In Reference to Linkedin Learning Course Building the RESTful web API with Django

https://www.linkedin.com/learning-login/share?forceAccount=false&redirect=https%3A%2F%2Fwww.linkedin.com%2Flearning%2Fbuilding-restful-web-apis-with-django%3Ftrk%3Dshare_ent_url&account=56973065

Building the RESTful web API with Django

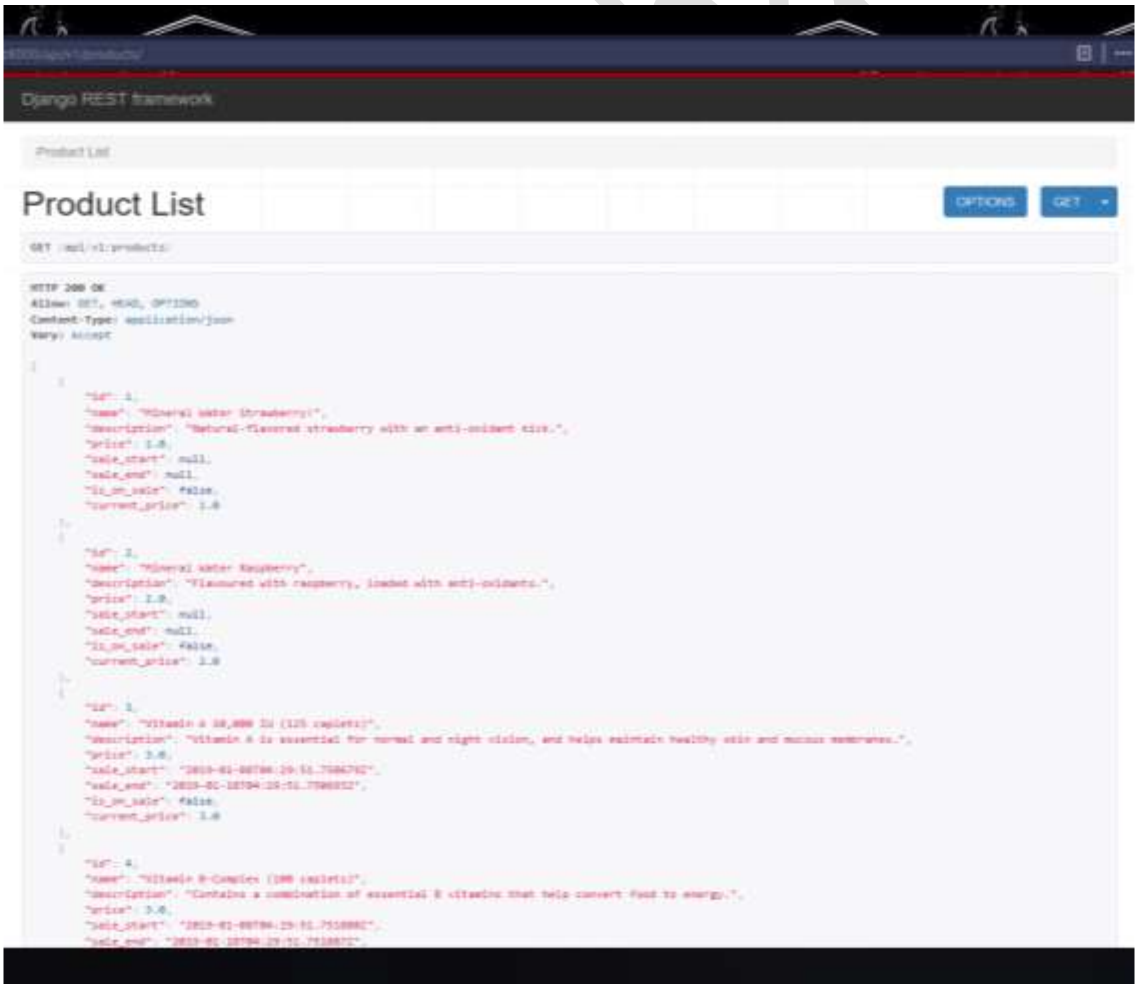
Serializing, Listing, Filtering, and Paginating Models

- We, will start with writing a serializer to serialize our model.
- Serializing means to convert the data into a format like JAON, YAML, and XML.
- So, we need to take our product model in this example and serialize it to JSON. This will then be served through our REST API.
- So, we are going to serialize the following fields:
  - Id
  - Name
  - Price
  - Sale Start Date
  - Sale End Date
- So, create the serializers.py file in the application folder. Then we write the following code in it:
  - So, we are importing the serializer from the rest_framework.
  - from rest_framework import serializers
  - Then we also import the Product model from the store.model.
  - from store.models import Product
  -
  - We, then create a model serializer.
  - class ProductSerializer(serializers.ModelSerializer):
  - This class has a meta class.
  -     class Meta:
  - Then we set the model that we want to serialize.
  -         model = Product
  - And also set the fields we want to serialize.
  -         fields = ('id', 'name', 'description', 'price', 'sale_start', 'sale_end')
  -
  - We, can also modify the serialize representation and add some fields to it. To do this we override the to_representation metod.(It takes in self and instance)
  -     def to_representation(self, instance):
  -
  - We, call the parent class "to_representation" implementation and set it to data.
  -         data = super().to_representation(instance)
  - Then we add to the data the following fields.
  - Boolean value for product being on sale or not.
  -         data['is_on_sale'] = instance.is_on_sale()
  - Then depending on if the product is on sale or not, we have our current price.
  -         data['current_price'] = instance.current_price()
  - So, we added both the fields now we can return the data.
  -         return data

