**What is an API?**

API refers to Application Programming Interface.

It is a set of subroutine definitions, communication protocols and tools for building application software.

It helps in connecting various software components.

An API makes it easier for developers to use certain technologies in building applications by using certain predefined operations.

An application programming interface (API) is code that allows two software programs to communicate with each other.

Other definition

API is a software intermediary that allows two applications to talk to each other.

Each time you use an app like Facebook, send an instant message, or check weather on your phone, you’re using an API.

**Example of an API:**When you use an application on your mobile phone, the application connects to the internet and sends data to a server.

The sever then retrieves that data, interprets it, performs the necessary actions and sends it back to your phone.

The application then interprets that data and presents you with the information you wanted in a readable way.0

This is what an API is, all of this happens via API.

**A familiar example:**

To explain this better, let us take a good example.

Imagine you are sitting at a table in a restaurant with a menu to order.

The kitchen is the part of the “system” that will prepare your order.

What is missing is the critical link to communicate your order to the kitchen and deliver your food back to your table.

That’s where the waiter or API comes in. The waiter is the messenger or API that takes your request or order and tells the kitchen (the system) what to do. Then the waiter delivers the response back to you, in this case it is food.

**REST framework:**

It describes an architecture which stands for Representational State Transfer.

It is used for web based architecture for data communication.

It uses HTTP to make calls between machines.

The following HTTP methods are most commonly used in a REST based architecture:

GET, PUT, POST, DELETE

GET is used to retrieve the resource.

PUT is used to Update the resource.

POST is used to create the resource.

Delete is used to remove the resource.

**What is a REST API?**

When web services use REST architecture, they are called RESTFUL API’s or REST API’s.

REST API is used to connect your application with any other third party application.

A REST API is a set of web addresses that respond with pure information, not a formatted web page.

An API returns a JSON, which is a common format. You will see all the information surrounded with quotation marks, {}, [] and descriptive titles for each bit of info.

**JSON:**

JSON (Javascript Object Notation) is most widely used data format for data interchange on the web. This data interchange can happen between two computer applications at different geographical locations or running within same hardware machine.

The good thing is that JSON is a human and machine readable format. So while applications/libraries can parse the JSON data, humans can also look at data and derive meaning from it.

A JSON document may contains text, {}, [], :, , (commas), “” (double quote), and other characters.

Primarily, JSON is built on two structures:

1. A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.

2. An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

**JSON Example:**

A sample JSON document looks like this:

//JSON Object

{

    "employee": {

        "id":           1,

        "name":         "Admin",

        "location":     "USA"

    }

}

//JSON Array

{

    "employees": [

                 {

                    "id":           1,

                    "name":         "Admin",

                    "location":     "USA"

                 },

                 {

                    "id":           2,

                    "name":         "User",

                    "location":     "USA"

                 },

                 {

                    "id":           3,

                    "name":         "User2",

                    "location":     "USA"

                 }

    ]

}

**Installation:**

Use the following commands to install django rest framework

pip install djangorestframework==3.7.3

pip install djangorestframework-jwt==1.11.0

jwt – JASON Web Tokens

Create the project named with sample\_project.

Create a application named with postings.

Add the following inside settings.py file

INSTALLED\_APPS = [

...

'rest\_framework',

'postings'

]

REST\_FRAMEWORK = {

'DEFAULT\_PERMISSION\_CLASSES': (

'rest\_framework.permissions.IsAuthenticated',

),

'DEFAULT\_AUTHENTICATION\_CLASSES': (

'rest\_framework\_jwt.authentication.JSONWebTokenAuthentication',

'rest\_framework.authentication.SessionAuthentication',

'rest\_framework.authentication.BasicAuthentication',

),

}

postings/models.py:

**from** django.db **import** models  
**from** django.contrib.auth.models **import** User  
  
*# Create your models here.***class BlogPost**(models.Model)**:** user **=** models.ForeignKey(User, on\_delete**=**models.CASCADE)  
 title **=** models.CharField(max\_length**=**200, null**=True**, blank**=True**)  
 content **=** models.TextField(max\_length**=**200, null**=True**, blank**=True**)  
 timestamp **=** models.DateTimeField(auto\_now\_add**=True**)  
  
 **def \_\_str\_\_**(self)**:  
 return** str(self.user.username)

Register this model inside the admin.py file.

