Django Rest Framework

Notes By

Naveen

django REST framework

Django Rest Framework (**DRF**)

Django REST framework is a powerful and flexible toolkit for building Web APIs.

Django REST framework is popular because:

- Its Web-browsable API is a huge usability win for your developers.
- Authentication policies include packages for **OAuth1** and **OAuth2**.
- Serialization supports both ORM and non-ORM data sources.
- It's customizable all the way down. Just use regular function-based views if you don't need the more powerful features.
- It has extensive documentation and great community support.
- It's used and trusted by internationally recognized companies including Mozilla, Red Hat, Heroku, and Eventbrite.

Requirements

REST framework requires the following:

- Python (3.5, 3.6, 3.7, 3.8)
- Django (1.11, 2.0, 2.1, 2.2, 3.0)

Clone the project from github.

git clone https://github.com/encode/django-rest-framework

Installation

pip install djangorestframework

pip install markdown # Markdown support for the browsable API.

pip install django-filter # Filtering support

Adding DRF to Django application

1) Add 'rest_framework' to INSTALLED_APPS in **settings.py** module.

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

2) Add a route into path function in urls.py

```
from django.urls import path
from django.urls import include
urlpatterns = [
    path('rest_example/',include('rest_framework.urls'))
]
```

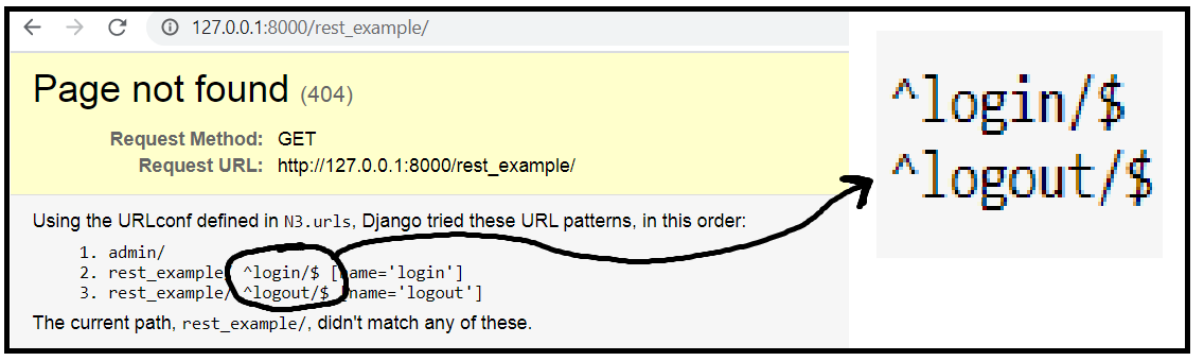
3) Open terminal and do makemigrations and migrate

4) Open terminal and create a super-user with providing username and password.

5) Once the Super-user created successfully start the Server

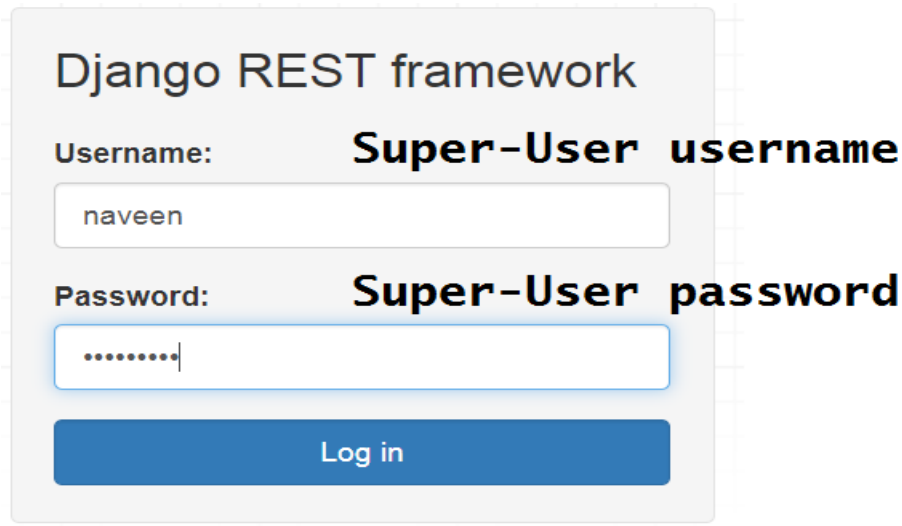
6) Open Browser and type url as
http://127.0.0.1:8000/rest_example/

Output

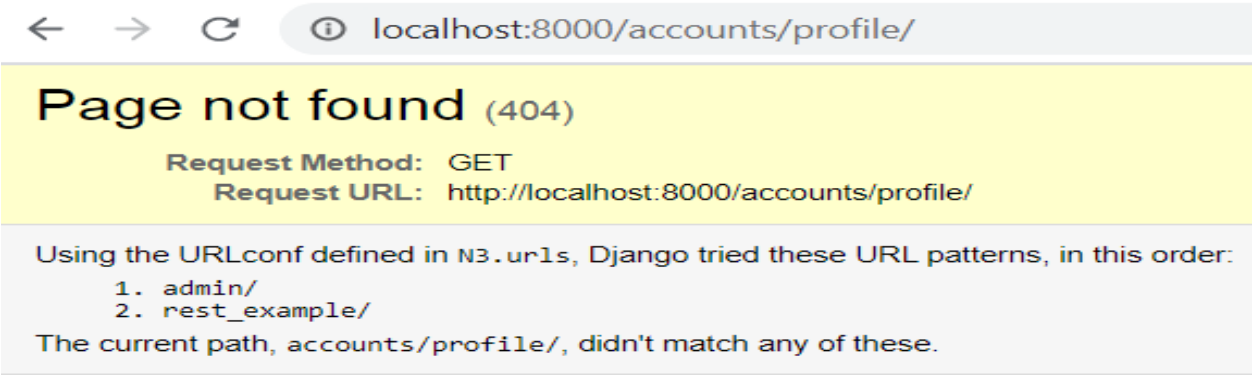


Change Url as : http://127.0.0.1:8000/rest_example/login/

Output



Note: We need to provide LOGIN_REDIRECT_URL in **settings.py** then it will open Your redirected Page else it will the output will be



Requests

If you're doing REST-based web service stuff ... you should ignore request.POST.

— Malcom Tredinnick, Django developers group

REST framework's **Request** class extends the standard **HttpRequest**, adding support for REST framework's flexible request parsing and request authentication.

Request parsing

REST framework's Request objects provide flexible request parsing that allows you to treat requests with JSON data or other media types in the same way that you would normally deal with form data.

.data

request.data returns the parsed content of the request body. This is similar to the standard **request.POST** and **request.FILES** attributes except that.

.query_params

request.query_params is a more correctly named synonym for **request.GET**.

For clarity inside your code, we recommend using **request.query_params** instead of the Django's standard **request.GET**.

Doing so will help keep your codebase more correct and obvious - any HTTP method type may include query parameters, not just GET requests.

Responses

Unlike basic HttpResponse objects, TemplateResponse objects retain the details of the context that was provided by the view to compute the response. The final output of the response is not computed until it is needed, later in the response process.

REST framework supports HTTP content negotiation by providing a Response class which allows you to return content that can be rendered into multiple content types, depending on the client request.

Response()

Signature: Response(data, status=None, template_name=None, headers=None, content_type=None)

Unlike regular **HttpResponse** objects, you do not instantiate Response objects with rendered content. Instead you pass in un-rendered data, which may consist of any Python primitives.

Arguments:

- **data:** The serialized data for the response.
- **status:** A status code for the response. Defaults to 200.
- **template_name:** A template name to use if HTMLRenderer is selected.
- **headers:** A dictionary of HTTP headers to use in the response.
- **content_type:** The content type of the response. Typically, this will be set automatically.

Serializers

Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into **JSON, XML** or other content types.

Serializers also provide **deserialization**, allowing parsed data to be converted back into complex types, after first validating the incoming data.

The **serializers** in **REST framework** work very similarly to Django's **Form** and **ModelForm** classes.

Declaring Serializers

Step1 : Create a model class

```
from django.db import models
class ProductModel(models.Model):
    no = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=30,unique=True)
    price = models.FloatField()
    quantity = models.IntegerField()
```

Step2 : Create a new python file in "app" and name it as **"serializers"** and define the serializer class.

Step3 : Your class must Inherit rest framework Serializer class.

```
from rest_framework import serializers
class ProductSerializer(serializers.Serializer):
    no = serializers.IntegerField()
    name = serializers.CharField(max_length=30)
```

```
price = serializers.FloatField()
quantity = serializers.IntegerField()
```

Note : we can give any name to the python file.

Parsers

REST framework includes a number of built in Parser classes, that allow you to accept requests with various media types.

Note: When developing client applications always remember to make sure you're setting the Content-Type header when sending data in an HTTP request.

If you don't set the content type, most clients will default to using 'application/x-www-form-urlencoded', which may not be what you wanted.

As an example, if you are sending json encoded data using jQuery with the .ajax() method, you should make sure to include the contentType: 'application/json' setting.

You can also set the parsers used for an individual view, or viewset, using the APIView class-based views.

```
from rest_framework.parsers import JSONParser

from rest_framework.views import APIView

class ExampleView(APIView):

    """ A view that can accept POST requests with JSON content.  """

    parser_classes = [JSONParser]

    def post(self, request, format=None):        ...............
```

JSONParser : It is a class which Parses JSON-serialized data.

.media_type : application/json

parse(self, stream):

It is a instance method of **JSONParser** class.
This method will Parses the incoming bytestream as JSON(dictionaty) and returns the resulting data.

1) Example program in insert the product into database using rest framework serializer. (Using post)

1) To insert data into database we need to define "create" method into serializer class.

```
from app3.models import ProductModel
class ProductSerializer(serializers.Serializer):
    ................
    def create(self, validated_data):
        return ProductModel.objects.create(**validated_data)
```

2) Defining post method in view class.

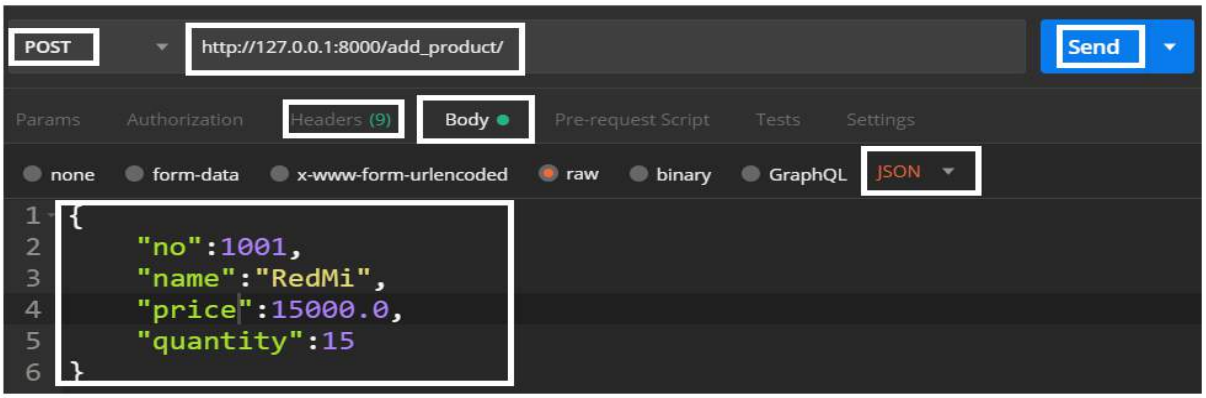
```
from django.http import HttpResponse
from django.views.generic import View
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser
from app3.serializers import ProductSerializer
import io

class ProductOperations(View):
    def post(self,request):
        byte_data = request.body # Will get data in bytes
```

```
    stm = io.BytesIO(byte_data) # converting bytes into streamed data
    dict_data = JSONParser().parse(stm) # Converting streamed data into dictionary
    ps = ProductSerializer(data=dict_data)
    if ps.is_valid():
        ps.save()
        message = {"message":"Product is Saved"}

else:
        message = {"errors":ps.errors}
    json_data = JSONRenderer().render(message)
    return HttpResponse(json_data,content_type= "application /json")
```

3) urls.py

```
path('add_product/',csrf_exempt(views.ProductOperations.as_view( ))),
```

4) Input using Postman



Result in Postman

2) Example program to read all Product details from database using rest framework serializer. (Using get)

1) serializers.py

```
from rest_framework import serializers
from app3.models import ProductModel

class ProductSerializer(serializers.Serializer):
    no = serializers.IntegerField()
    name = serializers.CharField(max_length=30)
    price = serializers.FloatField()
    quantity = serializers.IntegerField()
```

2) views.py

```
from django.http import HttpResponse
from django.views.generic import View
from rest_framework.renderers import JSONRenderer
from app3.serializers import ProductSerializer
from app3.models import ProductModel

class ProductOperations(View):
    def get(self,request):
        qs = ProductModel.objects.all()
        ps = ProductSerializer(qs,many=True)
        json_data = JSONRenderer().render(ps.data)
        return HttpResponse(json_data,content_type="application/
json")
```

3) urls.py

path('**view_products/**',csrf_exempt(views.ProductOperations.as_view())),

4) Output

GET http://127.0.0.1:8000/view_products

Pretty Raw Preview Visualize JSON


```
[
    {
        "no": 1001,
        "name": "RedMi",
        "price": 15000.0,
        "quantity": 15
    }
]
```


ProductSerializer(qs,many=True)

By setting **many=True** you tell DRF that **queryset** contains **"mutiple items"** (a list of items) so DRF needs to serialize each item with serializer class (and **serializer.data** will be a **list**)

If you don't set this argument it means **queryset** is a **"single instance"** and **serializer.data** will be a single object).

