1. Overview of the Application

The application consists of:

- **Database Class:** Manages user record insertion, updating, retrieval, and deletion actions in addition to the SQLite database connection.
- **UserService Class:** A service layer that offers an intuitive user interface for interacting with user data while abstracting database processes.
- **Main Program (main.py):** Manages user input and offers an interactive user interface for adding, editing, removing, and retrieving database users.

2. Code Implementation

2.1 Database Class (Database in user.py)

The Database class is responsible for interact with the SQLite database. The methods are as follows:

- **insert_user(name, age)**: Inserts a new user into the database.
- **get_user(user_id)**: Get a user from the database by ID.
- **update_user(user_id, name, age)**: Updates the user's name and age based on the provided user ID.
- **delete_user(user_id)**: Deletes the user with the mentioned ID from the database.

```python
import sqlite3

class User:
    def __init__(self, user_id, name, age):
        self.user_id = user_id
        self.name = name
        self.age = age

class Database:
    def __init__(self, db_name):
        self.conn = sqlite3.connect(db_name)
        self.cursor = self.conn.cursor()
        self.cursor.execute('''
            CREATE TABLE IF NOT EXISTS users
            (user_id INTEGER PRIMARY KEY, name TEXT, age INTEGER)
        ''')

    def insert_user(self, name, age):
        try:
            self.cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)', (name, age))
            self.conn.commit()
            return self.cursor.lastrowid
        except sqlite3.Error as e:
            print(f"Error inserting user: {e}")
            return None

    def get_user(self, user_id):
        try:
            self.cursor.execute('SELECT * FROM users WHERE user_id = ?', (user_id,))
```

```
Wrote HTML report to htmlcov\index.html
PS C:\Users\tilav\Desktop\Conestoga\Cloud\Code Coverage and Quality\Assignement 1\PROG8840\lab03>
```

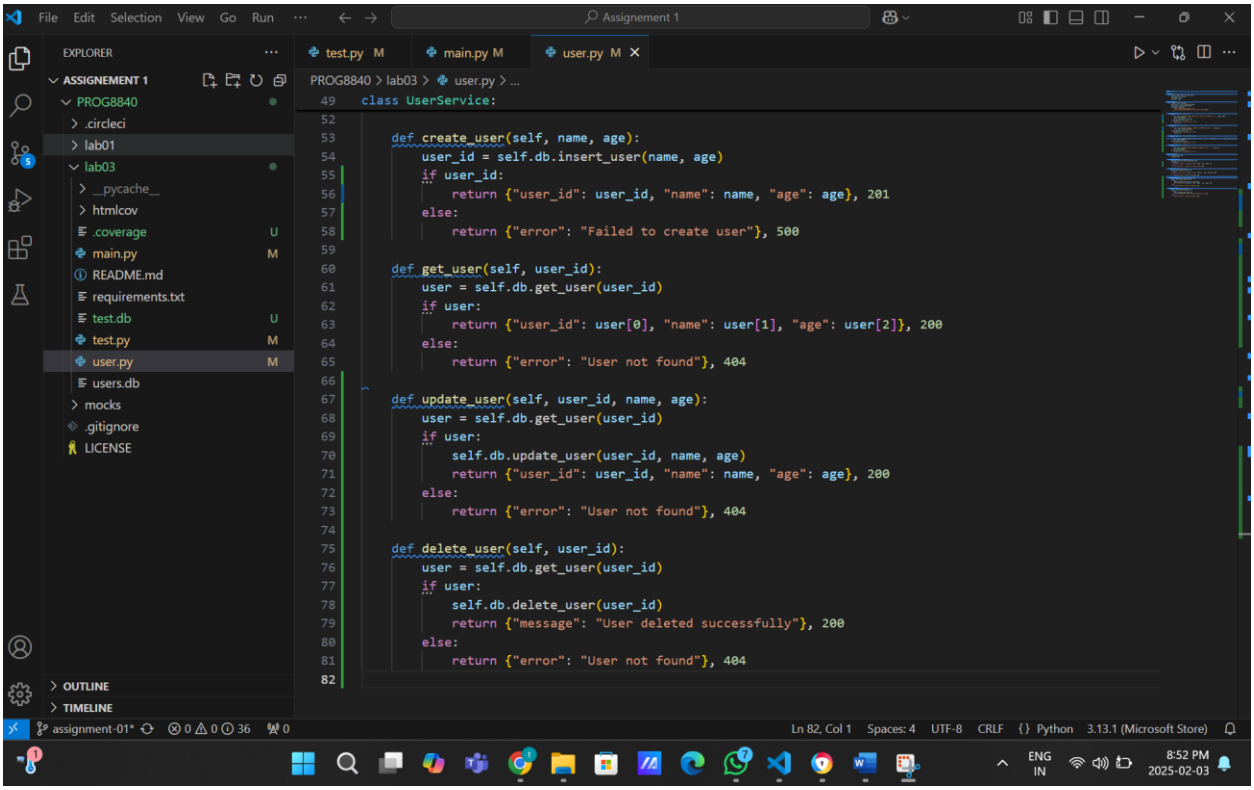
```python
        else:
            return {"error": "User not found"}, 404

    def update_user(self, user_id, name, age):
        user = self.db.get_user(user_id)
        if user:
            self.db.update_user(user_id, name, age)
            return {"user_id": user_id, "name": name, "age": age}, 200
        else:
            return {"error": "User not found"}, 404

    def delete_user(self, user_id):
        user = self.db.get_user(user_id)
        if user:
            self.db.delete_user(user_id)
            return {"message": "User deleted successfully"}, 200
        else:
            return {"error": "User not found"}, 404
```

