Budget Application Test Plan

**​Participants**

​Darius Smith

​Anthony Graber

​Maysun Dietrick

​Jose Guzman-Sanchez

1.0

Testing Purpose

The budget app needs to be tested for functionality, to make sure all features of the app are working as intended and all known errors are addressed before release.  The app also needs to be tested for security flaws to prevent unauthorized access to user’s information.

Objectives

1. Verify that all features of the app work as intended
2. Find and remove software bugs.
3. Verify that the app is secure from unauthorized users.
4. Verify that the app can store and retrieve information from the database.
5. Measure performance between user functions and response time between database and the app.

Testing Strategies and Methodologies

1. **Unit Testing:** Unit testing is necessary for this app since it focuses on testing the individual components and functions required to work correctly. The budgeting app’s functionality depends on multiple discrete operations such as calculations, data manipulation, and data entry. These components need to be tested successfully prior to the addition of more complex testing techniques in order to catch errors as early as possible.
2. Functional Testing: At this level, functional testing is used to test the app’s main features such as budget creation, financial tracking, data manipulation and report generation.
3. **Integration Testing:** Integration testing is used to verify that the interactions between different components of the app work in tandem correctly. This level of testing is useful in identifying incorrect behaviors between multiple components.
4. Functional Testing: Ensures that individual components that work alongside other components that perform functions such as data input, processing, and visualization are successfully implemented.
5. Usability Testing: Ensure integrated components provide a smooth user experience, which is vital since the successful creation of a budgeting app relies on its ease of use from a user perspective.
6. **System Testing:** The purpose of system testing is to evaluate the application in its entirety to verify that it meets the requirements specified. The process of creating, modifying, managing, and generating reports should be tested. This level of testing includes testing for security, ensuring that data is kept independent of other processes and kept secure.
7. Functional Testing: Checks the entire system’s behavior when using the app as a whole
8. Security Testing: Ensures the user’s data is protected from other users and other processes with unauthorized access.
9. **Acceptance testing**: This level of testing is needed to test real-world scenarios where this app is used. It is essential to validate the objectives of the app’s functions and meet the end users’ expectations.
10. Usability Testing: Used to ensure the app meets users’ needs and meets the specified requirements.
11. Functional Testing: Tests the app’s functions and confirms that all requirements are met.

Test Environment and Tools

Test Environment

* Required HW Configurations, SW configurations, and Test Data Requirements.

Hardware Configurations

**1. Server Hardware (Staging/Testing Environments)**

* **CPU**: Minimum of 4-core processor (Intel or AMD) to handle multiple user requests, simultaneous tests, and performance load testing.
* **RAM**: Minimum of 16 GB for smooth testing operations, especially when running multiple tests such as unit tests, performance testing with Locust, and integration tests.
* **Storage**: SSD (Solid-State Drive) with at least 100 GB storage. A fast SSD is recommended to handle the read/write operations efficiently, especially when working with MongoDB and test data logs.
* **Network**: A fast and stable network connection, preferably with 1 Gbps bandwidth for performance testing with multiple simulated users.

**2. Test Client Machines (Running Automated Tests)**

* **CPU**: Minimum of dual-core processor for local test execution.
* **RAM**: **8 GB** or more for running test suites (Selenium, unit tests, etc.) efficiently.
* **Browsers**: Install various browsers for compatibility testing (e.g., Chrome, Firefox, Safari, Edge).

Software Configurations

**1. Operating System**

* **Ubuntu 20.04+ (LTS)**: Recommended for server environments (staging or test servers).
* **Windows 10/11** or **macOS**: Can be used for local test machines or developer systems.

**2. Web Framework and Application**

* **Python 3.9+**: Required for running the Flask-based web application and the various Python testing tools (Pytest, unittest, etc.).
* **Flask**: The web framework for the application itself.

**3. Database**

* **MongoDB**: This will serve as the database for the application.
* MongoDB will be installed and running on both the staging server and test environments.
* MongoDB will be created with relevant collections and mock test data, such as users, transactions, and budget categories.

**4. Web Server for Staging**

* **Nginx** will be used in front of the Flask application for staging environments to simulate production-like behavior.

Tools and Framework

Tools to be used to support testing activities.

Test Environment

1. **Application Framework:**

* **Flask**: This is the web framework being used to develop the web-based budgeting application.

