Introduction

At BeautySalon, we offer a wide range of services and features to meet the needs of our community of beauty lovers.

Basic Requirements

Make sure you have the following packages installed:

- Python (latest version)
- Django (latest version)
- Django Rest Framework

Installation Steps

Configure Git: Set up your Git user name and email.

- `git config --global user.name "Your Name"`
- `git config --global user.email “youremail@example.com”`

Create a Local Repository: Initialize a local Git repository.

- `mkdir example-project`
- `cd example-project`
- `git init`

Dependencies and Virtual Environment

Creating a Virtual Environment: Create a virtual environment named `BeautySalon`.

```
python -m venv myprojectenv
```

Activate the virtual environment:

- On macOS/Linux: `source myprojectenv/bin/activate`
- On Windows: `myprojectenv\Scripts\activate`

Installing Django and Django Rest Framework: Install Django and DRF using pip.

- `pip install django`
- `pip install djangorestframework`

Creating a Django Project: Generate a new Django project.

- `django-admin startproject myproject`

Replace `BeautySalon` with your desired project name.

Running the Django Server

Navigate to your project directory and start the Django server.

- `cd BeautySalon`
- `pip install psycopg2`
- `pip install psycopg2-binary`
- `python manage.py runserver`

***pgadmin create database**

We move to the folder that has the project

*** python3 manage.py runserver**

In Django, an application is a web application that does something – a self-contained feature or functionality of your project. Think of it as a specific district or neighborhood within our city, dedicated to a particular purpose, like residential, commercial, or industrial. Create your first app with:

python manage.py startapp myapp

Replace myapp with a suitable name for your application. This command creates a new directory with your app's name, containing the basic files needed to start building your app.

Registering the App and DRF

Update the `INSTALLED_APPS` list in `settings.py` to include `'rest_framework'` and your app.

```
INSTALLED_APPS = [...,
    'rest_framework',
    'myapp',
    ...]
```

Configuring Django to Use PostgreSQL

Update the `DATABASES` settings in `settings.py` with your PostgreSQL credentials.

```
DATABASES = {
    'default': {
    'ENGINE': 'django.db.backends.postgresql',
    'NAME': 'beautysalondatabase',
    'USER': 'postgres',
    'PASSWORD': 'yourpassword',
    'HOST': 'localhost',
    'PORT': '5432',
  }
}
```

API Specification:

First, we need to define the endpoints that our API will have. This will depend on the specific functionality of your application, but here are some examples based on a blog system:

Data Models Design:

Now, let's create the data models in Django that will represent the main entities of our application. For a blog system, we could have the following models:

```python
from django.db import models
from django.contrib.auth.models import User


class Client(models.Model):
    name = models.CharField(max_length=100)
    phone = models.CharField(max_length=15)
```

```
    email = models.EmailField()


class Service(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    details = models.JSONField()
class Appointment(models.Model):
    client = models.ForeignKey(Client, on_delete=models.CASCADE)
    service = models.ForeignKey(Service, on_delete=models.CASCADE)
    date = models.DateField()
    time = models.TimeField()
    additional_info = models.JSONField(blank=True, null=True)
class Booking(models.Model):
    appointment = models.OneToOneField(Appointment,
on_delete=models.CASCADE)
    payment_status = models.BooleanField(default=False)
class UploadedFile(models.Model):
    file = models.FileField(upload_to='uploads/')
    uploaded_at = models.DateTimeField(auto_now_add=True)
```

Data Serializers Design:

```
from rest_framework import serializers
from .models import Client, Service, Appointment, Booking
from django.contrib.auth.models import User
from rest_framework import serializers
from .models import UploadedFile
```

```
class ClientSerializer(serializers.ModelSerializer):
    class Meta:
        model = Client
        fields = '__all__'
class ServiceSerializer(serializers.ModelSerializer):
    class Meta:
        model = Service
        fields = '__all__'
        extra_kwargs = {
            'url': {'view_name': 'service-detail', 'lookup_field': 'pk'}
        }
class AppointmentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Appointment
        fields = '__all__'
        extra_kwargs = {
            'url': {'view_name': 'client-detail', 'lookup_field': 'pk'}
        }


class BookingSerializer(serializers.ModelSerializer):
    class Meta:
        model = Booking
        fields = '__all__'


class UserSerializer(serializers.ModelSerializer):
    class Meta:
```

```
        model = User
        fields = ['id', 'username', 'email']
class UploadedFileSerializer(serializers.ModelSerializer):
    class Meta:
        model = UploadedFile
        fields = '__all__'
```

