

Updated - How to Build Your Own CDMS Microbot



This page is a Work in Progress and open to editing. If you see any changes/updates to be made, please feel free to edit or reach out to Arianne Malagayo via Slack.



Hello! Looks like you're wanting to build your own CDMS MicroBot.

Look no further than this user guide to help you understand the framework to get you on your way to making your own bot from beginning to end.

Though this guide may not explore every folder and file of the CDMS framework, it will explain the most relevant parts needed to help you through the bot-building process.

Index

- 1 [About](#)
- 2 [Environment \(E0\) Setup](#)
- 3 [CDMS Framework](#)
 - 3.1 [How does a Microbot Work?](#)
 - 3.2 [CDMS Core](#)
 - 3.3 [File Structure](#)
 - 3.3.1 [Actions](#)
 - 3.3.2 [Constants](#)
 - 3.3.3 [Entities](#)
 - 3.3.4 [Functions](#)
 - 3.3.5 [Services - DEPRECATED](#)
 - 3.3.6 [Sub-Intents](#)
 - 3.3.6.1 [sub_intents module](#)
 - 3.3.7 [Util](#)
 - 3.4 [Templates](#)
 - 3.4.1 [Adding Dynamic Variables](#)
- 4 [Requests Using Postman](#)
 - 4.1 [GET Memory](#)
 - 4.2 [DELETE Bot Context](#)

About

The AskAmex Code Dialog Management System (**CDMS**) is a code-based approach to creating and maintaining bot responses for Card Member Automation journeys. Driven by card member use cases, the 'flow' of conversation for each automated journey varies from bot to bot. Each **microbot**, directed to by the Main Bot, within CDMS handles a specific journey - whether it be getting a loan, reporting fraud, getting a card replaced, and more - in order to best fulfill the card member's inquiries and requests. Microbots execute **actions**, such as providing links, requesting more information, or asking questions in order to progress the conversation forward and complete the journey.

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.

Environment (E0) Setup



E0 Repository Setup has been updated for the python image change made in February 2022.



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

Pre-Requisites:

Download Python - [Python: Getting Started](#)

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

1. Create file path: \$HOME/.config/pip/pip.conf
2. Terminals offers a built-in text-editor called vim that can be used to copy and paste code in files without leaving the Terminal
 - Type: "vim pip.conf" (From \$HOME/.config/pip) to open the vim pip.conf file you have already created.
 - press 'i' and hit 'enter' in insert text.
 - copy and paste code block from step 3 into vim text editor; then at the last line, and last character of that code block type ":" which will take your cursor to the bottom of the page.
 - Types ":wq" or ":wq!" To save and close vim. This step should take you back to the terminal. Insert contents into pip.conf file:
3. Insert contents into pip.conf file:

pip.conf

```
[global]
index = https://artifactory.aexp.com/api/pypi/pypi/simple
index-url = https://artifactory.aexp.com/api/pypi/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= XXXXXXXXXXXXXXXXXXXX
aaa.functions.get.account.transaction.sor.v2.cas.api.id=XXXXXXXXXXXXXXXXXXXX
aaa.functions.cbis.get.a2a.token.api.client.id=XXXXXXXXXXXXXXXXXXXX
aaa.functions.cbis.get.a2a.token.api.client.secret=XXXXXXXXXXXXXXXXXXXX
askamex.cdms.nlp.api.key=XXXXXXXXXXXXXXXXXXXX
```

Step 3. Create the following directories and files locally:

- /opt/epaas/certs/dkey.key
- /opt/epaas/certs/ca-chain.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

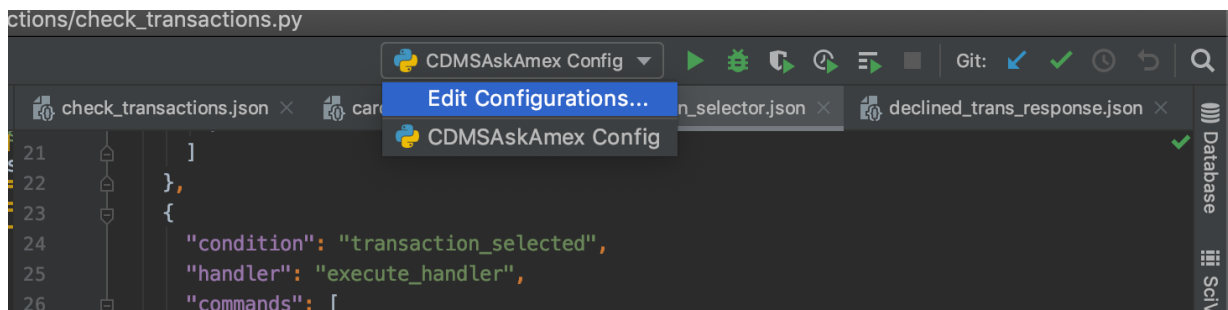
Step 5. We'll need to make a copy of the pem and key files to a local folder under /opt:

