

Slack Bot

The purpose of the bot is to assist users within their slack workspace by providing automated responses and performing user-specific actions.

Steps to create the Bot:

1) Set up the Slack App:

- ☐ Visit <https://api.slack.com/apps>
- ☐ Click on **Create New App** and choose **From Scratch**
- ☐ Enter a name for the app and choose the workspace in which you want to install the bot, and click on **Create App**

2) Set up the bots permissions and scopes:

- ☐ In your app's **Features** go to **OAuth & Permissions**
- ☐ Scroll down to find the **Bot User OAuth Token under OAuth tokens**. Note is down. (used in the code to authenticate the bot and provide sit with necessary permissions to interact with the workspace)
- ☐ Scroll down to the **Scopes** section and add the following tokens(based on the functionalities of the app)

Bot level tokens:

- ❖ **App_mentions_read**: View messages that directly mention @testbot in conversations that the app is in
- ❖ **channels:history**: View messages and other content in public channels that testbot has been added to
- ❖ **channels:manage**: Manage public channels that the testbot has been added to to create new ones
- ❖ **Channels:read**: View basic information about public channels in a workspace
- ❖ **chat:write**: Send messages as @testbot (bot name)
- ❖ **Commands**: add shortcuts and/or slash commands that people can use
- ❖ **im:history**: View messages and other content in direct messages that testbot has been added to
- ❖ **im:read**: View basic information about direct messages that testbot has been added to
- ❖ **im:write**: Start direct messages with people
- ❖ **mpim:history**: View messages and other content in group direct messages that the testbot has been added to.
- ❖ **mpim:read**: View basic information about group direct messages that testbot has been added to
- ❖ **mpim:write**: Start group direct messages with people

User-level tokens:

- ❖ **im:history:** View messages and other content in a user's direct messages

- ☐ Click **Install App to Workspace** and authorize the app. A Bot User OAuth Access Token will get created.

3) Set up Development Environment:

- a) Can be deployed on the cloud
- b) Can set up local development environment (explained below)
 - ☐ Create a new directory for the project and install the required dependencies
 - ☐ Create the bot script
 - ☐ Deploy the bot using a cloud platforms like AWS or azure or tools like Ngrok to expose the local server to the internet

4) Update Slack App Configuration:

- ☐ In your slack app **Features**, go to **Event Subscriptions**
- ☐ **Turn on Enable Events** and set the Request URL to the ngrok URL with /slack/events endpoint. For example, **https:// xyz.ngrok-free.app/slack/events**. We can find the Request URL in the terminal where we ran the ngrok http 3000 command.
- ☐ In your slack app **Features**, go to **Slash Commands**
- ☐ Create a **new slash command**(eg: /testbot) and set the Request URL to the ngrok URL with /slack/commands as the endpoint. For example: **https:// xyz.ngrok-free.app/slack/commands**
- ☐ In your slack app **Features**, go to **Interactivity & Shortcuts**
- ☐ **Turn on Interactivity** and set the request URI to the Ngrok URL with /slack/interactivity as the endpoint. For example: **https:// xyz.ngrok-free.app/slack/interactivity**.

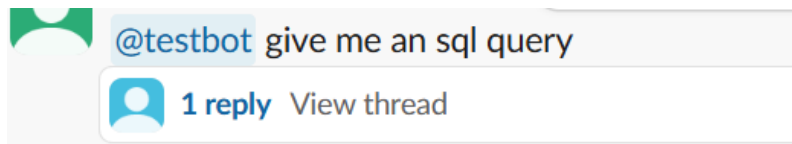
5) Test your bot:

- ☐ Invite the bot to the channels in the slack workspace where the app was created/installed