API_VIEW:

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from .models import Client, Service, Appointment
from .serializers import ClientSerializer, ServiceSerializer,
AppointmentSerializer
from django.contrib.auth.views import LoginView as BaseLoginView
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from .models import Client, Service, Appointment, Booking
from .serializers import ClientSerializer, ServiceSerializer,
AppointmentSerializer, BookingSerializer
from rest_framework.views import APIView
from rest_framework import generics
from django.contrib.auth.models import User
from django.contrib.auth.hashers import make_password
from rest_framework import serializers
from django.contrib.auth.models import User
from .serializers import UserSerializer
```

```
from rest_framework.permissions import IsAuthenticated

from rest_framework.parsers import FileUploadParser

from rest_framework.authtoken.views import ObtainAuthToken

from rest_framework.authtoken.models import Token


@api_view(['GET', 'POST'])

def client_list(request):

    if request.method == 'GET':

        clients = Client.objects.all()

        serializer = ClientSerializer(clients, many=True)

        return Response(serializer.data)

    elif request.method == 'POST':

        serializer = ClientSerializer(data=request.data)

        if serializer.is_valid():

            serializer.save()

            return Response(serializer.data,
status=status.HTTP_201_CREATED)

        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)


@api_view(['GET', 'PUT', 'DELETE'])

def client_detail(request, pk):

    try:

        client = Client.objects.get(pk=pk)

    except Client.DoesNotExist:

        return Response(status=status.HTTP_404_NOT_FOUND)


    if request.method == 'GET':
```

```
        serializer = ClientSerializer(client)
        return Response(serializer.data)
    elif request.method == 'PUT':
        serializer = ClientSerializer(client, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
    elif request.method == 'DELETE':
        client.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
@api_view(['GET', 'POST'])
def service_list(request):
    if request.method == 'GET':
        services = Service.objects.all()
        serializer = ServiceSerializer(services, many=True)
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = ServiceSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)


@api_view(['GET', 'PUT', 'DELETE'])
```

```
def service_detail(request, pk):
    try:
        service = Service.objects.get(pk=pk)
    except Service.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        serializer = ServiceSerializer(service)
        return Response(serializer.data)
    elif request.method == 'PUT':
        serializer = ServiceSerializer(service, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
    elif request.method == 'DELETE':
        service.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
@api_view(['GET', 'POST'])
def appointment_list(request):
    if request.method == 'GET':
        appointments = Appointment.objects.all()
        serializer = AppointmentSerializer(appointments, many=True)
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = AppointmentSerializer(data=request.data)
```

```
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)


@api_view(['GET', 'PUT', 'DELETE'])
def appointment_detail(request, pk):
    try:
        appointment = Appointment.objects.get(pk=pk)
    except Appointment.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)


    if request.method == 'GET':
        serializer = AppointmentSerializer(appointment)
        return Response(serializer.data)
    elif request.method == 'PUT':
        serializer = AppointmentSerializer(appointment, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
    elif request.method == 'DELETE':
        appointment.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

```python
@api_view(['GET', 'POST'])
def booking_list(request):
    if request.method == 'GET':
        bookings = Booking.objects.all()
        serializer = BookingSerializer(bookings, many=True)
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = BookingSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)


@api_view(['GET', 'PUT', 'DELETE'])
def booking_detail(request, pk):
    try:
        booking = Booking.objects.get(pk=pk)
    except Booking.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)


    if request.method == 'GET':
        serializer = BookingSerializer(booking)
        return Response(serializer.data)
    elif request.method == 'PUT':
        serializer = BookingSerializer(booking, data=request.data)
        if serializer.is_valid():
```

```
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
    elif request.method == 'DELETE':
        booking.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
class AppointmentCreate(APIView):
    def post(self, request):
        serializer = AppointmentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
class UserCreate(generics.CreateAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer

    def create(self, request, *args, **kwargs):
        password = make_password(request.data.get('password'))
        request.data['password'] = password
        return super().create(request, *args, **kwargs)

class ProtectedView(APIView):
    permission_classes = [IsAuthenticated]
```

```
    def get(self, request):
        content = {'message': 'This is a protected endpoint'}
        return Response(content)


class FileUploadView(APIView):
    parser_classes = [FileUploadParser]
    permission_classes = [IsAuthenticated]

    def post(self, request, *args, **kwargs):
        file_obj = request.data['file']
        # Handle the uploaded file
        return Response({'status': 'File uploaded successfully'})


class LoginView(ObtainAuthToken):
    def post(self, request, *args, **kwargs):
        serializer = self.serializer_class(data=request.data,
context={'request': request})
        serializer.is_valid(raise_exception=True)
        user = serializer.validated_data['user']
        token, created = Token.objects.get_or_create(user=user)
        return Response({'token': token.key})
```

