hw6

June 12, 2024

- Allow the user to submit messages to the bank.
- Allow the user to view a sample of the messages currently stored in the bank.
- screenshots
- discuss python functions

Allow the user to submit messages to the bank. Allow the user to view a sample of the messages currently stored in the bank.# Homework 6: Web Development

Due: Friday, June 7th, 8:30 AM, late submissions accepted without penalties until Sunday, June 9th, 8:30 AM. This may be extended if the situation allows.

In this exercise, you will create a simple webapp using Dash by Plotly and describe how you did it. The skills you will need are:

1. Dash fundamentals, including callbacks.
2. Database skills, including adding items to databases and displaying them.

You are not required to deploy your app to the internet, although you are certainly welcome to do so if you wish.

0.1 Overview

The app you're going to build is a simple message bank. It should do two things:

1. Allow the user to **submit** messages to the bank.
2. Allow the user to **view** a sample of the messages currently stored in the bank.

Additionally, you should style your app to look attractive and interesting! I encourage you to get creative on this.

Your Jupyter Notebook will contain all the code for the app so that when it is exported as a `.py` file, it can run a webapp with the command `python hw6.py`. It should also contain several screencaps from the functioning of your app, as well as a discussion of the Python functions you implemented to create your app.

You are free to (and indeed encouraged) build on any of the examples from class, as well as any other resources you are able to find. The lecture materials are good starting points.

The code for your app must be hosted in a private GitHub repository. I suggest you begin by creating such a repository. Commit and push each time you successfully add a new piece of functionality or resolve a bug. **You may be asked to make it public after June 9th, 8:30 am for peer review**.

0.2 Instructions

0.2.1 1. Enable Submissions

First, create a `submit` functionality in `app.py` with three user interface elements:

1. A text box for submitting a message.
2. A text box for submitting the name of the user.
3. A "submit" button.

Now, write two Python functions for database management in the file `app.py`.

- `get_message_db()` should handle creating the database of messages.
  1. Check whether there is a database called `message_db` defined in the global scope. If not, then connect to that database and assign it to the global variable `message_db`. To do this last step, write a line like `message_db = sqlite3.connect("messages_db.sqlite")`
  2. Check whether a table called `messages` exists in `message_db`, and create it if not. For this purpose, the SQL command `CREATE TABLE IF NOT EXISTS` is helpful. Give the table a `handle` column (text) and a `message` column (text).
  3. Return the connection `message_db`.
  4. Here is a helpful starter code:

```
message_db = None
def get_message_db():
  # write some helpful comments here
  global message_db
  if message_db:
      return message_db
  else:
      message_db = sqlite3.connect("messages_db.sqlite", check_same_thread=False)
      cmd = '' # replace this with your SQL query
      cursor = message_db.cursor()
      cursor.execute(cmd)
      return message_db
```

- The function `insert_message(handle, message)` should handle inserting a user message into the database of messages.
  1. Using a cursor, insert the message into the `message` database. Remember that you'll need to provide the handle and the message itself. You'll need to write a SQL command to perform the insertion.
  - **Note**: when working directly with SQL commands, it is necessary to run `db.commit()` after inserting a row into `db` in order to ensure that your row insertion has been saved.
  - A column called `rowid` is automatically generated by default. It gives an integer index to each row you add to the database.
  - Close the database connection within the function!
- Finally, write a callback function `submit()` to update the components. Maybe it would be nice to add a small note thanking the user for their submission and print an error message if it failed.
  1. Extract the `handle` and the `message` from the components. You'll need to ensure that your callback deals with the user input by appropriately specifying the property of the input elements.
  2. You might want to use the keyword argment `prevent_initial_call`.

0.2.2 2. Viewing Random Submissions

Write a function called `random_messages(n)`, which will return a collection of `n` random messages from the `message_db`, or fewer if necessary. This StackOverflow post might help. Don't forget to close the database connection within the function!

Next, write a new component to display the messages extracted from `random_messages()`. Once again, here is an example:

Finally, write a callback function `view()` to display random messages. This function should first call `random_messages()` to grab some random messages (I chose a cap of 5), and then display these messages using a loop. It should be triggered when the "update" button is pressed.

