Project Documentation – Internship Project

This project is a Django-based application named **internship_project**, developed as part of an internship assignment. The aim of the project was to build a complete **User and Task Management System** with authentication, database handling, and testing, while following clean architecture and proper documentation practices.

The development process was divided into four main stages: **Setup & Environment, Core API Development, Database & ORM, and Testing & Documentation.**

Stage 1: Setup & Environment

At the beginning, a new Django project called **internship_project** was created. The development environment was set up with all the required dependencies and configurations. For the database, **PostgreSQL** was chosen as it is a powerful and widely used relational database system. Proper database connection details such as name, user, password, host, and port were configured in the project settings.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'internship_db',
        'USER': 'postgres',
        'PASSWORD': 'Sam@123',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

As an alternative, SQLite could have been used, but PostgreSQL was selected for better scalability and industry relevance.

Once the initial setup was completed, the project was connected to a **public GitHub repository** to maintain version control and ensure easy access for review. This also demonstrated good development practices of maintaining source code in a central repository.

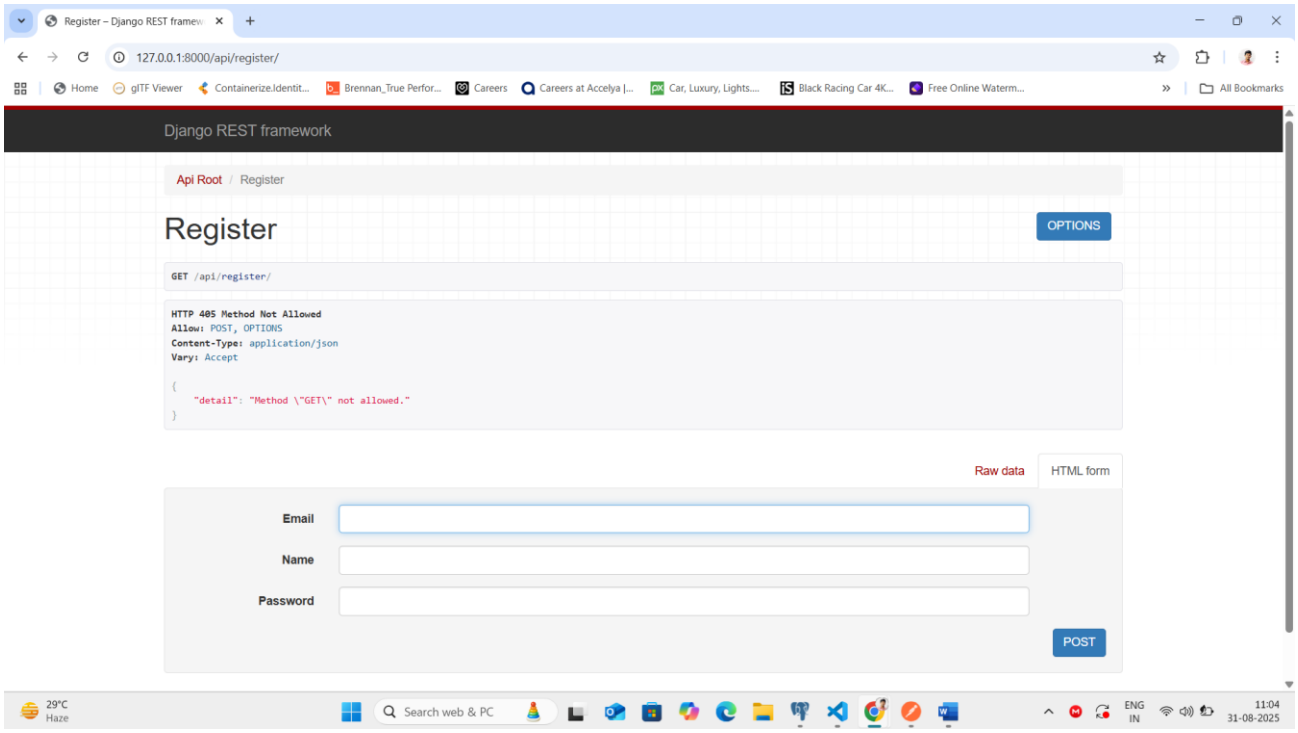
Stage 2: Core Task – API Development

The second stage focused on building the **core functionality of the application** using Django REST Framework.

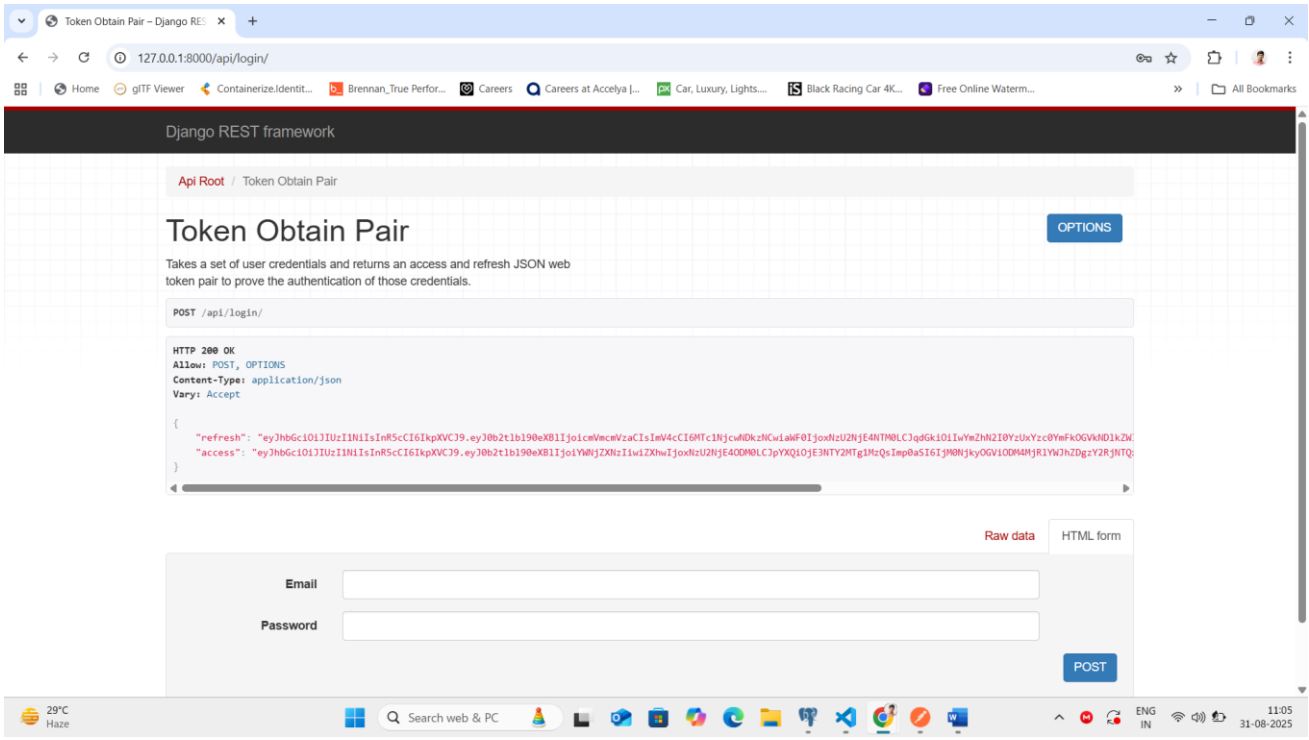
As part of the development, a **User Management API** was created by first setting up a dedicated user app within the Django project. The API was designed to handle the essential features of user authentication and profile management.

