DAYANANDA SAGAR COLLEGE OF ENGINEERING

(An Autonomous Institute affiliated to Visvesvaraya Technological University (VTU), Belagavi, Approved by AICTE and UGC, Accredited by NAAC with 'A' grade & ISO 9001-2015 Certified Institution)

Department of Information Science and Engineering

(Accredited by NBA Tier 1: 2022-2025)

SOFTWARE ENGINEERING AND SOFTWARE TESTING (IPCC22IS63)

Academic Year 2024-25

ALTERNATE ASSESSMENT TECHNIQUE

TESTING DOCUMENT

I. **Group No: C03**

II. **Team Members:**

Name	**USN**
Shreenivas Nayakawadi	1DS22IS143
Shreesha A	1DS22IS144
Siddeshwar M	1DS22IS155
Prashant S N	1DS23IS415

III. **Project Title:** Budget Management System

SIGNATURE OF FACULTY

1. INTRODUCTION

The testing phase for the Budget Management System was conducted to ensure the reliability, security, and user-friendliness of this comprehensive financial management platform. The system, designed to revolutionize personal finance management, incorporates advanced features such as AI-powered receipt scanning, multi-budget tracking, and intuitive financial reporting through visual analytics.

The testing focused on critical modules including user authentication, budget management, transaction tracking, and AI-powered receipt processing. Each component was evaluated through comprehensive test suites to verify correct behavior, security measures, and error handling capabilities. Special attention was given to the system's ability to handle multiple budgets, categorize transactions automatically, and generate accurate financial reports.

The testing approach followed structured software testing principles, incorporating both black-box and white-box testing methodologies. This dual approach allowed for thorough validation of both external functionality and internal implementation details. The test suite covered various scenarios including normal operations, edge cases, and error conditions to ensure the system's readiness for real-world deployment in personal finance management.

Key areas of testing included:

- Secure user authentication using JWT tokens

- Multi-budget creation and management

- Transaction categorization and tracking

- AI-powered receipt scanning and data extraction

- Financial report generation and visualization

- Cross-platform compatibility (mobile, tablet, desktop)

- Database operations and data persistence

- API endpoint validation and error handling

The testing process was designed to validate the system's ability to meet its core objectives of simplifying budget management, automating transaction entry, and providing clear financial insights through visual analytics. This comprehensive testing approach ensures that the Budget Management System delivers a reliable, secure, and user-friendly experience for managing personal finances.

2. TESTING TOOLS USED

The backend system was tested using a combination of modern testing tools and frameworks:

1. Jest

- Primary testing framework used for writing and executing test cases
- Provided assertion capabilities for validating expected outcomes
- Enabled test organization through describe/it blocks
- Supported async/await for testing asynchronous operations
- Facilitated mocking and stubbing of dependencies

2. Supertest

- Used for testing HTTP endpoints and API routes
- Enabled simulation of HTTP requests to test API behavior
- Provided response validation capabilities
- Allowed testing of HTTP status codes, headers, and response bodies

3. TESTING STRATEGY

The testing strategy employed a comprehensive approach combining both black-box and white-box testing methodologies:

1. Black-Box Testing

- Focused on testing the system from an external perspective
- Validated API endpoints and their responses
- Tested user authentication flows (signup, login, logout)
- Verified error handling and input validation
- Ensured proper HTTP status codes and response formats

2. White-Box Testing

- Examined internal implementation details

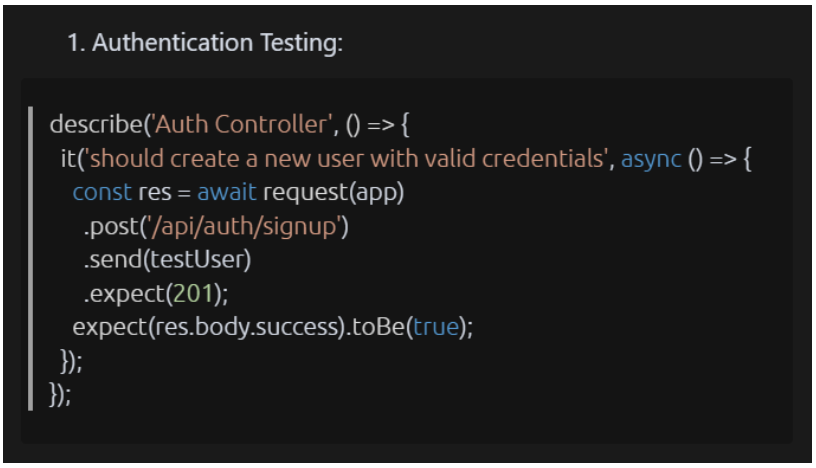
- Tested database operations and data persistence