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

# Add the pem and key files sent to you from a team member to the above .openshift folder
```

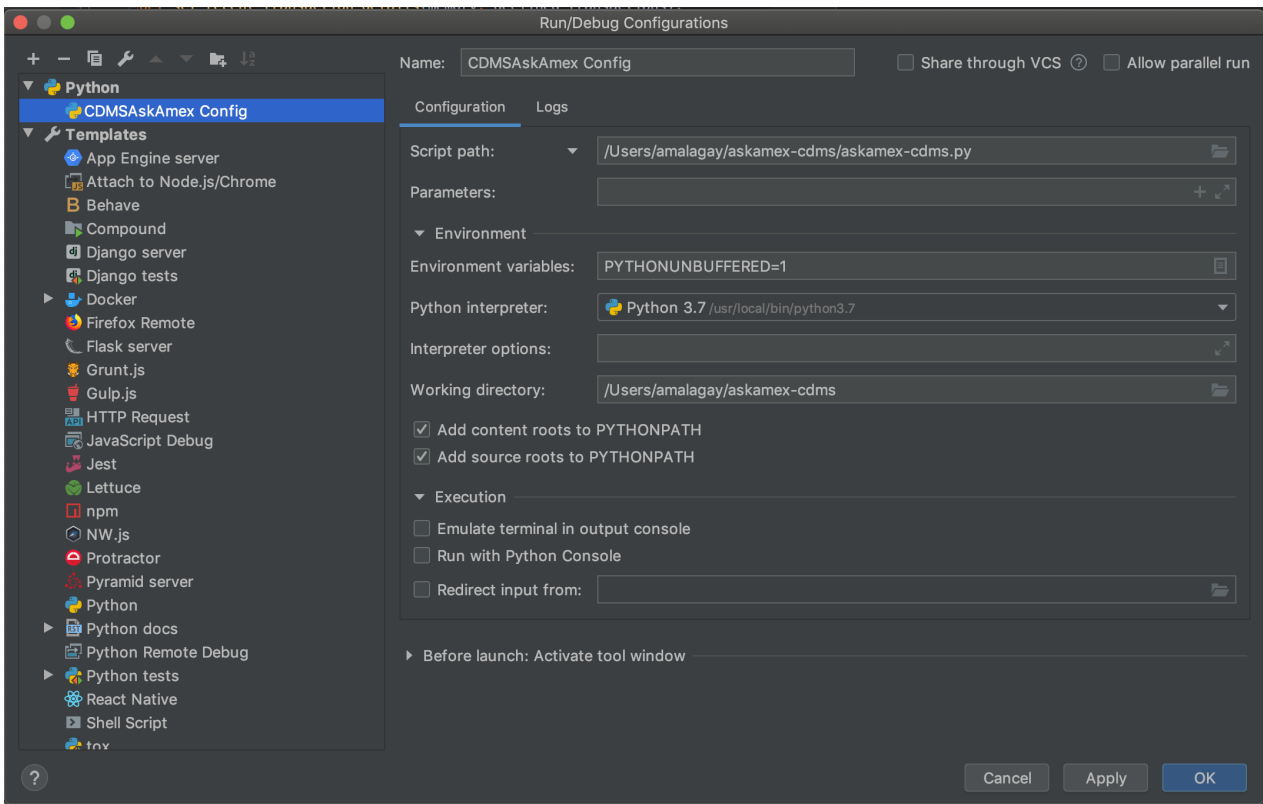
Step 6. Open PyCham to set up environment.

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



Select Python Template

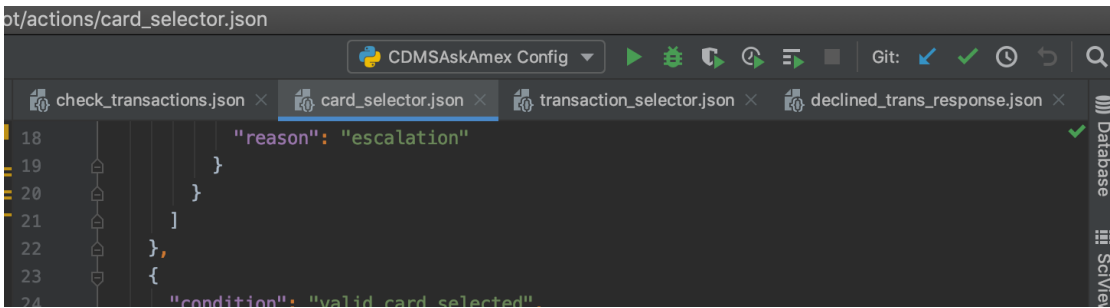
- Set *Script path* to location of **askamex-cdms.py** in your local **askamex-cdms** repo.
- Check that Python Interpreter is set to **Python >3.7** (in this case Python 3.8).
- Set *Working directory* to location of **askamex-cdms** repo.
- Click 'Apply' and 'Ok' when done



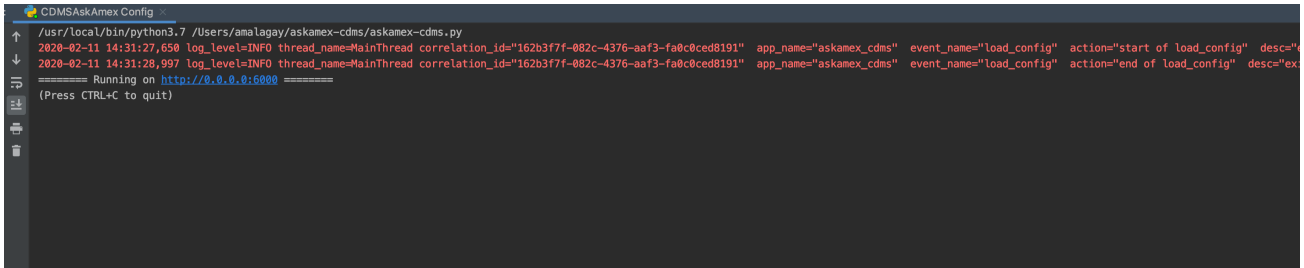
Navigate to **askamex-cdms** repo and install packages.

```
pip install -r .openshift/packages.txt
```

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!

CDMS Framework

How does a Microbot Work?

So, how exactly does a microbot work? When a user types in something into our chatbot such as 'What is my account balance?' or 'I want to replace my card', what they typed in is an **utterance**. This utterance is then mapped to an **intent**, a user's requested purpose. An intent is driven by the combined functions of the NLP (natural language processing) and the NLU (natural language understanding). For example, the utterance 'What is my account balance?' will be mapped to the intent 'accountBalance'. Once the intent is found, the Main Bot gives control over to the appropriate microbot to handle the user's request from there.

Microbots operate using **memory** and **actions**. The microbot **memory** keeps track of anything said or done by the user, such as selecting an answer from a multiple-choice dialog. The memory can be updated as the dialog flow continues in order to take the appropriate dialog path. **Actions** are what drive the microbot from one point of the conversation to the next point of dialog. Along the journey, conditional statements will dictate the actions that are taken. Therefore, there can be different actions taken that branch off into different dialog paths and responses.

At the end of the microbot journey, the microbot relinquishes control, either to another bot, a Customer Care Professional (CCP), or back to the Main Bot.

CDMS Core

The CDMS Core is the site-package that is the CDMS framework. In the CDMS core, you'll find several packages for handlers, notifiers, processors, routes, services, and more. Each package in the core is responsible for different processes that occur during the entire execution of the bot journey.

For example, the util package is responsible for defining functions used to execute actions, get the next action to take, and several other getters and setters necessary to parse the requests sent to the microbot.

Throughout the development of your microbot, you will be directly calling functions found in the CDMS core.

The script we set in our configuration, **askamex-cdms.py**, is the entry point of the program. The most relevant part of this file is the **setup_routes (application)** function that has lines relevant to obtain the proper authorization headers from the request. This is relevant when we get to using Postman to send requests.

.../cdms/routes/routes.py