Inside the postings application create api folder.

Inside the api folder create \_\_init\_\_.py file, so that to make it as a module or package.

Note:

1. API Endpoint(s) uri (url)
   1. Retrieve Update Delete
   2. Create, List & Search
   3. Permissions?? JWT
2. HTTP methods
   1. GET, POST, PUT, PATCH, DELETE

These are the methods we will use at the client side.

1. Data Types & Validation
   1. JSON -> Serializer
   2. Validation -> Serializer

Serializers:

Serializers allow complex data such as querysets and model instances to be converted to native python datatypes that can 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.

We provide a Serializer class which gives you a powerful, generic way to control the output of your responses, as well as a ModelSerializer class which provides a useful shortcut for creating serializers that deal with model instances and querysets.

**Difference between URI, URL and URN:**

Uniform Resource Identifier (URI) is a string of characters used to identify a name or a resource on the internet.

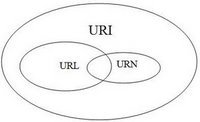
A URI identifies a resource either by location, or a name, or both. A URI has two specializations known as URL and URN.

A Uniform Resource Locator (URL) is a subset of the Uniform Resource Identifier (URI) that specifies where an identified resource is available and the mechanism for retrieving it.

URL defines how the resource can be obtained. It does not have to be HTTP URL (http://), a URL can also be (fttp://) or (smb://).

A Uniform Resource Name (URM) is a Uniform Resource Identifier (URI) that uses the URN scheme, and does not imply availability of the identified resorce.

Both URNs (names) and URLs (locators) are URIs, and a particular URI may be both a name and a locator at the same time.



The URNs are part of a larger internet information architecture which is composed of URNs, URCs and URLs.

bar.html is not a URN. A URN is similar to a person’s name, while a URL is like a street address. The URN defines something’s identity, while the URL provides a location. Essentially, “what” vs. “where”. A URN has to be of this form <URN> ::= "urn:" <NID> ":" <NSS> where <NID> is the Namespace Identifier, and <NSS> is the Namespace Specific String.

To put it differently:

A URL is a URI that identifies a resource and also provides the means of locating the resource by describing the way to access it

A URL is a URI

A URI is not necessarily a URL

I’d say the only thing left to make it 100% clear would be to have an example of an URI that is not an URL. We can use the examples in the RFC3986:

URL: ftp://ftp.is.co.za/rfc/rfc1808.txt

URL: http://www.ietf.org/rfc/rfc2396.txt

URL: ldap://[2001:db8::7]/c=GB?objectClass?one

URL: mailto:John.Doe@example.com

URL: news:comp.infosystems.www.servers.unix

URL: telnet://192.0.2.16:80/

URN (not URL): urn:oasis:names:specification:docbook:dtd:xml:4.1.2

URN (not URL): tel:+1-816-555-1212 (?)

Create views.py file inside the api folder.

Create serializers.py file inside the api folder.

serializers.py:

**from** rest\_framework **import** serializers  
**from** postings.models **import** BlogPost  
  
**class BlogPostSerializer**(serializers.ModelSerializer)**:** *# same like forms.ModelForm* **class Meta:** model **=** BlogPost  
 fields **=** [  
 **'pk'**,  
 **'user'**,  
 **'title'**,  
 **'content'**,  
 **'timestamp'**,  
 ]  
  
  
*# serialiazer will do the following:  
# converts to JSON and valid  
# validations for data passed*

views.py:

**from** rest\_framework **import** generics  
**from** postings.models **import** BlogPost  
**from** .serializers **import** BlogPostSerializer  
  
**class BlogPostRudView**(generics.RetrieveUpdateDestroyAPIView)**:** lookup\_field **= 'pk'** *# it can also be slug, id  
 # queryset = BlogPost.objects.all()* serializer\_class **=** BlogPostSerializer  
  