JSONRenderer().render(ps.data)

render() is an instance method of **JSONRenderer** class which will convert Serialized data into Json data.

3) Example program to read 1 Product data from database using rest framework serializer. (Using get)

1) serializers.py

```
from rest_framework import serializers
from app3.models import ProductModel

class ProductSerializer(serializers.Serializer):
    no = serializers.IntegerField()
    name = serializers.CharField(max_length=30)
    price = serializers.FloatField()
    quantity = serializers.IntegerField()
```

2) views.py

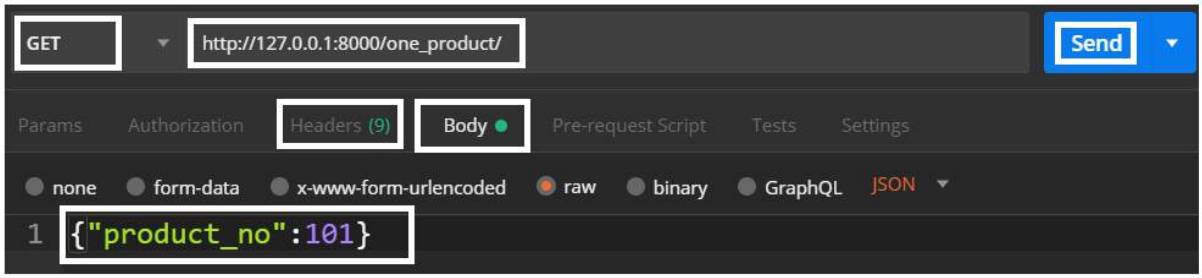
```
class ReadOneProduct(View):
    def get(self,request):
        b_data = request.body
        strm = io.BytesIO(b_data)
        d1 = JSONParser().parse(strm)
        product_no = d1.get("product_no",None)
        if product_no:
            try:
                res = ProductModel.objects.get(no=product_no)
                ps = ProductSerializer(res)
                json_data = JSONRenderer().render(ps.data)
            except ProductModel.DoesNotExist:
                message = {"error": "Invalid Product No"}
                json_data = JSONRenderer().render(message)
            return
```

```
HttpResponse(json_data,content_type="application/json")
    else:
        message = {"error": "Invalid Product No"}
        json_data = JSONRenderer().render(message)
        return HttpResponse(json_data,
content_type="application/json")
```

3) urls.py

```
path('one_product/',csrf_exempt(views.ReadOneProduct.as_view())
)
```

4) Input using Postman



5) Output

```
{
    "error": "Invalid Product No"
}
```

6) Input : `{"product_no":1001}`

7) Output :

```
{
    "no": 1001,
    "name": "RedMi",
    "price": 15000.0,
    "quantity": 15
}
```

4) Example program to read 1 or all Product details from database using rest framework serializer. (Using get)

Note: in this example we are using one url for 1 or all product details

1) serializers.py

```
from rest_framework import serializers
from app3.models import ProductModel

class ProductSerializer(serializers.Serializer):
    no = serializers.IntegerField()
    name = serializers.CharField(max_length=30)
    price = serializers.FloatField()
    quantity = serializers.IntegerField()
```

2) views.py

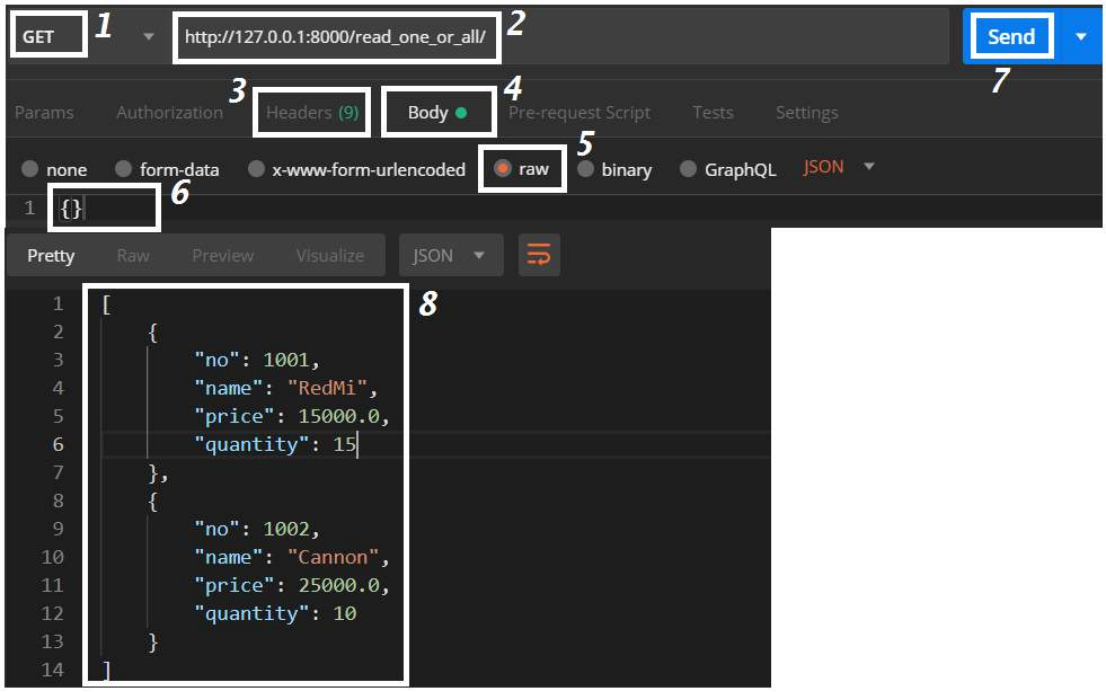
```
class Read1OrAll(View):
    def get(self,request):
        b_data = request.body
        strm = io.BytesIO(b_data)
        d1 = JSONParser().parse(strm)
        product_no = d1.get("product_no", None)
        if product_no:
            try:
                res = ProductModel.objects.get(no=product_no)
                ps = ProductSerializer(res)
                json_data = JSONRenderer().render(ps.data)
            except ProductModel.DoesNotExist:
                message = {"error":"Invalid Product No"}
```

```
            json_data = JSONRenderer().render(message)
        else:
            res = ProductModel.objects.all()
            ps = ProductSerializer(res,many=True)
            json_data = JSONRenderer().render(ps.data)

        return HttpResponse(json_data,content_type="application /json")
```

3) urls.py

```
path('read_one_or_all/',csrf_exempt(views.Read1OrAll.as_view())),
```

4) Using PostMan sending empty dict. ----- ({})



5) Input : `{"product_no":1001}`

6) Output :

```
{
    "no": 1001,
    "name": "RedMi",
    "price": 15000.0,
    "quantity": 15
}
```

5) Example program to Update 1 Product data using rest framework serializer. (Using update)

1) serializers.py

```
from rest_framework import serializers
from app3.models import ProductModel

class ProductSerializer(serializers.Serializer):
    no = serializers.IntegerField()
    name = serializers.CharField(max_length=30)
    price = serializers.FloatField()
    quantity = serializers.IntegerField()

    def create(self, validated_data):
        return ProductModel.objects.create(**validated_data)

    def update(self, instance, validated_data):
        instance.no = validated_data.get("no",instance.no)
        instance.name = validated_data.get("name",instance.name)
        instance.price = validated_data.get("price",instance.price)
        instance.quantity = validated_data.get("quantity",
                                                    instance.quantity)
        instance.save()
        return instance
```

2) urls.py

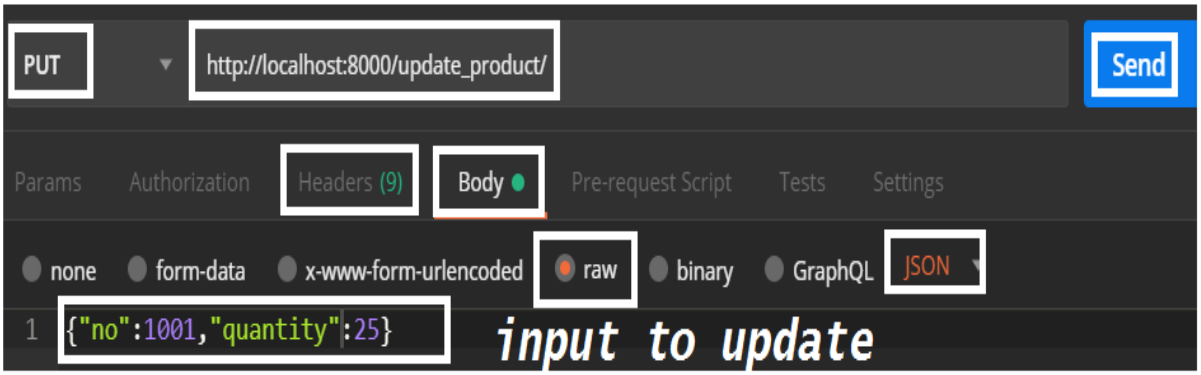
```
path('update_product/',csrf_exempt(views.UpdateProduct.as_view(
))),
```

3) views.py

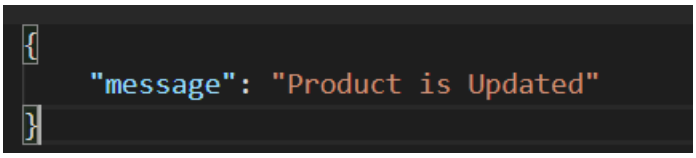
```
from django.http import HttpResponse
from django.views.generic import View
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser
from app3.serializers import ProductSerializer
from app3.models import ProductModel
import io

class UpdateProduct(View):
    def put(self,request):
        b_data = request.body
        strm = io.BytesIO(b_data)
        d1 = JSONParser().parse(strm)
        product_no = d1.get("no", None)
        if product_no:
            try:
                res = ProductModel.objects.get(no=product_no)
                ps = ProductSerializer(res,d1,partial=True)
                if ps.is_valid():
                    ps.save()
                    message = {"message": "Product is Updated"}
                    json_data = JSONRenderer().render(message)
                else:
                    message = {"error":ps.errors}
                    json_data = JSONRenderer().render(message)
            except ProductModel.DoesNotExist:
                message = {"error": "Invalid Product No"}
                json_data = JSONRenderer().render(message)
        else:
```

```
        message = {"error":"Please Provide Product No"}
        json_data = JSONRenderer().render(message)

    return HttpResponse(json_data, content_type=
                                        "application/json")
```

4) Input using Postman



5) Output



Note: In the above example to ProductSerializer class we are passing "partial=**True"** parameter, it means we can pass any no of fields to update else we need to provide all the fields to update.

6) Example program for Custom validations in rest framework serializer.

1) serializers.py

```
from rest_framework import serializers
from app3.models import ProductModel
class ProductSerializer(serializers.Serializer):
    no = serializers.IntegerField()
```

```
    # this method is used to validate product no
    def validate_no(self,product_no):
        if product_no >= 101:
            return product_no
        else:
            raise serializers.ValidationError("Please Check Product No")
```

7) Example program for Custom validations in rest framework serializer. (Using validators)

1) serializers.py

```
from rest_framework import serializers
from app3.models import ProductModel

def validate_amout(amount):
    if amount > 0:
        return amount
    else:
        raise serializers.ValidationError("Amount Must be Greater Than Zero")

class ProductSerializer(serializers.Serializer):
    price = serializers.FloatField(validators=[validate_amout])
```

Output

```
"errors": {
    "no": [
        "Please Check Product No"
    ],
    "price": [
        "Amount Must be Greater Than Zero"
    ]
}
```

8) Example program to Delete 1 Product. (Using delete)

Note: Not using any serializer class to delete

1) serializers.py

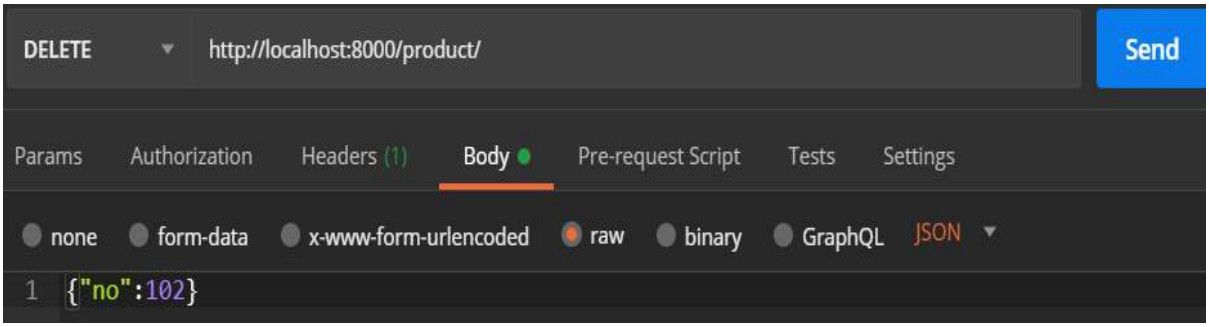
```
from rest_framework import serializers
from rest_framework import serializers

class ProductSerializer(serializers.Serializer):

            ----- No Changes ----
```

2) views.py

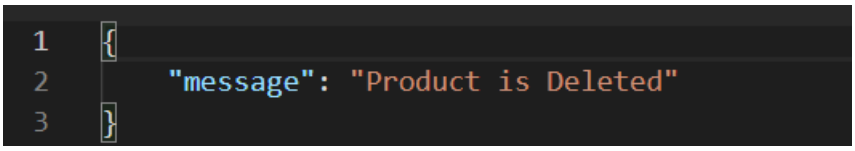
```
from django.http import HttpResponse
from django.views.generic import View
import io
from rest_framework.parsers import JSONParser
from rest_framework.renderers import JSONRenderer
from app12.models import ProductModel

class ProductOperations(View):
   def delete(self,request):
      stream = io.BytesIO(request.body)
      d1 = JSONParser().parse(stream)
      product_no = d1.get("no")
      if product_no:
         no_of_rows =
ProductModel.objects.filter(no=product_no).delete()
         if no_of_rows[0] != 0:
            message = {"message":"Product is Deleted"}
```

```
        else:
            message = {"error": "Please Check the Product No"}
    else:
        message = {"error": "Please Provide Product No"}
    json_data = JSONRenderer().render(message)
    return HttpResponse(json_data, content_type="application/
json")
```

Input



Output



ModelSerializer

The ModelSerializer class provides a shortcut that lets you automatically create a Serializer class with fields that correspond to the Model fields.

