django_api

Monday, February 28, 2022 5:57 AM

code/commands	Explanation
Building RESTful APIs with Django REST Framework (70m) Introduction What are RESTful APIs Resources Resource Representations HTTP Methods Installing Django REST Framework Creating API Views Creating Serializers Serializing Objects Creating Custom Serializer Fields Serializing Relationships Model Serializers Deserializing Objects Data ValidationSaving Objects Deleting Objects Exercise- Building the Collections API	

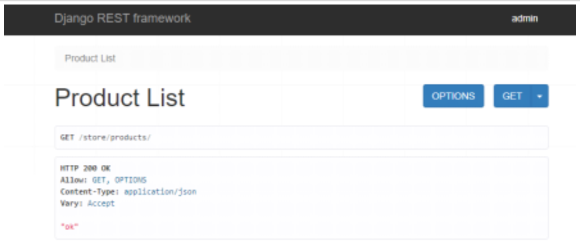
Setting up the project
 	
`# create a new project`
`django-admin` **startproject** `<project-name> .`
 	
`# create a new app`
`python manage.py` **startapp** `<playground>`
 	
`# create a new migartion file`
`python manage.py` **makemigrations**
 	
`# create actual changes in the database`
`python manage.py` **migrate**
 	
`# start the development web server`
`python manage.py` **runserver**
 	
`# creates a file having the required depencies for the project`
pip freeze `>requirements.txt`
 	
Set up the database for the project by goin to the setting module and set the "DATABASES". And set the configuration.
 	
`#creates super user`
`python manage.py` **createsuperuser**	

REST api There are 3 majar concepts for api: • Resources • Representations • HTTP methods	
Resources The resource in api is like object in our application like product,customers,etc. **URL(uniform resource locator)** is a **resource locator.** Each resource can be accessed using the url. and each resource may contain other resources. https://mosh.com/products/1	
Resource Representations It is the **output returned by the server** and these are the formats client understand. It can be • **HTML** • **XML** • **JSON**	
HTTP Methods Using these methods client can tell the server **what to do with the resource.** • **GET**: for **getting** a single or collection of resources. • **POST**: for **creating** a resource. • **PUT**: for **updating** the resource. • **PATCH**: for **updating a subset of properties** of the resource. • **DELETE**: for **deleting** a resource. Example: POST: Suppose a user hits a endpoint so the server knows we want to create a product. End point be like **"/products"** **Body of the request** need the title and price of the new product in json format.	

```
PUT: For updating all properties of the resource.
End point be like "/products"

PATCH: For updating a specific product.
End point be like "/products/1"

DELETE: For deleting a products. Its request doesn't need a body.
End point be like "/products/1"
```

```
Django REST framework

# install django rest framework
pip install djangorestframework

add 'rest_framework' in the list of installed apps in the settings module.
```

```
Creating API views
Views are like "controller" of MVC in django.
View function takes a request and returns a response.

Django has HttpRequest, HttpResponse methods. Django rest framwork also
has request and response method as Request and Response that is simpler
and better we'll use that.

ENDPOINT:
http://127.0.0.1:8000/store/products/

from rest_framework.decorators import api_view
from rest_framework.response import Response

step 1:
(store/views.py)
# creating a view function
# this decorator makes the "request" below instance of the Request class
# of rest_framework.
@api_view()
def product_list(request):
    return Response("ok")
# now map this function to a url pattern
# create a urls module/file in the working app

# the id comes from the URL and passed into the function here.
@api_view()
def product_detail(request,id):
    return Response(id)

step 2:
(store/urls.py)
# set the endpoint at which you want to return a reponse
# this is the url.py of the app,
urlpatterns = [
    path('products/', views.product_list),

# <id> is a parameter takes from the URL and pass it to view function.
# parameters can only be integer.
path('products/<int:id>', views.product_detail),
]

step 3:
(storefront/urls.py)
# also register the endpoint in the urls.py of the main project.

urlpatterns = [
    path('admin/', admin.site.urls),
    path('playground/', include('playground.urls')),

    # so if the url of the request starts with "store/" it will be
    # handled by "store.urls" module
    path('store/', include('store.urls')),

    path('__debug__/', include(debug_toolbar.urls)),
]
```

