Rest API using Generics

- **Definition**: DRF's generic views provide pre-built class-based views for common API patterns like List, Create, Retrieve, Update, and Destroy.
- **Primary Use Case**: Best suited when you need granular control over each operation but still want to use DRF's abstraction layers.

Core Generic Attributes/Views

Generic View	Purpose
ListAPIView	Handles GET requests to list resources.
CreateAPIView	Handles POST requests to create resources.
RetrieveAPIView	Handles GET requests for a specific resource.
UpdateAPIView	Handles PUT or PATCH requests to update resources.
DestroyAPIView	Handles DELETE requests to delete resources.
RetrieveUpdateAPIView	Combines Retrieve and Update operations.
RetrieveDestroyAPIView	Combines Retrieve and Destroy operations.
RetrieveUpdateDestroyAPIView	Combines Retrieve, Update, and Delete.

Advantages

1. **Granularity**: Separate views for each operation provide better control.
2. **Customization**: Easier to override specific methods without impacting others.
3. **Flexibility**: Customizable behavior for each HTTP method.

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_generics_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_generics_demo:

django-admin startproject api_generics_demo

5. Navigate into the Project Directory

cd api_generics_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_generics_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 generics to use pre-built views for common API operations (List, Create,
Retrieve, Update, Destroy)
from rest_framework import generics

# Import the Profile model to define the queryset for the views
from .models import Profile

# Import the ProfileSerializer to handle data validation and serialization
from .serializers import ProfileSerializer

# A view to handle both listing all Profile objects (GET) and creating a new Profile
(POST)
class ProfileListCreateView(generics.ListCreateAPIView):
    queryset = Profile.objects.all()  # Fetches all Profile objects from the database
    serializer_class = ProfileSerializer  # Uses ProfileSerializer to validate and
serialize data

# A view to handle retrieving (GET), updating (PUT/PATCH), and deleting (DELETE) a
single Profile object
class ProfileRetrieveUpdateDestroyView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Profile.objects.all()  # Fetches all Profile objects, filtered later by
primary key (pk)
    serializer_class = ProfileSerializer  # Uses ProfileSerializer to validate and
serialize data
```

11. Define the URL Pattern in project

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

```
from django.urls import path
from .api_views import ProfileListCreateView, ProfileRetrieveUpdateDestroyView

urlpatterns = [
    # URL for listing all profiles or creating a new profile
    path('profiles/', ProfileListCreateView.as_view(), name='profile-list-create'),
```