0.2.3 3. Customize Your App

Here's an example of the app so far:

Let's customize this app by changing styles! At least, you should

- Incorporate a non-default font.
- Use color in some way.

Feel free to add CSS or other stylesheets in order to give your app a personal feel. Extra credits may be given for a more sophiscated and appealing visual.

Your app should be a lot more colorful than the screencaps shown in this notebook!!

0.2.4 4. The Jupyter Notebook

For your notebook, write a tutorial describing how you constructed your webpage. You should include:

- Separate code blocks and explanations for each of the Python functions you used to build your app (there should be at least five of them).
- Display your app running within the notebook.

- Your report must include two screencaps:
  - In the first screencap, you should show an example of a user submitting a message. In the handle field, **please use either your name or your GitHub handle**.
  - In the second screencap, you should show an example of a user viewing submitted messages. Show at least two messages, one of which is the message you submitted in the previous screencap. **This message should show your name or GitHub handle**. Additionally, please include in your report a link to the GitHub repository containing the code for your app.

0.3 Specifications

0.3.1 Format

0. There is no autograder for this homework.

- For `code` section, please submit the `zip` file containing all the files in your GitHub repository.
  - This should at least include `hw6.ipynb`, `hw6.py` (the python file converted from this notebook), and the two screencaps.

  - If you used any other file (e.g., image or css style file), please also include them.
- For the `pdf` section, the URL to your GitHub repo for this homework must appear.
- **You may be asked to send in the URL to your GitHub repo before the final Sunday class and make the repo public during the class. This homework may involve peer review grading.**

0.3.2 Coding Problem

1. Each of the required functions is implemented in a logical way.
2. Each of the required functions appears to successfully achieve the required task.
3. Callback functions also include the appropriate additional functions. For example, the callback function `view()` should call `random_messages()`.
4. Some styling should be done; it should be different from what is shown in the class. You should change font and color to be used. Extra credits for a more sophiscasted and visually appealing approach.

0.3.3 Style and Documentation

5. Helpful comments are supplied throughout the code. **Docstrings are not required in this homework, and you don't need to show the testing of `get_message_db()`, `insert_message()`, and `random_messages()` outside the web app as well.**

0.3.4 Writing

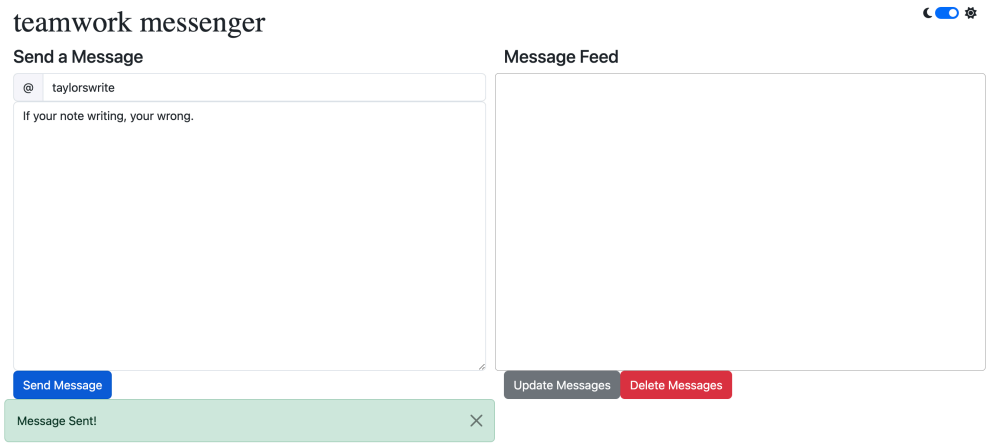
6. The overall report is written in engaging and unambiguous English prose. There is written explanations throughout the post, such that a student with the knowledge of the first five weeks of this course could learn to perform the demonstrated tasks by reading the post.
7. Each block of code has a clearly explained purpose.
8. The notebook is organized into clearly delimited sections using markdown headers (#), making it easier for the reader to navigate.
9. The notebook includes the two required screencaps demonstrating the submission and viewing pages of the app.
10. The notebook includes a discussion of all Python functions used to create the app. This should include, at minimum, `get_message_db()`, `insert_message()`, `random_messages()`, `submit()`, and `view()`.
11. The notebook launches the app inside it.
12. The notebook includes a link to the GitHub repository containing the code for the app.