1. **Database:**

* **MongoDB**: The NoSQL database used for storing user data, transactions, and other application-related data.

1. **Server Environment:**

* The application will be hosted in a local environment.

1. **Browsers for Testing:**

* Ensure compatibility testing on common browsers:
* Chrome, Firefox, Safari, Edge.

Test Tools Overview

**1. Python Unittest (Unit Testing)**

* **Purpose**: Unit tests focus on testing individual components of the code, such as functions and methods in the Flask app.
* **Tool**: The unittest module in Python will be used to write tests to check each function’s correctness.

**2. Locust (Performance Testing)**

* **Purpose**: To evaluate the performance of the web application under different user loads, simulating how it handles concurrent users performing operations like budget entries and data retrieval.
* **Tool**: Locust will simulate real-world traffic and measure response times, throughput, and the system's ability to scale.

**3. Selenium (Web Page Testing)**

* **Purpose**: To perform automated browser testing to verify that the web application works as expected across different browsers (e.g., form submission, navigation).
* **Tool**: Selenium will automate the user interactions with the web pages and perform end-to-end UI testing.

**4. Python Unittest (Integration Testing)**

* **Purpose**: To perform integration testing to ensure that different components of the application (Flask app, MongoDB, etc.) work together as expected.
* **Tool**: Python Unittest will be used for testing interactions between different modules, such as routes and database operations, as well as API response codes.

**5. Python requests (Security Testing)**

* **Purpose**: To ensure the application is secure and free of vulnerabilities such as cross-site scripting (XSS), SQL injection, and insecure access.
* **Tool**: The requests library is used to perform security tests on the application’s endpoints.

Test Scenarios and Cases

**User Registration**

* **Valid user registration**: Ensure that new users can register with valid credentials.
* **Invalid user registration**: Test registration with invalid credentials, such as invalid email formats or weak passwords.
* **Duplicate registration**: Attempt to register a user with an email or username that already exists in the system.

**Login**

* **Valid login**: Confirm that users can log in with valid credentials
* **Invalid login**: Test login attempts with incorrect passwords or non-existent usernames.

**Budget Creation**

* **Valid budget creation**: Check that users can create a new budget with valid data.
* **Invalid budget creation**: Try creating a budget with missing or invalid data fields.

**Expense Tracking**

* **Valid expense tracking**: Ensure users can log expenses with valid data
* **Invalid expense tracking**: Test adding expenses with incomplete or invalid data
* **Edge cases**: Enter zero or negative amounts to see how the system handles these scenarios

**Data Visualization**

* **Budget and expense reports**: Verify that users can generate visual reports for their budgets and expenses
* **Data accuracy**: Check that the visual data matches the underlying data in the system.

**Security**

* **Valid Password Test**: Ensure that only users with the correct password can login.
* **Invalid Password Test**: Ensure the system denies access to users with incorrect passwords.

**Database**

* **Data Insertion**
* **Valid data insertion**: Test inserting a complete and valid record into the database to ensure it accepts correct data.
* **Invalid data insertion**: Attempt to insert a record with missing or invalid fields to verify the system handles the error.
* **Duplicate data insertion**: Try inserting a duplicate record and check that the database enforces unique constraints
* **Data Retrieval**
* **Valid data retrieval**: Query the database for existing records and confirm that the correct data is returned.
* **Empty result retrieval**: Perform a query that should return no results and verify that the database handles it properly.

Test Data Requirements

* **Sample User Credentials (valid and invalid)**: Create test cases with a mix of valid and invalid user credentials to ensure the system handles both successful logins and authentication errors correctly.
* **Mock data for budgets and expenses**: Generate mock data representing various budget scenarios and expense entries to test the functionality of system features.
* **Edge cases (zero and negative values)**: Develop test cases that include edge scenarios, such as zero and negative values, to validate the system’s ability to process edge case inputs and handle errors.