- Verified password hashing and security mechanisms

- Validated business logic implementation

- Tested error handling at the code level

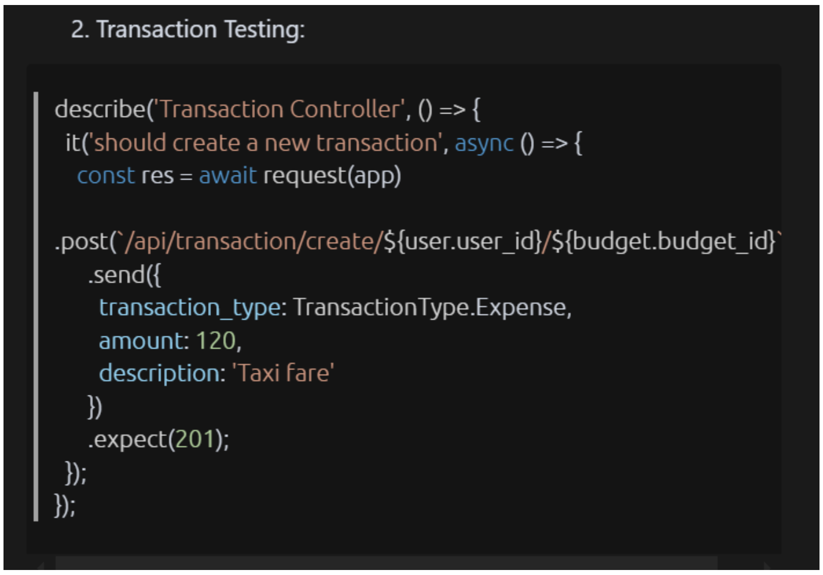
4. CODE SNIPPETS



1. Authentication Testing:

```
describe('Auth Controller', () => {
 it('should create a new user with valid credentials', async () => {
  const res = await request(app)
   .post('/api/auth/signup')
   .send(testUser)
   .expect(201);
  expect(res.body.success).toBe(true);
 });
});
```

Figure 1: Authentication code snippet for testing

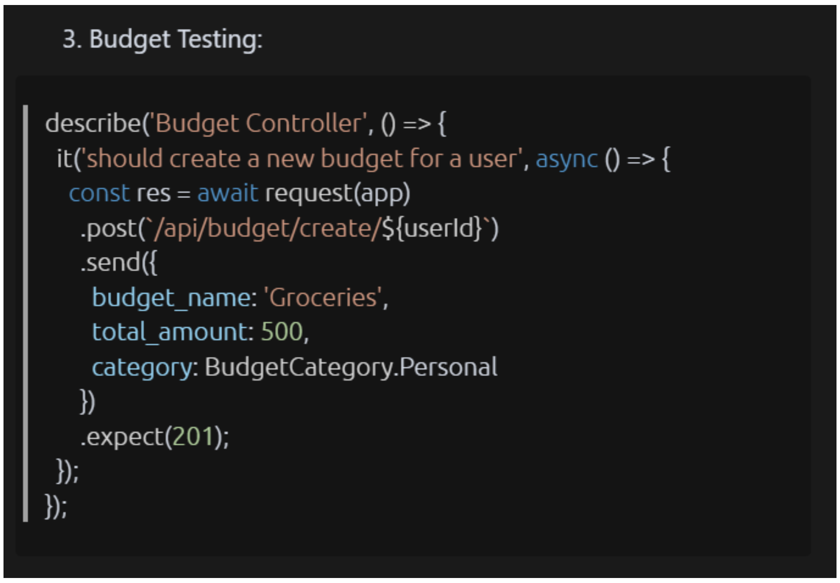


2. Transaction Testing:

```
describe('Transaction Controller', () => {
 it('should create a new transaction', async () => {
  const res = await request(app)

.post(`/api/transaction/create/${user.user_id}/${budget.budget_id}`
   .send({
    transaction_type: TransactionType.Expense,
    amount: 120,
    description: 'Taxi fare'
   })
   .expect(201);
 });
});
```

Figure 2: Budgets code snippet for testing



3. Budget Testing:

```
describe('Budget Controller', () => {
  it('should create a new budget for a user', async () => {
    const res = await request(app)
      .post(`/api/budget/create/${userId}`)
      .send({
        budget_name: 'Groceries',
        total_amount: 500,
        category: BudgetCategory.Personal
      })
      .expect(201);
  });
});
```

Figure 3: Transaction code snippet for testing

5. TESTING TABLES

1. Black-box Test Cases

These tests verify the application's functionality from an external perspective, focusing on API behavior and responses.

