**Finance Management System**

-Bavatharani S

**1. Project Overview**

The Finance Management System is designed to help individuals track and manage their personal finances through efficient recordkeeping and categorized expense management. The project applies object-oriented programming (OOP), SQL database connectivity, exception handling, and unit testing to demonstrate real-world software development concepts. Users can register, log in, add expenses, view categorized reports, and perform CRUD operations in a secure and structured environment.

**2. Objectives**

This system will allow users to:

* Register and log in securely.
* Add, update, delete, and view their expenses.
* Categorize expenses into various types (Food, Travel, Utilities, etc.).
* Generate reports of expenses by user and time period.
* Persist data in a SQL-based database with proper relational design.
* Maintain proper exception handling and unit testing.

**3. Technologies Used**

|  |  |
| --- | --- |
| Technology | Purpose |
| Python 3.x | Core programming language |
| MySQL | Relational database for storage |
| MySQL Connector | Python package for DB interaction |
| unittest / pytest | Testing framework |
| GitHub | Version control and submission |

**4. Functional Requirements**

* Users can register with a username, password, and email.
* Users can log in securely to access their account.
* Users can view all their expenses in a structured list.
* Users can update specific expenses (amount, date, category, description).
* Users can delete individual expense entries.
* Users can delete their account, which removes associated data.
* Users can generate reports based on date ranges and categories.

**Non-Functional Requirements**

* Security: Passwords are securely validated; user authentication is enforced.
* Performance: All operations should execute within <1 second for optimal user experience.
* Modularity: Code is structured across clear layers (entity, dao, util, main) for maintainability.
* Exception Handling: Custom exceptions handle invalid input (e.g., missing users or expenses).
* Testing: Critical operations are verified through unit tests using Python's unittest module.

**5. Database Schema**

1.Users Table

|  |  |  |
| --- | --- | --- |
| Column | Type | Description |
| user\_id | INT (PK) | Unique user identifier |
| username | VARCHAR | User’s login name |
| password | VARCHAR | User’s password |
| email | VARCHAR | Email address |

2. Expense Table

|  |  |  |
| --- | --- | --- |
| Column | Type | Description |
| expense\_id | INT (PK) | Unique expense identifier |
| user\_id | INT (FK) | References Users(user\_id) |
| category\_id | INT (FK) | References ExpenseCategories(category\_id) |
| amount | FLOAT | Expense amount |
| date | DATE | Date of the expense |
| description | VARCHAR | Optional description |

3. ExpenseCategories Table

|  |  |  |
| --- | --- | --- |
| Column | Type | Description |
| category\_id | INT (PK) | Unique category identifier |
| category\_name | VARCHAR | Name of the category |

**6. Python Package Structure**

* entity/: Contains model classes like User, Expense, and Category. These classes represent the database entities and include private attributes, constructors, and getter/setter methods.
* dao/: Includes the interface (IFinanceRepository) and its implementation (FinanceRepositoryImpl) for handling all database operations (CRUD logic).
* exception/: Holds custom exceptions such as UserNotFoundException and ExpenseNotFoundException, which are raised during invalid operations.
* util/: Provides utility functions for database connectivity. Includes DBPropertyUtil (reads config) and DBConnUtil (returns DB connection).
* main/: Contains the main driver program (FinanceApp.py) which offers a menu-driven console interface to interact with the system.
* test/: Includes unit test scripts using unittest (or pytest) to validate user creation, expense addition, and exception scenarios.
* config/: Stores the db.properties file which contains database credentials such as host, port, username, and password.

**7. Exception Handling**

UserNotFoundException - Raised when a user ID does not exist in the database.

ExpenseNotFoundException - Raised when an expense ID is invalid or missing

These exceptions are defined in the exception/ package, raised from the DAO layer, and caught in the main driver program to ensure smooth, user-friendly error handling.

**8. Unit Testing**

* Test User Creation  
  Verifies that a user can be created successfully and inserted into the database.
* Test Expense Creation  
  Confirms that an expense record is added with valid details such as amount, date, and category.
* Test Expense Retrieval  
  Ensures all expenses linked to a particular user can be fetched from the database.
* Test UserNotFoundException  
  Checks that the system raises a UserNotFoundException when operations are attempted on a non-existent user ID.
* Test ExpenseNotFoundException  
  Validates that an ExpenseNotFoundException is triggered when trying to delete or retrieve an invalid expense ID.