Spend-Wise Documentation:

USER REFERENCE:

SpendWise is a user-friendly Telegram Bot designed to simplify the task of recording daily expenses and efficiently manage recurring expenditures. In a world where tracking personal finances can be a daunting endeavor, SpendWise comes to the rescue. It offers a range of commands that empower users to effortlessly manage their financial transactions within the Telegram platform.

SpendWise application provide users with a hassle-free solution for keeping tabs on their daily expenses and monitoring recurring bills. With SpendWise, users can easily record and categorize their spending, set budgets, visualize expenditure patterns, and more, all within the familiar Telegram interface.

SpendWise has all these features:

"menu": "Display this menu",

"addRecurring": "Recording/ Adding a new recurring expense",

"showRecurringTransactions": "Display all the recurring transactions",

"displayUpcomingTransactions": "Display upcoming bills in the current month",

"add": "Record/Add a new spending",

"display": "Show sum of expenditure for the current day/month",

"history": "Display spending history",

"delete": "Clear/Erase all your records",

"edit": "Edit/Change spending details",

"budget": "Set budget for the month",

"chart": "See your expenditure in different charts",

"categoryAdd": "Add a new custom category",

"categoryList": "List all categories",

"categoryDelete": "Delete a category",

"download":"Download your history",

"displayDifferentCurrency": "Display the sum of expenditures for the current day/month in another currency", "sendEmail":"Send an email with an attachment showing your history",

"addSavingsGoal": "Record your target spending"

DEVELOPER REFERENCE AND DEVELOPER PAINPOINTS

Telegram Bot API Specific

1.
```
chat_id = str(message.chat.id)
user = user_list[chat_id]
```

chat_id: In the Telegram Bot API, each chat (individual or group conversation) has a unique identifier known as chat_id. It helps the bot know where to send responses.

message.chat.id: This is a part of the Telegram API's message object. It is used to extract the chat_id from the message received by the bot. Essentially, it helps identify which chat the message came from.

`user = user_list[chat_id]` – This line is used to retrieve a user object from a data structure(named user_list) using the chat_id. It will fetching the user object associated with the specific chat_id.

2.
```
bot.send_message(chat_id, "MESSAGE_CONTENT")
```

bot.send_message(chat_id, "Please select a menu option from below:") is a function that sends a text message to the user or chat identified by chat_id with the content "MESSAGE_CONTENT"

3.
```
bot.reply_to(message, "MESSAGE_CONTENT")
```
This will give a reply to the message passed to this function and add the "MESSAGE_CONTENT" as a chat reply to the previous message

4.
```
@bot.message_handler(commands=["COMMAND_NAME"])
def command_function (message):
```
Here, @bot.message_handler(commands=["COMMAND_NAME"])
This code defines a handler for a specific command in the Telegram bot. When the user sends the specified command, the command_function will be called, taking a message as a parameter. This function can be customized to perform actions based on the received command. The @bot.message_handler(commands=["COMMAND_NAME"]) decorator specifies which command triggers this function.

5.
```
bot.register_next_step_handler(message, FUNCTION_NAME, param1, param2…)
```

This method registers the next step to be executed. Here the next step to be execution is FUNCTION_NAME(param1, param2)

USING SpendWise API

1. start_and_menu_command

```
Handles the commands 'start' and 'menu'. Sends an intro text.

    :param m: telebot.types.Message object representing the message object
    :type: object
    :return: None
```

2. command_budget(message):

```
    Handles the commands 'budget'. Gets input for budget and stores it.

    :param message: telebot.types.Message object representing the message object
    :type: object
    :return: None
```

3. command_add(message):

```
    Handles the command 'add'. This function records the transaction and stores it
in user.transactions
    :param message: telebot.types.Message object representing the message object
    :type: object
    :return: None
```

4. command_addRecurring(message):

```
     Handles the command 'addRecurring'. This function records the Recurring
Transaction and adds it to the user.recurring_transactions

    :param message: telebot.types.Message object representing the message object
    :type: object
    :return: None
```

