CS Capstone Documentation
Team E
Multnomah County Chat Bot

==========
These programs are required to work with Composer directly on your system (It also works as a web application, which can be run as a docker container)
https://github.com/microsoft/BotFramework-Composer
https://github.com/Microsoft/BotFramework-Emulator/blob/master/README.md

*This is also using the code currently in MASTER as of 3/8/2021. API calls and such are subject to changes.*

# I. Variable Documentation

This section covers all the variables used and shared within the project, and their purpose. This also includes various definitions and terms used throughout the explanations. This is essential for understanding other sections.

**Bot Features:**
Language Understanding
Ex: Utterance → identification of intent and entities
Dialog
Ex: Conversation flow chart
Language Generation
Ex: Responses by bot, display strings, cards

**Bot Memory and Identification:**
**Intent**
I want to book a flight
**Intent + Entity**
I want to book a flight to **JFK**
**Intent + Entity (Roles)**
I want to book a flight from **BRU** to **JFK**
**Intent + Multiple Entities**
I want to book a flight **tomorrow** to **JFK**

**Variables**
**User**.xxxx
Saved throughout uses. Persistent throughout dialogs
**Dialog**.xxxx
Saved through the flow of each dialog. Only exists within a specific dialog, and any dialogs called from it.
**Conversation**.xxxx
Saved through the user session. Wiped when bot is closed.
**Turn**.xxxx

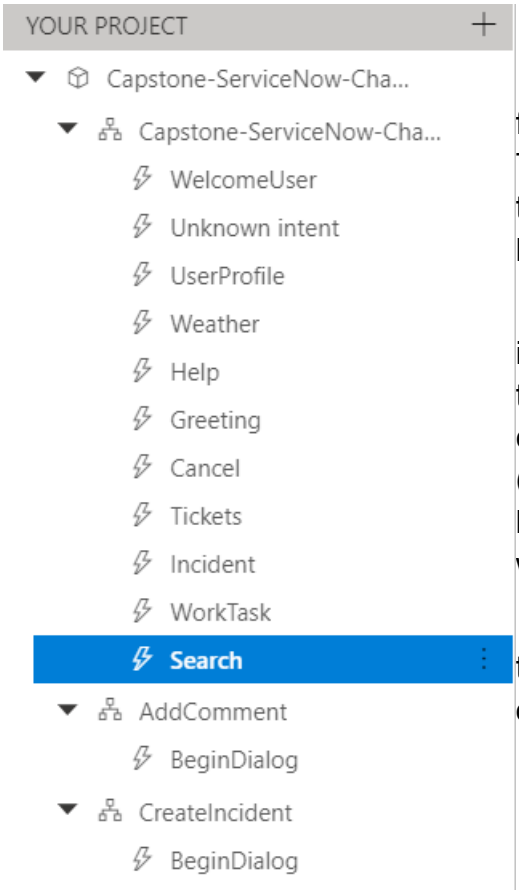
Turn is associated with a single turn. You can also think of this as the bot handling a single message from the user. Properties in the turn scope are discarded at the end of the turn

# II. How to Use the Composer

This section is primarily a quick guide to adding new dialog options, reactions, and other basic mechanics of using the Composer to the current project. There is also the basic tutorial given by Microsoft on how a bot is normally formed in **IV. External Links**. This section will also mention all the dialog currently used within the project, and what each of them do. This document will not include information on starting a brand new bot, as the bot already has been created in this project.

## II-a. Preparing a new dialog

There are several depths that are created when making a new dialog for the chat bot.

