AskAmex-CDMS-Core

This page is currently under construction!
Thank you for your patience ~

Before going into the code flow,the basic variable we need to know is **me mory.**Memory variable is sent as input to almost each function.The memory variable keeps on updating with the bot flow.We keeps updating only **conversationstringcontext** variable of memory.We try to update memory by calling cdms(memory)[variable_that_needs_to_be_updated]="value"

Here cdms is a function which returns memory['bot_context']['conversationStringContext']

CDMS -Core contains the code for handlers, routers and also the functions for each action(send_message, next_action).

On_message

In this function,it checks whether the microbot is ddds_bot or not.If it is ddds_bot, then Bot handler message_reeceived_ddds function is called. If micro-bot name is not ddds_bot, control is passed to message received function.

---

message_received

Firstly,here the **bot_config variable** is assigned from the request variables.In the request data,the microbotName variable is sent.We define bot_config function in every bot in the util folder.This bot_config function defines the configurations of bot like microbotName, destinationbot_name.

Next the memory variable is also updated in this function.The memory variable is updated with bot_context which is json of botname,channelid, version and so on.

Here memory variable is assigned and the microbot to be executed is also configured.If the flow-type is sub_intents then the process_sub_intent_bot_actions functions is executed or else process_async_bot_actions function is executed

---

process_async_bot_actions

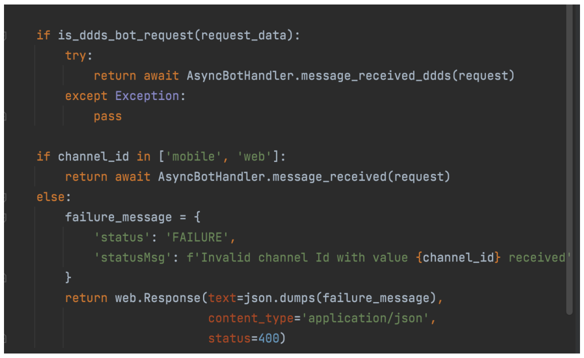
This is where the actual bot flow starts.This function will decide which action to be performed based on the presence of next_action in the request data.If the next action is present in the request,then the mentioned action of will be executed or else the intent_received file of the action will be called.

After deciding which action to be performed,it executes capture_entities, capture requirements and execute function of that action

Every file in actions folder consists of capture requirements,capture entitities and execute functions in common.

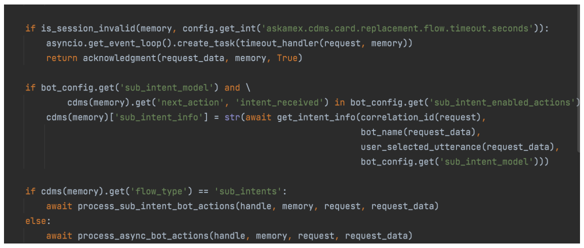
> **capture requirements -**There are 3 types of capturing requirements.The actual functionality of each these requirements is defined in the cdms core.
>
> Intent capture
>
> List capture
>
> Function capture

This is the screenshot for **on_message** function



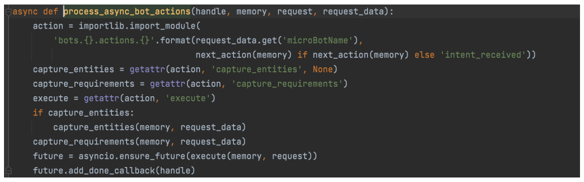
```
if is_ddds_bot_request(request_data):
    try:
        return await AsyncBotHandler.message_received_ddds(request)
    except Exception:
        pass

if channel_id in ['mobile', 'web']:
    return await AsyncBotHandler.message_received(request)
else:
    failure_message = {
        'status': 'FAILURE',
        'statusMsg': f'Invalid channel Id with value {channel_id} received'
    }
    return web.Response(text=json.dumps(failure_message),
                        content_type='application/json',
                        status=400)
```

message_received



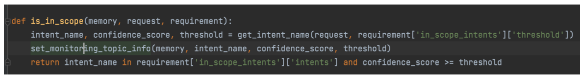
```
if is_session_invalid(memory, config.get_int('askamex.cdms.card.replacement.flow.timeout.seconds')):
    asyncio.get_event_loop().create_task(timeout_handler(request, memory))
    return acknowledgment(request_data, memory, True)

if bot_config.get('sub_intent_model') and \
        cdms(memory).get('next_action', 'intent_received') in bot_config.get('sub_intent_enabled_actions')
    cdms(memory)['sub_intent_info'] = str(await get_intent_info(correlation_id(request),
                                                                 bot_name(request_data),
                                                                 user_selected_utterance(request_data),
                                                                 bot_config.get('sub_intent_model')))

if cdms(memory).get('flow_type') == 'sub_intents':
    await process_sub_intent_bot_actions(handle, memory, request, request_data)
else:
    await process_async_bot_actions(handle, memory, request, request_data)
```

process_async_bot_actions



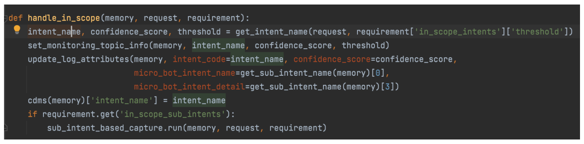
```
async def process_async_bot_actions(handle, memory, request, request_data):
    action = importlib.import_module(
        'bots.{}.actions.{}'.format(request_data.get('microBotName'),
                                    next_action(memory) if next_action(memory) else 'intent_received'))
    capture_entities = getattr(action, 'capture_entities', None)
    capture_requirements = getattr(action, 'capture_requirements')
    execute = getattr(action, 'execute')
    if capture_entities:
        capture_entities(memory, request_data)
    capture_requirements(memory, request_data)
    future = asyncio.ensure_future(execute(memory, request))
    future.add_done_callback(handle)
```

Intent capture-is_in_scope function



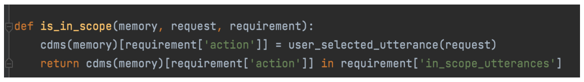
```
def is_in_scope(memory, request, requirement):
    intent_name, confidence_score, threshold = get_intent_name(request, requirement['in_scope_intents']['threshold'])
    set_monitoring_topic_info(memory, intent_name, confidence_score, threshold)
    return intent_name in requirement['in_scope_intents']['intents'] and confidence_score >= threshold
```

Intent capture-handler_in_scope function



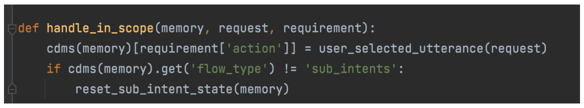
```
def handle_in_scope(memory, request, requirement):
    intent_name, confidence_score, threshold = get_intent_name(request, requirement['in_scope_intents']['threshold'])
    set_monitoring_topic_info(memory, intent_name, confidence_score, threshold)
    update_log_attributes(memory, intent_code=intent_name, confidence_score=confidence_score,
                          micro_bot_intent_name=get_sub_intent_name(memory)[0],
                          micro_bot_intent_detail=get_sub_intent_name(memory)[3])
    cdms(memory)['intent_name'] = intent_name
    if requirement.get('in_scope_sub_intents'):
        sub_intent_based_capture.run(memory, request, requirement)
```

list capture-is_in_scope function



```
def is_in_scope(memory, request, requirement):
    cdms(memory)[requirement['action']] = user_selected_utterance(request)
    return cdms(memory)[requirement['action']] in requirement['in_scope_utterances']
```

list capture-handle_in_scope function



```
def handle_in_scope(memory, request, requirement):
    cdms(memory)[requirement['action']] = user_selected_utterance(request)
    if cdms(memory).get('flow_type') != 'sub_intents':
        reset_sub_intent_state(memory)
```

card_selector function

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 Main bot gives the control to loan boat.This capture requirements function checks whether get_loan intent is in scope of loan bot or not.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 loan-bot tells main-bot 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.

In the below functions,bot is checking whether the intent is in scope of the intents that are configured to bot.

In the below function,we are updating the memory of the bot,if intent is in scope of bot intents and confidence is greater than threshold.

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.Here the list of options provided to user are static.

In_Scope function of list capture-This function validates whether the user utterance is in scope utterances of list options.The user selected utterance is stored in action variable configured in YAML file of bot memory.

handle_in_scope-This funciton updates the bot memory with the user selected utterance .

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.

capture entities

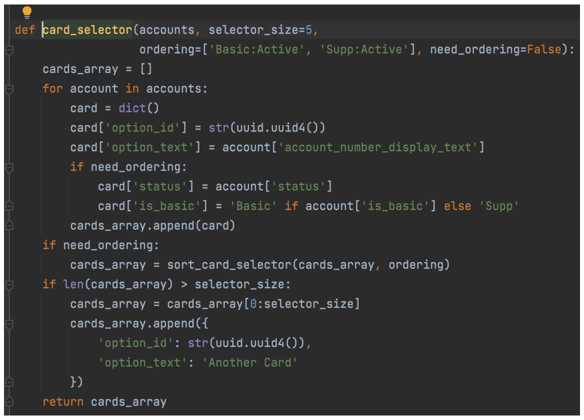
The entity type,entity value are all configured in the cdms bot.In CDMS-core we are actually updating the memory variable of the bot with the entity values we have captured in the user utterance. We are also calling the post_processor_entity function which actually checks the entities we have captured are present in the user account or the cards are associated with the user.

execute_handler

This handler executes the operations of the condition that is satisfied based on the user utterance.Certain operations need to be performed if the condition is satisfied.These operations are mentioned events dict variable in the corresponding .yaml file. More about these operations are discussed in CDMS page

Some examples of custom defined functions

Card selector-This is predefined function in aaa-templates.This function retrieves the cards in the user accounts and returns the card_array which consists of option_id and option_text of card.



```python
def card_selector(accounts, selector_size=5,
                  ordering=['Basic:Active', 'Supp:Active'], need_ordering=False):
    cards_array = []
    for account in accounts:
        card = dict()
        card['option_id'] = str(uuid.uuid4())
        card['option_text'] = account['account_number_display_text']
        if need_ordering:
            card['status'] = account['status']
            card['is_basic'] = 'Basic' if account['is_basic'] else 'Supp'
        cards_array.append(card)
    if need_ordering:
        cards_array = sort_card_selector(cards_array, ordering)
    if len(cards_array) > selector_size:
        cards_array = cards_array[0:selector_size]
        cards_array.append({
            'option_id': str(uuid.uuid4()),
            'option_text': 'Another Card'
        })
    return cards_array
```

custom-defined template-Some examples of custom defined templates are handle_multi_acocunt_with_no_card_selected.This handler is defined. when the user consists of multi-cards and user has not specified the particular card.This handler gives the user various cards the user is associated with.

get_account_token-This is custom defined function type capture.This function is used when the user is provided with dynamic options to chose from.This function updates the cdms memory variable with the user selected option.

CDMS- CORE

CONSTANTS

ERRORS

HANDLERS

ASYNC BOT HANDLER

message_received

Firstly,here the **bot_config variable** is assigned from the request variables.In the request data,the microbotName variable is sent.We define bot_config function in every bot in the util folder.This bot_config function defines the configurations of bot like microbotName, destinationbot_name.

Next the memory variable is also updated in this function.The memory variable is updated with bot_context which is json of botname,channelid,version and so on.

Here memory variable is assigned and the microbot to be executed is also configured.If the flow-type is sub_intents then the process_sub_intent_bot_actions functions is executed or else process_async_bot_actions function is executed

```
class AsyncBotHandler:

    @staticmethod
    async def message_received(request):
        request_data, request_header = await request.json(), request.headers
        bot_config = get_bot_config(request_data)

        log.info(**{'correlation_id': correlation_id(request),
                    'app_name': Constants.APP_NAME,
                    'event_name': 'message_received',
                    'action': f'{Constants.ACTION_START} message_received',
                    'desc': 'entering_message_received',
                    'result': Constants.LOG_SUCCESS_RESULT,
                    'reason': Constants.LOG_SUCCESS_REASON,
                    'micro_bot_name': bot_name(request_data),
                    'utterance': user_selected_utterance(request_data),
                    'user_id': user_id(request_data)
                    })

        def handle(fut):
            try:
                fut.result()
            except CdmsException as cdms_exception:
                asyncio.get_event_loop().create_task(exception_handler(cdms_exception, request, memory))

        try:
            memory = ''
            memory = await get_memory_handler(request, request_data, request_header)
            set_log_attribute(memory, 'correlation_id', correlation_id(request))
            if is_card_member_flow(request):
                set_and_get_accounts(memory, request_data)
            set_monitoring_utterance_info(memory, request_data, locale(request))

            if not log_attribute_value(memory, Constants.TRACE_ID):
                update_log_attributes(memory, correlation_id=correlation_id(request), trace_id=trace_id(),
                                      app_name=Constants.APP_NAME, channel_name=channel_name(request_data),
                                      locale=locale(request), micro_bot_name=bot_name(request_data),
                                      chat_session_id=chat_session_id(request_data),
                                      cardmember_id=user_id(request_data),
                                      user_id=user_id(request_data), user_type=user_type(request_data),
                                      conversation_id=conversation_id(request_data),
                                      interaction_id=interaction_id(request_data),
                                      secondary_conversation_id=secondary_conversation_id(request_data),
                                      chat_start_time=current_time_utc())

            if is_session_invalid(memory, config.get_int('askamex.cdms.card.replacement.flow.timeout.
seconds')):
                asyncio.get_event_loop().create_task(timeout_handler(request, memory))
                return acknowledgment(request_data, memory, True)

            if bot_config.get('sub_intent_model') and \
                    cdms(memory).get('next_action', 'intent_received') in bot_config.get
('sub_intent_enabled_actions'):
                cdms(memory)['sub_intent_info'] = str(await get_intent_info(correlation_id(request),
                                                                            bot_name(request_data),
                                                                            user_selected_utterance
(request_data),
                                                                            bot_config.get
('sub_intent_model')))

            if cdms(memory).get('flow_type') == 'sub_intents':
                await process_sub_intent_bot_actions(handle, memory, request, request_data)
            else:
                await process_async_bot_actions(handle, memory, request, request_data)
            return acknowledgment(request_data, memory, True)
        except CdmsException as error:
            log_attributes = {**get_log_attributes(memory, request_data), **{
                'event_name': 'message_received',
                'action': f'{Constants.ACTION_END} message_received',
                'desc': 'exiting_message_received',
                'result': Constants.LOG_FAILURE_RESULT,
                'reason': str(error),
                'chat_transcript_text': user_selected_utterance(request_data)
            }}
            return await cdms_error_response(memory, error, log_attributes)
        except Exception as error:
            log.error(**{**get_log_attributes(memory, request_data), **{
                'event_name': 'message_received',
                'action': f'{Constants.ACTION_END} message_received',
                'desc': 'exiting_message_received',
                'result': Constants.LOG_FAILURE_RESULT,
                'reason': str(error),
                'chat_transcript_text': user_selected_utterance(request_data)
            }})
            return await error_response(memory, error)
```