0.4 Messaging App Summary

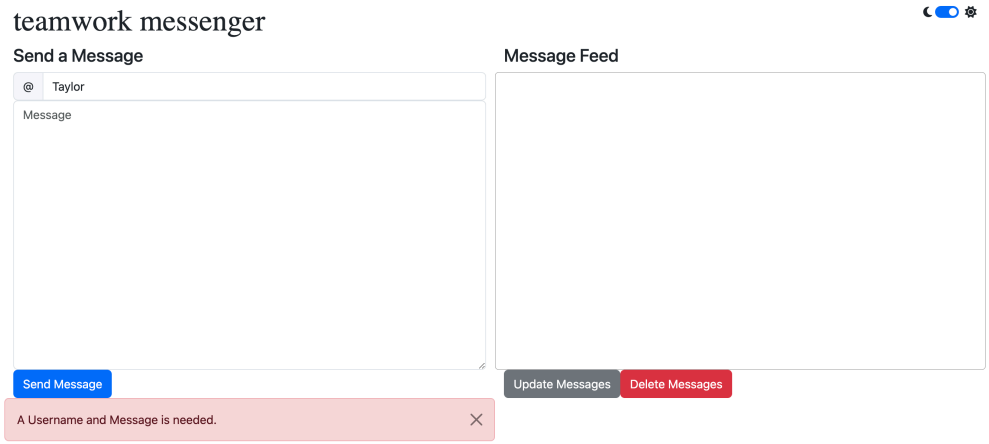
0.4.1 Introduction

The goal of this app is use Dash and sqlite to make a web app that allows users to submit their username/handle and a message to a database. In addition to allowing users to send messages to a database, the users can also access the database by pressing an update button that will randomly select eight messages to give the user a general idea of what messages are in the database. Finally, delete functionality was added to allow the user to quickly clear the database if need be.

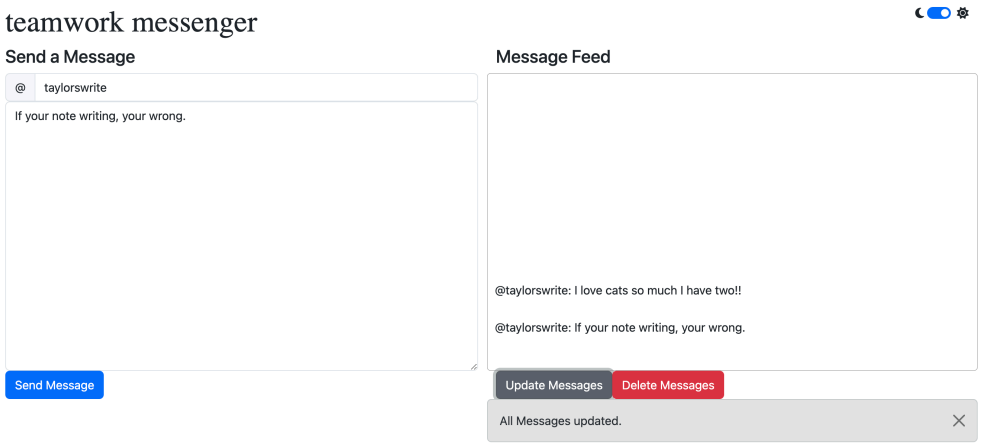
Successful Sent Message: Message that successfully sent



