

**Introduction**:

In the first part of this course, we built data model and the admin interface for an e-commerce application. In this part we will continue our journey and build a RESTful API for this application.

By the end of this part of course, we will have complete production grade backend for an online store.



And much more…

**What are RESTful APIs**:

Before we created and implemented admin panel for managing our data. But this interface is only supposed to be used by admins.

So we need a way to expose our data to clients like web application running inside a browser or a mobile app running on mobile device.



This is where APIs are used.

API is short for Application programming interface.

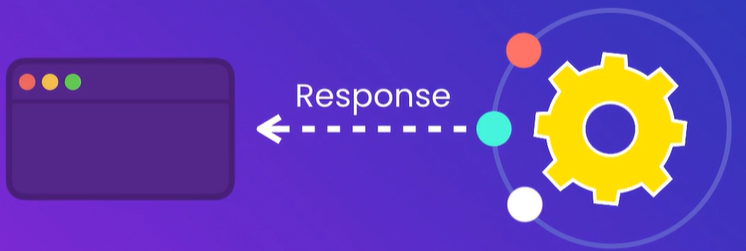
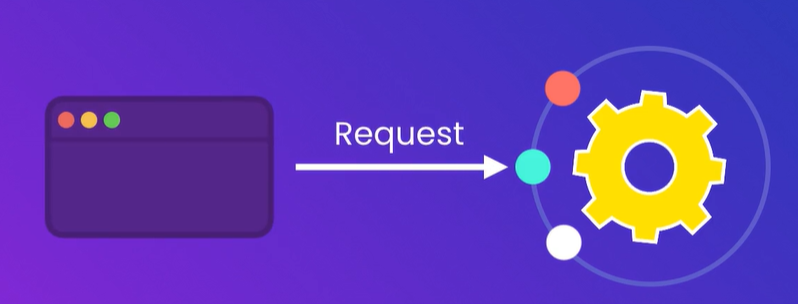
*“Building an API is essentially building an interface that client apps can use to get or save the data. It is like building a remote control with a bunch of buttons where each button provides certain functionality”*.

Similarly our API is going to have a bunch of endpoints for different purposes.

We can have an end point for getting the list of products as well as creating, updating and deleting them.



Other end points might include managing shopping cart, orders and so on.



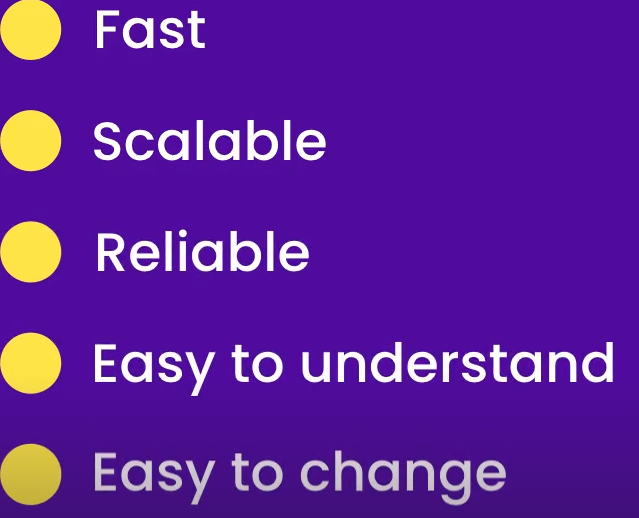
Client apps send request to these end points to get or save products, orders shopping cart and so on.

*What makes a API RESTful*:

REST is short for **Re**presentational **S**tate **T**ransfer.

*“REST defines a bunch of rules for clients and servers to communicate over the web”*.

Following these rules help us build systems that are



An API that confirms to these rules is called RESTful.

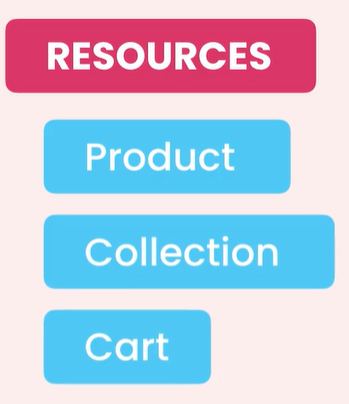
Practically speaking, there are *three key concepts* which everyone should know about RESTful APIs



**Resources**:

The first key concept you need to understand in RESTful APIs is the *concept of resources*.

A resource in RESTful API is like an object in our application like Product, Collection, and Shopping cart and so on.

🡨These resources are available on the web and client application can access them using a URL.

Remember, URL is short for Uniform *Resource* Locator. So it is a way to locate a resource on the web(*basically a web address*).

Assuming our website is hosted at <http://moshbuy.com>

We can access **all** products using [http://moshbuy.com/**products**](http://moshbuy.com/products)

and if you want to access an individual resource like an **individual** product, we *append its ID to the URL*,

[http://moshbuy.com/**products/1**](http://moshbuy.com/products/1)

A resource might contain other resources for example, a given product can have one or more reviews. Using below URL we can get all reviews for product #1

[http://moshbuy.com/**products/1/reviews**](http://moshbuy.com/products/1/reviews)

If need to get a particular review, we can reference it using its ID like this,

[http://moshbuy.com/**products/1/reviews**/1](http://moshbuy.com/products/1/reviews/1)

Note: As a rule of thumb, we should not nest our resources too deep. Two levels is fine but beyond that our URLs get complex and unmanageable.