The following endpoints were developed and exposed through the REST API:

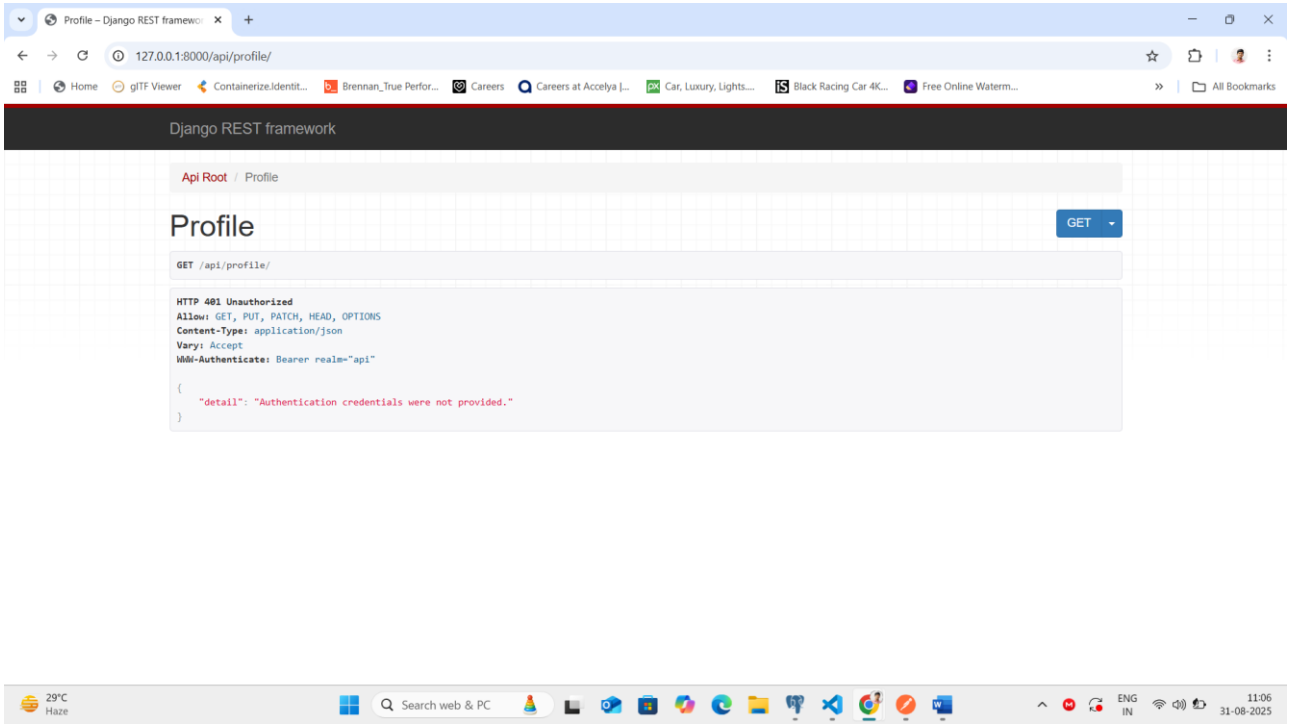
1. **User Registration** (http://127.0.0.1:8000/api/register/)
    - This endpoint allows new users to sign up by providing their **name, email, and password**.
    - Passwords are stored securely in the database using **hashing techniques** to ensure data safety.