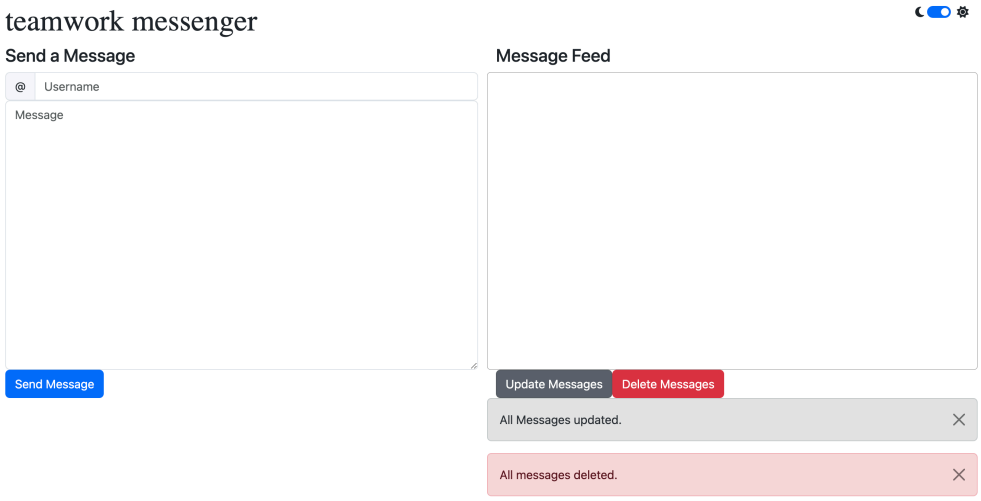
Empty Fields: Message that didn't send because it lacked all the fields



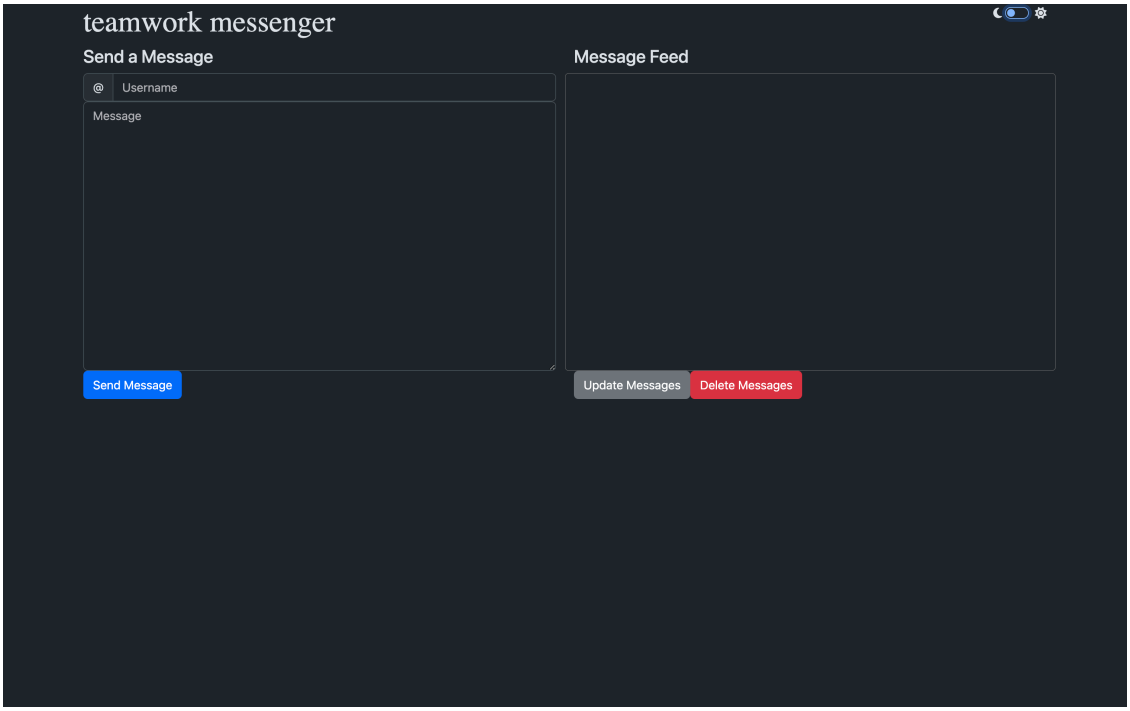
Update Random Messages: Updated messages list with 8 random messages



Delete All Messages: Deleted all messages in database



App Dark Mode Theme: Dark mode styling For Extra Credit!!!



0.4.2 Setup

To set up the web app a python file named app.py was created to house the dash app components such as the functions, UI elements, UI layout, and back-end server. The Dash app was to initialize with stylesheets for better and more sophisticated UI appearance.