Notice a pattern in all these URLs

[http://moshbuy.com/**products**](http://moshbuy.com/products)

[http://moshbuy.com/**products/1**](http://moshbuy.com/products/1)

[http://moshbuy.com/**products/1/reviews**](http://moshbuy.com/products/1/reviews)

[http://moshbuy.com/**products/1/reviews**/1](http://moshbuy.com/products/1/reviews/1)

This pattern is one of the attributes of RESTful APIs. So if we follow this pattern our API will be familiar and easy to understand to others.

**Resource Representations**:

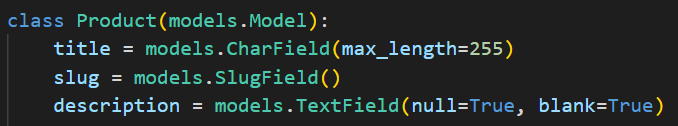
Now we know that we can identify a resource using its URL. When we hit that URL, the server is going to return a resource in certain format or representation.

It may return data in these forms,



But what’s Important here, is that *none of these are the internal representation of a resource on the server*.

In other words, on server we identify a resource like a product using an object or an instance of a python class like below,



But *when we return this object to the client, we are going to represent it as a HTML, XML or JSON because these are the formats that client understand*.

REST does not dictate what format or representation we should use. We may support one or multiple representations. If we support multiple, client should tell the server what representation it needs when asking for data.

*JSON*:

Stands for **J**ava**S**cript **O**bject **N**otation. It is a notation we use for representing objects in JavaScript.

🡨JSON represents an object using a pair of curly braces. Inside the braces we have a bunch of Key-value pairs or properties.

Note: In JSON Keys are always strings, so they should be surrounded with double quotes. Values on the other hand can be string, number, Boolean, object or array etc. Also throughout the course we will use this to send or receive data from the server.

**HTTP Methods**:

When building a RESTful API, we expose one or more end points for clients.

🡨Each end point supports various type of operations. Some end point may only allow reading data while others might allow modifying data as well.

This is where HTTP methods are helpful. Using these methods client can tell the server, what it wants to do with resource.



GET🡪 For getting a resource or a collection of resources.

POST🡪 For creating a resource

PUT🡪 For updating a resource

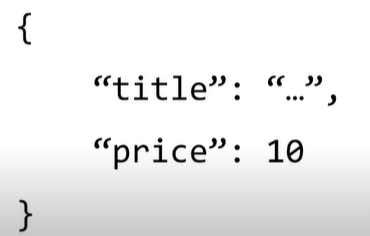
PATCH🡪 For updating a part of resource like a subset of properties.

DELETE🡪 For deleting a resource

Let us go through few scenario.

*Scenario 1*:

Assume we want to **create a product**. In our client app we should send a POST request to */products* end point. So that server knows that we want to create a product. But where is the product we want to create. **POST /products**

🡨 We are going to insert the Product as a JSON object in the body of request.

*Scenario 2*:

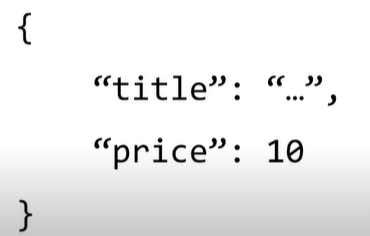
Assume we want to **update product #1**.

First we are going to ask *“Are we going to update all properties or just a subset of them*?”

If you want to update all properties we should send a PUT request otherwise we should send a PATCH request to */products/1* URL.

So either **PUT /products/1** or **PATCH /products/1**

Note: /products/1 is referencing a specific product not the products collection.

🡨 Similar to creating a product, we should include a product object in the body of request. So the server will extract this product from the request and update it accordingly.

*Scenario 3*:

To delete a product

Simply use, **DELETE /products/1**

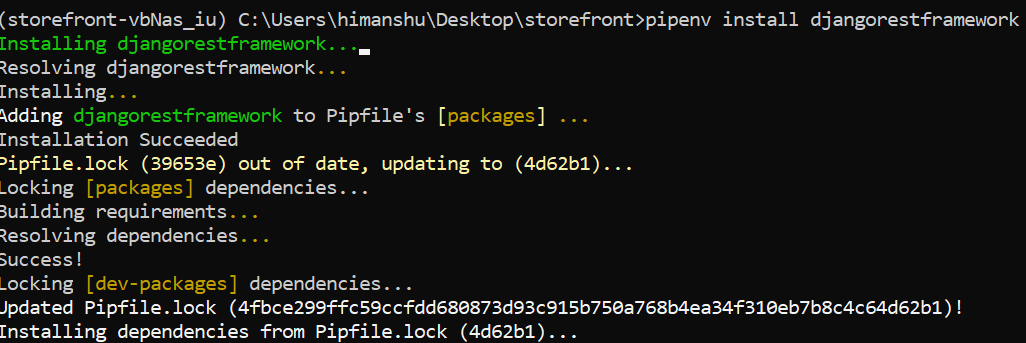
We do not have to add anything in the body of the request, because all the server needs to delete the product is its ID which is included in the URL anyway.

**Installing Django REST Framework**:

We will install Django REST Framework to build a RESTful API. This is a separate framework that sits on top of Django and makes it incredibly easier to build RESTful APIs.

In the terminal window write this command,

pipenv install djangorestframework



After installation is complete, Add it in the list of installed apps.

INSTALLED\_APPS = [

    "django.contrib.admin",

    "django.contrib.contenttypes",

    "django.contrib.sessions",

    "django.contrib.auth",

    "django.contrib.messages",

    "django.contrib.staticfiles",

    "rest\_framework", **🡨🡨🡨**

**Creating API Views**:

Let us see how we can create an endpoint like [http://127.0.0.1:8000/**store/product/**](http://127.0.0.1:8000/store/product/) **,**so if we send a request to this endpoint we should see all the products in our database.