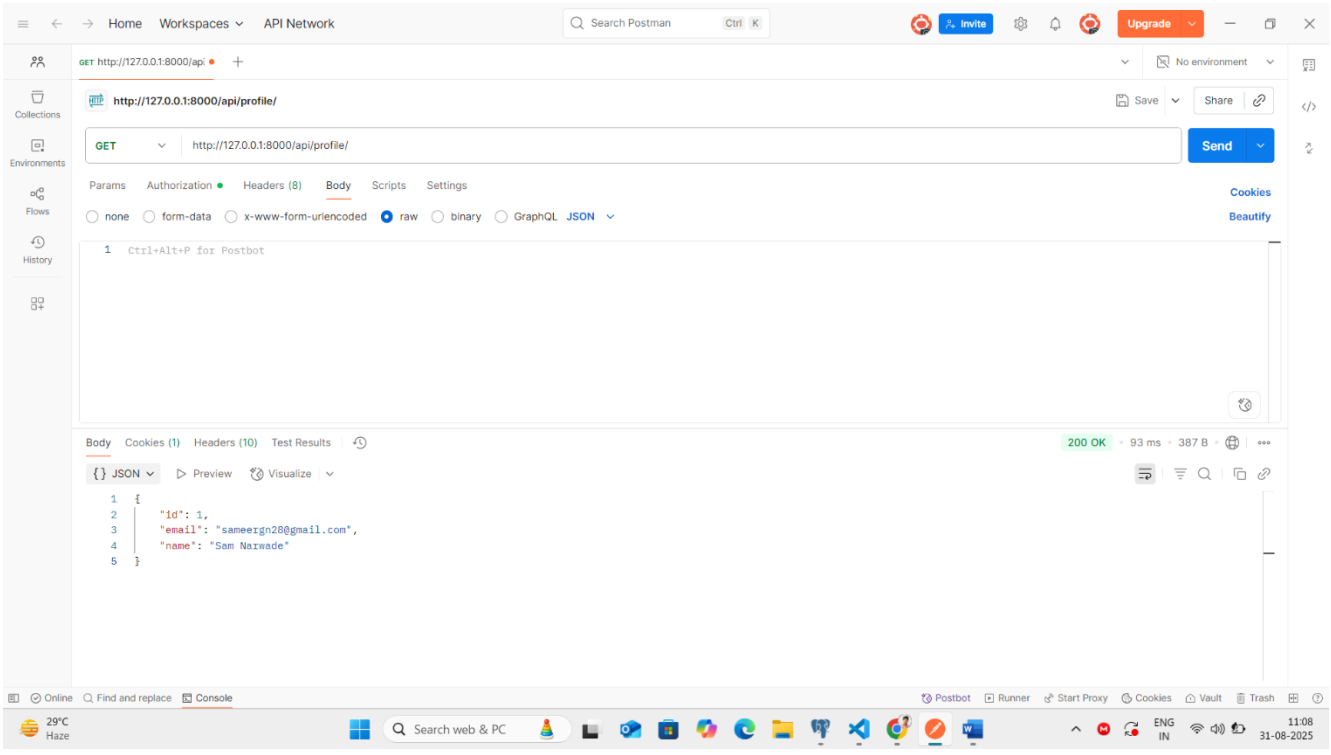


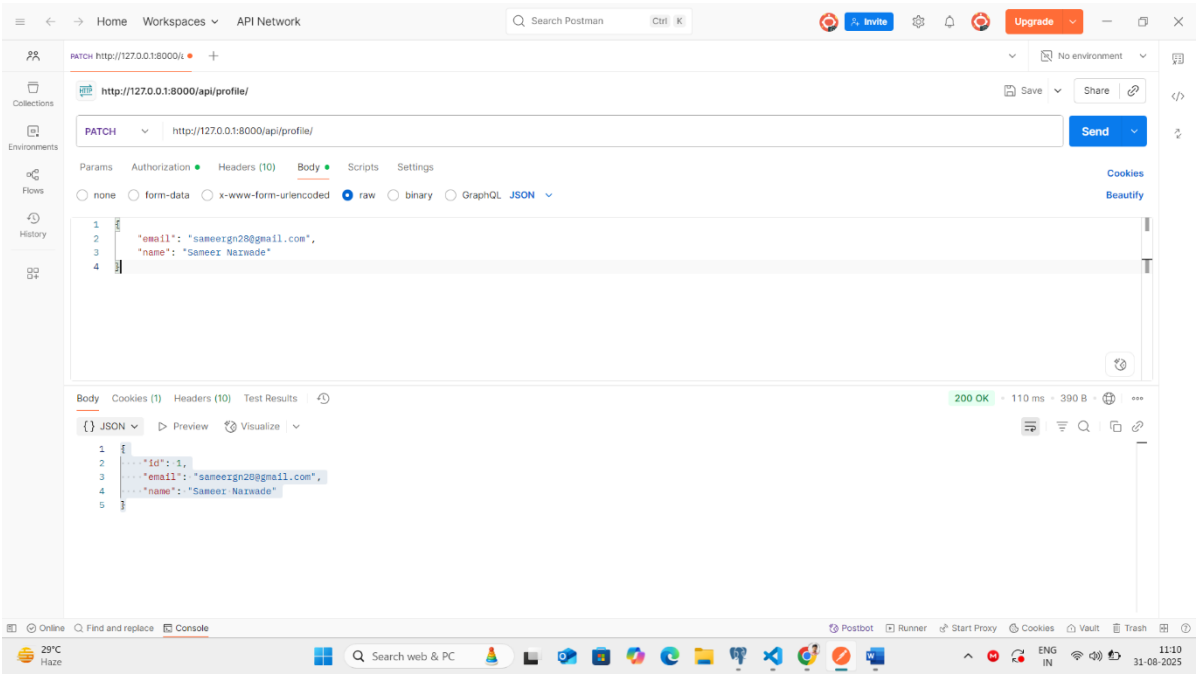
2. **User Login** (http://127.0.0.1:8000/api/login/)
    - A **JWT-based authentication system** was implemented for login.
    - Once a user provides valid login credentials, they receive a **JWT token**.
    - This token is required for all further authenticated requests to ensure secure access.



3. **User Profile** (http://127.0.0.1:8000/api/profile/)
    - This endpoint allows authenticated users to **view and update their profile details**.
    - Since this endpoint is protected, it cannot be directly tested in the browser. Instead, testing was carried out using **Postman**.
    - To test: first log in to obtain the **access token**, then paste this token under **Authorization → Bearer Token** in Postman. Once authenticated, users can successfully retrieve and update their profile data.


Profile – Django REST framework
127.0.0.1:8000/api/profile/
Django REST framework
Api Root / Profile
Profile
GET
GET /api/profile/
HTTP 401 Unauthorized
Allow: GET, PUT, PATCH, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
WWW-Authenticate: Bearer realm="api"
{
"detail": "Authentication credentials were not provided."
}


Home
Workspaces
API Network
Search Postman
Invite
Upgrade
http://127.0.0.1:8000/api/profile/
Save
Share
GET
http://127.0.0.1:8000/api/profile/
Send
Params
Authorization
Headers (8)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
Ctrl+Alt+P for Postbot
Body
Cookies (1)
Headers (10)
Test Results
200 OK
93 ms
387 B
JSON
Preview
Visualize
{
"id": 1,
"email": "sameergn28@gmail.com",
"name": "Sam Narwade"
}
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash



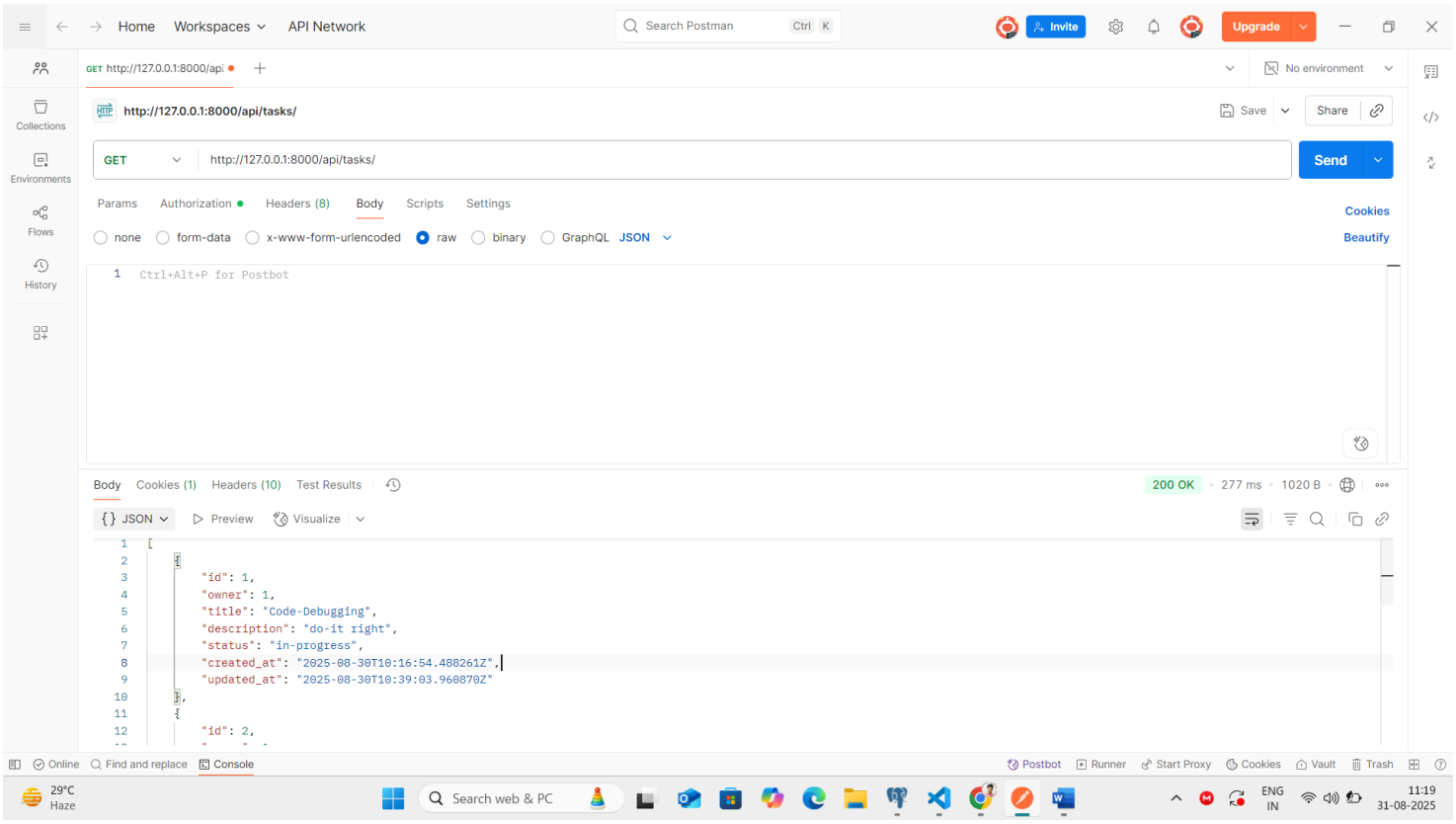
This API ensures secure user management by combining **hashed password storage**, **JWT authentication**, and **token-based access control**.