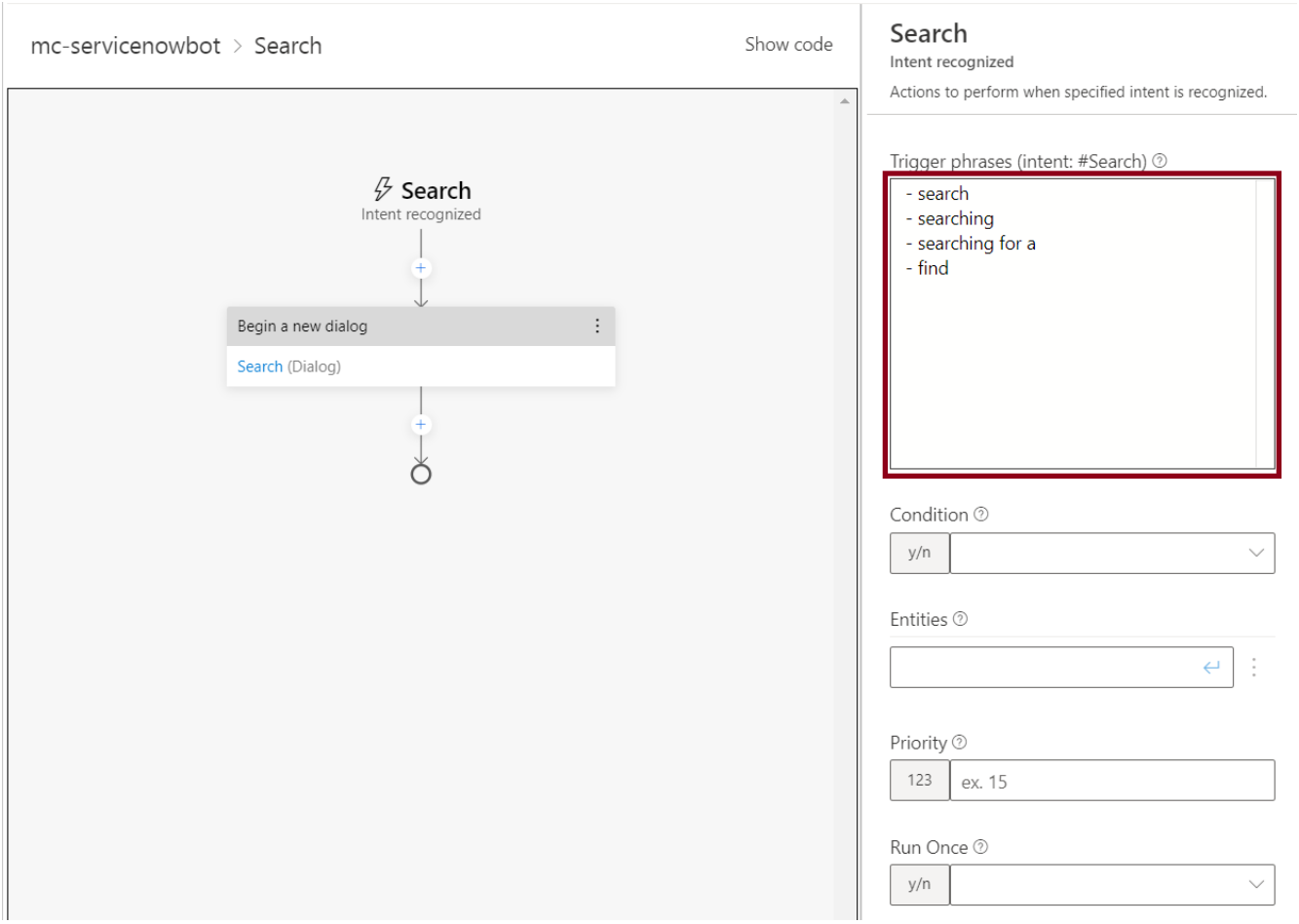



Specifically, there is the root bot functionality (The first drop down, indicated by the cube icon on the left). This is where someone can add more dialog options to the project by clicking the three dots that appear when hovering over the options, and selecting "+ *Add a Dialog*"

Next are the dialog options for the bot, which are indicated by the graph tree symbol on it's left. These are the basis for all future **triggers** that will occur to communicate with the bot in question. This first dialog (*Capstone-ServiceNow-Cha...*) is the core of the chat bot, and the bot automatically will return to this state whenever it finishes a branching dialog.

Finally, there are the **events** which will catch **intents** to move onto other dialog options with more specific operations. These are indicated by lightning bolt icons.

Each dialog in the Composer consists of triggers which activate once an **intent** is identified. **Intents** are key phrases that the bot uses to generally learn which additional phrases can work to activate a specific dialog.




