**Flet Application - Technical Documentation**

**Table of Contents**

1. **Overview**
2. **Modules and Packages**
   * main.py
   * Router
   * pages/
   * utils/
3. **Route and View Structure**
4. **Components**
   * LoginPage
   * SignupPage
   * DashboardPage
   * PostPage
5. **Utilities**
   * Database
   * Validation
   * style
6. **Configuration Management**
7. **Session Management**
8. **Error Handling and Logging**

**1. Overview**

This Flet application provides a structured framework for building a multi-page app with a routing system. It includes pages for logging in, signing up, a dashboard, and a note-posting feature. The Router class manages page navigation, while modular components provide the user interface and functionality for each page.

**2. Modules and Packages**

**main.py**

The main.py file serves as the entry point for the application, initializing the main Router instance with a page object, which controls the overall page layout, navigation, and updates.

**Router**

The Router class manages the routing logic by:

* Mapping specific URL routes to their respective view functions.
* Handling route changes through the route\_change method, which loads the appropriate view based on the route.

This class initializes routing when the application starts and enables navigation between views.

**pages/**

Each page of the app is represented as a class within this directory. The primary classes are:

* **LoginPage**: Handles user login and authentication.
* **SignupPage**: Manages user registration.
* **DashboardPage**: Displays the main dashboard with options for navigating to other sections.
* **PostPage**: Allows users to add notes with a priority level and view existing notes.

Each page class includes a view() method, which generates the layout and content for that particular page.

**utils/**

The utils directory contains helper classes:

* **Database**: Contains database functions for CRUD operations, enabling persistent storage of user and note data.
* **Validation**: Offers data validation utilities used across pages (e.g., validating input fields).
* **style**: Provides shared style definitions and configurations (e.g., colors, fonts).

**3. Route and View Structure**

| **Route** | **View Function** | **Description** |
| --- | --- | --- |
| / | LoginPage().view | The login page for user authentication. |
| /signup | SignupPage().view | The registration page for new users. |
| /dashboard | DashboardPage().view | The main dashboard after login. |
| /posting | PostPage().view | The page for adding and viewing notes. |

Routing is managed by the Router class. The app\_router dictionary maps each route to its respective view method.

**4. Components**

**LoginPage**

* **Purpose**: To authenticate users by taking in credentials.
* **Main Elements**:
  + Username and password input fields.
  + A login button that triggers the authentication process.
* **Methods**:
  + view(): Generates and returns the login page layout.

**SignupPage**

* **Purpose**: To allow new users to register.
* **Main Elements**:
  + Input fields for user information.
  + A sign-up button that validates and submits the information to the database.
* **Methods**:
  + view(): Generates and returns the sign-up page layout.

**DashboardPage**

* **Purpose**: Provides an overview and access to different sections, such as posting and dashboard features.
* **Main Elements**:
  + Sidebar navigation menu with buttons.
  + A header showing the page title and user profile picture.
  + Editable fields for channel and token settings.
* **Methods**:
  + view(): Generates the dashboard page layout.
  + save\_settings(e): Updates token and channel link configurations and stores them.

**PostPage**

* **Purpose**: To allow users to add, prioritize, and view notes.
* **Main Elements**:
  + Note input and priority dropdown.
  + A list to display saved notes.
* **Methods**:
  + view(): Generates the page layout for posting notes.
  + save\_note\_handler(e): Handles note submission and saves data to the database.
  + load\_notes(search\_query, sort\_by): Loads and displays notes based on search and sort criteria.
  + update\_notes\_view(e): Updates the notes view based on the current search or sort input.
  + delete\_note\_handler(note\_id): Deletes a selected note.

**5. Utilities**

**Database**

Handles database operations for creating, retrieving, and deleting records for users and notes. Key methods include:

* create\_note(): Adds a new note to the database.
* get\_user\_notes\_sorted(): Retrieves notes for a specific user with sorting and filtering options.
* delete\_note(): Deletes a note based on its unique identifier.

**Validation**

Provides methods for validating user input, such as checking field requirements or formats before submitting data to the database.

**style**

Defines reusable style constants used across the app (e.g., colors and fonts). Examples include:

* menuColorFont: Color for menu text.
* secondaryBqColor: Background color for secondary elements.
* defaultFontColor: Default font color for text elements.

**6. Configuration Management**

Environment variables for sensitive or configurable settings, such as TOKEN\_BOT and CHANNEL\_LINK, are managed using a .env file and loaded via dotenv. The DashboardPage allows users to set these variables directly from the UI.

**7. Session Management**

The application uses page.session to manage session-specific data, such as authentication status (auth\_user). This ensures user sessions persist across navigation and enables secure access control for pages like the dashboard.

**8. Error Handling and Logging**

**Error Handling**

Error handling is performed primarily through conditional checks and validation functions:

* Input fields are validated via the Validation class.
* Missing or incorrect values in the .env file are checked before loading sensitive configurations.

**Logging**

Debugging outputs (e.g., print statements) are used to display tokens and other data when loaded. These should ideally be replaced with a logging mechanism for production.