- Now, we go the Django shell to try this out.
- After activating our virtual environment. We run the following command to enter the shell:
    - Python manage.py shell
- Then we are going to import our model.
    - From store.models import Product
    - Then we take the first product in the database.
    - Product = Product.objects.all()[0]
    - Then we import the serializer that we just created.
    - From store.serializers import ProductSerializer
    - Then we instantiate the serializer.
    - Serializer = ProductSerializer()
    - Then we serialize the product with to_representation.
    - Data = serializer.to_representation(product)
    - Then to get this in the JSON format we are going to import the JSON Renderer and render the model into the json format.
    - From rest_framework.renderer import JSONRenderer
    - Renderer = JSONRenderer()
    - Renderer.render(data)
    - Output:
    - b'{"id":1,"name":"Mineral Water Strawberry!","description":"Natural-flavored strawberry with an anti-oxidant kick.","price":1.0,"sale_start":null,"sale_end":null,"is_on_sale":false,"current_price":1.0}'
    - So, as we can see that we serialize the data and then we rendered it into the JSON format using the JSON Renderer.
    - Django shell is not only used for serializing the data but also has application in testing and debugging.
- Now, that we have created a serialized for product. Therefore, we have the product serializer. Now we need to create a list API View and return product listing from our API.
- A List API View is a generic Class based View, and the Django REST Framework provides the following Generic Views:
    - List API View
    - Create API View
    - Destroy API View
    - Retrieve Update Destroy API View
- These help in speeding up the development process.
- So, first we create a api_views.py file inside the store project directory. And program inside it as follows:
    - So, we import the ListAPIView from the rest_framework.generic
    - from rest_framework.generics import ListAPIView
    -
    - We import our serializer and model.
    - from store.serilizer import ProductSerilizer
    - from store.models import Product
    -
    - Then we create a class for our Listing and inherit the ListAPIView.
    - class ProductList(ListAPIView):

- We, set the queryset and serializer class. And that's it the rest is taken care by the generic class in the rest_framework.
- queryset = Product.objects.all()
- serializer_class = ProductSerilizer
- In most of the cases we are just going to use the Django REST framework's generic API views and mixins.
- In only the rare case we are going to use the base APIView to build up the API from the ground up.

- Now, we move to the next step of how to connect the Product API List View to a URL route.
- We start by opening up the urls.py file.
- And then we add the following url path:
    - path('api/v1/products/',store.api_views.ProductList.as_view()),
    - Also, we need to import the api_views.py that we just created:
        - Import store.api_views
    - The good this about the Django api is that even in this stage we can check if everything is working by running the server:
        - Python manage.py runserver
        - And the we can go to the link that we just added.
        - And we will get an output like this:



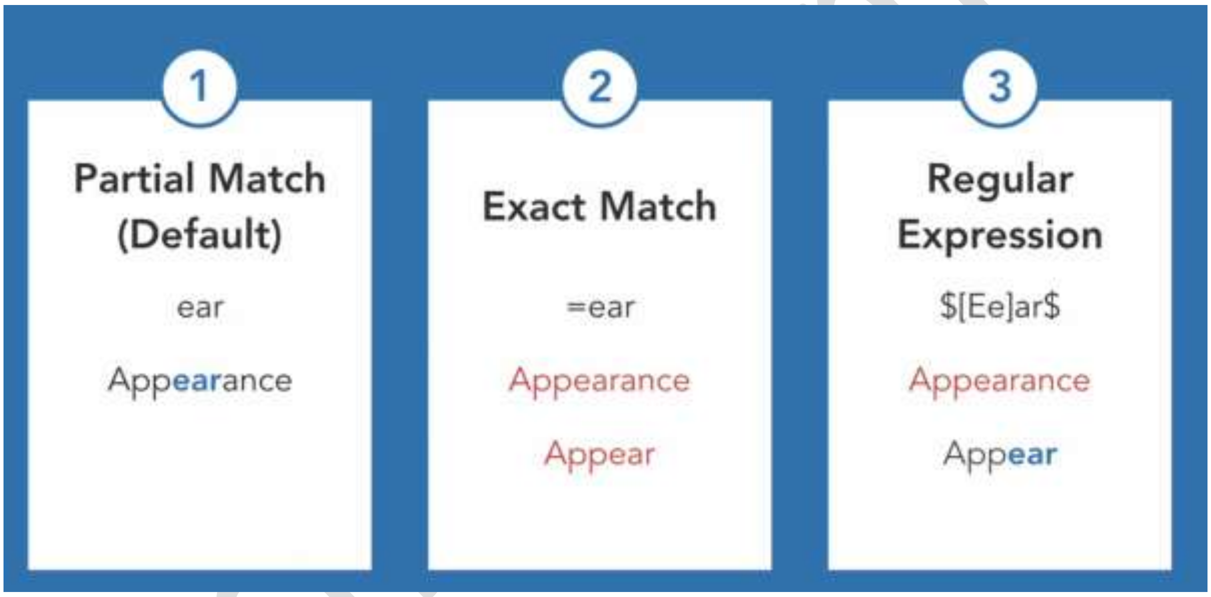
Product List

```
GET /api/v1/products/

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

[
    {
        "id": 1,
        "name": "Mineral Water Strawberry!",
        "description": "Natural-flavored strawberry with an anti-oxidant kick.",
        "price": 1.0,
        "sale_start": null,
        "sale_end": null,
        "is_on_sale": false,
        "current_price": 1.0
    },
    {
        "id": 2,
        "name": "Mineral Water Raspberry",
        "description": "Flavoured with raspberry, loaded with anti-oxidants.",
        "price": 2.0,
        "sale_start": null,
        "sale_end": null,
        "is_on_sale": false,
        "current_price": 2.0
    },
    {
        "id": 3,
        "name": "Vitamin A 10,000 IU (125 caplets)",
        "description": "Vitamin A is essential for normal and night vision, and helps maintain healthy skin and mucous membranes.",
        "price": 3.0,
        "sale_start": "2019-01-08T04:29:51.[illegible]",
        "sale_end": "2019-01-18T04:29:51.[illegible]",
        "is_on_sale": false,
        "current_price": 3.0
    },
    {
        "id": 4,
        "name": "Vitamin B-Complex (100 caplets)",
        "description": "Contains a combination of essential B vitamins that help convert food to energy.",
        "price": 3.0,
        "sale_start": "2019-01-08T04:29:51.[illegible]",
        "sale_end": "2019-01-18T04:29:51.[illegible]",
```

- So, we can see the data in json format.