*The trigger phrases are highlighted here*

Most of these will divert into the proper dialog interaction, as seen on the left. Also noted in the picture right below the trigger phrases are **Conditions, Entities, Priority,** and **Run Once**.

**Conditions** are prerequisites that need to be fulfilled before the trigger can actually fire. In the case of Search, you can activate it whenever. However, there are trigger conditions for both Help and Cancel. (Specifically tied to a score variable)

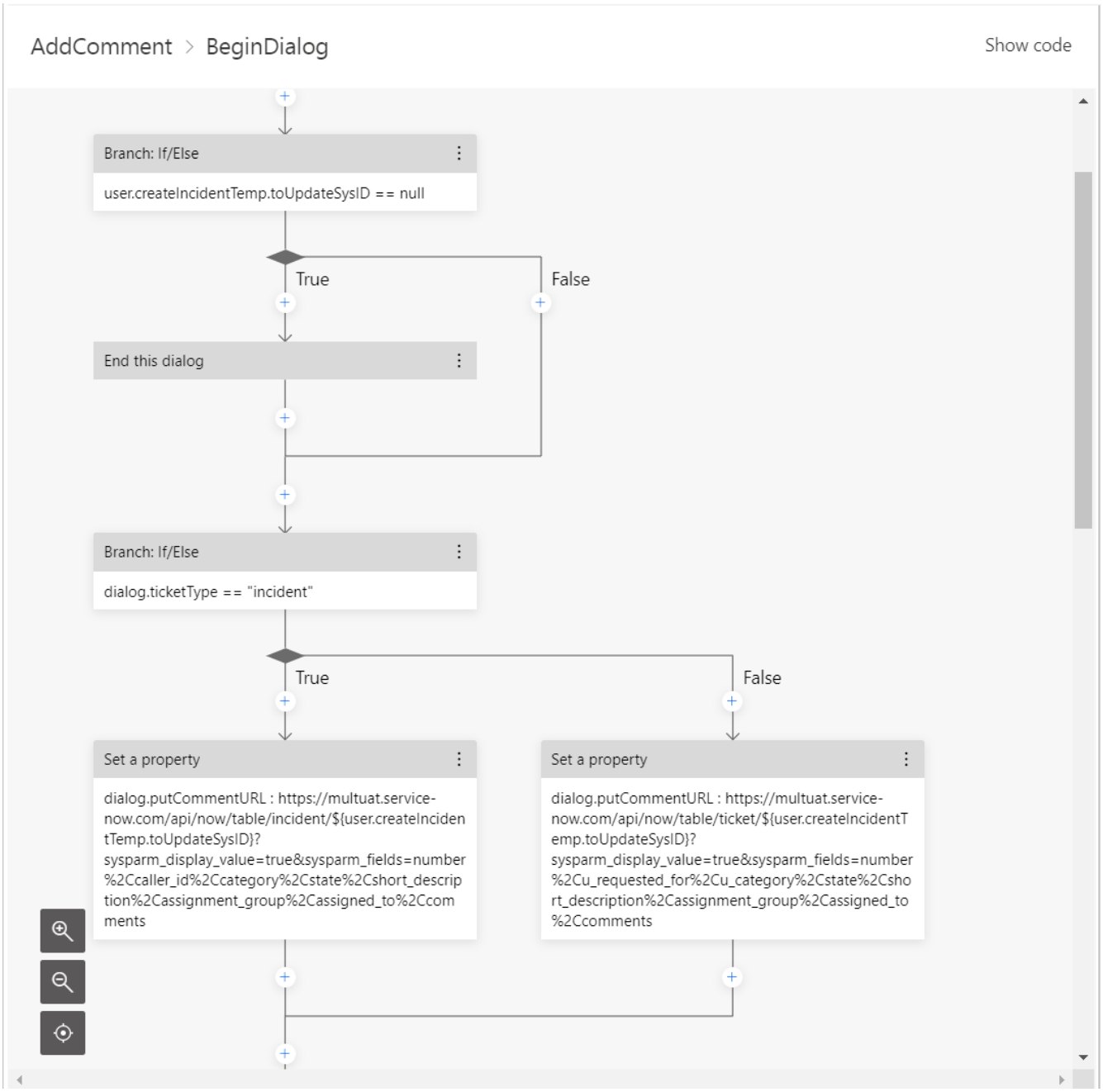
**Entities** are mentioned in the previous section, and are identifiable markers we have as a target. No dialog options in the project currently use this feature.

