1. Executive Summary

This project involved refactoring a provided Flask-based web application to fix existing bugs and implement two core features: a persistent **Note-Taking** system and a **Regex Matcher** utility. The resulting application allows users to store a history of notes and test Python-style regular expressions against custom text strings within a single, stable interface.

2. Bug Documentation & Technical Resolutions

Upon initial review of the source files (app.py and home.html), several critical architectural and logic bugs were identified. Below is a detailed report of the fixes applied:

Bug B001: Improper Route Methods (405 Method Not Allowed)

- **Description:** The original home route (/) was restricted to POST requests only. This prevented the application from loading when first accessed via a standard browser GET request.
- **Resolution:** Modified the route decorator to methods=["GET", "POST"]. This ensures the page loads initially via GET and handles note submissions via POST

Bug B002: Incorrect Data Retrieval Method

- **Description:** The backend attempted to retrieve user input using request.args.get(). In Flask, args targets URL parameters. Since the frontend forms use the POST method, the data was not being captured.
- **Resolution:** Refactored the backend to use request.form.get(), which is the correct method for accessing data sent in the request body of a POST form.

Bug B003: Missing Frontend Form Attributes

- **Description:** The HTML <form> tags lacked the method="POST" attribute. By default, browsers send form data via GET, appending the input to the URL and bypassing the backend's intended processing logic.
- **Resolution:** Explicitly added method="POST" to all form tags in home.html to ensure secure and correct data transmission.

Bug B004: Lack of Input Validation (Regex Crash)

- **Description:** Entering an invalid regular expression (e.g., an unclosed parenthesis () caused the Python re module to throw a re.error, resulting in a server-side crash (500 Internal Server Error).
- **Resolution:** Wrapped the matching logic in a try-except block. If an invalid regex is entered, the error is caught, and a user-friendly error message is displayed on the frontend instead of crashing the server.

3. Implementation Details

The Regex Matcher Engine

To satisfy the task of "cloning core regex101 functionality," the backend was updated to identify every instance of a match rather than just the first one.

- **Logic:** Utilized `re.findall(pattern, test_string)` to return a list of all matching substrings.
- **Rendering:** Implemented Jinja2 template logic (`{% for match in matches %}`) to dynamically generate the results list only when matches are found.

Note Persistence

A global Python list (`notes = []`) serves as the application's temporary state. The refactoring ensures that:

1. Adding a note redirects/renders the list immediately.
2. Running a Regex search does not wipe the existing notes history.

4. Verification & Testing

The application was verified through a combination of manual and automated testing:

- **Manual Testing:** Verified that the "Submit" button triggers the matching logic and that notes are appended correctly to the unordered list below the input field.
- **Automated Testing:** A custom `test_app.py` script was used to send simulated POST requests.
- **Test 1 (Notes):** Confirmed successful string injection into the HTML list.
- **Test 2 (Regex):** Confirmed that the backend correctly identifies multiple digit matches (e.g., `\d+`) in a complex string.

5. Conclusion

The refactored application is now fully functional and robust. It effectively separates the logic for note management and regex matching while providing a clean, bug-free user experience that adheres to the provided project requirements.