- So, now we have the ProductAPI using the Product model we can create product filtering using url query parameter.
- In specific we are going to match the product to identify if the match a particular id or they are on sale or not.
- To, do this we are going to modify the api_views.py file as follows:
    - from rest_framework.generics import ListAPIView
    - We import the Django Filter Backend form Django filters.rest_framework
    - from django_filters.rest_framework import DjangoFilterBackend
    -
    - from store.serializers import ProductSerializer
    - from store.models import Product
    -
    - class ProductList(ListAPIView):
    -   queryset = Product.objects.all()
    -   serializer_class = ProductSerializer
    - We, then set the backend filter and set filter to id to filter it on the bases of id.
    -   filter_backends = (DjangoFilterBackend,)
    -   filter_fields = ('id',)
- For the filter for on sale item it is a bit different.
- So, we override the get_queryset():
    - And write the code as following:
    -   def get_queryset(self):
    - And then we take the value of parameter passed.
    -     on_sale = self.request.query_params.get('on_sale', None)
    - If the parameter is not given then just return everything.
    -     if on_sale is None:
    -       return super().get_queryset()
    -     queryset = Product.objects.all()
    - Or if the on_sale was set to true then we check the data when the sale started and the sale end date to get to know if the item is still on sale.
    -     if on_sale.lower() == 'true':
    -       from django.utils import timezone
    -       now = timezone.now()
    - Here, we are checking for if the values are inside the required condition for an item to be on sale. If satisfied the filtered set is returned.
    -       return queryset.filter(
    -         sale_start__lte= now,
    -         sale_end__gte=now,
    -         )
    - Else return everything.
    -     return queryset
- In the next step we are going to enable full text search so that we can search through the product names and descriptions.
- We are going to use the Search filter which is a filter back end build into Django REST Framework. We use it in the same way as we use the filter backend by adding it to the product list filter back end list.
- We import the search filter as following:

- from rest_framework.filters import SearchFilter
- Then we edit the Product listing class as following:
  - class ProductList(ListAPIView):
  - queryset = Product.objects.all()
  - serializer_class = ProductSerializer
  - SearchFillter is added into the filter_backends
  - filter_backends = (DjangoFilterBackend, SearchFilter)
  - filter_fields = ('id',)
  - And we create a search filter fields list that search filter uses to se
  - search_fields = ('name', 'description')
- Through the search field variable, we can also control how the search for matching text is conducted. The default matching is a case insensitive partial matching. We can add an equal's sign if we want to add an exact match. It is also possible to use the regular expression for matching.



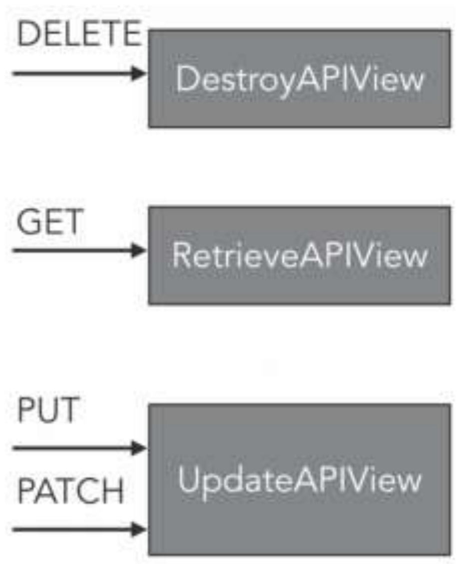
- If we have a lot of results that can be retrieved then we can use the page number pagination.
- We use the page number to paginate the results.
- Then the advanced version of this is the limit offset pagination. This provides the users to pass two parameters. Use a limit (That controls the number of results that can appear on a page) and (the page offset that controls which page appears) offset fields to more finely paginate the results.
- Then there a cursor pagination that uses a database cursor to paginate the results. This is effective when having a very large dataset and using the other pagination types will be to inefficient.
- So, to do this in practical add import line for importing the Limit Offset Pagination from rest_framework.pagination.
- Then we create the following class to provide some control over pagination.
  - class ProductPagination(LimitOffsetPagination):
  - So, extendin the LimitOffsetPagination class to set the default and limit values for the pagination limit that controls the number of results shown on a page.
  - default_limit = 10

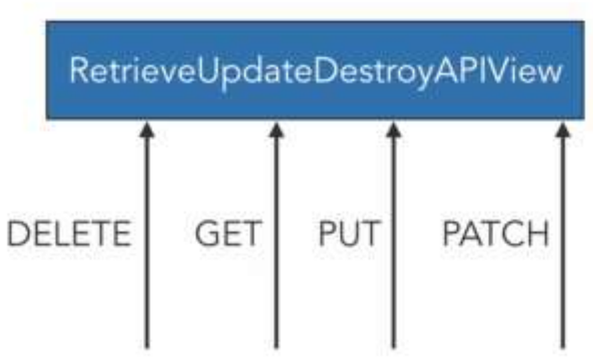
  - max_limit = 100
- Call to this is made in the Product List class.
  - Pagination_class = ProductPagination