**Priority** is a basis for which dialog activates first when there is a shared **intent** between two different dialog options. No dialog options in the project currently use this feature.

**Run Once** is a special condition that allows a specific dialog to only run once in a given session. This can be important for setting up specific information that you only want to update a single time in a session, like personal information from an employee. No dialog options in the project currently use this feature.

## II-b. Providing action for a dialog

Most actions that the composer will do is based off of checking a variable and using a branching if/else statement to figure out what to do with the information that it has. Shown below is an example from the dialog “AddComment”.




This is the very beginning of the dialog, and there are certain key features of the Composer in effect here. Right at the beginning there is a conditional check to make sure a certain variable (user.createIncidentTemp.toUpdateSysID) is filled or not. If it isn’t, the bot will kill the dialog immediately and return back as if the dialog was never called at all. The location that this is assigned is located near the end of the CreateIncident dialog, and will only read **NULL** if the user did not want to add a comment to the incident (since this is always called

after the create incident). There is also the chance that it reads **NULL** if the API request doesn't return all necessary information, but that will not occur during normal operation. The dialog then will continue onward, assigning new information when it needs to (we see that dialog.putCommentURL is assigned a new value here).

The unique aspect of Bot Composer is that a variable can be created on the spot mid-dialog and the Composer will recognize it as a variable that has always existed in it's current scope.

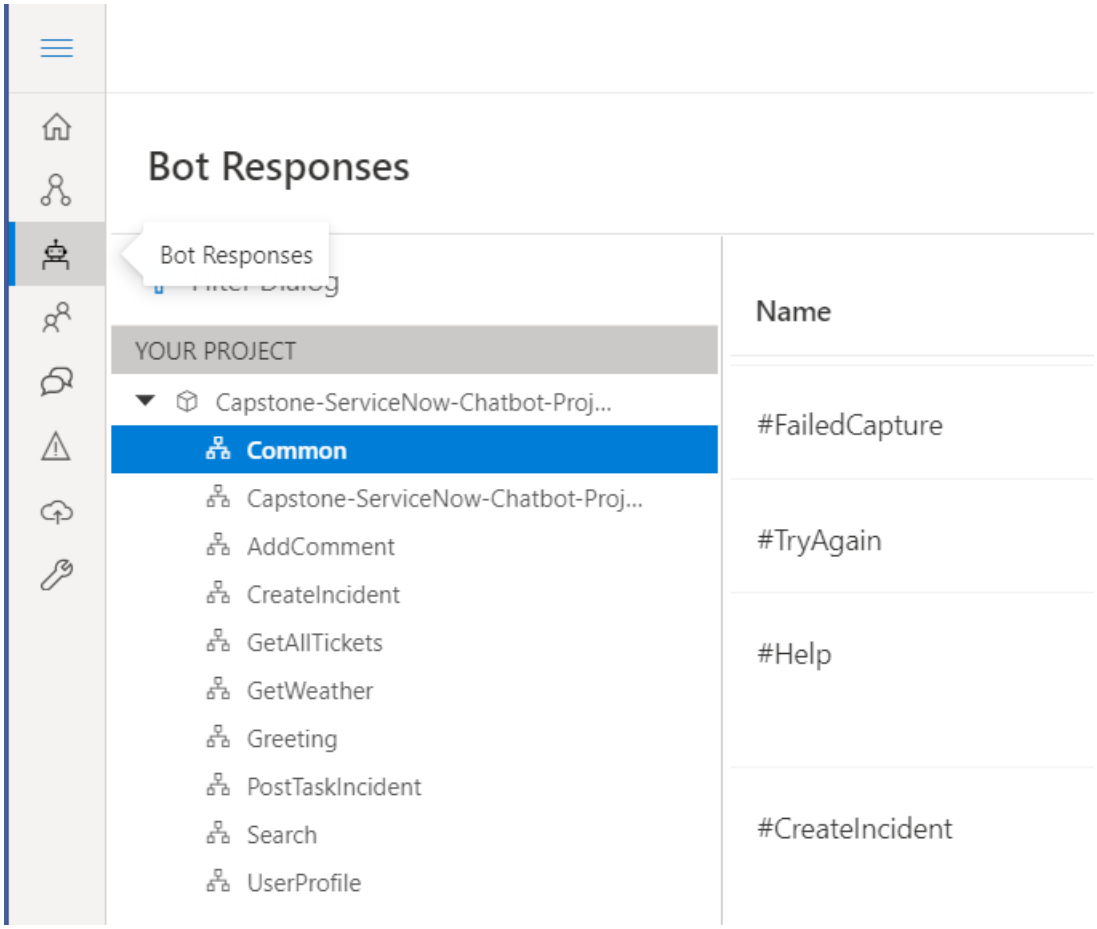
There are several options for what the Bot can do during a dialog, which include:

- Branch – If/Else
- Branch – Switch
- End Dialog
- Repeat Dialog
- Set Property
- Send HTTP Request
- Prompt/Respond (Bot responds with a statement/question)
- User Input
- Delete Properties

It also allows for specific forms to show up as a bot response.

## II-c. Forms

Forms are a vital part of the project that was established to give a better appearance and user friendly way to ask for information. These are specifically defined within the [Bot Responses] section of the Composer, shown below.

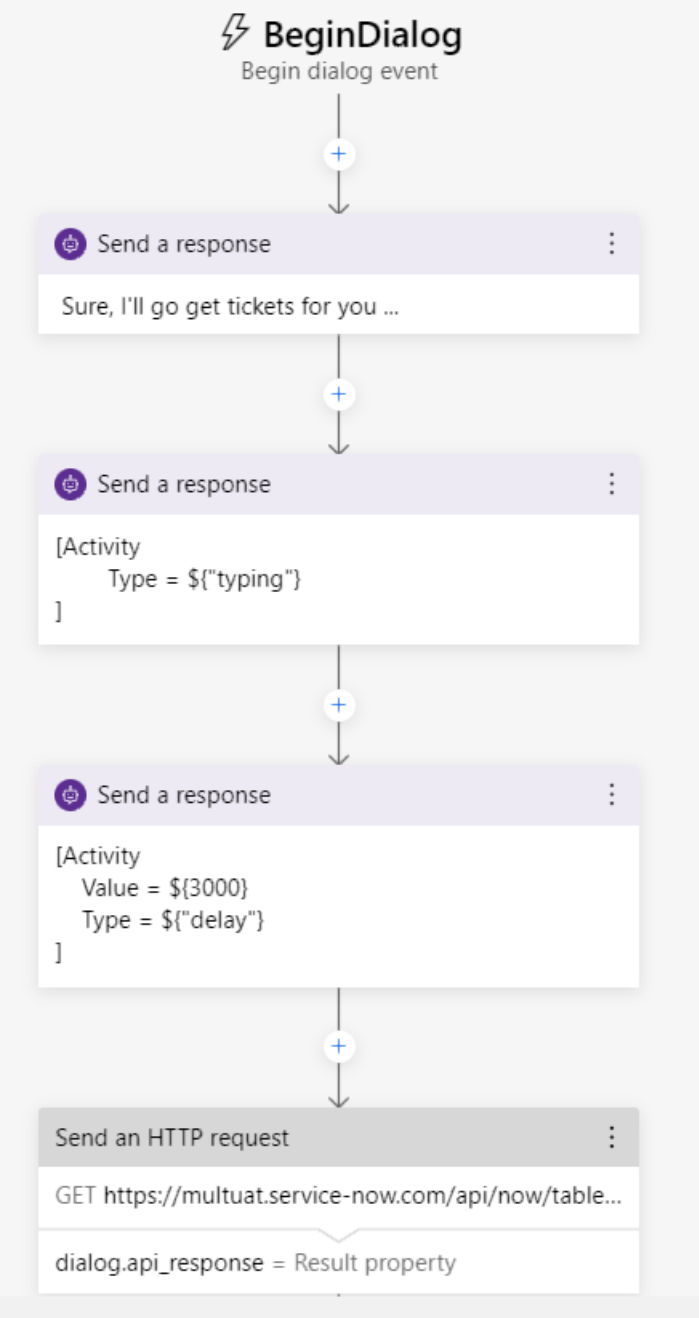



It is located within the Common portion of the responses, which is accessible by any of the dialog options (the other locations are responses dedicated to specific dialog). The way these forms were created was by using Adaptive Cards, with a builder being including in the **IV. External Links** section. These are basic json files that are directly imported into the bot.

## II-d. Creating a Thinking Indicator

There will be certain points in the program that require an indication that the bot is processing something, such as with lengthy API calls and various status checks. In order to indicate to the user that the bot isn't hanging, or has crashed/stopped working for some reason, the bot needs to show off that it is alive.

Adding an indicator of this is surprisingly easy, all you need to add is the "typing" activity before a major delay. This activity will indicate with a moving animation that the bot is attempting to perform a response, but is taking a bit to write it down. This also happens in most text chat programs whenever someone is in the middle of typing, so it gives the bot a more human look while also disguising delays. We also added in an arbitrary delay afterward, as it wouldn't visualize correctly in the Bot Framework without it (it however, works without it on the Web App).




*The exact use of the delay, in the* **GetAllTickets** *dialog*

## II-e. Publishing the Chatbot using Azure

This action only applies if you want to deploy it within the Azure Web App. Specifically, you need to follow these steps to publish the bot:

1. Access the final option on the left within the Composer (symbolized with a wrench icon)