```
def setup_routes(app):
    app.router.add_post('/askamexcdms/v1/message', on_message)
    app.router.add_post('/askamexcdms/v1/request_control', request_control)
    ...
```

The most relevant parameters that we'll be frequently dealing with are: **memory** and **requests**

memory: Certain inputs the user makes need to be saved so the bot has the requisite information (i.e. context) to assist them properly. The memory is a **dynamic dictionary**, which means that throughout the bot's flow, attributes will be added 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.

- To set an attribute to memory:

```
cdms(memory)['your_attribute_name'] = 'attribute'
```

- To get an attribute from memory:

```
cdms(memory).get('your_attribute_name')
```

requests: Requests contain the user's information. We can parse the request for various pieces of information that we need, like the user's name, address, eligibility, amongst many other things. Remember, even though the request is sent in JSON format, the **request itself is NOT a JSON object**. This means that if you want to parse the request without first converting it to a json (this is what request.json() does), you will run into errors. Don't worry, request handling is taken care of for us in the cdms handlers.

File Structure

Each bot consists of different python packages in order to import multiple modules.

It is important to note that the python packages needed will depend on the requirements of the bot flow you wish to implement, therefore not all of the outlined packages will be required.

```
askamex-cdms
.s2i
bots
  my_bot
    actions
    constants
    entities
    functions
    services - DEPRECATED
    sub_intents
    util
```

As we go through each package, we'll explore the structure and functions of relevant files that need to be created.



If you want to navigate to the definition of a function, use Command+Click

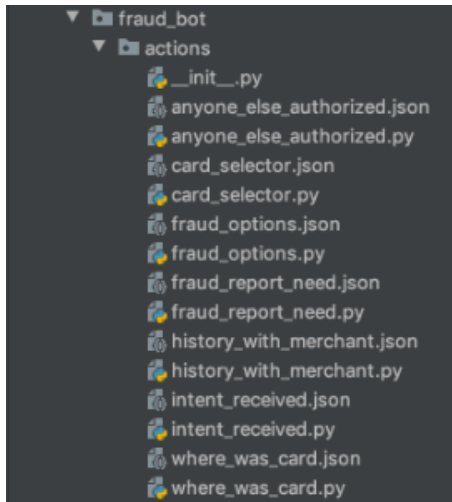
Actions

Actions are what drives the bot flow from one point of a dialogue to the next. The 'Actions' package will contain many modules (.py files) with a corresponding JSON file. Each of these module/JSON file pairs represents a certain action taken based on the conditions found in the bot memory.

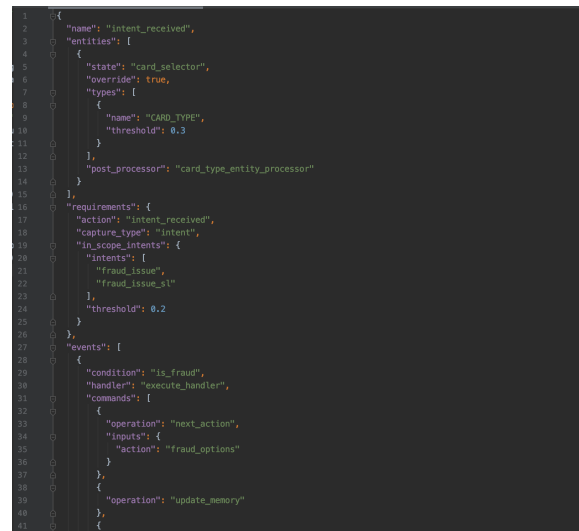
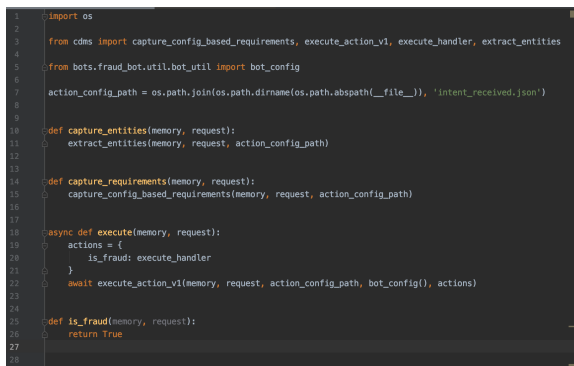


Update from JSON to YAML

For newer bot development, please use .yaml files instead of .json files. The YAML files follow the same structure and purpose as .json files, but are just formatted differently for readability.



The entry point into any action of a bot is the **intent_received.py** and **int_received.json** files.



Let's use the fraud bot as an example:

- **action_config_path** is the file path to the respective JSON file of the action.
- **capture_entities(memory, request):** Captures the entities from an utterance. Entities are parts of an utterance that provides the bot more context as to what the user is saying. Capturing entities helps guide the bot flow when it starts.
 - For example, the user utterance 'My Blue card has an unauthorized charge' has 'Blue' as a captured entity.



extract_entities is an imported function from the CDMS library that processes the 'entities' section of the JSON file.

Looking at the **intent_received.json** file, there is an 'entities' object that contains data defined for the object. In this example, we are wanting to capture a card entity.



- **'state'**: the name of the attribute that the entity will be linked to in memory
- **'types'**: the type of entity wanting to be capture (i.e. CARD_TYPE). The threshold defines confidence score that the entity is actually of the type that we want
- **'post_processor'**: References a helper function found in the 'entities' module. Post processors usually aid in finding relevant pieces info and setting it in the memory.

More details on entity post processor usage can be found in the Entities section [here](#).

- **capture_requirements(memory, request)**: Capturing requirements ensures that the user utterance that brought us to a particular point in a flow is within the scope of the proper inputs for it. Think of requirements as attributes that are needed in order to reach another question or point of a conversation.

There are three different types of requirements that can be captured:

1. **intent**: determines if the intent found from the utterance is within the 'scope' of the bot
2. **list**: captures utterance that user selects from a list of dialogues presented
 - Used to see if utterance selected is in the scope of the defined utterances
3. **function**: used to execute a user-defined function to capture a relevant attribute.
 - Works 'behind the scenes', outside of any direct user request/utterance



Similar to `extract_entities`, `capture_config_based_requirements` is found in the CDMS library and processes the 'requirements' section of the JSON file.

Different requirement captures can be combined together by defining it as a `non_topic_based_requirements`.