Creating, Retrieve, Update, and Delete (CRUD) Operations for Models

- Now the next step is to have the ability to create new products through the api.
- In the api_views.py then create the product creating api view.
- From that first import CreateAPIView from the rest_framework.generics.
- Also, import ValidationError from rest_framework.exceptions.
- Then for creating a view for creating products add the code to api_views.py as follows:
  - Create a class that inherits the CreateAPIView.
  - class ProductCreate(CreateAPIView):
  - Then instantiate the serializer_class.
  - serializer_class = ProductSerializer
  - Then override the create method.
  - def create(self, request, *args, **kwargs):
  - try:
  - Extract the price form the create method.
  - price = request.data.get('price')
  - Then check if price is set and above zero dollar.
  - if price is not None and float(price) <= 0.0:
  - If not match raise a validation error.
  - raise ValidationError({ 'price': 'Must be above $0.00'})
  - except ValueError:
  - If not even a number then this is going to through an exception that we can catch and raise a validation error for invalid input.
  - raise ValidationError({'price': 'A valid number is required'})
  - If every thing is checked then we call the create method.
  - return super().create(request, *args, **kwargs)
- In the raising of validation error, we can attach a custom error message for a particular field.
- Instead of creating a model thought the admin interface rest api can be used to create new interfaces that are very specific to certain scenario.
- On such example is, importing the data from data sources like, Excel spreadsheet, XML file, JSON file, or other databases. This can load the data form each column from the spreadsheet and load the data accordingly into the database. Then import the data through the rest api. Then populating the database. Hence, rest api can be used to load a lot of data very quickly then through a user-friendly web interface.
- Then the next step is to create a route in the urls.py for the product creating view.
  - path('api/v1/products/new', store.api_views.ProductCreate.as_view()),
  - Then we can use the curl script to share the working model development with the team.
- The curl command with the output is mentioned as follows:
  - C:\Users\anric>curl -X POST http://localhost:8000/api/v1/products/new -d price=1.00 -d name='My_Product' -d description='Hello_World'
  -
  - {"id":6,"name":"'My_Product'","description":"'Hello_World'","price":1.0,"sale_start":null,"sale_end":null,"is_on_sale":false,"current_price":1.0}

-
    -
    -
    - C:\Users\anric>curl -X POST http://localhost:8000/api/v1/products/new -d price=1.00 -d name="My Product" -d description="Hello World"
    -
    - {"id":7,"name":"My Product","description":"Hello World","price":1.0,"sale_start":null,"sale_end":null,"is_on_sale":false,"current_price":1.0}
- Now, the next step is to delete the products from the database. For this there is DestroyAPIView to do that. First import its DestroyAPIView form the rest_framework.generics.
- Then create the destroy view.
    - class ProductDestroy(DestroyAPIView):
    - these two are the only basic requirement for creating a destroyer.
    -   queryset = Product.objects.all()
    -   lookup_field = 'id'
    - But, in the real world if a product is getting destroyed then all the caches related to that must also be destroyed. Freeing up cache means that more memory is available for models that are sill been used.
    - Override the delete method.
    -   def delete(self, request,*args, **kwargs):
    - Extract the product id form the request.
    -     product_id = request.data.get('id')
    - Then proceeded with deleting the object.
    -     responce = super().delete(request, *args, *kwargs)
    - And if the product was deleted successfully.
    -     if responce.status_code == 204:
    - Import the cache.
    -       from django.core.cache import cache
    - And delete the cache data for that id.
    -       cache.delete('product_data_{}'.format(product_id))
    - Then return the response.
    -     return responce
- Then comes the part of connecting it to the urlpattern.
    - path('api/v1/products/<int:id>/destroy', store.api_views.ProductDestroy.as_view()),
- To test it the url would look like:
    - http://localhost:8000/api/v1/products/5/destroy
    - Then click on the delete button to delete the product.
- For deleting it from the curl cmd line:
    - Curl -X DELETE http://localhost:8000/api/v1/products/5/destroy





- So, then for everything and every time we want the functionalities like deleting, getting, updating, adding we need to create a separate view.
- Then is a Django rest framework generic view that combines all of these features.
- With the RetriveUpdateDestroyAPIView we can reuse the Django rest framework generic view.
- We can reuse the code and configuration.
- For example: the serializer_class or queryset
- One URL can be used to handle multiple HTTP methods.
- To do this we refactor the DestroyAPIView.
- After refactoring the code for destroy, the code would look like:
    - The change is made in the name of the class and the class that it extends.
    - class ProductRetrieveUpdateDestroy(RetrieveUpdateDestroyAPIView):
    -     queryset = Product.objects.all()
    - We set the lookup field to id.
    -     lookup_field = 'id'
    - Delete function in the same way as previously.
    -     def delete(self, request,*args, **kwargs):
    -         product_id = request.data.get('id')
    - Don't forget the serializer class.
    -         serializer_class = ProductSerializer
    -         responce = super().delete(request, *args, *kwargs)
    -         if responce.status_code == 204:
    -             from django.core.cache import cache
    -             cache.delete('product_data_{}'.format(product_id))
    -         return response
    - In the update method we take in the same parameters as the delete or update. We override the update method.
    -     def update(self, request, *args, **kwargs):
    - Then this method call is the one which does the updating. And we call the super update method.
    -         response = super().update(request, *args, **kwargs)

- If the response from the update method is successful that is 200.
- if response.status_code == 200:
- Then import the cache.
- from django.core.cache import cache
- Then take in the response data and set it to product.
- product = response.data
- To set the cache data we need to specify the format in which we need to save that.
- cache.set('product_data{}'.format(product['id']),
- {'name':product['name'],
- 'description':product['description'],
- 'price':product['price'],
- })
- And lastly, we return the response message.
- return response

- After this we need to update the url configuration to update the destroy refactoring.
  - path('api/v1/products/<int:id>/', store.api_views.ProductRetrieveUpdateDestroy.as_view()),
  - So, changing the destroy url path as follows will give us the desired results.

