Criteria C: Development

Technique	Explanation of Use in 'Baybai' Mobile Application
Object Oriented Programming (OOP)	Implemented OOP classes for app screens and functions, encapsulating specific properties and methods for each application segment.
Flask App Routing	Handled server-side URLs in Flask for user authentication, data retrieval, post submissions, etc.
Tokens	Generated & validated JWT tokens for secure, authenticated communication between the client app and server.
HTTP methods	Utilized HTTP GET for data retrieval (e.g., posts) and POST for sending user data like login details and new posts to the server.
Functions	Defined functions to perform specific tasks.
Kivy Widget and Widget Adding	Kivy Widget management entails positioning and rendering child widgets based on parent properties, enhancing the user interface for created posts.
if and else statements	Utilized for control flow in the app, deciding when to show or hide elements, validate user input, or handle errors.
Text Formatting	Applied string manipulation and formatting to display data like error messages adequately.
Arrays, Lists, and Dictionary	Managed data for features including flashcards, user credentials, etc.
for loops	Processed actions like generating flashcards and listing posts by iterating over data structures.
Password Encryption	Hashed user passwords before database storage for security purposes.
Database interaction	Performed CRUD operations on the SQLite database to manage user data, flashcards, and posts within the app.

Error Handling and Popups	Popups are used for error handling to provide immediate and context-sensitive feedback to users in a user-friendly manner.
Dynamic Content Loading	Dynamically loaded content from the database onto a template screen for flashcard display.
Kivy.network: UrlRequest	It was used for making asynchronous URL requests to the server without blocking the main application thread.
PyJWT	Utilized PyJWT to encode/decode JSON web tokens, verifying token expiration and handling invalid token errors.
datetime	Use to add timestamps to posts
pyperclip	Python module that provides a simple interface to the clipboard on most operating systems

Table 4. Coding techniques used in the project development process

Explanation of Techniques and Evidence

Modal Dialogs

In the app, actions like new post creation or logging out involve user confirmation. To avoid clutter, a modal dialog approach is implemented, overlaying the main content and demanding user interaction for crucial actions. This enhances user experience by emphasizing important functionalities without navigating away from the main screen.

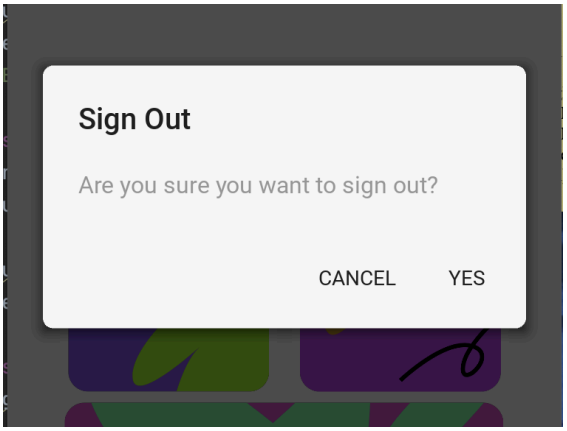
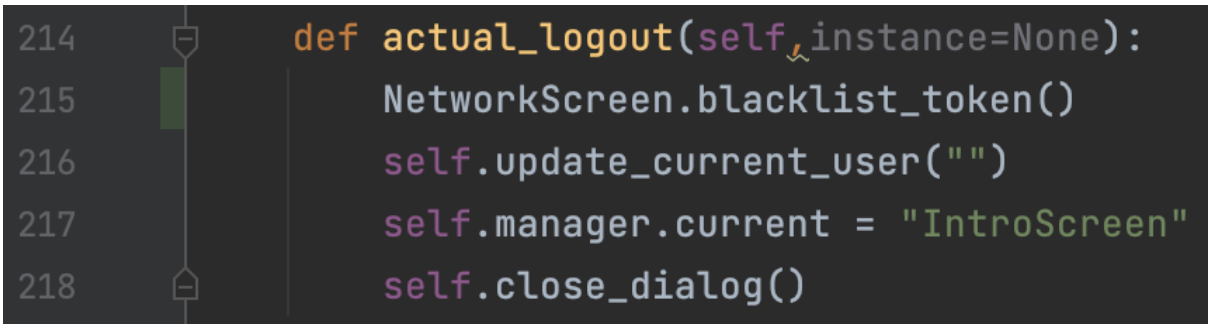


