5-2 Milestone: Enhancement 3: Databases

Carl Allen

CS-499 Computer Science Capstone

**Introduction:**

The artifact selected for Enhancement Three: Databases is a Python-based web application built with the Dash framework, which uses Dash Components to present data from animal shelters through a dynamic, user-filtered interface. Originally developed during my CS-340 Client/Server Development course in the Fall 2024 C-5 term, this project builds upon work from my second artifact, where I focused on improving search functionality. For this enhancement, I utilized PyCharm as my integrated development environment and expanded the application by incorporating a new database collection called “users” to manage user accounts. This update was part of my effort to meet the course outcomes related to database design and integration.

**Overview:**

The dashboard is set up to connect with a MongoDB instance running on localhost, allowing access to users on the same local network. As shown in Figure 1, a specific module is imported to make the Dash application available across networked devices, including smartphones and other computers within the same network.

**Figure 1**

Import component from Application.py.



Integrating “Waitress” into the application enables the use of the function serve(app.server, host=’0.0.0.0’, port=8050), which directs the app to operate on all available network interfaces (instructing the application to listen on all interfaces). By running the “WSGI” server, the dashboard becomes accessible via the device’s actual IP address (creates accessibility to the dashboard using your machine's actual IP address). To achieve this, you need to enter the following command in your IDE terminal: waitress-serve --host=0.0.0.0 --port=8050 WSGI\_Server:app.

To connect to the database, I created a client instance using client = MongoClient and accessed the database with db = client[‘AAC’]. I then defined collection = db[‘animals’] and collection\_users = db[‘users’] (grants access to the collections in the database). The ‘animals’ collection supplies data to the dashboard’s DataFrame, while the ‘users’ collection contains login credentials generated during user registration. Figure 2 provides a visual representation of the database connection setup.

**Figure 2**

Connecting to MongoDB in Application.py

A screen shot of a computer

Description automatically generated

The creation of additional collections begins with the assignment made to collection =, which defines the new collection. It's important to note that in MongoDB, a collection is automatically created upon the first data insertion. To support this, I developed two functional button components that allow users to insert data into the new collection. These buttons were integrated into the existing dashboard layout and are designed to open separate windows depending on which component is clicked, enhancing user interaction and data management.**Figure 3**

Usage of HTML elements for buttons in Application.py.

A screen shot of a computer

Description automatically generated

Figure 3 illustrates how the button is integrated into the dashboard layout using html button. This element is made available through the import html module from the Dash framework, which enables the creation of HTML components within the app. The button’s visibility is managed by the style attribute style={‘display’: ‘block’}, ensuring it is displayed as a block-level element. This approach is also applied to other interactive components, each assigned a unique ID to enable distinct functionality and event handling.

**Figure 4**

Login modal from Application.py.

A computer screen with text

Description automatically generated

To enable button functionality, I utilized Dash Bootstrap Components (dbc), specifically importing dbc.Modal, which handles user input and displays corresponding messages. Figure 4 illustrates how dbc.Modal interacts with each dcc.Input element to manage input and feedback. It's crucial to match modal IDs correctly—for example, id=login-modal should be linked to the login button and not confused with id=register-modal, which is reserved for the registration component.

By combining Dash’s dcc and html modules, I was able to leverage the pathname property in conjunction with dcc.Location to dynamically render different layouts. This integration supports multi-page functionality within a single-page application. Through the use of callbacks, as shown in the snippet below, the app determines which layout to display based on the values of pathname and url. This mechanism is essential for managing navigation within the app’s structure.

**Figure 5**

Callback referencing the login button component in Application.py.

A screenshot of a computer program

Description automatically generated

Writing the underlying logic alone isn’t enough to ensure the button components behave as intended—there must also be explicit instructions to handle actions like logging in or registering. To accomplish this, I implemented a callback, which is a function passed into another function to be executed based on user interaction. This callback monitors the n\_clicksproperty to detect when a button is pressed and uses if statements to determine which specific action to take. As shown in Figure 6, the app.callback structure is used to define this behavior, utilizing Output, Input, and State to manage dynamic interactions between components. With this setup, the application can interpret button presses and trigger the appropriate response.

**Figure 6**

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

A screenshot of a computer program

Description automatically generated

The authentication callback in my system uses conditional logic to determine the outcome when a user attempts to log in. The entered credentials are compared against the values stored in the database. If the credentials are valid, the user is redirected to the dashboard's main page with proper authentication. However, if the inputs don’t match the stored data, a message indicating an invalid username or password is displayed, and no redirection occurs. In the following code snippet, the logic for authentication is shown using if login\_button\_clicks: and if authenticated:, which prints a success message upon valid login. For incorrect credentials, the logic falls into the else block, using a return statement to display an error message. Figure 7 highlights these authentication conditions in detail.