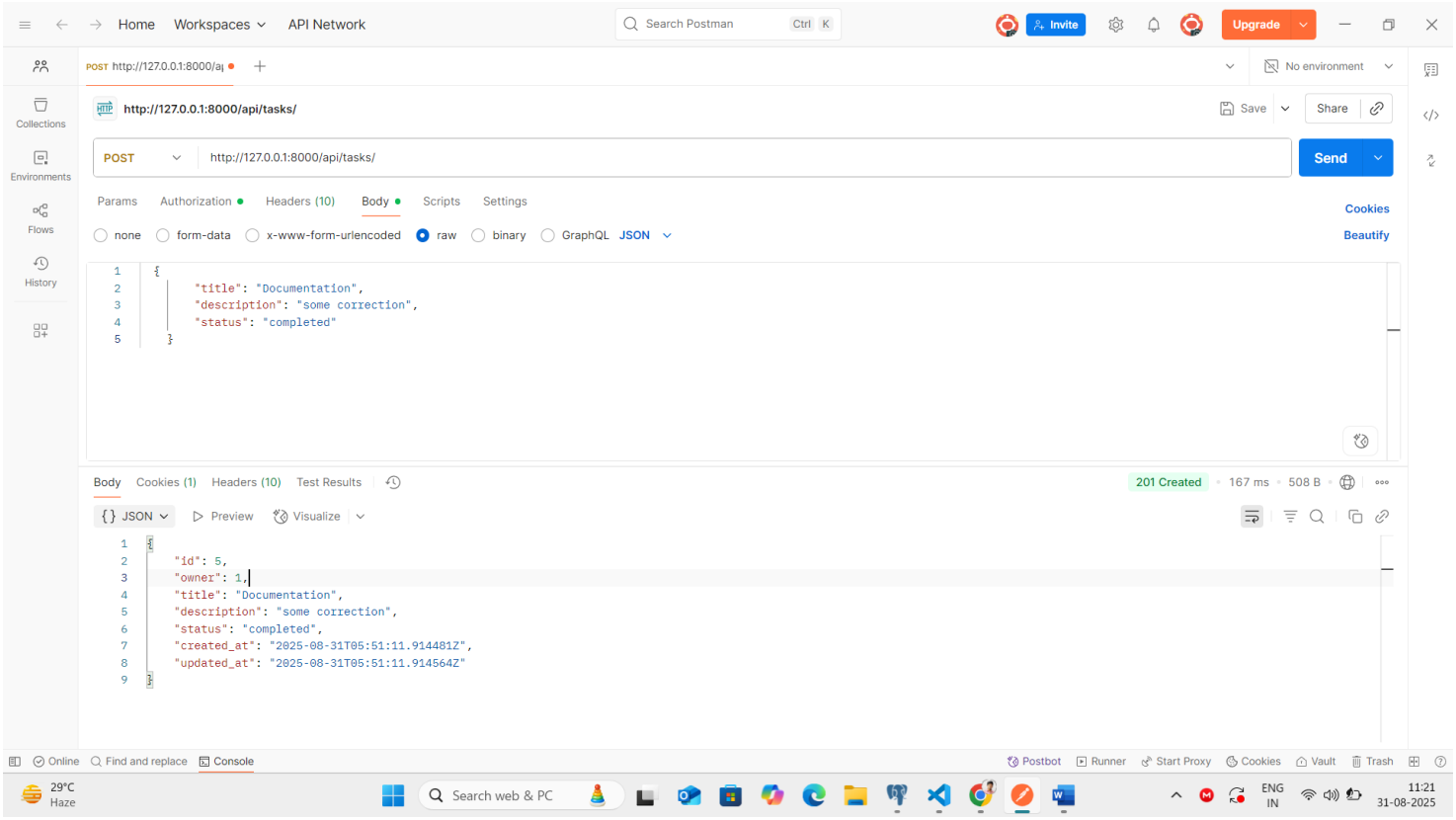
Next a dedicated **tasks app** was created within the Django project to handle task-related operations. This module was fully integrated with the authentication system to ensure that only logged-in users could access it.

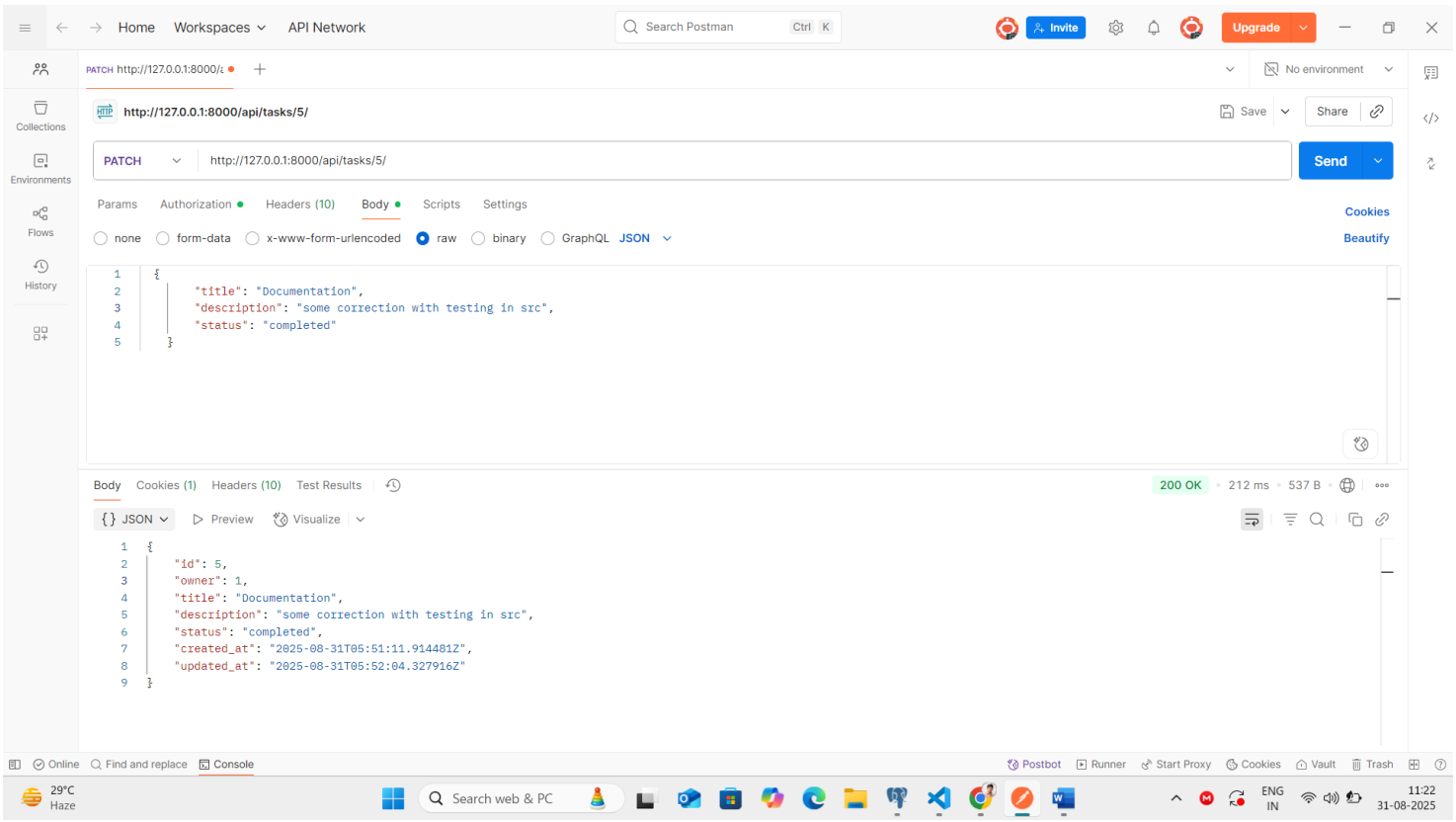
The Task Management Module included the following functionalities, all accessible through the endpoint:
http://127.0.0.1:8000/api/tasks/

