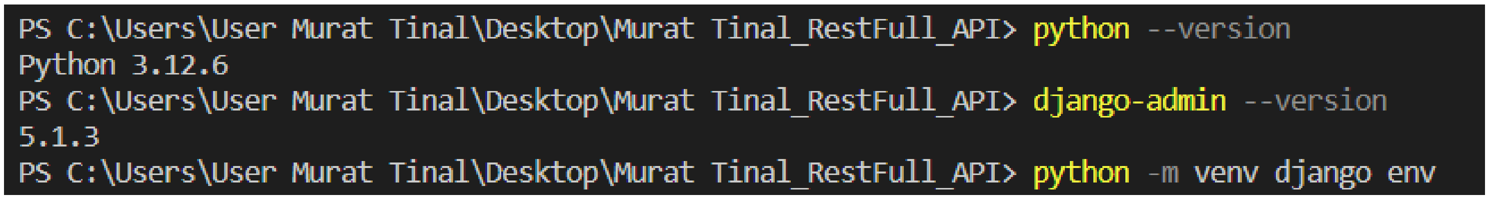
Building a RESTful API with Django Rest Framework

Murat Tinal
23MD0442
27.11.24

Exercise 1: Building a RESTful API with Django Rest Framework (DRF)

Objective: Create a fully functional RESTful API using Django Rest Framework.

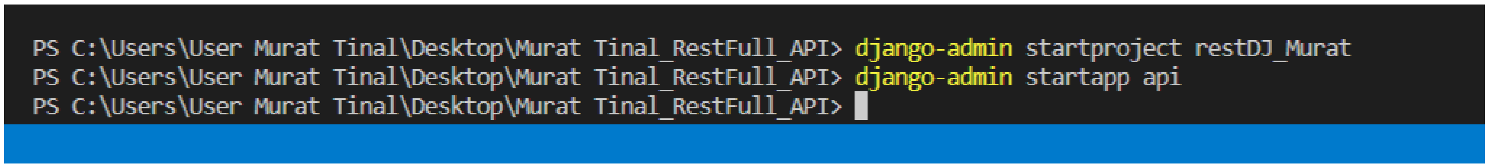
Django REST Framework is a Python-based toolkit for creating a web and REST API in Django components. It offers a range of features for seamless web development with Django, such as HTTP and application middleware, templates, forms, Model–View–Controller (MVP) architecture, security, data views, database management, caching, and so on.



```
PS C:\Users\User Murat Tinal\Desktop\Murat Tinal_RestFull_API> python --version
Python 3.12.6
PS C:\Users\User Murat Tinal\Desktop\Murat Tinal_RestFull_API> django-admin --version
5.1.3
PS C:\Users\User Murat Tinal\Desktop\Murat Tinal_RestFull_API> python -m venv django env
```

Fig.1

In Fig. 1, the first real step is to execute the following command to check if you have Python installed in your system. After that, run another command in the command prompt to check if the Django web framework is installed or not and to isolate dependencies, it would be great if you could build a virtual environment. From inside the projects folder, I execute the below-mentioned command to create the virtual environment.

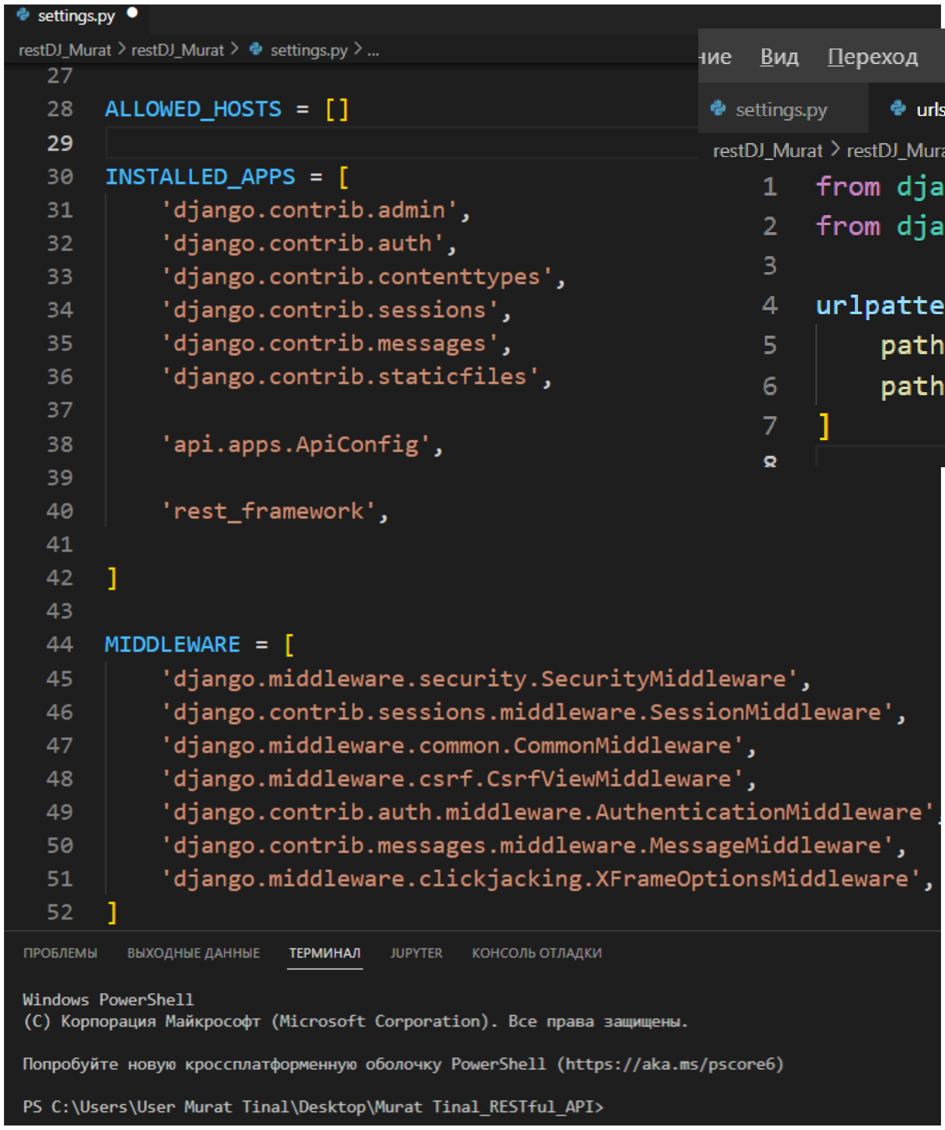


```
PS C:\Users\User Murat Tinal\Desktop\Murat Tinal_RestFull_API> django-admin startproject restDJ_Murat
PS C:\Users\User Murat Tinal\Desktop\Murat Tinal_RestFull_API> django-admin startapp api
PS C:\Users\User Murat Tinal\Desktop\Murat Tinal_RestFull_API> 
```