RESOURCE HANDLER

ERROR RESPONSE

If the input is in invalid format,this function throws an error like invalid input format with error in updating the memory

```
async def error_response(memory, error):
    if memory:
        memory_clone = copy.deepcopy(memory)
        memory['bot_context']['conversationStringContext'] = {}
        await update_memory(memory, False)
        return web.Response(text=json.dumps(failure_message('Invalid input format', error, memory_clone)),
                            content_type='application/json',
                            status=400)
    else:
        return web.Response(text=json.dumps(failure_message('Unknown error', error)),
                            content_type='application/json',
                            status=400)
```

CDMS ERROR RESPONSE

This function also throws error like out of scope or invalid input format based on log attributes and error the function receives as input

```
async def cdms_error_response(memory, e, log_attributes={}):
    if log_attributes and e and e.code == 'out_of_scope':
        log.info(**log_attributes)
    else:
        log.error(**log_attributes)
    memory_clone = copy.deepcopy(memory)
    memory['bot_context']['conversationStringContext'] = {}
    await update_memory(memory, False)
    if e.code == 'out_of_scope':
        return web.Response(text=json.dumps(out_of_scope_message(e, memory_clone)),
                            content_type='application/json',
                            status=200)
    else:
        return web.Response(text=json.dumps(failure_message('Invalid input format', e, memory_clone)),
                            content_type='application/json',
                            status=400)
```

TIMEOUT HANDLER

This function sends message with default error template when the time exceeds the mentioned time in the bot-config file of the relevant bot.It is only valid when the channel is mobile.

```
async def timeout_handler(request, memory):
    request_data = await request.json()
    bot_config = get_bot_config(request_data)
    if channel_name(request_data) == 'mobile':
        inputs = {'request': request,
                  'template': Constants.BOT_ERROR_DEFAULT_TEMPLATE,
                  'message_type': 'text',
                  'memory': memory,
                  'bot_config': bot_config
                  }
        await send_message(**inputs)

    if not bot_config.get('skip.relinquish', False):
        inputs = {'request': request,
                  'template': "Session timeout",
                  'action': 'relinquish_control',
                  'reason': '',
                  'memory': memory,
                  'bot_config': bot_config
                  }
        await relinquish_control(**inputs)
```

EXCEPTION HANDLER

```
async def exception_handler(cdms_exception, request, memory):
    request_data = await request.json()
    bot_config = get_bot_config(request_data)
    if channel_name(request_data) == 'mobile' and Constants.SEND_MSG_FAILED_ERROR_CODE not in cdms_exception.
message:
        inputs = {'request': request,
                  'template': Constants.BOT_ERROR_DEFAULT_TEMPLATE,
                  'message_type': 'text',
                  'memory': memory,
                  'bot_config': bot_config
                  }
        await send_message(**inputs)

    if not bot_config.get('skip.relinquish', False):
        inputs = {'request': request,
                  'template': Constants.BOT_ERROR_DEFAULT_TEMPLATE,
                  'action': 'relinquish_control',
                  'reason': 'transfer',
                  'memory': memory,
                  'bot_config': bot_config
                  }
        cdms(memory)['error'] = cdms_exception
        await relinquish_control(**inputs)
```

GET MEMORY HANDLER

This function retrieves the memory from the database and updates the memory with the context variable from the request

```
async def get_memory_handler(request, request_data, request_header):
    memory = await get_bot_memory(request, request_data, request_header)
    if request_data.get('request', {}).get('context', {}).get('data'):
        memory['bot_context']['conversationStringContext'] = {}
        await update_memory(memory, False)
        memory = await get_bot_memory(request, request_data, request_header)
        cdms(memory)['context'] = str(request_data.get('request', {}).get('context'))
    return memory
```

GET BOT MEMROY

This function calls get memory which make a post call to retrive the memory

```
async def get_bot_memory(request, request_data, request_header):
    memory = await get_memory(request_header.get('X-Axp-Correlationid'),
                              request_data.get('microBotName') or request_data.get('destBotName'),
                              request_data.get('request', {}).get('userId') or request_data.get('userId'),
                              request_data.get('request', {}).get('channelId') or request_data.get
('channelId'))
    cdms(memory)['lock_value'] = memory['lock_value']
    cdms(memory)['correlation_id'] = correlation_id(request)
    cdms(memory)['message_id'] = 'r_' + str(uuid.uuid4())
    cdms(memory)['main_bot_name'] = request_header.get('X-AXP-Main-BotName')
    return memory
```

LIBRARY

OPERATIONS

delegate control-This operation is performed within the 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.In this function,it updates the cdms memory with sub-intent action that needs to be performed next.This action that is to be performed next is assigned in **next_sub_intent_action.**Other variables such as **source bot** and **current action** are also updated in the memory and the process_sub_intent_bot_actions function is called.

```
async def run(**kwargs):
    command, memory, request, bot_config = \
        kwargs.get('command'), kwargs.get('memory'), kwargs.get('request'), \
        kwargs.get('bot_config')
    cdms(memory)['flow_type'] = command.get('inputs', {}).get('flow_type')
    cdms(memory)['bot_type'] = command.get('inputs', {}).get('bot_type')
    if command.get('inputs', {}).get('action'):
        cdms(memory)['next_sub_intent_action'] = command.get('inputs', {}).get('action')
    if command.get('inputs', {}).get('source_bot'):
        cdms(memory)['source_bot'] = command.get('inputs', {}).get('source_bot')
    if command.get('inputs', {}).get('current_action'):
        cdms(memory)['current_action'] = command.get('inputs', {}).get('current_action')
    await process_sub_intent_bot_actions(handle, memory, request, await request.json())
```

event tracking

```
def run(**kwargs):

    command, memory, request, bot_config, attachment = \
        kwargs.get('command'), kwargs.get('memory'), kwargs.get('request'), kwargs.get('bot_config'), \
        kwargs.get('attachment')

    event_inputs = {**command.get('inputs', {}),
              **{'request': request,
                 'bot_config': bot_config,
                 'memory': memory,
                 'attachment': attachment() if attachment else None}}

    di_util.ingest(**{'target': 'transcript_kafka_topic', 'message': event_inputs})
```

grant control-This operation is performed within the 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.In this function,it updates the cdms memory with sub-intent action that needs to be performed next.This action that is to be performed next is assigned in **next_sub_intent_action**.In addition to this variable,control_type variable is also updated with value grant.and the process_sub_intent_bot_actions function is called.

```
async def run(**kwargs):
    command, memory, request, bot_config = \
        kwargs.get('command'), kwargs.get('memory'), kwargs.get('request'), \
        kwargs.get('bot_config')
    cdms(memory)['flow_type'] = command.get('inputs', {}).get('flow_type')
    cdms(memory)['bot_type'] = command.get('inputs', {}).get('bot_type')
    if command.get('inputs', {}).get('action'):
        cdms(memory)['next_sub_intent_action'] = command.get('inputs', {}).get('action')
    cdms(memory)['control_type'] = 'grant'
    await process_sub_intent_bot_actions(handle, memory, request, await request.json())
```

include previous messages-Here in this function,we are updating the message attributes with the template if it send to the function as input or the template attribute is updated with message attributes of the message.Then we are sending message by calling send_message function.

```
async def run(**kwargs):
    memory, request, bot_config, template, message_attributes = \
        kwargs.get('memory'), kwargs.get('request'), kwargs.get('bot_config'), \
        kwargs.get('template'), kwargs.get('message_attributes')

    for previous_message in ast.literal_eval(cdms(memory)['previous_messages']):
        inputs = {**previous_message.get('inputs', {}),
                  **{'request': request,
                     'bot_config': bot_config,
                     'memory': memory
                     }}
        if template:
            inputs.update({'template': template})
        elif message_attributes:
            inputs.update(
                {'template': previous_message.get('inputs').get('template').format(*message_attributes)})
        await send_message(**inputs)
```

next action - Here we are updating the memory variable with the next action that need to be performed by the bot.This value is assigned to **next_action** variable

```
async def run(**kwargs):
    command, memory = kwargs.get('command'), kwargs.get('memory')
    set_next_action(memory, command.get('inputs', {}).get('action'))
```

next sub intent actions -

```
async def run(**kwargs):
    command, memory = kwargs.get('command'), kwargs.get('memory')
    set_next_sub_intent_action(memory, command.get('inputs', {}).get('action'))
```

no matching operations - This function raises an exception like Failed to get a valid operation if no operation matched the operations that are configured in the core.

```
async def run(**kwargs):
    raise CdmsException('FAILURE', 'Failed to get a valid operation')
```

relinquish control-TThis gives the control to customer care professional.

```
async def run(**kwargs):
    command, memory, request, bot_config, is_sync = \
        kwargs.get('command'), kwargs.get('memory'), kwargs.get('request'), \
        kwargs.get('bot_config'), kwargs.get('is_sync')
    memory_copy = copy.deepcopy(memory)
    inputs = {**command.get('inputs', {}), **{'request': await request.json(),
                                              'bot_config': bot_config, 'memory': memory_copy}}
    inputs.update({'request': request})
    return await relinquish_control(**inputs)
```

revert control-This operation just updates the variables like flow_type.. with None

```
async def run(**kwargs):
    cdms(kwargs.get('memory'))['flow_type'] = ''
    cdms(kwargs.get('memory'))['bot_type'] = ''
    cdms(kwargs.get('memory'))['next_sub_intent_action'] = ''
    cdms(kwargs.get('memory'))['sub_intent_type'] = ''
    cdms(kwargs.get('memory'))['current_action'] = ''
    cdms(kwargs.get('memory'))['destination_bot'] = ''
```

send message - In this function, if we are sending template to execute handler function as input then the template variable is updated with the template.But if we configure in the .yaml file, then it updates the template variable with the template that we have configured in .yaml file

```
async def run(**kwargs):
    command, memory, request, bot_config, template, message_attributes, attachment, is_sync = \
        kwargs.get('command'), kwargs.get('memory'), kwargs.get('request'), kwargs.get('bot_config'), \
        kwargs.get('template'), kwargs.get('message_attributes'), kwargs.get('attachment'), kwargs.get
('is_sync')

    inputs = {**command.get('inputs', {}),
              **{'request': request,
                 'bot_config': bot_config,
                 'memory': memory,
                 'attachment': attachment() if attachment else None}}
    if template:
        inputs.update({'template': template})
    elif message_attributes:
        inputs.update({'template': command.get('inputs').get('template').format(*message_attributes)})
    return await notification_message(**inputs) if is_sync else await send_message(**inputs)
```

s**et entity -** Here we are updating the entity name in the cdms memory

```
async def run(**kwargs):
    command, memory = kwargs.get('command'), kwargs.get('memory')
    inputs = command.get('inputs', {})
    cdms(memory)[inputs['name']] = inputs['value']
```

skip processing

```
async def run(**kwargs):
    pass
```

update memory - The memory variable is updated if the action is reset or else it makes a post call to database where the memory is saved into the database

```
async def run(**kwargs):
    memory = kwargs.get('memory')
    if kwargs.get('command', {}).get('inputs', {}).get('action') == 'reset':
        memory['bot_context']['conversationStringContext'] = {}
    await update_memory(memory)
```

update previous message -

```
async def run(**kwargs):
    memory, attachment =  kwargs.get('memory'), kwargs.get('attachment')
    previous_message_list = ast.literal_eval(cdms(memory).get('previous_messages'))
    previous_message_list[0]['inputs']['attachment'] = attachment() if attachment else None
    cdms(memory)['previous_messages'] = str(previous_message_list)
```

REQUIREMENTS

function based capture - This file has functions like run ,is_in_scope, i s_out_of_scope, handle_in_scope, handle_out_of_scope.

function run has actions variable with is_in_scope, is_out_scope as keys and handlers as execute handlers.