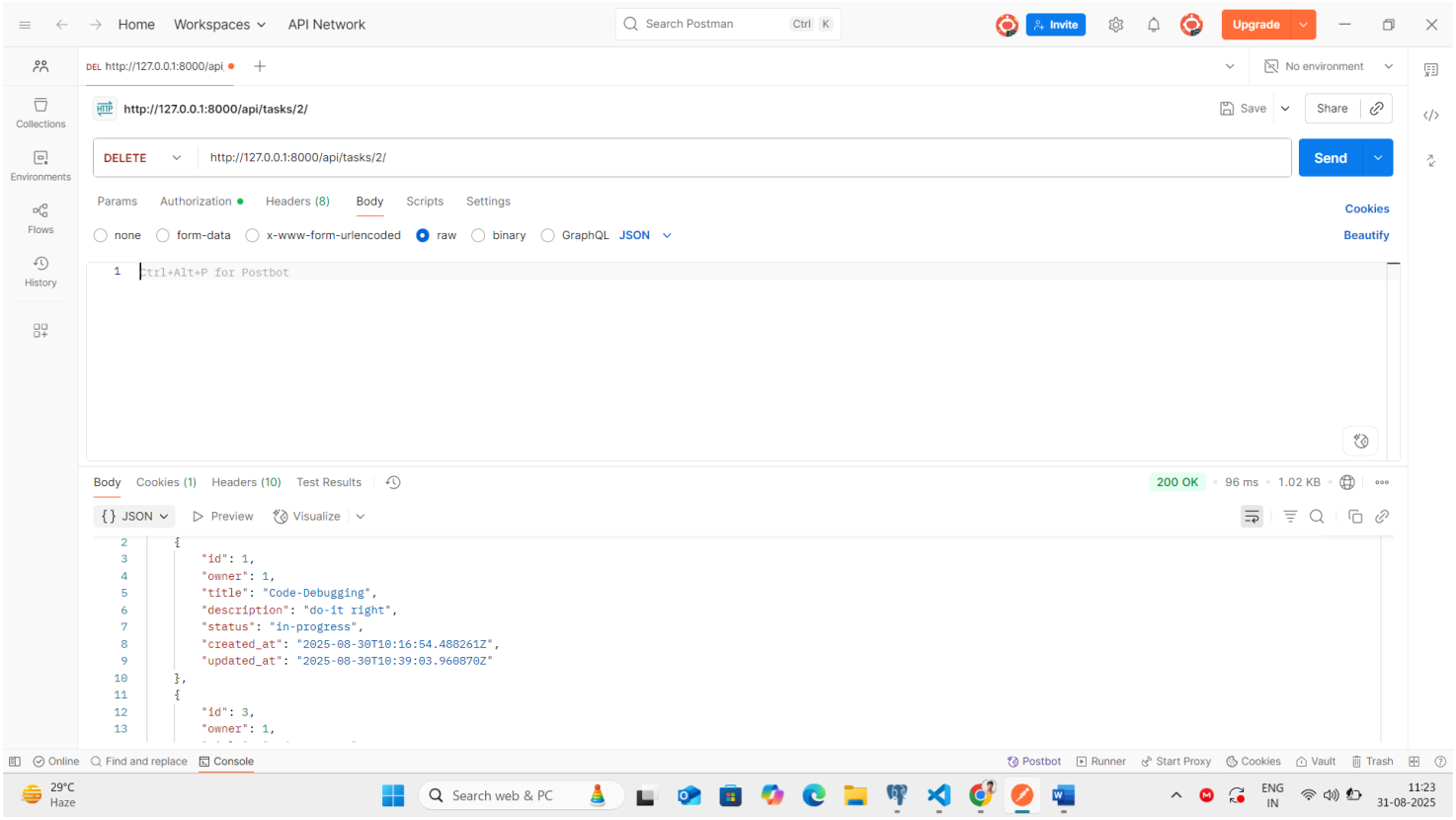
1. **Authentication Required**
    - To interact with this module, a user must first **log in** and obtain a valid **JWT access token**.
    - Without authentication, the API does not allow any task operations.
2. **Create Task**
    - An authenticated user can create a new task by sending details such as **title, description, and status**.
    - Each task is automatically linked to the user (as the **owner**).
3. **List Tasks**
    - A user can view only the tasks they have created.
    - Each task displays information like **title, description, status, created date, and updated date**.
    - No user can see another user's tasks.

4. **Update Task**
    - Users can update their own tasks (e.g., change the **title, description, or status**).
    - The **status field** allows marking tasks as *pending*, *in-progress*, or *completed*.
5. **Delete Task**
    - A user can delete their own tasks, but not tasks belonging to others.
6. **Ownership & Access Control**
    - Strict **ownership checks** were implemented.
    - Only the **creator of a task** can view, update, or delete it.
    - This ensures **data privacy** and prevents unauthorized access.
7. **Testing with Postman**
    - Since the API requires authentication, testing cannot be done directly in the browser.
    - Instead, **Postman** was used:
        - First, log in to get the **access token**.
        - Then, in Postman, add the token in **Authorization → Bearer Token**.
        - With this, the user can successfully **create, update, delete, and list their own tasks**.




Home
Workspaces
API Network
Search Postman
Ctrl K
Invite
Upgrade
No environment
http://127.0.0.1:8000/api/tasks/
Save
Share
POST
http://127.0.0.1:8000/api/tasks/
Send
Params
Authorization
Headers (10)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"title": "Documentation",
"description": "some correction",
"status": "completed"
}
Body
Cookies (1)
Headers (10)
Test Results
201 Created
167 ms
508 B
JSON
Preview
Visualize
{
"id": 5,
"owner": 1,
"title": "Documentation",
"description": "some correction",
"status": "completed",
"created_at": "2025-08-31T05:51:11.914481Z",
"updated_at": "2025-08-31T05:51:11.914564Z"
}
Collections
Environments
Flows
History
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash
29°C
Haze
Search web & PC
ENG
IN
11:21
31-08-2025


Home
Workspaces
API Network
Search Postman
Ctrl K
Invite
Upgrade
No environment
http://127.0.0.1:8000/api/tasks/5/
Save
Share
PATCH
http://127.0.0.1:8000/api/tasks/5/
Send
Params
Authorization
Headers (10)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"title": "Documentation",
"description": "some correction with testing in src",
"status": "completed"
}
Body
Cookies (1)
Headers (10)
Test Results
200 OK
212 ms
537 B
JSON
Preview
Visualize
{
"id": 5,
"owner": 1,
"title": "Documentation",
"description": "some correction with testing in src",
"status": "completed",
"created_at": "2025-08-31T05:51:11.914481Z",
"updated_at": "2025-08-31T05:52:04.327916Z"
}
Collections
Environments
Flows
History
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash
29°C
Haze
Search web & PC
ENG
IN
11:22
31-08-2025



This module ensured that every registered user had a **personalized and secure task management system**, with **proper authentication and access control** in place.