The ModelSerializer class is the same as a regular Serializer class, except that:

- It will automatically generate a set of fields for you, based on the model.
- It will automatically generate validators for the serializer, such as unique_together validators.
- It includes simple default implementations of .create() and .update().

Declaring a ModelSerializer looks like this:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ['id', 'account_name', 'users', 'created']
```

By default, all the model fields on the class will be mapped to a corresponding serializer fields.

Any relationships such as foreign keys on the model will be mapped to PrimaryKeyRelatedField.

Specifying which fields to include

It is strongly recommended that you explicitly set all fields that should be serialized using the **fields** attribute.

For example:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ['id', 'account_name', 'users', 'created']
```

You can also set the fields attribute to the special value **'__all__'** to indicate that all fields in the model should be used.

For example:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = '__all__'
```

You can set the exclude attribute to a list of fields to be **excluded** from the serializer.

For example:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        exclude = ['users']
```

Specifying read only fields

You may wish to specify multiple fields as read-only. Instead of adding each field explicitly with the **read_only=True** attribute, you may use the shortcut Meta option**, read_only_fields.**

This option should be a **list or tuple** of field names, and is declared as follows:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ['id', 'account_name', 'users', 'created']
        read_only_fields = ['account_name']
```

1) Example program to insert the product into database using rest framework ModelSerializer. (Using post)

1) serializers.py

```
from rest_framework import serializers
from app4.models import ProductModel

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = ProductModel
        fields = "__all__"
```

2) views.py

```
from django.http import HttpResponse
from django.views.generic import View
from rest_framework.parsers import JSONParser
from rest_framework.renderers import JSONRenderer
from app4.serializers import ProductSerializer
import io

class ProductOperations(View):
    def post(self,request):
        d1 = JSONParser().parse(io.BytesIO(request.body))
        ps = ProductSerializer(data=d1)
        if ps.is_valid():
            ps.save()
            message = {"message":"Product is Saved"}
        else:
            message = {"error":ps.errors}

        json_data = JSONRenderer().render(message)
        return
HttpResponse(json_data,content_type="application/json")
```

3) urls.py

```
path('product/',csrf_exempt(views.ProductOperations.as_view()))
```

4. Output in Postman

```
1  {
2      "message": "Product is Saved"
3  }
```

2) Custom validation in Django-rest framework ModelSerializer.

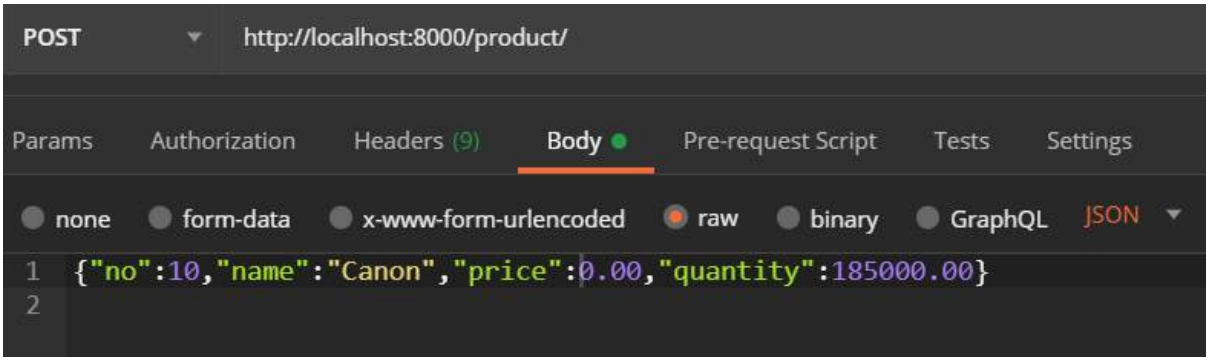
1) serializers.py

```
from rest_framework import serializers
from app4.models import ProductModel

def validate_amout(amount):
    if amount > 0:
        return amount
    else:
        raise serializers.ValidationError("Amount Must be Greater Than Zero")

class ProductSerializer(serializers.ModelSerializer):
    no = serializers.IntegerField()
    price = serializers.FloatField(validators=[validate_amout])

    # this method is used to validate product no
    def validate_no(self, product_no):
        if product_no >= 101:
            return product_no
        else:
            raise serializers.ValidationError("Please Check Product No")

    class Meta:
        model = ProductModel
        fields = "__all__"
```

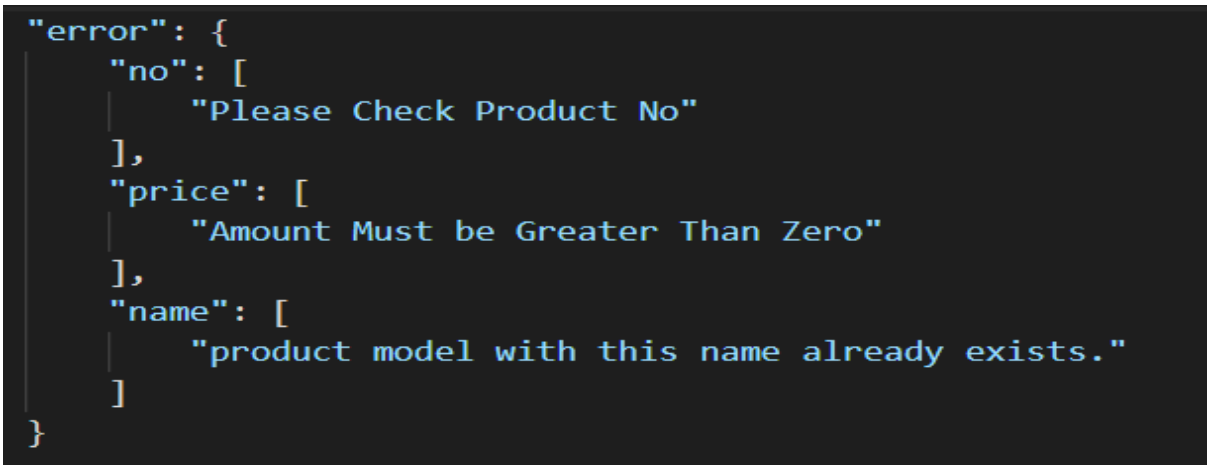
2) views.py

```
class ProductOperations(View):
    def post(self,request):

                ---- same ----
```

3) Input Using Postman



4) Output



Note: post, get, put all are built in with ModelSerializer class (No need write separate code.

Class-based Views

APIView

REST framework provides an **APIView** class, which subclasses Django's **View** class.

APIView classes are different from regular View classes in the following ways:

- Requests passed to the handler methods will be REST framework's Request instances, not Django's HttpRequest instances.
- Handler methods may return REST framework's Response, instead of Django's HttpResponse. The view will manage content negotiation and setting the correct renderer on the response.
- Any APIException exceptions will be caught and mediated into appropriate responses.
- Incoming requests will be authenticated and appropriate permission will be run before dispatching the request to the handler method.

Using the **APIView** class is pretty much the same as using a regular **View** class, as usual, the incoming request is dispatched to an appropriate handler method such as **.get()** or **.post().**

1) Example Program to send some dictionary in response and inheriting APIView class.

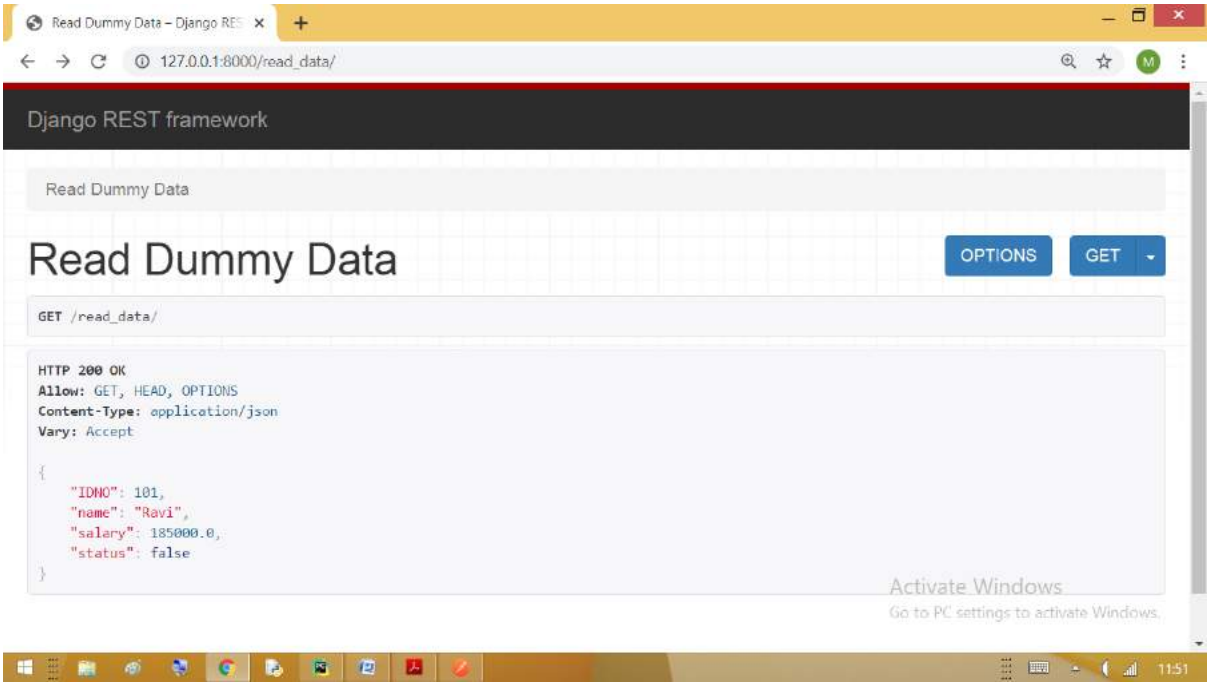
1) views.py

```
from rest_framework.views import APIView
from rest_framework.response import Response


class ReadDummyData(APIView):
   def get(self,request):
      employee_info =
{"IDNO":101,"name":"Ravi","salary":185000.00,"status":False}
      return Response(employee_info)
```

2) urls.py

```
        path('read_data/',views.ReadDummyData.as_view())
```

3) Output In a Web browser.



Note: The Response class will return ***TemplateResponse.***

CURD Operations Using API View and Response Class's

1) models.py

```
from django.db import models

class ProductModel(models.Model):
    no = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=30,unique=True)
    price = models.FloatField()
```

2) serializers.py

```
from rest_framework import serializers
from .models import ProductModel

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = ProductModel
        fields = "__all__"
```

3) views.py

```
from rest_framework.views import APIView
from rest_framework.response import Response
from app15.serializers import ProductSerializer
from app15.models import ProductModel

class InsertProduct(APIView):
    def post(self,request):
        ps = ProductSerializer(data=request.data)
        if ps.is_valid():
```

```
            ps.save()
            message = {"message":"Product Saved"}
        else:
            message = {"error":ps.errors}
        return Response(message)

    def get(self,request):
        qs = ProductModel.objects.all()
        ps = ProductSerializer(qs,many=True)
        return Response(ps.data)

class Read1Product(APIView):
    def get(self,request,product):
        try:
            res = ProductModel.objects.get(no=product)
            return Response(ProductSerializer(res).data)
        except ProductModel.DoesNotExist:
            message = {"error":"Product No is Invalid"}
            return Response(message)

    def put(self,request,product):
        try:
            res = ProductModel.objects.get(no=product)
            d1 = request.data
            ps = ProductSerializer(res,d1,partial=True)
            if ps.is_valid():
                ps.save()
                message = {"message": "Product Updated"}
            else:
                message = {"error": ps.errors}
```