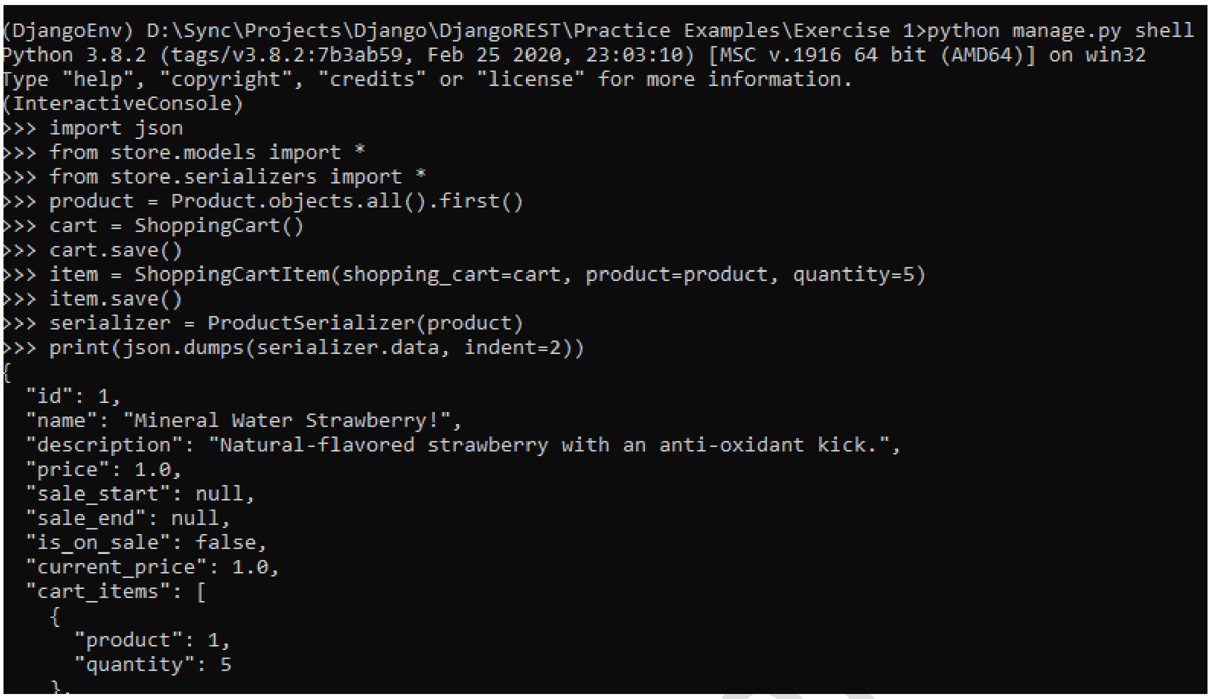
Managing Serializer Fields, Relations, and Validation

Serializer with only selected fields

- So, the changes can be made to the serializer class.
- Right now, is done by selecting some fields for serializer.
- But by doing some refactoring this can be achieved more easily. It can be done taking the attributes that are set in the to_representation and refactor them by using serializer fields.
- So, adding first a Boolean field for is on sale and setting it to read only.
- Then for current price it is a Float field which is also read only.
- Then add them to the meta fields so, they appear in the serializer.
- Talking about the serializer field configuration, there is a read_only which is for whether or not the field can be written to through the serializer.
- Another example of serializer field configuration is the source key word.
- Source is where the data for the serializer field will be populated from
- product_name = serializer.CharField(source=’name’)
- This can be used if it is required to rename a field.
- So, in the above example we are renaming the name field to product_name.
- So, then a description field can be added as following:
  - description = serializers.CharField(min_lenght=2,max_lenght=200)
  - //max and min length are not working.
  - So, after a few checks it worked with the order being
    - max_length=100, min_length=5
  - This is overriding the description in the serializer with a Char field. And it’s already in the meta field.
  - With minimum and maximum length set we also have validation.

Serializer that shows model relationships

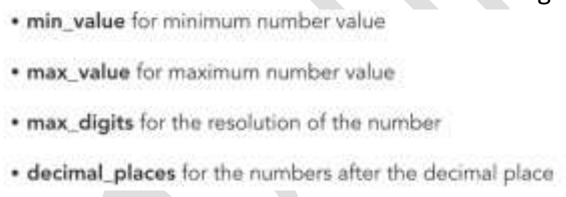
- So, for each customer there is a shopping cart. But from a developer prospective there are multiple shopping carts to look at.
- If required to build a sales report there are some questions that needs to be answered.
- How many particular items are sold or something around that line of analysis?
- So, for that it is required to go to the serializers.py the we need to create a shopping cart serializer.
- To do that add the following code:
    - class CartItemSerializer(serializers.ModelSerializer):
    - In the meta it is set that the model is a Shopping Cart Item and fields are product and the quantity.
    -    class Meta:
    -       model = ShoppingCartItem
    -       fields = ('product', 'quantity')
- Then import the shopping cart items from the model.
- Then add cart item as a field to product serializer and it is a serializer method field. Also , add this new field to the meta fields.
    - cart_items = serializers.SerializerMethodField()
- Then define the method the that will return the shopping cart items for this cart item field.
    -    def get_cart_items(self, instance):
    - So, the items all belongs to this product.
    -       items = ShoppingCartItem.objects.filter(product=instance)
    - Then return the cart item serializer with those items serialized into a list.
    -       return CartItemSerializer(items, many=True).data
- The serializer method field will call by defaults the get_cartitems.
- For the other fields it would use the get_ as a prefix to the field name for the method that is called.
- Then talking about the serializer for one or many instances.
- The many=True creates a list of serializer model instances.
- Many= false (the default) will serialize only one model instance.
- Then we can check if everything is working fine by the following shell command run:



```
(DjangoEnv) D:\Sync\Projects\Django\DjangoREST\Practice Examples\Exercise 1>python manage.py shell
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:03:10) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> import json
>>> from store.models import *
>>> from store.serializers import *
>>> product = Product.objects.all().first()
>>> cart = ShoppingCart()
>>> cart.save()
>>> item = ShoppingCartItem(shopping_cart=cart, product=product, quantity=5)
>>> item.save()
>>> serializer = ProductSerializer(product)
>>> print(json.dumps(serializer.data, indent=2))
{
  "id": 1,
  "name": "Mineral Water Strawberry!",
  "description": "Natural-flavored strawberry with an anti-oxidant kick.",
  "price": 1.0,
  "sale_start": null,
  "sale_end": null,
  "is_on_sale": false,
  "current_price": 1.0,
  "cart_items": [
    {
      "product": 1,
      "quantity": 5
    },
```