Run function checks whether the requirement that is captured in is in scope or not.If it is in scope, then the **requirement['action']** variable is updated in cdms memory with the user utterance message.If the condiiton of is_out_scope is satisfies, then the variable out_of_scope is set to true.In addition to that out_of_scope_msg' is also set with message Unsupported user selection

```
def run(memory, request, requirement):
    actions = {
        is_in_scope: handle_in_scope,
        is_out_of_scope: handle_out_of_scope,
    }
    condition_generator = (condition for condition in actions.keys()
                           if condition(memory, request, requirement))
    actions[next(condition_generator)](memory, request, requirement)


def is_in_scope(memory, request, requirement):
    cdms(memory)[requirement['action']] = user_selected_utterance(request)
    action = importlib.import_module(
        'bots.{}.actions.{}'.format(request.get('microBotName', request.get('destBotName')), requirement
['action']))
    in_scope_function = getattr(action, requirement['in_scope_function'])
    return in_scope_function(memory)


def is_out_of_scope(memory, request, requirement):
    return not is_in_scope(memory, request, requirement)


def handle_in_scope(memory, request, requirement):
    cdms(memory)[requirement['action']] = user_selected_utterance(request)
    if cdms(memory).get('flow_type') != 'sub_intents':
        reset_sub_intent_state(memory)


def handle_out_of_scope(memory, request, requirement):
    cdms(memory)['out_of_scope'] = True
    action_name = requirement['action']
    cdms(memory)['out_of_scope_msg'] = f'Within {action_name}. Unsupported user selection {cdms(memory)
[action_name]}'
    if requirement.get('in_scope_sub_intents'):
        sub_intent_based_capture.run(memory, request, requirement)
```

intent based capture -

This file has functions like run ,is_in_scope, i s_out_of_scope, handle_in_scope, handle_out_of_scope.

function run has actions variable with is_in_scope, is_out_scope as keys and handlers as execute handlers.

Run function checks whether the requirement that is captured in is in scope or not.If it is in scope, then the **intent_name** variable is updated in cdms memory with the intent name that is captured.If the condition of is_out_scope is satisfies, then the variable out_of_scope is set to true.In addition to that out_of_scope_msg' is also set with message Unsupported Intent received

```
def run(memory, request, requirement):
    actions = {
        is_in_scope: handle_in_scope,
        is_out_of_scope: handle_out_of_scope,
    }
    condition_generator = (condition for condition in actions.keys()
                           if condition(memory, request, requirement))
    actions[next(condition_generator)](memory, request, requirement)


def is_in_scope(memory, request, requirement):
    intent_name, confidence_score, threshold = get_intent_name(request, requirement['in_scope_intents']
['threshold'])
    set_monitoring_topic_info(memory, intent_name, confidence_score, threshold)
    return intent_name in requirement['in_scope_intents']['intents'] and confidence_score >= threshold


def is_out_of_scope(memory, request, requirement):
    return not is_in_scope(memory, request, requirement)


def handle_in_scope(memory, request, requirement):
    intent_name, confidence_score, threshold = get_intent_name(request, requirement['in_scope_intents']
['threshold'])
    set_monitoring_topic_info(memory, intent_name, confidence_score, threshold)
    update_log_attributes(memory, intent_code=intent_name, confidence_score=confidence_score,
                          micro_bot_intent_name=get_sub_intent_name(memory)[0],
                          micro_bot_intent_detail=get_sub_intent_name(memory)[3])
    cdms(memory)['intent_name'] = intent_name
    if requirement.get('in_scope_sub_intents'):
        sub_intent_based_capture.run(memory, request, requirement)


def handle_out_of_scope(memory, request, requirement):
    action_name = requirement['action']
    intent_name, confidence_score, threshold = get_intent_name(request, requirement['in_scope_intents']
['threshold'])
    set_monitoring_topic_info(memory, intent_name, confidence_score, threshold)
    update_log_attributes(memory, intent_code=intent_name, confidence_score=confidence_score,
                          micro_bot_intent_name=get_sub_intent_name(memory)[0],
                          micro_bot_intent_detail=get_sub_intent_name(memory)[3])
    cdms(memory)['out_of_scope'] = True
    cdms(memory)['out_of_scope_msg'] = f'Within {action_name}. Unsupported Intent received {intent_name}' \
        f' with confidence score {confidence_score}'
```

list based capture -

This file has functions like run ,is_in_scope, i s_out_of_scope, handle_in_scope, handle_out_of_scope.

function run has actions variable with is_in_scope, is_out_scope as keys and handlers as execute handlers.

Run function checks whether the requirement that is captured in is in scope or not.If it is in scope, then the **intent_name** variable is updated in cdms memory with the intent name that is captured.If the condition of is_out_scope is satisfies, then the variable out_of_scope is set to true.In addition to that out_of_scope_msg' is also set with message Unsupported Intent received.

```
def run(memory, request, requirement):
    actions = {
        is_in_scope: handle_in_scope,
        is_out_of_scope: handle_out_of_scope,
    }
    condition_generator = (condition for condition in actions.keys()
                           if condition(memory, request, requirement))
    actions[next(condition_generator)](memory, request, requirement)


def is_in_scope(memory, request, requirement):
    cdms(memory)[requirement['action']] = user_selected_utterance(request)
    return cdms(memory)[requirement['action']] in requirement['in_scope_utterances']


def is_out_of_scope(memory, request, requirement):
    return not is_in_scope(memory, request, requirement)


def handle_in_scope(memory, request, requirement):
    cdms(memory)[requirement['action']] = user_selected_utterance(request)
    if cdms(memory).get('flow_type') != 'sub_intents':
        reset_sub_intent_state(memory)


def handle_out_of_scope(memory, request, requirement):
    cdms(memory)['out_of_scope'] = True
    action_name = requirement['action']
    cdms(memory)['out_of_scope_msg'] = f'Within {action_name}. Unsupported user selection {cdms(memory)
[action_name]}'
    if requirement.get('in_scope_sub_intents'):
        sub_intent_based_capture.run(memory, request, requirement)
```

requirement capture - In this file we check whether the capture type that is configured in the .yaml file is list type,intent_capture or funciton_capture. Based on the capture type that is configured,the relevant handler appropriate to capture type is called.

```
def run(memory, request, requirement):
    intent_name, confidence_score, threshold = get_intent_name(request, 0.3)
    set_monitoring_intent_info(memory, intent_name, confidence_score, threshold)
    actions = {
        is_intent_based_capture: handle_intent_based_capture,
        is_list_based_capture: handle_list_based_capture,
        is_function_based_capture: handle_function_based_capture
    }
    condition_generator = (condition for condition in actions.keys()
                           if condition(memory, request, requirement))

    log.info(**log_parameters(memory, request, requirement, 'entry'))
    actions[next(condition_generator)](memory, request, requirement)
    log.info(**log_parameters(memory, request, requirement, 'exit'))

    if str(cdms(memory).get('out_of_scope', False)).capitalize() == 'True':
        log.info(**log_parameters(memory, request, requirement, 'out_of_scope'))
        raise CdmsException('out_of_scope', cdms(memory).get('out_of_scope_msg', 'out_of_scope_others'))


def is_intent_based_capture(memory, request, requirement):
    return requirement.get('capture_type') == 'intent'


def is_list_based_capture(memory, request, requirement):
    return requirement.get('capture_type') == 'list'


def is_function_based_capture(memory, request, requirement):
    return requirement.get('capture_type') == 'function'


def handle_intent_based_capture(memory, request, requirement):
    intent_based_capture.run(memory, request, requirement)


def handle_list_based_capture(memory, request, requirement):
    list_based_capture.run(memory, request, requirement)


def handle_function_based_capture(memory, request, requirement):
    function_based_capture.run(memory, request, requirement)
```

sub intent based capture -

```
def run(memory, request, requirement):
    actions = {
        is_in_scope: handle_in_scope,
        is_intent_based_out_of_scope: handle_intent_based_out_of_scope,
        is_not_intent_based_out_of_scope: handle_not_intent_based_out_of_scope
    }
    condition_generator = (condition for condition in actions.keys()
                           if condition(memory, request, requirement))
    actions[next(condition_generator)](memory, request, requirement)


def is_in_scope(memory, request, requirement):
    in_scope_sub_intent_list = get_in_scope_sub_intents(memory, requirement)

    intent_name, confidence_score, threshold, _ = get_sub_intent_name(memory)
    intent_type = sub_intent_type(requirement,intent_name)
    allowed_sub_intent_count = int(cdms(memory).get(f'allowed_{intent_type}_sub_intent_count', 0))
    return in_scope_sub_intent_list and \
           intent_name in in_scope_sub_intent_list and \
           confidence_score >= threshold and \
           allowed_sub_intent_count < config.get_int(f'askamex.cdms.allowed.{intent_type}.sub.intent.count')


def is_intent_based_out_of_scope(memory, request, requirement):
    return requirement.get('capture_type') == 'intent' and not is_in_scope(memory, request, requirement)


def is_not_intent_based_out_of_scope(memory, request, requirement):
    return requirement.get('capture_type') != 'intent' and not is_in_scope(memory, request, requirement)


def handle_in_scope(memory, request, requirement):
    in_scope_sub_intent_list = get_in_scope_sub_intents(memory, requirement)

    intent_name, confidence_score, threshold, _ = get_sub_intent_name(memory)
    intent_type = sub_intent_type(requirement,intent_name)
    allowed_sub_intent_count = int(cdms(memory).get(f'allowed_{intent_type}_sub_intent_count', 0))
    in_scope_sub_intent_list.remove(intent_name)

    cdms(memory)['out_of_scope'] = False
    cdms(memory)['sub_intent_name'] = intent_name
    cdms(memory)['sub_intent_type'] = intent_type
    cdms(memory)['in_scope_sub_intent_list'] = str(in_scope_sub_intent_list)
    cdms(memory)[f'allowed_{intent_type}_sub_intent_count'] = allowed_sub_intent_count + 1
    set_monitoring_sub_intent_info(memory, intent_name, confidence_score, threshold)


def handle_intent_based_out_of_scope(memory, request, requirement):
    pass


def handle_not_intent_based_out_of_scope(memory, request, requirement):
    cdms(memory)['out_of_scope'] = True


def sub_intent_type(requirement, sub_intent):
    return next((in_scope_sub_intent.get('type')
                for in_scope_sub_intent in requirement.get('in_scope_sub_intents') if
                sub_intent in in_scope_sub_intent.get('intents')), None)


def _sub_intents_from_requirements(requirement):
    intents = [in_scope_sub_intent.get('intents')
               for in_scope_sub_intent in requirement.get('in_scope_sub_intents')]
    return list(itertools.chain(*intents))


def _sub_intents_from_memory(memory):
    return ast.literal_eval(cdms(memory).get('in_scope_sub_intent_list', str([])))


def get_in_scope_sub_intents(memory, requirement):
    return _sub_intents_from_memory(memory) or _sub_intents_from_requirements(requirement)
```

util - This file consists of different functions.

reset_sub_intent_state function is for resetting the memory variables.

log_parameters function is to return response with entry log parameters, exit log parameters and out of scope parameters.

entry log parameters and exit log parameters functions define the response of log attributes while entering the requirement captures and exiting the requirement capture.

```
def reset_sub_intent_state(memory):
    cdms(memory)['in_scope_sub_intent_list'] = str([])
    cdms(memory)['sub_intent_name'] = ''
    cdms(memory)['sub_intent_type'] = ''
    cdms(memory)['allowed_faq_sub_intent_count'] = 0
    cdms(memory)['allowed_entity_sub_intent_count'] = 0


def log_parameters(memory, request, requirement, log_type):
    return {
        'entry': entry_log_parameters,
        'exit': exit_log_parameters,
        'out_of_scope': out_of_scope_log_parameters
    }.get(log_type)(memory, request, requirement)


def out_of_scope_log_parameters(memory, request, requirement):
    action_name = requirement['action']
    capture_type = requirement.get('capture_type')
    update_log_attributes(memory, chat_handling_time=get_chat_handling_time(current_time_utc(),
                                                                            log_attribute_value(memory,

'chat_start_time')))
    return {**get_log_attributes(memory, request), **{
        'event_name': f'{action_name}.{capture_type}_based_requirement_capture',
        'action': f'{Constants.ACTION_END} {action_name}',
        'desc': f'Exiting {action_name}.{capture_type}_based_requirement_capture',
        'result': Constants.LOG_SUCCESS_RESULT,
        'reason': 'out_of_scope',
        'chat_transcript_text': user_selected_utterance(request)
    }}


def entry_log_parameters(memory, request, requirement):
    action_name = requirement['action']
    capture_type = requirement.get('capture_type')

    return {**get_log_attributes(memory, request), **{
        'event_name': f'{action_name}.{capture_type}_based_requirement_capture',
        'action': f'{Constants.ACTION_START} {action_name}',
        'desc': f'Exiting {action_name}.{capture_type}_based_requirement_capture',
        'result': Constants.LOG_SUCCESS_RESULT,
        'reason': Constants.LOG_SUCCESS_REASON,
        'chat_transcript_text': user_selected_utterance(request)
    }}


def exit_log_parameters(memory, request, requirement):
    action_name = requirement['action']
    capture_type = requirement.get('capture_type')

    return {**get_log_attributes(memory, request), **{
        'event_name': f'{action_name}.{capture_type}_based_requirement_capture',
        'action': f'{Constants.ACTION_END} {action_name}',
        'desc': f'Exiting {action_name}.{capture_type}_based_requirement_capture',
        'result': Constants.LOG_SUCCESS_RESULT,
        'reason': Constants.LOG_SUCCESS_REASON,
        'chat_transcript_text': user_selected_utterance(request)
    }}
```