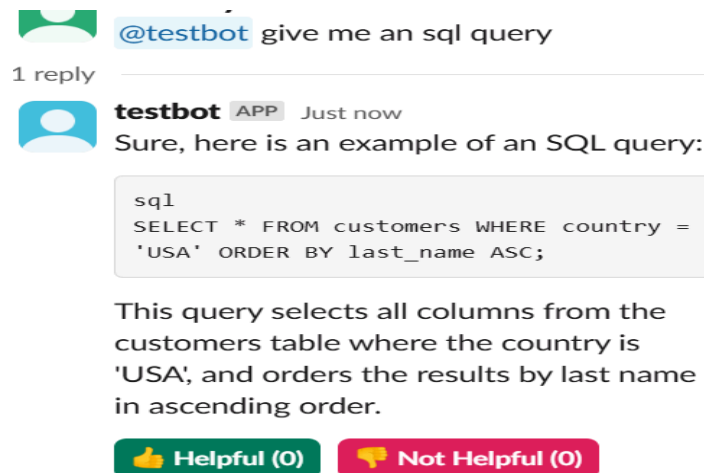
Functionalities added to the slack bot:

- 1) The bot has been integrated with OpenAI capabilities to have advanced processing and replying capabilities. When the bot is mentioned or triggered, it has the functionality to reply.
- 2) @testbot: When we mention the testbot by tagging it using @, it has the capability to respond to the message directly in the channel or chat it was tagged in, in the form of a thread. For example:

When we open the thread we can see the bot's response



We can continue the chat in the same thread and the bot has the functionality to reply to follow up questions. The bot has a memory feature which allows it to remember the context of previous messages, maintaining the flow of the conversations. From the image, we can see that the response also comes with two buttons, “**helpful**” and “**not helpful**”. Users can vote for the usefulness of the bot's response by clicking on the respective buttons. When a user clicks on the button the count represented in brackets(ex: helpful (0)) increases or decreases based on whether the user selects or deselects the button)



Ex: If the user feels like the bot's response is helpful they can click on the button and the the UI of the helpful button will change from helpful(0) to helpful (1), and when the user deselects the option, it will change from helpful(1) to helpful(0).

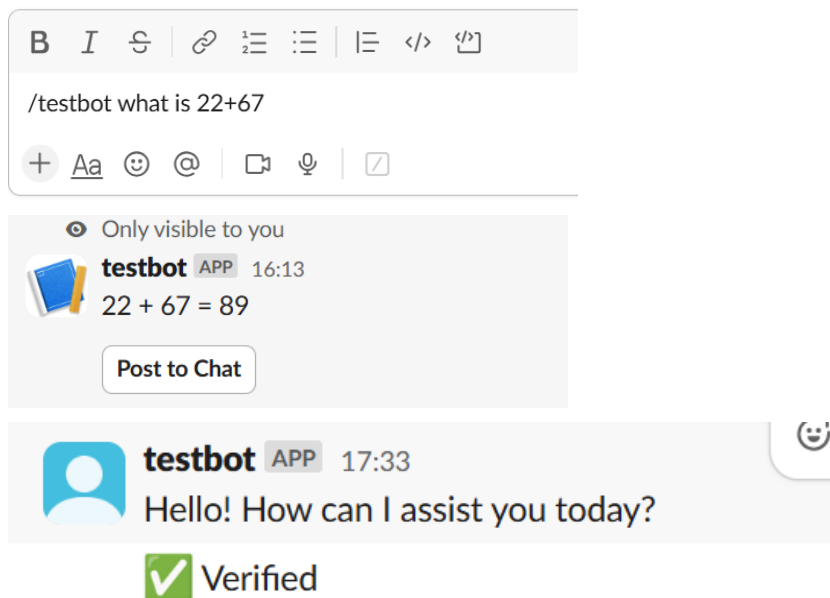
This query selects all columns from the customers table where the country is 'USA', and orders the results by last name in ascending order.

 Helpful (2)

 Not Helpful (0)

The functionality is working well as of now. Might have to do more intensive testing in the later stages to test the functionality of the count.

- 3) `/testbot` : this command triggers the testbot and we can ask it questions to which it will give an ephemeral reply which is only visible to you. You then have a button “**post to chat**” which can be used to post the message in that chat, which makes it visible to everyone.