2. Select under the Publish targets option “Add new publish profile”
3. Add a name and select to publish to Web App

4. Login using Azure portal credentials
5. Select “Import existing Azure resources”

Then you will need to paste this JSON into the given textbox:


```
{
 "name": "mc-mcso-servicenowchatbot-dev-rg",
 "environment": "composer",
 "runtimeIdentifier": "win-x64",
 "settings": {
 "applicationInsights": {
 "InstrumentationKey": "926bb52e-0f3d-4a07-8de6-35685e02e0d3"
 },
 "cosmosDb": null,
 "blobStorage": null,
 "luis": {
 "authoringKey": "26b36fbb80c24dbf9568303e93e2e762",
 "authoringEndpoint": "https://westus.api.cognitive.microsoft.com/",
 "endpointKey": "f51927f0bff44d1c9e14212ec0fb2a19",
 "endpoint": "https://westus.api.cognitive.microsoft.com/",
 "region": "westus"
 },
 "MicrosoftAppId": "eb02b2e7-641c-4c91-8f1d-1241d2e30165",
 "MicrosoftAppPassword": "XQ0fCUt7I.JB~CucLWd3-U4lgu.hyQ9ffz"
 },
 "hostname": "mc-mcso-servicenowchatbot-dev-rg",
 "luisResource": "mc-mcso-servicenowchatbot-dev-rg-luis"
}
```


Then you can save and publish the bot using the profile.

# III. Dialog Layout and API Use

This section covers all developed dialogues, where API calls are used, and how to modify them for further dialog in the future. These will be separated by dialog.

**AddComment**

This dialog is a helper dialog that cannot currently be called from the main bot program directly. It allows the user to either add a custom comment to a currently existing ticket/incident, or add in a comment provided when attempting to create the incident/ticket in question. It uses the following API call via PUT:

https://multuat.service-now.com/api/now/table/incident/${user.createIncidentTemp.toUpdateSysID}?sysparm_display_value=true&sysparm_fields=number%2Ccaller_id%2Ccategory%2Cstate%2Cshort_description%2Cassignment_group%2Cassigned_to%2Ccomments

This API call is put after the user decides to add in a comment, with the *user.createIncidentTemp.toUpdateSysID* variable used to identify which ticket/incident to apply the comment towards. There is a slight variation to the call to place a comment on a ticket, with that being shown below.

