Backend Testing

Introduction

At the heart of our commitment to code quality and reliability lies an integrated CI/CD pipeline, bolstering the efficiencies and accuracies of our development cycles. Leveraging the power of Continuous Integration (CI) and Continuous Deployment (CD), we have seamlessly woven automation into our workflow, ensuring that every code modification is automatically built, tested, and deployed through a standardized and reliable pipeline. More details about CI/CD: Backend Deployment | Continuous Integration (CI) with GitHub Actions

In our pursuit of maintaining code quality and reliability, we utilize Django's default testing framework for backend. Each distinct app within our project, such as 'User' and 'Data', has its dedicated `tests.py` file. This organizational choice ensures a clear separation and focused testing for each app's functionalities. By placing test cases specific to an app within its respective `tests.py`, we ensure modular, maintainable, and organized test management. This approach not only guarantees compatibility and ease within our Django-centric development environment but also leverages Django's built-in tools and utilities for efficient test execution.

API Testing Documentation

Introduction to API Testing Documentation

In the dynamic landscape of software development, ensuring the reliability and security of our systems is paramount. This document delineates the rigorous testing processes we have instituted for our API. From functional checks to security verifications, each test case is meticulously described, underscoring our commitment to delivering a robust and dependable API. The methodologies captured within cater to various facets of testing:

1. **Functional Testing**: Validates the system's actions under given conditions.
2. **Regression Testing**: Ensures bug fixes or new features haven't impacted existing functionalities.
3. **Security Testing**: Verifies the system's ability to guard against unauthorized access or malicious activities.
4. **Automation Testing**: Demonstrates our drive towards efficiency by automating repetitive tests.
5. **Additional Negative Testing**: Tests the system's resilience against adverse conditions or unexpected user behavior.

By adhering to the standards presented in this document, we aim to uphold the highest quality, security, and reliability standards for our API.

1. Functional Testing for API

Test Scenario ID	View/Class	Description	Test Scenario	Expected Result	Response Status	Pass/Fail
TS_001	RegisterView	Ensure users register with correct details.	Register user account with valid data.	User is registered successfully.	201 CREATED	Pass
TS_002	RegisterView	Ensure users cannot register with incorrect details.	Register user account with invalid data.	User registration fails.	400 BAD REQUEST	Pass
TS_003	EditProfileView	Ensure users can edit profile with valid data.	Edit profile with valid data for authenticated users.	Profile is edited successfully.	200 OK	Pass
TS_004	EditProfileView	Ensure users cannot edit profile with invalid data.	Edit profile with invalid data for authenticated users.	Profile edit fails.	400 BAD REQUEST	Pass
TS_005	EditProfileView	Ensure unauthenticated users cannot access the edit profile endpoint.	Edit profile with valid data whilst unauthenticated.	Access denied.	401 UNAUTHORIZED	Pass
TS_006	ChangePasswordView	Ensure users who are authenticated can change their password.	Change password by inputting new then old password.	Password changed successfully.	200 OK	Pass
TS_007	ChangePasswordView	Ensure users cannot change password with incorrect old password.	Change password by inputting new then incorrect old password.	Password change fails.	400 BAD REQUEST	Pass
TS_008	ChangePasswordView	Ensure users cannot change password if unauthenticated.	Change password whilst being unauthenticated.	Access denied.	401 UNAUTHORIZED	Pass

2. Regression Testing for API

- **Description**: After each bug fix or new feature development, run the entire suite of functional tests to ensure no existing functionality is broken.

3. Security Testing for API

Test Scenario ID	View/Class	Description	Test Scenario	Expected Result	Response Status	Pass/Fail
TS_009	SecurityTests	Ensures users cannot access an authenticated	Access an authenticated	Access denied.	401 UNAUTHORIZED	Pass

		endpoint without providing an authentication token.	endpoint without a token.			
TS_010	SecurityTests	Ensures users cannot access an authenticated endpoint with a fake token.	Access an authenticated endpoint with a fake token.	Access denied.	401 UNAUTHORIZED	Pass