Fig.2

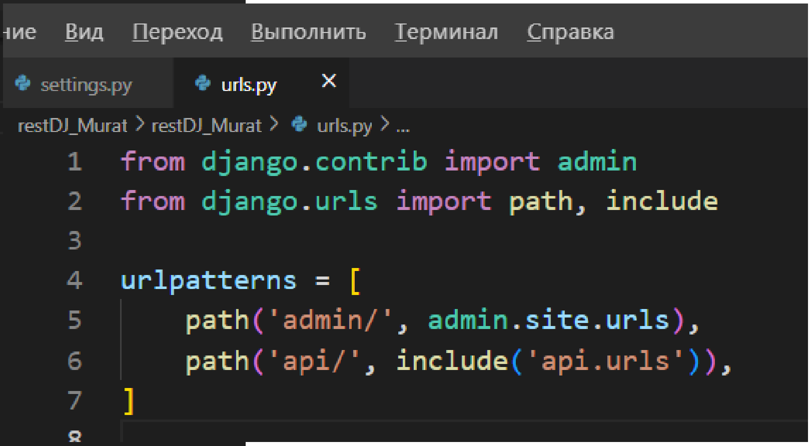
In Fig.2, in order to create a Django app, we have to create a Django project first. I called it: "***restDJ_Murat***" and created the Django app called: ***"api"***.

In Fig.4, In the INSTALLED_APPS file, I need to register the "api" as well as the Django REST Framework in the project settings. This is an important step as Django won't recognize your app without registration. This code that in Fig.3 defines URLs for a Django Project. There are: "admin/": Opens the admin panel and "api/": Connects to the URL in the api app.



```python

ALLOWED_HOSTS = []

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'api.apps.ApiConfig',

    'rest_framework',

]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

```
Windows PowerShell
(C) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

Попробуйте новую кроссплатформенную оболочку PowerShell (https://aka.ms/pscore6)

PS C:\Users\User Murat Tinal\Desktop\Murat Tinal_RESTful_API>
```



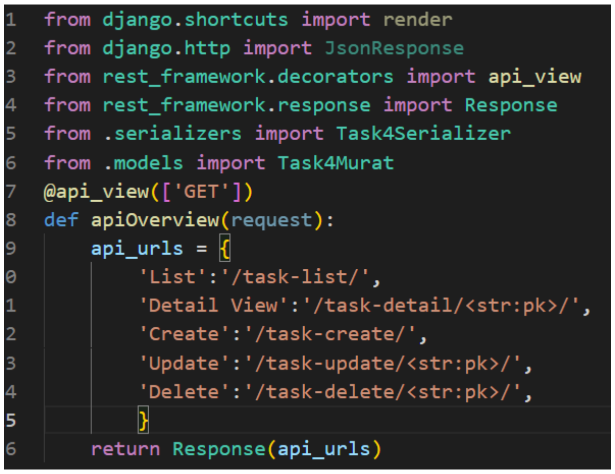
ние Вид Переход Выполнить Терминал Справка
settings.py
urls.py
restDJ_Murat > restDJ_Murat > urls.py > ...

```python
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('api.urls')),
]
```

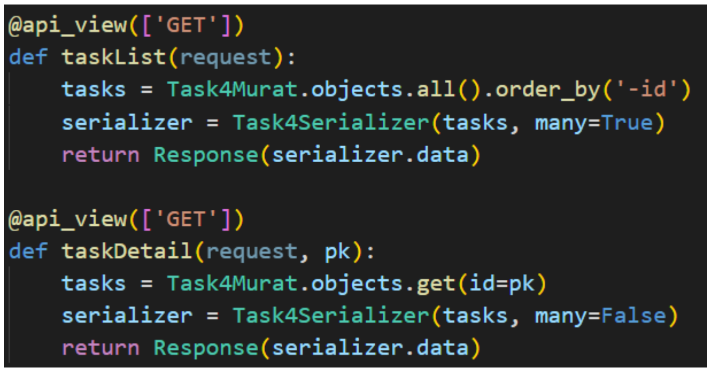
Fig.4

Fig.3

In order to prevent errors, I added a dummy view to the **views.py** file of the app. From the Django REST framework, I had to import the @api_view decorator and Response object. This is because @api_view displays the API while Response returns sterilized data in JSON format.



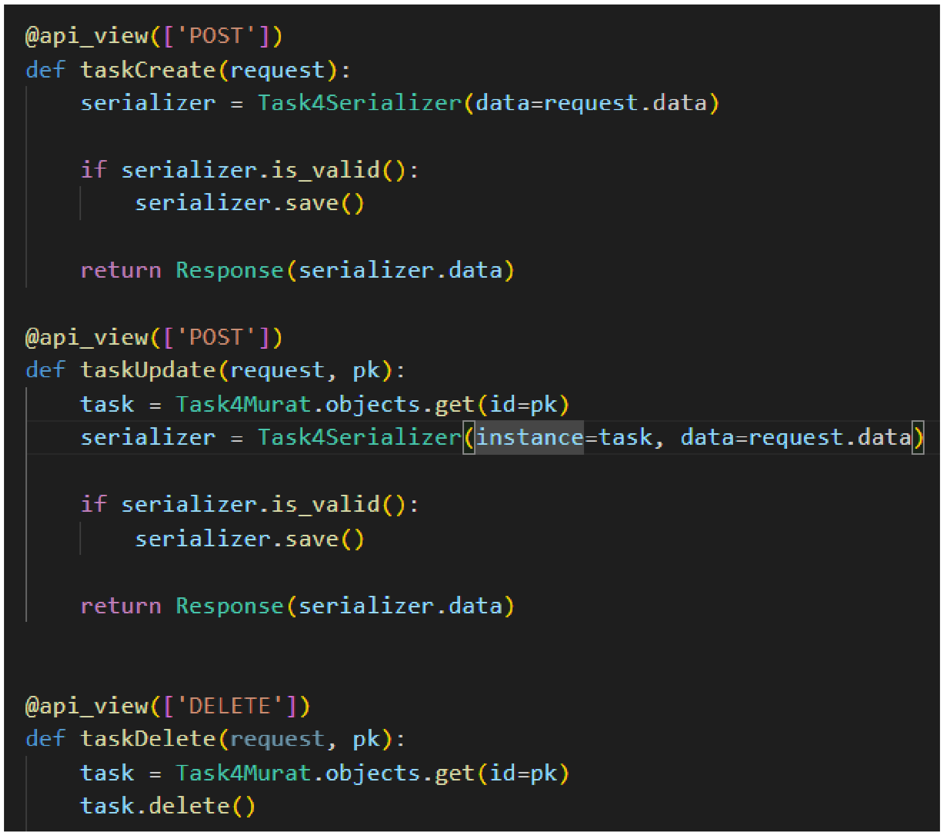
```python
from django.shortcuts import render
from django.http import JsonResponse
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .serializers import Task4Serializer
from .models import Task4Murat
@api_view(['GET'])
def apiOverview(request):
    api_urls = {
        'List':'/task-list/',
        'Detail View':'/task-detail/<str:pk>/',
        'Create':'/task-create/',
        'Update':'/task-update/<str:pk>/',
        'Delete':'/task-delete/<str:pk>/',
    }
    return Response(api_urls)
```