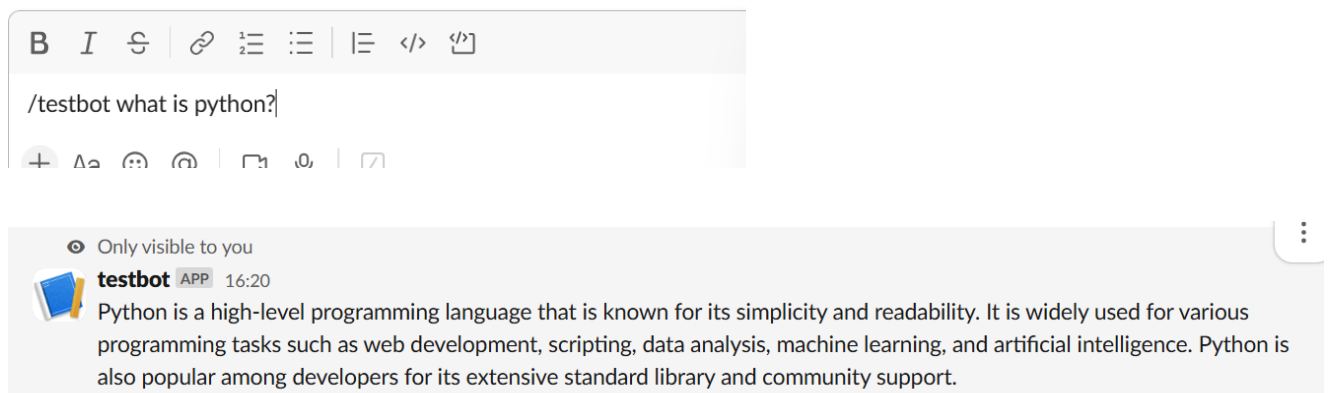


By clicking on the **post to chat** button it posts the reply in the chat, making it visible to everyone in the channel or group. It also shows verification which indicates that the user who asked the query and posted in chat has verified the response..

Since the replies given by the bot when we trigger it using `/testbot` are ephemeral, once we refresh the page, they disappear unless we post it to the chat.

Sometimes, they do not disappear, this is a result of caching. In order for them to disappear we need to clear the cache by logging out if on the web or clearing app cache if using the desktop app.

The **post to chat** option is not visible in private chat between users, as the bot does not have permission to post messages in chats it is not a part of. But, we can use /testbot to ask it any question and it will give an ephemeral response(visible only to the user who asked the question). In that case if we want to post the message in the chat, we need to manually copy and paste the message.This functionality was introduced by slack in order to ensure privacy between users. In a private chat it is not possible to tag/ mention (@testbot) the bot as it is not present in the chat.



If we want the bot to reply in a conversation with another user, we need to invite the bot to the chat, this creates a group with the user and the bot. We can mention/tag the bot (@testbot) to ask a question which will prompt the bot to reply in a thread as we have seen before. We can also trigger the bot by /testbot in order to produce an ephemeral response to the query which can then be posted in the chat by using the **post to chat** option.

A group can be created by clicking on the “+” symbol near the **direct messages** and adding the specific users and bot.

Deletion of ephemeral messages: So far, by using the slash command to trigger the bot(/testbot) gives us ephemeral messages which can be posted to chat by using the button. The very nature of ephemeral messages (the ones that are only visible to you) is that they disappear after the session is restarted or the user logs out and logs back in. There is absolutely no way we can automatically delete these messages after a certain timeframe.

During testing, we observed that the ephemeral messages do not disappear a few times even after refreshing or closing the session. In such cases, we had to log out and log back in for the cache to get cleared, or we had to clear the cache in order for those messages to disappear.

According to the slack documentation, the responses resulting from the slash command cannot be deleted by using direct calls like `chat.delete` or `delete_original`, given by slack. These responses also cannot be updated by using `replace_original`, which is why clearing cache is the only option we have so far in order to get rid of the few ephemeral (visible to you) messages left.

The `delete_original` field is not supported for use with [slash commands](#), so sending it will not remove or retain a user-posted message that is invoked by the slash command.