https://multuat.service-now.com/api/now/table/ticket/${user.createIncidentTemp.toUpdateSysID}?sysparm_display_value=true&sysparm_fields=number%2Cu_requested_for%2Cu_category%2Cstate%2Cshort_description%2Cassignment_group%2Cassigned_to%2Ccomments

**CreateIncident**

This is the primary function of the bot: To input a ticket/incident report to the ServiceNow system. The dialog first does a check if it came from a different location (such as repeating the dialog due to an error), then checks to see if this bot was stopped before it could submit the information and asks if they want to try resubmitting it. If yes, they skip by the actual form and go to submit the data. If not (or if there wasn't a submission interruption), the bot will provide a form to fill out concerning the information, with there being a different form based on it being a Ticket or Incident.

Then we reach the following API Call, which is used in a GET:

https://multuat.service-now.com/api/now/table/incident?sysparm_query=sys_created_by%3Dmdash-api%5Ecategory%3D${user.CreateIncidentTemp.category}&sysparm_display_value=true&sysparm_fields=number%2Cu_customer%2Cu_category%2Cshort_description%2Ccomments%2Cassignment_group%2Cstate

This API uses the given category that is provided to check among the currently existing incidents to give back all that are similar to the one that the user wants to report. There is once again a slight variation for tickets, down below:

https://multuat.service-now.com/api/now/table/ticket?sysparm_query=u_category%3D${user.CreateIncidentTemp.category}%5Esys_created_by%3Dmdash-api&sysparm_display_value=true&sysparm_fields=number%2Cu_requested_for%2Cu_category%2Cshort_description%2Ccomments%2Cassignment_group%2Cstate%2Csys_id

The program will then check to see if there are any repeated incidents/tickets based on the category given. The bot then prompts the user if it wishes to review the other cases (if there are any). If the user doesn't, it goes on as if there weren't any similar incidents. If yes, it will then display each related incident/ticket and ask if this is the same issue. This can go two ways:

If yes, it will record that and ask if the user wanted to comment on the task/ticket. If so, it starts an **AddComment** Dialog here.

If not, then it will end the dialog and return back.

If there isn't a match or the user doesn't want to look over previous tickets/incidents, it goes to actually creating the incident/ticket and submitting this to ServiceNow. This is done in the **PostTaskIncident** dialog.

After all of that, the dialog ends.

**GetAllTickets**

This task is a very simple GET request dialog that allows the user to receive all currently existing tickets. It will report on ALL existing valid tickets/incidents, with little user input, aside from error checking if there was a problem with the callback. The GET API call is this:

https://multuat.service-now.com/api/now/table/task?sysparm_query=sys_created_by=mdash-api&sysparm_display_value=true&sysparm_fields=number,u_requested_for,u_customer,short_description,comments,assignment_groupt,state,sys_id

**PostTaskIncident**

This dialog is a helper dialog, only called after a **CreateIncident** dialog, and only in non-comment cases. This actually submits a brand new ticket/incident to ServiceNow. Initially, it will do a check to make sure the category and description of the ticket exist (if not, it will kill the dialog). It then splits to two different POST calls depending on if it's an incident or a ticket.

The Ticket Information:

Post URL: https://multuat.service-now.com/api/now/table/ticket?sysparm_display_value=true&sysparm_fields=number%2Cu_requested_for%2Cu_customer%2Cshort_description%2Ccomments%2Cassignment_group

Post Body: {"u_requested_for":"ab37c565db851c907c1b78fdaa961951","short_description":"${user.CreateIncidentTemp.description}","u_category":"${user.CreateIncidentTemp.category}","comments":"${user.CreateIncidentTemp.comments}","assignment_group":"dffcb03c7c69600028a2b45fd3ff6758"}

The Incident Information:

Post URL: https://multuat.service-now.com/api/now/table/incident?sysparm_display_value=true&sysparm_fields=number%2Ccaller_id%2Ccategory%2Cshort_description%2Cassignment_group%2Ccomments

Post Body: {"short_description": "${user.CreateIncidentTemp.description}", "caller_id": "ab37c565db851c907c1b78fdaa961951", "category": "${user.CreateIncidentTemp.category}", "assignment_group": "dffcb03c7c69600028a2b45fd3ff6758", "comments": "${user.CreateIncidentTemp.comments}"}

It will then submit the post, and re-attempt to if it fails. It will wipe the temporary data used and return to the main dialog.

