Rest API using APIViewSet

This Django project demonstrates the creation of API Endpoints using APIViewSet and default router.

Prerequisites

- Python (version 3.7 or higher)
- Django (version 3.x or higher)
- Djangorestframework (version 3.15.2 or higher
- MySQLClient / PyMYSQL
- Virtual environment tool (optional but recommended)

Setup Instructions

Follow these steps to set up and run the Django "api_view_set_demo" project:

1. Create and Activate a Virtual Environment (Optional but Recommended)

It's a good practice to create a virtual environment for each project to manage dependencies.

a. **Create a virtual environment**:

python -m venv .venv

b. **Activate the virtual environment:**

On Windows:
.venv\Scripts\activate

On macOS/Linux:
source .venv/bin/activate

2. Install Django

Once the virtual environment is active, install Django:

pip install django
pip install djangorestframework

pip install mysqlclient
or
pip install PyMySql

3. Check version

Python 3

python3 –version

python3 -m django –version

Earlier than Pthan Python 3

python –version

python -m django --version

4. Create a Django Project

To create a new Django project named api_view_set_demo:

django-admin startproject api_view_set_demo

5. Navigate into the Project Directory

cd api_view_set_demo

6. Create a Django App

Create an app named student:

python manage.py startapp student

7. Configure the App in the Project

Open `api_view_set_demo/settings.py` and add the newly created `student and rest framework` to the INSTALLED_APPS list:

```
INSTALLED_APPS = [
    # Django default apps
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
```

```
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Add your app here
    'rest_framework',    # To implement Django rest framework (DRF)
    'student',           # App created

]
```

8. Define the Models in app

In `student/models.py`, create student profile model as below

```
class Profile(models.Model):
    # Basic Information
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    email = models.EmailField(unique=True)

    def __str__(self):
        return f"{self.first_name} {self.last_name}"
```

9. Define the Profile Serializer in app

In `student/serializers.py`, create student profile serrializer as below

```
from rest_framework import serializers  # Import the serializers module from Django
REST Framework
from .models import Profile  # Import the Profile model from the current app's models

# Define a serializer for the Profile model
class ProfileSerializer(serializers.ModelSerializer):
    # Meta class defines metadata for the serializer
    class Meta:
        model = Profile  # Specify the model to serialize
        fields = ['id', 'first_name', 'last_name', 'email']  # Fields to include in
the serialization

    # Custom validation method for the 'email' field
    def validate_email(self, value):
        """
        Check if the email is unique in the Profile model.
        If the email already exists, raise a ValidationError.
        """
        if Profile.objects.filter(email=value).exists():  # Check if a Profile with
this email already exists
            raise serializers.ValidationError("A student with this email already
exists.")  # Raise error if not unique
        return value  # Return the value if it's unique
```