```
    # URL for retrieving, updating, or deleting a specific profile by its primary key
(pk)
    path('profiles/<int:pk>/', ProfileRetrieveUpdateDestroyView.as_view(),
name='profile-detail'),
]
```

In `api_generics_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

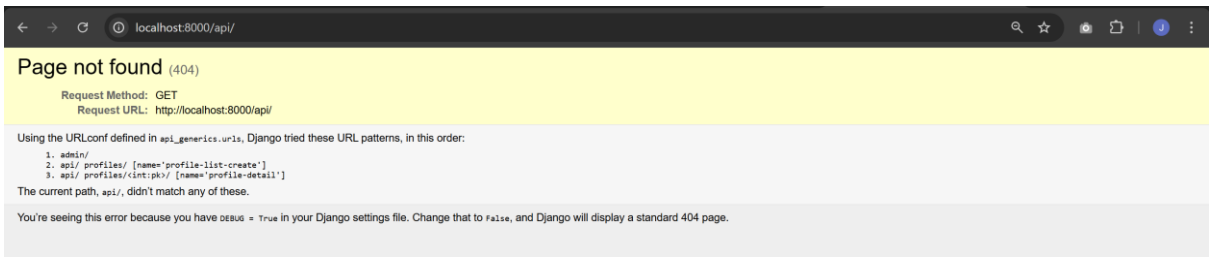
Finally, start the Django development server:

python manage.py runserver

14. Launch the Application and see the result

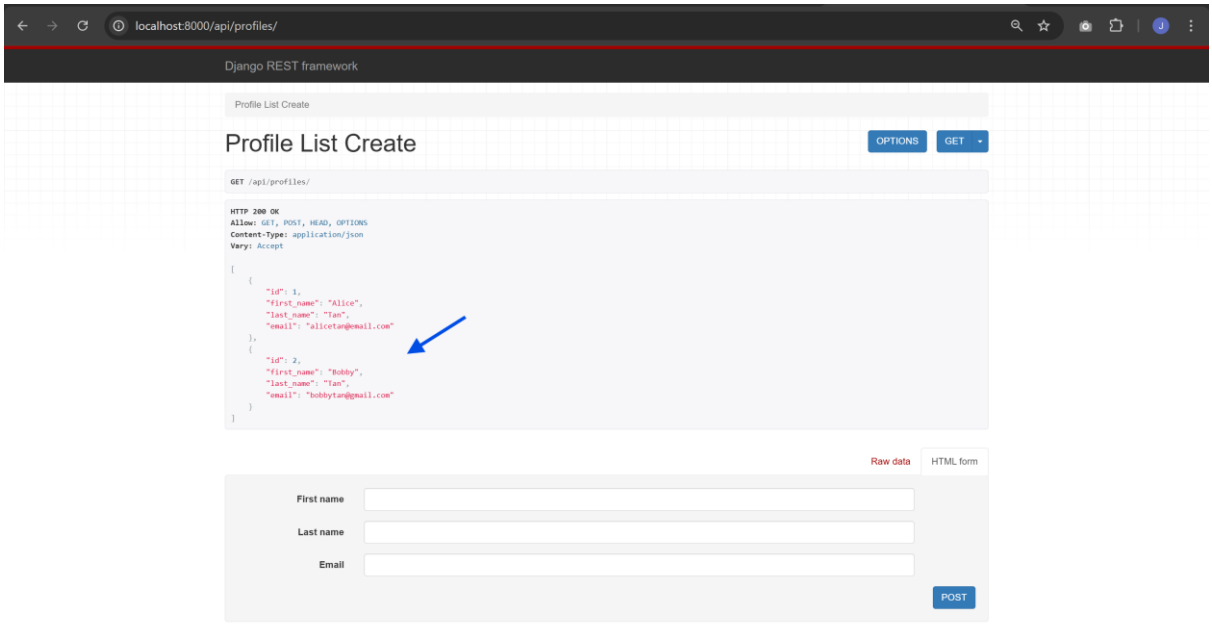