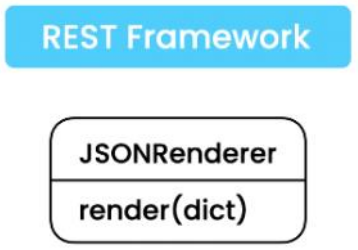


Product List

```
GET /store/products/
```

```
HTTP 200 OK
Allow: GET, OPTIONS
Content-Type: application/json
Vary: Accept

"ok"
```

```
Creating Serializers
A serializer can convert a model object to python dictionary.

Serializer:
Model object -> dictionary

And then this python dictionary can be converted to json using
"JSONRenderer".

JSONRenderer:
dictionary -> JSON

(store/serializers.py)
from rest_framework import serializers

# sometimes we dont want to show some data to the API
# that's why we make two representation of every
# model an internal representation and an external representation.

# this is external representation of the Product model and contains
# that fields which we want to expose in the response.
```



```
# NOTE: this model is completly independent of the internal Product model.
class ProductSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title=serializers.CharField(max_length=255)
    unit_price=serializers.DecimalField(max_digits=6,decimal_places=2)
```

Serializing Objects
All of the above object to json convertion takes place inside serializer object.

```
from django.shortcuts import get_object_or_404
from .models import Product
from .serializers import ProductSerializer

@api_view()
def product_list(request):
    queryset=Product.objects.all()

    # convert each product object to a dictioanary
    serializer=ProductSerializer(queryset,many=True)
    return Response(serializer.data)

     # now map this function to a url pattern
     # create a url module/file in the working app

@api_view()
def product_detail(request,id):

    # get the product object otherwise return a 404 error
    product = get_object_or_404(Product,id=id)

    # convert object to dictionary and then to json
    serializer = ProductSerializer(product)

     # by default the serializer will return a string for
    # unit_price but we've set it to a decimal.
    # to solve this problem add
    # REST_FRAMEWORK = {
    # "COERCE_DECIMAL_TO_STRING": False}
    # in the settings module.
    return Response(serializer.data)
```

Creating custom serializer fields

The **api models** is the external interface of our data whereas the **data model** is the internal interface.
Internal models remains constant whereas external may change depend on the usecase.

Lets add a new field in the api model that doesn't exist in data model.

```
from decimal import Decimal
from rest_framework import serializers
from store.models import Product


class ProductSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField(max_length=255)
    unit_price = serializers.DecimalField(max_digits=6, decimal_places=2)

    # now add a new field here that doesn't exist
    #  in the data model.
    # Create a custom function and set it to 'price_with_tax' field.
    price_with_tax =
     serializers.SerializerMethodField(method_name='calculate_tax')

     # type annotation for annotating that price is a object.
     def calculate_tax(self, price: Product):
        return price.unit_price * Decimal(1.1)
```

You can also change the name of the fields in api model but you have to tell the source of the field as the fields mismatch from the data model.

```
(serializers.py)
class ProductSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField(max_length=255)

    # this field mismatch from the data model so provide source
    #  field name from the data model.
    price = serializers.DecimalField(max_digits=6, decimal_places=2,
     source='unit_price')
```

Serializing Relationships
Serialize foreign key fields that have relationship in other models.