```python
@api_view(['GET'])
def taskList(request):
    tasks = Task4Murat.objects.all().order_by('-id')
    serializer = Task4Serializer(tasks, many=True)
    return Response(serializer.data)

@api_view(['GET'])
def taskDetail(request, pk):
    tasks = Task4Murat.objects.get(id=pk)
    serializer = Task4Serializer(tasks, many=False)
    return Response(serializer.data)
```

Fig.4

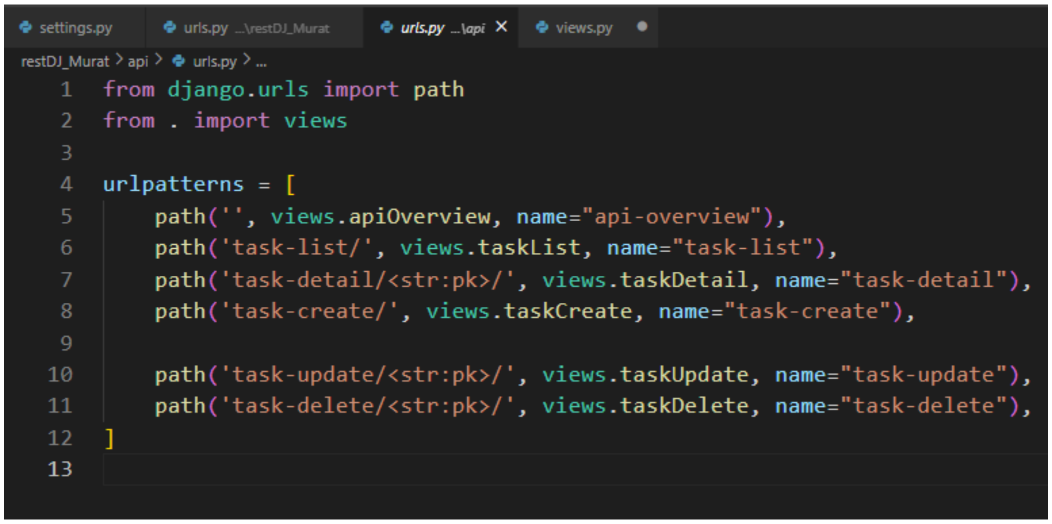
Fig.5



```python
@api_view(['POST'])
def taskCreate(request):
    serializer = Task4Serializer(data=request.data)

    if serializer.is_valid():
        serializer.save()

    return Response(serializer.data)

@api_view(['POST'])
def taskUpdate(request, pk):
    task = Task4Murat.objects.get(id=pk)
    serializer = Task4Serializer(instance=task, data=request.data)

    if serializer.is_valid():
        serializer.save()

    return Response(serializer.data)


@api_view(['DELETE'])
def taskDelete(request, pk):
    task = Task4Murat.objects.get(id=pk)
    task.delete()
```

Fig.6

In Fig.4 and ***Fig.7***, the function shows the main API routes.
/task-list/ - Get all tasks.
/task-detail/<str:pk>/ - Get details of one task.
/task-create/ - Add a new task.
/task-update/<str:pk>/ - Edit a task.
/task-delete/<str:pk>/ - Remove a task.

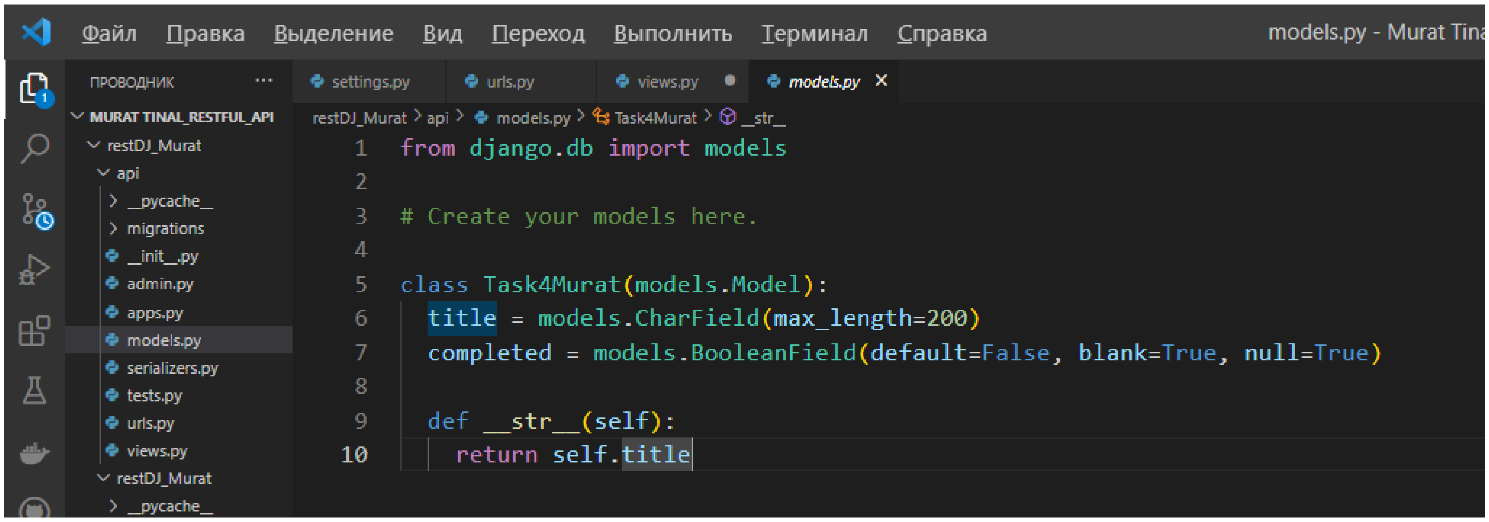


