is relevant to your bot.



This is the utterance that you want to send.



Change to the name of your microbot.

To retrieve the memory of the bot, you can send a **GET** request from Postman. This is helpful if you want to see what attributes have been saved to memory.

Step 1. Create a new **GET** request in Postman. Set the request URL to: https://stateprocessl-qa.aexp.com/v1/state/sessioncontexts

Step 2. Add in the following parameters under the 'Params' tab, these params will append to the request URL.

Key	Value
botName	*your bot name, found under 'microBotName' of POST request*
cUserId	*your channel userId, found in 'userId' of POST request*
channelId	**mobile**
version	**v1**
transId	**sadasddassa**

Step 3. Add in headers

Key	Value
Content-Type	**application/json**
Accept	**application/json**
lockSessionTime	**30**
transId	**12212122**
lockSession	**true**

Step 4. Add in **raw JSON (applicaiton/json)** Body. It will be formatted as the following:

```
{
     "botName": "your_bot_name",
     "cUserId": "your cUserId",
     "channelId": "mobile",
     "version": "v1"
}
```

Step 5. Send **GET** request. In the Postman console you will see the bot memory. If you get an error, then the bot memory was most likely deleted.

You can see the conversational flow of the bot in the conversationstringcontext of bot memory.You can observe varaibles that you have stored in memory like card_selector,selected_transaction, accounts,next_action and previous_messages.

This is the conversationstringcontext of response body for card_declined bot

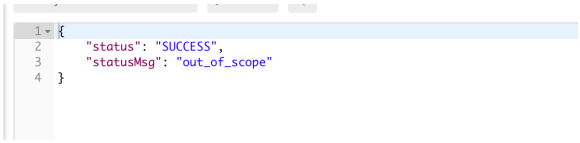
Change to the relevant intents of your bot.Here if you want to send addiitonal details like entities,then you can add those details under concepts variable with the entity_type and the entity value as parameter. If you oberve above you can see that under concepts card_type (entity_type) and blue as entity value.

Step 4. Send the **POST** request. Once the request is sent, you will get one of a few HTTP codes:

200

A correct successful request with have a similar response schema to the following:

You may get a successful request but with a status message 'out_of_scope'. This means that the utterance you entered was not within the set of acceptable inputs detailed within the requirements capture. Check the console logs in Pycharm to traceback.



```
{
    "status": "SUCCESS",
    "statusMsg": "out_of_scope"
}
```

401 Unauthorized

This means something is wrong with the headers in Postman, check the error in the Postman console.

500

This usually means that there's an error within the CDMS code or in the Postman JSON body, check the error in the Pycharm console.

Suppose for card_Declined bot.The first request we send is "card declined"

```
"message": {
    "content": {
        "desc": "Mycard declined"
    },
    "type": "text",
```

This is the response body,when we make get request to the bot memory.

Now,as you can see in the above response body,the user is being asked to choose one of the transactions.Now we send the transaction_display_text as message content in the next request.Below image is the request body with message as the next selected transaction.

DELETE Bot Context

You will need to delete the bot context if you are wanting to 'start over', or return to the beginning of the flow by clearing out the bot memory. If you don't, any **POST** requests you send will be sent to the last 'point' of the flow it was left at. This can cause utterances not being recognized correctly, for example.

Step 1. Create a new **DELETE** request from Postman. Set the request URL to: https://stateprocess-qa.aexp.com/v1/state/sessioncontexts

Step 2. Add in the following parameters under the 'Params' tab, these params will append to the request URL.

Key	Value
botName	*your bot name, found under 'microBotName' of POST request*
cUserId	*your channel userId, found in 'userId' of POST request*
channelId	**mobile**
version	**v1**

Step 3. Add in the following headers.

Key	Value
Content-Type	**application/json**
Accept	**application/json**
transId	**12212122**
lockValue	**1553019728042852352**

As the transaction is selected,the request is escalated to ccp. All the bot_memory is updated with empty conversationstringcontext.Bot flow starts from the beginning.