You cannot use `replace_original` to modify the user-posted message that invokes a [slash command](#).

Problems/Challenges(which could not be solved):

When the bot is triggered using `/testbot` we sometimes get an “operation timed out” message. Slack slash commands have an acknowledgement time of 3 seconds, and if it does not acknowledge it within 3 seconds we get the above error. Basically, the request has been sent but the API has not been able to respond back within 3 seconds due to various delays like latency or response time issues by OpenAI. It can also occur if the bot’s processing time itself takes more than 3 seconds.

Could not find any fix for this problem so far. Also, this problem does not occur when we mention the bot using `@testbot`.

Link to the code:

 [slackbot code.ipynb](#)

Summary of the code:

The code sets up the functionalities of a slackbot, which stores message and feedback data in databases. It uses the OpenAI Api key to generate responses. The code keeps track of the count of the button clicks in order to track feedback and stores the feedback for each message and user specifically, which are all aggregated based on the message id and displayed on the button when clicked. The bot can respond to mentions and triggers(slash) but using their own functionalities. When mentioned the bot replies in a thread and when triggered it gives an ephemeral reply which can be posted to chat.

CODE:

```

import os
from flask import Flask, request, jsonify
from slack_sdk import WebClient
from slack_sdk.errors import SlackApiError
import openai
import json
import sqlite3

# Replace with your bot token( details in the document) and OpenAI API key
SLACK_BOT_TOKEN = 'xoxb-8318598919778-8317026072743-xsVaN5LzDbCIIPiIGnf93LvF'
OPENAI_API_KEY =
'sk-proj-gAlNZf8h6z6F8Di19Y9oxyTMoiQ_ON3wSqcMhxgAWuFuoLN4K0dfc25o5TAybfEGPxOzKfpch1T3B
lbkFJf4GpzcLTOKBkH0wpPYjoOZaBNVYpJ70WSZNJ6-sEYCuowE7A2CbiPNme4pKCRQnvKlA42rC0AA'

client = WebClient(token=SLACK_BOT_TOKEN) #enables bot to communicate with the slack
API
app = Flask(__name__)
openai.api_key = OPENAI_API_KEY

PROCESSED_MESSAGES = set() # keeps track of all the messages that have been processed
MAX_PROCESSED_MESSAGES = 100

# Initialize SQLite database to store the processed messages
conn = sqlite3.connect('processed_messages.db', check_same_thread=False)
c = conn.cursor()
c.execute('''CREATE TABLE IF NOT EXISTS messages (ts TEXT PRIMARY KEY)''') # table
called message is created, with the timestamp of the message as primary key
c.execute('''CREATE TABLE IF NOT EXISTS feedback (
    message_ts TEXT,
    user_id TEXT,
    feedback_type TEXT,
    PRIMARY KEY (message_ts, user_id)
)''')# table called feedback is created to store the feedback button data which
contains the timestamp of message and user_id as the primary key in order to
differentiate between different users and their feedback

conn.commit()

#the following function is used to check if the messages are duplicate or not (returns
True if duplicate) If it is not a duplicate it records the timestamp and returns
False.
def is_duplicate_message(ts):
    """Check if the message is a duplicate."""
    c.execute('SELECT ts FROM messages WHERE ts=?', (ts,))
    if c.fetchone():
        return True

```

```

c.execute('INSERT INTO messages (ts) VALUES (?)', (ts,))
conn.commit()
return False

#this function generates responses to the queries using the OpenAI API key
def generate_ai_response(text):
    try:
        response = openai.ChatCompletion.create(
            model="gpt-3.5-turbo",
            messages=[{"role": "user", "content": text}],
            max_tokens=150
        )
        return response.choices[0].message['content'].strip()
    except Exception as e:
        return f"Error: {str(e)}"

#retrieves the count for each feedback option and stores it in a dictionary "counts"
def get_feedback_counts(message_ts):
    """Get the current helpful and not helpful counts for a message."""
    c.execute('''
        SELECT feedback_type, COUNT(*)
        FROM feedback
        WHERE message_ts = ?
        GROUP BY feedback_type
    ''', (message_ts,))

    results = c.fetchall()
    counts = {'helpful': 0, 'not_helpful': 0}

    for feedback_type, count in results:
        if feedback_type == 'helpful_feedback':
            counts['helpful'] = count
        elif feedback_type == 'not_helpful_feedback':
            counts['not_helpful'] = count

    return counts

# fetches the feedback of the specific user for a specific message
def get_user_feedback(message_ts, user_id):
    """Get the current feedback from a specific user for a message."""
    c.execute('''
        SELECT feedback_type
        FROM feedback
        WHERE message_ts = ? AND user_id = ?
    ''', (message_ts, user_id))

    result = c.fetchone()