 **def get\_queryset**(self)**:  
 return** BlogPost.objects.all()  
  
 *# def get\_object(self):  
 # pk = self.kwargs.get('pk')  
 # return BlogPost.objects.get(pk=pk)*

Create urls.py file inside api folder.

api/urls.py:

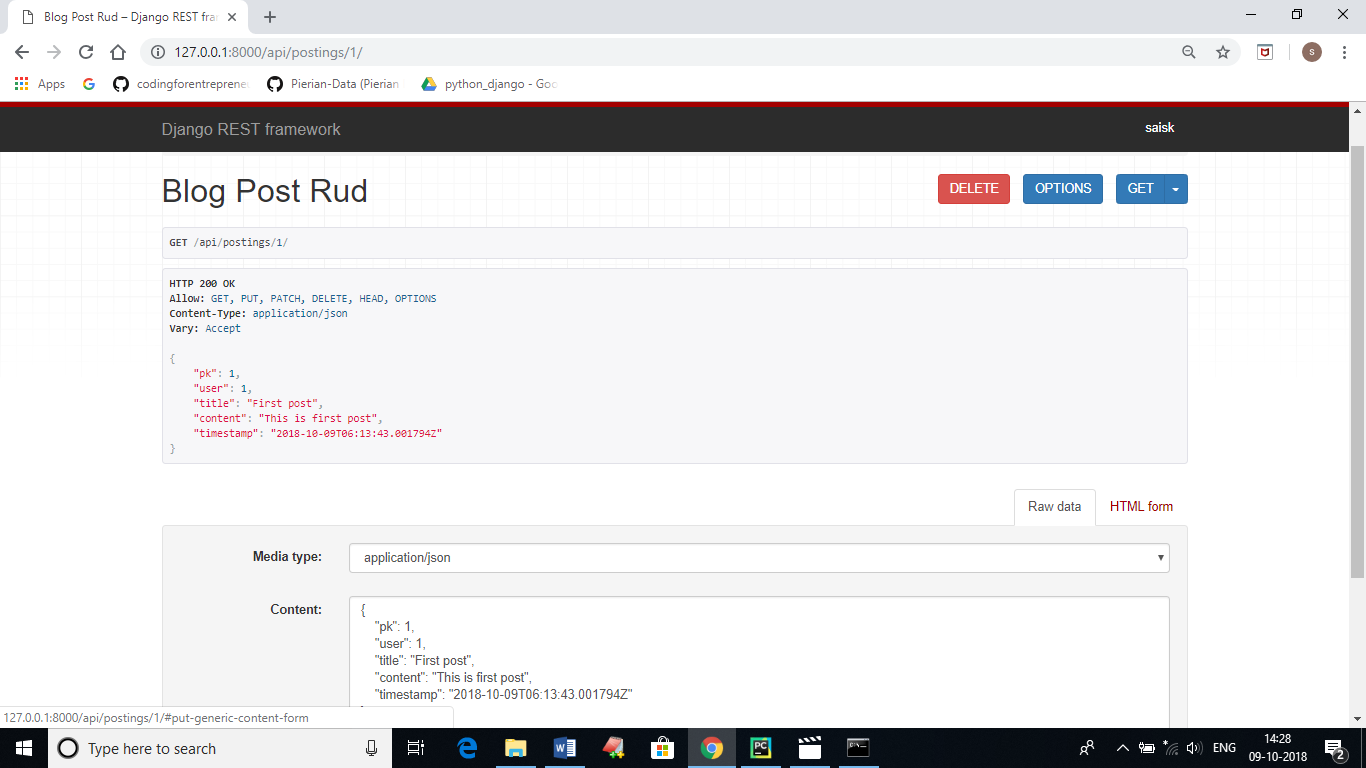
**from** django.conf.urls **import** url  
**from** .views **import** BlogPostRudView  
  
urlpatterns **=** [  
 url(**r'^(?P<pk>\d+)/$'**, BlogPostRudView.as\_view(), name**='post\_rud'**)  
]