10. Define API Views in app

In `student/api_views.py`, create views to access endpoints to interface

```
# Import necessary modules
from rest_framework.views import APIView  # Add this import for APIView
from .models import Profile  # Import the Profile model to interact with the database
from .serializers import ProfileSerializer  # Import the ProfileSerializer to convert
the Profile model into JSON format
from rest_framework.response import Response  # Import the Response class to return
HTTP responses
from rest_framework.exceptions import NotFound  # Import the NotFound exception to
handle missing objects
from rest_framework import status  # Import HTTP status codes for response statuses

# Class-based view to list all profiles
class ProfileListView(APIView):
    # GET request handler
    def get(self, request):
        Profiles = Profile.objects.all()  # Retrieve all Profile objects from the
database
        serializer = ProfileSerializer(Profiles, many=True)  # Serialize the queryset
into JSON format
        return Response(serializer.data, status=status.HTTP_200_OK)  # Return the
serialized data as a response with status 200 OK

# Class-based view to create a new profile
class ProfileCreateView(APIView):
    # POST request handler
    def post(self, request):
        serializer = ProfileSerializer(data=request.data)  # Deserialize the incoming
data into a ProfileSerializer instance
        if serializer.is_valid():  # Check if the serializer data is valid
            serializer.save()  # Save the new Profile object into the database
            return Response(serializer.data, status=status.HTTP_201_CREATED)  # Return
the created data with status 201 CREATED
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)  #
Return errors with status 400 if validation fails

# Class-based view to retrieve details of a specific profile
class ProfileDetailView(APIView):
    # Helper method to get the Profile object by its primary key (pk)
    def get_object(self, pk):
        try:
            return Profile.objects.get(pk=pk)  # Try to get the profile by its primary
key
        except Profile.DoesNotExist:  # If the profile doesn't exist, raise a NotFound
exception
            raise NotFound("Profile not found.")

    # GET request handler for retrieving a specific profile by pk
    def get(self, request, pk):
        Profile = self.get_object(pk)  # Retrieve the Profile object using the helper
method
        serializer = ProfileSerializer(Profile)  # Serialize the Profile object into
JSON
        return Response(serializer.data)  # Return the serialized data as a response

# Class-based view to update an existing profile
class ProfileUpdateView(APIView):
    # Helper method to get the Profile object by its primary key (pk)
    def get_object(self, pk):
        try:
            return Profile.objects.get(pk=pk)  # Try to get the profile by its primary
key
        except Profile.DoesNotExist:  # If the profile doesn't exist, raise a NotFound
exception
            raise NotFound("Profile not found.")

    # PUT request handler for updating a profile by pk
    def put(self, request, pk):
        Profile = self.get_object(pk)  # Retrieve the Profile object using the helper
method
        serializer = ProfileSerializer(Profile, data=request.data, partial=True)  #
Deserialize the updated data
        if serializer.is_valid():  # Check if the updated data is valid
            serializer.save()  # Save the updated profile back to the database
            return Response(serializer.data)  # Return the updated profile data as a
response
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)  #
Return errors with status 400 if validation fails
```

```
# Class-based view to delete a profile
class ProfileDeleteView(APIView):
    # Helper method to get the Profile object by its primary key (pk)
    def get_object(self, pk):
        try:
            return Profile.objects.get(pk=pk)  # Try to get the profile by its primary
key
        except Profile.DoesNotExist:  # If the profile doesn't exist, raise a NotFound
exception
            raise NotFound("Profile not found.")

    # DELETE request handler for deleting a profile by pk
    def delete(self, request, pk):
        Profile = self.get_object(pk)  # Retrieve the Profile object using the helper
method
        Profile.delete()  # Delete the Profile object from the database
        return Response(status=status.HTTP_204_NO_CONTENT)  # Return an empty response
with status 204 NO CONTENT (successful deletion) Profile instances
```

11. Define the URL Pattern in project

In `student/urls.py`, define URL patterns to access endpoints

```
from django.urls import path
from . import api_views  # Import the views from api_views.py

urlpatterns = [
    # URL pattern for listing profiles
    path('profiles/', api_views.ProfileListView.as_view(), name='profile-list'),

    # URL pattern for creating a new profile
    path('profiles/create/', api_views.ProfileCreateView.as_view(), name='profile-
create'),

    # URL pattern for viewing a specific profile's details
    path('profiles/<int:pk>/', api_views.ProfileDetailView.as_view(), name='profile-
detail'),

    # URL pattern for updating a specific profile
    path('profiles/<int:pk>/update/', api_views.ProfileUpdateView.as_view(),
name='profile-update'),

    # URL pattern for deleting a specific profile
    path('profiles/<int:pk>/delete/', api_views.ProfileDeleteView.as_view(),
name='profile-delete'),
]
```

In `api_view_set_demo/urls.py`, include the student app URLs:

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

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

12. Run Database Migrations

Before running the server, apply the initial migrations:

python manage.py makemigrations

python manage.py migrate

13. Run the Development Server

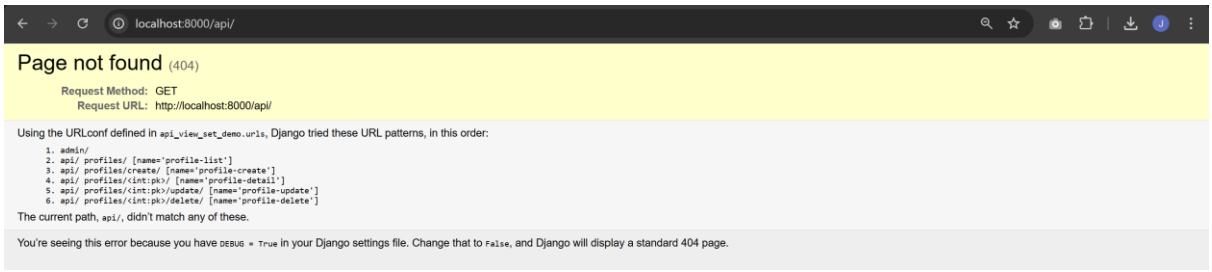
Finally, start the Django development server:

python manage.py runserver

14. Launch the Application and see the result

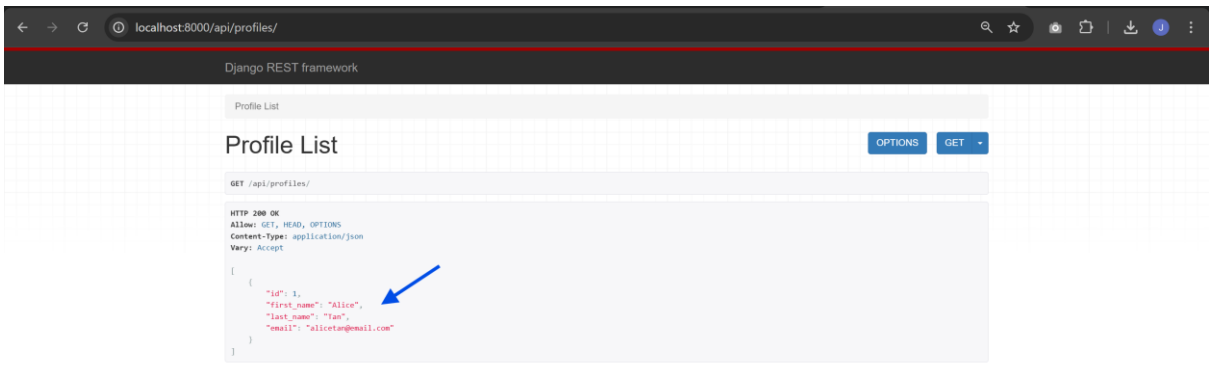
Open your web browser and visit:

http://127.0.0.1:8000/api/

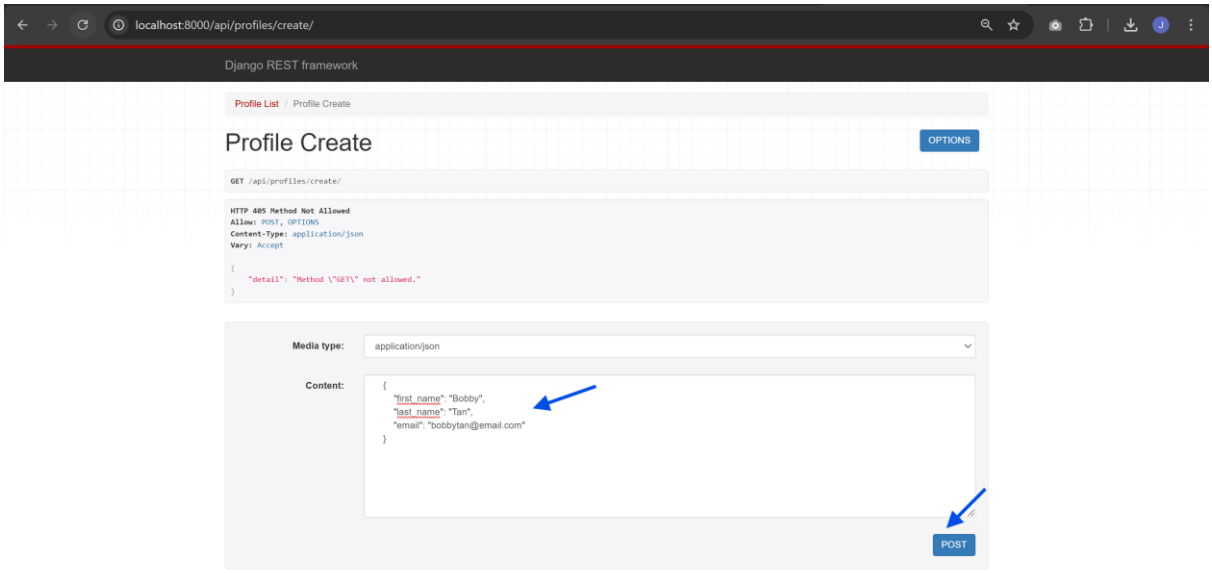


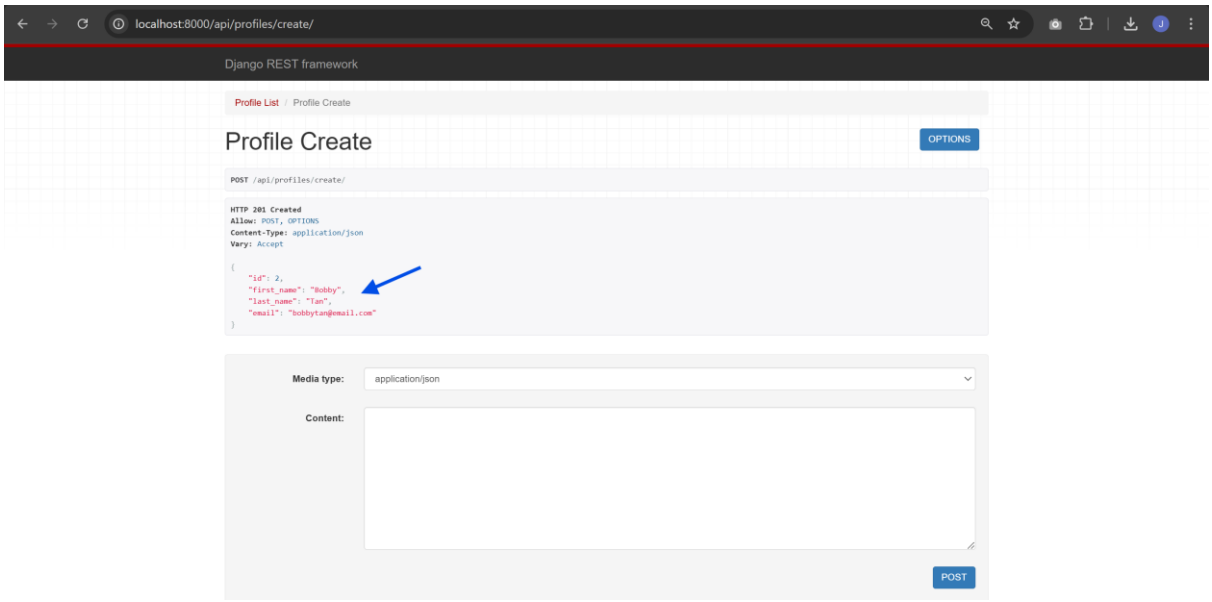
To list all the profiles – http://127.0.0.1:8000/api/profiles/