**Search**

This dialog is called whenever a user wants to find a specific ticket or incident (or similar ones, if there are multiple). At the beginning, there is a button prompt for the choice of searching through tickets or incidents. Afterward, it will display the specific form to fill out for the choice the user chose (both forms are slightly different). The options to fill out are the *Date* of the incident/ticket, the *State* of the incident/ticket, and the *Category* of the incident/ticket. Each of these are checked in a similar manner.

First, it checks if the value given was filled out at all. If not, it skips to the next variable filled out. Then it will check with a GET command if there are any incidents/tickets matching that variable (Each variable has a different GET command, listed at the end of this section). If it finds any, it will loop through each incident/ticket available. If there aren't any, it will state as such.

After each one is found, there is a final check – if nothing was entered in the first place. The bot will ask the user if they would like to view all tickets/incidents. If yes, it then calls **GetAllTickets** and prints out the result. If not, it kills the dialog and returns to main. If there was a search completed, it will then ask the user if another one would like to be done. If yes, the dialog will repeat. If not, it will end.

Here are the API calls used.

GET Date: https://multuat.service-now.com/api/now/table/task?sysparm_query=sys_created_onON${dialog.date}%40javascript%3Ags.dateGenerate('${dialog.date}'%2C'start')%40javascript%3Ags.dateGenerate('${dialog.date}'%2C'end')%5Esys_created_by%3Dmdash-api&sysparm_display_value=true&sysparm_fields=number%2Cu_requested_for%2Ccategory%2Cu_customer%2Cshort_description%2Ccomments%2Cassignment_group%2Cstate

GET State – Ticket: https://multuat.service-now.com/api/now/table/ticket?sysparm_query=state%3D${dialog.state}%5Esys_created_by%3Dmdash-api&sysparm_display_value=true&sysparm_fields=number%2Cu_customer%2Cu_category%2Cshort_description%2Ccomments%2Cassignment_group%2Cstate

GET State – Incident: https://multuat.service-now.com/api/now/table/incident?sysparm_query=state%3D${dialog.state}%5Esys_created_by%3Dmdash-api&sysparm_display_value=true&sysparm_fields=number%2Cu_customer%2Cu_category%2Cshort_description%2Ccomments%2Cassignment_group%2Cstate

GET Category – Ticket: https://multuat.service-now.com/api/now/table/ticket?sysparm_query=u_category%3D${dialog.category}%5Esys_created_by%3Dmdash-api&sysparm_display_value=true&sysparm_fields=number%2Cu_requested_for%2Cu_category%2Cshort_description%2Ccomments%2Cassignment_group%2Cstate%2Csys_id

GET Category – Incident: https://multuat.service-now.com/api/now/table/incident?sysparm_query=u_category%3D${dialog.category}%5Esys_created_by%3Dmdash-api&sysparm_display_value=true&sysparm_fields=number%2Cu_customer%2Cu_category%2Cshort_description%2Ccomments%2Cassignment_group%2Cstate%2Csys_id

As much as this dialog works, there is a glaring flaw in the design: it is designed as an OR search, rather than an AND search – so it will show all incidents/tickets that match one or the other, rather than just the ones that match both conditions. If this is not the desired result,

it can be fixed rather simply by a check at the start to see if multiple are filled out. Another option is having a ‘default’ value for each variable that, if not changed, will ignore that condition.

# IV. External Links

These are some external resources that are helpful for continued development of this project.

https://docs.microsoft.com/en-us/composer/

^ This link is Microsoft’s own documentation on the composer and how to add on more dialog options that weren’t used in the baseline project.

https://adaptivecards.io/designer/

^ This link is the editor we used to be able to directly see what our JSON output would be for the form appearance. It has a lot of different uses aside from making forms, most of which are able to be used with the composer and the Bots.