0.4.3 Application Framework

2. Initialize Database and Database Functions
3. Initialize App
4. Define UI Elements
5. Define UI Layout
6. Server Callback Functions
7. Run Application

0.4.4 Initialize Database and Database Functions

`get_message_db` For organizational purposes, the database functions were placed at the top of the python file. The function, `get_message_db`, is where the app back-end starts. It checks the global environment if a database already exists and connects to that database. If a database does not exist, then the app will create a new database with the correct column names. This is an empty database.

```
# At startup init message_db
message_db = None

# Create a messaging database if it doesn't exist
def get_message_db():
    global message_db # retrieve database from global
```

```
    if message_db: # if contains values
        # connect to database
        message_db = sqlite3.connect(
            "messages_db.sqlite",
            # Different User Threads can access the same database
            check_same_thread=False
        )
        return message_db # return the the
    else:
        # connect to database
        message_db = sqlite3.connect(
            "messages_db.sqlite", # open database file
            # Different User Thread can access the same database
            check_same_thread=False
        )
        cmd = ( # Create the database if it doesn't exist
            """
            CREATE TABLE
            IF NOT EXISTS message_db (handle TEXT, message TEXT)
            """
        )
        cursor = message_db.cursor() # open database for executions
        cursor.execute(cmd) # Execute command
        return message_db # save the database to variable
```

insert_message insert_message inserts a username and message into the database. It accomplishes this by calling get_message_db to open a connection to the database and uses SQL commands to insert value inputs into the SQL table. After inserting the values, the new additions to the table are committed to the database to append the values to the SQL table. After adding the values, the database connection is closed. closing a database is good practice to prevent resources from being used when not needed.

```
def insert_message(username, message): # triggered on submit
    conn = get_message_db() # open database within function
    cursor = conn.cursor() # open an execution
    cursor.execute( # insert username and message into database table
        "INSERT INTO message_db (handle, message) VALUES (?, ?)",
        (username, message)
    )
    conn.commit() # commit the transaction to database
    conn.close() # close the database
```

random_messages The database function, random_messages, used get_message_db to open the database connection and execute an SQL command to randomly order the database and supply and n number of messages. At the end of the function, the database connection was closed, and the values were formatted into a readable string and returned.

```
def random_messages(n):
    conn = get_message_db() # open database within function
    cursor = conn.cursor() # open an execution
    row = cursor.execute( # Randomly message from database(prompt)
        f"SELECT * FROM message_db ORDER BY RANDOM() LIMIT {n};"
    )
    rows = cursor.fetchall() # aggregate all
    messages = list()
    for row in rows: # iterate over rows
        messages.append(f"@{row[0]}: {row[1]}")# username column
    conn.close() # close the database
    return messages # return username and message
```

delete_messages The function `delete_messages` was used to delete all rows of the database. Accomplish this by using `get_message_db` to connect to the database and using an SQL command to delete all rows, but keep the table.

```
def delete_messages():
    conn = get_message_db() # open database within function
    cursor = conn.cursor() # open an execution
    row = cursor.execute( # Delete all rows in database
        f"DELETE FROM message_db"
    )
    conn.commit() # commit the transaction to database
    conn.close() # close the database
```

0.4.5 App Initialization

The initialization of the app follows basic Dash guidelines; however, it uses external sheets to significantly improve the user interface of the app. Specifically bootstrap was the main style sheet that was used to create UI elements, such as input text boxes, alerts, and submit buttons. Additionally icons from Font Awesome we're added to be used in the light and dark mode switch.

```
# App initialization
app = Dash(
    __name__,
    external_stylesheets=[
        dbc.themes.BOOTSTRAP, # Use bootstrap stylesheet
        dbc.icons.FONT_AWESOME # Use Font_Awesome icons (for light/dark switch)
    ]
)
```

0.4.6 Define UI Elements

For organizational purposes, as well as ease of reading, the UI elements were defined outside of the UI layout. This was done so that it would be easier to see and modify the UI layout as well as make changes to any UI element. There were two types of UI elements that were used. There was HTML elements and bootstrap style sheet elements. The HTML elements were used to create titles/headers for the app as well as its main components and the all messages box. The bootstrap

stylesheet elements were used to make the interactive and stylized user input elements. Some examples of both are showing below. in the next section, these elements will be added to the app inside of the UI layout. The most important parts of a UI input element is its ID, placeholder value, status, and appearance. The ID is important because it is used to be reference the UI element in function callbacks in the server, back-end of the app. The ID allows for the callback to get the input, output, or status of a UI element and use it within the function. For the HTML UI elements, the appearance was modified using CSS.

