Objective

Define a comprehensive testing plan to ensure the stability and quality of the application, covering key user flows, UI validations, and app state transitions. This plan will also include contributing to and coordinating automated test coverage.

Responsibilities

- **Prepare and maintain a manual QA testing strategy** that includes:
  - **Key user flows**
    - Identify and document critical end-to-end scenarios that reflect real user interactions.
    - Example: User registration, login/logout, hotel search, booking, payment, favorites management.
  - **UI validations**
    - Verify layout, responsiveness, alignment, and consistency across different devices.
    - Check compliance with design specifications (colors, fonts, element sizes).
    - Validate error messages, tooltips, and other UI feedback.
  - **App state transitions**
    - Test behavior across app state changes (e.g., background/foreground, offline/online, screen rotation).
    - Verify proper data persistence and recovery after crashes or force-stops.
- **Prioritize test cases**
  - Classify test cases as:
    - **High priority:** Core functionality, critical user journeys, payment flows.
    - **Medium priority:** Non-critical workflows, settings, and preferences.
    - **Low priority:** Cosmetic elements, non-blocking issues.
  - Define smoke and regression suites based on priority.
- **Test data preparation**
  - Design and prepare test data sets for:
    - Positive and negative scenarios.
    - Boundary value conditions.
    - Edge cases (e.g., large inputs, invalid formats).
  - Ensure availability of mock or sandbox environments for safe test data usage.

Execution Plan

- **Manual testing**
  - Execute planned test cases for every release/sprint.
  - Log defects with detailed steps, screenshots, and environment info.
  - Re-test and close defects after fixes.
- **Automated testing**
  - Identify repetitive or critical paths for automation (e.g., login, hotel search, booking flow).
  - Develop and maintain automated test scripts (e.g., using Maestro, Appium, Robot Framework).
  - Integrate automated tests in CI/CD pipelines to enable continuous testing.
- **Integration / Widget Test Execution Strategy**

Step	**Activity**	**Example**
Test identification	List key widgets and integrated modules that need validation	Search bar, hotel card, booking summary, favorites toggle
Setup test harness	Configure app's test environment for integration/widget tests	Flutter's testWidgets, custom mock services, use of MockProvider, Mockito, fakeAsync
Write tests	- Verify UI renders as expected. - Validate interaction triggers the correct actions. - Check state changes (enabled/disabled buttons, dynamic fields).	- Verify SearchBar shows suggestions on input. - Verify FavoriteIcon updates its state on tap.
Mock dependencies	Isolate API / database calls using stubs/mocks	Mock hotel service returning fixed hotel data
Automate execution	Add to CI (GitHub Actions, Jenkins, etc.) for automated runs	Example: flutter test integration_test/ or maestro test suite

Tools

- **Flutter** : flutter_test
- **Maestro/Robotframework** : for lightweight UI integration validation
- **CI/CD :** GitHub Actions to automate test runs

- **Reporting**
    - Share test execution results and defect reports after each cycle.
    - Maintain dashboards for tracking test coverage and defect trends.