5-2 Milestone Four: Enhancement Three: Databases

Dustin Morris

CS-499 Computer Science Capstone 23EW2

Prof Brooke

11/25/2023

Introduction:

The artifact for Enhancement Three: Databases is a Python application designed to create a web interface using the Dash web app framework and integrating Dash Component modules for displaying animal data from animal shelters, including results that are filtered based on user criteria. This application was developed during my CS-340 Client/Server Development course in the 22EW4 term. These applications originated in my second artifact, where I developed an enhanced search algorithm. Choosing PyCharm as my development environment (IDE) and using the Python programming language, I designed and developed a useful enhancement to this application, I accomplished this by implementing an additional collection, "users" into my database to handle user accounts. My goal is to align my work with all-course outcomes related to databases.

Overview:

The dashboard is configured to work with the MongoDB on a localhost, granting permission to anyone within the same network. Figure 1 shows the imported module that enables the Dash framework to be accessible on machines in the same network, including mobile devices.

Figure 1

Import component from Application.py.

```
from waitress import serve
```

Defining "Waitress" in the application provides a function to the script, **`serve(app.server, host='0.0.0.0', port=8050)`,** instructing the application to listen on all interfaces. Running "**WSGI** " creates accessibility to the dashboard using your machine's actual

IP address. This requires you to run the following command in your IDE terminal `**waitress-serve --host=0.0.0.0 --port=8050 WSGI_Server:app**`.

To establish a connection to the database I assigned, `**client = MongoClient**`, and `**db = client[‘AAC’]**`. Also assigning `**collection = db[‘animals’]** and `**collection_users = db[‘users’]**` grants access to the collections in the database. The ‘animals’ collection stores the data provided to the dashboard DataFrame, and the ‘users’ collection stores usernames and passwords created during registration. Figure 2 visualizes how I establish a database connection.

Figure 2

Connecting to MongoDB in Application.py

```
# Connections to MongoDB
client = MongoClient('mongodb://myUserAdmins2:123456@localhost:27017/AAC?authSource=AAC')
db = client['AAC']
collection = db['animals']
collection_users = db['users']
```

The process to create additional collections lies in the assignment provided for `**collection =**`. This variable creates the additional collection, also noting that inserting data will create the collection automatically. To insert data in the new collection I designed two functioning button components. The components were added to the existing layout of the dashboard to open separate windows about which components clicked.

Figure 3

Usage of HTML elements for buttons in Application.py.

```
    html.Button( children: 'Open Login', id='open-login-button', style={'display': 'block'}),
    login_modal,
    html.Div(id='auth-status'),
    html.Br(),
    html.Hr(),

    # Map that uses the library leaflet to show where the animal is located, if data contains it
    html.Div(
        id='map-id',
        style={'width': '700px', 'height': '450px', 'margin': 'auto'},
    )
```

In Figure 3 the button is being implemented into the layout using `**html.Button**`, The implementation is provided by `**import html**`, a module from the Dash framework to create HTML elements. The visibility is defined by `**style={'display': 'block'}),**`. This design extends to other clickable components, each is given unique IDs to provide individual functionality.

Figure 4

Login modal from Application.py.

```
login_modal = dbc.Modal(
    children: [
        dbc.ModalHeader("Login"),
        dbc.ModalBody(
            [
                dcc.Input(id='username-input-modal', type='text', placeholder='Enter your username'),
                dcc.Input(id='password-input-modal', type='password', placeholder='Enter your password'),
                html.Div(id='login-message-modal', style={'color': 'red'})
            ]
        ),
        dbc.ModalFooter(
            dbc.Button( children: 'Login', id='login-button-modal', n_clicks=0, color='primary'),
        ),
    ],
    id='login-modal',
    is_open=False,
)
```

To provide functionality for the buttons I imported Dash Bootstrap Components (dbc), and `**dbc.Modal**` which execute user input and display messages. Figure 4 allows you to visualize how `**dbc.Modal**` is used to control inputs and messages within each `**dcc.Input**` element. Importantly ensure `**id=login-modal**` strictly matches the corresponding button, not to get confused with `**id=register-modal**`, since it corresponds with the '**register-modal**'.