Header (HTML/CSS)

```
header = html.H1("teamwork messenger", style={
    "font-family": "Cosmic Sans"}
)
```

Text Box (HTML/CSS)

```
# Message Box
message_box = html.Div(
    id='all_messages',
    style={
        'height': '400px',
        'overflow-y': 'scroll',
        'scroll-behavior': 'smooth',
        'border': '0.2px solid grey',
        'border-radius': '5px',
        'padding': '20px 10px',
        'display': 'flex',
        'flex-direction': 'column-reverse',
         'background-clip': 'padding-box',

    }
)
```

Username/Handle and User Message (Bootstrap)

```
# Text Input Boxes (Username and User Message)
input_groups = html.Div([
        dbc.InputGroup([
            dbc.InputGroupText("@"),
            dbc.Input(id='username', placeholder="Username")],
        ),
        dbc.InputGroup([
            dbc.Textarea(
                id='message',
                style={'height': 362},
                placeholder="Message"
        )
    ])
])
```

Submit Button (Bootstrap)

```
submit_button = dbc.Button(
    "Send Message",
    id='submit_button',
    color="primary"
)
```

Sent Notification (Bootstrap)

```
sent_alert = dbc.Alert(
    "Message Sent!",
    id="sent",
    dismissable=True,
    is_open=False,
    duration=3000,
    color="success",
)
```

Light and Dark Mode (Bootstrap/FontAwesome)

```
color_mode_switch =  html.Span(
    [
        dbc.Label(className="fa fa-moon", html_for="switch"), # Dark icon
        dbc.Switch( # switch element that is True or False
            id="switch", # callback
            value=True,  # Start at True
            className="d-inline-block ms-1",  # element calss
            persistence=True), # Always apearing
        dbc.Label(className="fa fa-sun", html_for="switch") # Light icon
    ]
)
```

0.4.7 Define UI Layout

Defining the layout of the app sounds like a daunting task, but it is relatively easy compared to defining the UI elements. The UI layout is one of the most important parts of a dash app. For any UI element to appear in the final production, it must appear in the UI layout section. The layout section is powerful, for example by using nested rows and columns, UI elements were easily aligned and grouped together and if further stylization was needed, CSS could be used to alter the appearance. This section could look like a jumbled mess, but additional steps were taken to make sure that the layout section was readable by defining all UI elements outside of the UI layout.

UI Layout (Bootstrap)

```
app.layout = dbc.Container([
    # Header Row
    dbc.Row([ # 1 of 3 main rows
        dbc.Col([header], width=6), # column within row and width of column
        dbc.Col([color_mode_switch], style={'text-align': 'right'}, width=6)
```

```
    ]),

    # Message Row
    dbc.Row([ # 2 of 3 main rows
        # User Input
        dbc.Col([ # 1 of 2 columns within row
            dbc.Row([user_header]), # header
            dbc.Row([input_groups]) # username and user message
        ]),
        # Message Database
        dbc.Col([ # 2 of 2 columns within row
            dbc.Row([feed_header]), # header
            dbc.Row([message_box]), # all messages
        ])
    ]),

    # Button Row
    dbc.Row([ # 3 of 3 main rows
        dbc.Col([ # 1 of 2 columns
            dbc.Row([ # row within column
                dbc.Col([submit_button]), # Added button in column for formating
            ]),
            dbc.Row([sent_alert]), # show sent notification below button
            dbc.Row([bad_message_alert]) # show error notification
        ]),
        dbc.Col([ # 2 of 2 columns
            dbc.Row([ # row within column
                dbc.Col([update_button,delete_button]), # in col for formating
            ]),
            dbc.Row([update_alert]),
            dbc.Row([deleted_alert]) # show updated notification below button
        ])
    ])
])
```

0.4.8 Server Callback Functions

Server callback functions are needed in order to make the app perform actions. Without server callback functions, when a button is pushed or text is entered into an input field, nothing will happen. additionally, without server callbacks, the functions we defined above in the database functions section can't be called by clicking a submit button, can't take inputs from fields in the app, and can't output to the app. callbacks are essential for the app to be interactive. There are three main components in a callback, an input, output, and state. Callbacks use the UI element ID to link the inputs and outputs to a function to the UI element of the app. An input acts like a trigger where it can be used to function. An output takes the return value of a function and send it to the UI element based on its ID. The state is used to get the value of a UI element, but not use it as a trigger to call the function. there are too many server callback functions to talk about in this summary therefore, the two main callback functions will be covered, the `submit` and `view`

callback functions.