APP URLS:

```
from django.urls import path
from django.contrib import admin
from rest_framework.authtoken.views import obtain_auth_token
from drf_yasg.views import get_schema_view
from drf_yasg import openapi
from rest_framework import permissions
```

```
from . import views
from .views import AppointmentCreate, UserCreate, ProtectedView,
FileUploadView, LoginView
from rest_framework.authtoken.views import obtain_auth_token


# Schema view for API documentation
schema_view = get_schema_view(
    openapi.Info(
        title="My Awesome API",
        default_version='v1',
        description="Descripción de mi API",
    ),
    public=True,
    permission_classes=[permissions.AllowAny],
)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('clients/', views.client_list, name='client-list'),
     path('api/token/', obtain_auth_token, name='obtain_auth_token'),
    path('clients/<int:pk>/', views.client_detail, name='client-detail'),
    path('services/', views.service_list, name='service-list'),
    path('services/<int:pk>/', views.service_detail,
name='service-detail'),
    path('appointments/', views.appointment_list,
name='appointment-list'),
    path('appointments/<int:pk>/', views.appointment_detail,
name='appointment-detail'),
    path('api/appointments/create/', AppointmentCreate.as_view(),
name='create-appointment'),
    path('api/users/register/', UserCreate.as_view(), name='user-create'),
    path('swagger/schema/', schema_view.without_ui(cache_timeout=0),
name='schema-json'),
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0),
name='schema-swagger-ui'),
    path('api-docs/', schema_view.with_ui('swagger', cache_timeout=0),
name='schema-swagger-ui'),
    path('api/login/', obtain_auth_token, name='login'),
    path('protected/', ProtectedView.as_view(), name='protected'),
```

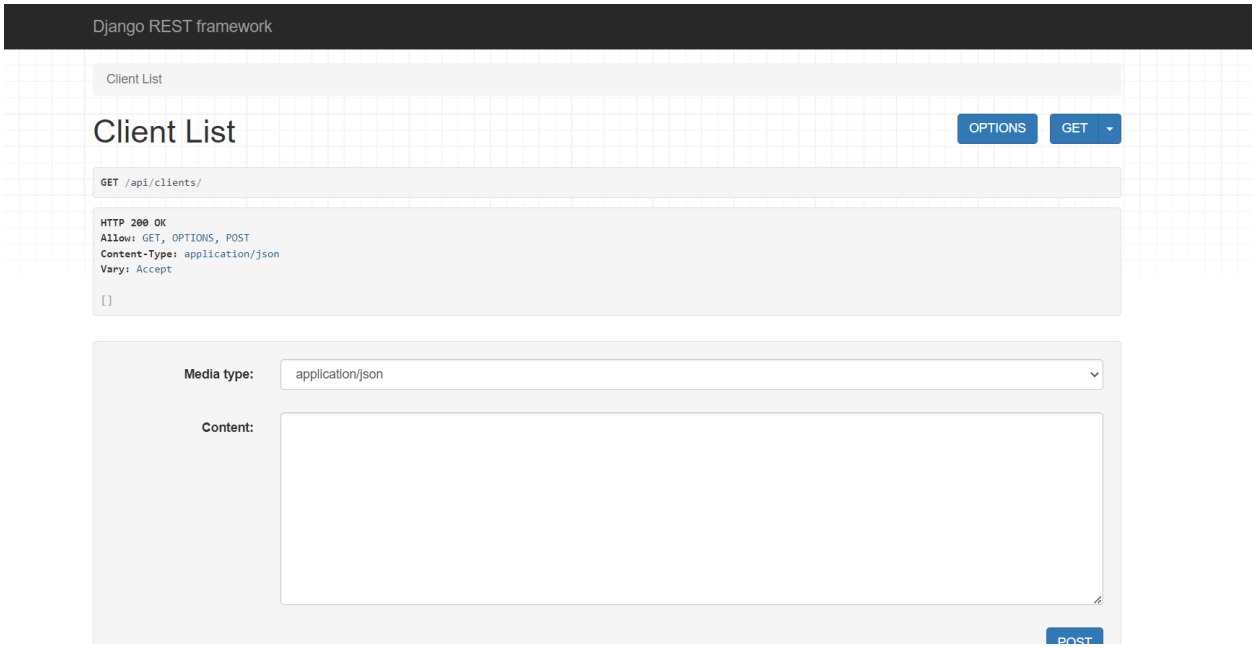
```
    path('upload/', FileUploadView.as_view(), name='file-upload'),
    path('accounts/login/', LoginView.as_view(), name='custom_login'),
]
```