Figure 10. Display of the sign-out process, where a modal dialog prompts the user to confirm their intention to log out.

```
192     def logout(self):
193         self.dialog = None
194         if not self.dialog:
195             self.dialog = MDDialog(
196                 title='Sign Out',
197                 text='Are you sure you want to sign out?',
198                 buttons=[
199                     MDFlatButton(
200                         text='CANCEL',
201                         on_release=self.close_dialog
202                     ),
203                     MDFlatButton(
204                         text='YES',
205                         on_release=self.actual_logout
206                     ),
207                 ],
208             )
209         self.dialog.open()
```

Figure 11. Method that triggers the modal to popup

For example, Figure 11 illustrates the use of OOP to structure a logout confirmation within a modal dialog by encapsulating the UI logic. Line 193 sets the instance variable `dialog` to `None` to ensure it starts with no value. Lines 196-205 detail the dialog's setup like buttons and texts. Line 199-201 links the 'CANCEL' button on the modal dialog to the method `close_dialog` and the 'YES' button to method `actual_logout`. Finally, line 209 invokes the dialog's `open` method to display it to the user.



```
214     def actual_logout(self, instance=None):
215         NetworkScreen.blacklist_token()
216         self.update_current_user("")
217         self.manager.current = "IntroScreen"
218         self.close_dialog()
```

Figure 12. Methods that trigger the modal dialog to close or to perform the action

Figure 12 shows the method `actual_logout` handling logout tasks like clearing `current_user`, blacklisting the existing token in `NetworkScreen` thereby rendering it invalid & unusable, redirecting to IntroScreen, and closing the dialog.

JWT Tokens

After developing the login system, I faced the challenge of maintaining reliable user sessions, which is vital to fulfill Success Criteria #1. Initially considering cookies, I found them easier targets for unauthorized access and manipulation as they're stored client-side. This led to adopting JSON Web Tokens (JWT) tokens, which offer encrypted user data with enhanced security through token expiration and hashing with a unique secret key.

Token Generation

Successful login triggers the `create_token` method in the `NetworkScreen` to generate a JWT.

Figure 13. `generate_token` method

Upon successful login, the `NetworkScreen` class's `generate_token` method creates a JWT which is then used by the server to verify the user's identity, granting access to resources or allowing specific actions like fetching posts. A variable token is created and encoded (Line 679). `jwt.encode(..)` is a function call from the library PyJWT that codes a JWT; the JWT's payload consists of username and expiration, and is 'signed' with a secret key for additional security. `algorithm=...` in this same line specifies the algorithm used to sign the token.

Token Expiration

The `datetime` module's `utcnow()` function returns the current date and time without any timezone information (Line 678); it is used to set the token's expiration 1 hour from the method's execution time. The JWT library will reject the token with an `ExpiredSignatureError` if the current UTC time exceeds the `exp` claim timestamp, treating the token as expired. Lines 44-45 in Figure 15 show that posting with an expired token triggers an `Error 401`, an HTTP status code that stands for "Unauthorized", indicating authentication failure from expired credentials.

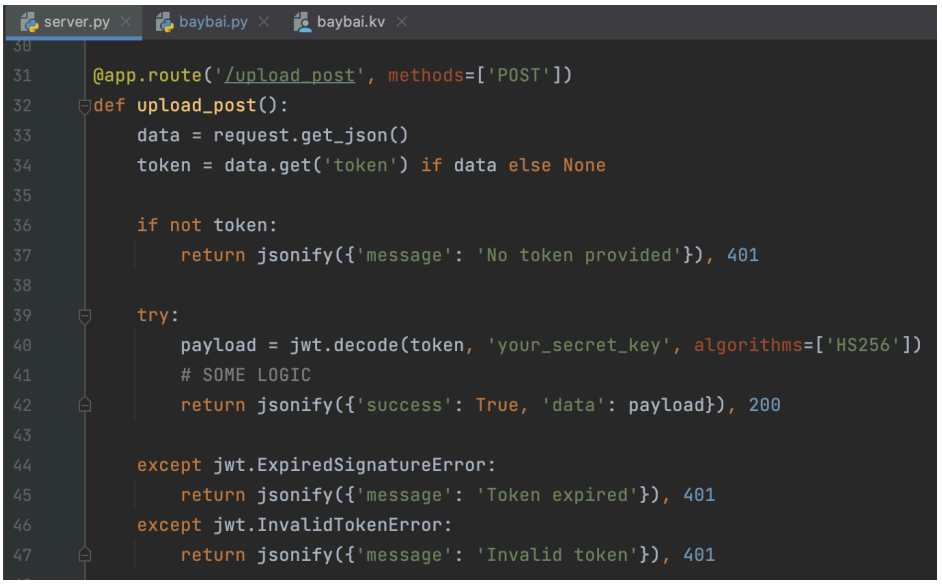


Figure 14. Flask server-side python file `server.py` containing the `upload_post` function