```
        return Response(message)
    except ProductModel.DoesNotExist:
        message = {"error":"Product No is Invalid"}
        return Response(message)

  def delete(self,request,product):
        res = ProductModel.objects.filter(no=product).delete()
        if res[0] != 0:
            message = {"message": "Product is Deleted"}
        else:
            message = {"message": "Invalid Product"}
        return Response(message)
```

4) urls.py

```
path('product/',csrf_exempt(views.InsertProduct.as_view())),
path('read_product/<product>',csrf_exempt(views.Read1Product.as
_view())),
```

Note: use web Browser for output.

View Sets

Django REST framework allows you to combine the logic for a set of related views in a single class, called a ViewSet.

In other frameworks you may also find conceptually similar implementations named something like 'Resources' or 'Controllers'.

A ViewSet class is simply **a type of class-based View, that does not provide any method handlers** such as .get() or .post(), and instead provides actions such as .list() and .create().

The method handlers for a ViewSet are only bound to the corresponding actions at the point of finalizing the view, using the .**as_view()** method.

View Set

The ViewSet class inherits from APIView. You can use any of the standard attributes such as permission_classes, authentication_classes in order to control the API policy on the viewset.

The ViewSet class does not provide any implementations of actions. In order to use a ViewSet class you'll override the class and define the action implementations explicitly.

View Set actions

The default routers included with REST framework will provide routes for a standard set of **create/retrieve/update/destroy** style actions, as shown below:

```
class UserViewSet(viewsets.ViewSet):
    def list(self, request):
        pass
    def create(self, request):
        pass
    def retrieve(self, request, pk=None):
        pass
    def update(self, request, pk=None):
```

```
        pass
    def partial_update(self, request, pk=None):
        pass
    def destroy(self, request, pk=None):
        pass
```

get() means **list() and retrieve()**
post() means **create()**
put() means **update()**
patch() means **partial_update()**
delete() means **destroy()**

1) models.py

```
from django.db import models

class ProductModel(models.Model):
    no = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=30,unique=True)
    price = models.FloatField()
```

2) serializers.py

```
from rest_framework import serializers
from app5.models import ProductModel

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = ProductModel
        fields = "__all__"
```

3) views.py

```
from rest_framework import viewsets
from rest_framework.response import Response
from app5.models import ProductModel
from app5.serializers import ProductSerializer

class ReadData(viewsets.ViewSet):
    def list(self,request):
        data = ProductModel.objects.all()
        ps = ProductSerializer(data,many=True)
        return Response(ps.data)

    def retrieve(self, request, pk=None):
        try:
            pm = ProductModel.objects.get(no=pk)
            ps = ProductSerializer(pm)
            return Response(ps.data)
        except ProductModel.DoesNotExist:
            return Response({"error": "Invalid Product No"})

    def create(self, request):
        ps = ProductSerializer(data=request.data)
        if ps.is_valid():
            ps.save()
            return Response({"message":"saved"})
        else:
            return Response({"error": ps.errors})

    def update(self, request, pk=None):
        try:
```

```
        pm = ProductModel.objects.get(no=pk)
        ps = ProductSerializer(pm,request.data,partial=True)
        if ps.is_valid():
            ps.save()
            return Response({"message": "Updated"})
        else:
            return Response({"error": ps.errors})
    except ProductModel.DoesNotExist:
        return Response({"error": "Invalid Product No"})

  def destroy(self, request, pk=None):
    res = ProductModel.objects.filter(no=pk).delete()
    if res[0] != 0:
        return Response({"message": "Product Deleted"})
    else:
        return Response({"error": "Invalid Product No"})
```

4) urls.py

```
path('read_data_all/',views.ReadData.as_view({'get': 'list'})),
path('read_data_one/<int:pk>',views.ReadData.as_view({'get':
'retrieve'})),
path('save/',views.ReadData.as_view({'post': 'create'})),
path('update/<int:pk>',views.ReadData.as_view({'put': 'update'})),
path('delete/<int:pk>',views.ReadData.as_view({'delete':
'destroy'})),
```

Note: View the output in Web Browser.

ModelViewSet

The ModelViewSet class inherits from **GenericAPIView** and includes implementations for various actions, by mixing in the behavior of the various mixin classes.

The actions provided by the **ModelViewSet** class are **.list(), .retrieve(), .create(), .update(), .partial_update(),** and **.destroy().**

Example

Because **ModelViewSet** extends **GenericAPIView**, you'll normally need to provide at least the queryset and serializer_class attributes.

For example:

1) models.py

```
class Product(models.Model):
    no = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=30,unique=True)
    price = models.FloatField()
```

2) serializers.py

```
from rest_framework import serializers
class ProductSerializer(serializers.ModelSerializer):
    no = serializers.IntegerField(min_value=101)
    class Meta:
        model = Product
        fields = "__all__"
```

3) views.py

```
class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

4) urls.py

```
path('all_products/',views.ProductViewSet.as_view({'get': 'list'}))
```

Note: View the output in Web Browser.

ReadOnlyModelViewSet

The **ReadOnlyModelViewSet** class also inherits from **GenericAPIView**.

As with **ModelViewSet** it also includes implementations for various actions, but unlike **ModelViewSet** only provides the **'read-only'** actions, **.list() and .retrieve().**

Example

As with **ModelViewSet**, you'll normally need to provide at least the **queryset** and **serializer_class** attributes.

For example:

```
class ProductViewSet(viewsets.ReadOnlyModelViewSet):

    queryset = Product.objects.all()

    serializer_class = ProductSerializer
```

Generic views

Django's generic views... were developed as a shortcut for common usage patterns...

One of the key benefits of class-based views is the way they allow you to compose bits of reusable behavior. REST framework takes advantage of this by providing a number of pre-built views that provide for commonly used patterns.

The generic views provided by REST framework allow you to quickly build API views that map closely to your database models.

If the generic views don't suit the needs of your API, you can drop down to using the regular APIView class, or reuse the mixins and base classes used by the generic views to compose your own set of reusable generic views.

GenericAPIView

This class extends REST framework's **APIView** class, adding commonly required behavior for standard list and detail views.

Each of the concrete generic views provided is built by combining **GenericAPIView**, with one or more mixin classes.

Mixins

The mixin classes provide the actions that are used to provide the basic view behavior.

Note that the mixin classes provide action methods rather than defining the handler methods, such as .get() and .post(), directly.

This allows for more flexible composition of behavior.

The mixin classes can be imported from **rest_framework.mixins**.

ListModelMixin

Provides a **.list(request, *args, **kwargs)** method, that implements listing a queryset.

If the queryset is populated, this returns a 200 OK response, with a serialized representation of the queryset as the body of the response. The response data may optionally be paginated.

CreateModelMixin

Provides a .**create(request, *args, **kwargs)** method, that implements creating and saving a new model instance.

If an object is created this returns a 201 Created response, with a serialized representation of the object as the body of the response.

If the representation contains a key named url, then
the Location header of the response will be populated with that value.

If the request data provided for creating the object was invalid, a 400 Bad Request response will be returned, with the error details as the body of the response.

RetrieveModelMixin

Provides a **.retrieve(request, *args, **kwargs)** method, that implements returning an existing model instance in a response.

If an object can be retrieved this returns a 200 OK response, with a serialized representation of the object as the body of the response. Otherwise it will return a 404 Not Found.

UpdateModelMixin

Provides a **.update(request, *args, **kwargs)** method, that implements updating and saving an existing model instance.

Also provides a **.partial_update(request, *args, **kwargs)** method, which is similar to the update method, except that all fields for the update will be optional. This allows support for HTTP PATCH requests.

If an object is updated this returns a 200 OK response, with a serialized representation of the object as the body of the response.

If the request data provided for updating the object was invalid, a 400 Bad Request response will be returned, with the error details as the body of the response.

DestroyModelMixin

Provides a **.destroy(request, *args, **kwargs)** method, that implements deletion of an existing model instance.

If an object is deleted this returns a 204 No Content response, otherwise it will return a 404 Not Found.

Concrete View Classes

The following classes are the concrete generic views. The view classes can be imported from **rest_framework.generics**.

CreateAPIView

Used for **create-only** endpoints.

Provides **a post** method handler.

Extends: GenericAPIView, CreateModelMixin

ListAPIView

Used for **read-only** endpoints to represent a **collection of model instances**.

Provides **a get** method handler.

Extends: GenericAPIView, ListModelMixin

RetrieveAPIView

Used for **read-only** endpoints to represent a **single model instance**.

Provides **a get** method handler.

Extends: GenericAPIView, **RetrieveModelMixin**

DestroyAPIView

Used for **delete-only** endpoints for a **single model instance**.

Provides **a delete** method handler.

Extends: GenericAPIView, DestroyModelMixin

UpdateAPIView

Used for **update-only** endpoints for a **single model instance**.

Provides **put and patch** method handlers.

Extends: GenericAPIView, UpdateModelMixin

ListCreateAPIView

Used for **read-write** endpoints to represent a **collection of model instances**.

Provides **get and post** method handlers.

Extends: GenericAPIView, ListModelMixin, CreateModelMixin

RetrieveUpdateAPIView

Used for **read or update** endpoints to represent a **single model instance**.

Provides **get, put and patch** method handlers.

Extends: GenericAPIView, RetrieveModelMixin, UpdateModelMixin

RetrieveDestroyAPIView

Used for **read or delete** endpoints to represent a **single model instance**.

Provides **get and delete** method handlers.

Extends: GenericAPIView, RetrieveModelMixin, DestroyModelMixin

RetrieveUpdateDestroyAPIView

Used for **read-write-delete** endpoints to represent a **single model instance**.

Provides **get, put, patch and delete** method handlers.

Extends: GenericAPIView, RetrieveModelMixin, UpdateModelMixin , DestroyModelMixin

Concrete View Classes Examples

Note: In these examples we are using common model and serializer classes.

1) models.py

```
from django.db import models
class Product(models.Model):
    no = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=30,unique=True)
    price = models.FloatField()
```

2) serializers.py

```
from rest_framework import serializers
class ProductSerializer(serializers.ModelSerializer):
    no = serializers.IntegerField(min_value=101)
    class Meta:
        model = Product
        fields = "__all__"
```

Example on **CreateAPIView**

1) views.py

```
from app16.models import Product,ProductSerializer
from rest_framework.generics import CreateAPIView
class ProductCreate(CreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

2) urls.py

```
path('create_product/',views.ProductCreate.as_view())
```

Example on **ListAPIView**

1) views.py

```
from app16.models import Product,ProductSerializer
from rest_framework.generics import ListAPIView

class ProductViewAll(ListAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

2) urls.py

```
path('view_all/',views.ProductViewAll.as_view()),
```

Example on **RetrieveAPIView**

1)views.py

```
from app16.models import Product,ProductSerializer
from rest_framework.generics import RetrieveAPIView

class ProductRetrieve(RetrieveAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

2) urls.py

```
path('view_one/<pk>',views.ProductRetrieve.as_view()),
```

3) In web browser type url as http://127.0.0.1:8000/view_one/101

Example on **DestroyAPIView**

1) views.py

```
from app16.models import Product,ProductSerializer
from rest_framework.generics import DestroyAPIView

class ProductDestroy(DestroyAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

2) urls.py

```
path('delete_one/<pk>',views.ProductDestroy.as_view()),
```

3) In web browser type url as http://127.0.0.1:8000/delete_one/101

Example on **UpdateAPIView**

1) views.py

```
from app16.models import Product,ProductSerializer
from rest_framework.generics import UpdateAPIView
class ProductUpdate(UpdateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

2) urls.py

```
path('update/<pk>',views.ProductUpdate.as_view()),
```

3) In web browser type url as http://127.0.0.1:8000/delete_one/101

Example On **ListCreateAPIView**

1) views.py

```
from app16.models import Product,ProductSerializer
from rest_framework.generics import ListCreateAPIView
class ProductListNCreate(ListCreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

2) urls.py

```
path('viewallncreate/',views.ProductListNCreate.as_view()),
```

Example On **RetrieveUpdateAPIView**

1) views.py

```
from app16.models import Product,ProductSerializer
from rest_framework.generics import RetrieveUpdateAPIView
class ProductRetriveNUpdate(RetrieveUpdateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

2) urls.py

```
path('viewonenupdate/<pk>',views.ProductRetriveNUpdate.as_view
()),
```

3) In web browser type url as http://127.0.0.1:8000/ **viewonenupdate** /101

Example On **RetrieveDestroyAPIView**

1) views.py

```
from app16.models import Product,ProductSerializer
from rest_framework.generics import RetrieveDestroyAPIView
class ProductRetriveNDestroy(RetrieveDestroyAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

2) urls.py

```
path('viewonendelete/<pk>',views.ProductRetriveNDestroy.as_view
()),
```

3) In web browser type url as http://127.0.0.1:8000/ **viewonendelete** /101

Example On **RetrieveUpdateDestroyAPIView**

1) views.py

```
from app16.models import Product,ProductSerializer
from rest_framework.generics import
RetrieveUpdateDestroyAPIView
class
ProductRetriveUpdateNDestroy(RetrieveUpdateDestroyAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

2) urls.py

