ITSM/Ticketing System Integration with AWS Report

Henry Tolenaar

February 15, 2024

Abstract

This report provides a detailed account of the project undertaken to integrate an ITSM/ticketing system with Amazon Web Services (AWS). It outlines the approach taken, challenges encountered, and solutions implemented throughout the project. It is structured as a chronological account of the thought processes and actions taken, highlighting problems faced and their corresponding solutions.

Chronological Account

Upon reviewing the project brief, I decomposed the solution into three logical steps:

1. Present a user-friendly form for issue reporting.
2. Retrieve user data from the form.
3. Place the user data in AWS Simple Queue Service (SQS) queues for further processing.

1 Microsoft Teams Webhook App

The goal was to enable users to input issue information into Microsoft Teams (Teams) in an intuitive format with error checking and data sanitation. Initially, an outgoing webhook was created in Teams to send a POST request to a backend API developed using Flask. This API would then return information to display to the user. A custom app was created for a dedicated Team, and an outgoing webhook was configured to activate when the app was mentioned in a channel. To obtain a callback URL for the webhook to send data to, a free static domain name was set up using ngrok, which port-forwarded requests to localhost where the Flask API would run. The outgoing webhook app was configured to use this domain name as its callback URL, ensuring data from posts mentioning the app would be sent to the backend API.

2 Flask Backend API

Next, a Flask backend API was created to accept POST requests on localhost port 5000, retrieve information from the Teams post, and process it accordingly.

Problem 1: Ports

Initially, data was not appearing in the backend API when the outgoing webhook app was mentioned in a Teams post, despite both ngrok and the Flask API running.

Solution

The issue was identified as a port conflict, with ngrok running on port 80 and the Flask API on port 5000. By changing the ngrok port to 5000, message data from the Teams post was successfully received in the backend API and printed to the console.

3 Adaptive Cards

Problem 2: Data validation

While the system functioned, it lacked data validation, allowing users to post unformatted text in Teams posts. A challenge arose in guiding users on how to format tickets for compatibility with the backend API, as there was no form or structure for the messages.

Teams forms were explored but deemed more suitable for one-time surveys rather than general information capture. Subsequently, Adaptive Cards were investigated as an open card exchange format that enables developers to exchange UI content consistently. A custom adaptive card was created using the Teams Developer Portal, incorporating fields for ticket title, description, and priority selection.

Solution

A key benefit of Adaptive Cards is the ability to require all input areas to be filled and add regular expression pattern requirements. A regular expression was employed to ensure that input fields contained at least one character, preventing users from submitting empty spaces as input.

4 Power Automate

The next challenge was to convey information from the adaptive cards to the backend Flask API.

Problem 3: Workflows

Further research revealed the need for a Power Automate workflow to extract information from the card and transmit it to the backend API. The challenge lay in selecting the appropriate Power Automate blocks and structuring them effectively.

Solution

The solution involved creating a block that listens for a specific keyword in the channel, in this case, "NEW." Upon detecting the keyword, the card is posted, and user input is awaited. Once the user inputs data and presses the submit button, the data is returned to the workflow and used in an HTTP block that sends a POST request to the ngrok URL with the user-inputted data from the card. This data is then forwarded to the Flask backend API. With this step completed, the remaining task is to place the data into the appropriate AWS SQS queues.

5 AWS SQS

The final challenge was to transfer the data from the Flask backend API to AWS SQS queues. Three queues were required, corresponding to the priority levels of low, medium, and high.

Problem 4: Creating the SQS queues

There were two primary methods for creating SQS queues: through the AWS Management Console or through the AWS CLI with the AWS SDK. I opted for the latter approach using boto3, a Python library for interacting with AWS services. However, setting up an IAM user with the necessary permissions was required.

Solution

The AWS Management Console was used to create an IAM user, in an IAM group, which had permissions to do any action within AWS SQS. Next an access token pair was created and used the AWS CLI command:

```
aws configure
```

to allow boto3 to act as the IAM user to create and add to the SQS queues.

Conclusion

With all three logical steps completed, the project was finished and fully functional. The most enjoyable part of this project was learning new tools such as

Power Automate to create a fully automated end-to-end system which incorporates lots of different and diverse services.

Improvements

- More could be added to the end of the system to take the user data out of the queues and process the data with other AWS services, such as AWS Lamba or Fargate.
- Power Automate is able to work directly with AWS to send the data in the adaptive cards straight into AWS SQS queues, omitting the need for the Flask backend API entirely.