```



```

        return result[0] if result else None

#handles the way the count increases and decreases in the databse by using insert and
delete commands,the functionality of the buttons
def handle_feedback_action(message_ts, user_id, action_id, channel_id):
    """Handle feedback action by updating the database."""
    try:
        current_feedback = get_user_feedback(message_ts, user_id)

        if current_feedback == action_id:
            c.execute('''
                DELETE FROM feedback
                WHERE message_ts = ? AND user_id = ?
            ''', (message_ts, user_id))
        else:
            c.execute('''
                DELETE FROM feedback
                WHERE message_ts = ? AND user_id = ?
            ''', (message_ts, user_id))

            c.execute('''
                INSERT INTO feedback (message_ts, user_id, feedback_type)
                VALUES (?, ?, ?)
            ''', (message_ts, user_id, action_id))

        conn.commit()
        return True
    except Exception as e:
        conn.rollback()
        return False

# Buttons creation for feedback ( helpful and not helpful along with the counts)
def create_feedback_buttons(message_ts):
    """Create initial feedback buttons block for messages with counters."""
    counts = get_feedback_counts(message_ts)
    return [
        {
            "type": "section",
            "text": {"type": "mrkdwn", "text": ""}
        },
        {
            "type": "actions",
            "block_id": f"feedback_{message_ts}",
            "elements": [
                {
                    "type": "button",

```

```

        "text": {"type": "plain_text", "text": f"👍 Helpful
({counts['helpful']})"},
        "value": f"helpful_{message_ts}",
        "action_id": "helpful_feedback",
        "style": "primary"
    },
    {
        "type": "button",
        "text": {"type": "plain_text", "text": f"👎 Not Helpful
({counts['not_helpful']})"},
        "value": f"not_helpful_{message_ts}",
        "action_id": "not_helpful_feedback",
        "style": "danger"
    }
]
}
]

#updating the feedback buttons when clicked- updates the count based on the counts
dictionary mentioned before
def update_feedback_buttons(message_ts, user_id, channel_id, message_text, counts):
    """Update the message with current feedback counts."""
    blocks = [
        {
            "type": "section",
            "text": {"type": "mrkdwn", "text": message_text}
        },
        {
            "type": "actions",
            "block_id": f"feedback_{message_ts}",
            "elements": [
                {
                    "type": "button",
                    "text": {"type": "plain_text", "text": f"👍 Helpful
({counts['helpful']})"},
                    "value": f"helpful_{message_ts}",
                    "action_id": "helpful_feedback",
                    "style": "primary"
                },
                {
                    "type": "button",
                    "text": {"type": "plain_text", "text": f"👎 Not Helpful
({counts['not_helpful']})"},
                    "value": f"not_helpful_{message_ts}",
                    "action_id": "not_helpful_feedback",
                    "style": "danger"
                }
            ]
        }
    ]