Open your web browser and visit:

http://127.0.0.1:8000/api/

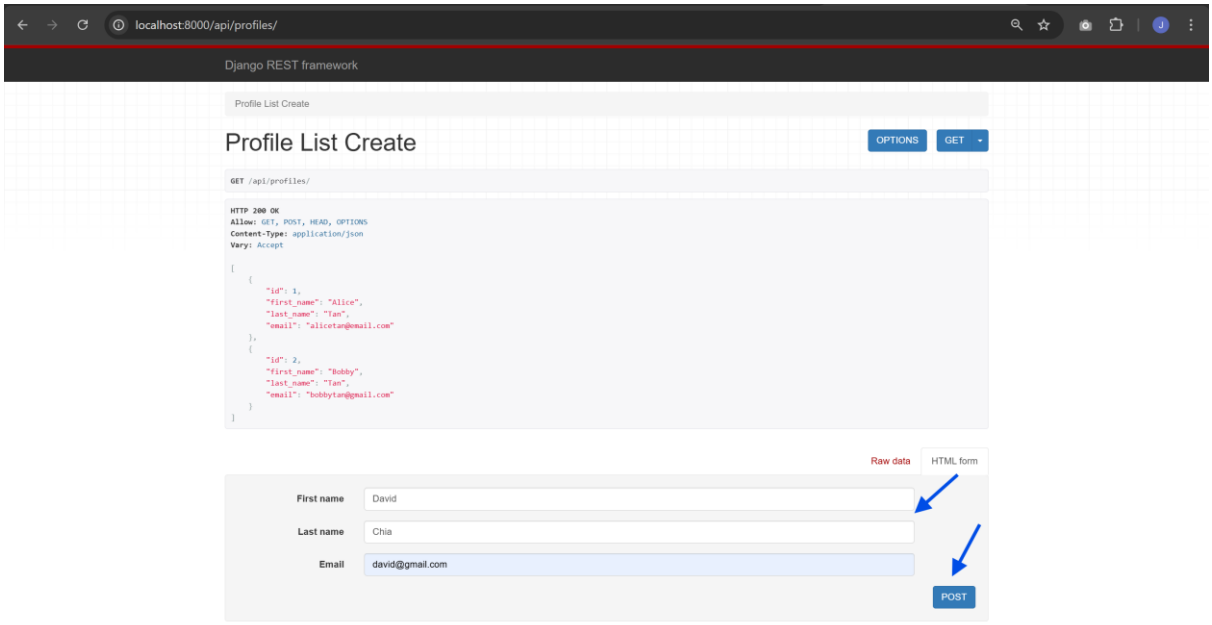


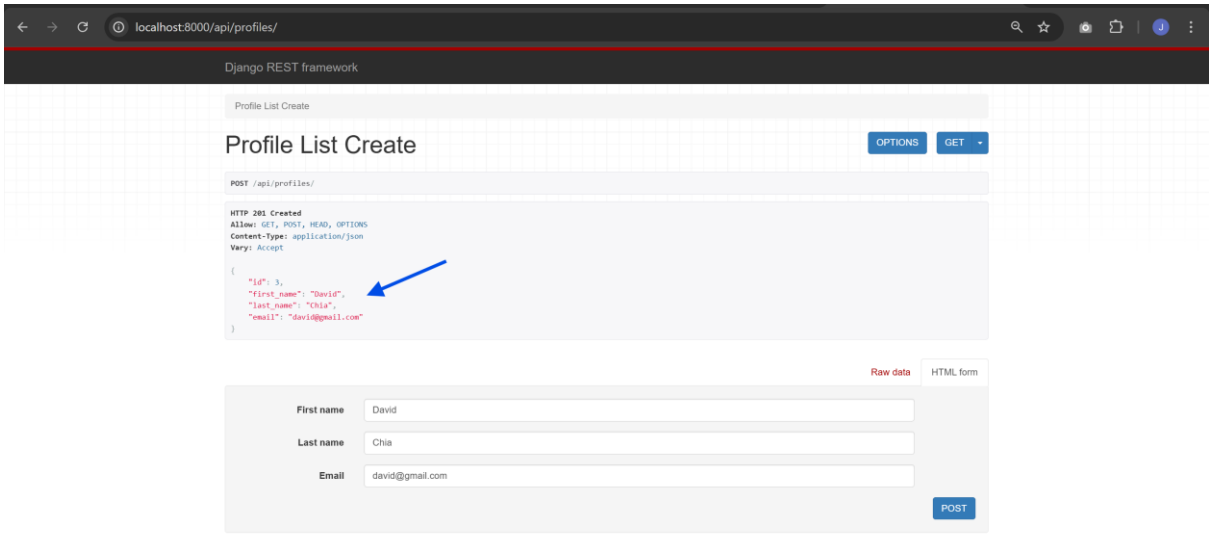
To list all the profiles – http://127.0.0.1:8000/api/profiles/

You can observe list of profiles

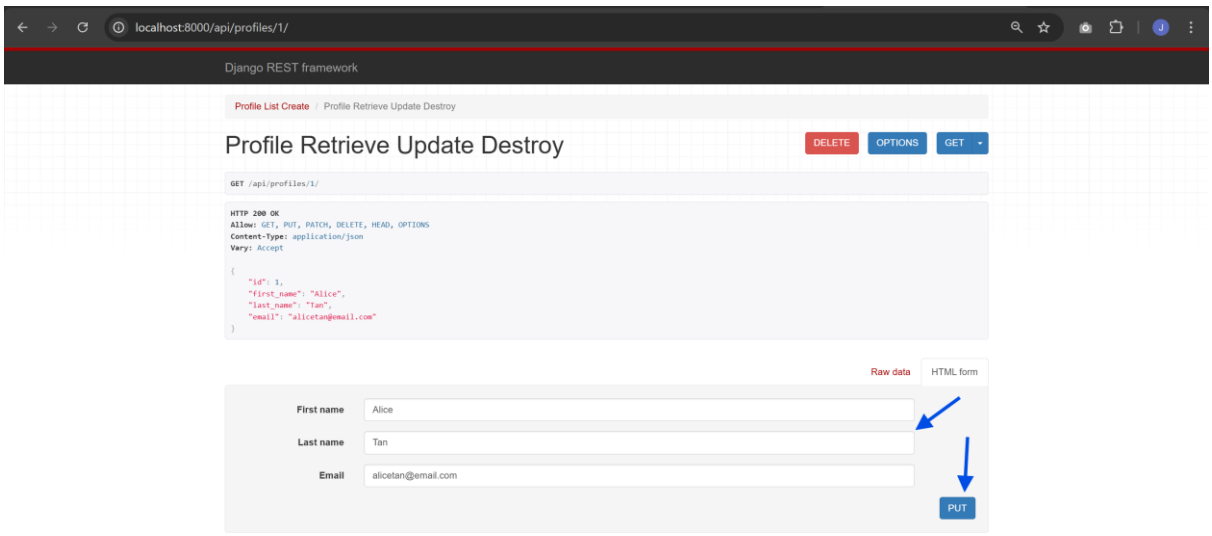


Enter the details and click POST. Observe that the profile being created and listed

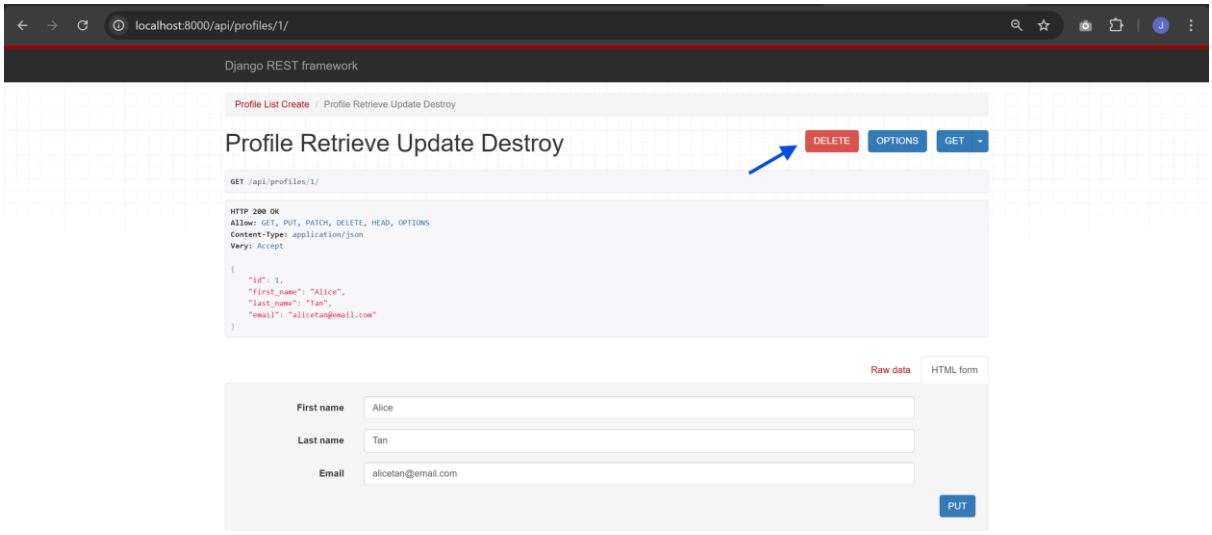




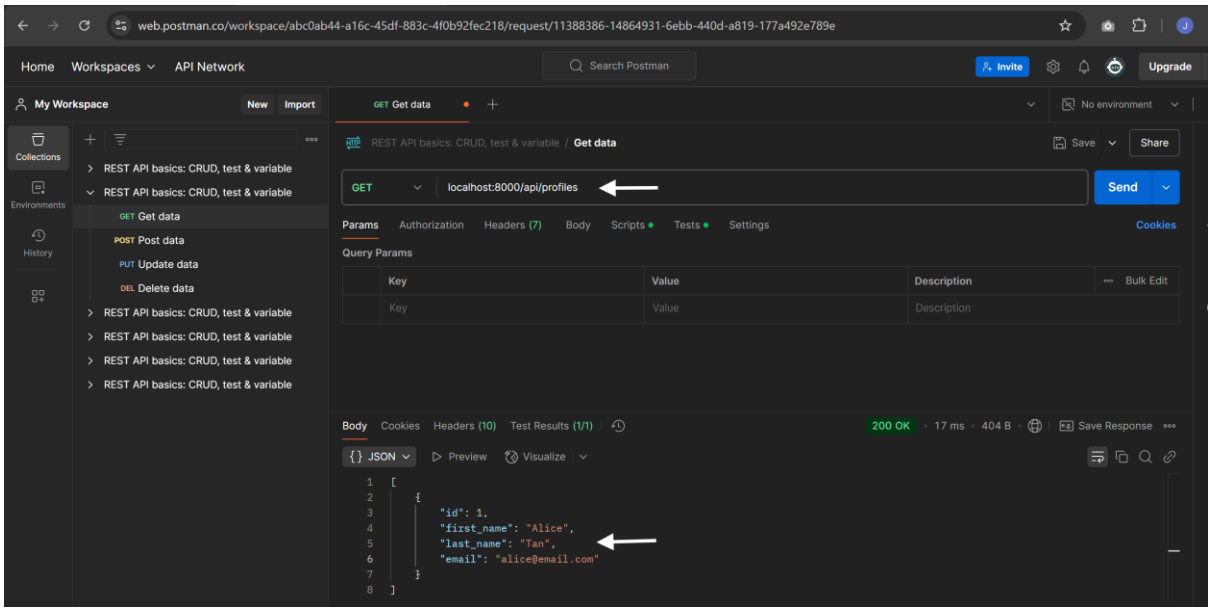
Select the profile based on id and make changes to the data to update the profile details.