Include this urls.py to the project level urls.py file.

sample\_project/urls.py:

*"""sample\_project URL Configuration  
  
The `urlpatterns` list routes URLs to views. For more information please see:  
 https://docs.djangoproject.com/en/1.11/topics/http/urls/  
Examples:  
Function views  
 1. Add an import: from my\_app import views  
 2. Add a URL to urlpatterns: url(r'^$', views.home, name='home')  
Class-based views  
 1. Add an import: from other\_app.views import Home  
 2. Add a URL to urlpatterns: url(r'^$', Home.as\_view(), name='home')  
Including another URLconf  
 1. Import the include() function: from django.conf.urls import url, include  
 2. Add a URL to urlpatterns: url(r'^blog/', include('blog.urls'))  
"""***from** django.conf.urls **import** url, include  
**from** django.contrib **import** admin  
  
urlpatterns **=** [  
 url(**r'^admin/'**, admin.site.urls),  
 url(**r'^api/postings/'**, include(**'postings.api.urls'**, namespace**='api\_postings'**)),  
]

The following will be the output, it is the api console



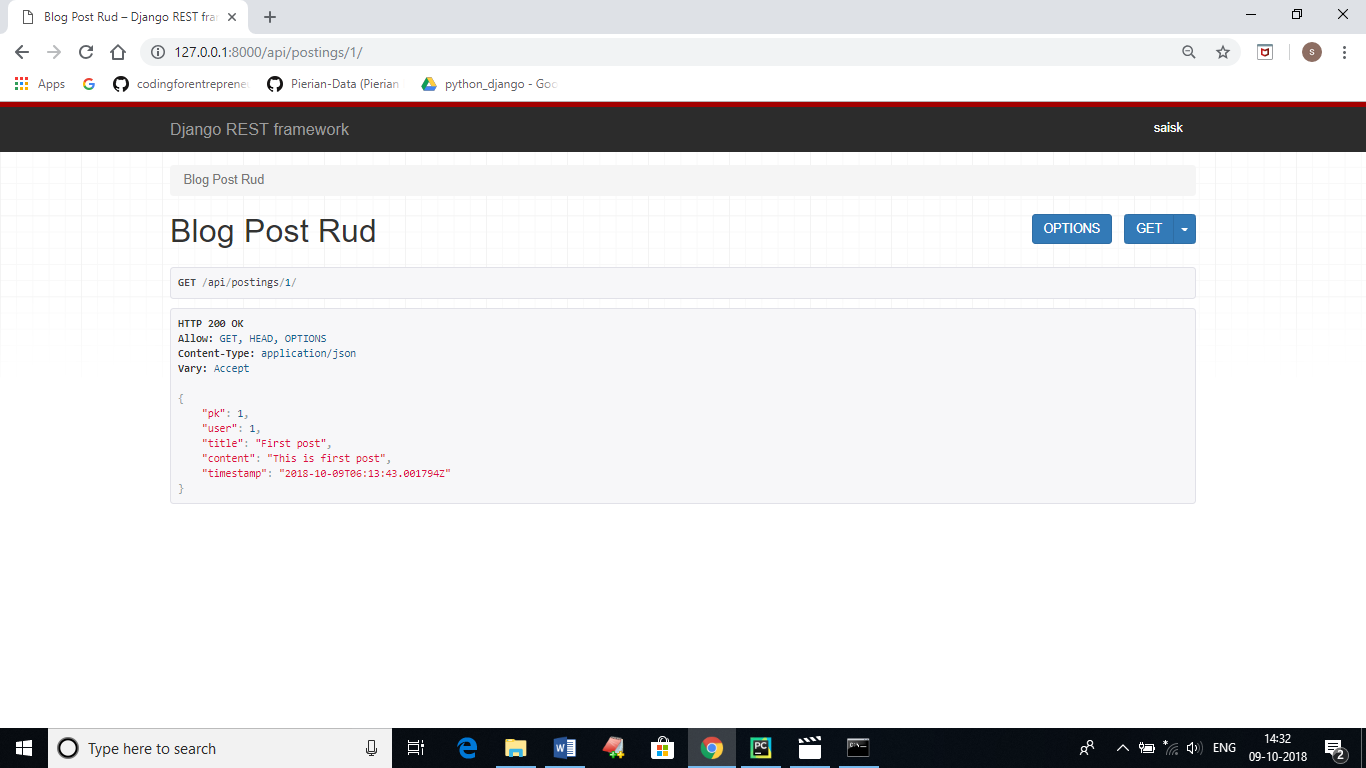
The following is the class which is present in postings/api/views.py.