```python
from django.urls import path
from . import views

urlpatterns = [
    path('', views.apiOverview, name="api-overview"),
    path('task-list/', views.taskList, name="task-list"),
    path('task-detail/<str:pk>/', views.taskDetail, name="task-detail"),
    path('task-create/', views.taskCreate, name="task-create"),

    path('task-update/<str:pk>/', views.taskUpdate, name="task-update"),
    path('task-delete/<str:pk>/', views.taskDelete, name="task-delete"),
]
```

Fig.7

In Fig.5, The code has functions for working with tasks in a Django app. **taskList**: This function gets all tasks from the database, sorts them by newest first, and sends them back as a response. **taskDetail**: This function gets one task by its ID, prepares its details, and sends them back. **taskCreate**: This function adds a new task to the database if the provided data is correct and sends back the added task. **taskUpdate**: This function updates a task using its ID, saves the changes, and sends back the updated task. **taskDelete**: This function removes a task from the database using its ID.

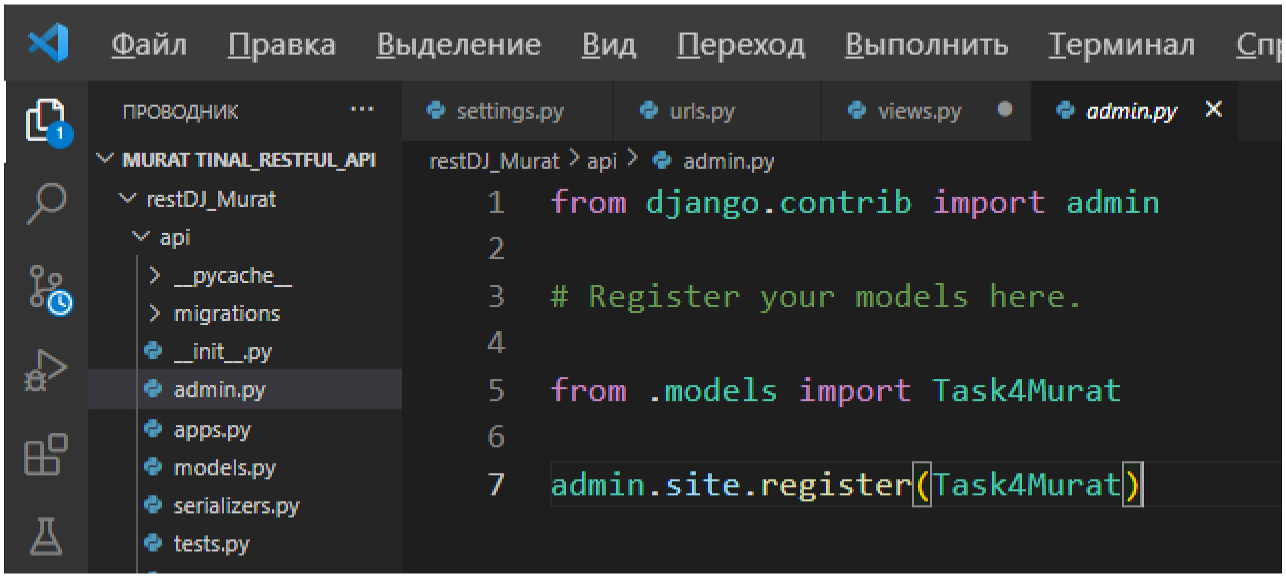
The name of the model class of my app is Task4Murat in ***Fig.8*** and this is how it should look:



```python
from django.db import models

# Create your models here.

class Task4Murat(models.Model):
    title = models.CharField(max_length=200)
    completed = models.BooleanField(default=False, blank=True, null=True)

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

Fig.8

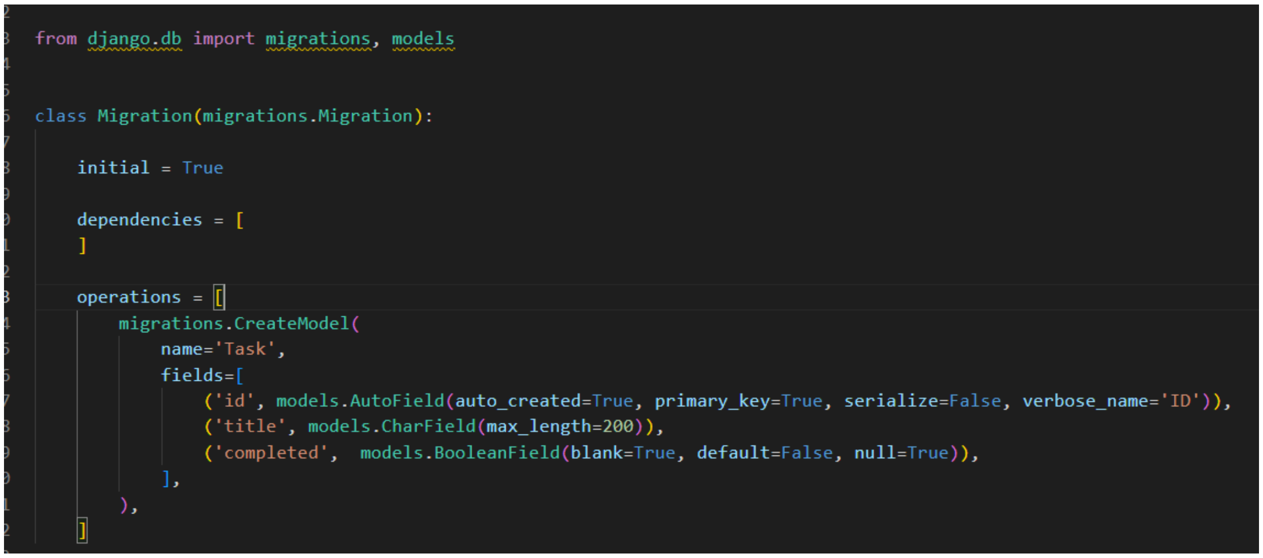
Now in the admin.py file, I added to register the model. Here's how:



```python
from django.contrib import admin

# Register your models here.

from .models import Task4Murat

admin.site.register(Task4Murat)
```