**First step** is, Go to store app and open its views module. In this module we will create a view function.

Note: A view function is simply a function that takes a request and returns a response.

So we will create a function called *product\_list* because we want to see the list of all products and inside it we will return a response object as HttpResponse(). Inside the body of response we are simply passing “OK”.

from django.shortcuts import render

from django.http import HttpResponse

def product\_list(request):

    return HttpResponse("OK")

**Second step** is, Map this view function to a URL pattern. Currently in *store* app, we do not have urls module, so we will create one. Inside this module,

from django.urls import path

from . import views

urlpatterns = [path("products/", views.product\_list)]

If you have a request to the *products/* endpoint, that request will be handled by *product\_list* function.

This urls module belong to the store app, so **third step** is that we need to import it into the *main or root* URLs module (of *storefront* project).

urlpatterns = [

    path("admin/", admin.site.urls),

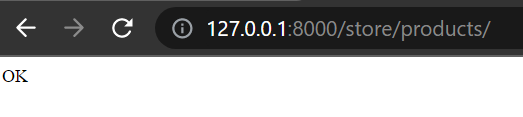
    path("playground/", include("playground.urls")),

    path("store/", include("store.urls")), 🡨🡨🡨Add Here…

    path("\_\_debug\_\_/", include("debug\_toolbar.urls")),

]

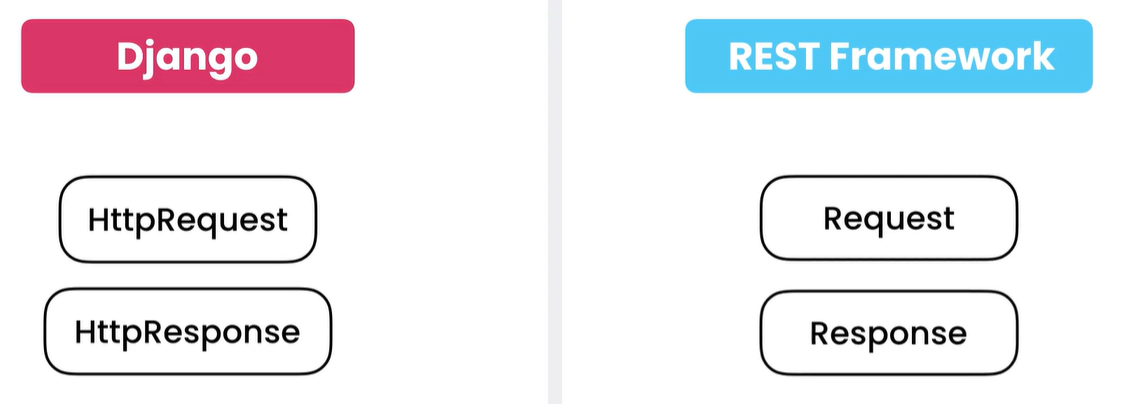
As per implementation up to this point:



Our endpoint is working fine.

*Now REST Framework comes in picture*:

In Django we have two classes HttpRequest and HttpResponse but Django REST framework also comes with its own Request and Response classes.



These classes are simple and more powerful than the ones that come with Django

🡪 Import api\_view decorator from rest\_framework

from rest\_framework.decorators import api\_view

If we apply this decorator to our view function like this,

@api\_view()

def product\_list(request):

    return HttpResponse("OK")

the request *object* we receive will instead be an instance of the request *class* that comes with rest framework (simpler and more powerful).

🡪 Now, we will replace old HttpResponse object we are returning from our view function with the newer one that comes with rest framework.

So import Response class from rest\_framework

from rest\_framework.response import Response

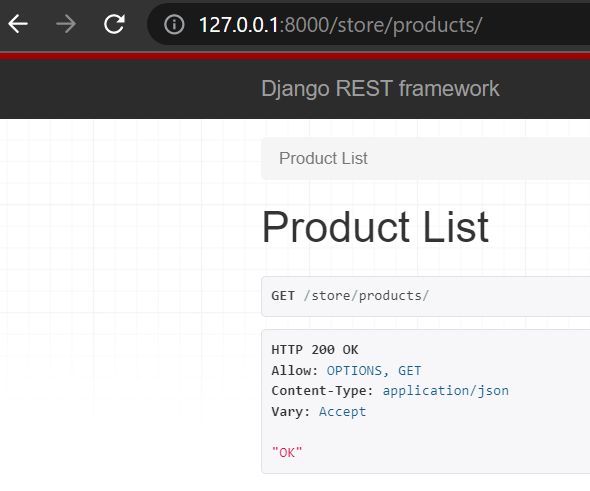
Simply replace HttpResponse with Response.

@api\_view()

def product\_list(request):

    return Response("OK")

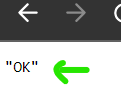
Just with these two simple changes, we see this on our page,



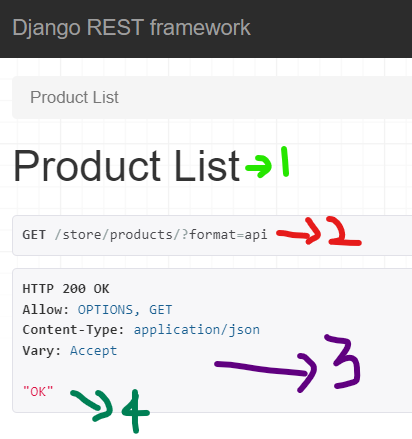
This beautiful page is called ***browsable API***. This browsable API makes it incredibly easy to test our API endpoints in the browser.

