Detailed Report on API Testing Code

Vaibhav Mittal(IMT2022126)
Valmik Belgaonkar(IMT2022020)
Krish Dave(IMT2022043)
Aaditya Joshi(IMT2022092)

1. Framework and Tools Used

APITestCase

- A subclass of `unittest.TestCase`, provided by Django REST Framework (DRF) to test API endpoints.
- Sets up a test client (`APIClient`) to simulate HTTP requests and responses.
- Ensures each test runs in isolation with an independent database.

APIClient

- A client that sends requests to API endpoints and retrieves responses.
- Supports HTTP methods like GET, POST, PUT, PATCH, and DELETE.

Database Setup

- The database is reset after each test case execution.
- Sample data is created in the `setUp()` method for testing purposes.

2. Explanation of Test Cases

a. `PatientsAPITestCase`

This test case validates the functionality of patient-related endpoints.

Setup:

- A `Patient` object is created with attributes such as `email`, `username`, `blood group`, and `disease`.
- The `set_password()` method ensures the password is hashed before saving.

Tests:

- `test_get_all_patients`: Simulates a GET request to retrieve all patients and validates the response.
- `test_get_patient_by_id`: Fetches a specific patient by their ID and checks attributes.
- `test_get_patient_not_found`: Simulates a GET request with a non-existent ID and expects an error message.
- `test_get_patient_wrong_type`: Tests querying a `Doctor` object instead of `Patient`, expecting an error.

b. `DoctorsAPITestCase`

This test case validates doctor-related endpoints.

Setup:

- A `Doctor` object is created with attributes like `specialization`, `availability`, and `about`.

Tests:

- `test_get_doctor_by_id`: Fetches a doctor by ID and verifies their details.
- `test_get_doctors_by_specialization`: Retrieves doctors by specialization and ensures the correct subset is returned.
- `test_get_all_doctors`: Fetches all doctors and validates the total count.
- `test_post_doctor`: Creates a new doctor with a POST request.
- `test_put_doctor`: Updates an existing doctor's details using a PUT request.
- `test_patch_doctor`: Partially updates a doctor's availability with a PATCH request.
- `test_delete_doctor`: Deletes a doctor using a DELETE request.
- `test_get_doctors_with_no_free_specialists`: Tests when no free specialists are available for a given specialization.

c. `NewUsersAPITestCase`

This test case validates general user-related operations.

Setup:

- Two `NewUser` objects are created for testing.

Tests:

- `test_get_user_by_id`: Retrieves a user by ID.
- `test_get_nonexistent_user_by_id`: Tests querying a non-existent user.

- `test_get_all_users`: Fetches all users and validates the total count.
- `test_post_new_user`: Tests the creation of a new user.
- `test_put_update_user`: Updates user details with a PUT request.
- `test_patch_update_user`: Partially updates a user's email using PATCH.
- `test_delete_user`: Deletes a user.
- `test_delete_nonexistent_user`: Tests deleting a non-existent user.

d. `IntermediateAPITestCase`

This test case manages intermediate users (e.g., admins, coordinators).

Setup:

- A sample intermediate user is created.
- `valid_data` and `invalid_data` dictionaries are prepared.

Tests:

- `test_get_all_intermediates`: Retrieves all intermediate users.
- `test_get_single_intermediate_valid_id`: Fetches an intermediate user by a valid ID.
- `test_get_single_intermediate_invalid_id`: Tests querying a non-existent intermediate user.
- `test_post_valid_intermediate`: Creates a new intermediate user.
- `test_post_invalid_intermediate`: Ensures validation errors for invalid data.
- `test_patch_intermediate`: Partially updates an intermediate user's `about` field.
- `test_delete_intermediate`: Deletes an intermediate user.
- `test_delete_invalid_intermediate`: Tests deleting a non-existent intermediate user.

e. `AppointmentAPITestCase`

This test case focuses on appointment management endpoints.

Setup:

- Two `Appointment` objects are created.

Tests:

- `test_get_all_appointments`: Fetches all appointments.
- `test_get_filtered_appointments`: Filters appointments by `meeting_Type`.
- `test_create_valid_appointment`: Creates a new appointment with valid data.
- `test_update_appointment`: Updates an appointment's `disease` field.
- `test_delete_appointment`: Deletes an appointment.

3. Key Features Tested

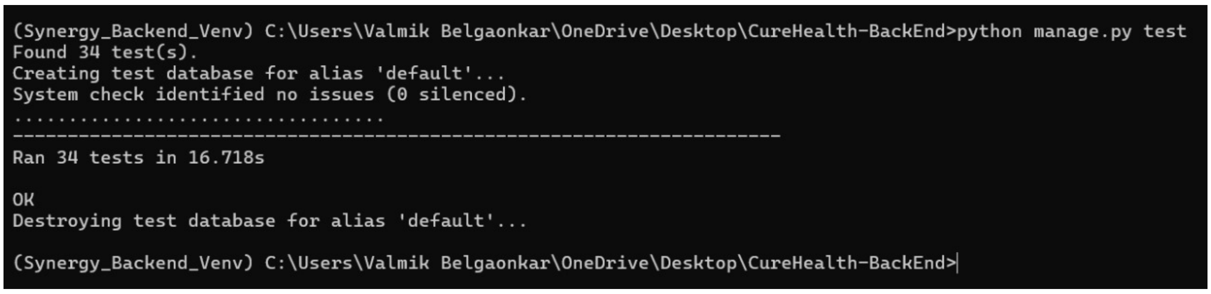
- CRUD operations.
- Validation for invalid data or non-existent IDs.
- Filtering results by attributes.
- Secure password hashing.

4. Improvements Suggested

- Uncomment and refine commented tests.
- Add assertions for response status codes.
- Enhance validation tests for a broader range of inputs.
- Include tests for user permissions and authentication.

5. Conclusion

The test suite is robust, leveraging DRF's `APITestCase` for comprehensive API endpoint coverage. Each test ensures correctness, reliability, and resilience of the backend.



```
(Synergy_Backend_Venv) C:\Users\Valmik Belgaonkar\OneDrive\Desktop\CureHealth-BackEnd>python manage.py test
Found 34 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..................................
----------------------------------------------------------------------
Ran 34 tests in 16.718s

OK
Destroying test database for alias 'default'...

(Synergy_Backend_Venv) C:\Users\Valmik Belgaonkar\OneDrive\Desktop\CureHealth-BackEnd>
```

Figure 1: . => SUCCESS, F => FAILURE, E => ERROR

6. Github Repositories

Frontend `https://github.com/KrishDave1/CureHealth-FrontEnd.git`
Backend `https://github.com/valmikGit/CureHealth-BackEnd.git`
NOTE: Please Refer to Readme files of both repositories for more features/ functionality of our project.