AWS Prescriptive Guidance Library workflow notifications through slack API

Achyutha Santhoshi

Dr Johnson P. Thomas, Dr Christopher John Crick, Dr Douglas Roger Heisterkamp

Introduction

AWS Prescriptive Guidance Library[2] is a product of Amazon Web Services. APG Library provides patterns, guides, strategies which help customers to migrate from one cloud platform to another cloud platform. For an example, If a customer wants to migrate from AZURE web services to AMAZON WEB SERVICES

At present, APG Library workflow notifications are sent through E-mail. Though, E-mail is the primary contact for APG users, there is a possibility they miss important time bound notifications due to multiple emails waiting for their attention in their inbox. We want to introduce a new feature which pushes notifications through slack messages to make the process faster.

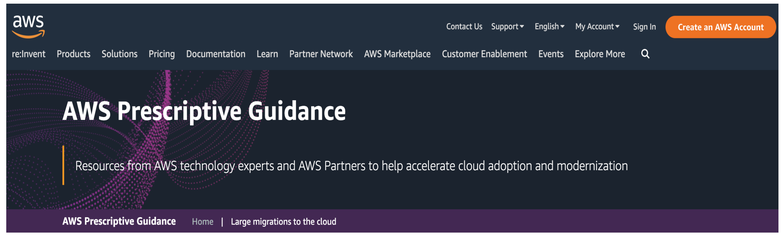

Figure 1. AWS Prescriptive Guidance website

Related Work

Previously implemented methods for Slack Integration in AWS,

- Webhooks
  - A webhook needs to be setup in slack workspace manually and then the hook url is used to post request to slack
  - Limitation: In webhooks, we need to provide the channel name to post messages. We can provide channel name for the public channel. Since the private channels are intended to be created when a context is created, we can't use this approach for private channel
- Slack Bot[3]
  - Implemented with Node.js and uses events api to post messages
  - Events API is used for receiving notifications from slack

Method used for Slack Integration for this project:
WebAPI and Conversational API[1]

- Similar to REST APIs, The Web API is a collection of HTTP RPC-style methods
- We can pass the bearer token for authentication in the request

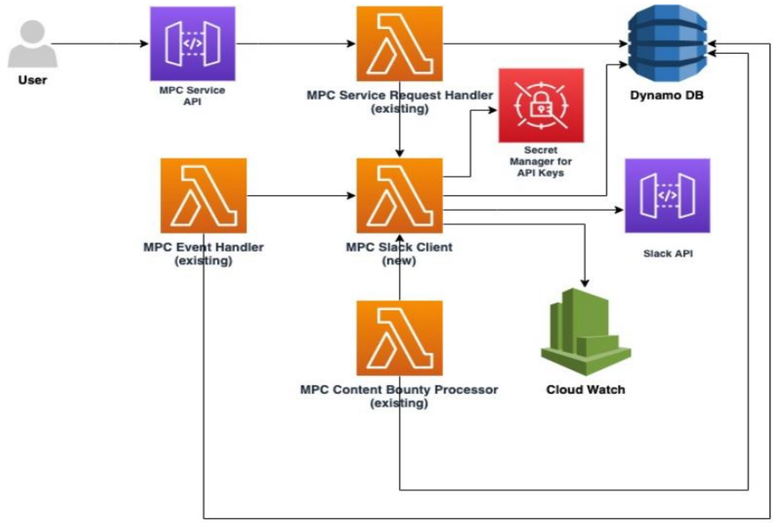

Figure 2. High level architecture for Slack Client Integration

Technical Approach

•A new Slack Sandbox[4] is created for testing these scenarios, Slack app "APGL-Notifications" is created within the sandbox
•Then a "MPC Slack Client" lambda is created as part of MPC Service which handles all the slack notifications on the existing "MPC Service Request Handler", "MPC Event Handler", "MPC Content Bounty Processor" methods which are now changed to call Slack Client
•The new slack client uses secrets manager to retrieve the token used for calling the slack application to publish informational messages, errors to the CloudWatch logs
•New "MPC Slack Client" publishes metrics related to channels creation, deletion and number of notifications sent
•User notification preferences are obtained from Dynamo DB
•Slack APIs[1] that will be used: conversations.create, conversations.invite, conversations.kick, conversations..archive, chat.postMessage, user.lookupByEmail

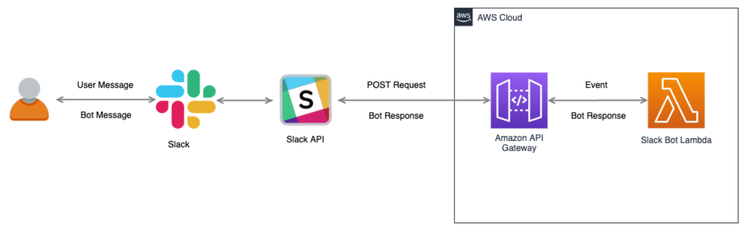

Figure 3. High level architecture for Slack Client and Slack API integration

Results

- Created a method to create a channel and invite members by passing channel name. This method allows us to create a channel and invite the required members to join the channel. If channel exists, it checks and invites members if not already on the channel.
  - **createChannel (channelName, members?)**
- Created a method to post to a public channel by passing channel name, message.
  - **notifyPublicChannel(channelName, message)**
- Created a method to post to a private channel by passing channel name, message. This method creates channel if it doesn't exist and optionally adds members
  - **notifyPrivateChannel (channelName, message, members?)**
- Created a method to add members to existing private channel : This method allows us to add members to the existing channel
  - **changeChannelMembers(channelName, members)**
- Created a method to delete members from existing private channel : This method allows to delete members from the existing channel
  - **changeChannelMembers(channelName, members)**
- A method to check whether the channel exists
  - **checkChannelExists(channelName)**
- A method to delete the channel
  - **DeleteChannel(channelName)**
- Before adding member to slack channel, user preference is checked using UserResourceDao→getUserProfile() method
- All methods log informational and error messages to cloudwatch logs
- Throws error back from lambda in case of errors so that lambda framework retries requests.
- Slack API token will be retrieved from SecretsManager → apgl/slack-api-token

Conclusions

- The lambda that invokes the slack client lambda does not wait for the lambda to finish and immediately returns.
- Pros:
  - it is cost effective as only one lambda is in process at a time
  - It does not impact request processing
  - if the lambda function fails, retries are the responsibility of the lambda framework
  - Manage and deploy changes independently
  - Easier to scale by having a separate lambda
- Cons:
  - It can be unstable and unpredictable because some asynchronous invocations may get canceled and some might even run multiple times(double runs).
  - Some function executions would get delayed, Which is acceptable in our case

References

[1]https://api.slack.com/
[2]https://aws.amazon.com/prescriptive-guidance/
[3]https://docs.aws.amazon.com/lex/latest/dg/slack-bot-assoc-create-app.html
[4]https://aws.amazon.com/api-gateway/