```

9 def capture_config_based_requirements(memory, request, action_config_path):
10     requirements = get_requirements(action_config_path)
11     if requirements['capture_type'] == 'intent':
12         intent_based_requirement_capture(memory, request,
13                                         requirements['in_scope_intents'],
14                                         float(requirements['in_scope_intents'] / threshold)))
15     elif requirements['capture_type'] == 'list':
16         list_based_requirement_capture(memory, request, requirements['action'], requirements['in_scope_utterances'])
17     elif requirements['capture_type'] == 'function':
18         action = importlib.import_module('bots.actions.{}'.format(request.get('microbotname')),
19                               requirements['action']))
20         in_scope_function = getattr(action, requirements['in_scope_function'])
21         function_based_requirement_capture(memory, request, requirements['action'], in_scope_function)
22     else:
23         raise CdmException('FAILURE', 'Unable to find matching requirement capture criteria')
24

```

```

16 "requirements": {
17     "action": "intent_received",
18     "capture_type": "intent",
19     "in_scope_intents": {
20         "intents": [
21             "fraud_issue",
22             "fraud_issue_sl"
23         ],
24         "threshold": 0.2
25     }
26 },

```

```

1 {
2     "name": "fraud_report_need",
3     "requirements": {
4         "action": "fraud_report_need",
5         "capture_type": "list",
6         "in_scope_utterances": [
7             "Yes",
8             "No"
9         ]
10    },

```

```

1 {
2     "name": "card_selector",
3     "requirements": {
4         "action": "card_selector",
5         "capture_type": "function",
6         "in_scope_function": "get_account_token"
7    },

```

List and intent based requirement capture

Function and intent based requirement capture

```

requirements:
  non_topic_based_requirements:
    - capture_type: list
      in_scope_utterances:
        - 'Added to Card'
        - 'Redeemed'
        - 'Want to add'
        - 'Something else'
    - capture_type: intent
      in_scope_intents:
        intents:
          - offers_promotion
          - offers_promotion_sl
          - offers_refer
          - offers_refer_sl
          - offers_generic
          - offers_generic_sl
      threshold: 0.2

3  "requirements": {
4    "action": "transaction_details",
5    "non_topic_based_requirements": [
6      {
7        "capture_type": "function",
8        "in_scope_function": "get_selected_transaction"
9      },
10     {
11       "capture_type": "intent",
12       "in_scope_intents": {
13         "intents": [
14           "dispute_pending_charge",
15           "dispute_need",
16           "fraud_report_need",
17           "fraud_issue"
18         ],
19         "threshold": 0.3
20       }
21     }
22   ]
23 },

```



Tip: Remember that not every action will need entities or requirements to be captured! This will depend on what that action is trying to accomplish.

- **execute(memory, request):** Executes events based on if a particular condition(s) are met.
 - **actions** : Dictionary that contains a key-value pair of possible conditions of a defined **event** and its **handler**.

Events - In the JSON file, 'events' is a list of dictionaries where each dictionary represents each possible condition with it's respective handler and set of commands.

- **condition** - Defined in the .py file. If this condition is evaluated to be TRUE, it will go through with associated handler and commands.
- **handler** - basically handles what to do with all the commands given to it such as sending a message or executing an action. In the scope of building our own bot, the inner works of the handler can be abstracted.

- **'execute_handler'** is the a pre-defined default handler
- In certain cases, you may need to define your own handler for a particular condition (*Figure 1*). An example of needing a unique handler is when we need to print a list of the user's cards to select

```

10 async def execute(memory, request):
11     actions = {
12         is_fraud: execute_handler
13     }
14     await execute_action_v1(memory, request, action_config_path, bot_config(), actions)
15
16 def is_fraud(memory, request):
17     return True
18
19
20

```

```

1  {
2    "condition": "The user has account with no card selected",
3    "handler": "Print the list of user's cards to select",
4    "commands": [
5      {
6        "operation": "print_cards",
7        "properties": {
8          "action": "print_cards"
9        }
10     },
11     {
12       "operation": "select_card",
13       "properties": {
14         "action": "select_card"
15       }
16     },
17     {
18       "operation": "fraud_report",
19       "properties": {
20         "action": "fraud_report"
21       }
22     },
23     {
24       "operation": "fraud_issue",
25       "properties": {
26         "action": "fraud_issue"
27       }
28     }
29   ]
30 }


```

Figure

1. Example of user-defined handler and its usage in JSON file (above)

from.

- **commands** - List of what operations need to be done if the condition is TRUE
 - **operation** - There are different kinds of operations that can be applied:
 1. **execute_action**: this is when something (say, an eligibility check) needs to be performed before we can proceed. Putting this in the commands list will do that operation immediately. This is not the same as next_action.
 2. **next_action**: This updates `cdms(memory)` [`next_action`] so that on the next request, the bot knows which action .py file to map to
 3. **update_memory**: update_memory will update the actual database so the point of the flow the bot is in doesn't remain local
 4. **send_message**: Outputs something to the user, there's various message_types that can be used here:
 - a. **text**: use when you don't want to prompt the user for further inputs
 - b. **single_choice**: This appears when the user doesn't have free inputs (they must pick from one in a list)
 - c. **richtext**: This appears when the output includes a deeplink like a hyperlink or a phone number

 Templates are where the actual text output is placed, if **template_name** appears, then the execution handler knows to reference a template created with the [AskAmex bot management services](#) (more on this [here](#)).