You can observe list of profiles

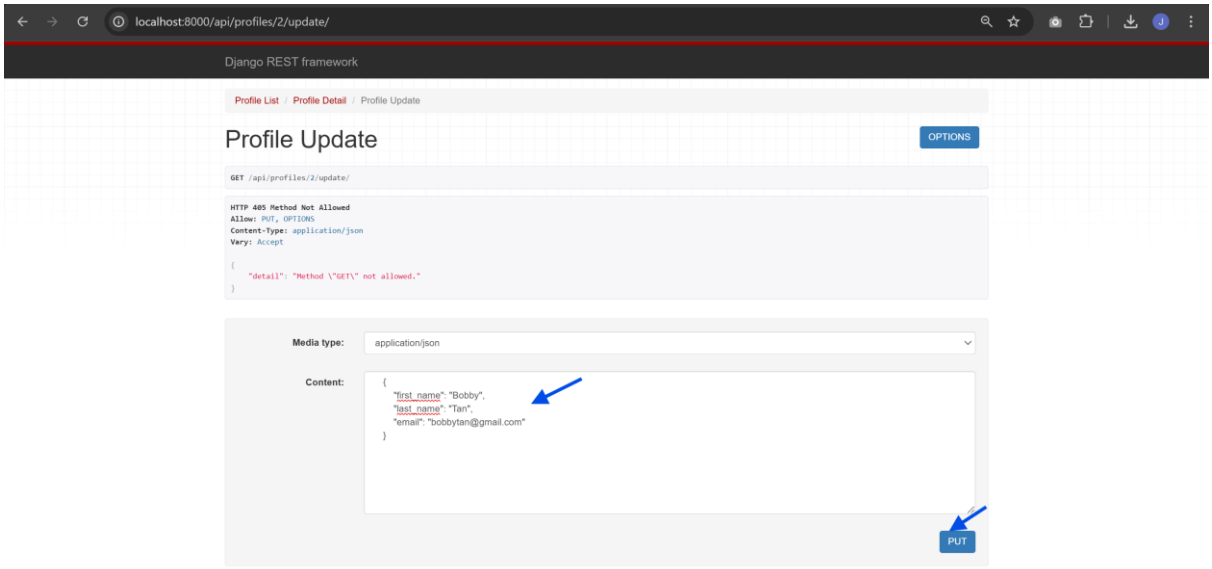


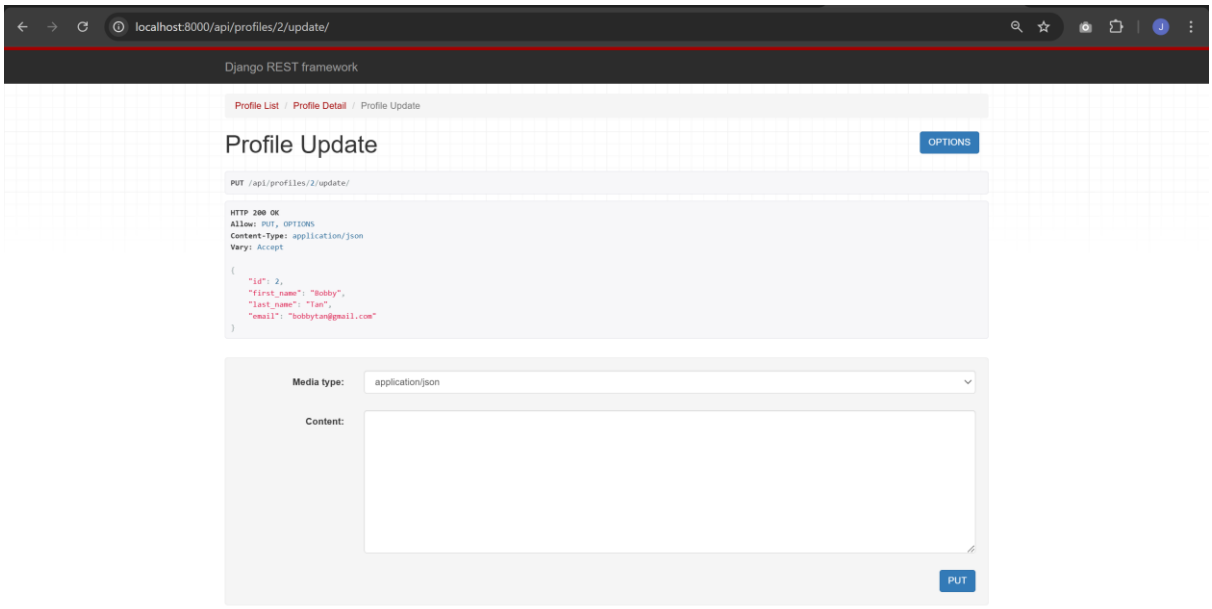
Enter the details and click POST. Observe that the profile being created and listed