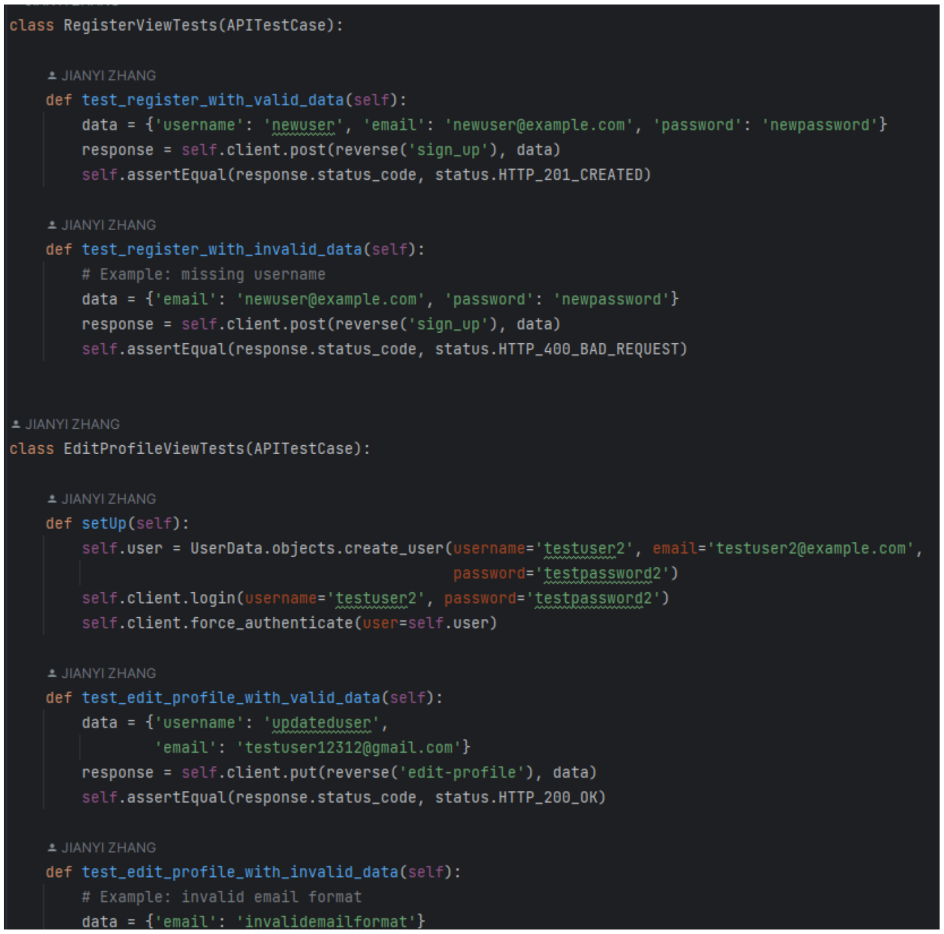
4. Automation Testing for API

- **Description**: All tests in the `tests.py` file are automated and can be run using Django's test runner. Regularly run these tests, especially after code changes.

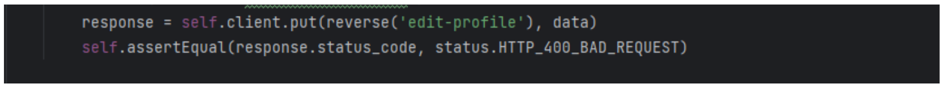
5. Additional Negative Testing for API

Test Scenario ID	View/Class	Description	Test Scenario	Expected Result	Response Status	Pass/Fail
TS_010	AdditionalNegativeTests	Ensures users cannot register with a username which exceeds the allowed character limit.	Register with a long username.	Registration fails due to long username.	400 BAD REQUEST	Pass
TS_011	AdditionalNegativeTests	Ensures users cannot change password with a password that is below the minimum character limit.	Change password with a very short new password.	Password change fails due to short new password.	400 BAD REQUEST	Pass

Example code



```
class RegisterViewTests(APITestCase):

    JIANYI ZHANG
    def test_register_with_valid_data(self):
        data = {'username': 'newuser', 'email': 'newuser@example.com', 'password': 'newpassword'}
        response = self.client.post(reverse('sign_up'), data)
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)

    JIANYI ZHANG
    def test_register_with_invalid_data(self):
        # Example: missing username
        data = {'email': 'newuser@example.com', 'password': 'newpassword'}
        response = self.client.post(reverse('sign_up'), data)
        self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)


JIANYI ZHANG
class EditProfileViewTests(APITestCase):

    JIANYI ZHANG
    def setUp(self):
        self.user = UserData.objects.create_user(username='testuser2', email='testuser2@example.com',
                                                 password='testpassword2')
        self.client.login(username='testuser2', password='testpassword2')
        self.client.force_authenticate(user=self.user)

    JIANYI ZHANG
    def test_edit_profile_with_valid_data(self):
        data = {'username': 'updateduser',
                'email': 'testuser12312@gmail.com'}
        response = self.client.put(reverse('edit-profile'), data)
        self.assertEqual(response.status_code, status.HTTP_200_OK)

    JIANYI ZHANG
    def test_edit_profile_with_invalid_data(self):
        # Example: invalid email format
        data = {'email': 'invalidemailformat'}
```


```
    response = self.client.put(reverse('edit-profile'), data)
    self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
```