```
27 "events": [  
28   {  
29     "condition": "is_fraud",  
30     "handler": "execute_handler",  
31     "commands": [  
32       {  
33         "operation": "next_action",  
34         "inputs": {  
35           "action": "fraud_options"  
36         }  
37       },  
38       {  
39         "operation": "update_memory"  
40       },  
41       {  
42         "operation": "send_message",  
43         "inputs": {  
44           "template_name": "learn_more",  
45           "message_type": "richtext"  
46         }  
47       },  
48       {  
49         "operation": "send_message",  
50         "inputs": {  
51           "template_name": "fraud_options",  
52           "message_type": "single_choice"  
53         }  
54       }  
55     ]  
56   },  
57 ]  
58 }
```

- **execute_action_v1** - asynchronous function that takes arguments: memory, request, action_config_path, bot_config(), actions. Will always

be used at the end of every execute function definition.

Constants

Pretty straight-forward package, the Constants class contains all the constants that are needed to be referenced for the bot. You can add your own constants as needed to verify things like a dialog option that a customer selects.

```
1 class Constants:
2     CONTENT_TYPE_KEY = 'Content-Type'
3     CONTENT_TYPE_VALUE = 'application/json'
4     CORRELATION_ID_KEY = 'X-AXP-CorrelationId'
5     AUTHORIZATION_KEY = 'Authorization'
6     APP_NAME = 'askamex_cdms'
7     ACTION_START = 'action-start'
8     ACTION_END = 'action-end'
9     LOG_SUCCESS_REASON = 'SUCCESS'
10    LOG_SUCCESS_RESULT = 'success'
11    LOG_FAILURE_RESULT = 'failure'
12    PERSONAL_LOAN_PRODUCT = 10
13    LOAN_DECLINE_CODES = [5430]
14    LOAN_INTENTS = ['loan', 'loan_sl']
15    CIUMC_INTENTS = ['ciumc', 'ciumc_sl']
16    LOAN_OPTIONS = ['Getting a new loan', 'Pending application', 'Managing your loan']
17
```

For example, Constants.LOAN_OPTIONS is a list of the three possible dialog options for the customer. Usage of this would be to check that the user utterance is in the list of in-scope utterances, and assigns that utterance to a CDMS memory attribute based on the action name.

```
1 {
2     "name": "loan_options",
3     "requirements": {
4         "action": "loan_options",
5         "capture_type": "list",
6         "in_scope_utterances": [
7             "Getting a new loan",
8             "Pending application",
9             "Managing your loan"
10        ]
11    },
12
```

```
24
25 def is_getting_a_new_loan(memory, request):
26     return cdms(memory)['loan_options'] == Constants.LOAN_OPTIONS[0]
27
28
29 def is_pending_application(memory, request):
30     return cdms(memory)['loan_options'] == Constants.LOAN_OPTIONS[1]
31
32
33 def is_manage_your_loan(memory, request):
34     return cdms(memory)['loan_options'] == Constants.LOAN_OPTIONS[2]
```

Entities

As mentioned [here](#), entities are parts of an utterance that provides the bot more context as to what the user is saying. In the 'entities' package, an entity post-processing module is necessary to assign a particular entity to relevant piece of CDMS memory. For each entity you are wanting to capture, you will need to provide it's own post-processor.

In the example below *card_type_entity_processor.py*, the 'selected_card_account' is set by finding if the captured entity, a card type in this case, matches a card type that's found in the user's account.

```
6
7 def execute(memory, request):
8     if cdms(memory).get('card_selector'):
9         accounts = ast.literal_eval(cdm(memory).get('accounts'))
10        card_type_entity = cdms(memory).get('card_selector').split(' ')[0].strip()
11        matching_card = [account['account_number_display_text']
12                        for account in accounts
13                        if card_type_entity.lower() in account['account_number_display_text'].lower()]
14        cdms(memory)['card_selector'] = matching_card[0] if len(matching_card) == 1 else None
15    if cdms(memory)['card_selector']:
16        cdms(memory)['selected_card_account'] = str(selected_card_account(accounts, cdms(memory)['card_selector']))
17
```

Functions

The functions directory contains the python file *aaa-functions.py* which has all related functions that sends a request to aaa-functions to get the data a bot needs. Through these functions is how we integrate the business/SOR functions we have in aaa-functions to CDMS. This is not to be mistaken with aaa-functions, which is a separate service from CDMS. Instead, you can think of these functions as the middleman between CDMS and aaa-functions.

Here is an example of how you would write a function to make a request to its respective aaa-function:

aaa-function from dispute_bot

```
async def get_account_dispute_activity(account_token, dispute_types, correlation_id, user_jwt,
merchant_name=None):
    function_name = 'get_account_dispute_activity'
    try:
        request_url = config.function_url('askamex.cdms.aaa.functions.url',
'get_account_dispute_activity_sor_v1')

        request_data = {
            "context": {
                "accountTokens": [account_token],
                "disputeTypes": dispute_types,
                "userAuthorization": user_jwt,
                "useEncryptedDisputeId": True
            },
            "user": {},
            "map": {}
        }
        if merchant_name is not None:
            request_data['context']['merchantName'] = merchant_name

        log_data = {'correlation_id': correlation_id,
                    'app_name': Constants.APP_NAME,
                    'event_name': function_name,
                    'bot_name': bot_config().get('botname')
                    }

        response_data = await aio_http_post(request_url,
                                            request_data,
                                            _headers(correlation_id, user_jwt),
                                            config.get_int('askamex.cdms.dispute.timeout'),
                                            log_params=log_data)

        if 'error' not in response_data:
            return response_data.get('response', {}).get('data', {}), 200
        else:
            return f'{function_name}_api_failure', 400
    except Exception as error:
        log.error(**{'error': "get_account_dispute_activity Exception", 'reason': str(error)})
        raise CdmsException('transfer_ccp', f'Error while retrieving {function_name} ' + str(error))
```

Of course, the request_data fields will depend on what parameters the specific aaa-function needs. But the basic structure of the function itself like needing the function_name, request_url, request_data, etc. remains the same across the majority of the functions in aaa_functions.



This module is deprecated and is no longer used for current bot development.

Services - DEPRECATED

Modules under Services serve as 'behind-the-scenes' type functions that give information to the bot to deliver to the customer.

Classes found in services include:

- **FaasServices**: used to retrieve or submit from a customer's account. For example, it is used to retrieve available loan offers or account limits via an API call or submit a line increase request.