Methods:
- Primary keys
- String fields
- Nested objects
- Hyperlinks

Method.1(Using primary keys)

Using Nested objects:
```
[
  {
    "id": 648,
    "title": "7up Diet, 355 Ml",
    "price": 79.07,
    "price_with_tax": 86.977,
    "collection": {
      "id": 5,
      "title": "Stationary"
    }
  },
```

(serializers.py)
```
class ProductSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField(max_length=255)
    # this field mismatch from the data model so provide source
    #  field name from the data model.
    price = serializers.DecimalField(max_digits=6, decimal_places=2,
     source='unit_price')
    # now add a new field here that doesn't exist
    #  in the data model.
    # Create a custom function and set it to 'price_with_tax' field.
    price_with_tax =
     serializers.SerializerMethodField(method_name='calculate_tax')

    # this is a foreign field and it is serialized here
    # using the primary key
    collection=serializers.PrimaryKeyRelatedField(
        queryset=Collection.objects.all()
    )
```

(views.py)
```
@api_view()
def product_list(request):

    # load the foreign field 'collection' to prevent extra queries
    queryset=Product.objects.select_related('collection').all()

    # convert each product object to a dictioanary
    serializer=ProductSerializer(queryset,many=True)
    return Response(serializer.data)
```

Method.2(Using String field)

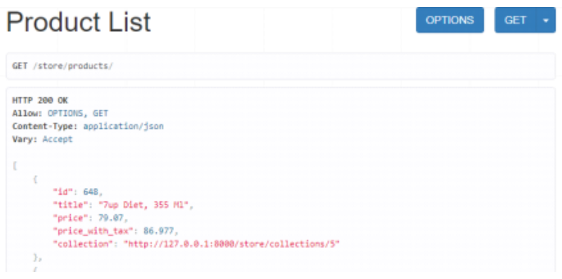
```
class ProductSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField(max_length=255)
    # this field mismatch from the data model so provide source
    #  field name from the data model.
    price = serializers.DecimalField(max_digits=6, decimal_places=2,
     source='unit_price')
    # now add a new field here that doesn't exist
    #  in the data model.
    # Create a custom function and set it to 'price_with_tax' field.
    price_with_tax =
     serializers.SerializerMethodField(method_name='calculate_tax')

    # this is a foreign field and it is serialized here
    collection=serializers.StringRelatedField()


@api_view()
def product_list(request):

    # load the foreign field 'collection' to prevent extra queries
    queryset=Product.objects.select_related('collection').all()

    # convert each product object to a dictioanary
    serializer=ProductSerializer(queryset,many=True)
    return Response(serializer.data)
```

Method.3(Nested Objects):

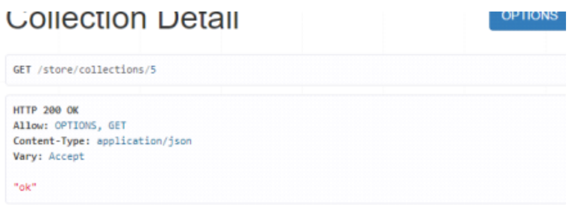
```
# create a collection serializer and pass it to the collection field.
class CollectionSerializer(serializers.Serializer):

    id = serializers.IntegerField()
    title = serializers.CharField(max_length=255)

class ProductSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField(max_length=255)

    # this field mismatch from the data model so provide source
    #  field name from the data model.
    price = serializers.DecimalField(max_digits=6, decimal_places=2,
     source='unit_price')

    # now add a new field here that doesn't exist
    #  in the data model.
    # Create a custom function and set it to 'price_with_tax' field.
    price_with_tax =
     serializers.SerializerMethodField(method_name='calculate_tax')

    # this is a foreign field and it is serialized here
    # using the primary key
    collection=CollectionSerializer()
```

Method.4(Using hyperlinks)
It will generate a link to the collection model.