`submit` Callback Function The `submit` function is special because the callback it has two outputs, one input, and two states. The two outputs are used to notify the user if a message was sent or if the user has not inputted enough information. The input and trigger is the submit button where when the submit button is clicked, the function is triggered. The two states are the username/handle and the user message inputs. The username/handle and the user message are inputted into the `insert_message` function which was defined above. This inserts, the values into the SQL table. If a username/handle or message is empty, a lack of username/handle or message notification will be sent to the user, otherwise a successful sent message notification will be sent. There are two outputs so two boolean values are returned to trigger the notifications. Additionally, to prevent an initial call option is set to true so that the callback function is not triggered upon app start up.

```
# Callback function to send message
@app.callback( # funciton decorator wrapper for dash app
    Output('sent', 'is_open'), # Output message sent notification
    Output('error', 'is_open'), # Output message error notification
    Input('submit_button', 'n_clicks'), # Trigger callback on submit button
    State('username', 'value'), # username contents
    State('message', 'value'), # message contents
    prevent_initial_call=True # Don't perform callback on app startup
)
def submit(submit_button, username, message): # triggered on submit
    if username and message: # If username and message aren't empty
        insert_message(username, message)
        return True, False # show sent notification
    else: # if username or message are empty
        return False, True # show error notification
```

`View` Callback Function The `view` callback function has two outputs and one input. One output is all of the random messages selected and the second output is the updated notification sent to the user. The input and trigger is when the update button is pressed when the update button is pressed, the callback function `view` is called. When it's called, a list of messages from `random_messages` is used to append all HTML text elements. This is returned and sent to the HTML text box with the ID all_messages. A true statement is used to send the user a notification that the messages have been updated.

```
# Function callback to update database
@app.callback( # function decorator
    Output('all_messages', 'children'), # Output html text
    Output('updated', 'is_open'), # Updated notification
    Input('update_button', 'n_clicks'), # Trigger is update button
    prevent_initial_call=True # Don't Run on startup
)
def view(n_clicks): # Triggered by update button
    messages = random_messages(8) # show the past 8 messages
    all_rand_messages = list()
```

```
    for message in messages: # iterate over message dict
        all_rand_messages.append( # append html
            html.Div([
                html.P([message]),
            ],style={'margin-bottom': '10px'})
        )
    return all_rand_messages, True
```

0.4.9 Run Application

Finally, the app is ran by running the python code below. The name space needs to be main to run the app. This is to prevent the app from running upon importing into a notebook or python file. The port can also be specified if the port is already occupied.

```
# Run the app
if __name__ == '__main__':
    app.run_server(debug=False, port=8050)
```

Run the code below to start the app!

[1]:
```
%run -i 'app.py'
```

```
<IPython.lib.display.IFrame at 0x105d9cb20>
```

0.5 Sources

0.5.1 Bootstrap

- Boostrap Alerts
- Bootstrap Button
- Bootstrap Input Groups
- Bootstrap Layout
- Bootstrap Text Box Input
- Bootstrap Text Box Sidebar
- Bootstrap Themes
- Bootstrap Themes
- Bootstrap Toast
- CSS Color Title

0.5.2 CSS

- CSS Text Align
- CSS Text Align
- CSS Text Box Border
- CSS Text Box Border
- CSS Text Box Formatting
- CSS Text Box Padding
- CSS Text Box Scroll
- CSS Text Outline
- CSS Text Padding

0.5.3 Dash

- Dash Clientside Callbacks
- Dash HTML and Children Outputs
- Dash Importing Images
- Dash Multiple Callback Outputs
- Dash Themes

0.5.4 SQL

- SQL Creat Table with Column Names
- SQL Create Table if it Doesn't Exist
- SQL Delete Table Rows
- SQL Delete Table Rows
- SQL Insert Values
- SQL Inserting Python Value into Table
- SQL Storing Python Value into Table