```
path('viewone_updatendelete/<pk>',views.ProductRetriveUpdateN
Destroy.as_view()),
```

Authentication AND Authorization

Authentication is the mechanism of associating an incoming request with a set of identifying credentials, such as the user the request came from, or the token that it was signed with.

The permission and throttling(attack) policies can then use those credentials to determine if the request should be permitted.

Authentication is always run at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed.

Note: Don't forget that **authentication by itself won't allow or disallow an incoming request**, it simply identifies the credentials that the request was made with.

Basic Authentication

This authentication scheme uses **HTTP Basic Authentication**, signed against a user's username and password. Basic authentication is generally only appropriate for **testing**.

Token Authentication

This authentication scheme uses a simple token-based HTTP Authentication scheme.

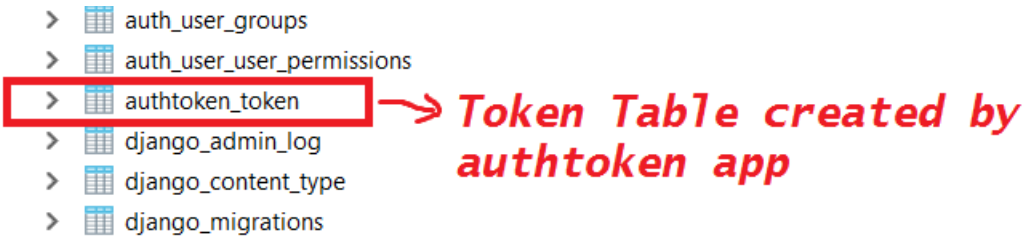
Token authentication is appropriate for client-server setups, such as native desktop and mobile clients.

1) To use the **TokenAuthentication** scheme you'll need to configure the authentication classes to include **TokenAuthentication**, and additionally include **rest_framework.authtoken** in your **INSTALLED_APPS** setting.

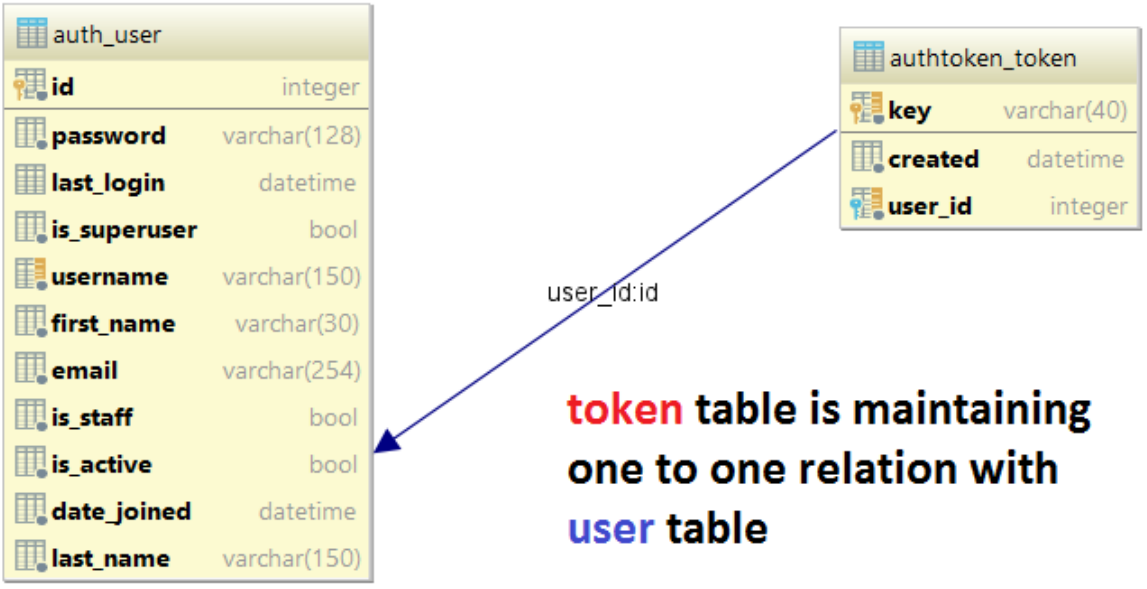
```
INSTALLED_APPS = [
    ...
    'rest_framework.authtoken'
]
```

2) Make sure to run **manage.py migrate** after changing your settings. The **rest_framework.authtoken** app provides Django database migrations.

After Migrate in Database



token table is maintaining one to one relation with user table.



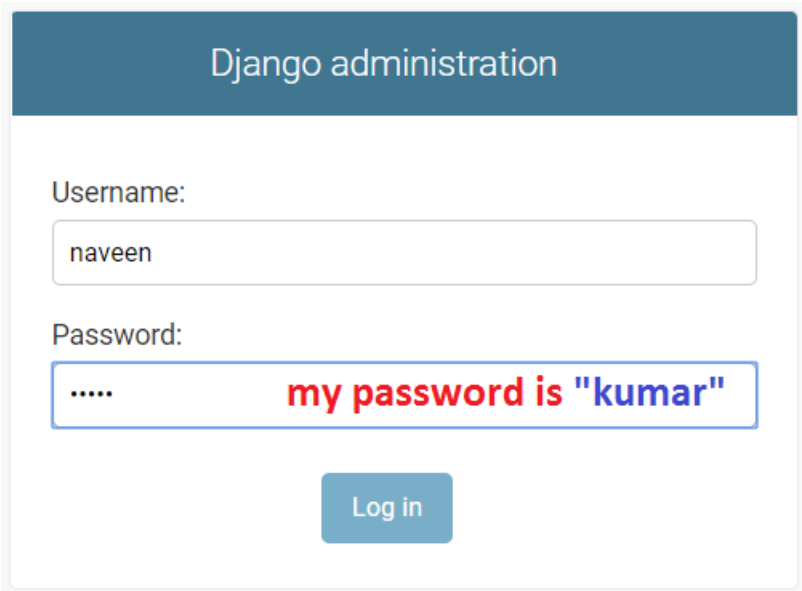
3) Create a super user by providing username and password.

```
F:\Naveen Class Room\Django 8pm Project\N6>python manage.py createsuperuser
Username (leave blank to use 'android'): naveen
Email address: pythonwithnaveen@gmail.com
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

4) Run the Server and open admin in browser

5) Url : http://127.0.0.1:8000/admin/login/?next=/admin/

6) Login by providing username and password.



7) Once you login successfully you can see the **AuthToken.**

Note: Before adding a token add multiple user's in.

8) As of know only 1 user i.e super user.

USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
naveen	pythonwithnaveen@gmail.com			✔

1 user

9) After adding multiple users.

USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
mohan	mohan@gmail.com			✖
naveen	pythonwithnaveen@gmail.com			✔
ravi	ravi@gmail.com			✖

3 users

10) Open **Tokens** Table view from **Auth Token** in admin panel and see, so u can see Zero(0) token.

11) Click on "Add Token"

11) From dropdown select "naveen" and click on "Save" button.

KEY	USER	CREATED
8ed6030bbf3af7c59cc5ca33d56102cf82a20fc6	naveen	Python With Naveen

1 Token

12) Successfully token is generated for user "**naveen**".

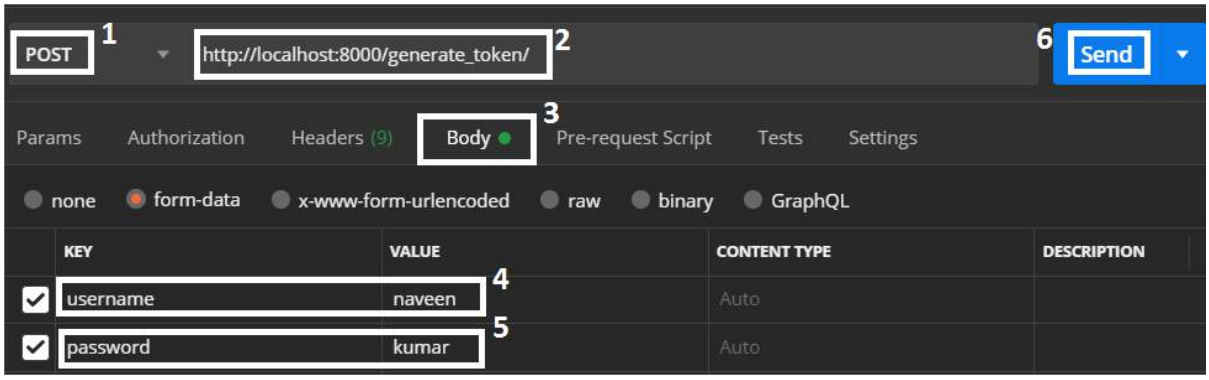
13) Open urls.py

```
from rest_framework.authtoken import views as au
urlpatterns = [
   path('generate_token/',au.obtain_auth_token),
]
```

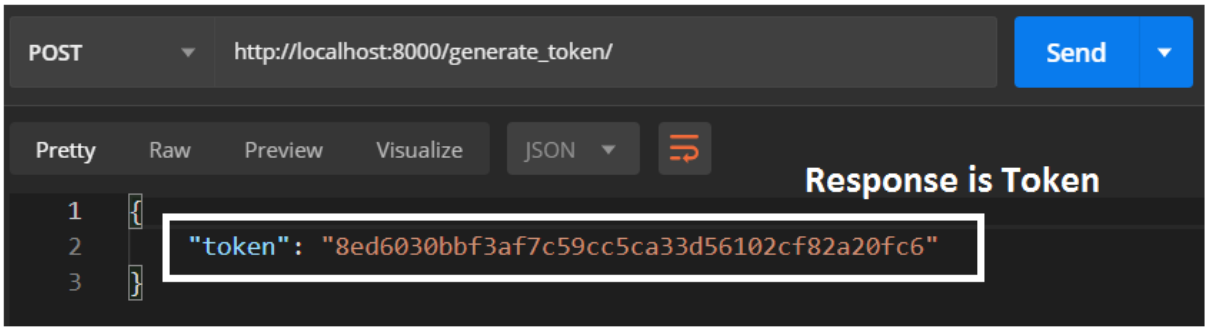
14) Open Postman to send a request and get a token.

Note: 1) Use Post request

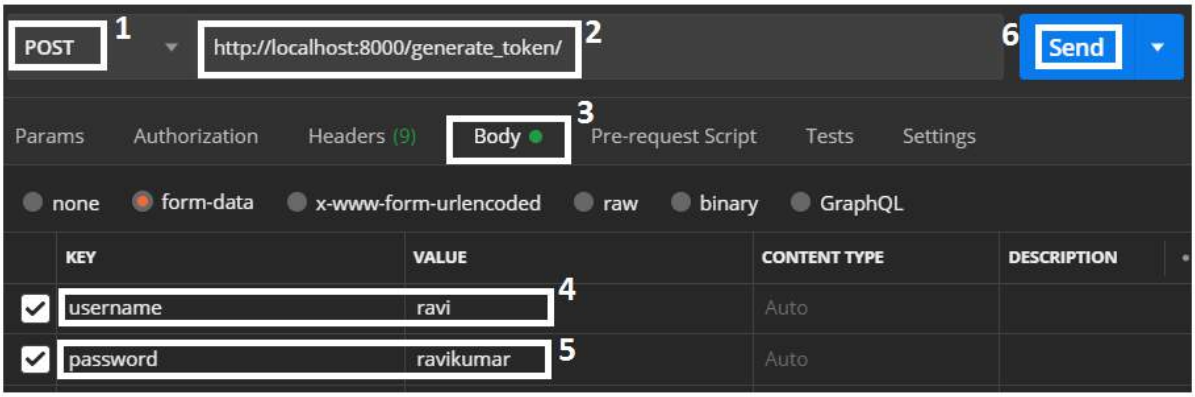
2) Send Username and password in body



Response is Generated Token.



Note: If Token is not available then the "**authtoken**" will generate a new token and it will return.




```
"token": "41a6a21383a38c2877fd86dba69bc68b5c4d784d"
```

New Token.

15) After sending user "ravi" request.

KEY	USER	CREATED
41a6a21383a38c2877fd86dba69bc68b5c4d784d	ravi	Python With Naveen
8ed6030bbf3af7c59cc5ca33d56102cf82a20fc6	naveen	Python With Naveen

2 Tokens

Important Point's.

1. The "Auth Token" app will authenticate the username and password.
2. If the given username and password is invalid it will return "Unable to log in with provided credentials".
3. If the given username and password is valid, the "Auth Token" app will return available "token", if the token is not available the "Auth Token" app will generate a new token and it will return.

Setting Authentication Scheme to Rest-API

We can set authentication in different ways like **globally** it means to entire rest application or **locally** to a specific class.

1) Default (Globally)

The default authentication schemes may be set globally, using the DEFAULT_AUTHENTICATION_CLASSES setting.