```

10
11 class FaasServices:
12
13     @classmethod
14     async def get_loan_offers(cls, bbv, gk, at, locale, correlation_id):
15         func_name = 'get_loan_offers'
16         try:
17             request_url = function_url_config('askamex.cdms.faas.url', 'get_loan_offer_sor_v1')
18             request_data = {
19                 "context": {
20                     'blueboxvalues': bbv,
21                     'gatekeeper': gk,
22                     'accountToken': at,
23                     'loan_product': Constants.PERSONAL_LOAN_PRODUCT,
24                     'channel_type': config('askamex.cdms.loan.channel.type'),
25                     'country_code': config('askamex.cdms.loan.country.code.{}'.format(locale)),
26                     'client_id': config('askamex.cdms.loan.client.id')
27                 },
28                 "user": {},
29                 "map": {}
30             }
31             log.info(**{'correlation_id': correlation_id,
32                        'app_name': Constants.APP_NAME,
33                        'event_name': func_name,
34                        'action': f'{Constants.ACTION_START} {func_name}',
35                        'desc': 'entering_message_received',
36                        'result': Constants.LOG_SUCCESS_RESULT,
37                        'reason': Constants.LOG_SUCCESS_REASON,
38                        'bot_name': bot_config().get('botname')})

```

- **TemplateServices:** used to format customer metadata such as card numbers or previous transaction information to be output as a dialog option. Usage is most common in creating an attachment to be used in a the handler. This is different from the templates that are set up in [askAmex bot management services](#).

```

3
4 class TemplateServices:
5     @staticmethod
6     def get_card_lists_template(accounts):
7         cards_array = []
8         for account in accounts:
9             card = dict()
10             card['option_id'] = str(uuid.uuid4())
11             card['option_text'] = account['account_number_display_text']
12             cards_array.append(card)
13             if len(cards_array) > 3:
14                 cards_array = cards_array[0:3]
15                 card['option_id'] = str(uuid.uuid4())
16                 card['option_text'] = "Another Card"
17                 cards_array.append(card)
18             return cards_array
19
20

```

Sub-Intents

As the name suggests, sub-intents are intents found under a general intent. You can think of them as a specific category within a broader topic. Note that NOT all bots are required to handle sub-intents.

Before getting into the actual sub_intents module, let's look at the *actions/intent_received.yaml* file of the payment late bot (pl_bot) as an example:

sub_intent_based_requirements: Has more specifically named intents that falls under category found under *in_scope_intents*. (Fig A.)

So if a card member comes in requesting for help to get a payment late waiver, the utterance is evaluated to see if it's in scope of the sub-intent threshold.

If a sub_intent entity is found, then control is "switched" to process the sub-intent actions, which takes us into the sub_intents module.

sub_intents module

File Structure

```
sub_intents
  entities
    actions
    util
```

The sub_intents module is similar to that of your standard bot file structure and follows the same logic. Think of it as a bot within a bot.

Fig A. intent_received.yaml

```
requirements:
  topic_based_requirements:
    - capture_type: intent
      in_scope_intents:
        intents:
          - payment_late
          - payment_late_sl
        threshold: 0.2
      sub_intent_based_requirements: <----- include
to capture sub_intents
    - capture_type: sub_intent <----- define
sub_intents
      in_scope_sub_intents:
        - threshold: 0.2
          type: entity
          intents:
            - payment_late_waiver
            - payment_late_other

events:
  ...
  - condition: is_entity_sub_intent_exists
    handler: execute_handler
    commands:
      - operation: next_action
        inputs:
          action: intent_received
      - operation: delegate_control <---- switch
control to sub_intent bot
        inputs:
          flow_type: sub_intents
          bot_type: entities
          source_bot: pl_bot
          current_action: intent_received
```

Util

Package that contains the **bot_util.py** module. This file contains bot configurations *bot_config()* along with any other helper functions that you would need to call at any point in the bot flow.

ALL bots are required to have a bot_config. An example from the loan bot is shown below.

```
6 def bot_config():
7     return {
8         'notify.sleep.seconds': config('askamex.cdms.loan.notify.sleep.seconds'),
9         'notify.retries': config('askamex.cdms.loan.notify.retries'),
10        'master.botname': config('askamex.cdms.loan.master.botname'),
11        'botname': config('askamex.cdms.loan.botname'),
12        'destination.botname': config('askamex.cdms.loan.destination.botname'),
13        'message_id.prefix': config('askamex.cdms.loan.message_id.prefix'),
14        'bot.user_id': config('askamex.cdms.loan.bot.user_id'),
15        'journey': 'loan',
16        'logger.version': 'v1'
17    }
18
```

✓ TIP: If you are just starting out building a bot, you most likely won't be able to use any of the config(...) values. For the purpose of being able to actually run your bot, just hardcode your own config values (using your bot name where appropriate)

These values are pulled from the config server and will need to be added in.

The functions below are examples of helper functions found in the loan bot **bot_util.py**