Expected Results and Acceptance Criteria

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Expected Result** | **Acceptance Criteria** |
| Valid User Registration | Users can register with valid credentials without errors. | Registration works for valid credentials. |
| Invalid User Registration | Attempts to register with invalid credentials are properly denied, with clear error message displayed. | System provides appropriate error messages for invalid registration attempts. |
| Valid User Login | Users can log in with valid credentials without issues. | Valid credentials grant access. |
| Invalid User Login | Login attempts with incorrect credentials are properly denied, with appropriate error messages displayed. | Incorrect credentials deny access with clear error messages. |
| Valid Budget Creation | Users can create budgets with valid data without errors. | Budgets are created and saved correctly for valid data inputs. |
| Invalid Budget Creation | Attempts to create budgets with invalid data are properly denied, with clear error messages displayed. | System prevents budget creation with invalid data, providing appropriate error messages. |
| Valid Expense Tracking | Users can log expenses with valid data without errors. | Expenses are tracked and saved correctly for valid data entries. |
| Invalid Expense Tracking | Attempts to log expenses with invalid data are properly denied, with clear error messages displayed. | System prevents expense logging with invalid data, providing appropriate error message. |
| Expense Tracking with Edge Cases | System handles edge cases, such as zero and negative values, correctly. | System accurately processes zero and negative values and handles errors correctly. |
| Budget and Expense Reports | Users can generate visual reports for their budgets and expenses. | Users successfully and easily generate reports for their budgets and expenses. Reports are accurate and match the underlying data. |
| Visual Data Accuracy | Visual data and reports accurately reflect the budget and expenses logged by the user. | Generated reports match the data accurately and present it in a clear, understandable format. |
| Valid Passwords | Users can log in with valid credentials without issues. | Valid credentials grant access without errors. |
| Invalid Passwords | Login attempts with incorrect passwords are properly denied, with appropriate error messages displayed. | Incorrect passwords deny access with clear error messages. |
| Valid Data Insertion | Valid records are inserted successfully. | System accepts and stores valid data entries without issues. |
| Invalid Data Insertion | Invalid records are rejected with clear error messages. | System prevents storage of invalid data, providing appropriate feedback. |
| Duplicate Data Insertion | Duplicate records are correctly handled based on system rules. | System either prevents duplicate entries or handles them as specified, with clear error messages. |
| Valid Data Retrieval | Database is queried for existing records and the correct data is returned. | Accurate and complete data returned for valid queries. |
| Empty Result Retrieval | Query is performed that should return no results and the system handles it properly. | System handles empty results correctly, providing appropriate error messages. |

Test Case and Results Documentation

All test cases will include the following criteria:

1. A unique identifier
2. A description of what the test aims to achieve.
3. A list of the test steps
4. An expected result

**Integration Tests - API Endpoints**

**Reporting Endpoints**

* STP-1: Authenticated Access to Generate Report
  + Verify that an authenticated user can successfully access the reporting endpoint and receive the correct report data
* STP-2: Unauthenticated User Generate Report
  + Verifies that an unauthenticated user cannot generate a report for an existing budget
* STP-3: Generate Report for Non-Existent Budget
  + Verifies that the system correctly handles requests to generate a report for a non-existent budget.

**Registration and Login Endpoints**

* STP-12: Test Registration Authenticated
  + Verify that a new user can successfully register and is redirected to the login page.
* STP-13: Test Registration Duplicate
  + Verify that attempting to register a user that already exists returns an appropriate error.
* STP-14: Test Login Authenticated
  + Verify that a registered user can successfully log in with valid credentials.
* STP-15: Test Login Unauthorized
  + Verify that a login attempt with incorrect credentials returns an appropriate error.

**Unit Tests – Model Functionality**

**Budget: Create Budget**

* STP-5: Check creating budget with valid input
  + Check that budget was added to the database
* STP-6: Test creating a budget with invalid input
  + Check that budget was not added to the database
* STP-7: Verify no budget was added with invalid data
  + Check that budget was not added to the database
* STP-8: Test creating a budget with missing amount field
  + Check that budget was not added to the database

**Budget: Generate Report**

* STP-16: Test Generate Report Normal
  + Verify that a report is generated correctly for a budget with both incomes and expenses.
* STP-17: Test Generate Report No Income
  + Verify that a report is generated correctly for a budget with no incomes.
* STP-18: Test Generate Report No Expense
  + Verify that a report is generated correctly for a budget with no expenses.
* STP-19: Test Generate Report Empty Input
  + Verify that a report is generated correctly for a budget with no incomes and no expenses.
* STP-20: Test Generate Report Mixed Input
  + Verify that a report is generated correctly for a budget with mixed (integer and float) incomes and expenses.