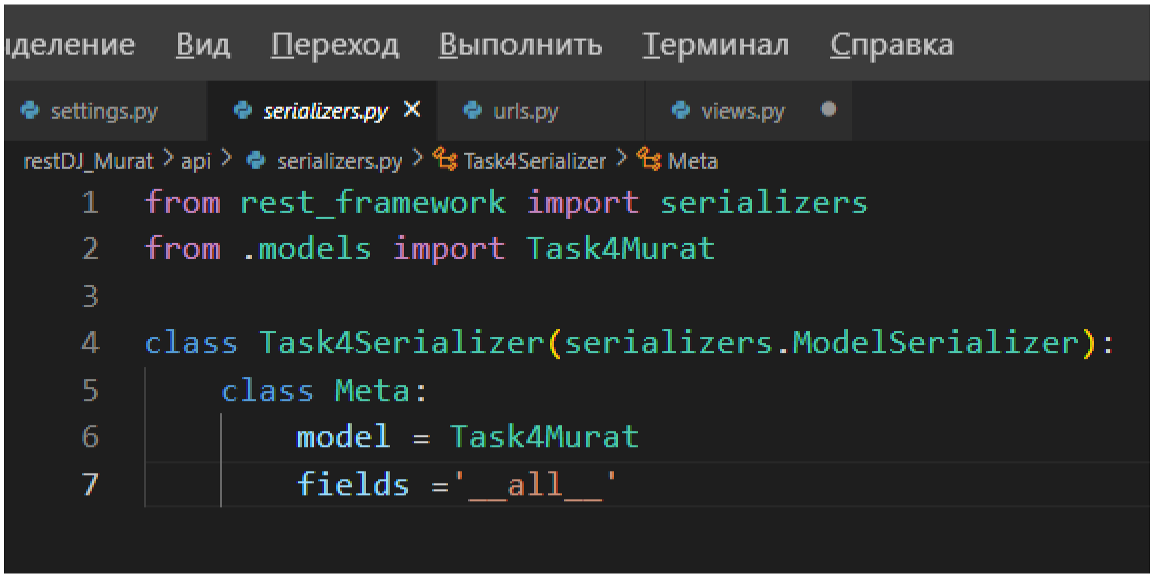
Fig.9

The Task4Murat class is a database model in Django with a title field for storing the task name and a completed field, which is a boolean to track if the task is done, defaulting to false while allowing blank and null values, and the __str__ method returns the task's title as a string representation.



```python
from django.db import migrations, models


class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Task',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('title', models.CharField(max_length=200)),
                ('completed',  models.BooleanField(blank=True, default=False, null=True)),
            ],
        ),
    ]
```

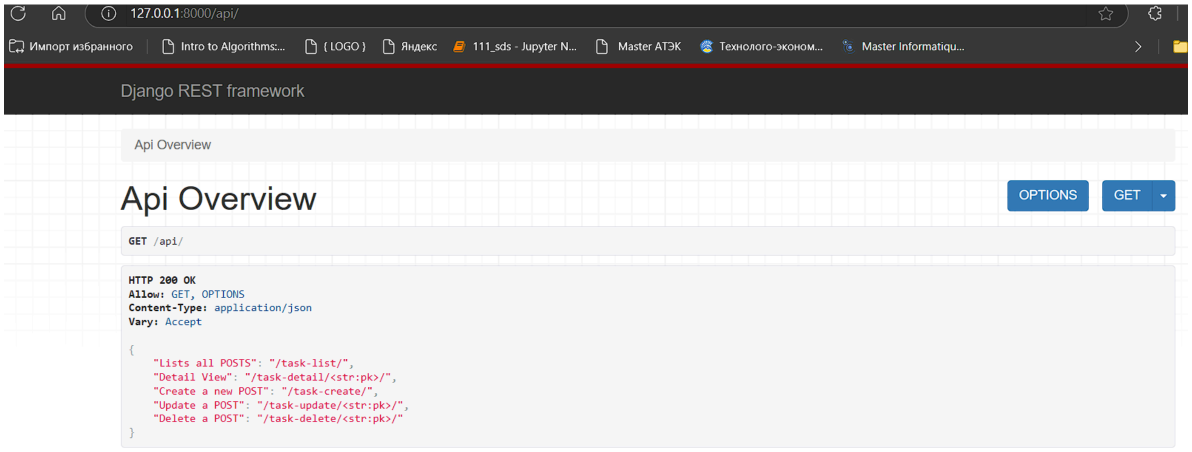
Fig. 10


ıделение Вид Переход Выполнить Терминал Справка
settings.py
serializers.py
urls.py
views.py
restDJ_Murat > api > serializers.py > Task4Serializer > Meta

```python
from rest_framework import serializers
from .models import Task4Murat

class Task4Serializer(serializers.ModelSerializer):
    class Meta:
        model = Task4Murat
        fields ='__all__'
```

Fig. 11

Serializer is used to convert complex data, such as Django model instances, into a format that is easy to work with, like JSON. This makes it easier to send data over the internet via APIs. It can also convert JSON data back into Python objects for processing. ***In Fig. 11*** code defines a serializer class Task4Serializer using Django REST Framework. The serializer converts Task4Model model instances into JSON format and vice versa. The Meta class specifies that the serializer should include all fields from the Task4Model model.

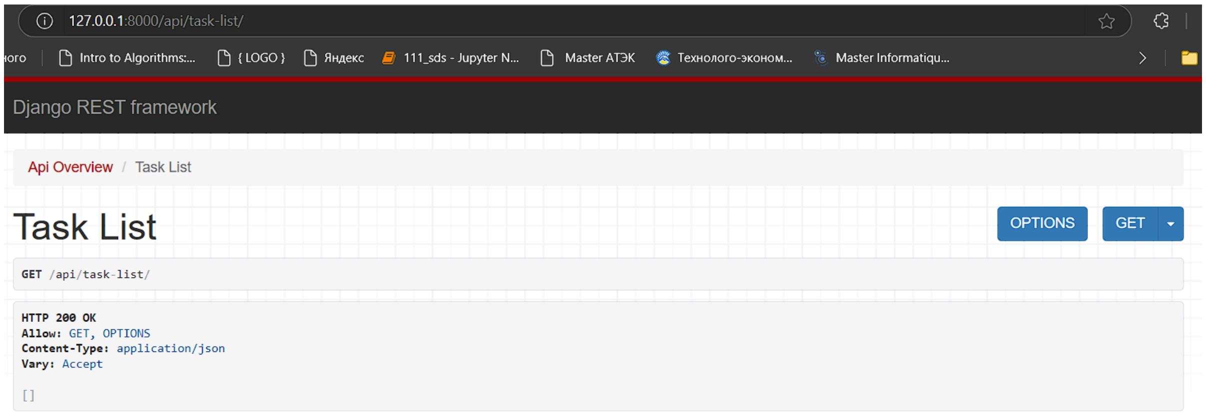


Api Overview

```
GET /api/
```

```
HTTP 200 OK
Allow: GET, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "Lists all POSTS": "/task-list/",
    "Detail View": "/task-detail/<str:pk>/",
    "Create a new POST": "/task-create/",
    "Update a POST": "/task-update/<str:pk>/",
    "Delete a POST": "/task-delete/<str:pk>/"
}
```