Stage 3: Database & ORM

In this stage, the **database models** were carefully designed with proper fields and relationships to handle both user and task data.

1. **User Model (users app → models.py)**
    - The **User model** was implemented to store essential user details such as **name, email, and password**.
    - Passwords were stored securely using **hashing mechanisms** provided by Django.
    - This model served as the foundation for authentication and was linked to tasks through a relationship.

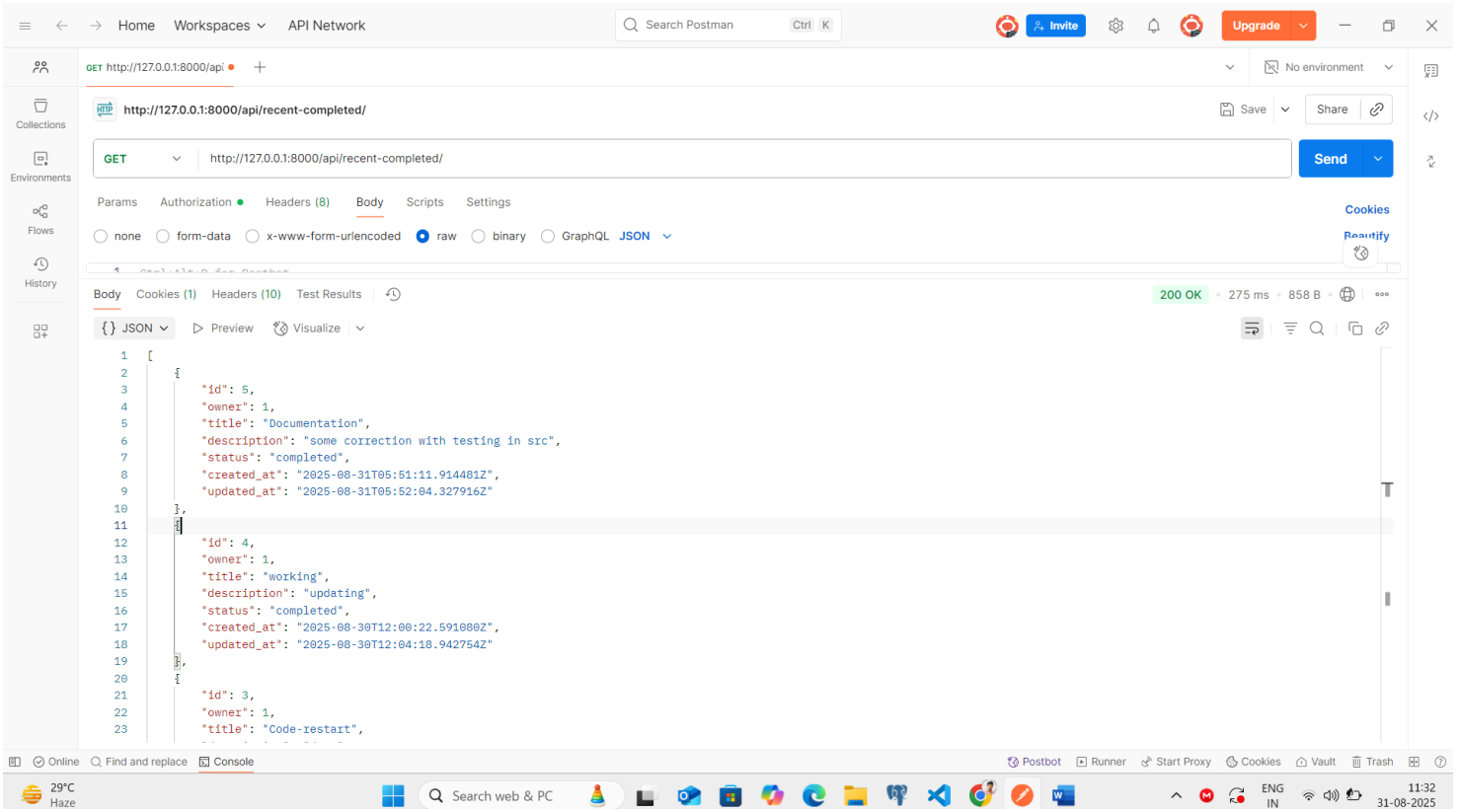
```python
from django.db import models
from django.contrib.auth.models import (
    AbstractBaseUser, BaseUserManager, PermissionsMixin
)

class UserManager(BaseUserManager):
    def create_user(self, email, name, password=None, **extra_fields):
        if not email:
            raise ValueError("Email required")
        email = self.normalize_email(email)
        user = self.model(email=email, name=name, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, name, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        return self.create_user(email, name, password, **extra_fields)

class User(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(unique=True)
    name = models.CharField(max_length=150)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    objects = UserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['name']

    def __str__(self):
        return self.email
```

2. **Task Model (tasks app → models.py)**
    - The **Task model** was implemented to store information about each task.
    - Key fields included:
        - **title** – short heading of the task
        - **description** – detailed information
        - **status** – pending, in-progress, or completed
        - **created date** – when the task was added
        - **updated date** – when the task was last modified
    - Each task was linked to its **owner (User)** using a **ForeignKey relationship**.

- This ensured that every user maintained their **own set of tasks** without interference from others.

```python
from django.conf import settings
from django.db import models

class Task(models.Model):
    STATUS_CHOICES = [
        ('pending', 'Pending'),
        ('in-progress', 'In Progress'),
        ('completed', 'Completed'),
    ]

    owner = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)
    status = models.CharField(max_length=20, choices=STATUS_CHOICES, default='pending')
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title
```

3. **Django ORM Usage**
   - Django's **Object Relational Mapper (ORM)** was used to interact with the database efficiently.
   - ORM queries handled all basic operations such as creating, updating, and deleting tasks, as well as filtering tasks by ownership.
4. **Custom Query – Recent Completed Tasks**
   - A **custom ORM query** was implemented to fetch all **completed tasks from the last seven days**.
   - This demonstrated how Django ORM can be extended for advanced use cases beyond basic CRUD operations.
   - To access this feature, a dedicated endpoint was created:
     - **http://127.0.0.1:8000/api/recent-completed/**
   - This endpoint required authentication and was tested using **Postman** by providing the JWT token under **Authorization → Bearer Token**.



By properly designing the models in the users and tasks apps, and by leveraging Django ORM with optimized queries, the project achieved **accuracy, performance, and secure data handling**.

Stage 4: Testing & Documentation

The final stage focused on **testing and documentation**. At least three unit tests were written for the main endpoints, such as user registration, login, and task creation. These tests were created using Django's test framework, ensuring that the APIs behaved correctly and consistently under different scenarios. The presence of tests improved the reliability of the application and gave confidence in future updates.