Note: We will see this page only if we hit correct end points in the browser. If a client app like a mobile app hits our endpoint, it is not going to see the browsable API instead it will only see data in response.

 If you click on JSON,

🡨Our client app will just see this,

We have a couple of things on browsable API page



Number wise , I will explain these components on browsable API page,

1. On the top we have *Heading* that is generated based on name of our view function (product\_list).
2. Next we have information about our request which is *GET* request on */store/products/* end point.
3. Information about the response.
   1. *Status* of response, 200 OK
   2. *Allow* tells us what HTTP methods are supported at this end point.
   3. *Content-type* tells us type of content in our response(application/JSON).
   4. *Vary* is used for caching.
4. Last we have *body* of the response.

Let us make one more API view, which will be another view function to see the details of a product.

@api\_view()

def product\_detail(request, id):

    return Response(id)

Since we will use product’s id to get an individual product detail, so we use ‘id’ as a parameter in our view function.

Next we map this function to list of urlpatterns.

urlpatterns = [

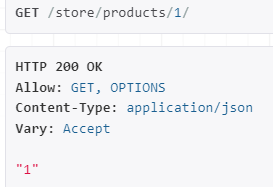
    path("products/", views.product\_list),

    path("products/<id>/", views.product\_detail),

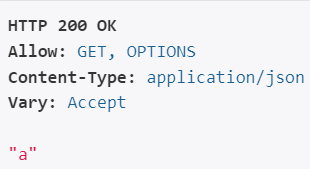
]

Notice ‘id’ is between <>, In this place we will insert our product-id

When we go to [http://127.0.0.1:8000/**store/products/1/**](http://127.0.0.1:8000/store/products/1/) , we see



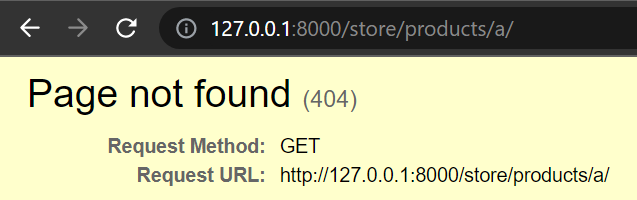
Note: If we give a non – numeric ID like “a” we still get a response

🡨Technically, this should not be allowed because our product-id is in integers.

So in our urls module, we can apply a converter to this parameter.

    path("products/<int:id>/", views.product\_detail),

Now if we hit this end point with “a” as product-id, we see an error, because our products end point only accepts integer values for the ‘id’ parameter. Pass 1 or 2 here and it will work.

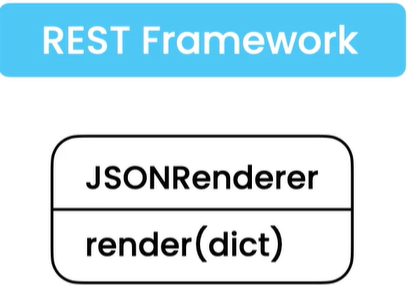


**Creating Serializers**:

We have created an API view but instead of giving a plain ‘OK’ message in response, we want to include the list of products.

So we need a way to convert product objects to JSON objects.

In Django rest framework, we have a class called JSON renderer, *this class has a method called* render *that takes a dictionary object and returns a JSON object*.

 So if we convert a product object to a python dictionary, we can pass it to this render method and get a JSON object.

This is where **Serializers** come into picture, *It is an object that knows how to convert a model instance(like a product object) to a python dictionary*.

Model 🡪*using serializer* 🡪 Dictionary 🡪*using render*🡪 JSON

Back in the *store* app, create a new file called serializers.py

On the top, from rest\_framework import serializers module

from rest\_framework import serializers

Next we create a class called *ProductSerializer* which will inherit the Serializer class defined in serializers module.

class ProductSerializer(serializers.Serializer):

Now ,we need to decide which fields of product class we want to serialize or *what fields we want to include in python dictionary*. so quickly jump back to Product class from models module.

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(1)])

    last\_update = models.DateTimeField(auto\_now=True)

    collection = models.ForeignKey(Collection, on\_delete=models.PROTECT)

    promotions = models.ManyToManyField(Promotion, blank=True)

    def \_\_str\_\_(self) -> str:

        return self.title

    class Meta:

        ordering = ["title"]

Look, our product class has so many fields, but *what we return from API does not necessarily need to have all these fields*. What we are seeing here is the **internal** representation of the product but what we return from API is the **external** representation of our product.

Note: Sometimes we have sensitive information here, which we do not want to expose to outside world. That is why we need two separate representations of a model.

Here we only want three things to return *product*-id, *title* and *price*.

Inside *ProductSerializer* class we define three fields exactly like how we define fields in a model.

class ProductSerializer(serializers.Serializer):

    id = serializers.IntegerField()