2.2 UserService Class (UserService in user.py)

The UserService class abstracts the database operations and provides a high level interface for managing users. This class include the following methods:

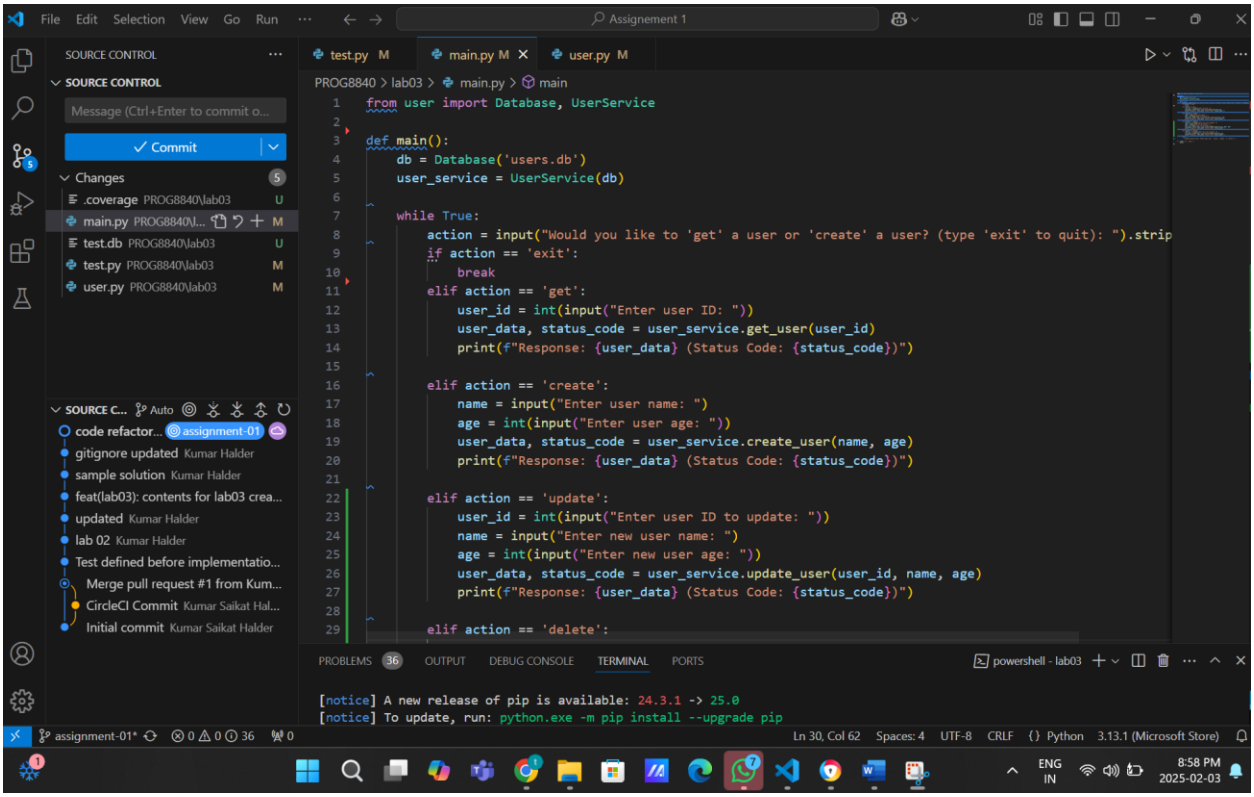
- **create_user(name, age)**: Calls the database's insert_user method and returns a success response.
- **get_user(user_id)**: Calls the database's get_user method and returns user data if get.
- **update_user(user_id, name, age)**: Calls the database update_user method if is there any user exists.