Number field with serializers

- Add this field for quantity in the CartItemSerializer
  - quantity = serializers.IntegerField(min_value=1, max_value=50)
- Add serializer for price in the Product serializer:
  - price = serializers.FloatField(min_value=1.00, max_value=10000)
- Then there is the decimal field with the configuration



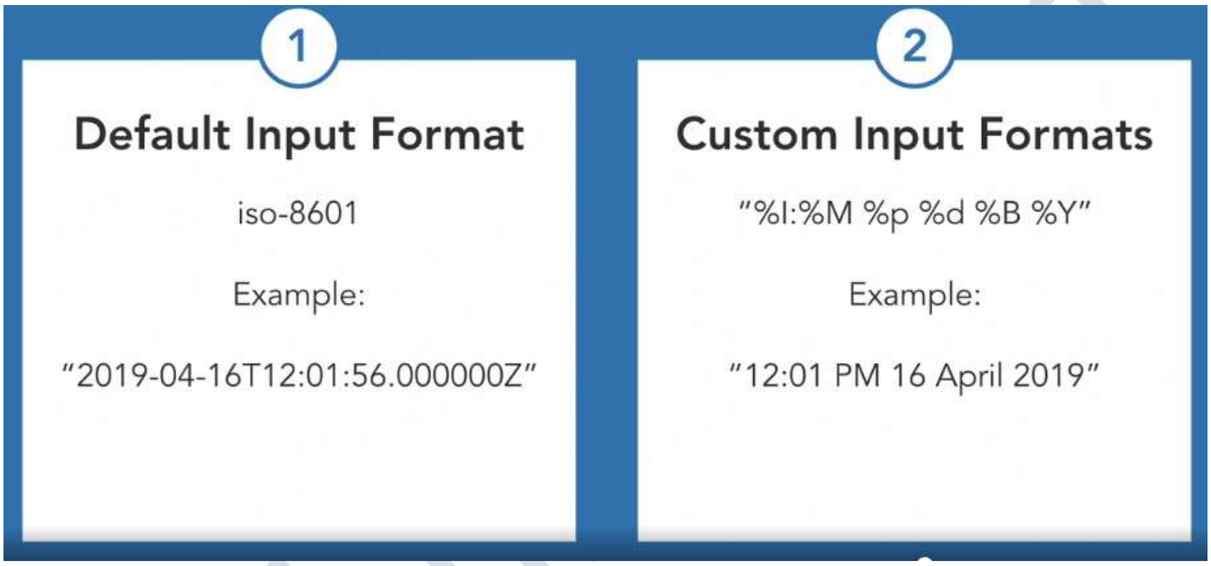
- **min_value** for minimum number value
- **max_value** for maximum number value
- **max_digits** for the resolution of the number
- **decimal_places** for the numbers after the decimal place

- So, then add the following code below the price float field:
  - price = serializers.DecimalField(
  - min_value=1.00, max_value=10000
  - ,max_digits= None, decimal_places=2)

Date and time fields with serializers

- Now its time for sales dates.
- Date time field configuration:
  - Input_formats for the date/time input formats (default is iso-8601)
  - Format is the output format (default is datetime object)
  - Help_text that appears in the browser for the REST API
  - Style controls how the field appears in the browser for the REST api (placeholder and input_type and input_type can be set with this)
- So, to set the over rider for the sale start and end date we write the code as following in the Product Serializer class.
  - sale_start = serializers.DateTimeField(