Integrating the `**dcc**` and `**html**` components of Dash provides the useability of `**pathname**` in collaboration with `**dcc.Location**` to dynamically switch between layouts. This was implemented because the application required multi-page applications to open within a single-page application. Using callbacks, for example in the snippet below you can see how the output values of `**pathname**`, and `**url**` are checked, this is how the application knows which layout to return.

Figure 5

Callback referencing the login button component in Application.py.

```
# Callback for the login button component
@app.callback(
    *_args: [
        Output( component_id: 'auth-status', component_property: 'children'),
        Output( component_id: 'login-modal', component_property: 'is_open'),
        Output( component_id: 'login-message-modal', component_property: 'children'),
        Output( component_id: 'url', component_property: 'pathname'),
    ],
    [
```

Recognizing that creating just the logic is not sufficient for the button components to work, instructions are required to interpret `**login**` and `**register**`. To overcome this, I created a callback, a function that is passed to another function as an argument to handle, this analyzes

whether either button is clicked utilizing `**n_clicks**`, and corresponding `**if**` statements. Figure 6 shows the callback `**app.callback**`**,** using control over `**Output**`, `**Input**`, and `**State**`, making sure it responds to button interactions correctly. With the combination of `**n_clicks**`, and conditional statements inside the callback block, the application now understands what it's being instructed to achieve.

Figure 6

Callback referencing the output states for components in Application.py.

```
@app.callback(
    *_args: [
        Output( component_id: 'auth-status', component_property: 'children'),
        Output( component_id: 'login-modal', component_property: 'is_open'),
        Output( component_id: 'login-message-modal', component_property: 'children'),
        Output( component_id: 'url', component_property: 'pathname'),
    ],
    [
        Input( component_id: 'login-button-modal', component_property: 'n_clicks'),
        Input( component_id: 'open-login-button', component_property: 'n_clicks'),
    ],
    [
        State( component_id: 'username-input-modal', component_property: 'value'),
        State( component_id: 'password-input-modal', component_property: 'value'),
        State( component_id: 'url', component_property: 'pathname'),
        State( component_id: 'login-modal', component_property: 'is_open'),
    ],
```

My authentication system callback utilizes conditional statements to handle what happens when a user tries to log in. Inputs in their respective fields must match against what is stored in the database. If conditions are satisfied, you're redirected to the dashboard's main page with authentication. Invalid inputs that don't match the stored data will print a string announcing an invalid username or password message, not redirecting you. In the next code snippet, you can see

the logic for checking authentication using `**if login_button_clicks:**` and `**if authenticated**:` it prints the success message. If wrong credentials are used, `**else**` and `**return**` functions print a message for the input being invalid, in Figure 7 you are shown the conditions.

Figure 7

Logic implementation for authenticating when activating button clicks in Application.py.

```
# Checking if the login button is clicked on
if login_button_clicks:
    # Authenticate the user
    authenticated = authenticate_user(username, password)

    if authenticated:
        # If authenticated the window will close and display a welcome message with the users name
        return (
            f"Authentication Status: Welcome, {username}!",
            False,
            None,
            url_pathname,
        )
    else:
        # What shows if the wrong credentials are entered.
        return (
            "Authentication Status: Invalid username or password.",
            True,
            "Invalid username or password.",
            url_pathname,
        )
```

The logic in the register callback is designed to work similarly to the login logic. When clicking the register button component, I included a function to check the `**users**` collection in the database for existing usernames. The functions in Figure 8 show how `**if**` and `**elif**` are used to check for existing users. If the existing username is found `return` prints a string message confirming it already exists while making sure the window remains open utilizing `**is_register_modal_open**`.

Figure 8

Referencing if and elif for existing users in Application.py.

```
# Checks if you try to open register window
if open_clicks and not register_button_clicks:
    print("Open Register button clicked.")
    return not is_register_modal_open, None, None

# Checks if you click the register button inside open register window
elif register_button_clicks:
    print("Register button clicked.")
    # How I check if the username entered is already taken
    existing_user = collection_users.find_one({'username': register_username})
    if existing_user:
        print("Username already exists.")
        return is_register_modal_open, "Username already exists. Choose a different one.", None
```

The registration implementation I added operates using a callback function providing conditional statements and functions to handle what happens when `**Register**` is clicked, it's like the `**Login**` operation. When valid inputs are recognized, it stores the new data inside the 'users' collection, which are now defined as valid credentials. The purpose of authenticating is to provide access to a functional `**Add Animal**` button component that is initially disabled and non-functional using a `**disabled**` component_property inside the `**@app.callback**` `**Output**` function. This permits registered users to log in and insert new pet data in the '`**Add New Animal Data**` input fields. The data you input will be immediately stored inside the database in the animal collection when the application receives the input by clicking `**Add Animal**`. The inserted pet information is now visible when applying filters.