**class BlogPostRudView**(generics.RetrieveUpdateDestroyAPIView)**:**

generics.RetrieveUpdateDestroyAPIView is responsible to get the above output.

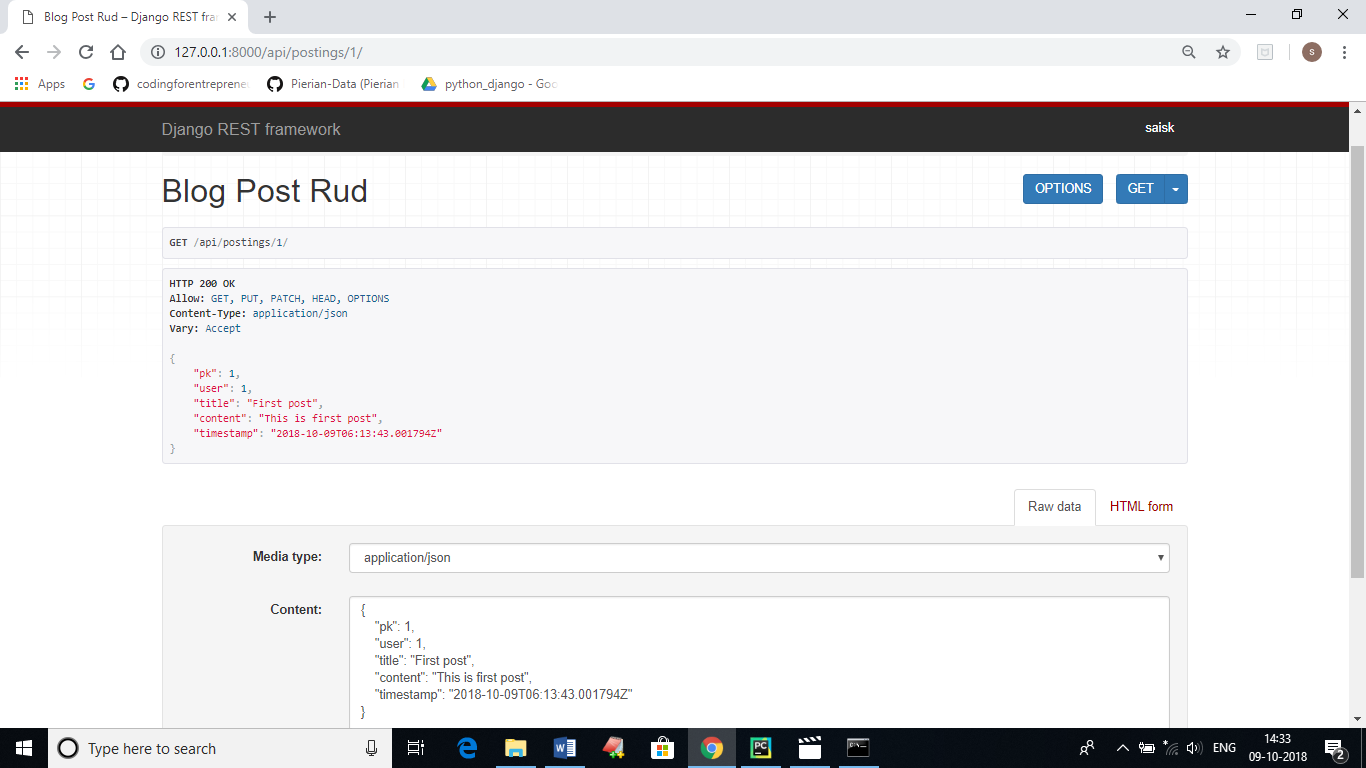
Now if we change the class to only retrieve then

**class BlogPostRudView**(generics.RetrieveAPIView)**:**



Similarly, for the class

**class BlogPostRudView**(generics.RetrieveUpdateAPIView)**:**



Now when you changed the value of pk and content in the api console the changes will be reflected.