(store/urls.py)
```
urlpatterns = [
    path('products/', views.product_list),
```



```
GET /store/products/

HTTP 200 OK
Allow: OPTIONS, GET
Content-Type: application/json
Vary: Accept

[
    {
        "id": 648,
        "title": "7up Diet, 355 Ml",
        "price": 79.07,
        "price_with_tax": 86.977,
        "collection": "http://127.0.0.1:8000/store/collections/5"
    },
```



```
GET /store/collections/5

HTTP 200 OK
Allow: OPTIONS, GET
Content-Type: application/json
Vary: Accept

"ok"
```

```
    path('products/<int:id>', views.product_detail),
    # django rest_framework has a convention for URL
    # argument for hyperlink type to be pk.
    path('collections/<int:pk>',
         views.collection_detail, name='collection-detail'),
]

(views.py)
@api_view()
def product_list(request):
    # load the foreign field 'collection' to prevent extra queries
    queryset = Product.objects.select_related('collection').all()

     # context is required for hyperlinked serializer
    serializer = ProductSerializer(
        queryset, many=True, context={'request': request})
    return Response(serializer.data)


@api_view()
def collection_detail(request, pk):
    return Response('ok')


(serializers.py)
class ProductSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField(max_length=255)
    # this field mismatch from the data model so provide source
    #  field name from the data model.
    price = serializers.DecimalField(
        max_digits=6, decimal_places=2, source='unit_price')
    # now add a new field here that doesn't exist
    #  in the data model.
    # Create a custom function and set it to 'price_with_tax' field.
    price_with_tax = serializers.SerializerMethodField(
        method_name='calculate_tax')

    # this is a foreign field and it is serialized here
    # using the primary key.
    # 'view_name' is the URL to be redirected to.
    collection = serializers.HyperlinkedRelatedField(
        queryset=Collection.objects.all(),
        view_name='collection-detail',
    )
```

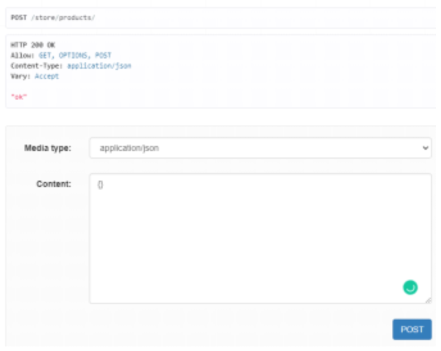
Model Serializers

So we know we have internal and external models and lets say if we need to change validation rule for title of the product then we've to change both the serializer(api model/external model) and product model(internal model).
To avoid this we use model serializers. This model serializer will first look the fields here(Product serializer class) and if it doesn't find it here it will look for it in the product model.

```
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        # give the model which we want to serialize
        model = Product

         # give the fields that we want to serialize include the
        # extra fields here that are not in the product model if any.
       #  order of the fields matter.
        fields = ['id', 'title', 'unit_price', 'price_with_tax',
        'collection']

     #  model serializer will first look for 'price_with_tax'
    #  in the Product model and if it doesn't find it, it will look
    # it here in the 'ProductSerializer' class.
    price_with_tax = serializers.SerializerMethodField(
        method_name='calculate_tax')

    def calculate_tax(self, price: Product):
        return price.unit_price * Decimal(1.1)
```

Deserialization of objects
Opposite of serialization.
Required when client POST a request with some data.
You can see the **empty "{}" request** that returns a **"ok"** response.

```
@api_view(['GET', 'POST'])
def product_list(request):
    if request.method =='GET':
        queryset = Product.objects.select_related('collection').all()
        serializer = ProductSerializer(
            queryset, many=True, context={'request': request})
        return Response(serializer.data)

    # here we ill define the POST request logic
    elif request.method == 'POST':
        # Deserialize the data in the POST request.
        serializer=ProductSerializer(data=request.data)
        return Response("ok")
```



```
POST /store/products/

HTTP 200 OK
Allow: GET, OPTIONS, POST
Content-Type: application/json
Vary: Accept

"ok"
```