Figure 12: This is an overview of my api. On this page, you can find information about pages, including options to list, create, update, and delete pages

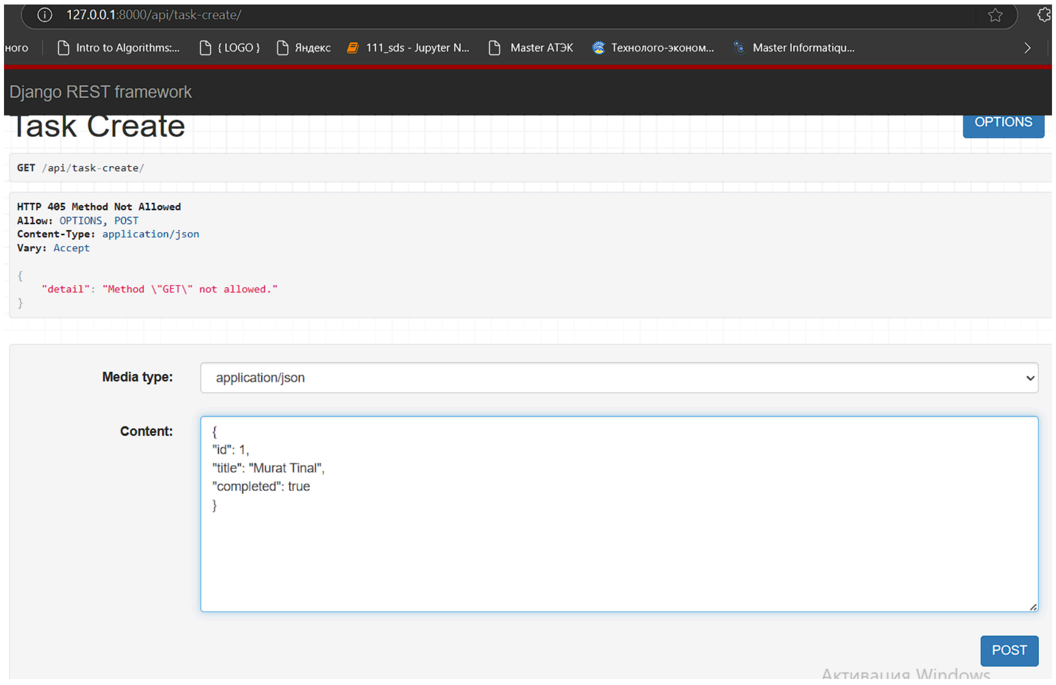


Task List

```
GET /api/task-list/
```

```
HTTP 200 OK
Allow: GET, OPTIONS
Content-Type: application/json
Vary: Accept

[]
```

Figure 13: This shows an empty page



Task Create

```
GET /api/task-create/
```

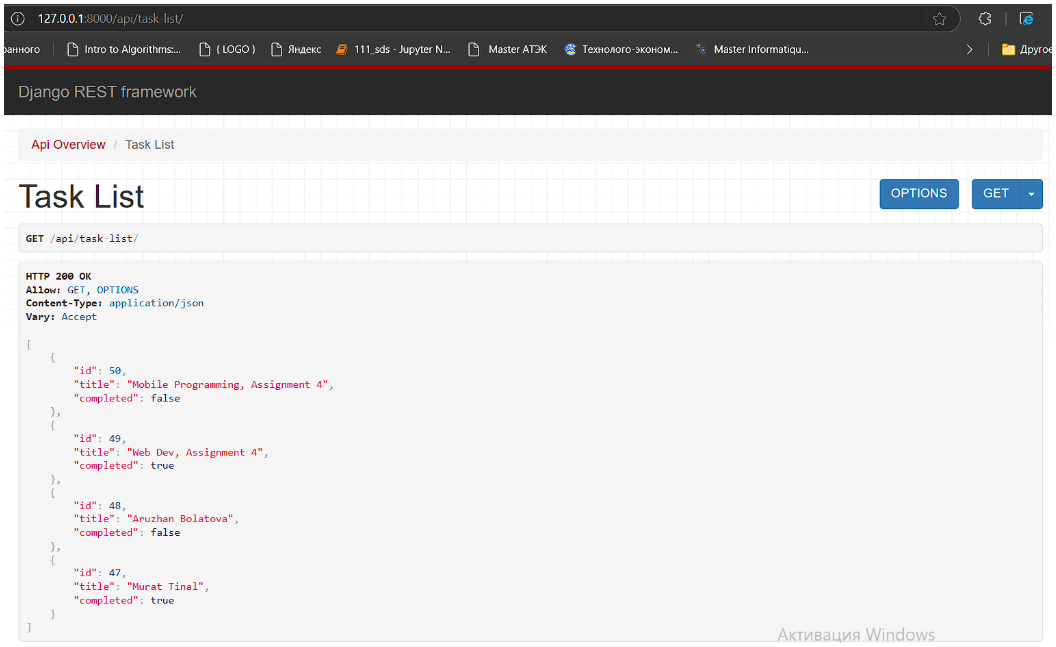
```
HTTP 405 Method Not Allowed
Allow: OPTIONS, POST
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

Media type: application/json

Content:

```
{
"id": 1,
"title": "Murat Tinal",
"completed": true
}
```

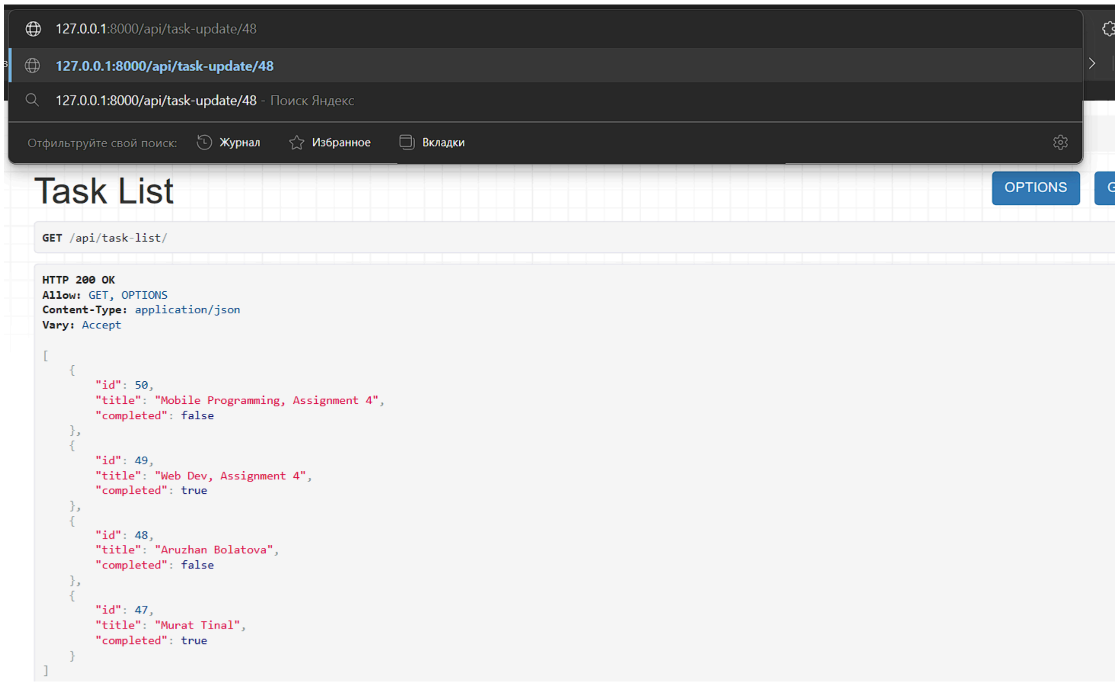
Figure 14: By using /task-create/, you can navigate to the Create Page. Here, I added information that appears on the /task-list/ page.



Task List

```
GET /api/task-list/

HTTP 200 OK
Allow: GET, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 50,
        "title": "Mobile Programming, Assignment 4",
        "completed": false
    },
    {
        "id": 49,
        "title": "Web Dev, Assignment 4",
        "completed": true
    },
    {
        "id": 48,
        "title": "Aruzhan Bolatova",
        "completed": false
    },
    {
        "id": 47,
        "title": "Murat Tinal",
        "completed": true
    }
]
```

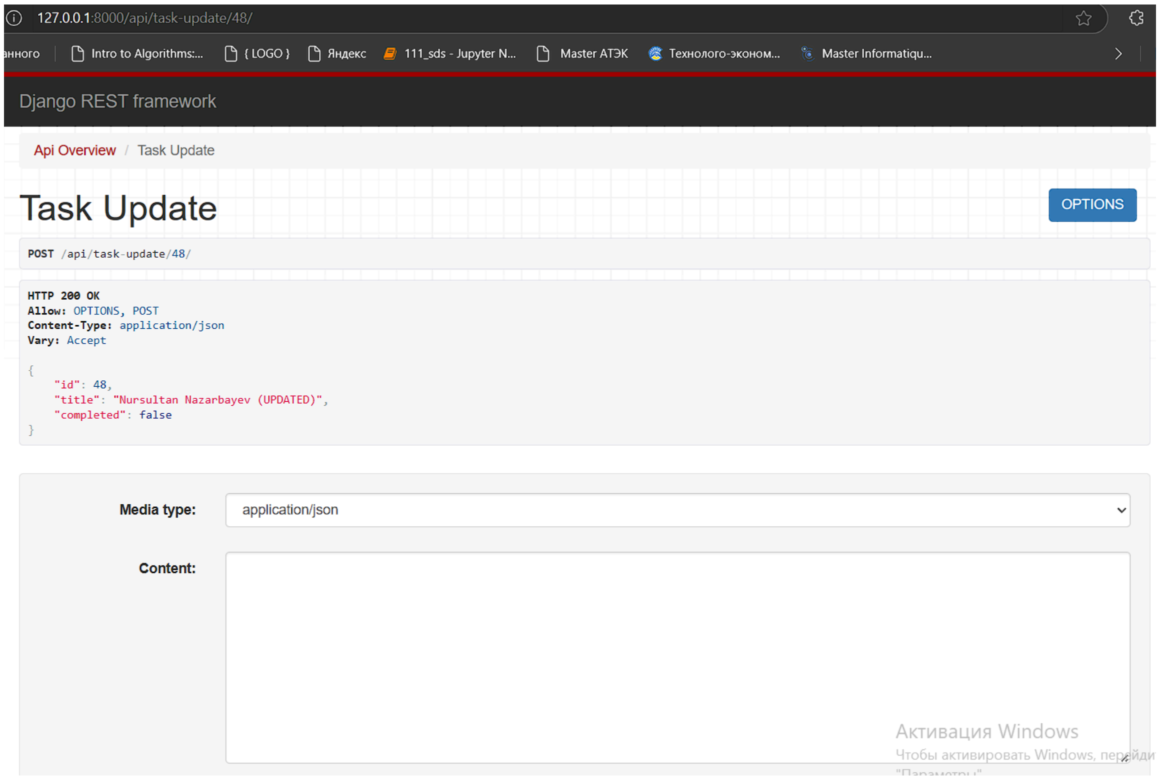
Figure 15: This displays the IDs and titles I created on the /task-create/ page (see Figure 14).


127.0.0.1:8000/api/task-update/48
127.0.0.1:8000/api/task-update/48
127.0.0.1:8000/api/task-update/48 - Поиск Яндекс
Отфильтруйте свой поиск:
Журнал
Избранное
Вкладки
OPTIONS

Task List