**Figure 7**

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

A screenshot of a computer screen

Description automatically generated

The logic behind the register callback closely mirrors the structure of the login process. When the register button is clicked, the system runs a function that checks the users collection in the database to see if the entered username already exists. As shown in Figure 8, if and elif statements are used to evaluate whether the username is already taken. If a match is found, a return statement triggers a message informing the user that the username already exists, while also keeping the registration window open by maintaining the state of is\_register\_modal\_open.

**Figure 8**

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

A screen shot of a computer program

Description automatically generated

The registration feature I implemented functions through a callback that uses conditional logic to handle what occurs when the Register button is clicked, following a process similar to the Login logic. When valid input is detected, the new user data is saved to the users collection, effectively registering the credentials for future authentication. This authentication is crucial for enabling access to the Add Animal button, which is initially inactive through the use of the disabled property in the @app.callback's Output function. Once logged in, registered users gain access to input fields labeled ‘Add New Animal Data’, allowing them to submit pet information. When the Add Animal button is clicked, the input data is immediately written to the animals collection in the database and becomes visible through the filtering options.

To enhance privacy and security, I avoided using hard-coded credentials for dashboard access. Instead, I required all users to register before being allowed to add animal data. Usernames and passwords are stored securely in the users collection, and to further safeguard sensitive data, I implemented password hashing. Before passwords are stored, they are encrypted using a key derivation function from the Passlib library, imported with from passlib.hash import pbkdf2\_sha256. Figure 9 demonstrates how this is integrated within the Modal callbacks. Additionally, to reinforce session security, the system requires users to re-authenticate if the dashboard page is closed or refreshed.

**Figure 9**

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

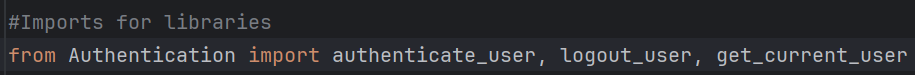
A screen shot of a computer

Description automatically generated

To keep authentication logic organized and secure, all user authentication and password hashing definitions are placed in a separate file called Authentication.py. This separation ensures that any issues related to authentication can be addressed quickly without interfering with unrelated code. Prioritizing security and maintainability, I’ve structured the code for easy navigation. Functions can be modularized further by moving them into additional files and importing them as needed. In the upcoming code snippets, you’ll see how the main application file (Application.py) interacts with Authentication.py through the use of from and import statements, enabling clean and efficient integration between modules.

**Figure 10**

I am importing `authenticate\_user`, `logout\_user`, `get\_current\_user` functions from Authentication.py.



For instance, in Application.py, I call the function authenticate\_user(username, password), but this function isn’t defined within that file. Instead, it resides in Authentication.py. By including the import statement from Authentication import authenticate\_user at the top of Application.py, I enable seamless access to the function, allowing the main application to utilize authentication logic defined in a separate module. This modular approach keeps the codebase organized and makes maintenance and debugging more efficient.

**Conclusion:**

have demonstrated my knowledge and skills by meeting the course outcomes, particularly through the effective use of established and innovative techniques, tools, and practices to build computing solutions that deliver value and align with industry needs (software engineering/design/database). I achieved this by integrating third-party tools such as passlib for password hashing, showcasing my ability to leverage external libraries to enhance functionality and security. My proficiency with MongoDB is reflected in the creation of a dedicated users collection for storing new user data. I expanded the dashboard’s functionality by implementing 'Login' and 'Register' button components, which required developing logic that deepened my understanding of application behavior and user interaction. I also followed sound coding practices by placing authentication-related functions in a separate auth.py file and addressed potential user experience issues by preventing duplicate usernames during registration.

I’ve developed a strong security mindset, anticipating threats and designing the application to mitigate vulnerabilities while protecting user privacy. This is evident in my implementation of password hashing, ensuring that credentials are securely stored before being written to the database. My authentication system includes verification against stored hashes and duplicate username checks to prevent unauthorized access. Additionally, I restricted the “Add Animal” button to authenticated users only, helping protect the animal data from irrelevant or malicious input. These design choices reflect my ability to prioritize data security in both architecture and functionality.

Furthermore, I’ve met the outcome related to fostering collaborative environments that support diverse users and organizational decision-making in computer science. The security measures I implemented promote user trust, which is essential for encouraging data sharing in any application-driven setting. By enabling the application to run across multiple machines on the same network, I also demonstrated scalability, allowing for broader adoption and collaboration within team or organizational environments.