REQUIREMENTS V2

non topic based capture -

```
def run(params):
    actions = {
        is_in_scope: handle_in_scope,
        is_out_of_scope: handle_out_of_scope
    }

    inner_params = namedtuple('params', ['memory', 'request', 'requirements', 'action',
                                         'topic_based', 'sub_intent_requirements'])
    condition_generator = (condition for condition in actions.keys() if condition(params))
    actions[next(condition_generator)](inner_params(params.memory, params.request, params.requirements,
                                                    params.action, params.topic_based, params.
sub_intent_requirements))
    if str(cdms(params.memory).get('out_of_scope', False)).capitalize() == 'True':
        raise CdmsException('out_of_scope', cdms(params.memory).get('out_of_scope_msg',
'out_of_scope_others'))


def is_in_scope(params):
    in_scope = False
    in_scope_dict = {
        'function': _is_function_based_in_scope,
        'list': _is_list_based_in_scope,
        'intent': _is_intent_based_in_scope
    }
    for requirement in params.requirements:
        inner_params = namedtuple('inner_params', ['memory', 'request', 'requirement', 'action'])
        inner_params = inner_params(params.memory, params.request, requirement, params.action)
        log.info(**log_parameters(params.memory, params.request, requirement, params.action, 'entry'))
        if in_scope_dict.get(requirement.get('capture_type'))(inner_params):
            cdms(params.memory)['in_scope_requirements'] = requirement
            in_scope = True
    return in_scope


def handle_in_scope(params):
    cdms(params.memory)['out_of_scope'] = False
    in_scope_dict = {
        'function': _handle_function_based_in_scope,
        'list': _handle_list_based_in_scope,
        'intent': _handle_intent_based_in_scope
    }
    requirement = cdms(params.memory).get('in_scope_requirements')
    return in_scope_dict.get(requirement.get('capture_type'))(params)


def is_out_of_scope(params):
    return not is_in_scope(params)


def handle_out_of_scope(params):
    requirement = params.requirements[0]
    cdms(params.memory)['out_of_scope'] = True
    out_of_scope_dict = {
        'function': _handle_function_based_out_of_scope,
        'list': _handle_list_based_out_of_scope,
        'intent': _handle_intent_based_out_of_scope
    }
    log.info(**log_parameters(params.memory, params.request, requirement,
                              params.action, 'exit'))
    out_of_scope_dict.get(requirement.get('capture_type'))(params)
    if params.sub_intent_requirements.get('in_scope_sub_intents'):
        sub_intent_based_capture.run(params)


def _is_function_based_in_scope(params):
    cdms(params.memory)[params.action] = user_selected_utterance(params.request)
    action = importlib.import_module(
        'bots.{}.actions.{}'.format(
            params.request.get('microBotName', params.request.get('destBotName')), params.action))
    in_scope_function = getattr(action, params.requirement['in_scope_function'])
    return in_scope_function(params.memory)
```

```
def _is_list_based_in_scope(params):
    cdms(params.memory)[params.action] = user_selected_utterance(params.request)
    return cdms(params.memory)[params.action] in params.requirement['in_scope_utterances']


def _is_intent_based_in_scope(params):
    intent_name, confidence_score, threshold = \
        get_intent_name(params.request, params.requirement['in_scope_intents']['threshold'])
    set_monitoring_intent_info(params.memory, intent_name, confidence_score, threshold)
    return intent_name in params.requirement['in_scope_intents']['intents'] and confidence_score >= threshold


def _handle_function_based_in_scope(params):
    cdms(params.memory)[params.action] = user_selected_utterance(params.request)
    if cdms(params.memory).get('flow_type') != 'sub_intents':
        reset_sub_intent_state(params.memory)


def _handle_list_based_in_scope(params):
    cdms(params.memory)[params.action] = user_selected_utterance(params.request)
    if cdms(params.memory).get('flow_type') != 'sub_intents':
        reset_sub_intent_state(params.memory)


def _handle_intent_based_in_scope(params):
    requirement = cdms(params.memory).get('in_scope_requirements')
    intent_name, confidence_score, threshold = \
        get_intent_name(params.request, requirement['in_scope_intents']['threshold'])
    set_monitoring_intent_info(params.memory, intent_name, confidence_score, threshold)
    update_log_attributes(params.memory, intent_code=intent_name, confidence_score=confidence_score,
                          micro_bot_intent_name=get_sub_intent_name(params.memory)[0],
                          micro_bot_intent_detail=get_sub_intent_name(params.memory)[3])
    cdms(params.memory)['intent_name'] = intent_name
    cdms(params.memory)[params.action] = intent_name


def _handle_function_based_out_of_scope(params):
    cdms(params.memory)['out_of_scope_msg'] = \
        f'Within {params.action}. Unsupported user selection {cdms(params.memory)[params.action]}'


def _handle_list_based_out_of_scope(params):
    cdms(params.memory)['out_of_scope_msg'] = \
        f'Within {params.action}. Unsupported user selection {cdms(params.memory)[params.action]}'


def _handle_intent_based_out_of_scope(params):
    intent_name, confidence_score, threshold = \
        get_intent_name(params.request, params.requirement['in_scope_intents']['threshold'])
    set_monitoring_intent_info(params.memory, intent_name, confidence_score, threshold)
    update_log_attributes(params.memory, intent_code=intent_name, confidence_score=confidence_score,
                          micro_bot_intent_name=get_sub_intent_name(params.memory)[0],
                          micro_bot_intent_detail=get_sub_intent_name(params.memory)[3])
    cdms(params.memory)['out_of_scope_msg'] = f'Within {params.action}. Unsupported Intent received
{intent_name}' \
        f' with confidence score {confidence_score}'
```

requirement capture -This file validates whether the requirement capture is topic based or non topic based.Based on topic based or non topic based, it executes the appropriate handler.

```
def run(memory, request, requirements, action_name):
    params = namedtuple('params', ['memory', 'request', 'requirements', 'action'])

    intent_name, confidence_score, threshold = get_intent_name(request, 0.3)
    set_monitoring_intent_info(memory, intent_name, confidence_score, threshold)

    actions = {
        is_non_topic_based_requirements: handle_non_topic_based_requirements,
        is_topic_based_requirements: handle_topic_based_requirements
    }

    condition_generator = (condition for condition in actions.keys()
                           if condition(params(memory, request, requirements, action_name)))

    log.info(**log_parameters(memory, request, requirements, action_name, 'entry'))
    actions[next(condition_generator)](params(memory, request, requirements, action_name))
    log.info(**log_parameters(memory, request, requirements, action_name, 'exit'))

    if str(cdms(memory).get('out_of_scope', False)).capitalize() == 'True':
        log.info(**log_parameters(memory, request, requirements, action_name, 'out_of_scope'))
        raise CdmsException('out_of_scope', cdms(memory).get('out_of_scope_msg', 'out_of_scope_others'))

    try:
        del cdms(memory)['in_scope_requirements']
    except KeyError:
        pass


def is_non_topic_based_requirements(params):
    return params.requirements.get('non_topic_based_requirements')


def is_topic_based_requirements(params):
    return params.requirements.get('topic_based_requirements')


def handle_non_topic_based_requirements(params):
    inner_params = namedtuple('inner_params', ['memory', 'request', 'requirements', 'action',
                                               'topic_based', 'sub_intent_requirements'])
    non_topic_based_capture.run(
        inner_params(params.memory, params.request,
                     params.requirements.get('non_topic_based_requirements'),
                     params.action, False,
                     params.requirements.get('sub_intent_based_requirements', [{}])[0]))


def handle_topic_based_requirements(params):
    inner_params = namedtuple('inner_params', ['memory', 'request', 'requirement', 'action',
                                               'topic_based', 'sub_intent_requirements'])
    topic_based_capture.run(
        inner_params(params.memory, params.request,
                     params.requirements.get('topic_based_requirements')[0], params.action, True,
                     params.requirements.get('sub_intent_based_requirements', [{}])[0]))
```

sub intent based capture -

```
def sub_intent_data(params):
    request_data = params.request
    memory = params.memory
    sub_intent_message = {
        "secondaryConversationId": request_data['secondaryConversationId'],
        "interactionId": request_data.get('interactionId'),
        "botName": cdms(memory).get('main_bot_name'),
        "messageId": cdms(memory).get('message_id'),
        "chatSessionId": request_data.get('chatSessionId'),
        "cUserId": request_data.get('request', {}).get('userId') or request_data.get('userId'),
        "channelId": request_data.get('request', {}).get('channelId') or request_data.get('channelId'),
        "message": {
            "messageid": cdms(memory).get('message_id')
        },
        "microBotName": request_data.get('microBotName') or request_data.get('destBotName'),
        "microBotIntentName": get_sub_intent_name(memory)[0],
        "microBotIntent": get_sub_intent_name(memory)[3]
    }
    return sub_intent_message


def run(params):
    actions = {
        is_in_scope: handle_in_scope,
        is_topic_based_out_of_scope: handle_topic_based_out_of_scope,
        is_non_topic_based_out_of_scope: handle_non_topic_based_out_of_scope
    }
    condition_generator = (condition for condition in actions.keys()
                           if condition(params))
    actions[next(condition_generator)](params)


def is_in_scope(params):
    in_scope_sub_intent_list = get_in_scope_sub_intents(params.memory, params.sub_intent_requirements)

    intent_name, confidence_score, threshold, _ = get_sub_intent_name(params.memory)
    intent_type = _sub_intent_type(params.sub_intent_requirements, intent_name)
    allowed_sub_intent_count = int(cdms(params.memory).get(f'allowed_{intent_type}_sub_intent_count', 0))
    return in_scope_sub_intent_list and \
           intent_name in in_scope_sub_intent_list and \
           confidence_score >= threshold and \
           allowed_sub_intent_count < config.get_int(f'askamex.cdms.allowed.{intent_type}.sub.intent.count')


def is_topic_based_out_of_scope(params):
    return params.topic_based and not is_in_scope(params)


def handle_topic_based_out_of_scope(params):
    intent_name, confidence_score, threshold, _ = get_sub_intent_name(params.memory)
    set_monitoring_sub_intent_info(params.memory, intent_name, confidence_score, threshold)
    di_util.ingest(**{'target': 'transcript_kafka_topic', 'message': sub_intent_data(params)})


def is_non_topic_based_out_of_scope(params):
    return not params.topic_based and not is_in_scope(params)


def handle_in_scope(params):
    in_scope_sub_intent_list = get_in_scope_sub_intents(params.memory, params.sub_intent_requirements)

    intent_name, confidence_score, threshold, _ = get_sub_intent_name(params.memory)
    intent_type = _sub_intent_type(params.sub_intent_requirements, intent_name)
    allowed_sub_intent_count = int(cdms(params.memory).get(f'allowed_{intent_type}_sub_intent_count', 0))
    in_scope_sub_intent_list.remove(intent_name)

    cdms(params.memory)['out_of_scope'] = False
    cdms(params.memory)['sub_intent_name'] = intent_name
    cdms(params.memory)['sub_intent_type'] = intent_type
    cdms(params.memory)['in_scope_sub_intent_list'] = str(in_scope_sub_intent_list)
    cdms(params.memory)[f'allowed_{intent_type}_sub_intent_count'] = allowed_sub_intent_count + 1
    set_monitoring_sub_intent_info(params.memory, intent_name, confidence_score, threshold)

    di_util.ingest(**{'target': 'transcript_kafka_topic', 'message': sub_intent_data(params)})


def handle_non_topic_based_out_of_scope(params):
    cdms(params.memory)['out_of_scope'] = True
    intent_name, confidence_score, threshold, _ = get_sub_intent_name(params.memory)
    set_monitoring_sub_intent_info(params.memory, intent_name, confidence_score, threshold)
    di_util.ingest(**{'target': 'transcript_kafka_topic', 'message': sub_intent_data(params)})


def _sub_intent_type(requirement, sub_intent):
    return next((in_scope_sub_intent.get('type')
                for in_scope_sub_intent in requirement.get('in_scope_sub_intents') if
                sub_intent in in_scope_sub_intent.get('intents')), None)


def _sub_intents_from_requirements(requirement):
    intents = [in_scope_sub_intent.get('intents')
               for in_scope_sub_intent in requirement.get('in_scope_sub_intents')]
    return list(itertools.chain(*intents))


def _sub_intents_from_memory(memory):
    return ast.literal_eval(cdms(memory).get('in_scope_sub_intent_list', str([])))


def get_in_scope_sub_intents(memory, requirement):
    return _sub_intents_from_memory(memory) or _sub_intents_from_requirements(requirement)
```

topic based capture - Here we are validating whether the intent that is captured is in scope intents that are configured to the bot.If the captured intent does not belong to in scope intents,then out of scope variables are updated in cdms memory.

```
def run(params):
    actions = {
        is_in_scope: handle_in_scope,
        is_out_of_scope: handle_out_of_scope,
    }
    condition_generator = (condition for condition in actions.keys()
                           if condition(params))
    actions[next(condition_generator)](params)


def is_in_scope(params):
    intent_name, confidence_score, threshold = get_intent_name(params.request, params.requirement
['in_scope_intents']['threshold'])

    type = params.request.get('request', {}).get('type', params.request.get('type'))

    if type == 'composite':
        sub_type = params.request.get('request', {}).get('sub_type', params.request.get('sub_type'))
        intent_name, confidence_score, threshold = sub_type, 1, 1

    if params.request.get('request', {}).get('requestType', '') == 'intent':
        intent_name = params.request.get('request', {}).get('context', {}).get('intent')
        confidence_score, threshold = 1, 1

    return intent_name in params.requirement['in_scope_intents']['intents'] and confidence_score >= threshold


def is_out_of_scope(params):
    return not is_in_scope(params)


def handle_in_scope(params):
    intent_name, confidence_score, threshold = get_intent_name(params.request, params.requirement
['in_scope_intents']['threshold'])
    update_log_attributes(params.memory, intent_code=intent_name, confidence_score=confidence_score,
                          micro_bot_intent_name=get_sub_intent_name(params.memory)[0],
                          micro_bot_intent_detail=get_sub_intent_name(params.memory)[3])
    cdms(params.memory)['intent_name'] = intent_name
    set_monitoring_topic_info(params.memory, intent_name, confidence_score, threshold)
    cdms(params.memory)['out_of_scope'] = False
    if params.sub_intent_requirements.get('in_scope_sub_intents'):
        sub_intent_based_capture.run(params)


def handle_out_of_scope(params):
    action_name = params.action
    intent_name, confidence_score, threshold = get_intent_name(params.request, params.requirement
['in_scope_intents']['threshold'])
    update_log_attributes(params.memory, intent_code=intent_name, confidence_score=confidence_score,
                          micro_bot_intent_name=get_sub_intent_name(params.memory)[0],
                          micro_bot_intent_detail=get_sub_intent_name(params.memory)[3])
    set_monitoring_topic_info(params.memory, intent_name, confidence_score, threshold)
    cdms(params.memory)['out_of_scope'] = True
    cdms(params.memory)['out_of_scope_msg'] = f'Within {action_name}. Unsupported Intent received
{intent_name}' \
        f' with confidence score {confidence_score}'
```