Data Testing Documentation

Introduction

In the realm of precision and accuracy, the Data Testing segment stands as a sentinel, ensuring the reliability and integrity of our task data management functionalities. This documentation unfolds the systematic testing strategies employed to validate the functionalities and robustness of the data-related components, primarily revolving around models such as `TaskData` and `TaskMetaData`.

Objectives

Our objective is to ensure that every aspect of data handling, from creation to manipulation and retrieval, operates seamlessly and accurately. The tests are meticulously crafted to validate the integrity and reliability of the data models and their associated operations.

Testing Environment

The tests are executed in a controlled Django environment, leveraging Django's robust testing framework. This ensures consistency, isolation, and accurate assessment of each test case, facilitating a focused evaluation of each functional aspect.

Test Cases

While the details and complexities of each test case are meticulously outlined in the `tests.py` file, here's an overview of the categories of tests we have implemented:

- **Model Validations:** Tests that focus on validating the integrity and consistency of the `TaskData` and `TaskMetaData` models.

- **Data Creation and Retrieval:** Tests ensuring that data can be accurately created, saved, and retrieved from the database.
- **Data Manipulation:** Verifying the accuracy and reliability of data update and deletion operations.
- **Associative Operations:** Tests aimed at validating the associations and relationships between different data entities and models.

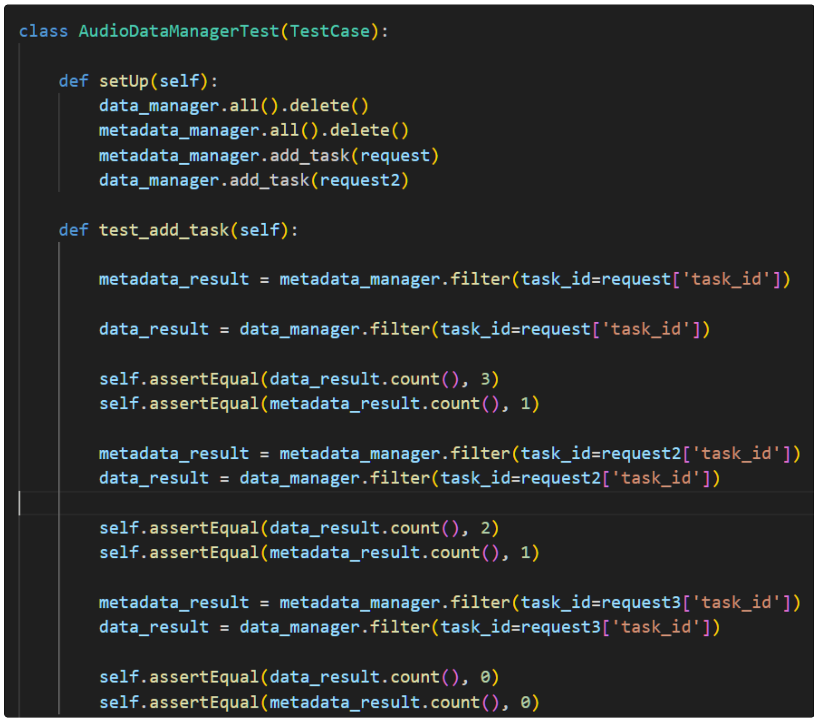
Execution and Automation

Each test is executed automatically, ensuring that all aspects of data functionality are continuously validated for accuracy and reliability. This automated execution fosters a continuous validation approach, ensuring immediate detection and resolution of issues.