For example.

```
REST_FRAMEWORK = {
   'DEFAULT_AUTHENTICATION_CLASSES': [
      'rest_framework.authentication.TokenAuthentication', ] }
```

2) Locally

a) To a Class

You can also set the authentication scheme on a per-view or per-viewset basis, using the APIView class-based views.

```
from rest_framework.authentication import TokenAuthentication

from rest_framework.permissions import IsAuthenticated

from rest_framework.response import Response

from rest_framework.views import APIView

class ExampleView(APIView):

    authentication_classes = [TokenAuthentication]

    permission_classes = [IsAuthenticated]

    def get(self, request, format=None):

        content = {

            'message':"Ok"

        }

        return Response(content)
```

b) To a Function

if you're using the @api_view decorator with function based views.

```
@api_view(['GET'])

@authentication_classes([SessionAuthentication, BasicAuthentication])
```

```
@permission_classes([IsAuthenticated])

def example_view(request, format=None):

    content = {

        'user': unicode(request.user),  # `django.contrib.auth.User`
instance.

        'auth': unicode(request.auth),  # None

    }

    return Response(content)
```

Unauthorized and Forbidden responses

When an unauthenticated request is denied permission there are two different error codes that may be appropriate.

- HTTP 401 Unauthorized
- HTTP 403 Permission Denied

Using django-rest-auth

The django rest auth aim to solve this demand by providing django-rest-auth, a set of REST API endpoints to handle User **Registration** and **Authentication** tasks.

https://django-rest-auth.readthedocs.io/en/latest/index.html

Features

- User Registration with activation
- Login/Logout
- Retrieve/Update the Django User model
- Password change
- Password reset via e-mail
- Social Media authentication

Apps structure

- **"rest_auth"** has basic auth functionality like login, logout, password reset and password change
- **"rest_auth.registration"** has logic related with registration and social media authentication

Installation

1. Install package: pip install django-rest-auth
2. Add **rest_auth** app to INSTALLED_APPS in your django settings.py.
3. Include **"rest_auth"** urls in path of urls.py.

settings.py

```
INSTALLED_APPS = (
        'rest_auth'
)
```

urls.py

```
from django.urls import path,include
urlpatterns = [
        path('gen_token/',include('rest_auth.urls')),
        ]
```

Available users in Database to login

USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
mohan	mohan@gmail.com			✗
naveen	pythonwithnaveen@gmail.com			✓
ravi	ravi@gmail.com			✗

3 users

Note: If server is already started just re built the server else start the server to access the end point.

Open the browser and type the deferent url for deferent operations.

http://localhost:8000/gen_token/login/

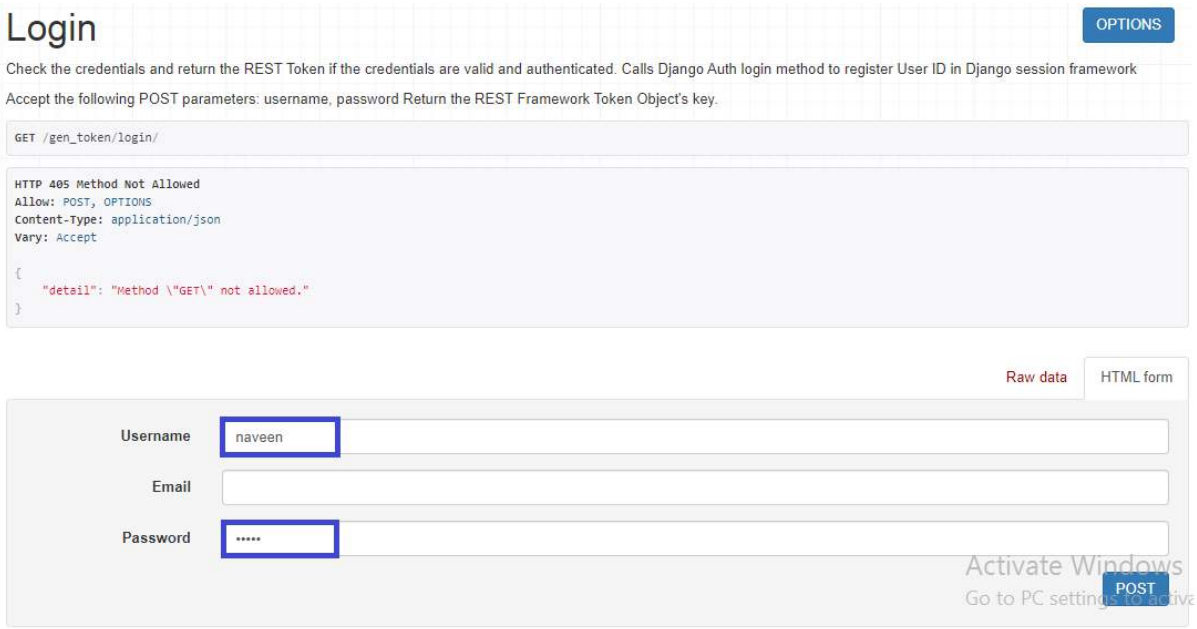
http://localhost:8000/gen_token/logout/

http://localhost:8000/gen_token/user/

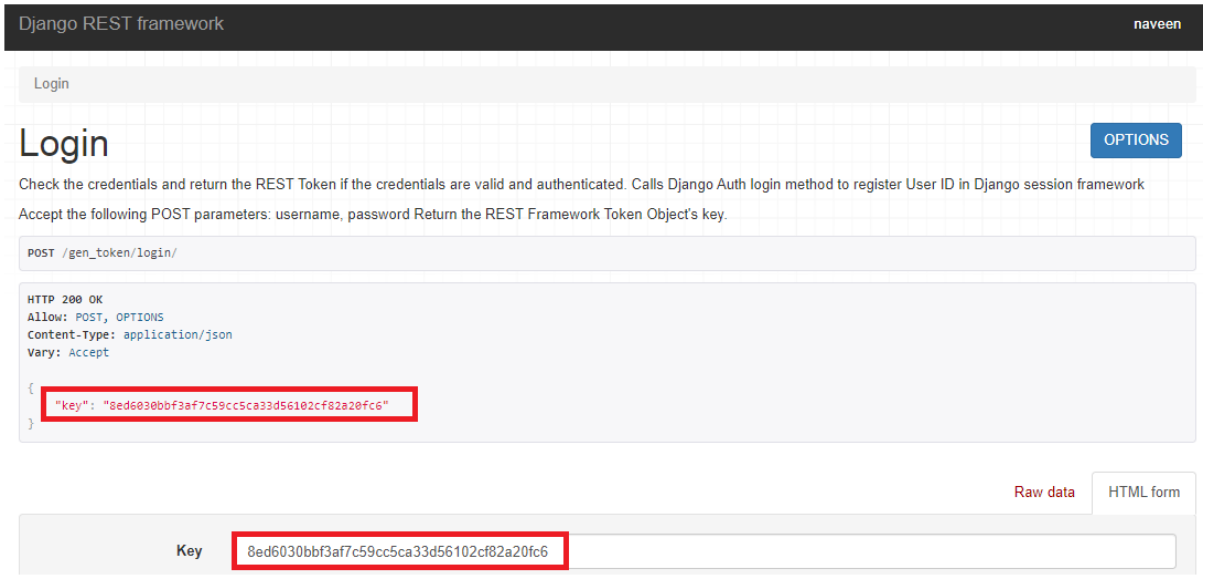
http://localhost:8000/gen_token/password/change/

etc., Look at the documentation link given.

http://localhost:8000/gen_token/login/ Output



Enter Valid username and password and click on "POST" button.



Permissions

Together with authentication and throttling, permissions determine whether a request should be granted or denied access.

Permission checks are always run at the very start of the view, before any other code is allowed to proceed.

Permission checks will typically use the authentication information in the **request.user** and **request.auth** properties to determine if the incoming request should be permitted.

Permissions are used to grant or deny access for different classes of users to different parts of the API.

The simplest style of permission would be to allow access to any authenticated user, and deny access to any unauthenticated user.

Different Types of Permissions

AllowAny

The AllowAny permission class will allow unrestricted access, **regardless of if the request was authenticated or unauthenticated**.

This permission is not strictly required, since you can achieve the same result by using an empty list or tuple for the permissions setting.

IsAuthenticated

The IsAuthenticated permission class will deny permission to any unauthenticated user, and allow permission otherwise.

This permission is suitable if you want your API to only be accessible to registered users.

IsAdminUser

The IsAdminUser permission class will deny permission to any user, unless **user.is_staff** is True in which case permission will be allowed.

This permission is suitable if you want your API to only be accessible to a subset of trusted administrators.

IsAuthenticatedOrReadOnly

The IsAuthenticatedOrReadOnly will allow authenticated users to perform any request. Requests for unauthorised users will only be permitted if the request method is one of the "safe" methods; GET, HEAD or OPTIONS.

This permission is suitable if you want to your API to allow read permissions to anonymous users, and only allow write permissions to authenticated users.

DjangoModelPermissions

This permission class ties into Django's standard **django.contrib.auth** model permissions.

This permission must only be applied to views that have a **.queryset** property set. Authorization will only be granted if the user *is authenticated* and has the *relevant model permissions* assigned.

- POST requests require the user to have the add permission on the model.
- PUT and PATCH requests require the user to have the change permission on the model.

- DELETE requests require the user to have the delete permission on the model.

DjangoModelPermissionsOrAnonReadOnly

Similar to DjangoModelPermissions, but also allows unauthenticated users to have read-only access to the API.

Setting the permission policy to Rest-API

We can set permission in different ways like **globally** it means to entire rest application or **locally** to a specific class.

1) Default (Globally)

The default permission policy may be set globally, using the DEFAULT_PERMISSION_CLASSES setting.

For example.

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ]
}
```

If not specified, this setting defaults to allowing unrestricted access:

```
'DEFAULT_PERMISSION_CLASSES': [
    'rest_framework.permissions.AllowAny',
]
```

2) Locally

a) To a Class

You can also set the authentication policy on a per-view, or per-viewset basis, using the APIView class-based views.

```
from rest_framework.permissions import IsAuthenticated

from rest_framework.response import Response

from rest_framework.views import APIView

class ExampleView(APIView):

    permission_classes = [IsAuthenticated]

    def get(self, request, format=None):

        content = {

            'status': 'request was permitted'

        }

        return Response(content)
```

b) to a Function

if you're using the **@api_view** decorator with function based views.

```
from rest_framework.decorators import api_view,
permission_classes

from rest_framework.permissions import IsAuthenticated

from rest_framework.response import Response

from rest_framework.response import Response
```

```
@api_view(['GET'])
@permission_classes([IsAuthenticated])
def example_view(request, format=None):
    content = {
        'status': 'request was permitted'
    }
    return Response(content)
```

Example Program on Token Authentication Globally.

1) Create a project and app

2) In project

3) Open settings.py file

```
INSTALLED_APPS = [
    .......
    'rest_framework',
    'rest_framework.authtoken',
]
# setting authentication and authorization globally
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES':
['rest_framework.authentication.TokenAuthentication', ],
'DEFAULT_PERMISSION_CLASSES':('rest_framework.permissions.IsAuthenticated',)
}
```

4) In app

5) Open models.py file

```
from django.db import models
class ProductModel(models.Model):
    no = models.IntegerField(primary_key=True)
    name = models.CharField(unique=True,max_length=30)
    price = models.FloatField()
```

6) Open serializers.py file

```
from rest_framework import serializers
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = ProductModel
        fields = "__all__"
```

7) Open views.py file

```
from rest_framework.generics import ListCreateAPIView
from app19.models import ProductModel,ProductSerializer
class ProductOperations(ListCreateAPIView):
    queryset = ProductModel.objects.all()
    serializer_class = ProductSerializer
```

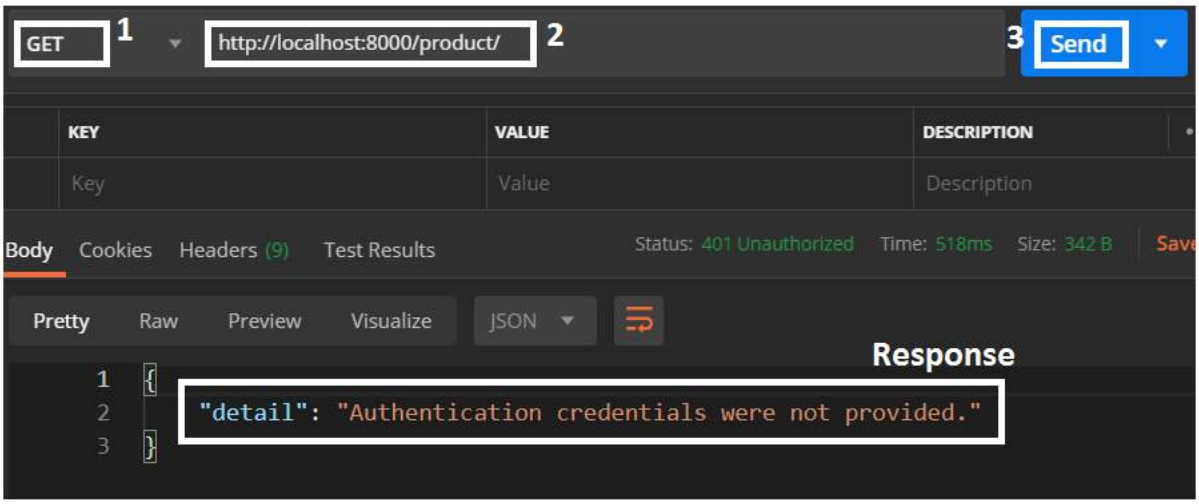
8) Open urls.py