Dynamic Content Loading

Implementing a template resolved the challenge of displaying varied data in a uniform layout for flashcards, achieving Success Criteria #2.

```
850                 FloatLayout:
851                     MDCard:
852                         border_radius:20
853                         id: card_input
854                         radius:[14]
855                         md_bg_color: '#9851FF'
856                         size_hint: 0.7, 0.6
857                         pos_hint: {'center_x':.5,'center_y':0.7}
858                         on_release: root.flip_card()
859 
860                         MDLabel:
861                             text:root.flashcard_contents[0]
862                             halign: 'center'
863                             id: flashcard_content
864                             font_name: 'Noto'
865                             color: 'white'
866                             font_size: '250px'
867                             bold: True
```

Figure 15. A code snippet of the FlashcardsScreen interface.

I designed the Flashcards Screen interface as a template, pulling content from the `Flashcards` class's 'flashcard_contents' list (Line 860).

```
268     def __init__(self, **kwargs, ):
269         super().__init__(**kwargs)
270         self.flashcard_contents = ['']
271         self.saved_cards = []
272 
273     def on_enter(self, *args):
274         self.update_flashcard_content()
```

Figure 16. `FlashcardsScreen` Class containing the `flashcard_contents` attribute in which the kivymd template screen takes its contents from

The `on_enter` method activates the moment the user is in the `Flashcards` class; it calls the `update_flashcard_content` method, also defined within the same class.

```
def update_flashcard_content(self):
    # Update the flashcard content
    if self.is_tagalog:
        self.ids.flashcard_content.text = self.tagalog[self.current_card_index]
    else:
        self.ids.flashcard_content.text = self.baybayin[self.current_card_index]

    # Update the star icon based on the starred state of the current card
    if self.starred_states[self.current_card_index]:
        self.ids.star_button.icon = "star"
    else:
        self.ids.star_button.icon = "star-outline"
```

Figure 17. `update_flashcard_content` method in the `FlashcardsScreen` class

The `update_flashcard_content` method dynamically updates flashcard content in the app. It checks the `is_tagalog` attribute to determine the flashcard side being viewed and updates the `flashcard_content` widget's text. This allows users to switch between Tagalog and Baybayin sides and view different flashcards without changing screens, improving user experience.

Database Interaction

Database interaction is crucial for Success Criteria 1,2,5, and 6. To handle frequent database interactions, I created a `database_handler` class with multiple methods, reducing repetitive code and streamlining the implementation of frequently used functions in the program.

```
36  path = 'baybai.db'
37  class database_handler:
38      def __init__(self,namedb:str):
39          self.connection = sqlite3.Connection(namedb)
40          self.cursor = self.connection.cursor()
41
42      def run_query(self, query: str):
43          self.cursor.execute(query)
44          self.connection.commit()
45      def close(self):
46          self.connection.close()
47
48      def search(self,query:str):
49          result = self.cursor.execute(query).fetchall()
50          return result
```

Figure 18. `database_handler` class

Figure 18 shows a `database_handler` class that encapsulates database operations such as initializing connections, executing queries, and closing connections for efficient and

organized database interactions. Encapsulating these functions in `database_handler` reduces redundancy and enhances efficiency and organization in database handling throughout the app development process.

```
97  def try_register(self):
98      global path
99      db = database_handler(namedb=path)
100     db.run_query("""CREATE TABLE if not exists users(
101                     id INTEGER primary key autoincrement,
102                     name TEXT not null,
103                     uname TEXT not null,
104                     password TEXT not null);""")
105     #TAKE ALL INPUTS, VALIDATE, AND APPEND TO DATABASE
106     name,uname,pass1,con_pass = self.ids.name.text,self.ids.uname.text,self.ids.password.text,self.ids.confirm_password.text
107
108     if self.validate_register(uname, pass1, con_pass):
109         db.run_query(f"INSERT INTO USERS (name,uname,password) VALUES ('{name}','{uname}','{encrypt_password(pass1)}')")
110         self.popup("[color=ACFF3C]Registration completed. Welcome![/color]")
111         db.close()
112
113         self.ids.name.text,self.ids.uname.text,self.ids.password.text,self.ids.confirm_password.text="","","",""
114         self.update_current_user(uname)
115         self.clear_inputs()
116         self.parent.current = "HomeScreen"
```

Figure 19. `try_register` method in the `RegisterScreen` class which performs the user registration method

Database handling is vital for the user registration process for example. Lines 99-104 show the use of the `database_handler` class to establish a database connection, create the 'users' table if absent, and collect user registration inputs. These inputs are validated by the method `validate_register` and, if correct, the user's details are inserted into the database. A modal confirms successful registration.