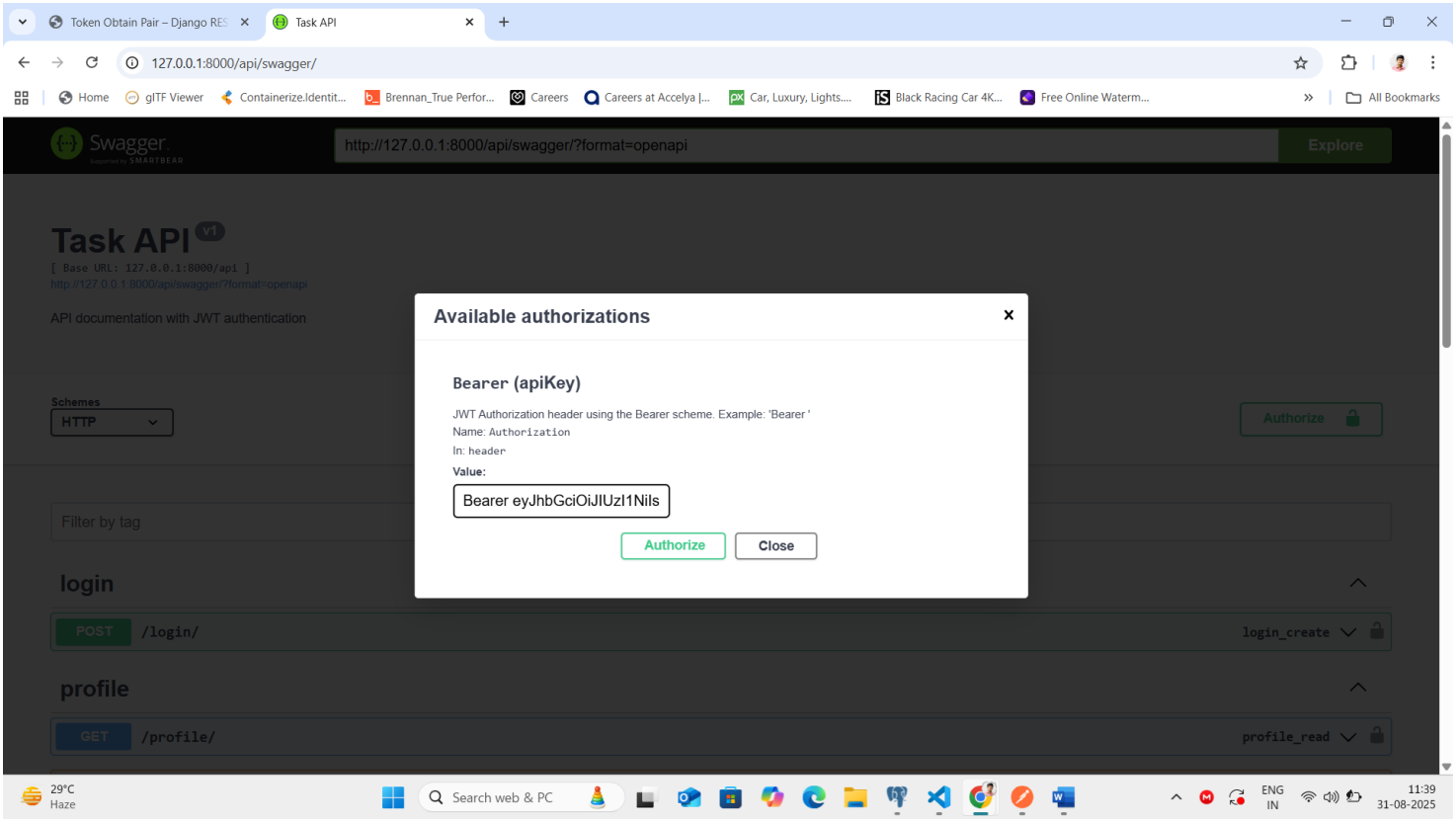
Test.py :

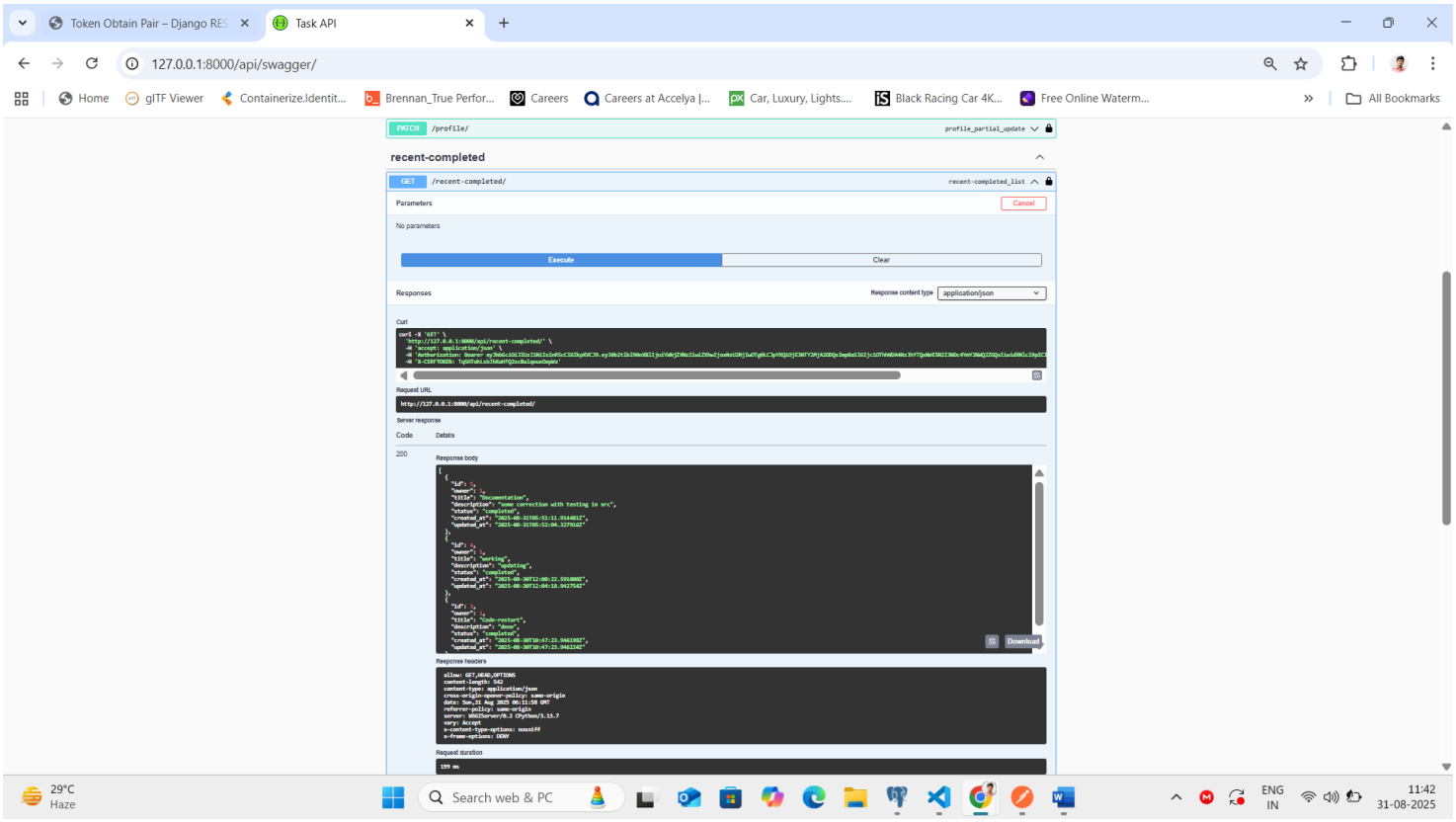
```python
from django.contrib.auth.models import User
from rest_framework.test import APITestCase
from rest_framework import status
from .models import Task

class TaskAPITestCase(APITestCase):
    def setUp(self):
        # Create a test user
        self.user = User.objects.create_user(username='testuser', password='testpass')

        # Create tasks
        Task.objects.create(owner=self.user, title='Task 1', status='completed')
        Task.objects.create(owner=self.user, title='Task 2', status='pending')
        Task.objects.create(owner=self.user, title='Task 3', status='completed')

        # Get JWT access token
        response = self.client.post('/api/token/', {'username': 'testuser', 'password': 'testpass'})
        self.token = response.data['access']

        # Include token in all requests
        self.client.credentials(HTTP_AUTHORIZATION=f'Bearer {self.token}')

    # Endpoint 1: Recent Completed Tasks
    def test_recent_completed_tasks(self):
        response = self.client.get('/api/recent-completed/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        for task in response.data:
            self.assertEqual(task['status'], 'completed')

    # Endpoint 2: Create Task
    def test_create_task(self):
        data = {'title': 'New Task', 'status': 'pending'}
        response = self.client.post('/api/tasks/', data)
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(response.data['title'], 'New Task')

    # Endpoint 3: Get Task List
    def test_get_task_list(self):
        response = self.client.get('/api/tasks/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertTrue(len(response.data) >= 3)
```