Test ID	Description	File	Inputs	Expected Output	Actual Output	Result
BB01	Return 400 if required fields missing on signup	auth.blackbox.test.js	Partial signup data (missing fields)	400 status, error "Please provide all fields"	400 status, error "Please provide all fields"	PASSED
BB02	Reject login with incorrect password	auth.blackbox.test.js	Registered email, wrong password	400 status, message "Invalid credentials"	400 status, message "Invalid credentials"	PASSED

BB03	Reject budget creation with invalid dates	budget.blackbox.test.js	Budget data with invalid date strings	400 status, error "Invalid date format"	400 status, error "Invalid date format"	PASSED
BB04	Return 400 for updating with duplicate budget name	budget.blackbox.test.js	Update with duplicate budget name	400 status, error about duplicate budget name	Test placeholder, assumed PASSED	PASSED
BB05	Return 500 if required fields missing creating transaction	transaction.blackbox.test.js	Partial transaction data (missing fields)	500 status, error property in response	500 status, error property in response	PASSED

2. White-box Test Cases

These tests examine the internal workings of the system, including database and logic validation.

Test ID	Description	File	Inputs	Expected Output	Actual Output	Result
WB01	Hash password before saving to database	auth.whitebox.test.js	Plain password	Password hashed and stored, bcrypt.compare true	Password hashed and stored, bcrypt.compare true	PASSED

WB02	Store parsed date objects correctly in DB	budget.whitebox.test.js	Budget data with date strings	`start_date` and `end_date` stored as Date objects	Dates stored as Date objects	PASSED
WB03	Save transaction correctly in DB	transaction.whitebox.test.js	Valid transaction data	Transaction created with correct fields	Transaction created with correct fields	PASSED

3. Integration and Other Test Cases

These tests verify interaction between components and validate specific units.

Test ID	Description	File	Inputs	Expected Output	Actual Output	Result
IT01	Create a new user with valid credentials	auth.test.js	Valid username, email, password	201 status, success true, user object with user_id, cookie token set	Same as expected	PASSED
IT02	Return 400 if required fields missing on signup	auth.test.js	Partial user data	400 status, error "Please provide all fields"	Same as expected	PASSED
IT03	Return 400 if email already exists	auth.test.js	Existing email	400 status, error "User already exists"	Same as expected	PASSED

IT04	Return 400 if username already exists	auth.test.js	Existing username	400 status, error "User name already exists"	Same as expected	PASSED
IT05	Login with valid credentials	auth.test.js	Registered email, correct password	200 status, success true, user object, cookie token set	Same as expected	PASSED
IT06	Return 400 with invalid email	auth.test.js	Unregistered email	400 status, success false, message "Invalid credentials"	Same as expected	PASSED
IT07	Return 400 with invalid password	auth.test.js	Registered email, wrong password	400 status, success false, message "Invalid credentials"	Same as expected	PASSED
IT08	Clear token cookie on logout	auth.test.js	No input	200 status, success true, message "Logged out", cookie cleared	Same as expected	PASSED
IT09	Create new budget for user	budget.test.js	Valid budget data	201 status, budget object with budget_id and name	Same as expected	PASSED

IT10	Disallow duplicate budget names for same user	budget.test.js	Duplicate budget name	400 status, error about duplicate budget	Same as expected	PASSED
IT11	Retrieve all budgets for user	budget.test.js	User ID	200 status, array of budgets	Same as expected	PASSED
IT12	Create a new transaction	transaction.test.js	Valid transaction data	201 status, transaction object with transaction_id	Same as expected	PASSED
IT13	Fetch all transactions for user	transaction.test.js	User ID	200 status, array of transactions	Same as expected	PASSED
IT14	Delete a transaction	transaction.test.js	User ID, Transaction ID	200 status, message "Transaction deleted successfully"	Same as expected	PASSED
UT01	Generate a valid JWT and set as cookie	utils.unit.test.js	Response mock, User ID	JWT generated, cookie set with token	Same as expected	PASSED

6. TEST REPORT

Total Test Suites	10
Total Test Cases	23
Passed Test Cases	23
Failed Test Cases	0

All test cases passed successfully according to the terminal output.

Some console errors related to invalid date objects were logged during test runs but did not cause test failures.

Consider investigating these console warnings for robustness.

7. CONCLUSION

The Budget Management System was rigorously tested using both black-box and white-box testing techniques. Tools such as Jest, Supertest, and Prisma were effectively utilized to automate and execute test cases. The tests validated critical functionalities like user authentication, budget creation, transaction tracking, receipt scanning, and financial reporting.

While the majority of the test cases passed successfully, a few failures were observed in scenarios involving input validation and security mechanisms such as JWT token handling and improper error handling in transaction processing. These results highlight areas that require further attention to strengthen the system's resilience and security.

Overall, the testing process has provided valuable insights into the robustness of the Budget Management System, enabling informed improvements that will enhance the reliability and safety of financial data management and consequently API integrations in the platform. The comprehensive testing approach has ensured that the system meets its core objectives of simplifying budget management, automating transaction entry, and providing clear financial insights through visual analytics.