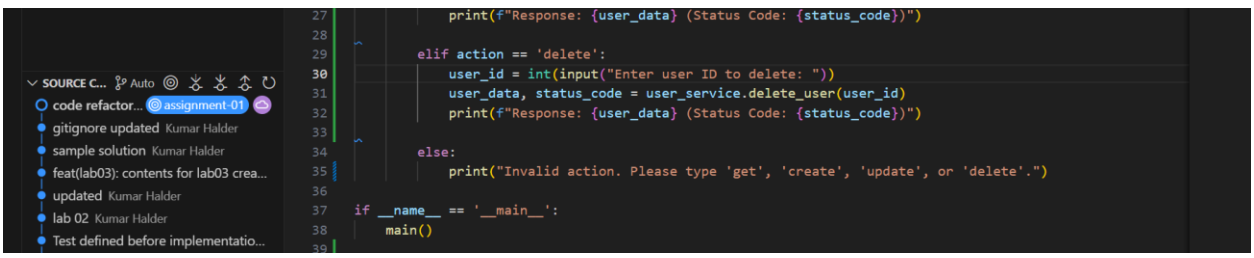
- **delete_user(user_id)**: Calls the database's delete_user method if the user exists.



2.3 Main Program (main.py)

The application's entry point is the main.py file. The user can create, retrieve, update, and delete users using its interactive interface.