NOTIFIERS

NOTIFY

```
def error_callback(retry_state):
```

```python
    raise CdmsException(Constants.SEND_MSG_FAILED_ERROR_CODE,
                        'Failed to send message::' + str(retry_state.outcome.result()))


def retry_condition(response):
    return response['statusMsg'].lower() == Constants.MAIN_BOT_SEND_MESSAGE_FAILURE_CODE


async def send_message(**kwargs):
    template, request, retry_count, attachment, bot_config, memory, template_type, scenario = \
        kwargs.get('template'), kwargs['request'], \
        kwargs.get('retry', 1), kwargs.get('attachment'), kwargs['bot_config'], \
        kwargs.get('memory'), kwargs.get('template_type'), kwargs.get('scenario')
    wait_time = int(bot_config['notify.sleep.seconds'])

    @retry(retry=retry_if_result(retry_condition), stop=stop_after_attempt(3),
           wait=wait_fixed(wait_time),
           retry_error_callback=error_callback)
    async def send_message_helper(**kwargs):
        try:
            request_data = await request.json()
            header = request.headers
            json_data = await notification_message(**kwargs)
            log_data = {**get_log_attributes(memory, request_data,  header), **{
                'event_name': 'send_message',
                'utterance': template,
                'text_send_to_receive_api_with_retry': f'payload:{str(json_data)}'
            }}

            response = await aio_http_post(config.get('askamex.cdms.receive.api.url'),
                                           json_data, headers(request_data, header),
                                           config.get_int('askamex.cdms.receive.api.timeout'),
                                           log_params=log_data)

            return response
        except Exception as error:
            log.error(**{'error': "send_message_helper Exception", 'reason': str(error)})
            raise error

    return await send_message_helper(**kwargs)


async def notification_message(**kwargs):
    template, template_name, request, config_message_type, attachment, bot_config, memory, template_type,
scenario = \
        kwargs.get('template'), kwargs.get('template_name'), kwargs['request'], kwargs.get('message_type'), \
        kwargs.get('attachment'), kwargs['bot_config'], \
        kwargs.get('memory'), kwargs.get('template_type'), kwargs.get('scenario')

    template_message_type = ''
    request_data = await request.json()
    header = request.headers
    channel_id = request_data.get('request', {}).get('channelId', request_data.get('channelId'))

    if template_name:
        if scenario:
            cdms(memory)['scenario'] = scenario
        template, attachment, template_message_type = await get_template(correlation_id(request),
                                                                         template_name,
                                                                         bot_config['botname'],
                                                                         channel_id,
                                                                         header['X-AXP-Locale'], memory,
                                                                         bot_config.get
('template_api_version', 'v1'),
                                                                         template_type)

    message_type = config_message_type or template_message_type
    current_time = current_time_utc()
    message_id = bot_config['message_id.prefix'] + str(uuid.uuid4())
    update_log_attributes(memory, message_id=message_id, time_stamp=current_time)
    return {
        'mainBotName': header.get('X-AXP-Main-BotName', bot_config.get('main.botname')),
        'sourceBotName': bot_config['botname'],
        'userId': request_data.get('request', {}).get('userId', request_data.get('userId')),
        'userType': request_data.get('userType'),
        'conversationId': request_data.get('request', {}).get('conversationId', request_data.get
('conversationId')),
        'secondaryConversationId': request_data['secondaryConversationId'],
        'interactionId': request_data.get('interactionId'),
        'chatSessionId': request_data.get('chatSessionId'),
        'channelId': channel_id,
        'clientMessageId': message_id,
        'requestType': request_data.get('request', {}).get('requestType', 'message'),
        'message': {
            'messageid': message_id,
            'conversationid': request_data.get('request', {}).get('conversationId',
                                                                   request_data.get('conversationId')),
            'userid': bot_config['bot.user_id'],
            'type': message_type_by_channel(message_type, channel_id),
            'content': {
                'desc': template
            },
            'attachment': attachment_by_type(message_type, attachment, channel_id),
            'create_time': current_time,
            'update_time': current_time
        },
        'microBotMemory': {
            'conversationStringContext': {
                'journeys': bot_config['journey']
            }
        }
    }


def message_type_by_channel(cdms_message_type, channel_id='mobile'):
    message_types = [
        {'richtext': {
            'web': 'text',
            'mobile': 'richtext'}}
    ]
    return next((msg_type.get(cdms_message_type).get(channel_id, cdms_message_type)
                 for msg_type in message_types if msg_type.get(cdms_message_type)), cdms_message_type)


def attachment_by_type(message_type, attachment, channel_id):
    message_type = message_type_by_channel(message_type, channel_id)
    if message_type == 'text':
        return {}
    elif message_type == 'single_choice':
        return {
            "question_id": generate_question_id(channel_id),
            "options": attachment
        }
    elif message_type == 'richtext':
        return {
            "text_keys": attachment
        }
    else:
        return {}


def headers(request_data, header):
    return {
        Constants.CONTENT_TYPE_KEY: Constants.CONTENT_TYPE_VALUE,
        'transid': str(uuid.uuid4()),
        'channel': request_data.get('request', {}).get('channelId', request_data.get('channelId')),
        'X-AXP-Locale': header['X-AXP-Locale']
    }
```

RELINQUISH

```
async def relinquish_control(**kwargs):
    template, request, action, reason, memory, retry, bot_config = \
        kwargs['template'], kwargs['request'], kwargs['action'], \
        kwargs['reason'], kwargs.get('memory'), kwargs.get('retry', 1), kwargs['bot_config']

    try:
        await asyncio.sleep(int(bot_config['notify.sleep.seconds']))
        request_data = await request.json()
        header = request.headers
        kwargs['request'] = request_data
        kwargs['header'] = header
        request_payload = relinquish_payload(**kwargs)

        log_data = {**get_log_attributes(memory, request_data), **{
            'event_name': 'relinquish_control',
            'utterance': template,
            'text_send_to_relinquish_api_with_retry': f'retry count:{str(retry)} '
                                                      f'and payload:{str(request_payload)}'
        }}

        response = await aio_http_post_with_headers(config.get('askamex.cdms.conversation.control.api.url'),
                                                    request_payload,
                                                    headers(header, request_data),
                                                    config.get_int('askamex.cdms.conversation.control.api.
timeout'),
                                                    log_params=log_data)

        if retry == 1:
            update_log_attributes(memory, chat_handling_time=get_chat_handling_time(current_time_utc(),
                                                                                   log_attribute_value
(memory,

'chat_start_time')))

        log.info(**{**get_log_attributes(memory, request_data), **{
            'event_name': 'relinquish_control',
            'utterance': template,
            'relinquished_status': str(response),
            'retry_count': retry,
            'relinquish_reason': reason,
            'is_valid_response': is_valid_response(response)
        }})

        if not is_valid_response(response):
            inputs = {'request': request,
                      'template': template,
                      'action': action,
                      'reason': reason,
                      'memory': memory,
                      'retry': retry,
                      'bot_config': bot_config,
                      'header': kwargs.get('header'),
                      'intent': kwargs.get('intent'),
                      'destination_bot': kwargs.get('destination_bot')
                      }
            if retry < int(bot_config['notify.retires']):
                retry += 1
                inputs.update({'retry': retry})
                await relinquish_control(**inputs)
            elif retry >= int(bot_config['notify.retires']) and reason == 'callback':
                inputs.update({'reason': 'transfer', 'retry': 1})
                await relinquish_control(**inputs)

        if not memory.get('memory_updated'):
            if reason != 'callback':
                memory['bot_context']['conversationStringContext'] = {}
                await update_memory(memory, False)
            else:
                await update_memory(memory)
            memory['memory_updated'] = True
    except Exception as error:
        log.error(**{'error': "relinquish_control Exception", 'reason': str(error)})


def relinquish_payload(**kwargs):
    template, action, reason, memory, bot_config, destination_bot, intent, request, header = \
        kwargs['template'], kwargs['action'], \
        kwargs['reason'], kwargs.get('memory'), kwargs['bot_config'], \
        kwargs.get('destination_bot'), kwargs.get('intent'), kwargs['request'], kwargs['header']

    channel_id = request.get('request', {}).get('channelId', request.get('channelId'))
    message_id = bot_config['message_id.prefix'] + str(uuid.uuid4())
    payload = {
        'mainBotName': header.get('X-AXP-Main-BotName', bot_config.get('main.botname')),
        'sourceBotName': bot_config['botname'],
        'destBotName': destination_bot_name(channel_id, destination_bot),
        'userId': request.get('request', {}).get('userId', request.get('userId')),
        'userType': request.get('userType'),
        'channelId': channel_id,
        'action': action,
        'reasonCode': reason,
        'reasonText': template,
        'lastMessageId': message_id,
        'conversationId': request.get('request', {}).get('conversationId', request.get('conversationId')),
        'secondaryConversationId': request['secondaryConversationId'],
        'interactionId': request.get('interactionId'),
        'chatSessionId': request.get('chatSessionId'),
        'messageId': message_id,
        'mainBotMemory': request.get('mainBotMemory'),
        'miscInfo': get_misc_info(memory),
        'botControlData': get_bot_handler_data(request, memory['bot_context']['conversationStringContext'],
                                               bot_config, intent)
    }
    if bot_config.get('send_intent_info'):
        payload.update({
            'intentCode': bot_config.get('journey'),
        })
    return payload


def headers(header, request_data):
    return {"Content-Type": "application/json",
            "transid": "{}".format(str(uuid.uuid4())),
            "channel": request_data.get('request', {}).get('channelId', request_data.get('channelId')),
            "X-AXP-Locale": header['X-AXP-Locale'],
            "X-AXP-User-Legacy-Authorization": header['X-AXP-User-Legacy-Authorization']
            }


def get_bot_handler_data(request, memory, bot_config, intent):
    account = ast.literal_eval(memory.get('selected_card_account', {})) \
        if memory.get('selected_card_account') else {}
    micro_bot_memory = copy.deepcopy(memory)
    micro_bot_memory['next_action'] = ""
    return {
        "intentName": intent,
        "confidenceScore": 1.0,
        "userJwt": request.get('mainBotMemory', {}).get('conversationStringContext', {}).get('userJwt'),
        "accountToken": account.get('account_token'),
        "factName": bot_config.get('authn_fact_name'),
        "factValue": memory.get('selected_replace_reason_code'),
        "microBotMemory": str(micro_bot_memory)
    }


def is_valid_response(response):
    return response['status_code'] == 200 and response['response']['status'] == 'SUCCESS'
```

```
def destination_bot_name(channel_id, destination_from_config=None):
    channel_based_destination = config.get('askamex.cdms.destination.bot.web') \
        if channel_id == 'web' else config.get('askamex.cdms.destination.bot.mobile')
    return destination_from_config if destination_from_config else channel_based_destination
```

PROCESSORS

ACTION PROCESSOR

process_async_bot_actions

This is where the actual bot flow starts.This function will decide which action to be performed based on the presence of next_action in the request data. If the next action is present in the request,then the mentioned action will be executed or else the intent_received file of the action will be called.

After deciding which action to be performed,it executes capture_entities,capture requirements and execute function of that action

Every file in actions folder consists of capture requirements,capture entitities and execute functions in common.

```
async def process_async_bot_actions(handle, memory, request, request_data):
    action = importlib.import_module(
        'bots.{}.actions.{}'.format(request_data.get('microBotName'),
                                    next_action(memory) if next_action(memory) else 'intent_received'))
    capture_entities = getattr(action, 'capture_entities', None)
    capture_requirements = getattr(action, 'capture_requirements')
    execute = getattr(action, 'execute')
    if capture_entities:
        capture_entities(memory, request_data)
    await capture_requirements(memory, request_data)
    future = asyncio.ensure_future(execute(memory, request))
    future.add_done_callback(handle)

async def process_requirement_capture(memory, request_data):
    action = importlib.import_module(
        'bots.{}.actions.{}'.format(request_data.get('destBotName'),
                                    next_action(memory) if next_action(memory) else 'intent_received'))
    capture_requirements = getattr(action, 'capture_requirements')
    await capture_requirements(memory, request_data)


async def process_convo_control_bot_actions(handle, memory, request, request_data):
    action = importlib.import_module(
        'bots.{}.actions.{}'.format(request_data.get('destBotName'),
                                    next_action(memory) if next_action(memory) else 'intent_received'))
    execute = getattr(action, 'execute')
    future = asyncio.ensure_future(execute(memory, request))
    future.add_done_callback(handle)


async def process_convo_control_bot_transfer_actions(handle, memory, request, request_data):
    action = importlib.import_module(
        'bots.{}.actions.{}'.format(request_data.get('destBotName'),
                                    next_action(memory) if next_action(memory) else 'intent_received'))
    capture_requirements = getattr(action, 'capture_requirements')
    execute = getattr(action, 'execute')
    await capture_requirements(memory, request_data)
    future = asyncio.ensure_future(execute(memory, request))
    future.add_done_callback(handle)


async def process_sub_intent_bot_actions(handle, memory, request, request_data):
    action = importlib.import_module(
        'bots.{}.{}.{}.actions.{}'.format(request_data['microBotName'], cdms(memory)['flow_type'],
                                          cdms(memory)['bot_type'],
                                          next_sub_intent_action(memory) if next_sub_intent_action(memory)
                                          else 'intent_received'))
    capture_requirements = getattr(action, 'capture_requirements', None)
    if capture_requirements:
        await action.capture_requirements(memory, request_data)
    future = asyncio.ensure_future(action.execute(memory, request))
    future.add_done_callback(handle)
```

ENTITY PROCESSOR

This file is related to entities.If the override variable is configured with True in .yaml file, then the entity variable is updated with None value in cdms memory.Next entity value is assigned to entity name in cdms.