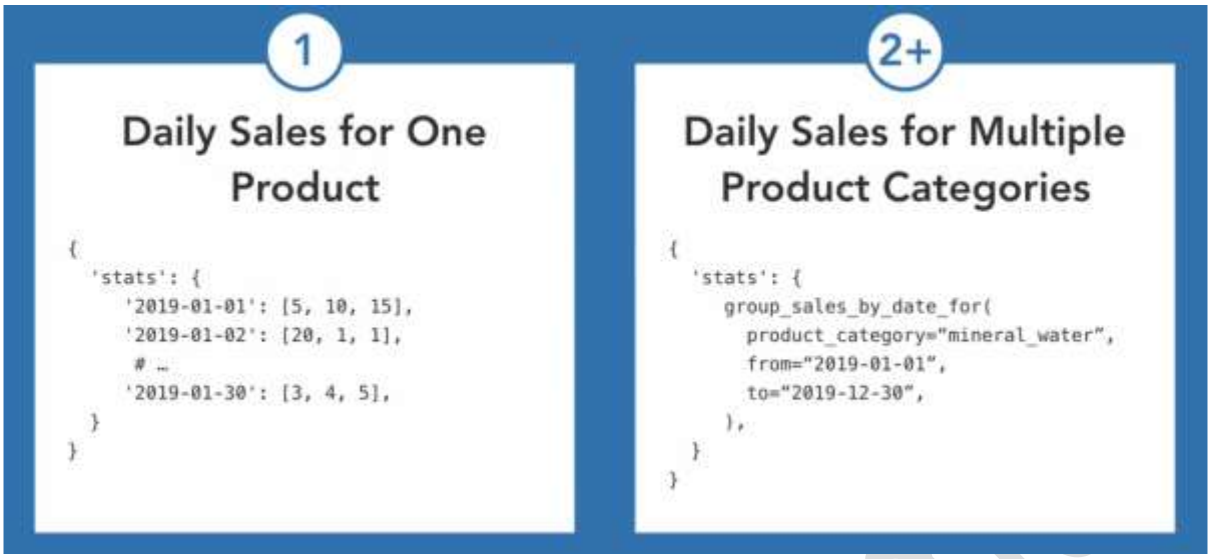
- input_formats=['%I:%M %p %d %B %Y'], format=None, allow_null=True, help_text='Accpted format is 12:01 AM 28 July 2020',
- style={'input_type': 'text', 'placeholder': '12:01 AM 28 July 2020'},
-)
- The format is set to none so that the sales start date and the sales end date are always date time objects.
- sale_end = serializers.DateTimeField(
- input_formats=['%I:%M %p %d %B %Y'], format=None, allow_null=True, help_text='Accpted format is 12:01 AM 28 July 2020',
- style={'input_type': 'text', 'placeholder': '12:01 AM 28 July 2020'},
-)
- The default is iso-8601, here custom input format was used.



List, dicts, and JSON objects

- In order to gather daily, monthly, or weekly data for sales report, it is required to create a new serializer that uses the composite fields.
- So, then a serializer for this is created in the following way:
  - Its just a plain serializer as it is not tied to any specific model.
  - class ProductStatSerializer(serializers.Serializer):
  - So, it only contains one field.
  - stats = serializers.DictField(
  - Dictionary field which has the child parameter set to a list field.
  - child = serializers.ListField(
  - Which has its own child parameter set to an integer field.
  - child = serializers.IntegerField(),
  -)
  -)
  - So, this is a composite of a composite field. The dictionary keys will be a date as a string. For each value in the dictionary it is expected a list of numbers which is the quantity from the shopping cart that the product appeared in.
- Next is to create a generic API view.
- Import the generic api view form the rest_framework.generics

  - GenericAPIView
- Also import the Response form rest_framework.response
- Then import the ProductStatSerializer that we just created form the store.serializers
- Now, to create the stat api view.
  - ProductStat that is extending the GenericAPIView.
  - class ProductStats(GenericAPIView):
  -    lookup_field = 'id'
  -    serializer_class = ProductStatSerializer
  -    queryset = Product.objects.all()
  - Override the get method.
  -    def get(self, request, format=None, id=None):
  - Get the object which is the product
  -       obj = self.get_object()
  - Load the serializer and provide it with the fake sales data.
  -       serializer = ProductStatSerializer({
  -          'stats': {
  - Date and list of numbers.
  -             '2019-01-01': [5, 10, 15],
  -             '2019-01-02': [20, 1, 1],
  -          }
  -       })
  - Return response with the serialized data.
  -       return Response(serializer.data)
- Then add the view into the url patterns.
- <int:id> is the specification of lookup id for to be filed while request.
  - path('api/v1/products/<int:id>/stats', store.api_views.ProductStats.as_view()),
- The data that comes back is structured as a dictionary as a list of numbers. It's also all stored under the stats field.
- With the data in this there is no one to one mapping between the sales data and the product model or the shopping cart item model.
- For example, we can group sales data from multiple product categories rather than for just on product like this.
- Compost field are highly useful when need to structure the data in a specific way that may not map to any model.



Daily Sales for One Product

```
{
  'stats': {
     '2019-01-01': [5, 10, 15],
     '2019-01-02': [20, 1, 1],
     # …
     '2019-01-30': [3, 4, 5],
  }
}
```

Daily Sales for Multiple Product Categories

```
{
  'stats': {
     group_sales_by_date_for(
       product_category="mineral_water",
       from="2019-01-01",
       to="2019-12-30",
     ),
  }
}
```

Serializer with ImageField and FileFields

- With the rest api it is possible to update images of product or upload new photos. So, now working on the photo field.
    - photo = serializers.ImageField(default=None)
    - As, the model does not have the product warranty field we are going to add the write only field to true.
    - warranty = serializers.FileField(write_only=True, default=None)
    - Here, a photo and a warranty field are set.
- So, the fields can be set as read only but not only that, they can also be set as write only.
- Serializer Field Configuration
    - Write_only = True means a field can be written to but will not appear in any API response.
- And also add them to the fields.
    - fields = ('id', 'name', 'description', 'price',
    - 'sale_start', 'sale_end', 'is_on_sale',
    - 'current_price', 'cart_items', 'photo', 'warranty')
- Then override the update method, so that warranty field can be used:
    - def update(self, instance, validated_data):
    - So, if a warranty field is supplied.
    - if validated_data.get('warranty', None):
    - Then add it to the description of the product.
    - instance.description +='\n\nWarranty Information;\n'
    - instance.description +=b'; '.join(
    - Read all the lines from the file.
    - validated_data['warranty'].readlines()
    -).decode()
    - And then return the instance.
    - return instance
- Validated_data: Data that has already passed through serializer and model validation process. It's used to create or update a model.
- It is safe to access because it has already passed through the validation process.
- So, in this example the price will be between $1 and $100000.