Media type: application/json

Content: {}

Data Validation
Validate the incoming data otherwise reject the coming data through the POST request.

```python
@api_view(['GET', 'POST'])
def product_list(request):
    if request.method =='GET':
        queryset = Product.objects.select_related('collection').all()
        serializer = ProductSerializer(
            queryset, many=True, context={'request': request})
        return Response(serializer.data)

    # here we ill define the POST request logic
    elif request.method == 'POST':
        serializer=ProductSerializer(data=request.data)

        # checking if the data is valid, otherwise return an error.
        # validation means like in signup form, we need to check
        # if the password and confirm password field match otherwise don’t
        # allow the data to process further.
       # you can make that custom method in the serializer.py module for
         # validation
        serializer.is_valid(raise_exception=True)
        serializer.validated_data
        return Response(serializer.data)
```

Below is the output validated data:

```
OrderedDict([('title', 'a'),
 ('unit_price', Decimal('2.00')),
 ('collection', <Collection: Grocery>)])
```

Saving Objects
ModelSerializer class has a **.save()** method for creating and updating a product. This save method will automatically create or save objects depending upon the state of the serializer.

```python
from rest_framework import status

@api_view(['GET', 'POST'])
def product_list(request):
    if request.method =='GET':
        queryset = Product.objects.select_related('collection').all()
        serializer = ProductSerializer(
            queryset, many=True, context={'request': request})
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer=ProductSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)

        # using ".save()" method we dont have to use
        #  "serializer.validate_data"
        serializer.save()

        # print(serializer.validated_data)
        return Response(serializer.data)

@api_view(['GET', 'PUT'])
def product_detail(request, id):
    product = get_object_or_404(Product, id=id)
    if request.method == 'GET':
        serializer = ProductSerializer(product)
        return Response(serializer.data)
    elif request.method == 'PUT':

       # for PUT get the data, validate it and save the product
        #  object in the database
        # Also for updation we need to pass product instance because
        #  the serializer will try to update the product instance
        #  with the data in the request.
        serializer = ProductSerializer(product, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()

       # give the 201 status code for successful update
        return Response(serializer.data,status=status.HTTP_201_CREATED)
```

Creating a object:
Just post using the test data.

```
{
    "title": "a",
    "slug": "a",
    "inventory":1,
    "unit_price": 1,
    "collection": 1
}
```

And the object will be created automatically.

For updation
Using the PUT request use the data below and PUT a request.
http://127.0.0.1:8000/store/products/2

```
{
    "title": "Island Oasis - Raspberry",
    "slug": "-",
    "inventory": 40,
    "description": "maecenas tincidunt lacus at velit vivamus vel nulla eget eros
     elementum pellentesque",
        "unit_price": 84.64,
        "collection": 3
     }
```

Added and extra **"+" at the beginning of title.**The update data becomes:

```
{
    "title": "+Island Oasis - Raspberry",
    "slug": "-",
    "inventory": 40,
    "description": "maecenas tincidunt lacus at velit vivamus vel nulla eget eros
     elementum pellentesque",
        "unit_price": 84.64,
        "collection": 3
     }
```

Deleting objects

(views.py)

```python
@api_view(['GET', 'PUT', 'DELETE'])
def product_detail(request, id):
    product = get_object_or_404(Product, id=id)
    if request.method == 'GET':
        serializer = ProductSerializer(product)
        return Response(serializer.data)
    elif request.method == 'PUT':
        # for PUT get the data, validate it and save the product
        #  object in the database
        # Also for updation we need to pass product instance because
        #  the serializer will try to update the product instance
        #  with the data in the request.
        serializer = ProductSerializer(product, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        # give the 201 status code for successful update
        return Response(serializer.data, status=status.HTTP_201_CREATED)
    elif request.method == 'DELETE':

        product.delete()
        # because orderitem_set is a foreign key to product, so
        # product cant be deleted..
        # we've changed the orderitem_set to orderitems for a good name
        if product.orderitems.count() > 0:
            return Response(status=status.HTTP_405_METHOD_NOT_ALLOWED)

        # normally the delete request dont have any return value
        # but it depend on you can also return deleted object.
        return Response(status=status.HTTP_204_NO_CONTENT)
```

(model.py)

```
class OrderItem(models.Model):
    order = models.ForeignKey(Order, on_delete=models.PROTECT)

    # we've changed the OrderItem class name in the Product model to
     #'orderitems'
    product = models.ForeignKey(Product, on_delete=models.PROTECT,
     related_name='orderitems')
```

Exercise:Building the Collections API

Using **"related_name="** is used for overwriting djangos default convention for naming related/foreign fields.