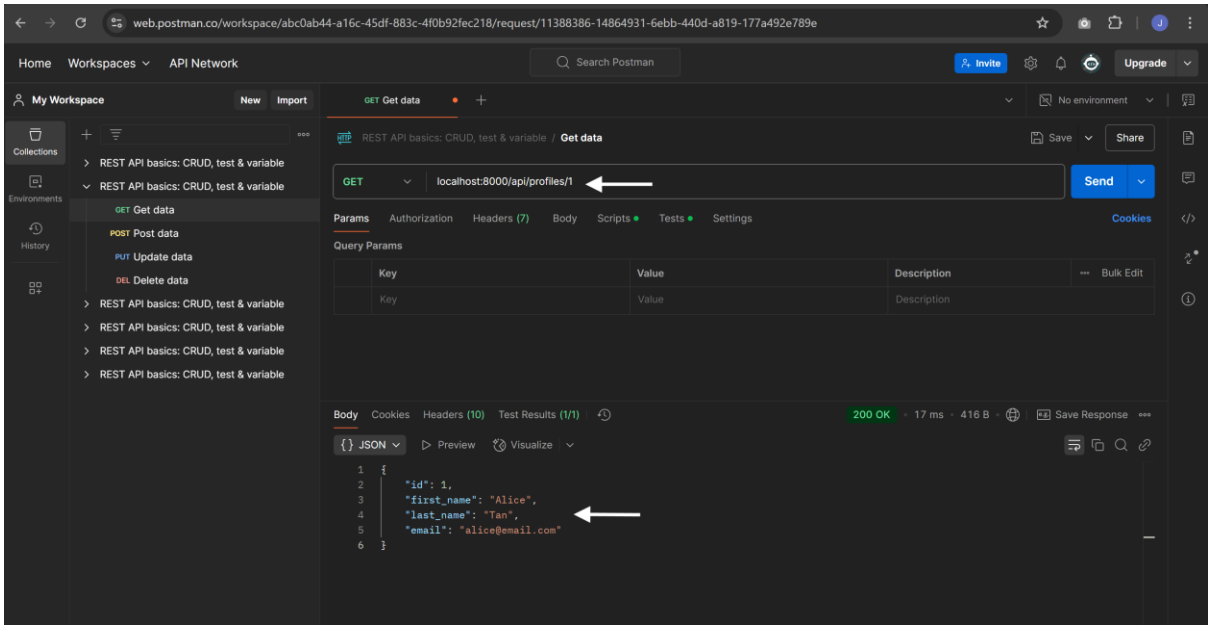


Select the profile based on id and delete the profile

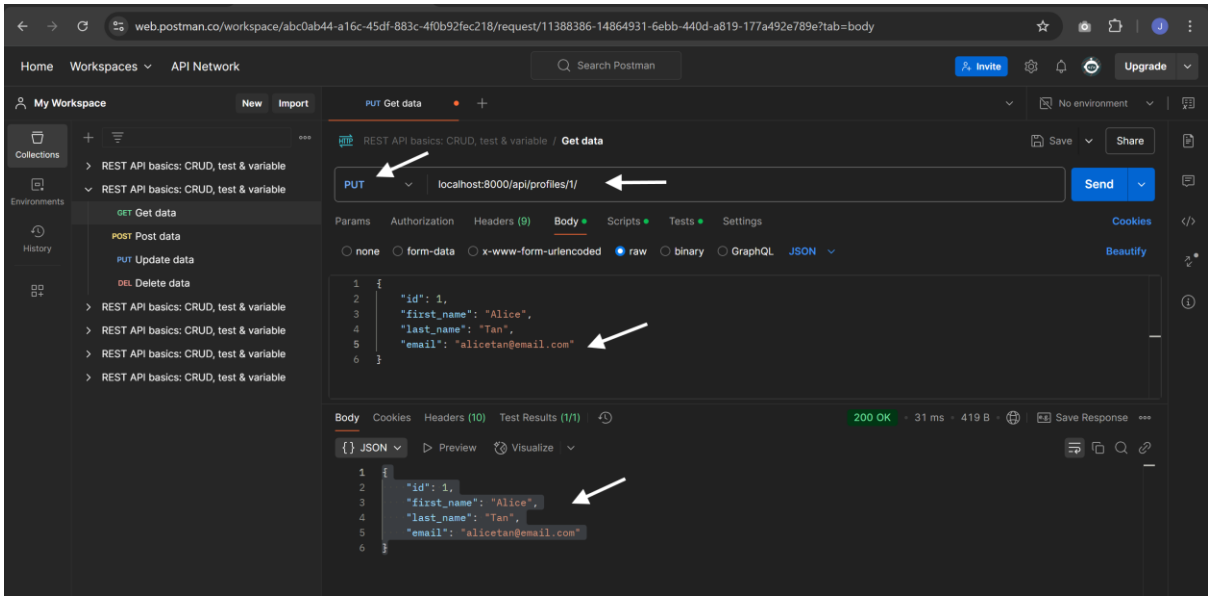


You can test the same using postman





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. APIViewSet vs Generics

Feature	APIViewSet	Generics
Abstraction Level	Higher (single class for all actions).	Moderate (separate class for each operation).
Router Requirement	Requires a router to map actions.	No router required; URLs are defined manually.
Ease of Use	Easier for CRUD operations.	Better for granular control.
Customization	Customizable by overriding methods.	Customizable by overriding methods.
Code Conciseness	More concise; all operations in one class.	Requires multiple classes for CRUD.
Use Case	Resource-based APIs with minimal boilerplate.	APIs needing granular control over operations.
Scalability	Scales well for large projects.	Suitable for smaller projects or specific use cases.

When to Use Which?

- **Use APIViewSet**:
    - When building resource-based APIs with standard CRUD operations.
    - When using DRF routers for automatic URL routing.
    - For larger projects where conciseness and maintainability are important.
- **Use Generics**:
    - When you need granular control over specific operations.
    - For smaller projects or when operations need significant customization.
    - When you do not want to use routers or prefer defining URL patterns manually.