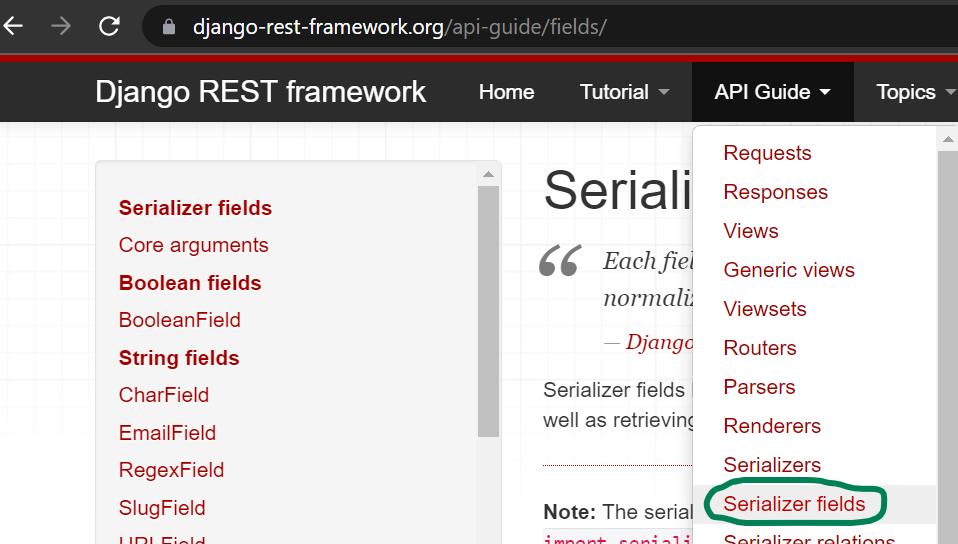
    title = serializers.CharField(max\_length=255)

    unit\_price = serializers.DecimalField(max\_digits=6, decimal\_places=2)

This is how you define a serializer.

Note: If you go to <https://www.django-rest-framework.org/> , under API Guide you will see Serializer fields which includes fields like BooleanField, CharField, EmailField and so on.

Pretty much everything that we need to create a model is also available here.



Now all these fields have some core arguments that are available everywhere like read\_only, write\_only, required, default etc. But certain fields depending on their type also support additional arguments.

**Serializing Objects**:

Now that we have a serializer we can use it to convert a product object to a JSON object and then include it in response.

First in our views module, we need to import a couple of classes.

from .models import Product

from .serializers import ProductSerializer

@api\_view()

def product\_detail(request, id):

    return Response(id)

Now we need to modify our product\_detail view function, we need to get product with its ‘id’ and then include it in response. (*Remember ORM methods that we learnt earlier to get an instance and querysets from database*)

@api\_view()

def product\_detail(request, id):

    product = Product.objects.get(pk=id)

    return Response(id)

Next we create a serializer and give it this *product* object.

@api\_view()

def product\_detail(request, id):

    product = Product.objects.get(pk=id)

    serializer = ProductSerializer(product)

    return Response(id)

Note: The moment we create the serializer, this serializer will convert our product object to a dictionary and we can get that dictionary from serializer.data. Instead of id in response we return the serializer.data.

@api\_view()

def product\_detail(request, id):

    product = Product.objects.get(pk=id)

    serializer = ProductSerializer(product)

    return Response(serializer.data)

When we go to <http://127.0.0.1:8000/store/products/1/>

🡨We see our product with id=1

Under the Hood: The magic of JSON renderer happens under the hood. At some point Django creates a JSON render object and give it this dictionary. The JSON renderer will then convert that dictionary to JSON object. Finally that JSON object will end up in our response. That process is hidden from us. **Learn more about it**…

Model 🡪*using serializer* 🡪 Dictionary 🡪*using render*🡪 JSON

*Overriding Django Rest Framework default settings*:

In the response we see that *unit price* is rendered as a string , even though it is a decimal field.

It is because, one of the default settings in Django REST framework. But we can easily overwrite it.

Go to Settings module and define a new setting called

REST\_FRAMEWORK and set it to a new object.

REST\_FRAMEWORK = {

}

In this object add a key exactly like below and set it to False.

REST\_FRAMEWORK = {

    'COERCE\_DECIMAL\_TO\_STRING':False

}

, Now our unit price is rendered as decimal value.

*Handling exception if product does not exist*:

If we search for product id *zero*, which does not exist we see an exception on page.



This is not good because one of the conventions of RESTful API is that even if object does not exist, we should return a response and the status of the response should be 404 in this case.

*First way to handle exception*:

Wrap the code in try block and catch the exception using DoesNotExist method inherited in Product model.

@api\_view()

def product\_detail(request, id):

    try:

        product = Product.objects.get(pk=id)

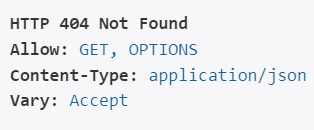
        serializer = ProductSerializer(product)

        return Response(serializer.data)

    except Product.DoesNotExist:

        return Response(status=404)

If we refresh the page now, instead of exception, we get a response with status 404 Not Found.



*Using* status *module of Rest Framework*:

In status module we have a bunch of constants for various HTTP status codes.

🡪 Import status module

from rest\_framework import status

🡪 Replace 404 in status as below

@api\_view()

def product\_detail(request, id):

    try:

        product = Product.objects.get(pk=id)

        serializer = ProductSerializer(product)

        return Response(serializer.data)

    except Product.DoesNotExist:

        return Response(status=status.HTTP\_404\_NOT\_FOUND)

Same result and our code is already more readable.

*Using Django shortcuts(best approach)*:

With this shortcut we do not need to wrap our logic inside

try: except block, Django shortcut function will take care of it.

🡪 Import get\_object\_or\_404 function from django.shortcuts

from django.shortcuts import get\_object\_or\_404

🡪 remove try except block and instead of using manager use the shortcut function

@api\_view()

def product\_detail(request, id):

        product = get\_object\_or\_404()

        serializer = ProductSerializer(product)

        return Response(serializer.data)

🡪 First argument of this function is type of model which is *Product* and second is lookup parameter(*pk=id*).

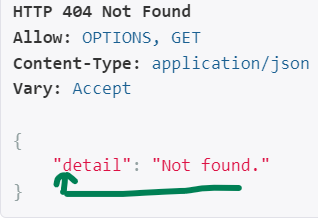
@api\_view()

def product\_detail(request, id):

        product = get\_object\_or\_404(Product, pk=id)

        serializer = ProductSerializer(product)

        return Response(serializer.data)

 This time along with HTTP status 404, we also get body of the response. This is the best approach so far.