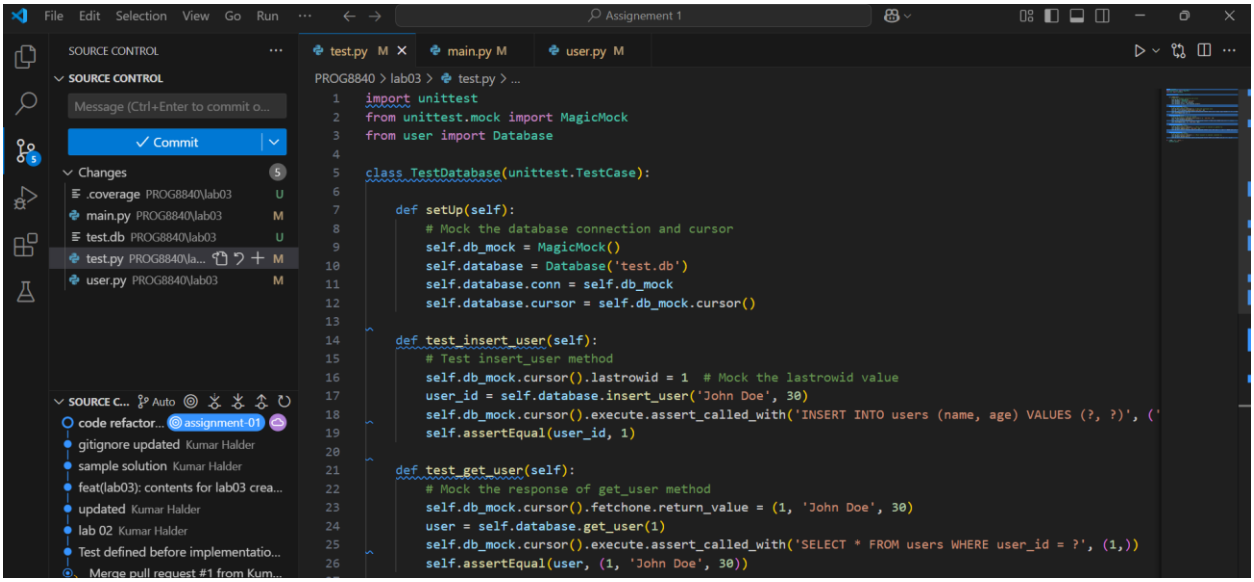


3. Unit Testing

Unit tests were written for the Database class using the unittest framework along with unittest.mock for mocking the database interactions. The tests cover the following database operations:

- **Inserting a user**
- **Retrieving a user**
- **Updating a user**
- **Deleting a user**

Here is the unit test code (test.py):



```python
import unittest
from unittest.mock import MagicMock
from user import Database

class TestDatabase(unittest.TestCase):

    def setUp(self):
        # Mock the database connection and cursor
        self.db_mock = MagicMock()
        self.database = Database('test.db')
        self.database.conn = self.db_mock
        self.database.cursor = self.db_mock.cursor()

    def test_insert_user(self):
        # Test insert_user method
        self.db_mock.cursor().lastrowid = 1  # Mock the lastrowid value
        user_id = self.database.insert_user('John Doe', 30)
        self.db_mock.cursor().execute.assert_called_with('INSERT INTO users (name, age) VALUES (?, ?)', ('
        self.assertEqual(user_id, 1)

    def test_get_user(self):
        # Mock the response of get_user method
        self.db_mock.cursor().fetchone.return_value = (1, 'John Doe', 30)
        user = self.database.get_user(1)
        self.db_mock.cursor().execute.assert_called_with('SELECT * FROM users WHERE user_id = ?', (1,))
        self.assertEqual(user, (1, 'John Doe', 30))
```

```python
    def test_update_user(self):
        # Test update_user method
        self.db_mock.cursor().rowcount = 1  # Mock rowcount to simulate an updated row
        self.database.update_user(1, 'Jane Doe', 28)
        self.db_mock.cursor().execute.assert_called_with('UPDATE users SET name = ?, age = ? WHERE user_id

    def test_delete_user(self):
        # Test delete_user method
        self.db_mock.cursor().rowcount = 1  # Mock rowcount to simulate a deleted row
        self.database.delete_user(1)
        self.db_mock.cursor().execute.assert_called_with('DELETE FROM users WHERE user_id = ?', (1,))

if __name__ == '__main__':
    unittest.main()
```

4. Coverage Report

To ensure comprehensive testing, a code coverage report was generated using coverage.py.