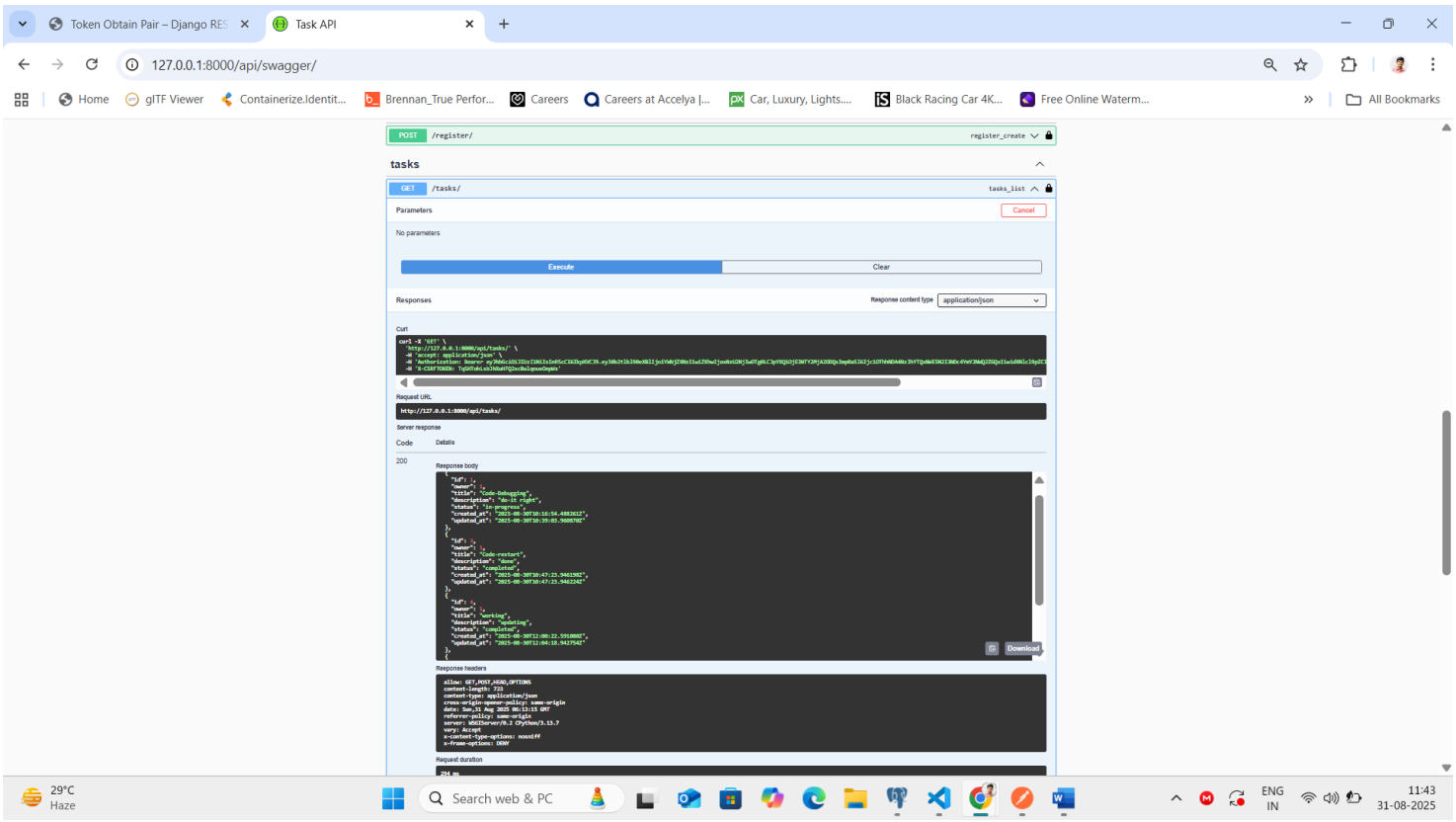
As part of the documentation process, I integrated **API documentation using Swagger**. This provides an interactive interface where all API endpoints are clearly listed, along with their request and response formats. It helps developers and users understand how to use the API without manually checking the code. The Swagger interface also supports authentication, where users need to provide their access token in order to test protected endpoints. While entering the token, it is important to prefix it with the word **“Bearer”** followed by a space and then the access token (for example, *Bearer*

your_access_token). This ensures that the authentication header is properly recognized by the system. With this setup, anyone can easily explore, test, and integrate with the API.

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'users',
    'rest_framework',
    'rest_framework_simplejwt',
    'tasks',
    'drf_yasg',
]
```




Token Obtain Pair – Django RES
Task API
127.0.0.1:8000/api/swagger/
PATCH /profile/
profile_partial_update
recent-completed
GET /recent-completed/
recent-completed_list
Parameters
Cancel
No parameters
Execute
Clear
Responses
Response content type
application/json
Curl
Request URL
http://127.0.0.1:8000/api/recent-completed/
Server response
Code
Details
200
Response body
Download
Response headers
Request duration
29°C
Haze
Search web & PC
ENG
IN
11:42
31-08-2025


Token Obtain Pair – Django RES
Task API
127.0.0.1:8000/api/swagger/
POST /register/
register_create
tasks
GET /tasks/
tasks_list
Parameters
Cancel
No parameters
Execute
Clear
Responses
Response content type
application/json
Curl
Request URL
http://127.0.0.1:8000/api/tasks/
Server response
Code
Details
200
Response body
Download
Response headers
Request duration
29°C
Haze
Search web & PC
ENG
IN
11:43
31-08-2025

Conclusion

The **internship_project** successfully implemented a complete backend system with **user authentication, task management, database integration, testing, and documentation.** It demonstrated the ability to set up a professional Django environment, build secure and scalable APIs, work with PostgreSQL efficiently, and follow best practices such as version control, testing, and proper documentation.

This project serves as a strong foundation for real-world applications and showcases the capability to handle backend development tasks from setup to deployment.