We are done with this view function, now let us modify the other view function product\_list.

In this view function, we need to get all products

@api\_view()

def product\_list(request):

    return Response("OK")

🡪 Here we can use all method which returns a queryset.

@api\_view()

def product\_list(request):

    queryset = Product.objects.all()

🡪 Same like before, we create a serializer and give it queryset

@api\_view()

def product\_list(request):

    queryset = Product.objects.all()

    serializer = ProductSerializer(queryset)

🡪Now return serializer.data as Response.

@api\_view()

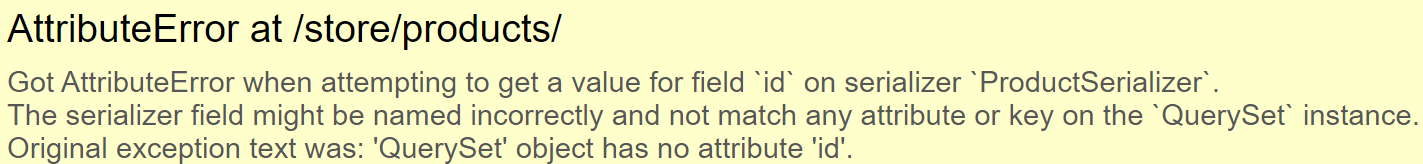
def product\_list(request):

    queryset = Product.objects.all()

    serializer = ProductSerializer(queryset)

    return Response(serializer.data)

We get this error…



To solve this error

@api\_view()

def product\_list(request):

    queryset = Product.objects.all()

    serializer = ProductSerializer(queryset, many=True)

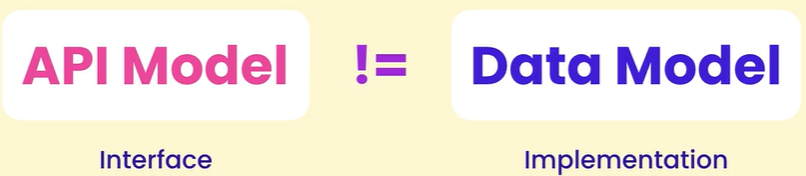
    return Response(serializer.data)

We set many=True , *so the serializer knows that it should iterate over this queryset and convert each product object into a dictionary*.

🡨 Now we see all products here.

**Creating custom serializer fields**:

As we know that objects that we return from API , do not have to necessarily look like the objects in our application.



Main reason behind is that our *data models are really implementation details* of our application and that implementation may change in the future.

We can add new fields in the Product class or rename existing fields or even delete them. We do not want these details to be exposed to outside world.

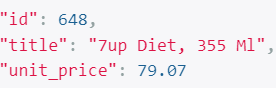
It is like *“The buttons on the remote control do not change although the technology/chip inside remote might change and these changes are hidden from consumers”.*

On the other hand, our API represents the interface (*buttons of remote*) of our application. So we should try to keep it as stable as possible, otherwise existing clients may break.

If you want to change our API we have to properly study the impact of the change and potentially provide different versions of our API(*a topic for the future*).

Now Back to our products endpoint,

As you can see we are missing a few fields of the Product class,



Let us say this was a deliberate decision of not including those fields here. Similarly *we can decide to add new fields here that do not exist in the Product class*.

In our ProductSerializer,

class ProductSerializer(serializers.Serializer):

    id = serializers.IntegerField()

    title = serializers.CharField(max\_length=255)

    unit\_price = serializers.DecimalField(max\_digits=6, decimal\_places=2)

🡪 We will add a new field called price\_with\_tax and set it to serializer.**SerializerMethodField**,

class ProductSerializer(serializers.Serializer):

    id = serializers.IntegerField()

    title = serializers.CharField(max\_length=255)

    unit\_price = serializers.DecimalField(max\_digits=6, decimal\_places=2)

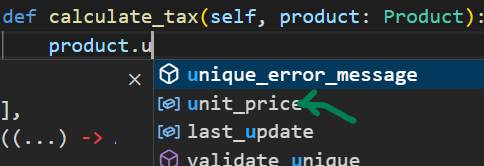
    price\_with\_tax=serializers.SerializerMethodField()

🡪 That means we are going to define a method here and that method is going to return the value for this field.

def calculate\_tax(self, product: Product):

Notice that we give it 2 parameters, self and product (model being serialized)

Note: We use type annotation as ‘*product****: Product****’*, so that we get auto suggestions from intellisense.



🡪 We return new value from this function

    def calculate\_tax(self, product: Product):

        return product.unit\_price \* Decimal(1.1)

Note: 1.1 is wrapped inside Decimal method because we are multiplying it with unit price which is a decimal field. We get an error if do not do that.

Remember to import Decimal

from decimal import Decimal

🡪 Reference this new method inside SerializerMethodField,

price\_with\_tax = serializers.SerializerMethodField(method\_name="calculate\_tax")

*Rename API model Fields*:

If we try to rename our unit\_price field to price, simply by changing in ProductSerializer like below,

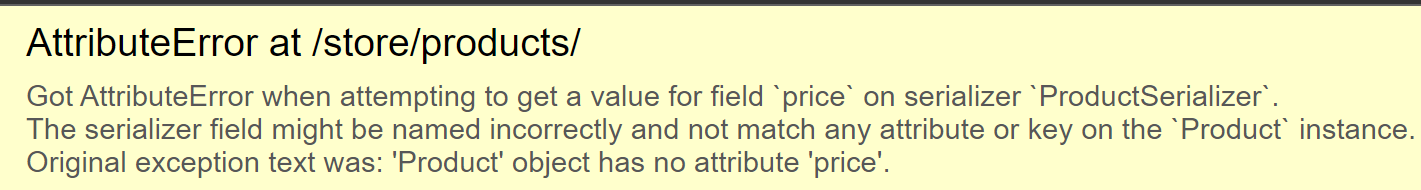
class ProductSerializer(serializers.Serializer):

    id = serializers.IntegerField()

    title = serializers.CharField(max\_length=255)

    price = serializers.DecimalField(max\_digits=6, decimal\_places=2)

We get an exception.