Copy Transcription to Clipboard

```
553 def copy_translation(self):
554     def popup():
555         if self.ids.translation.text == '':
556             self.dialog = MDDialog(text="Clipboard empty. Please enter a word to be translated")
557         else:
558             self.dialog = MDDialog(text="Baybayin script copied to clipboard!")
559         self.dialog.open()
560         Clock.schedule_once(self.dismiss_popup,1)
561     pyperclip.copy(str(self.ids.translation.text))
562     popup()
```

Figure 20. `copy_translation` method which enables app to copy the transcription to user's operating system clipboard

The user clicks a clipboard icon which calls the copy_translation method that uses Pyperclip for clipboard operations. This fulfills SC#4. The method contains a nested function `popup` that contains a conditional statement checking the `translation` text attribute (accessed via `self.ids`). Depending on this check, it initializes `self.dialog` with an appropriate message. After creating the dialog (line 786), it's displayed with `self.dialog.open()`, and a callback to `self.dismiss_popup` is scheduled after 1 second to close it (line 787). Outside `popup`, `pyperclip.copy` transfers `translation` text to the clipboard (line 788), and `popup()` is

executed (line 789), merging UI interaction, clipboard manipulation, and event scheduling in the development of the application.

HTTP Methods (GET and POST)

The `CreatePostScreen` class is responsible for handling the creation of new posts in the application's online forum (Success Criteria #6). When the user submits a new post, the `new_post` function is triggered. This function gathers the post's title, content, and a timestamp and then clears the input fields. It also retrieves the current user's username from a global variable.

```
779 class CreatePostScreen(MDScreen):
780     def new_post(self):
781         timestamp = datetime.now().strftime("%b-%d-%Y")
782         title, content = self.ids.post_title.text, self.ids.post_content.text
783         self.ids.post_title.text, self.ids.post_content.text = "", ""
784         global current_user
785         encoded_data = json.dumps({'title': title, 'content': content, 'timestamp':timestamp, 'username':current_user})
786         headers = {'Content-type': 'application/json', 'Accept': 'application/json'}
787         UrlRequest('http://127.0.0.1:5000/upload_post', method='POST', on_success=self.on_upload_success,
788                    req_body=encoded_data, req_headers=headers)
```

Figure 21. `CreatePostScreen` class and the `new_post` method which shows an example of HTTP interaction.

The key element is using `UrlRequest` to send an HTTP POST request to `http://127.0.0.1:5000/upload_post` (line 787). When a user creates a post, the app forms a JSON object with post details, including title, content, timestamp, and username (line 785). This JSON-encoded data is sent to the server, which processes and stores it in the forum database. This facilitates user interactions in the online forum, meeting the app's success criteria.

```
@app.route('/upload_post', methods=['POST'])
def upload_post():
    data = request.get_json()
    token = data.get('token') if data else None

    if not token:
        return jsonify({'message': 'No token provided'}), 401

    try:
        payload = jwt.decode(token, 'your_secret_key', algorithms=['HS256'])
        # SOME LOGIC
        return jsonify({'success': True, 'data': payload}), 200

    except jwt.ExpiredSignatureError:
        return jsonify({'message': 'Token expired'}), 401
    except jwt.InvalidTokenError:
        return jsonify({'message': 'Invalid token'}), 401
```

Figure 22. Server-Side Implementation of the Create Post Functionality

The code defines a Flask route (`/upload_post`) for handling HTTP POST requests. It expects JSON data with a token. Without a token, it responds with a 401 Unauthorized status. With a token, it decodes using 'your_secret_key' (placeholder for this documentation) and the HS256 algorithm. Successful decoding leads to further logic execution and a JSON success response. Expired or invalid tokens result in a 401 status and error message. This setup manages token-based authentication and JSON data for post uploads.

Word Count: 999