RAIZ URLS:

```
from django.contrib import admin

from django.urls import path, include

from django.contrib.auth import views as auth_views


urlpatterns = [

    path('admin/', admin.site.urls),

    path('api/', include('BeautySalonapp.urls')),

    path('accounts/login/',
auth_views.LoginView.as_view(template_name='registration/login.html'),
name='login'),

]
```

What django would look like:

http://localhost:8000/api/posts/



---

use Insomnia:

In the view serializer url, the authentication and login configuration is already found.

First, you need a token. Assuming you have set up a token authentication system and a user in your Django application, you can obtain a token by making a POST request to your login or token obtain endpoint. This request will include the username and password of your user.

- **Create a New Request** in Insomnia.
- **Set the Method** to POST.
- **Enter the URL** for your token obtain endpoint (`http://localhost:8000/api/login/`).

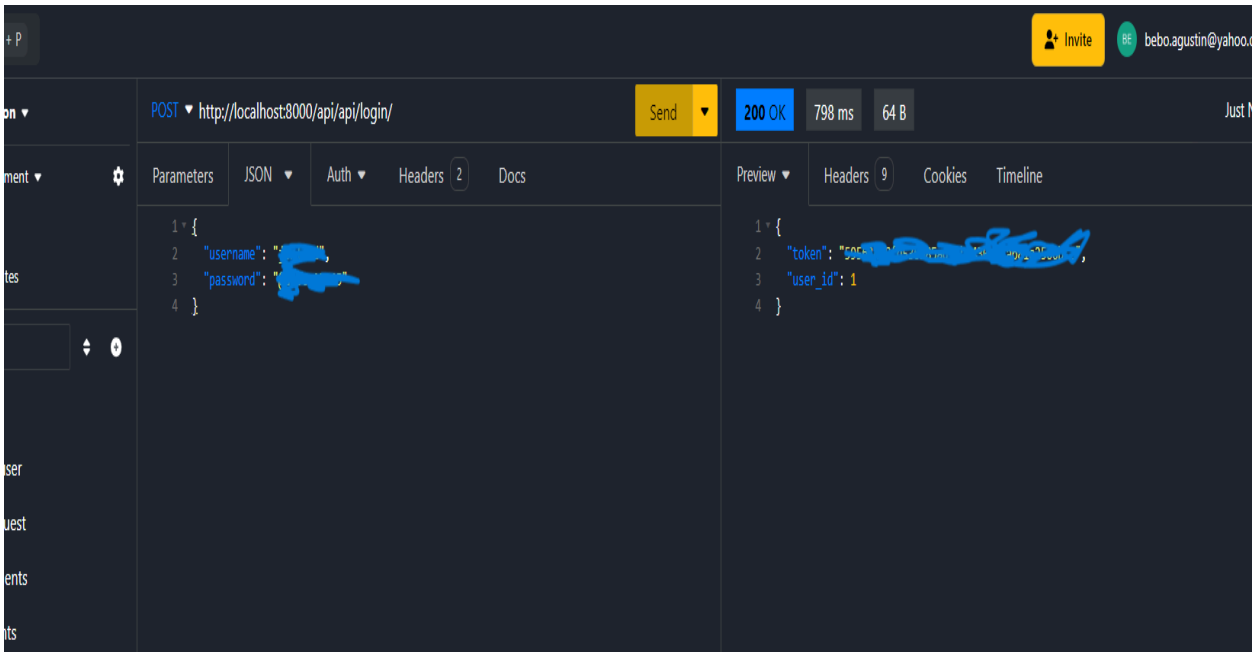
Set the Body to JSON and include your username and password:

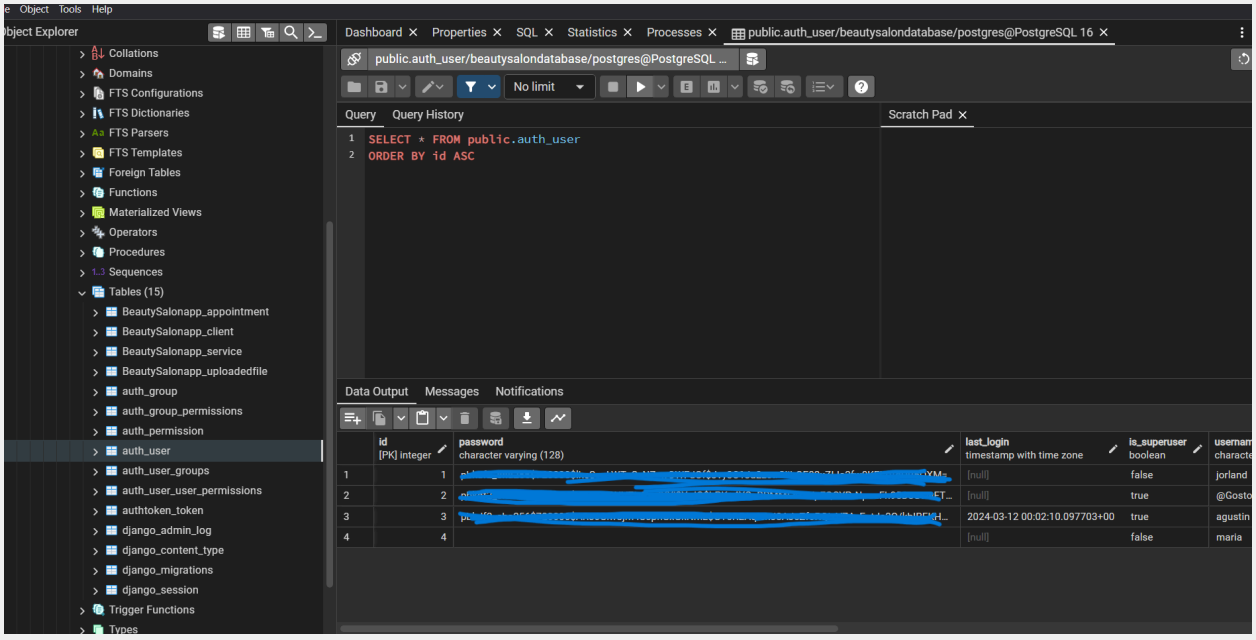
```
{
  "username": "your_username",
  "password": "your_password"
}
```

- **Send the Request**. You should receive a response that includes the token.

Now, with your token, you can make an authenticated request to the protected route.

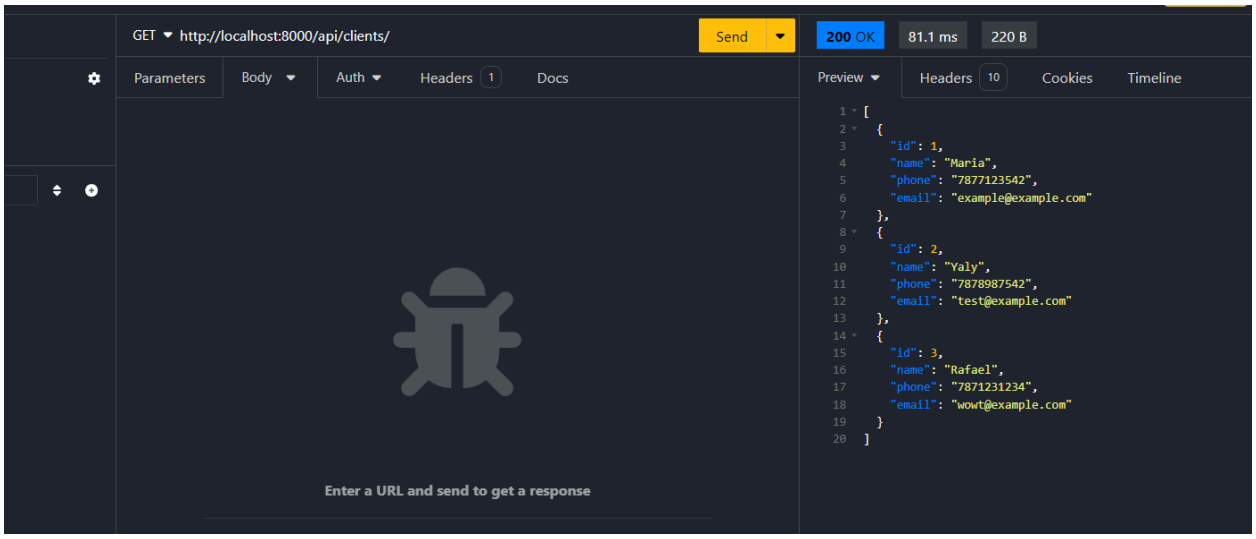
- **Create a New Request** in Insomnia for the protected route.
- **Set the Method** to GET.
- **Enter the URL** for the protected route (e.g., `http://localhost:8000/protected/`).
- **Add a Header** named `Authorization` with the value `Token your_token_here`, replacing `your_token_here` with the actual token you received from the login request.
    - It should look like this: `Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b`
- **Send the Request**. Since you are authenticated, yo