‘Product’ object has no attribute ‘price’

The reason we are saying this is because, by default Django assumes that we have a field by this name in the Product class.

So it will try to read the value of price field from there.

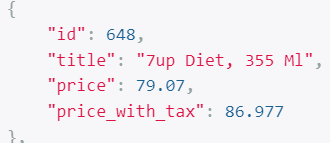
*We just need to tell Django, where to look for this field’s* ***source***.

All our API model fields has an argument called ‘source’.

    price = serializers.DecimalField(

        max\_digits=6, decimal\_places=2, source="unit\_price")

We set the source to unit\_price and now we have no error and a renamed field.



As we can see, our internal and external representation of this Product model is starting to evolve differently.

**Serializing Relationships**:

When returning a product, we can include a related object like a *collection*. There are few ways we can do this.

First import Collection class from models module.

from store.models import Product, Collection

Define a new field in ProductSerializer class called *collection* and set it to serializer.PrimaryKeyRelatedField.

class ProductSerializer(serializers.Serializer):

    id = serializers.IntegerField()

    collection = serializers.PrimaryKeyRelatedField()

With this we can include primary key or the ID of each collection in a product object.

But first we need to set an argument called *queryset*, which we should set to queryset for looking at collections.

collection = serializers.PrimaryKeyRelatedField(

        queryset= Collection.objects.all()

    )

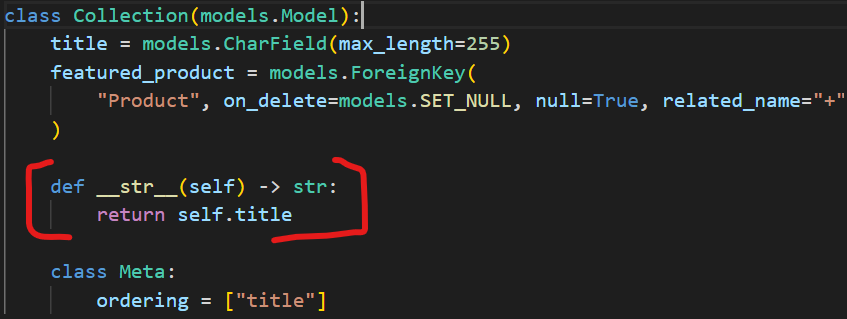
🡨 Now in each product we can see ID of the collection. This is one way to serialize a relationship.

*Returning Collection as a string*:

We can also return name of each collection. In that case we can use StringRelatedField.

With this method, Django will convert each collection into a String object and return it here.

Remember: We override the string representation of a class by using \_\_str\_\_ method. In our case, we are returning title.



Back to the code, we use StringRelatedField method,

class ProductSerializer(serializers.Serializer):

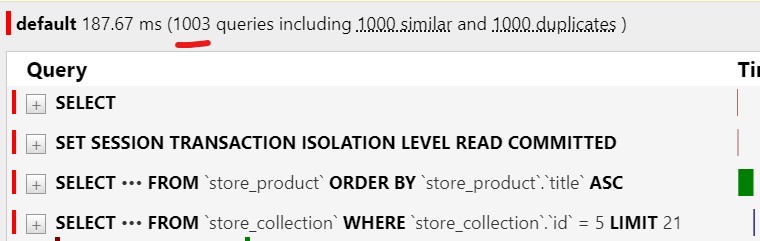
    id = serializers.IntegerField()

    collection = serializers.StringRelatedField()

and get Collection names instead of their ID



But there is one big **problem** in this implementation, look at the number of queries,

🡨We are getting 1000 extra queries and this is because of lazy loading and for each Product, we have an extra query to read this collection.

To solve this problem, we need to load products and collection together. *We will do this related field pre-loading inside views module*.

@api\_view()

def product\_list(request):

    queryset = Product.objects.all()

    serializer = ProductSerializer(queryset, many=True)

    return Response(serializer.data)

This is the view to get the list of products where we can see a queryset for retrieving all products.

So before calling all method, we will call select\_related and pass collection.

@api\_view()

def product\_list(request):

    queryset = Product.objects.select\_related('collection').all()

    serializer = ProductSerializer(queryset, many=True)

    return Response(serializer.data)

The problem is solved now.

*Serialize a relationship using nested object*:

Another approach to serialize a relationship is by including a nested object. That means we can include a ‘collection’ object inside our ‘product’ object.

Let us start first by creating a *CollectionSerializer* in our serializers module.

class CollectionSerializer(serializers.Serializer):

In this class we will define two fields id and title.

class CollectionSerializer(serializers.Serializer):

    id = serializers.IntegerField()

    title = serializers.CharField(max\_length=255)

To use this serializer as a nested object, we simply have to use an instance of this CollectionSerializer class as collection field inside our ProductSerializer like this,

class ProductSerializer(serializers.Serializer):

    id = serializers.IntegerField()

    collection = CollectionSerializer()

🡨 Now each collection is rendered as an object.