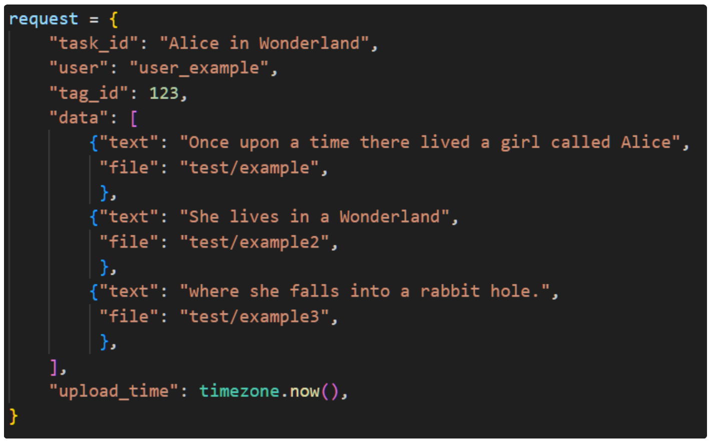
Each method within our models are accompanied with unit testing to ensure the purpose of each method functions correctly and as intended.

Unit Test ID	Function Name	Test Scenario	Expected Result	Pass/Fail
UT_001	add_task()	Adding a new task	Task is added successfully	Pass
UT_002	get_user_tasks()	Retrieving tasks assigned to a user	List of tasks assigned to the user	Pass
UT_003	delete_existing_audio_data()	Deleting a existing audio data from a task	Audio data is deleted from the specified task	Pass
UT_004	get_audio()	Retrieving audio data associated with a task	Audio data is successfully retrieved	Pass

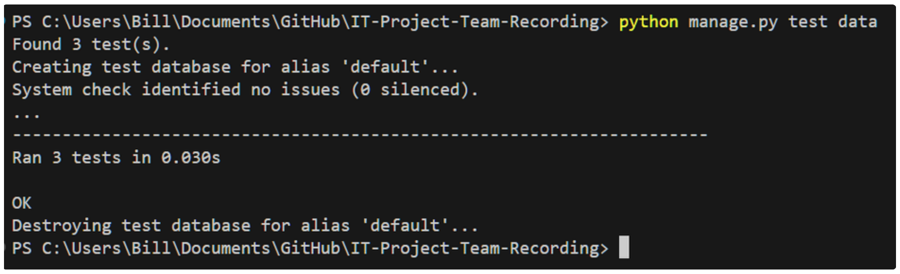
Example code



```python
class AudioDataManagerTest(TestCase):

    def setUp(self):
        data_manager.all().delete()
        metadata_manager.all().delete()
        metadata_manager.add_task(request)
        data_manager.add_task(request2)

    def test_add_task(self):

        metadata_result = metadata_manager.filter(task_id=request['task_id'])

        data_result = data_manager.filter(task_id=request['task_id'])

        self.assertEqual(data_result.count(), 3)
        self.assertEqual(metadata_result.count(), 1)

        metadata_result = metadata_manager.filter(task_id=request2['task_id'])
        data_result = data_manager.filter(task_id=request2['task_id'])

        self.assertEqual(data_result.count(), 2)
        self.assertEqual(metadata_result.count(), 1)

        metadata_result = metadata_manager.filter(task_id=request3['task_id'])
        data_result = data_manager.filter(task_id=request3['task_id'])

        self.assertEqual(data_result.count(), 0)
        self.assertEqual(metadata_result.count(), 0)
```



```
request = {
    "task_id": "Alice in Wonderland",
    "user": "user_example",
    "tag_id": 123,
    "data": [
        {"text": "Once upon a time there lived a girl called Alice",
         "file": "test/example",
         },
        {"text": "She lives in a Wonderland",
         "file": "test/example2",
         },
        {"text": "where she falls into a rabbit hole.",
         "file": "test/example3",
         },
    ],
    "upload_time": timezone.now(),
}
```



```
PS C:\Users\Bill\Documents\GitHub\IT-Project-Team-Recording> python manage.py test data
Found 3 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...
----------------------------------------------------------------------
Ran 3 tests in 0.030s

OK
Destroying test database for alias 'default'...
PS C:\Users\Bill\Documents\GitHub\IT-Project-Team-Recording>
```

Conclusion

Our Data Testing Documentation encapsulates the rigorous and systematic testing approaches employed to fortify our data handling functionalities against inconsistencies and errors. By adhering to these comprehensive testing strategies, we aim to uphold and ensure the utmost accuracy, reliability, and integrity of our data-related functionalities and features.