```
20 def selected_card_account(member_accounts, selected_card):
21     return selected_card_account(member_accounts, selected_card).get('account_token')
22
23 def selected_card_account(member_accounts, selected_card=None):
24     return member_accounts[0] if len(member_accounts) == 1 else \
25         next((account for account in member_accounts if account['account_number_display_text'] == selected_card), ())
26
27 def get_auth_headers(header):
28     auth_header = header['X-ADP-User-Legacy-Authorization'].split(',')
29     return auth_header[0].split('=')[1], auth_header[1].split('=')[1]
30
31 def currency_formatter(number, currency, locale):
32     return format_currency(number, currency, locale=locale.replace('-', '_'), format='u', currency_digits=False)
```



Templates are now to be set up in the Dialog Management System (DMS) Explorer - <https://c3ipbot-qa.aexp.com/explorer>

For E2 access, please message Sathish Chandrasekaran to add your adslid.

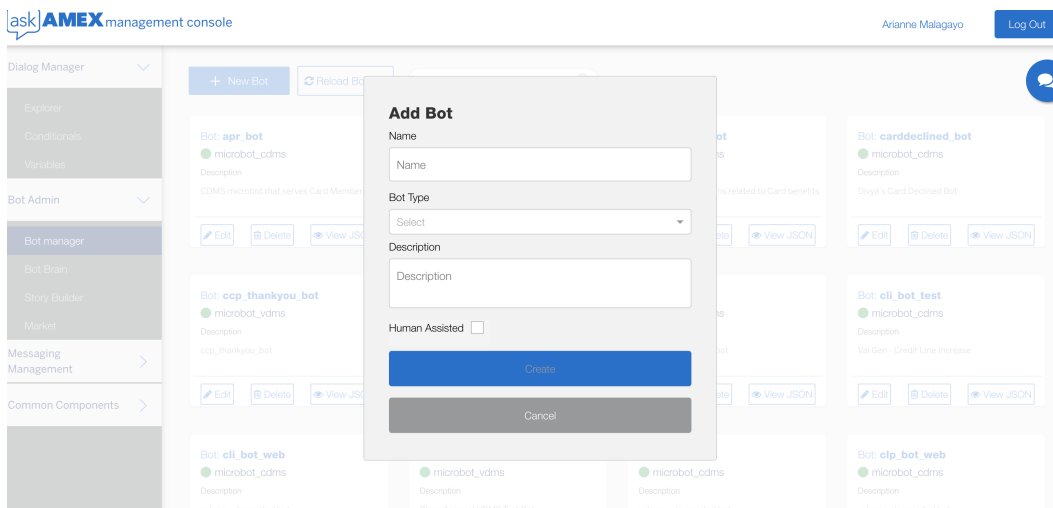
Templates

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 Dialog Manager](#).

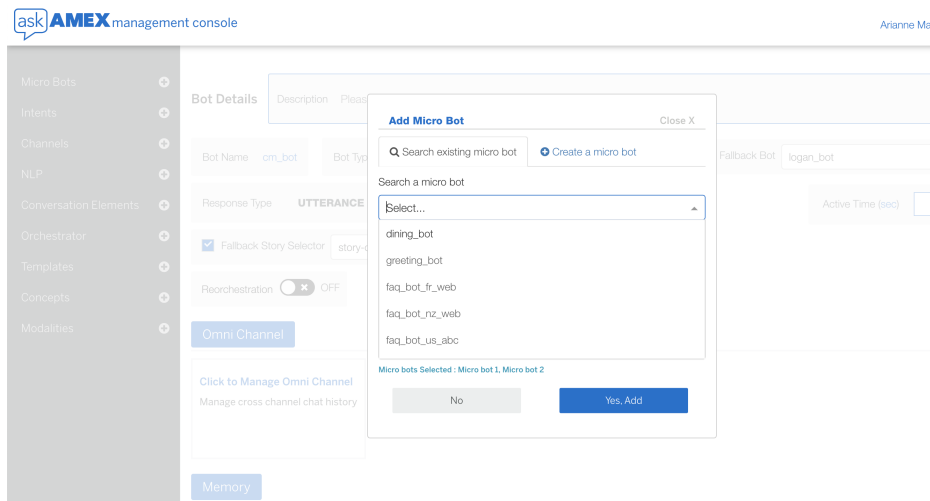
Step 1. Navigate to the **Explorer** section under **Dialog Manager** and select 'Add response'.

Step 2. Select the market, locale, channel, and bot. For all of our CDMS bots the market will be US, locale is en-US, channel is mobile.

If you're adding a new bot that isn't in the list, go to Bot Admin > Bot Manager > Add Bot. Add in your bot name and select Bot Type: microbot cdms. Click Create.



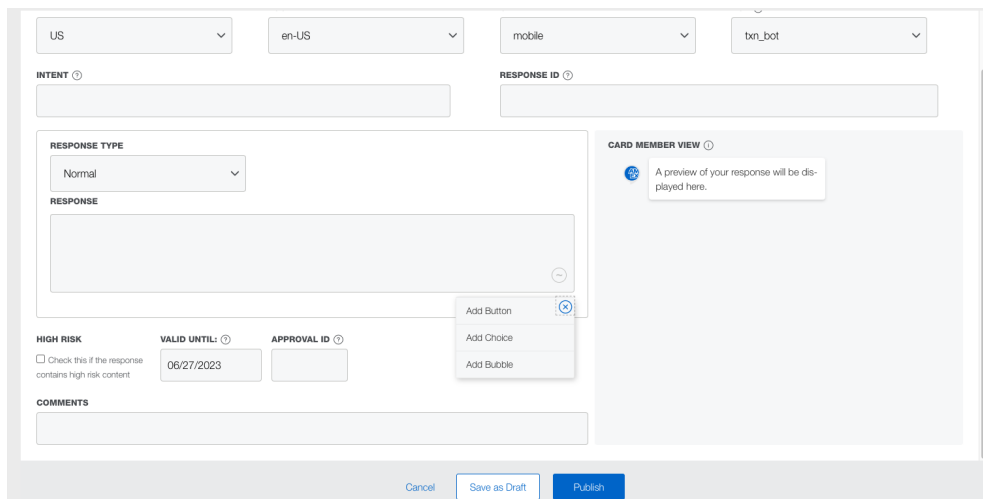
Then go to Bot Manager > cm_bot > click Micro Bots (left side bar) > Select the microbot you just created > Yes, Add > Scroll down the page a click Publish



Step 3. Add the name of your template in **Response ID**.

Step 4. Add the the content you want for your template.

If your template requires hyperlinks or deeplinks, choose **Add Button**. If your template requires multi-choice options, choose **Add Choice**.



Once your content is added, click **Publish**.

Adding Dynamic Variables

If your template requires dynamic variables (i.e. using variables stored in CDMS memory), you will need to add them to the **variable registry**. You can use this [Variable Migration Tool](#) to add your variables to the registry. If you prefer to update the registry via Postman, please reach out to an Automation team member to get the collection.

The list of variables in the registry can be found under Dialog Manager > Variables

ask **AMEX** management console Arianne Malagayo Log Out

NAME	DATA TYPE	DESCRIPTION	BOT	MANDATORY	FORMATS	VERSION
Account key	STRING	This variable will be used when constructing Offers dynamic deeplinks.	offers_bot	FALSE	en-IN: ##### en-NZ: ##### en-AU: ##### en-SG: ##### en-US: ##### en-GB: #####	1.0
Account key	STRING	This variable will be used when constructing Offers dynamic deeplinks.	offers_bot	FALSE	en-IN: ##### en-NZ: ##### en-AU: ##### en-SG: ##### en-US: ##### en-GB: #####	1.0
Auto Pay Date	STRING	This variable will be used to display the Auto pay scheduled date	payments_bot	FALSE	en-IN: ##### en-NZ: ##### en-AU: ##### en-SG: ##### en-US: ##### en-GB: #####	1.0

Dynamic variables are references by the variable name wrapped in two curly braces. e.g. `{{variable_name}}`

If the variable name in the card member view appears in red instead of purple, then the variable is not present in the variable registry.

RESPONSE

We can send you an email once this charge posts to your account, usually within 72 hours of when it was made.

An email will be sent to:
`{{display_mail_address}}`

CHOICE

DISPLAY TEXT

Send alert

EDITED CARD MEMBER VIEW

We can send you an email once this charge posts to your account, usually within 72 hours of when it was made.

An email will be sent to:
`display_mail_address`

Send alert

No, thanks

Requests Using Postman

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`

POST my_bot_request

×

+

...

No Environment

👁

▶ my_bot_request

Examples (t

POST

http://127.0.0.1:6000/askamexcdms/v1/message

Send

Save

Params

Authorization

Headers

Body

Pre-request Script

Tests

Cookies

Code

Commer

KEY	VALUE	DESCRIPTION	...	Bulk
Key	Value	Description		

Response

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	Appld 661ef271-3037-4ae8-a3f0-916fa4f19b8d
X-AXP-User-Authorization	Appld 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.