```

```

    ]
}
]

try:
    result = client.chat_update(
        channel=channel_id,
        ts=message_ts,
        blocks=blocks
    )
except SlackApiError as e:
    pass

## handles events where the bot is mentioned(@testbot), ignores the bot's own
messages( also includes the generation of feedback buttons along with the response)
def handle_bot_mention(event):
    try:
        ts = event.get("ts")
        if is_duplicate_message(ts):
            return

        bot_id = client.auth_test()["user_id"]
        text = event.get("text", "")
        thread_ts = event.get("thread_ts")

        if event.get("user") == bot_id:
            return

        if f"<@{bot_id}>" in text:
            cleaned_text = text.replace(f"<@{bot_id}>", "").strip()
            response = generate_ai_response(cleaned_text)

            blocks = create_feedback_buttons(ts)
            blocks[0]["text"]["text"] = response

            client.chat_postMessage(
                channel=event.get("channel"),
                thread_ts=thread_ts or ts,
                text=response,
                blocks=blocks
            )
    except Exception:
        pass

##handles /testbot triggers and sends responses based on whether it is a Dm or
channel.
@app.route('/slack/commands', methods=['POST'])

```

```

def handle_slash_command_request():
    data = request.form
    text = data.get('text')
    channel_id = data.get('channel_id')

    response = generate_ai_response(text)

    if channel_id.startswith('D'):
        return jsonify({
            'response_type': 'ephemeral',
            'text': response
        })
    else:
        return jsonify({
            'response_type': 'ephemeral',
            'text': response,
            'blocks': [
                {
                    "type": "section",
                    "text": {"type": "mrkdwn", "text": response},
                    "block_id": "response_block"
                },
                {
                    "type": "actions",
                    "elements": [
                        {
                            "type": "button",
                            "text": {"type": "plain_text", "text": "Post to Chat"},
                            "value": f"{channel_id}|{response}|",
                            "action_id": "post_to_chat"
                        }
                    ]
                }
            ]
        })

# challenge message to establish connection to slack events
@app.route('/slack/events', methods=['POST'])
def slack_events():
    data = request.get_json()

    if 'challenge' in data:
        return jsonify({'challenge': data['challenge']})

    if data.get('event', {}).get('type') == 'message':
        handle_bot_mention(data['event'])

```

```

    return jsonify({'status': 'ok'})

#handles the functionalities of the interactive components like buttons.
@app.route('/slack/interactivity', methods=['POST'])
def handle_interactivity():
    try:
        payload = json.loads(request.form.get('payload', '{}'))

        if not payload:
            return jsonify({'error': 'No payload'}), 400

        action = payload.get('actions', [{}])[0]
        channel_id = payload['channel']['id']

        if action.get('action_id') == 'post_to_chat':
            try:
                value = action['value']
                value_parts = value.split('|')

                if len(value_parts) == 2:
                    channel_id, message = value_parts
                    thread_ts = None
                elif len(value_parts) == 3:
                    channel_id, message, thread_ts = value_parts
                else:
                    raise ValueError("Unexpected value format")

                post_kwargs = {
                    'channel': channel_id,
                    'text': message
                }

                if thread_ts:
                    post_kwargs['thread_ts'] = thread_ts

                client.chat_postMessage(**post_kwargs)

                client.chat_postMessage(
                    channel=channel_id,
                    text=":white_check_mark: Verified",
                    thread_ts=thread_ts
                )

            return jsonify({
                'response_type': 'ephemeral',
                'replace_original': True,
                'delete_original': True,

```

```

        'text': 'Message posted to chat successfully with verification!'
    })

    except Exception as e:
        return jsonify({
            'response_type': 'ephemeral',
            'text': f"Error: {str(e)}"
        }), 200

    if action.get('action_id') in ['helpful_feedback', 'not_helpful_feedback']:
        try:
            message_ts = payload['message']['ts']
            user_id = payload['user']['id']

            handle_feedback_action(message_ts, user_id, action['action_id'],
channel_id)

            counts = get_feedback_counts(message_ts)

            update_feedback_buttons(
                message_ts,
                user_id,
                channel_id,
                payload['message']['blocks'][0]['text']['text'],
                counts
            )

            return jsonify({'status': 'ok'})
        except Exception as e:
            return jsonify({
                'status': 'error',
                'message': str(e)
            }), 500

    return jsonify({'status': 'ok'})
except Exception:
    return jsonify({'error': 'Unexpected error'}), 500

#to start flask application
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=int(os.environ.get('PORT', 3000)))

```