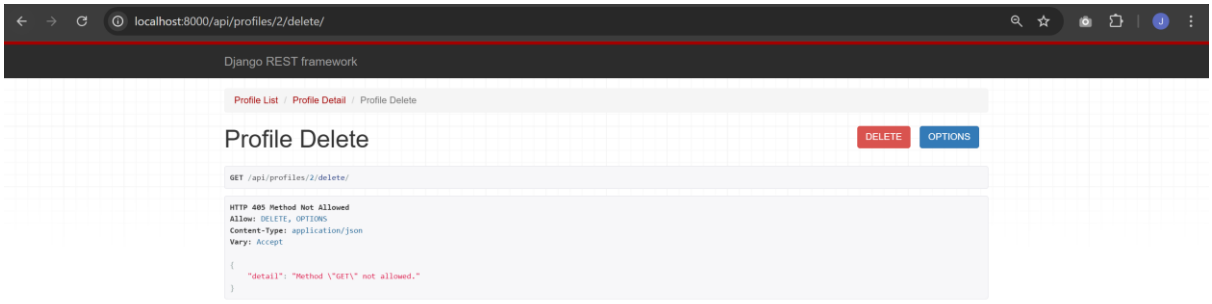


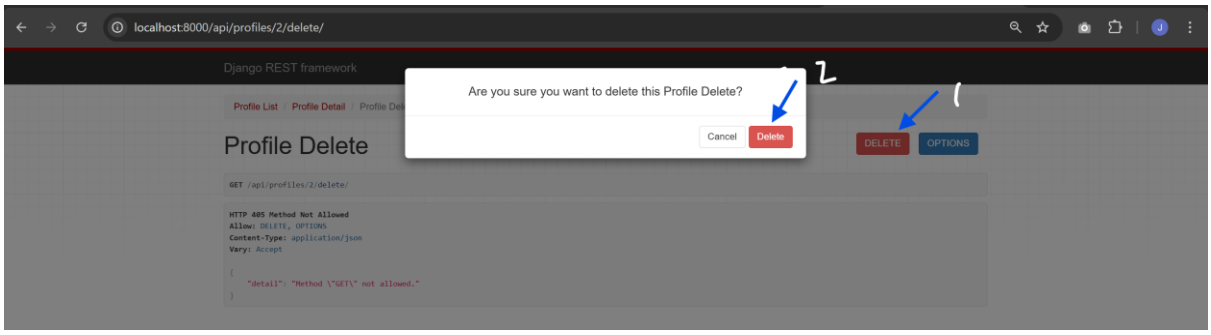
Select the profile based on id and make changes to the data to update the profile details.