To heighten security and privacy measures I decided against the use of hard-coded credentials for dashboard access. I made requirements mandating all users be registered to insert animal data into the database. The system is set up to store usernames and passwords in the `**users**` collection in the database to protect the user's information, further increasing the security

protocol to protect passwords I implemented hashing passwords. Before the password is inserted and stored in the database it is hashed. To hash passwords, I use `**from passlib. hash import pbkdf2_sha256**` in the headers import section to execute a key derivation function from the `**Passlib**` library, Figure 9 shows how it is implemented with the `**Modal**` callbacks. Touching on additional security features, if a user is logged into the dashboard and the page is closed or refreshed it requires users to authenticate themselves again.

Figure 9

Variable, `hashed_password` storing hashed password using PBKDF2 algorithm in Application.py.

```
# Hashes the password for security
hashed_password = pbkdf2_sha256.hash(register_password)

# adds the new user to the users collection in the database
result = collection_users.insert_one({'username': register_username, 'password': hashed_password})
```

The definitions for user authentication and password hashes are stored in the `**Authentication.py**` file so any authentication issues can be addressed promptly and not mixed in with other functions. Security is very important, and the code can be navigated through quickly. Sections of code can be made into additional files to be spread out by moving the functions to the new file and adding import to the header to include the newly created file. In these next code snippets, you can see how the files `**Application.py**` and `**Authentication.py**` files can interact using `**from**` and `**import**` statements.

Figure 10

I am importing `authenticate_user`, `logout_user`, `get_current_user` functions from Authentication.py.

```
#Imports for libraries
from Authentication import authenticate_user, logout_user, get_current_user
```

For example, in `**Application.py**` I use `**authenticate_user(username, password)**`, this function is not defined in `**Application.py**`, it's defined in `**Authentication.py**`. In the import header, `**from Authentication import authentic_user**` allows `**Application.py**` usage of the functions defined in `**Authenticaton.py**` by integration.

Conclusion:

I have showcased my knowledge and skills, meeting the outcomes of the course by demonstrating an ability to use well-founded and innovative techniques, skills, and tools in computing practices to implement computer solutions that deliver value and accomplish industry-specific goals (software engineering/design/database).

I meet this outcome by using external tools and libraries, in my case `**passlib**`, for password hashing which exhibits my knowledge and understanding of how to utilize third-party tools. I highlight my strong abilities utilizing MongoDB by creating a collection for storing new user data. I enhanced functionality by adding button components for '**Login**' and '**Register**,' further expanding my dashboard. The logic for my new components provides a deeper understanding of application behavior and user interactions. I demonstrate good coding practices by using separate authentication-related functions in `**auth.py**`. I showcase problem-solving skills by preventing duplicate usernames during registration.

I developed a security mindset that anticipates adversarial exploits in software architecture and designs to expose potential vulnerabilities, mitigate design flaws, and ensure privacy and enhanced security of data and resources. I achieved this outcome by implementing password hashing that creates a secure password storage function before adding it to the

database, protecting user credentials. My authentication system includes checks for duplicate usernames. The passwords are validated against the stored hashed passwords in the database, stopping unauthorized access. I included a limited access feature only allowing registered users to have access to the “Add Animal” button component to prevent misuse that could overload the animal data with unrelated information. I showcase my ability to develop a security mindset in both aspects of design and implementation. I created measures to stop unauthorized access and protect user data to ensure the privacy and security of my application.

I also meet the outcome for employing strategies for building collaborative environments that enable diverse audiences to support organizational decision making in the field of computer science. The security measures I used provide privacy and security which would help users feel confident and protected in sharing data within my application which is needed for organizational decision-making processes. I implemented the option to run the application from different machines on the same network showing the option to expand my application to be used by a greater range of users.