```
from django.urls import path
from app19 import views
urlpatterns = [
    path('product/',views.ProductOperations.as_view()),

]
```

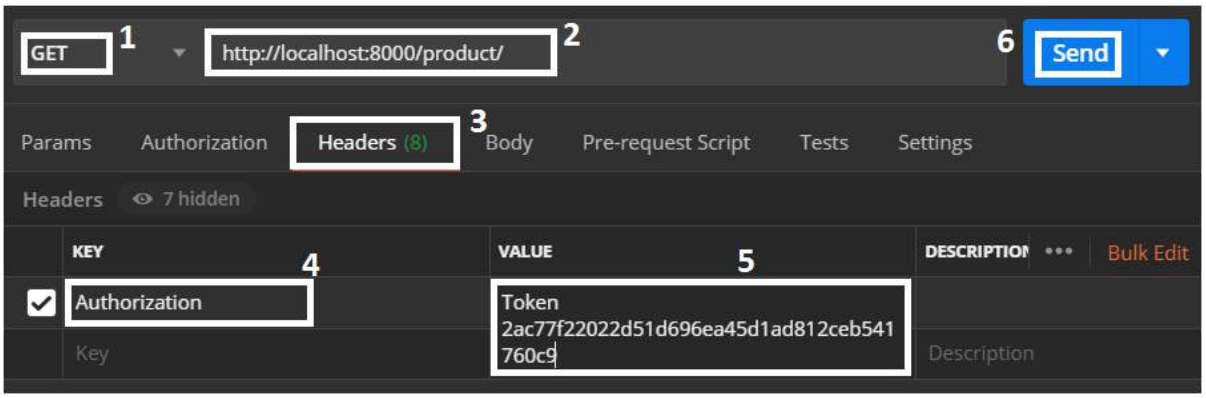
9) Run the Server.

10) Use Postman to run the program.

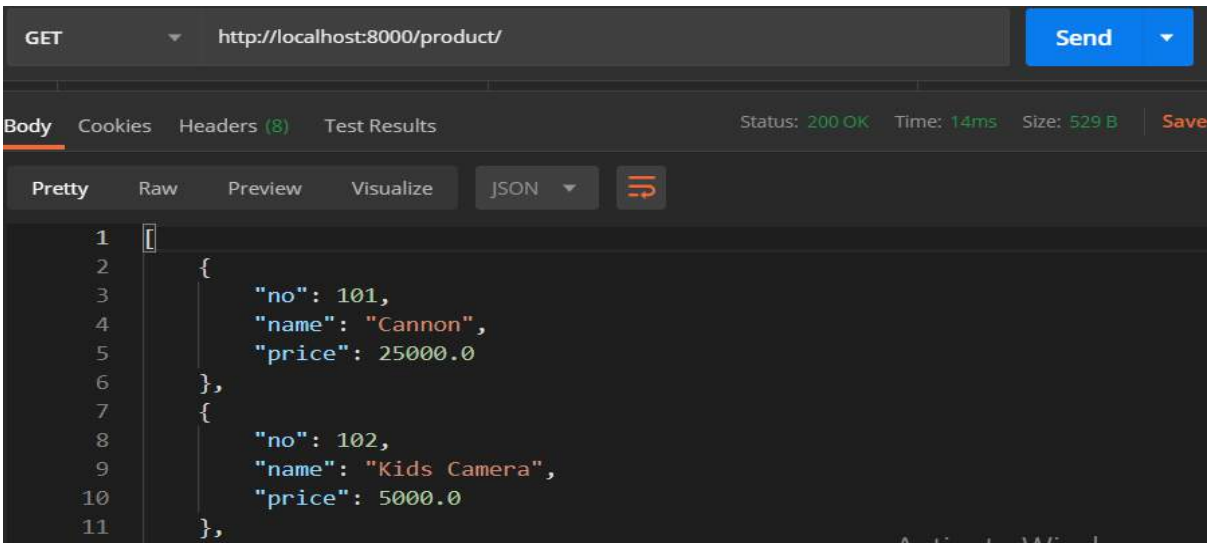
Sending request **without** providing authentication details in postman



Sending request **with** providing authentication details like **token** in postman



Response:



Example Program on Token Authentication Locally.

1) Create a project and app

2) In project

3) Open settings.py file

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

4) Open models.py file

```
from django.db import models
class ProductModel(models.Model):
    no = models.IntegerField(primary_key=True)
    name = models.CharField(unique=True,max_length=30)
    price = models.FloatField()
```

5) Open serializers.py file

```
from rest_framework import serializers
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = ProductModel
        fields = "__all__"
```

6) Open views.py file

```
from rest_framework.generics import ListCreateAPIView
from rest_framework.authentication import TokenAuthentication
from rest_framework.permissions import IsAuthenticated
```

```
from app19.models import ProductModel,ProductSerializer
class ProductOperations(ListCreateAPIView):
    queryset = ProductModel.objects.all()
    serializer_class = ProductSerializer
    authentication_classes = [TokenAuthentication]
    permission_classes = [IsAuthenticated]
```

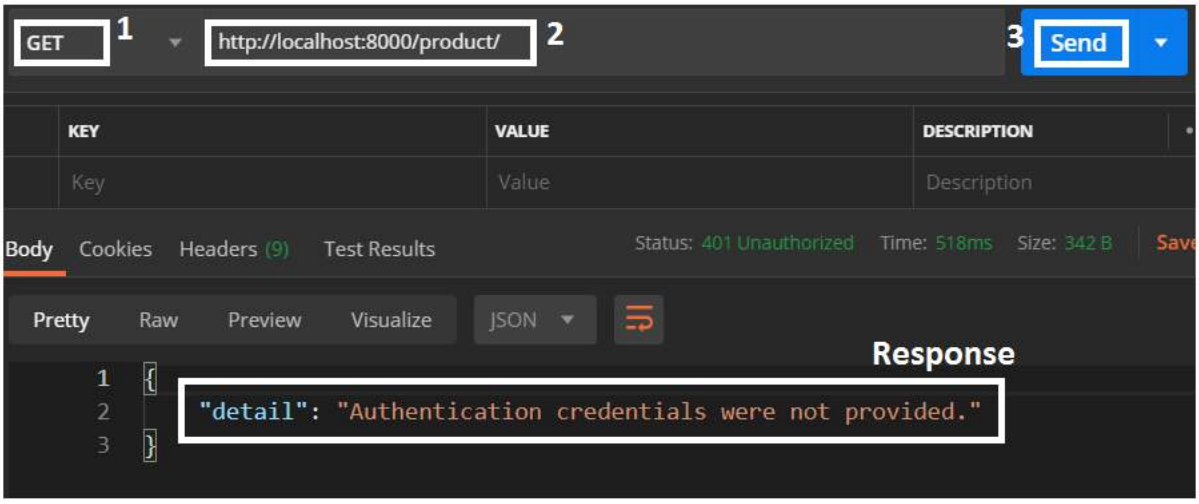
7) Open urls.py

```
from django.urls import path
from app19 import views
urlpatterns = [
    path('product/',views.ProductOperations.as_view()),

]
```

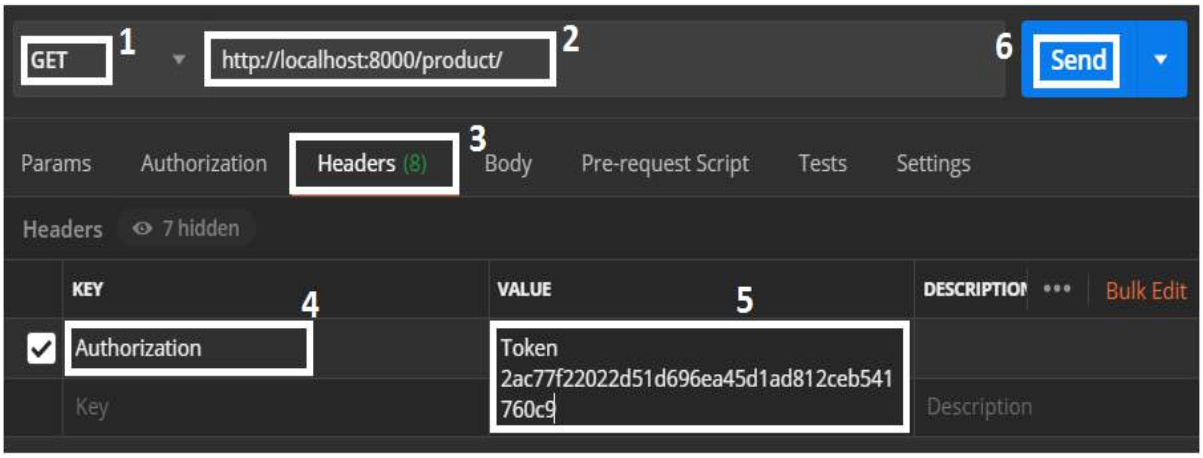
8) Run the Server.

9) Use Postman to run the program.

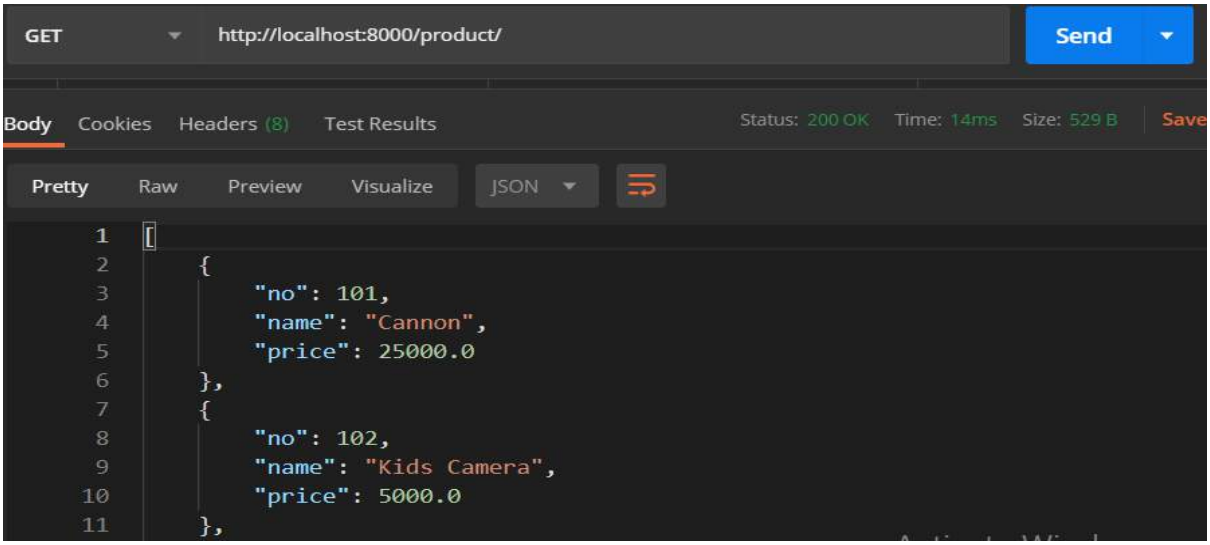
Sending request **without** providing authentication details in postman



Sending request **with** providing authentication details like **token** in postman



Response:



SessionAuthentication

This authentication scheme uses Django's default session backend for authentication.

Session authentication is appropriate for AJAX clients that are running in the same session context as your website.

Unauthenticated responses that are denied permission will result in an **HTTP 403 Forbidden** response.

Warning: Always use Django's standard login view when creating login pages. This will ensure your login views are properly protected.

views.py