The **related_name** attribute specifies the name of the reverse relation from the User model back to your model.
If you don't specify a related_name, Django automatically creates one using the name of your model with the suffix **_set**, for instance **User.map_set.all()**.

(store/urls.py)

```
urlpatterns = [
    path('products/', views.product_list),
    path('products/<int:id>', views.product_detail),

    # creating an endpoint for listing collections
    path('collections/', views.collection_list),

    path('collections/<int:pk>',
         views.collection_detail, name='collection-detail'),
]
```

(serializer.py)

```
class CollectionSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField(max_length=255)
    class Meta:
        model=Collection
        fields=['id','title','products_count']

    # because the collection class doesnt have this field
    # we have to define it here
    products_count=serializers.IntegerField()

# two enddpoints for getting and creating a new collection
@api_view(['GET', 'POST'])
def collection_list(request):
    if request.method == 'GET':
        queryset = Collection.objects.annotate(
            products_count=Count('products')).all()
        serializer = CollectionSerializer(queryset, many=True)
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = CollectionSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
```

(models.py)

```
class Product(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField()
    description = models.TextField(null=True, blank=True)
    unit_price = models.DecimalField(
        max_digits=6,
        decimal_places=2,
        validators=[MinValueValidator(1)])
    inventory = models.IntegerField(validators=[MinValueValidator(0)])
    last_update = models.DateTimeField(auto_now=True)

     # so using related_field each collection has an
    # attribute products.
    collection = models.ForeignKey(Collection, on_delete=models.PROTECT,
     related_name='products')

    promotions = models.ManyToManyField(Promotion, blank=True)
    def __str__(self) -> str:
        return self.title
    class Meta:
        ordering = ['title']

@api_view(['GET', 'PUT', 'DELETE'])
def collection_detail(request, pk):
    collection = get_object_or_404(
        Collection.objects.annotate(
            products_count=Count('products')), pk=pk)
    if request.method == 'GET':
        serializer = CollectionSerializer(collection)
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = CollectionSerializer(collection, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)

    elif request.method=='DELETE':
        if collection.products.count() > 0:
            return Response({'error':'cant delete because products are
           available'}})

        collection.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
    return Response('ok')

(views.py)
# two endpoints for getting and creating a new collection
@api_view(['GET', 'POST'])
def collection_list(request):
    if request.method == 'GET':
        queryset = Collection.objects.annotate(

            # products is the field in the product model
            products_count=Count('products')).all()
        serializer = CollectionSerializer(queryset, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = CollectionSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)

(store/admin.py)
@admin.register(models.Collection)
class CollectionAdmin(admin.ModelAdmin):
    autocomplete_fields = ['featured_product']
    list_display = ['title', 'products_count']
    search_fields = ['title']
    @admin.display(ordering='products_count')
    def products_count(self, collection):
        url = (
            reverse('admin:store_product_changelist')
            + '?'
            + urlencode({
                'collection__id': str(collection.id)
            }))
        return format_html('<a href="{}">{} Products</a>', url,
        collection.products_count)
    def get_queryset(self, request):
        return super().get_queryset(request).annotate(

          #here we also change products because previously it was product.
            products_count=Count('products')
        )
```

Advanced API concepts

1. Class-based views.
2. Generic views.
3. viewsets.
4. routers.
5. searching, filtering and paginiation.