```
GET /api/task-list/

HTTP 200 OK
Allow: GET, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 50,
        "title": "Mobile Programming, Assignment 4",
        "completed": false
    },
    {
        "id": 49,
        "title": "Web Dev, Assignment 4",
        "completed": true
    },
    {
        "id": 48,
        "title": "Aruzhan Bolatova",
        "completed": false
    },
    {
        "id": 47,
        "title": "Murat Tinal",
        "completed": true
    }
]
```

Figure 16: I opened the Update Page using /task-update/"id_number".

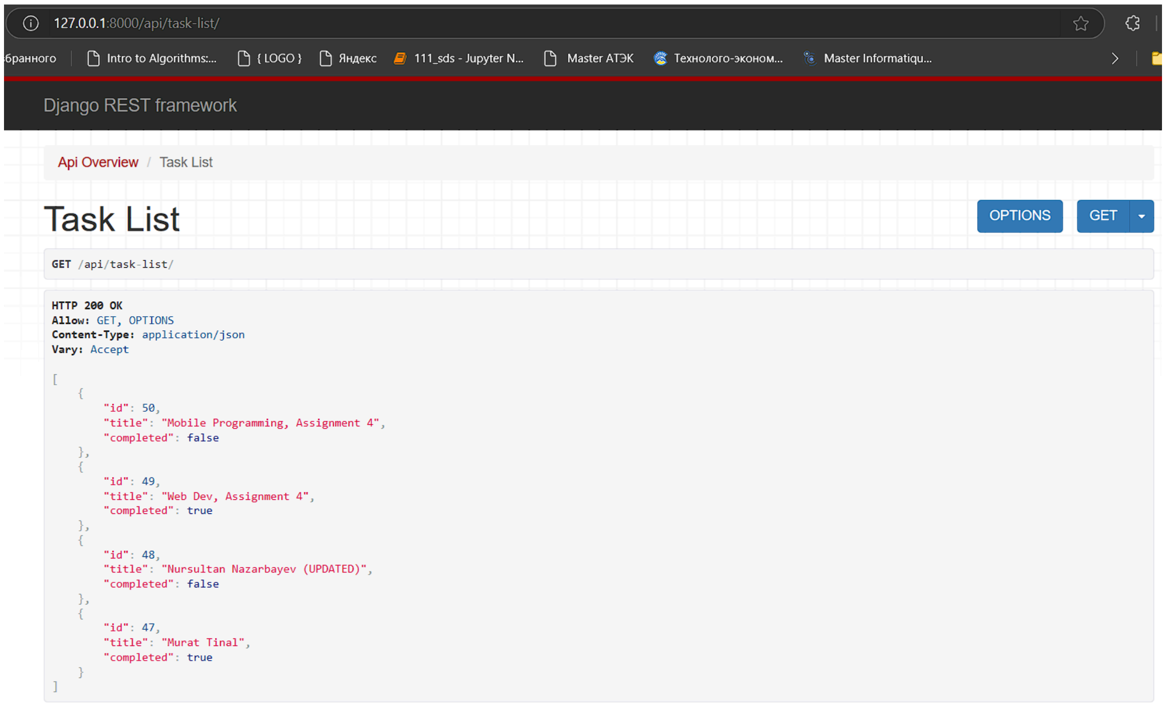

127.0.0.1:8000/api/task-update/48/
Django REST framework
Api Overview / Task Update
OPTIONS
Media type:
application/json
Content:
Активация Windows
Чтобы активировать Windows, перейди

Task Update

```
POST /api/task-update/48/
```

```
HTTP 200 OK
Allow: OPTIONS, POST
Content-Type: application/json
Vary: Accept

{
    "id": 48,
    "title": "Nursultan Nazarbayev (UPDATED)",
    "completed": false
}
```

Figure 17: I updated "Aruzhan Bolatova" to "Nursultan Nazarbayev."

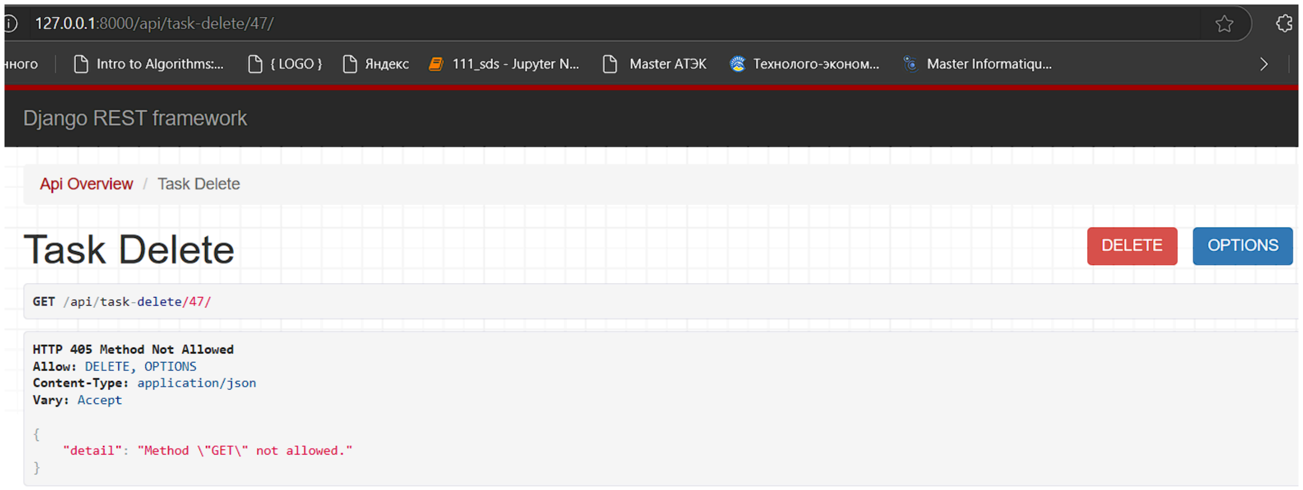


Task List

```
GET /api/task-list/
```

```
HTTP 200 OK
Allow: GET, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 50,
        "title": "Mobile Programming, Assignment 4",
        "completed": false
    },
    {
        "id": 49,
        "title": "Web Dev, Assignment 4",
        "completed": true
    },
    {
        "id": 48,
        "title": "Nursultan Nazarbayev (UPDATED)",
        "completed": false
    },
    {
        "id": 47,
        "title": "Murat Tinal",
        "completed": true
    }
]
```

Figure 18: This shows the updated list



```
GET /api/task-delete/47/
```

```
HTTP 405 Method Not Allowed
Allow: DELETE, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

Figure 19: I deleted the post with the ID number 47.

Conclusion

Completing Assignment 4, "Building a RESTful API with Django Rest Framework (DRF)," provided valuable hands-on experience in designing and implementing APIs using Django. Through this project, I developed a comprehensive understanding of RESTful principles, DRF's capabilities, and the practicalities of web app development.

References

- **https://radixweb.com/blog/create-rest-api-using-django-rest-framework**
- **https://www.youtube.com/watch?v=c708Nf0cHrs**
- **https://apidog.com/blog/django-rest-framework-create-api/**
- **https://medium.com/@ahmalopers703/getting-started-with-django-rest-api-for-beginners-9c121a2ce0d3**
- **https://appliku.com/post/building-restful-api-django-rest-framework/**

(My GitHub code: https://github.com/kiraakz/Murat_Tinal_Assign4_WebDev)