```
from rest_framework.generics import ListCreateAPIView
from rest_framework.authentication import SessionAuthentication
from rest_framework.permissions import IsAuthenticated
from app19.models import ProductModel,ProductSerializer

class ProductOperations(ListCreateAPIView):
    queryset = ProductModel.objects.all()
    serializer_class = ProductSerializer
    authentication_classes = [SessionAuthentication]
    permission_classes = [IsAuthenticated]
```

JSON Web Token

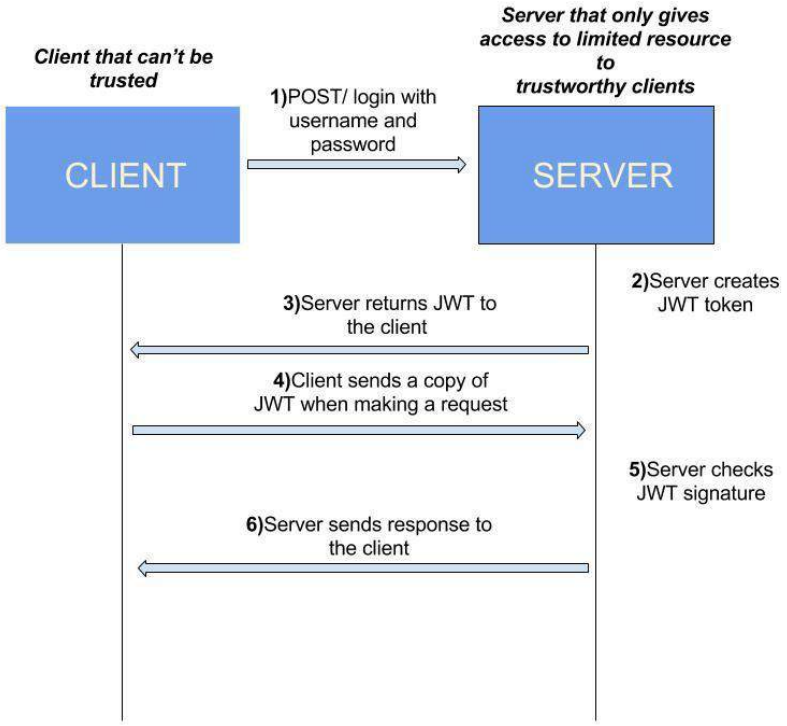
JSON Web Token (JWT) is an open standard RFC 7519 (Request for Comments) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

This information can be verified and trusted because it is digitally signed.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on *signed* tokens.

Signed tokens can verify the *integrity* of the claims contained within it, while encrypted tokens *hide* those claims from other parties.

When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.



When should you use JSON Web Tokens?

- **Authorization**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token.
- **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

What is the JSON Web Token structure?

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

- Header
- Payload
- Signature

Therefore, a JWT typically looks like the following.

xxxxx.yyyyy.zzzzz

Let's break down the different parts.

Header

The header *typically* consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

For example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: ***registered***, ***public***, and ***private*** claims.

- **Registered claims**: These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub** (subject), **aud** (audience), and others.

Notice that the claim names are only three characters long as JWT is meant to be compact.

- **Public claims**: These can be defined at will by those using JWTs.

- **Private claims**: These are the custom claims created to share information between parties that agree on using them and are neither *registered* or *public* claims.

An example payload could be:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the **HMAC SHA256** algorithm, the signature will be created in the following way:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Putting all together

The output is three strings separated by dots that can be easily passed in HTML and HTTP environments.

The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```


If you want to play with JWT and put these concepts into practice, you can use jwt.io Debugger to decode, verify, and generate JWTs.

Installation & Setup

pip install djangorestframework_simplejwt

settings.py

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ],
}

INSTALLED_APPS = [
    'rest_framework',
]
```

models.py

```
from django.db import models

class Product(models.Model):
    no = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=30)
    price = models.FloatField()
```

seroalizers.py

```
from rest_framework import serializers

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = "__all__"
```

views.py

```
from rest_framework.generics import CreateAPIView
from rest_framework.generics import ListAPIView
from app.models import Product
from app.models import ProductSerializer
from rest_framework.permissions import IsAuthenticated

class InsertProduct(CreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    permission_classes = [IsAuthenticated]
```

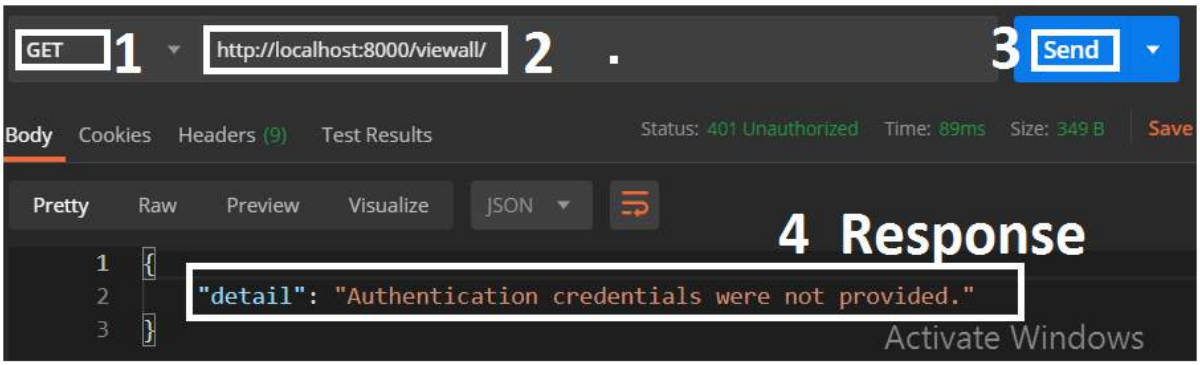
```
class ViewAllProducts(ListAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    permission_classes = [IsAuthenticated]
```

urls.py

```
from rest_framework_simplejwt import views as v
from app import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('token/', v.TokenObtainPairView.as_view()),
    path('token/refresh/', v.TokenRefreshView.as_view()),

    path('insert/',views.InsertProduct.as_view()),
    path('viewall/',views.ViewAllProducts.as_view()),
]
```

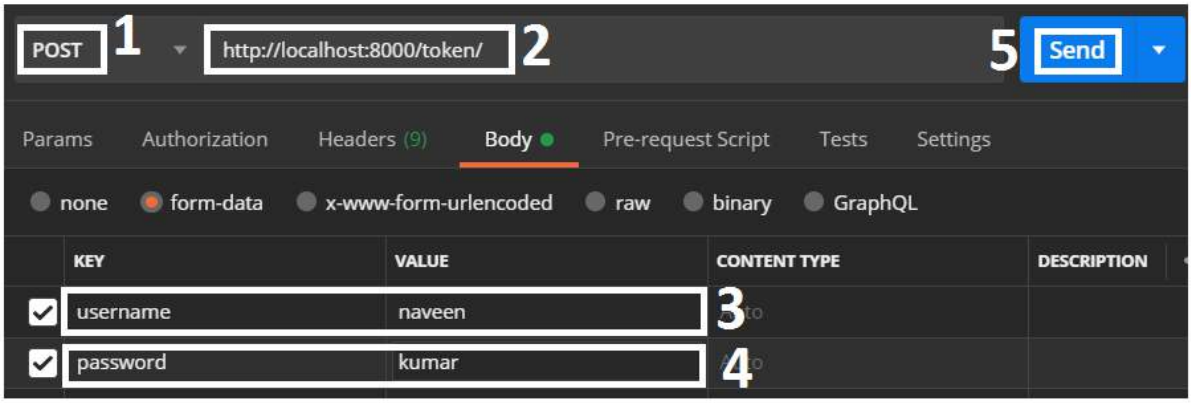
Running the Application using postman

Step 1 :

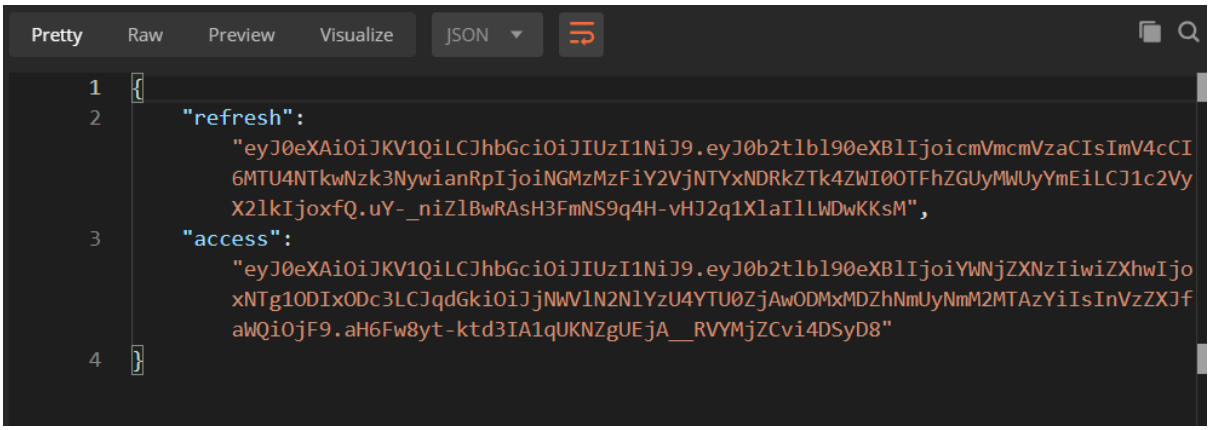


Step 2: Create a Super user

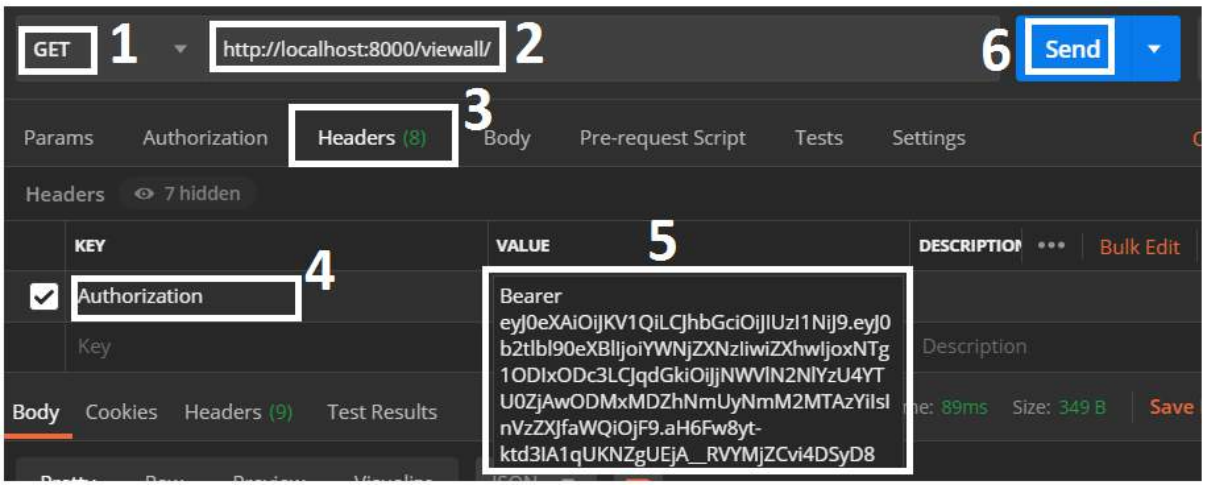
Step 3:



Step 4: Response for above request is



Step 5:



Step 6: Got the response

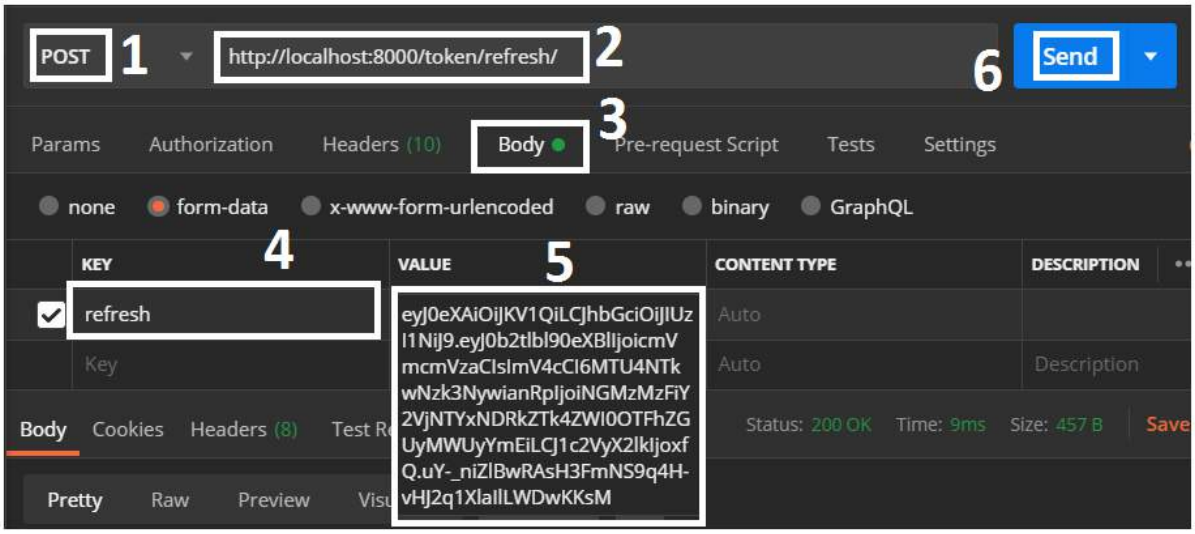
```
{
    "no": 101,
    "name": "Cannon",
    "price": 25000.0
},
{
    "no": 102,
    "name": "Kids Camera",
    "price": 5000.0
}
```

Note: You can use this **access token** for the next five minutes only. After five min, the token will expire, and if you try to access the view again, you are going to get the following error:

```
{
    "detail": "Given token not valid for any token type",
    "code": "token_not_valid",
    "messages": [
        {
            "token_class": "AccessToken",
            "token_type": "access",
            "message": "Token is invalid or expired"
        }
    ]
}
```

Refresh Token

To get a new **access token**, you should use the refresh token endpoint **token/refresh/** posting the **refresh token**:



Response is


```
{
    "access":
        "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjo
        xNTg1ODIyNjkzLCJqdGkiOiJjYmE4YTdhNTQ3YzA0YWRkODYyMjY3MzYzY2FkODAxNSIsInVzZXJf
        aWQiOjF9.uFzWsPl3d1QcQeA8lW1noUitVriBXq9qDqJCqmcJh8E"
}
```


Use this Token for further Use.

The return is a new **access token** that you should use in the subsequent requests.

The **refresh token** is valid for the next 24 hours. When it finally expires too, the user will need to perform a full authentication again using their username and password to get a new set of **access token** + **refresh token**.