To validate the title, we will create a validate\_title() method inside serializers.py

serializers.py:

**from** rest\_framework **import** serializers  
**from** postings.models **import** BlogPost  
  
**class BlogPostSerializer**(serializers.ModelSerializer)**:** *# same like forms.ModelForm* **class Meta:** model **=** BlogPost  
 fields **=** [  
 **'pk'**,  
 **'user'**,  
 **'title'**,  
 **'content'**,  
 **'timestamp'**,  
 ]  
 read\_only\_fields **=** [**'user'**] *# you will not get the error and no changes will reflect  
  
  
# serialiazer will do the following:  
# converts to JSON and valid  
# validations for data passed* **def validate\_title**(self, *value*)**:** qs **=** BlogPost.objects.filter(title\_\_iexact**=***value*)  
  
 **if** self.instance**:** qs **=** qs.exclude(pk**=**self.instance.pk)  
 *# the above two lines will not give error fot the same instance* **if** qs.exists()**:  
 raise** serializers.ValidationError(**"This title has already used"**)  
 **return** *value*

When we try to change the value of user then the changes will not reflect, as we made the user as read\_only\_fields.

When you try to enter the same title then we will get a error saying “This title has already used”.

**if** self.instance**:** qs **=** qs.exclude(pk**=**self.instance.pk)

When we try to click the PUT button for the second time, in order to not get the error message we are using the above two lines of code.

Now we will create CreateView in views.py file

**class BlogPostAPIView**(generics.CreateAPIView)**:** lookup\_field **= 'pk'** serializer\_class **=** BlogPostSerializer  
  
 **def get\_queryset**(self)**:  
 return** BlogPost.objects.all()  
  
 **def perform\_create**(self, *serializer*)**:** *serializer*.save(user**=**self.request.user)

Also do required changes inside the urls.py.

Now we will make CreateView and ListView inside views.py

views.py:

**from** rest\_framework **import** generics, mixins  
**from** postings.models **import** BlogPost  
**from** .serializers **import** BlogPostSerializer  
  
**class BlogPostRudView**(generics.RetrieveUpdateDestroyAPIView)**:** lookup\_field **= 'pk'** *# it can also be slug, id  
 # queryset = BlogPost.objects.all()* serializer\_class **=** BlogPostSerializer  
  
 **def get\_queryset**(self)**:  
 return** BlogPost.objects.all()  
  
 *# def get\_object(self):  
 # pk = self.kwargs.get('pk')  
 # return BlogPost.objects.get(pk=pk)  
  
  
# class BlogPostAPIView(generics.CreateAPIView):  
# lookup\_field = 'pk'  
# serializer\_class = BlogPostSerializer  
#  
# def get\_queryset(self):  
# return BlogPost.objects.all()  
#  
# def perform\_create(self, serializer):  
# serializer.save(user=self.request.user)  
  
# Make changes in the same above class***class BlogPostAPIView**(mixins.CreateModelMixin, generics.ListAPIView)**:** lookup\_field **= 'pk'** serializer\_class **=** BlogPostSerializer  
  
 **def get\_queryset**(self)**:  
 return** BlogPost.objects.all()  
  
 **def perform\_create**(self, *serializer*)**:** *serializer*.save(user**=**self.request.user)  
  
 *# This will just List the posts  
  
 # To make it to work also as a create view  
 # Add the post() method  
  
 # def post(self, \*args, \*\*kwargs):  
 # return #  
 # But this a hard code way of approach  
 # Instead we will use mixins  
  
 # This is for mixins approch* **def post**(self, *request*, *\*args*, *\*\*kwargs*)**:  
 return** self.create(*request*, **\****args*, **\*\****kwargs*)

Now we will implement the search operation.