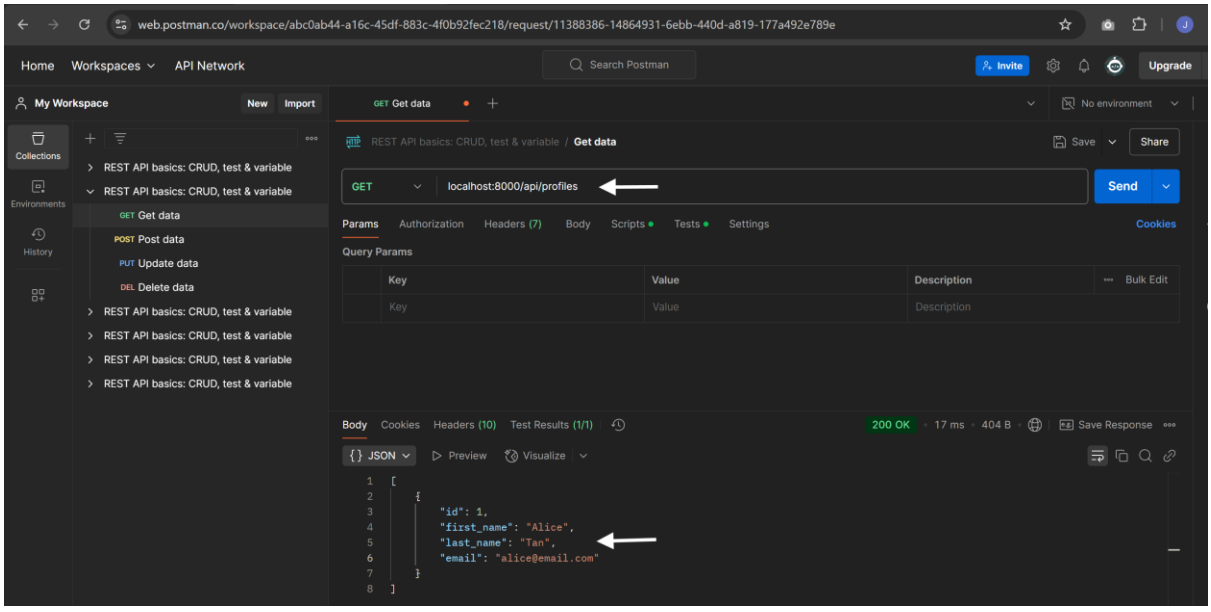


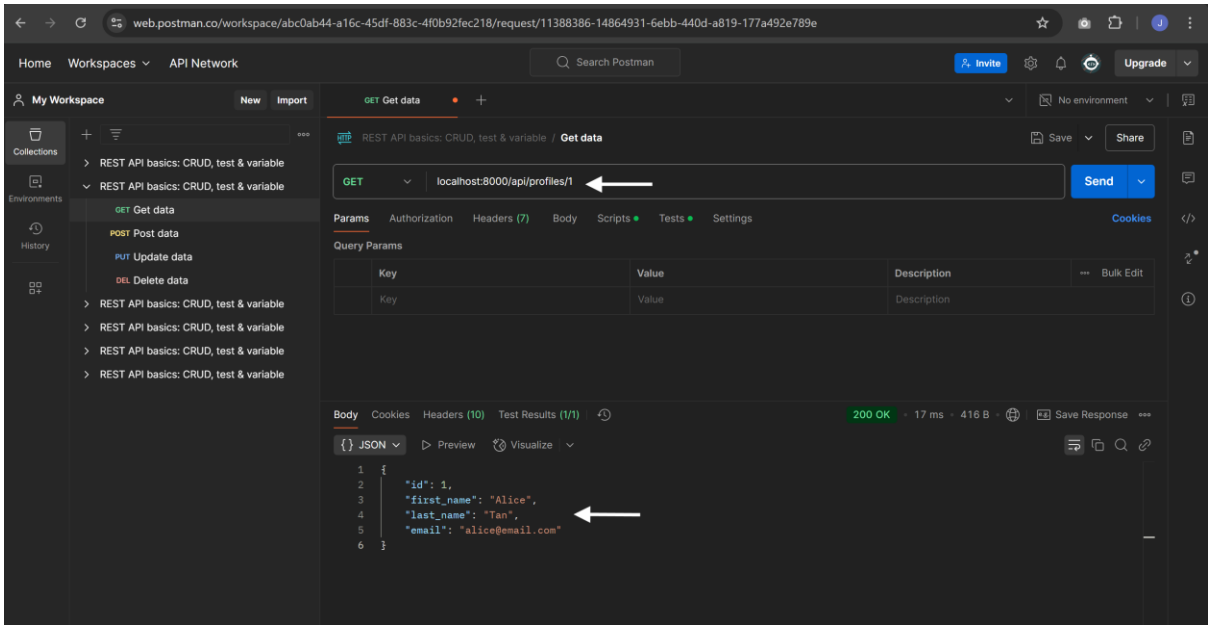
Select the profile based on id and delete the profile



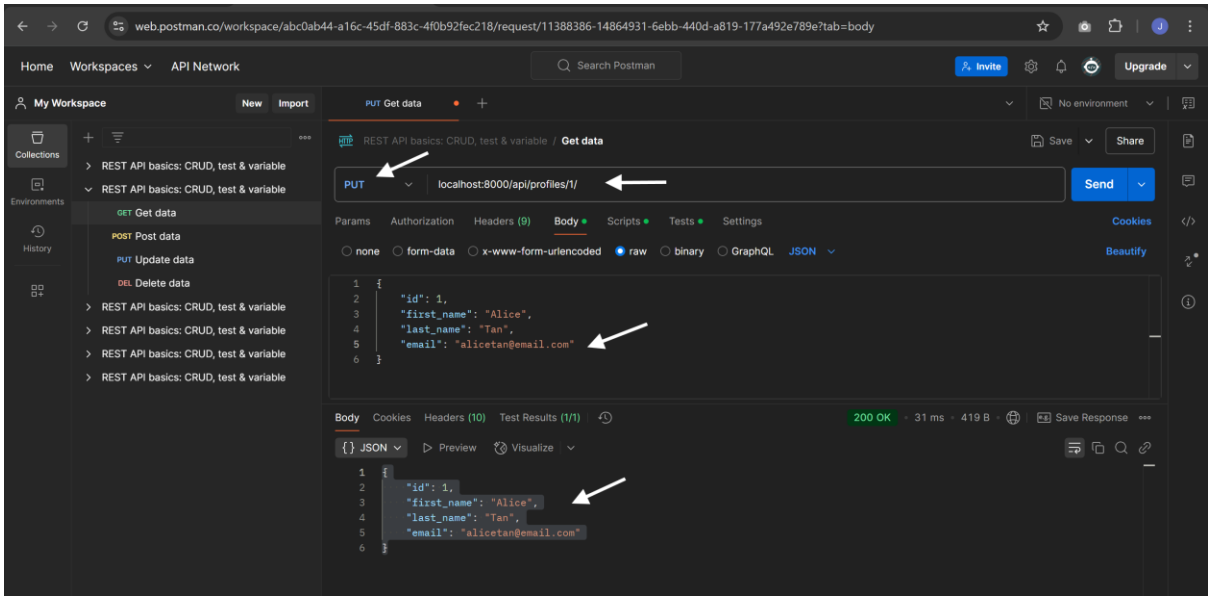


You can test the same using postman





Edit the raw data to update the profile, HTTP method (PUT) and Click Send to observe update