Testing API views

Test case for a CreateAPIView subclass

- Testing Django Rest framework api views is a bit different then testing in the Django views.
- Django rest framework provides test case classes:
    - APISimpleTestCase
    - APITransactionTestCase
    - APITestCase
    - APILiveServerTestCase
- ALL of the test case classes implement the same interface as Django Test case class.
- Remember to use the JSON format when testing API client requenst: self.client.post(url, data, format='json')
- So, now we will create the test case for product create view.
    - Instead of test case from Django, import the api test case form the rest_framework.test.
    - from rest_framework.test import APITestCase
    - Then import the product model.
    - from store.models import Product
    - Start to write the test.
    - class ProductCreateTestCase(APITestCase):
    -    def test_create_product(self):
    - Check the initial product count.
    -       initial_product_count = Product.objects.count()
    - Description of the new product.
    -       product_attrs = {
    -          'name' : 'New Product',
    -          'description' : 'Awesome product',
    -          'price' : '23.43',
    -       }
    - Checking the response from the client and posting to response end point
    -       response = self.client.post('/api/v1/products/new', product_attrs)
    - If we couldn't create the product.
    -       if response.status_code !=201:
    - Print out the response that is received.
    -          print(response.data)
    - Make sure that the product was created by checking the count of all the product.
    -       self.assertEqual(
    -          Product.objects.count(),
    -          initial_product_count + 1,
    -       )
    - Then check the values that are set for the product.
    -       for attr, expected_value in product_attrs.items():
    -          self.assertEqual(response.data[attr], expected_value)
    - Then check for the custom fields.
    -       self.assertEqual(response.data['is_on_sale'], False)
    -       self.assertEqual(
    -          response.data['current_price'],

  - float(product_attrs['price']),
  -)
- In the real world with the custom api's there can be a lot of additional custom fields. This is done so that clients of the api have access to more data. However, it is needed to thoroughly test this custom data that's added to api responses to ensure the api consumers don't fail or crash.
- REST APIs with custom data
  - Rest api can return custom data fields that are created through serializer fields or through overriding the to_representation method.
  - For Example, is_on_sale and current_price are update from method calls.
  - It's important to test custom data to ensure its correct; one untested field can cause API consumers and client to fail or crash.
- So, it fails as the sales start and sales end date are not specified.
  - Add required = false to both of them.
- So, then running again gives that the warranty is not part of product creation.
  - So, implement the create method and remove the warranty data and return the product as created with the warranty field.
  - def create(self, validated_data):
  - validated_data.pop('warranty')
  - return Product.objects.create(**validated_data)

Other test cases created are:

- Test case for a DestroyAPIView subclass
- Test case for a ListAPIView subclass
- Unit test for a UpdateAPIView subclass
- Handling image uploads in a unit test

And is done as following:

```
import os.path

from django.conf import settings


from rest_framework.test import APITestCase


from store.models import Product


class ProductCreateTestCase(APITestCase):

    def test_create_product(self):

        initial_product_count = Product.objects.count()

        product_attrs = {

            'name' : 'New Product',

            'description' : 'Awesome product',
            'price' : '23.43',
        }
        response = self.client.post('/api/v1/products/new', product_attrs)
        if response.status_code !=201:
            print(response.data)
        self.assertEqual(
            Product.objects.count(),
            initial_product_count + 1,
        )
        for attr, expected_value in product_attrs.items():
            self.assertEqual(response.data[attr], expected_value)
        self.assertEqual(response.data['is_on_sale'], False)
        self.assertEqual(
            response.data['current_price'],
            float(product_attrs['price']),
        )

class ProductDestroyTestCase(APITestCase):
    def test_delete_product(self):
        initial_product_count = Product.objects.count()
        product_id = Product.objects.first().id
        self.client.delete('/api/v1/products/{}/'.format(product_id))
        self.assertEqual(
            Product.objects.count(),
            initial_product_count - 1,
        )
        self.assertRaises(
            Product.DoesNotExist,
            Product.objects.get, id=product_id,
        )
class ProductListTestCaseJ(APITestCase):
    def test_list_product(self):
        product_count = Product.objects.count()
        response = self.client.get('/api/v1/products/')
        self.assertIsNone(response.data['next'])
        self.assertIsNone(response.data['previous'])
        self.assertEqual(response.data['count'], product_count)
        self.assertEqual(len(response.data['results']),product_count)

class ProductUpdateTestCase(APITestCase):
    def test_update_product(self):
        product = Product.objects.first()
        response = self.client.patch(
            '/api/v1/products/{}/'.format(product.id),
            {
            'name' : 'New Product',
            'description' : 'Awesome product',
            'price' : '23.43',
            },
            format='json',
        )
        updated = Product.objects.get(id=product.id)
        self.assertEquals(updated.name, 'New Product')

    def test_upload_product_photo(self):
        product = Product.objects.first()
        original_photo = product.photo
        photo_path = os.path.join(
            settings.MEDIA_ROOT, 'products', 'vitamin-iron.jpg',
        )
        with open(photo_path, 'rb') as photo_data:
            response = self.client.patch('/api/v1/products/{}/'.format(product.id), {
```

```
        'photo': photo_data,
    }, format='multipart')
self.assertEqual(response.status_code, 200)
self.assertNotEqual(response.data['photo'], original_photo)
try:
    updated = Product.objects.get(id=product.id)
    expected_photo = os.path.join(settings.MEDIA_ROOT, 'products', 'vitamin-iron')
    self.assertTrue(updated.photo.path.startswith(expected_photo))
finally:
    os.remove(updated.photo.path)
```