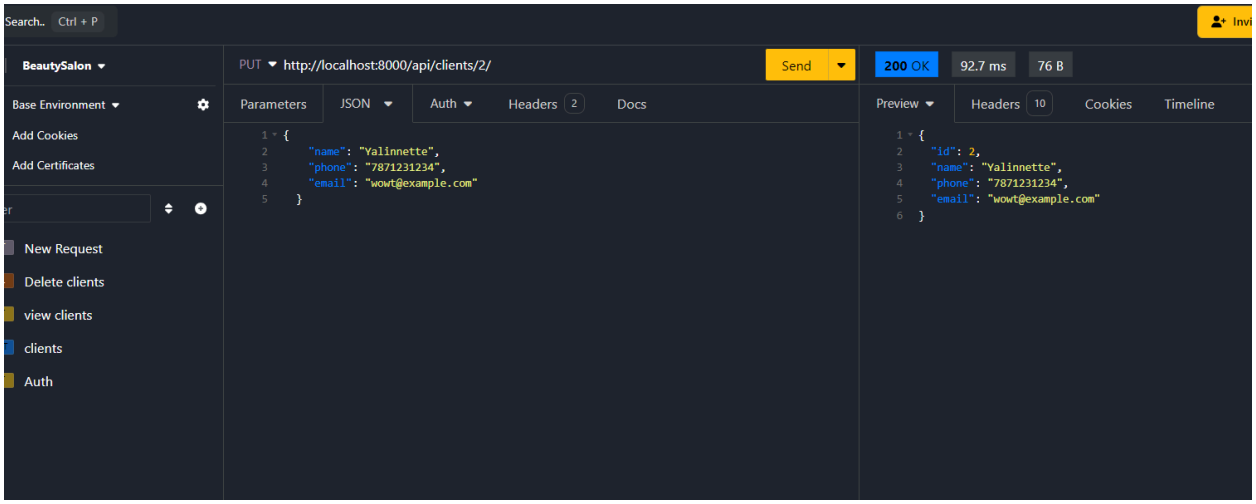




Adding data

1. GET - shows me the added information
2. POST - post the information we want
3. DELETE - deletes the information we want
4. PUT - updates the information we want





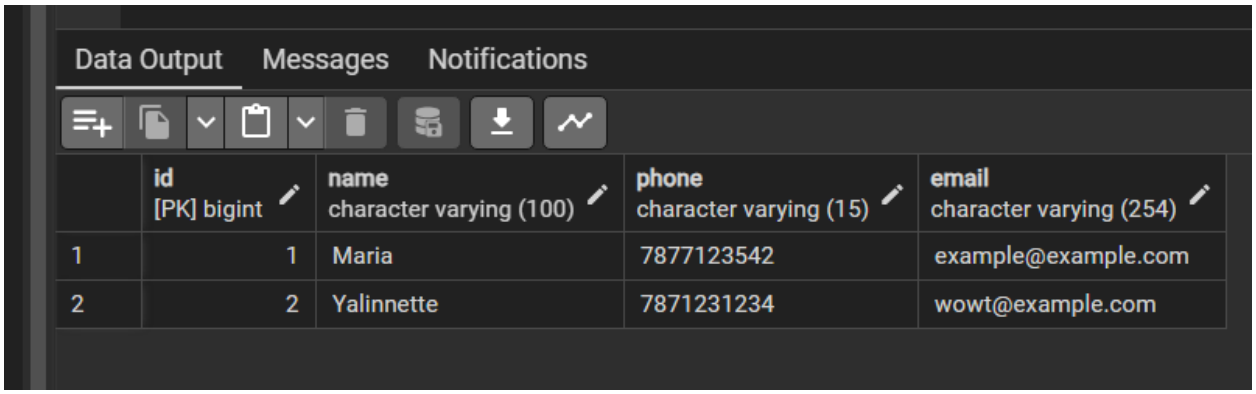
Here we are adding clients with POST we add them, and with GET we can visualize the clients. and with DELETE eliminate us.

```
post http://localhost:8000/api/clients/

{

"name": "Yalinnette",

"phone": "7879899898",

"email": "example@example.com"

}


get http://localhost:8000/api/clients/

{

"name": "Yalinnette",

"phone": "7879899898",

"email": "example@example.com"

}


{

"name": "Maria",

"phone": "7879890008",

"email": "example@example.com"

}
```

delete http://localhost:8000/api/clients/2/

we delete that client

With the PUT we update information.