Update the following inside the views.py file of api application.

**class BlogPostAPIView**(mixins.CreateModelMixin, generics.ListAPIView)**:** lookup\_field **= 'pk'** serializer\_class **=** BlogPostSerializer  
  
 **def get\_queryset**(self)**:** qs **=** BlogPost.objects.all()  
 query **=** self.request.GET.get(**"q"**)  
 **if** query **is not None:** qs **=** qs.filter(Q(title\_\_icontains**=**query)**|** Q(content\_\_icontains**=**query)  
 ).distinct()  
 **return** qs  
  
 **def perform\_create**(self, *serializer*)**:** *serializer*.save(user**=**self.request.user)  
  
 **def post**(self, *request*, *\*args*, *\*\*kwargs*)**:  
 return** self.create(*request*, **\****args*, **\*\****kwargs*)

**Permissions:**

When we try the same page (i.e. the same output) on other browser with <http://127.0.0.1:8000/api/postings/>

Then we will get error saying not granted permissions.

To make it as read only for the other users (unauthorized users) we will update settings.py file.

REST\_FRAMEWORK **=** {  
 **'DEFAULT\_PERMISSION\_CLASSES':** (  
 **'rest\_framework.permissions.IsAuthenticatedOrReadOnly'**,  
 ),  
 **'DEFAULT\_AUTHENTICATION\_CLASSES':** (  
 **'rest\_framework\_jwt.authentication.JSONWebTokenAuthentication'**,  
 **'rest\_framework.authentication.SessionAuthentication'**,  
 **'rest\_framework.authentication.BasicAuthentication'**,  
 ),  
}

Just we updated ‘rest\_framework.permissions.IsAuthenticatedOrReadOnly’.

If we provide permission\_classes = [], inside views.py, it means we are permitting all permissions to the others.

**class BlogPostAPIView**(mixins.CreateModelMixin, generics.ListAPIView)**:** lookup\_field **= 'pk'** serializer\_class **=** BlogPostSerializer  
 permission\_classes **=** []

We will create permissions.py file to make read only for unauthorized users.

permissions.py:

**from** rest\_framework **import** permissions  
  
**class IsOwnerOrReadOnly**(permissions.BasePermission)**:** *"""  
 Object-level permission to only allow owners of an object to edit it.  
 Assumes the model instance has an `owner` attribute.  
 """* **def has\_object\_permission**(self, *request*, *view*, *obj*)**:** *# Read permissions are allowed to any request,  
 # so we'll always allow GET, HEAD or OPTIONS requests.* **if** *request*.method **in** permissions.SAFE\_METHODS**:  
 return True** *# Instance must have an attribute named `owner`.* **return** *obj*.owner **==** *request*.user

Update models.py of postings application.

**class BlogPost**(models.Model)**:** user **=** models.ForeignKey(User, on\_delete**=**models.CASCADE)  
 title **=** models.CharField(max\_length**=**200, null**=True**, blank**=True**)  
 content **=** models.TextField(max\_length**=**200, null**=True**, blank**=True**)  
 timestamp **=** models.DateTimeField(auto\_now\_add**=True**)  
  
 **def \_\_str\_\_**(self)**:  
 return** str(self.user.username)  
  
 **@property  
 def owner**(self)**:  
 return** self.user

And update the views.py file api folder.

**class BlogPostAPIView**(mixins.CreateModelMixin, generics.ListAPIView)**:** lookup\_field **= 'pk'** serializer\_class **=** BlogPostSerializer  
 *# permission\_classes = [] # This is like permitting all permissions* permission\_classes **=** [IsOwnerOrReadOnly]

permissions\_classes is a array, which can allow many classes.

Now create a tests.py file inside api folder.

tests.py is used to test our api automatically.