Steps:

1. **Run the tests**:

   coverage run -m unittest discover

2. **Generate the report**:

   coverage report -m

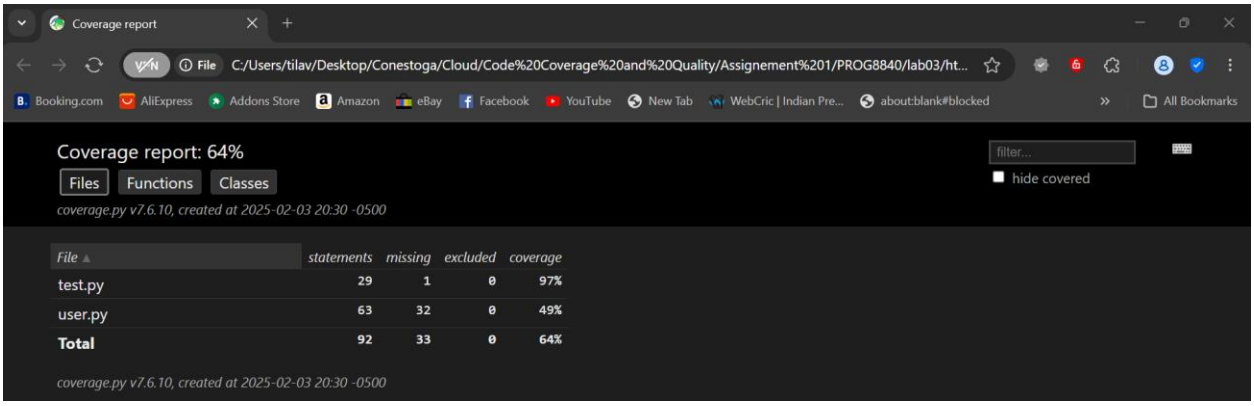
3. **Generate HTML report**:

   coverage html

This report provides detailed coverage information, showing that the tests cover almost all aspects of the Database class.

```
PROBLEMS 36   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS                                   powershell - lab03

PS C:\Users\tilav\Desktop\Conestoga\Cloud\Code Coverage and Quality\Assignement 1\PROG8840\lab03> pip install coverage
Collecting coverage
  Downloading coverage-7.6.10-cp312-cp312-win_amd64.whl.metadata (8.4 kB)
Downloading coverage-7.6.10-cp312-cp312-win_amd64.whl (211 kB)
Installing collected packages: coverage
Successfully installed coverage-7.6.10

[notice] A new release of pip is available: 24.3.1 -> 25.0
[notice] To update, run: python.exe -m pip install --upgrade pip
PS C:\Users\tilav\Desktop\Conestoga\Cloud\Code Coverage and Quality\Assignement 1\PROG8840\lab03> coverage run -m unittest disco
[notice] A new release of pip is available: 24.3.1 -> 25.0
[notice] To update, run: python.exe -m pip install --upgrade pip
PS C:\Users\tilav\Desktop\Conestoga\Cloud\Code Coverage and Quality\Assignement 1\PROG8840\lab03> coverage run -m unittest disco
ver
....
----------------------------------------------------------------------
Ran 4 tests in 0.058s

OK
PS C:\Users\tilav\Desktop\Conestoga\Cloud\Code Coverage and Quality\Assignement 1\PROG8840\lab03> coverage report -m
>>
Name      Stmts   Miss  Cover   Missing
-------------------------------------
[notice] A new release of pip is available: 24.3.1 -> 25.0
[notice] To update, run: python.exe -m pip install --upgrade pip
PS C:\Users\tilav\Desktop\Conestoga\Cloud\Code Coverage and Quality\Assignement 1\PROG8840\lab03> coverage run -m unittest disco
ver
....
----------------------------------------------------------------------
Ran 4 tests in 0.058s
```

Code coverage report:



---

5. Code Improvements

- **Error Handling**: Improved error handling in database operations. If there's an error during a query (insert, update, select, delete), the system will print the error message and return None or an appropriate response.
- **Security**: Used parameterized queries to prevent SQL injection.
- **Code Refactoring**: The UserService class was created to abstract the database operations, improving maintainability and separation of concerns.

6.Conclusion: Using SQLite and Python, this assignment activity entailed creating a small user management system, writing unit tests to confirm that database operations were correct, and creating a code coverage report to guarantee thorough test coverage. The code was also restructured to enhance error handling and maintainability.