	id [PK] bigint	name character varying (100)	phone character varying (15)	email character varying (254)
1	1	Maria	7877123542	example@example.com
2	2	Yalinnette	7871231234	wowt@example.com

AWS

Before integrating AWS S3 with your Django project, you need to set up an S3 bucket:

- **Create an AWS Account**: Sign up or log in to your AWS account at aws.amazon.com.
- **Create an S3 Bucket**: In the AWS Management Console, navigate to S3 and create a new bucket. Ensure the bucket name is unique across AWS, and decide on the region closest to your users for optimal performance.
- **Bucket Settings**: Disable 'Block all public access' settings if you want your files to be publicly accessible. Be cautious with this setting and only make public the files that need to be accessible to the end-users.
- **Get Your AWS Access Keys**: Navigate to your account's security credentials page and create a new access key under the Access Keys (Access Key ID and Secret Access Key) section. Note these down securely.

To connect your Django project with AWS S3, you'll use the `django-storages` package and configure your project's settings:

- **Install django-storages and boto3**: Run `pip install django-storages boto3` in your terminal. `boto3` is the Amazon Web Services (AWS) SDK for Python, which allows Python developers to write software that uses services like Amazon S3.
- **Update `settings.py`**: Add `storages` to your `INSTALLED_APPS`. Then, configure the following settings with your AWS credentials and bucket name:

```
AWS_ACCESS_KEY_ID = 'asd1234567890wertyuiopasdfghjkl'
AWS_SECRET_ACCESS_KEY =
'1111111111111111111111111/222222222222222222'
AWS_STORAGE_BUCKET_NAME = 'rafrodriguezbucket'
AWS_S3_FILE_OVERWRITE = False
AWS_DEFAULT_ACL = 'public-read'
AWS_S3_REGION_NAME = 'us-east-2'
AWS_S3_CUSTOM_DOMAIN = f'{AWS_STORAGE_BUCKET_NAME}.s3.amazonaws.com'

# Static files configuration
STATIC_URL = f'https://{AWS_S3_CUSTOM_DOMAIN}/static/'

# Media files configuration
DEFAULT_FILE_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'
MEDIA_URL = f'https://{AWS_S3_CUSTOM_DOMAIN}/media/'
```

To allow users to upload files, create a model to store file references, serialize it, and create a view:

Model: Define a model in `models.py` that includes a `FileField` or `ImageField`.

```
class UploadedFile(models.Model):
    file = models.FileField(upload_to='uploads/')
    uploaded_at = models.DateTimeField(auto_now_add=True)
```

Serializer: Create a serializer in `serializers.py` for the model.

```
class UploadedFileSerializer(serializers.ModelSerializer):
    class Meta:
        model = UploadedFile
        fields = '__all__'
```

View: In `views.py`, implement a view to handle file uploads.

```
class FileUploadView(APIView):
    parser_classes = [FileUploadParser]
    permission_classes = [IsAuthenticated]
```

```
    def post(self, request, *args, **kwargs):

        file_obj = request.data['file']

        # Handle the uploaded file

        return Response({'status': 'File uploaded successfully'})
```

- This view uses `MultiPartParser` and `FormParser` to handle form data, including files.

URLs: Add a URL pattern in `urls.py` to route to the upload view.

use command

1. python manage.py makemigrations
2. python manage.py migrate

Install

1. pip install "uvicorn[standard]"
2. pip install wsproto
3. pip install channels

adding nested serializer and hyperlinkedmodelserializer

```
class ServiceSerializer(serializers.ModelSerializer):

    class Meta:

        model = Service

        fields = '__all__'

        extra_kwargs = {

            'url': {'view_name': 'service-detail', 'lookup_field': 'pk'}

        }


class AppointmentSerializer(serializers.ModelSerializer):

    class Meta:

        model = Appointment

        fields = '__all__'

        extra_kwargs = {
```

```
            'url': {'view_name': 'client-detail', 'lookup_field': 'pk'}
        }


class BookingSerializer(serializers.ModelSerializer):
```

Implementing Throttling in Django REST Framework (DRF)

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ],
   'DEFAULT_FILTER_BACKENDS': [
        'django_filters.rest_framework.DjangoFilterBackend'
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day',
        'user': '1000/day'
    }
}
from rest_framework.throttling import AnonRateThrottle, UserRateThrottle


class ExampleView(APIView):
    throttle_classes = [AnonRateThrottle, UserRateThrottle]
    def get(self, request, format=None):
        # Ejemplo de uso de los serializers
```