```
Params Authorization Headers (9) Body Pre-request Script Tests Cookies Code Comme
● none ● form-data ● x-www-form-urlencoded ● raw ● binary JSON (application/json) ▼ Bea

1 {
2   "request": {
3     "userId": "u_068bed9a6b17dfab550df990e8cfce6c",
4     "conversationId": "conv_63c04dacd4c42e228e52e1cc540017b0",
5     "channelId": "mobile",
6     "requestType": "message",
7     "message": {
8       "content": {
9         "desc": "Getting a new loan"
10      },
11      "type": "text",
12      "userId": "u_068bed9a6b17dfab550df990e8cfce6c",
13    }
14  }
15 }
```

This is the utterance that you want to send.

```
16   "update_time": "2019-03-26T22:14:02.000805"
17 },
18 },
19 "secondaryConversationId": "sconv_6f2a3dd3-8089-476b-81a3-bd2947d3412c",
20 "microBotName": "loan_bot"
21 }
22 "user": {
23   "isAgent": "false",
24   "cUserId": "u_63c04dacd4c42e228e52e1cc540017b0",
25   "cUserIdType": "public_guid",
26   "version": "v1",
27   "createTime": "2019-02-19 21:53:55",
28   "updateTime": "2019-02-19 21:53:55",
29   "data valid min time": "2019-03-15 21:57:30".
30 }
```

Change to the name of your microbot.

```
204 "masterBotMemory": {
205   "conversationStringContext": {
206     "nlpData": {
207       "type": "amex_nlp",
208       "confidenceMinThreshold": 0.0,
209       "nlpResponse": {
210         "intentName": "loan",
211         "confidenceScore": 1.0,
212         "secondaryIntents": [],
213         "concepts": [
214           {
215             "name": "CARD_TYPE",
216             "type": "CARD_TYPE",
217             "value": "Blue"
218           }
219         ],
220         "words": ["Blue"],
221         "confidence": 1,
222         "sentiments": [],
223         "relatedIntents": []
224       }
225     }
226   }
227 }
```

Change to the relevant intents of your bot.

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 will have a similar response schema to the following:

```
Body Cookies Headers (4) Test Results Status: 200 OK Time: 194 ms Size: 6
Pretty Raw Preview JSON ▼

1 {
2   "status": "SUCCESS",
3   "messageId": "r_a155e661-379a-4c2d-8b14-6b25124f6c30",
4   "conversationId": "conv_63c04dacd4c42e228e52e1cc540017b0",
5   "message": {
6     "content": {
7       "desc": "My card was declined"
8     },
9     "type": "text",
10    "userId": "u_068bed9a6b17dfab550df990e8cfce6c",
11    "conversationId": "conv_63c04dacd4c42e228e52e1cc540017b0",
12    "create_time": "2019-03-26T22:14:02.000805",
13    "update_time": "2019-03-26T22:14:02.000805",
14    "messageId": "r_a155e661-379a-4c2d-8b14-6b25124f6c30"
15  }
16 }
```

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.

```
1 {
2   "status": "SUCCESS",
3   "statusMsg": "out_of_scope"
4 }
```

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.

GET Memory

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	<i>your bot name, found under 'microBotName' of POST request</i>
cUserId	<i>your channel userId, found in 'userId' of POST request</i>
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, than the bot memory was most likely deleted.

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	<i>your bot name, found under 'microBotName' of POST request</i>
cUserId	<i>your channel userId, found in 'userId' of POST request</i>
channelId	mobile
version	v1

Step 3. Add in the following headers.

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

Step 4. Send the request. You should see 'SUCCESS' as the status and statusMsg. If you get an error, then the bot context was most likely deleted already.