Next the entity processor function is executed after entity is captured.

```
def extract_entities(memory, request, action_config_path):
    entity_requirements = get_entity_requirements(action_config_path)
    for entity_requirement in entity_requirements:
        if entity_requirement.get('override'):
            cdms(memory)[entity_requirement['state']] = None

        for entity_type in entity_requirement['types']:
            if not cdms(memory).get(entity_requirement['state']):
                cdms(memory)[entity_requirement['state']] = get_entity_value(request,
                                                                              entity_type['name'],
                                                                              entity_type['threshold'])

        if entity_requirement.get('post_processor'):
            post_processor = importlib.import_module('bots.{}.entities.{}'
                                                     .format(request.get('microBotName'),
                                                             entity_requirement['post_processor']))
            post_processor_execute = getattr(post_processor, 'execute')
            post_processor_execute(memory, request)


def get_entity_requirements(action_config_path):
    return get_action_config(action_config_path)['entities']
```

REQUIREMENT CAPTURE

```
async def capture_config_based_requirements(memory, request, action_config_path):
    action_config = await async_get_action_config(action_config_path)
    return run(memory, request, action_config.get('requirements'))


async def capture_config_based_requirements_v2(memory, request, action_config_path):
    action_config = await async_get_action_config(action_config_path)
    return run_v2(memory, request, action_config.get('requirements'), action_config['name'])
```

RESOURCES

CONVERSATION CONTROL

```
@lazy_load_config(app_name='askamex-cdms')
@authorize_resource
async def conversation_control(request):
    request_data, request_header = await request.json(), request.headers
    log.info(**{'correlation_id': correlation_id(request),
                'app_name': Constants.APP_NAME,
                'event_name': 'conversation_control',
                'action': f'{Constants.ACTION_START} conversation_control',
                'desc': 'entering_conversation_control',
                'result': Constants.LOG_SUCCESS_RESULT,
                'reason': Constants.LOG_SUCCESS_REASON,
                'micro_bot_name': get_destination_bot(request_data),
                'bot_control_data': str(get_bot_control_data(request_data)),
                'user_id': user_id(request_data)
                })

    def handle(fut):
        try:
            fut.result()
        except CdmsException as cdms_exception:
            asyncio.get_event_loop().create_task(exception_handler(cdms_exception, request, memory))
    try:
        memory = ''
        memory = await get_memory_handler(request, request_data, request_header)
        if not log_attribute_value(memory, Constants.TRACE_ID):
            update_log_attributes(memory, correlation_id=correlation_id(request), trace_id=trace_id(),
                                  app_name=Constants.APP_NAME, channel_name=channel_name(request_data),
                                  locale=locale(request), micro_bot_name=get_destination_bot(request_data),
                                  chat_session_id=chat_session_id(request_data),
                                  cardmember_id=user_id(request_data),
                                  user_id=user_id(request_data), user_type=user_type(request_data),
                                  conversation_id=conversation_id(request_data),
                                  interaction_id=interaction_id(request_data),
                                  secondary_conversation_id=secondary_conversation_id(request_data),
                                  chat_start_time=current_time_utc())

        if is_session_invalid(memory, config.get_int('askamex.cdms.card.replacement.flow.timeout.seconds')):
            asyncio.get_event_loop().create_task(timeout_handler(request, memory))
            return acknowledgment(request_data, memory, True)
        if not request_data['reasonCode'] == 'callback_return':
            memory['bot_context']['conversationStringContext'].update(ast.literal_eval(
                get_bot_control_data(request_data).get('microBotMemory')))
            request_data['microBotName'] = get_destination_bot(request_data)
            request_data['mainBotMemory'] = {
                'conversationStringContext': {
                    'nlpResponse': Template("{\'intentName\':\'$intent\',\'confidenceScore\':$score}").
substitute(
                        intent=get_bot_control_data(request_data).get('intentName'),
                        score=get_bot_control_data(request_data).get('confidenceScore'))
                }
            }
            await process_convo_control_bot_transfer_actions(handle, memory, request, request_data)
        else:
            await process_convo_control_bot_actions(handle, memory, request, request_data)
        return conversation_control_response(200, 'success', 'Conversation Transfer Successful')
    except CdmsException as error:
        log_attributes = {'correlation_id': correlation_id(request),
                          'app_name': Constants.APP_NAME,
                          'event_name': 'conversation_control',
                          'action': f'{Constants.ACTION_END} conversation_control',
                          'desc': 'exiting_conversation_control',
                          'result': Constants.LOG_FAILURE_RESULT,
                          'reason': str(error),
                          'micro_bot_name': get_destination_bot(request_data),
                          'bot_control_data': str(get_bot_control_data(request_data)),
                          'user_id': user_id(request_data)
                          }
        return await cdms_error_response(memory, error, log_attributes)
    except Exception as error:
        log.error(**{'correlation_id': correlation_id(request),
                     'app_name': Constants.APP_NAME,
                     'event_name': 'conversation_control',
                     'action': f'{Constants.ACTION_END} conversation_control',
                     'desc': 'exiting_conversation_control',
                     'result': Constants.LOG_FAILURE_RESULT,
                     'reason': str(error),
                     'micro_bot_name': get_destination_bot(request_data),
                     'bot_control_data': str(get_bot_control_data(request_data)),
                     'user_id': user_id(request_data)
                     })
        return await error_response(memory)
```

MESSAGE

```
@lazy_load_config(app_name='askamex-cdms')
@authorize_resource
async def on_message(request):

    request_data = await request.json()
    log.info(**{'correlation_id': correlation_id(request),
                'app_name': Constants.APP_NAME,
                'event_name': 'on_message',
                'action': f'{Constants.ACTION_START} on_message',
                'desc': 'entering_on_message',
                'result': Constants.LOG_SUCCESS_RESULT,
                'reason': Constants.LOG_SUCCESS_REASON,
                'micro_bot_name': bot_name(request_data),
                'utterance': user_selected_utterance(request_data),
                'user_id': user_id(request_data),
                'member_accounts': member_accounts(request_data) if is_card_member_flow(request) else {}
                })

    channel_id = request_data.get('request', {}).get('channelId')

    if channel_id in ['mobile', 'web']:
        return await AsyncBotHandler.message_received(request)
    else:
        failure_message = {
            'status': 'FAILURE',
            'statusMsg': f'Invalid channel Id with value {channel_id} received'
        }
        return web.Response(text=json.dumps(failure_message),
                            content_type='application/json',
                            status=400)


def member_accounts(request_data):
    return [get_member_details(member_account)
            for member_account in request_data['user'].get('memberAccounts', [{}])
            if get_member_details(member_account)]
```

REQUEST CONTROL

```
@lazy_load_config(app_name='askamex-cdms')
@authorize_resource
async def request_control(request):
    request_data, request_header = await request.json(), request.headers

    request_control_response_message = (200, 'SUCCESS', 'request_control_granted')
    try:
        memory = await get_memory_handler(request, request_data, request_header)
        memory_copy = copy.deepcopy(memory)

        log_attributes = {
            'interaction_id': interaction_id(request_data),
            'conversation_id': conversation_id(request_data),
            'chat_session_id': chat_session_id(request_data),
            'secondary_conversation_id': secondary_conversation_id(request_data),
            'micro_bot_name': request_data.get('destBotName'),
            'event_name': 'request_control',
            'chat_transcript_text': request_data.get('utterance')
        }

        log.info(**{**get_log_attributes(memory, request_data, request_header), **log_attributes, **{
            'action': f'{Constants.ACTION_START} request_control',
            'desc': 'entering_request_control',
            'result': Constants.LOG_SUCCESS_RESULT,
            'reason': Constants.LOG_SUCCESS_REASON
        }})

        if request_data.get('reasonCode') == 'reset_bots':
            memory['bot_context']['conversationStringContext'] = {}
            response = await update_memory(memory, False)
            if response.get('status') != Constants.SUCCESS:
                request_control_response_message = (400, Constants.FAILURE, 'Unknown error')
            return request_control_response(*request_control_response_message)
        else:
            request_data['request'] = {
                'message': {
                    'content': {
                        'desc': request_data.get('utterance')
                    }
                }
            }
            await process_requirement_capture(memory, request_data)
            await update_memory(memory_copy)
            request_control_response_message = \
                (200, Constants.SUCCESS, 'request_control_rejected', 'In-scope utterance')
            return request_control_response(*request_control_response_message)
    except CdmsException as cdms_exception:
        if cdms_exception.code == 'out_of_scope':
            request_control_response_message = (200, Constants.SUCCESS, 'request_control_granted')
        else:
            request_control_response_message = (400, Constants.FAILURE, cdms_exception.message)
        return await get_bot_handler_response(memory, request_control_response_message)
    except Exception as e:
        request_control_response_message = (400, Constants.FAILURE, f'Unknown error {str(e)}')
        return await get_bot_handler_response(memory, request_control_response_message)
    finally:
        log.info(**{**get_log_attributes(memory, request_data, request_header), **log_attributes, **{
            'action': f'{Constants.ACTION_END} request_control',
            'desc': 'exiting_request_control',
            'result': Constants.LOG_SUCCESS_RESULT,
            'reason': request_control_response_message
        }})


async def get_bot_handler_response(memory, request_control_response_message):
    memory['bot_context']['conversationStringContext'] = {}
    response = await update_memory(memory, False)
    if response.get('status') != Constants.SUCCESS:
        request_control_response_message = (400, Constants.FAILURE, 'Unknown error')
    return request_control_response(*request_control_response_message)
```

RESPONSES

RESOURCE RESPONSE

```
def acknowledgement_message(request_data, memory):
    bot_config = get_bot_config(request_data)
    message_id = cdms(memory)['message_id']
    response_body = {
        'status': 'SUCCESS',
        'messageId': message_id,
        'conversationId': request_data.get('request', {}).get('conversationId'),
        'secondaryConversationId': request_data.get('secondaryConversationId'),
        'interactionId': request_data.get('interactionId'),
        'chatSessionId': request_data.get('chatSessionId'),
        'message': {
            'content': {
                'desc': request_data.get('request', {}).get('message', {}).get('content', {}).get('desc', '')
            },
            'type': 'text',
            'userid': request_data.get('request', {}).get('message', {}).get('userid'),
            'conversationid': request_data.get('request', {}).get('message', {}).get('conversationid'),
            'create_time': request_data.get('request', {}).get('message', {}).get('create_time'),
            'update_time': request_data.get('request', {}).get('message', {}).get('update_time'),
            'messageid': message_id,
            'messageId': bot_config['message_id.prefix'] + str(uuid.uuid4())
        }
    }
    return response_body


def failure_message(message='', e=('unknown', 'unknown error'), memory={}):
    return {
        'status': 'FAILURE',
        'statusMsg': message,
        'miscInfo': get_misc_info(memory, False, e)
    }


def out_of_scope_message(e, memory):
    return {
        'status': 'SUCCESS',
        'statusMsg': 'out_of_scope',
        'miscInfo': get_misc_info(memory, e)
    }


def acknowledgment(request_data, memory, success):
    if success:
        return web.Response(text=json.dumps(acknowledgement_message(request_data, memory)),
                            content_type='application/json',
                            status=200)
    else:
        return web.Response(text=json.dumps(failure_message(memory=memory if memory else {})),
                            content_type='application/json',
                            status=500)


def unauthorized_response(actual_app_id, expected_app_id):
    return {
        'status': 'failure',
        'statusMsg': f'Unauthorized request: AppId mismatch: Actual - {actual_app_id}  & Expected - 
{expected_app_id}'
    }


def request_control_response(code, status, status_message, reason_text=None):
    response_message = {
        'status': status,
        'statusMsg': status_message
    }
    if reason_text:
        response_message['reasonText'] = reason_text

    return web.Response(text=json.dumps(response_message),
                        content_type='application/json',
                        status=code)


def conversation_control_response(code, status, status_message, reason_text=None):
    response_message = {
        'status': status,
        'statusMsg': status_message
    }
    if reason_text:
        response_message['reasonText'] = reason_text

    return web.Response(text=json.dumps(response_message),
                        content_type='application/json',
                        status=code)
```

ROUTES-

In this file we configure different API endpoints the code supports.We also configure the handlers that need to be executed when that particular end point is hit

```
def setup_routes(app):
    app.router.add_post('/askamexcdms/v1/message', on_message)
    app.router.add_post('/askamexcdms/v1/request_control', request_control)
    app.router.add_post('/askamexcdms/v1/conversation_control', conversation_control)
    app.router.add_get('/', index)
    app.router.add_get('/health', health)
    app.router.add_get('/env', env)


async def index(request):
    return web.Response(text="<h1> Hello, World! from Python async app running from PaaS using Gunicorn <
/h1>",
                        content_type='text/html')


async def health(request):
    return web.Response(text="<h1> Health OK! </h1>",
                        content_type='text/html')


async def env(request):
    return web.Response(text="<h1> Environment: " + get_env() + " </h1>",
                        content_type='text/html')


def get_env():
    env = os.environ[constants.EPAAS_ENV] if constants.EPAAS_ENV in os.environ else constants.DEFAULT_ENV
    env_label = os.environ[constants.EPASS_LABEL] if constants.EPASS_LABEL in os.environ else ''
    return '{}-{}'.format(env_label, env) if env_label else env
```