```
        clients = Client.objects.all()
        client_serializer = ClientSerializer(clients, many=True)

        services = Service.objects.all()
        service_serializer = ServiceSerializer(services, many=True)

        appointments = Appointment.objects.all()
        appointment_serializer = AppointmentSerializer(appointments,
many=True)

        response_data = {
            'clients': client_serializer.data,
            'services': service_serializer.data,
            'appointments': appointment_serializer.data,
            'message': 'Hello, world!'
        }
        return Response(response_data)


   'DEFAULT_FILTER_BACKENDS': [
        'django_filters.rest_framework.DjangoFilterBackend'
    ],
```

Filtered out:

```
  from django_filters.rest_framework import DjangoFilterBackend from
  rest_framework import filters
```

```
class ClientViewSet(ModelViewSet): queryset = Client.objects.all()
serializer_class = ClientSerializer filter_backends =
[DjangoFilterBackend] filterset_fields = ['name', 'email']
```

Search:

```
class ClientViewSet(ModelViewSet): ... filter_backends =
[filters.SearchFilter] search_fields = ['name', 'email']
```

Ordination:

```
class ClientViewSet(ModelViewSet): ... filter_backends = [filters
```

```
Create permissions.py

from rest_framework.permissions import BasePermission



class CanViewClients(BasePermission):

    def has_permission(self, request, view):

        # Verifica si el usuario puede ver todos los clientes

        return request.user.has_perm('yourapp.view_client')



class CanAddService(BasePermission):

    def has_permission(self, request, view):

        # Verifica si el usuario puede agregar un nuevo servicio

        return request.user.has_perm('yourapp.add_service')
```

View.py

```
class ServiceCreateView(APIView):

    permission_classes = [CanAddService]



    def post(self, request):

        if not request.user.has_perm('yourapp.add_service'):

            return Response({"message": "No tienes permiso para agregar servicios"}, status=status.HTTP_403_FORBIDDEN)



        return Response({"message": "Servicio creado exitosamente"}, status=status.HTTP_201_CREATED)
```

throttling and permissions in your views, you must add the code that defines the MyThrottle class and the MyPermission class to your permissions and throttling files, respectively. Then, in your views where you want to apply these constraints, you import these classes and assign them to the corresponding views.

First, let's define the permissions and throttling classes in their respective files:

1. **Throttling (throttling.py):**

python

```
from rest_framework.throttling import UserRateThrottle

class MyThrottle(UserRateThrottle):

    rate = '5/minute'
```

2. **Permisos (permissions.py):**

```
python

from rest_framework.permissions import IsAuthenticated

class MyPermission(IsAuthenticated):

    pass

View

from rest_framework.views import APIView

from rest_framework.response import Response

from rest_framework import status

from .throttling import MyThrottle

from .permissions import MyPermission

class MyProtectedView(APIView):

    throttle_classes = [MyThrottle]  # Aplica throttling a esta vista

    permission_classes = [MyPermission]  # Requiere autenticación para
acceder

    def get(self, request):

        # Lógica de la vista

        return Response({"message": "Bienvenido a la vista protegida"},
status=status.HTTP_200_OK)

To use pagination in your views, simply follow these steps:
```

Define the pagination class: First, define a pagination class that inherits from PageNumberPagination in your views file or in a separate file.

```
# pagination.py

from rest_framework.pagination import PageNumberPagination

class MyPagination(PageNumberPagination):

    page_size = 10 # Number of objects per page

    page_size_query_param = 'page_size' # Parameter to specify the page size

    max_page_size = 1000 # Maximum page size allowed
```

Assign the pagination class to your views: In your views where you want to apply pagination, assign the pagination class that you have defined.

```
# views.py

from rest_framework.views import APIView

from rest_framework.response import Response

from rest_framework import status

from .pagination import MyPagination
```

```
class MyListView(APIView):

      pagination_class = MyPagination



      def get(self, request):

      # Get the queryset of the objects you want to page

      queryset = YourModel.objects.all()



      # Paginate the results

      paginator = self.pagination_class()

      paginated_queryset = paginator.paginate_queryset(queryset, request)



      # Serialize paged objects

      serializer = YourSerializer(paginated_queryset, many=True)



      # Return the paginated response

      return paginator.get_paginated_response(serializer.data)
```

Configure the pagination response: The pagination class will take care of splitting the results into pages and providing metadata in the response. You just need to serialize the paginated objects and return the paginated response using paginator.get_paginated_response(serializer.data)