5. display_upcoming_transactions(message):

```
     Display the list of Upcoming Transactions for the current month i.e. Bills
Due for Payment in the current month
```

6. show_history(message):

```
    Handles the command 'history'. Lists the transaction history.

    :param message: telebot.types.Message object representing the message object
    :type: object
    :return: None
```

7. `download_history(message):`

```
Handles the command 'download'. Downloads a csv file.

:param message: telebot.types.Message object representing the message object
:type: object
:return: None
```

8. `send_email(message):`

```
Handles the command 'sendEmail'. Sends an email to the user with the csvFile.
```

9. `command_display(message):`

```
 Handles the command 'display'. If the user has no transaction history, a
message is displayed. If there is any transaction history, user is given choices
of time periods to choose from – Current Day or Current Month and the transactions
along with the total amount is displayed.
    :param message: telebot.types.Message object representing the message object
    :type: object
    :return: None
```

10. `command_display_currency(message):`

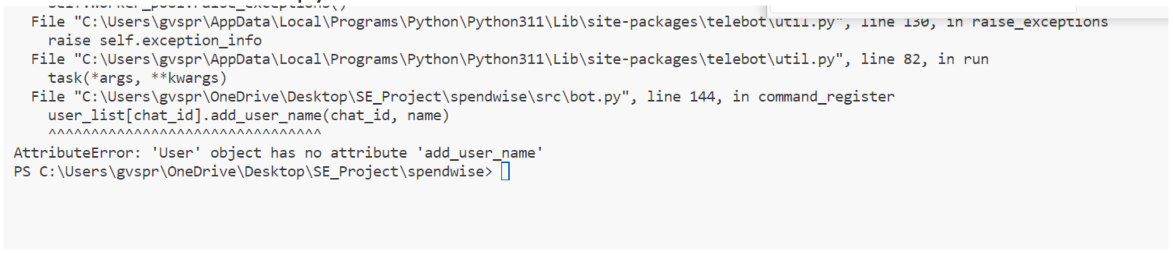
```
    This will fetch the real-time currencies conversion rates and then display
the total_amount_spent in the selected currency.

    :param message: telebot.types.Message object representing the message object
    :type: object
    :return: None
```

Some Common Issues that you might typically face while using the project:

1. This project uses pickle to store the user data. So you typically might have to delete the pickle files if you store anything in the user object for e.g. If you add a new attribute or method in the class ‘User’ you typically must create a delete the existing pickle files so that new pickle file in order for that change to get reflected. The pickle file will automatically be created when you run the project.

Example: In this case, we have tried to access the add_user_name already created in the User file but we did not delete the pickle files which are already existing so it is not reflected to the bot.py function .



```
  File "C:\Users\gvspr\AppData\Local\Programs\Python\Python311\Lib\site-packages\telebot\util.py", line 130, in raise_exceptions
    raise self.exception_info
  File "C:\Users\gvspr\AppData\Local\Programs\Python\Python311\Lib\site-packages\telebot\util.py", line 82, in run
    task(*args, **kwargs)
  File "C:\Users\gvspr\OneDrive\Desktop\SE_Project\spendwise\src\bot.py", line 144, in command_register
    user_list[chat_id].add_user_name(chat_id, name)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'User' object has no attribute 'add_user_name'
PS C:\Users\gvspr\OneDrive\Desktop\SE_Project\spendwise>
```

2. `bot.register_next_step_handler(message, FUNCTION_NAME, param1, param2…)`
   `calls the function with decorator`

```
@bot.message_handler(commands=["COMMAND_NAME"])
      def FUNCTION_NAME (message, param1, param2..):
```

3. Make sure to update these according to your local system's attributes.

```
from user import User
sys.path.append("YOUR_LOCAL_PATH")
try:
   from src.user import User
except:
   from user import User
api_token = "YOUR_TELEGRAM_BOT_TOKEN"
```