SERVICES

NLP SERVICES

```
async def get_intent_info(correlation_id, bot_name, utterance, model_name):
    try:
        log_data = {'correlation_id': correlation_id,
                    'app_name': Constants.APP_NAME,
                    'event_name': 'get_intent_info',
                    'micro_bot_name': bot_name
                    }
        headers = {
            Constants.CONTENT_TYPE_KEY: Constants.CONTENT_TYPE_VALUE,
            'X-AXP-CorrelationId': correlation_id,
            'X-API-KEY': config.get('askamex.cdms.nlp.api.key', vault_property=True)
        }

        json_data = {
            "rawText": utterance,
            "appName": config.get('askamex.cdms.nlp.sub.intent.app.name'),
            "modelName": model_name
        }

        response = await aio_http_post(config.url('askamex.cdms.nlp.api.url'), json_data, headers,
                                       config.get_int('askamex.cdms.nlp.api.timeout'), log_params=log_data)
        if response.get('status') == 'success':
            intent_name = response.get('intent').get('intentName')
            threshold = get_intent_threshold(response, intent_name,
                                             config.get_float('askamex.cdms.nlp.intent.default.threshold'))
            return intent_name, response.get('intent').get('confidenceScore'), threshold, response.get
('intent')
        else:
            return None, 0, 0, None
    except Exception as error:
        log.error(**{'error': "get_intent_info Exceptions", 'reason': str(error)})
        return None, 0, 0, None
```

TOOLS

SECURITY

```
def authorize_resource(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        request = args[0]
        auth = request.headers.get('Authorization')
        if not auth or str(auth.split(' ')[1]) != config.get('askamex.cdms.app.id'):
            return web.Response(text=json.dumps(unauthorized_response(auth, config.get('askamex.cdms.app.
id'))),
                                content_type='application/json',
                                status=401)
        else:
            return func(*args, **kwargs)
    return wrapper
```

UTIL

ACCOUNT UTIL

```
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')


def selected_card_type(memory):
    selected_card = cdms(memory).get('card_selector')
    accounts = ast.literal_eval(cdms(memory)['accounts'])
    return selected_card_account(accounts, selected_card).get('card_type')


def selected_payment_type(memory):
    selected_card = cdms(memory).get('card_selector')
    accounts = ast.literal_eval(cdms(memory)['accounts'])
    return selected_card_account(accounts, selected_card).get('payment_type')


def selected_last_5_digits(memory):
    selected_card = cdms(memory).get('card_selector')
    accounts = ast.literal_eval(cdms(memory)['accounts'])
    return selected_card_account(accounts, selected_card).get('last_5_digit')


def is_single_account(memory):
    accounts = ast.literal_eval(cdms(memory)['accounts'])
    cdms(memory)['single_account'] = len(accounts) == 1
    single_account = cdms(memory)['single_account']
    if single_account:
        accounts = ast.literal_eval(cdms(memory)['accounts'])
        cdms(memory)['card_selector'] = selected_card_account(accounts)['account_number_display_text']
        cdms(memory)['product_desc'] = selected_card_account(accounts)['product_desc']
    return single_account


def get_decrypted_card_number(memory):
    selected_card = cdms(memory).get('card_selector')
    accounts = ast.literal_eval(cdms(memory)['accounts'])
    return selected_card_account(accounts, selected_card).get('account_token_decrypted')


def is_basic_account(memory):
    cdms(memory)['is_basic_card'] = ast.literal_eval(cdms(memory)['selected_card_account']).get('is_basic')
    return ast.literal_eval(cdms(memory)['selected_card_account']).get('is_basic')


def get_account_token(memory):
    selected_card = cdms(memory)['card_selector']
    accounts = ast.literal_eval(cdms(memory)['accounts'])
    cdms(memory)['selected_card_account'] = str(selected_card_account(accounts, selected_card))
    accounts.append({
        'account_number_display_text': 'Another Card',
        'account_token': '000000000000000'
    })
    return selected_card_account_token(accounts, selected_card)
```

ACTION EXECUTOR

```
async def execute_action_v1(memory, request, action_config_path, bot_config, actions, sub_intent_actions=[]):
    action_config = get_action_config(action_config_path)
    events = action_config['events']

    if not sub_intent_actions:
        return await execute_intent(action_config['name'], request, memory, actions, events, bot_config)
    else:
        valid_sub_intent = await execute_intent(action_config['name'],
                                                request, memory, sub_intent_actions,
                                                events, bot_config,
                                                is_sub_intent_actions=True)
        return valid_sub_intent if valid_sub_intent else \
            await execute_intent(action_config['name'], request, memory, actions, events, bot_config)


async def execute_intent(action_name, request, memory, actions, events, bot_config,
is_sub_intent_actions=False):
    request_data, request_header = await request.json(), request.headers
    log.info(**{**get_log_attributes(memory, request_data, request_header), **{
                'event_name': f'{action_name}.execute',
                'action': f'{Constants.ACTION_START} {action_name}',
                'desc': f'entering_{action_name}_execute',
                'result': Constants.LOG_SUCCESS_RESULT,
                'reason': Constants.LOG_SUCCESS_REASON,
                'chat_transcript_text': user_selected_utterance(request)
                }})
    is_handler_found = False
    handler_response = ''
    log_handler = ''
    log_condition = ''
    param = namedtuple('param', ['memory', 'request'])
    try:
        param = param(memory, request)
        condition_generator = (condition for condition in actions.keys() if
                               (await condition(param) if
                                inspect.iscoroutinefunction(condition) else condition(param)))

        try:
            condition = await condition_generator.__anext__()
            handler = actions[condition]
            is_handler_found = True
            log_handler = handler.__name__
            log_condition = f'{action_name}_{condition.__name__}' \
                if bot_config.get('logger.version') == 'v1' else condition.__name__
            set_monitoring_action_info(memory, action_name, condition.__name__, log_handler)
            event = get_event(events, condition, handler)
            log.info(**{**get_log_attributes(memory, request_data, request_header), **{
                                  'event_name': f'Starting handler {log_handler} with condition '
                                  f'{log_condition} for given event {str(event)}',
                                  'action': f'{Constants.ACTION_START} {log_condition}',
                                  'desc': f'entering {log_condition} condition',
                                  'result': Constants.LOG_SUCCESS_RESULT,
                                  'reason': Constants.LOG_SUCCESS_REASON,
                                  'chat_transcript_text': user_selected_utterance(request)
                                  }})

            handler_response = await handler(event, request, memory, bot_config) if \
                inspect.iscoroutinefunction(handler) else handler(event, request, memory, bot_config)
            log.info(**{**get_log_attributes(memory, request_data, request_header), **{
                                  'event_name': f'Exiting handler {log_handler} with condition '
                                  f'{log_condition} for given event {str(event)}',
                                  'action': f'{Constants.ACTION_END} {log_condition}',
                                  'desc': f'exiting from condition {log_condition}',
                                  'result': Constants.LOG_SUCCESS_RESULT,
                                  'reason': Constants.LOG_SUCCESS_REASON,
                                  'chat_transcript_text': user_selected_utterance(request)
                                  }})
        except StopAsyncIteration:
            if not is_sub_intent_actions:
                raise CdmsException('FAILURE', 'Unable to find matching action criteria')
        return handler_response or is_handler_found
    except Exception as error:
        log.error(**{**get_log_attributes(memory, request_data, request_header), **{
                     'event_name': f'{action_name}.execute',
                     'action': f'{Constants.ACTION_END} {action_name}',
                     'desc': f'exiting_{action_name}_execute',
                     'result': Constants.LOG_FAILURE_RESULT,
                     'reason': f'Issue Within {action_name} Step: {str(error)} '
                               f'while executing handler {log_handler} with condition {log_condition}',
                     'chat_transcript_text': user_selected_utterance(request)
                     }})

        raise CdmsException('FAILURE', 'Issue Within {} Step: {} while executing handler {} with condition
{}'
                            .format(action_name, str(error), log_handler, log_condition))


async def execute_handler(event, request, memory, bot_config, attachment=None, message_attributes=None,
template=None):
    return await execute_handler_async(event, request, memory, bot_config, attachment, message_attributes,
template)


async def execute_handler_async(event, request, memory, bot_config, attachment=None,
message_attributes=None, template=None):
    set_previous_messages(event, memory, attachment)

    command_dict = {
        'send_message': send_message.run,
        'set_entity': set_entity.run,
        'relinquish_control': relinquish_control.run,
        'execute_action': execute_action.run,
        'next_action': next_action.run,
        'include_previous_messages': include_previous_messages.run,
        'update_previous_message': update_previous_message.run,
        'update_memory': update_memory.run,
        'delegate_control': delegate_control.run,
        'grant_control': grant_control.run,
        'next_sub_intent_action': next_sub_intent_action.run,
        'revert_control': revert_control.run,
        'event_tracking': event_tracking.run,
        'no_matching_operation': no_matching_operation.run
    }
    kwargs = {
        'memory': memory,
        'request': request,
        'bot_config': bot_config,
        'template': template,
        'message_attributes': message_attributes,
        'attachment': attachment
    }

    for command in event.get('commands', []):
        kwargs.update({'command': command})
        await command_dict.get(command.get('operation'), 'no_matching_operation')(**kwargs)


async def execute_handler_sync(event, request, memory, bot_config, attachment=None, message_attributes=None,
template=None):
    command_dict = {
        'send_message': send_message.run,
        'set_entity': set_entity.run,
        'relinquish_control': relinquish_control.run,
        'execute_action': execute_action.run,
        'next_action': next_action.run,
        'include_previous_messages': skip_processing.run,
        'update_memory': update_memory.run,
        'event_tracking': event_tracking.run,
        'no_matching_operation': no_matching_operation.run
    }
    kwargs = {
        'memory': memory,
        'request': request,
        'bot_config': bot_config,
        'template': template,
        'message_attributes': message_attributes,
        'attachment': attachment,
        'is_sync': True
    }

    tasks = []
    result = []
    for command in event.get('commands', []):
        kwargs.update({'command': command})
        if command.get('operation') in ['send_message']:
            tasks.append(command_dict[command.get('operation')](**kwargs))
        elif command.get('operation') in ['relinquish_control']:
            asyncio.ensure_future(command_dict[command.get('operation')](**kwargs))
        elif command.get('operation') in ['execute_action']:
            result.extend(await command_dict[command.get('operation')](**kwargs))
        else:
            await command_dict.get(command.get('operation'), 'no_matching_operation')(**kwargs)

    messages = await asyncio.gather(*tasks)
    messages[0]['messages'] = [message.get('message') for message in messages if message.get('message')]
    del messages[0]['message']
    return messages[0]


def set_previous_messages(event, memory, attachment=None):
    if cdms(memory).get('flow_type') != 'sub_intents' and \
            len(event.get('commands', {})) > 0 and \
            event.get('commands', {})[0].get('operation') != 'delegate_control':
        cdms(memory)['previous_messages'] = str([])
        for command in event.get('commands', []):
            if command.get('operation') == 'send_message':
                previous_message_list = ast.literal_eval(cdms(memory).get('previous_messages'))
                command['inputs'] = {**command.get('inputs', {}),
                                     **{'attachment': attachment() if attachment else None}}
                previous_message_list.append(command)
                recent_template = \
                    command.get('inputs', {}).get('template_name', command.get('inputs', {}).get('template',
''))
                set_monitoring_template_info(memory, recent_template)
                cdms(memory)['previous_messages'] = str(previous_message_list)


def get_bot_config(request_data):
    config = importlib.import_module(
        'bots.{}.util.bot_util'.format(request_data.get('microBotName') or request_data.get('destBotName')))
    bot_config = getattr(config, 'bot_config')
    return bot_config()
```