15. Deactivating the Virtual Environment

To deactivate the virtual environment when you're done working:

Deactivate

16. ModelViewSet vs APIViewSet

In Django REST Framework (DRF), both ModelViewSet and APIViewSet are used to handle the logic for creating, reading, updating, and deleting resources (CRUD operations). However, they differ in how they are defined and the level of abstraction they provide.

1. ModelViewSet

- **Definition**: A ModelViewSet is a specialized subclass of DRF's ViewSet that provides a set of common behaviors for interacting with models.
- **Usage**: It is designed to work specifically with Django models and provides CRUD operations automatically.
- **Key Features**:
    - **Automatic CRUD Operations**: Provides default implementations for operations like list, create, retrieve, update, and destroy based on the model linked to it.
    - **Serializer Integration**: Automatically uses the serializer class defined in the ModelViewSet.
    - **URL Configuration**: When you use ModelViewSet, you typically pair it with a Router for automatic URL routing.
    - **Authentication, Permissions, Filtering**: You can define authentication, permissions, and filtering easily.
- **Advantages:**
    - **Reduced Boilerplate**: Handles most of the CRUD logic for you.
    - **Efficient**: Ideal for working directly with models and databases.
    - **Integration with Routers**: Simplifies the URL routing process.
- **Disadvantages:**
    - **Less Flexibility**: It may be less suitable if you need to perform complex logic or operations beyond basic CRUD.

2. APIViewSet (Custom ViewSet)

- **Definition**: APIViewSet is a more flexible base class that allows you to build views with more control over the behavior of each action.

- **Usage**: While ModelViewSet is tied to Django models, APIView and its subclass APIViewSet allow you to handle any kind of custom logic or non-model-based views.
- **Key Features**:
    - **Custom Logic**: Allows you to define custom methods for each action (get, post, put, delete, etc.).
    - **Manual CRUD Implementation**: You have to manually implement the CRUD logic.
    - **No Automatic Model Binding**: Unlike ModelViewSet, you must define how data is serialized and deserialized, and how database queries are handled.
    - **More Control**: You get more control over the behavior of the view, which may be needed for complex use cases.
- **Advantages**:
    - **Full Flexibility**: You can define every aspect of your view, including custom actions and complex logic.
    - **No Direct Model Binding**: Ideal for API views that do not work directly with Django models.
- **Disadvantages**:
    - **More Boilerplate**: You need to manually implement each CRUD operation.
    - **More Code**: Requires more code compared to the ModelViewSet.

Comparison

Feature	**ModelViewSet**	**APIViewSet**
Automatic CRUD	Yes, automatically handles CRUD operations	No, requires manual implementation
Model Integration	Directly tied to Django models	No direct model integration
Custom Logic	Limited flexibility for custom logic	Full flexibility for custom logic

Feature	ModelViewSet	APIViewSet
Serializer Handling	Automatically uses serializers based on model	Requires custom serialization logic
URL Routing	Uses DRF Routers for automatic URL routing	Requires manual URL routing
Use Case	Best for simple APIs that interact with models	Best for complex or custom API logic
Boilerplate Code	Less boilerplate, as most logic is handled	More boilerplate, as you need to implement the logic yourself

---

When to Use Which?

- **Use ModelViewSet** when:
    - You are working with a Django model and want to quickly expose CRUD functionality.
    - You want automatic URL routing with minimal effort.
    - You prefer convention over configuration for simple use cases.
- **Use APIViewSet** when:
    - You need more control over your views.
    - You want to implement custom logic that doesn't directly involve Django models.
    - You are handling non-model-based data or have complex API requirements.

Both are powerful tools in Django REST Framework, and the choice depends on the level of customization required for your API.