**Budget: Calculate Savings**

* STP-21: Test Calculate Savings Normal
  + Verify that savings are calculated correctly for a budget with both incomes and expenses.
* STP-22: Test Calculate Savings No Income
  + Verify that savings are calculated correctly for a budget with no incomes.
* STP-23: Test Calculate Savings No Expenses
  + Verify that savings are calculated correctly for a budget with no expenses.
* STP-24: Test Calculate Savings Empty Input
  + Verify that savings are calculated correctly for a budget with no incomes and no expenses.
* STP-25: Test Calculate Savings Mixed Input
  + Verify that savings are calculated correctly for a budget with mixed incomes and expenses.
* STP-26: Test Calculate Savings Negative Input
  + Verify that savings are calculated correctly for a budget with negative income and expense entries.
* STP-27: Test Calculate Savings Multiple Steps
  + Verify that savings are calculated correctly when incomes and expenses are added incrementally.

**Budget: Remove Income and Expenses**

* STP-44: Test Remove Single Income
  + Verify that a single income entry can be removed from the budget.
* STP-45: Test Remove Multiple Incomes
  + Verify that multiple income entries can be removed from the budget.
* STP-46: Test Remove Single Expense
  + Verify that a single expense entry can be removed from the budget.
* STP-47: Test Remove Multiple Expenses
  + Verify that multiple expense entries can be removed from the budget.
* STP-48: Test Remove Incomes and Expenses
  + Verify that both income and expense entries can be removed from the budget.

**Budget: Add Income**

* STP-28: Test Add Income
  + Verify that an income entry can be added to the budget.
* STP-29: Test Add Income Zero Amount
  + Verify that an income entry with a zero amount is not added to the budget.
* STP-30: Test Add Income Negative Amount
  + Verify that an income entry with a negative amount is not added to the budget.
* STP-31: Test Add Income Multiple Incomes
  + Verify that multiple income entries can be added to the budget.
* STP-32: Test Add Income Large Amount
  + Verify that a large income entry can be added to the budget.
* STP-33: Test Add Income Float Amount
  + Verify that an income entry with a float amount can be added to the budget.
* STP-34: Test Add Income Empty Amount
  + Verify that adding an income entry with an empty amount raises a ValueError
* STP-35: Test Add Income Empty Source
  + Verify that adding an income entry with an empty source raises a ValueError.
* STP-36: Test Add Income None Amount
  + Verify that adding an income entry with a None amount raises a ValueError.
* STP-37: Test Add Income String Amount
  + Verify that adding an income entry with a string amount raises a TypeError.

**Budget: Calculate Total Income and Expenses**

* STP-38: Test Calculate Total Income Single
  + Verify that the total income is calculated correctly when there is a single income entry.
* STP-39: Test Calculate Total Income Multiple
  + Verify that the total income is calculated correctly when there are multiple income entries.
* STP-40: Test Calculate Total Income Empty
  + Verify that the total income is calculated correctly when there are no income entries.
* STP-41: Test Calculate Total Expenses Single
  + Verify that the total expenses are calculated correctly when there is a single expense entry.
* STP-42: Test Calculate Total Expenses Multiple
  + Verify that the total expenses are calculated correctly when there are multiple expense entries.
* STP-43: Test Calculate Total Expenses Empty
  + Verify that the total expenses are calculated correctly when there are no expense entries.

Budget: Add Expense

* STP-49: Test Add Expense Valid
  + The expense list remains empty after attempting to add a negative amount expense.
* STP-50: Test Add Expense Zero Amount
  + Tests whether adding an expense with a zero amount is ignored (i.e., not added to the expense list).
* STP-51: Test Add Expense Negative Amount
  + Tests whether adding an expense with a negative amount is ignored (i.e., not added to the expense list).

**System Tests**

**Selenium (Web-Based) Tests**

* STP-10: Verify Login using Selenium
* STP-11: Verify Budget Creation using Selenium

**Acceptance Tests**

* STP-52: Add Expense to Budget
* STP-53: Add Income to Budget
* STP-54: Generate Budget Report
* STP-55: User Manages Budget
* STP-56: User Registration
* STP-57: User Login

Test Case Tracking

Regular reports will be generated summarizing the status of test cases, highlighting passed and failed tests. Reports will also include our current test coverage status.

All tests will be tracked and documented with the following criteria:

1. Name of Test Suite
2. Report Date
3. Total Tests Executed
4. Number of Tests Passed
5. Number of Tests Failed
6. Coverage
7. For each test case:
   1. Test Case ID
   2. Test Case Description
   3. Status
   4. Comments