BOT UTIL

```
def current_time_utc():
    return datetime.utcnow().strftime("%Y-%m-%dT%H:%M:%S.%f")


def next_action(memory):
    return memory.get('bot_context', {}).get('conversationStringContext', {}).get('next_action')


def next_sub_intent_action(memory):
    return memory.get('bot_context', {}).get('conversationStringContext', {}).get('next_sub_intent_action')
```

```
def generate_question_id(channel):
    return str(int(time.time())) if channel == 'mobile' else int(time.time())


def user_selected_utterance(request_body):
    channel_id = request_body.get('request', {}).get('channelId')
    message = request_body.get('request', {}).get('message', {})
    return message.get('attachment', {}).get('options', [{}])[0].get('option_text') \
        if channel_id == 'web' and message.get('type') == 'single_choice_select' \
        else message.get('content', {}).get('desc')


def get_intent_name(request, default_threshold):
    nlp_response = ast.literal_eval(request.get('mainBotMemory', {}).
                                    get('conversationStringContext', {}).get('nlpResponse', {}))
    intent_name = nlp_response.get('intentName')
    return intent_name, float(nlp_response.get('confidenceScore')), \
           get_intent_threshold(nlp_response, intent_name, default_threshold)


def get_sub_intent_name(memory):
    return ast.literal_eval(cdms(memory).get('sub_intent_info', '(None, 0, 0, None)'))


def get_entities(request):
    return ast.literal_eval(request.get('mainBotMemory', {}).get('conversationStringContext', {})
                                .get('nlpResponse', {})).get('concepts', [{}])


def get_entity_value(request, entity_type, threshold):
    nlp_response = ast.literal_eval(request.get('mainBotMemory', {}).
                                    get('conversationStringContext', {}).get('nlpResponse', {}))
    filtered_entities = [(entity['value'], entity['type'], get_entity_threshold(nlp_response, entity,
threshold))
                         for entity in get_entities(request)
                         if entity['type'] == entity_type and
                         float(entity['confidence']) >= get_entity_threshold(nlp_response, entity,
threshold)]
    return sorted(filtered_entities, key=lambda x: x[2], reverse=True)[0][0] if len(filtered_entities) else
None


def user_id(request_body):
    return request_body.get('request', {}).get('userId', request_body.get('userId'))


def correlation_id(request):
    return request.headers.get('X-AXP-CorrelationId')


def get_action_config(action_config_path):
    with open(action_config_path) as action_config:
        return json.load(action_config) if action_config_path.split('.')[1] == 'json' \
            else yaml.load(action_config, Loader=yaml.FullLoader)


async def async_get_action_config(action_config_path):
    async with aiofiles.open(action_config_path, mode='r') as action_config:
        config = await action_config.read()
        return json.loads(config) if action_config_path.split('.')[1] == 'json' \
            else yaml.load(config, Loader=yaml.FullLoader)


def get_events(action_config_path):
    return get_action_config(action_config_path)['events']


def get_requirements(action_config_path):
    return get_action_config(action_config_path)['requirements']
```

```
async def async_get_requirements(action_config_path):
    return await async_get_action_config(action_config_path)['requirements']


def get_action_name(action_config_path):
    return get_action_config(action_config_path)['name']


def set_and_get_accounts(memory, request_data):
    accounts = cdms(memory).get('accounts', None)
    if not (accounts and cdms(memory).get('next_action', None)):
        accounts = [get_member_details(member_account)
                    for member_account in request_data['user'].get('memberAccounts', [{}])
                    if get_member_details(member_account)]
        cdms(memory)['accounts'] = str(accounts)
    else:
        accounts = ast.literal_eval(accounts)
    return accounts


def get_account_tokens(memory):
    return [{'accountToken': account['account_token']} for account in ast.literal_eval(cdms(memory)
['accounts'])]


def get_member_details(member_account):
    if member_account.get('status', {}).get('accountStatus')[0] in ['Active']:
        return {
            'account_token': member_account.get('accountToken'),
            'account_token_decrypted': member_account.get('accountTokenDecrypted'),
            'decrypted_last_4_digit': member_account.get('accountTokenDecrypted', 'BLNK')[-4:],
            'is_basic': member_account.get('identifiers', {}).get('isBasic'),
            'account_key': member_account.get('identifiers', {}).get('accountKey'),
            'first_name': member_account.get('holder', {}).get('profile', {}).get('firstName'),
            'last_name': member_account.get('holder', {}).get('profile', {}).get('lastName'),
            'product_desc': member_account.get('product', {}).get('digitalInfo', {}).get('productDesc'),
            'card_product_id': member_account.get('product', {}).get('digitalInfo', {}).get('productId'),
            'display_account_number': member_account.get('identifiers', {}).get('displayAccountNumber'),
            'member_name': '{} {}'.format(member_account.get('holder', {}).get('profile', {}).get
('firstName', ''),
                                          member_account.get('holder', {}).get('profile', {}).get
('lastName', '')),
            'account_number_display_text': '{} ending -{}'.format(
                member_account.get('product', {}).get('digitalInfo', {}).get('productDesc', ''),
                member_account.get('identifiers', {}).get('displayAccountNumber')[-5:]),
            'last_5_digit': '-{}'.format(member_account.get('identifiers', {}).get('displayAccountNumber')
[-5:]),
            'card_type': member_account.get('product', {}).get('accountTypes', {}).get('lineOfBusinessType'),
            'payment_type': member_account.get('product', {}).get('accountTypes', {}).get('paymentType'),
            'status': member_account.get('status', {}).get('accountStatus')[0]
        }
    else:
        return None


def format_json(data):
    return json.dumps(data, sort_keys=True, indent=2, separators=(',', ': '))


def bot_name(request):
    return request.get('microBotName')


def bot_name_from_memory(memory):
    return memory.get('bot_context', {}).get('botName')


def get_bot_control_data(request):
    return request.get('botControlData')
```

```
def get_destination_bot(request):
    return request.get('destBotName')


def get_intent_threshold(nlp_response, intent_name, default_threshold):
    return float(next((intent_config.get('maxConfidenceScore')
                       for intent_config in nlp_response.get('modelConfigResponse', {}).get('intents', [])
                       if intent_config.get('intent') == intent_name), default_threshold))


def get_entity_threshold(nlp_response, concept, default_threshold):
    return float(next((model_config_concept.get('maxConfidenceScore')
                       for model_config_concept in nlp_response.get('modelConfigResponse', {}).get
('concepts', [])
                       if model_config_concept.get('entity') == concept.get('name')), default_threshold))


def is_sync_flow(request_data):
    return request_data.get('request', {}).get('channelId') in ['web']


def set_next_action(memory, action):
    cdms(memory)['next_action'] = action


def set_next_sub_intent_action(memory, action):
    cdms(memory)['next_sub_intent_action'] = action


def cdms(memory):
    return aaa_memory(memory)


def is_session_invalid(memory, flow_timeout):
    return cdms(memory).get('accounts') and cdms(memory).get('last_updated') and \
          int(time.time() - float(cdms(memory).get('last_updated'))) > flow_timeout


def channel_name(request_data):
    return request_data.get('request', {}).get('channelId')


def secondary_conversation_id(request_data):
    return request_data.get('secondaryConversationId')


def conversation_id(request_data):
    return request_data.get('request', {}).get('conversationId', request_data.get('conversationId'))


def interaction_id(request_data):
    return request_data.get('interactionId')


def user_type(request_data):
    return request_data.get('userType')


def chat_session_id(request_data):
    return request_data.get('chatSessionId')


def locale(request):
    return request.headers.get('X-AXP-Locale') if request.headers else None


def trace_id():
    return f'aaa_{str(uuid.uuid4())}'


def get_log_attributes(memory, request_data, headers={}):
    log_attributes = {**ast.literal_eval(cdms(memory).get('log_attributes')), **{'created_by': Constants.
CREATED_BY}} if cdms(
        memory).get('log_attributes') else {'created_by': Constants.CREATED_BY}
    if headers:
        return {**log_attributes,
                **{'correlation_id': headers.get('X-AXP-CorrelationId')}}
    return log_attributes


def update_log_attributes(memory, **kwargs):
    log_attributes = ast.literal_eval(cdms(memory).get('log_attributes')) if cdms(memory).get(
        'log_attributes') else cdms(memory).get('log_attributes', {})
    log_attributes.update(dict(kwargs))
    cdms(memory)['log_attributes'] = str(log_attributes)


def log_attribute_value(memory, attribute_name):
    return ast.literal_eval(cdms(memory).get('log_attributes')).get(attribute_name) if cdms(memory).get(
        'log_attributes') else None


def set_log_attribute(memory, name, value):
    log_attributes = ast.literal_eval(cdms(memory).get('log_attributes')) \
        if cdms(memory).get('log_attributes') else {}
    log_attributes[name] = value
    cdms(memory)['log_attributes'] = str(log_attributes)


def get_chat_handling_time(chat_end_time, chat_start_time):
    try:
        cht = str(((datetime.strptime(chat_end_time, '%Y-%m-%dT%H:%M:%S.%f')) - (
            datetime.strptime(chat_start_time, '%Y-%m-%dT%H:%M:%S.%f'))).total_seconds()*1000)
        return f'{cht} ms'
    except Exception as error:
        return None


def is_card_member_flow(request):
    return request.headers.get('X-AXP-ClientSourceId') not in ['nlpe', 'bca', 'cca']
```

MONITORING UTIL

```
def get_misc_info(memory, out_of_scope=False, error=False):
    bot_monitoring_info = ast.literal_eval(cdms(memory).get('bot_monitoring_info', '{}'))

    topic_name = bot_monitoring_info.get('topic_info', {}).get('topic_name', '')
    topic_confidence = bot_monitoring_info.get('topic_info', {}).get('topic_confidence', '')
    topic_threshold = bot_monitoring_info.get('topic_info', {}).get('topic_threshold', '')

    misc_info = {
        'action_name': bot_monitoring_info.get('action_info', {}).get('action_name', ''),
        'condition_name': bot_monitoring_info.get('action_info', {}).get('condition_name', ''),
        'handler_name': bot_monitoring_info.get('action_info', {}).get('handler_name', ''),

        'previous_action_name': bot_monitoring_info.get('action_info', {}).get('previous_action_name', ''),
        'previous_condition_name': bot_monitoring_info.get('action_info', {}).get('previous_condition_name',
''),
        'previous_handler_name': bot_monitoring_info.get('action_info', {}).get('previous_handler_name', ''),

        'previous_utterance': bot_monitoring_info.get('utterance_info', {}).get('previous_utterance', ''),
        'current_utterance': bot_monitoring_info.get('utterance_info', {}).get('current_utterance', ''),

        'previous_template': bot_monitoring_info.get('template_info', {}).get('previous_template', ''),
        'current_template': bot_monitoring_info.get('template_info', {}).get('current_template', ''),

        'topic_name': topic_name,
        'topic_confidence': topic_confidence,
        'topic_threshold': topic_threshold,

        'intent_name': bot_monitoring_info.get('intent_info', {}).get('intent_name', topic_name),
        'intent_confidence': bot_monitoring_info.get('intent_info', {}).get('intent_confidence',
topic_confidence),
        'intent_threshold': bot_monitoring_info.get('intent_info', {}).get('intent_threshold',
topic_threshold),
        'locale': bot_monitoring_info.get('locale')
    }

    sub_intent_info = bot_monitoring_info.get('sub_intent_monitoring_info', {})
    if sub_intent_info:
        misc_info.update({
            'sub_intent_name': sub_intent_info.get('sub_intent_name', ''),
            'sub_intent_confidence': sub_intent_info.get('sub_intent_confidence', ''),
            'sub_intent_threshold': sub_intent_info.get('sub_intent_threshold', '')
        })

    if out_of_scope:
        misc_info.update({
            'out_of_scope_code': 'sub_intent_duplicate_out_of_scope' if _is_sub_intent_duplication(
                memory) else out_of_scope.code,
            'out_of_scope_reason': 'Out of scope due to sub-intent duplication' if _is_sub_intent_duplication
(
                memory) else out_of_scope.message
        })

    if error or cdms(memory).get('error'):
        error = error or cdms(memory).get('error')
        misc_info.update({
            'error_detail_code': error.code if hasattr(error, 'code') else 'unknown',
            'error_detail_reason': error.message if hasattr(error, 'message') else 'unknown error'
        })

    return misc_info


def set_monitoring_topic_info(memory, name, confidence, threshold):
    data = {
        'topic_info': {
            'topic_name': name,
            'topic_confidence': confidence,
            'topic_threshold': threshold,
        }
    }
    _ingest_bot_monitoring_info(memory, data)


def set_monitoring_intent_info(memory, name, confidence, threshold):
    data = {
        'intent_info': {
            'intent_name': name,
            'intent_confidence': confidence,
            'intent_threshold': threshold,
        }
    }
    _ingest_bot_monitoring_info(memory, data)


def set_monitoring_sub_intent_info(memory, name, confidence, threshold):
    bot_monitoring_info = ast.literal_eval(cdms(memory).get('bot_monitoring_info', '{}'))
    sub_intent_info = bot_monitoring_info.get('sub_intent_monitoring_info', {})
    previous_sub_intent = sub_intent_info.get('sub_intent_name')

    data = {
        'sub_intent_monitoring_info': {
            'sub_intent_name': name,
            'sub_intent_confidence': confidence,
            'sub_intent_threshold': threshold,
            'duplicate_sub_intent': name == previous_sub_intent
        }
    }
    _ingest_bot_monitoring_info(memory, data)


def set_monitoring_utterance_info(memory, request_data, locale):
    bot_monitoring_info = ast.literal_eval(cdms(memory).get('bot_monitoring_info', '{}'))
    data = {
        'locale': locale,
        'utterance_info': {
            'current_utterance': user_selected_utterance(request_data),
            'previous_utterance': bot_monitoring_info.get('utterance_info', {}).get('current_utterance', '')
        }
    }
    _ingest_bot_monitoring_info(memory, data)


def set_monitoring_template_info(memory, current_template):
    bot_monitoring_info = ast.literal_eval(cdms(memory).get('bot_monitoring_info', '{}'))
    data = {
        'template_info': {
            'current_template': current_template,
            'previous_template': bot_monitoring_info.get('template_info', {}).get('current_template', '')
        }
    }
    _ingest_bot_monitoring_info(memory, data)


def set_monitoring_action_info(memory, action, condition, handler):
    bot_monitoring_info = ast.literal_eval(cdms(memory).get('bot_monitoring_info', '{}'))
    data = {
        'action_info': {
            'action_name': action,
            'condition_name': condition,
            'handler_name': handler,

            'previous_action_name': bot_monitoring_info.get('action_info', {}).get('action_name', ''),
            'previous_condition_name': bot_monitoring_info.get('action_info', {}).get('condition_name', ''),
            'previous_handler_name': bot_monitoring_info.get('action_info', {}).get('handler_name', '')
        }
    }
    _ingest_bot_monitoring_info(memory, data)


def _ingest_bot_monitoring_info(memory, data):
    bot_monitoring_info = ast.literal_eval(cdms(memory).get('bot_monitoring_info', '{}'))
    bot_monitoring_info.update(data)
    cdms(memory)['bot_monitoring_info'] = str(bot_monitoring_info)


def _is_sub_intent_duplication(memory):
    faq_count = config.get_int('askamex.cdms.allowed.faq.sub.intent.count')
    entity_count = config.get_int('askamex.cdms.allowed.entity.sub.intent.count')
    bot_monitoring_info = ast.literal_eval(cdms(memory).get('bot_monitoring_info', '{}'))
    sub_intent_info = bot_monitoring_info.get('sub_intent_monitoring_info', {})
    return sub_intent_info.get('duplicate_sub_intent') or \
           int(cdms(memory).get('allowed_faq_sub_intent_count', 0)) >= faq_count or \
           int(cdms(memory).get('allowed_entity_sub_intent_count', 0)) >= entity_count
```