*Serialize a relationship as a hyperlink to an endpoint*:

There is one more way to serialize a relationship. Instead of including a nested object, we can include an hyperlink to an endpoint for viewing that collection.

Now this one has a couple of steps. So pay close attention.

**Step1**:

Set the collection fields inside our Serializer as serializers.HyperlinkedRelatedFields.

class ProductSerializer(serializers.Serializer):

    id = serializers.IntegerField()

    collection = serializers.HyperlinkedRelatedField()

**Step 2**:

Here we set two arguments. First one is queryset which we set to Collection.objects.all() and second one is view\_name and set it to a string called ‘*collection-detail’*.

class ProductSerializer(serializers.Serializer):

    id = serializers.IntegerField()

    collection = serializers.HyperlinkedRelatedField(

        queryset = Collection.objects.all(),

        view\_name='collection-detail'

    )

Note: view\_name argument is used for generating hyperlinks.

**Step 3**:

Currently we do not have a URL by ‘*collection-detail’* name, so we need to create it in the urls module.

Here we define a new route and say if you have a request to collections/<int:id>/, we want this request to be handled by a view function called collection\_detail(will create in step 4).

    path("collections/<int:id>/", views.collection\_detail),

]

The path function has one more attribute called name which we set to view\_name defined earler.(‘*collection-detail’*).

urlpatterns = [

    path("products/", views.product\_list),

    path("products/<int:id>/", views.product\_detail),

    path("collections/<int:id>/", views.collection\_detail, name="collection-detail"),

Next, we will define our *collection\_detail* view function in views module.

**Step 4**:

Create a function called *collection\_detail* in views module, For now we will simply return a “OK” response.

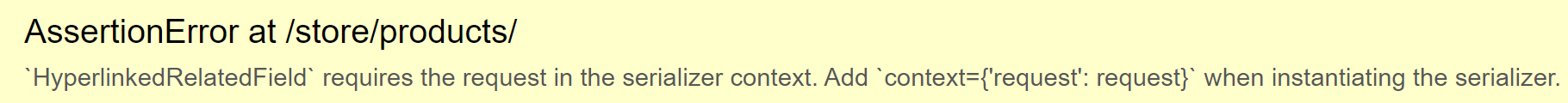
@api\_view()

def collection\_detail(request, id):

    Response("OK")

Let us test our implementation up to this point,

We are getting an exception



This means we need to pass our request object to serializer because the request contains information about URLs. It is giving us hint about adding `context={'request': request}`while instantiating the serializer.

In our views module,

@api\_view()

def product\_list(request):

    queryset = Product.objects.select\_related("collection").all()

    serializer = ProductSerializer(queryset, many=True)

    return Response(serializer.data)

This product\_list view function is retrieving all Products and the line *where ProductSerializer is instantaited, we can use context object and give it some extra stuff*.

@api\_view()

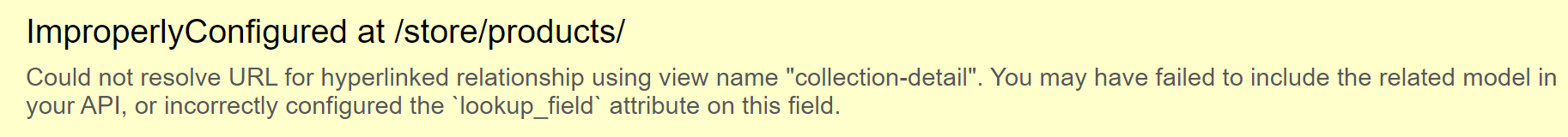
def product\_list(request):

    queryset = Product.objects.select\_related("collection").all()

    serializer = ProductSerializer(queryset, many=True, context={"request": request})

    return Response(serializer.data)

After configuring context, Now, we get a different type of exception



The reason we are seeing this error because Django rest framework expects a certain convention in our urlpatterns path In our collections URL our parameter should be called ‘pk’ instead of ‘id’.

urlpatterns = [

    path("collections/<int:pk>/", views.collection\_detail, name="collection-detail"),

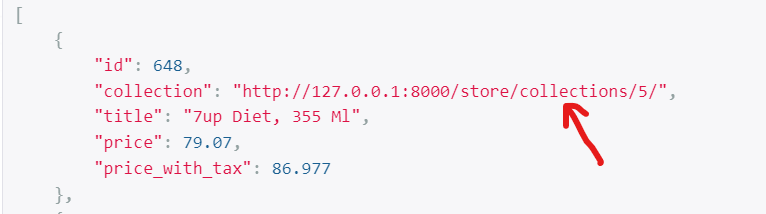
]

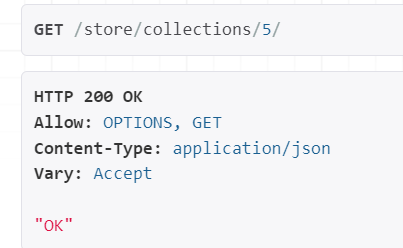
Change in collection\_detail view function as well,

@api\_view()

def collection\_detail(request, pk):

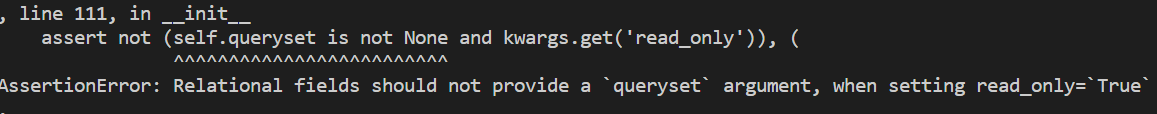
    Response("OK")

🡨Now we get a hyperlink for each collection. If you click on this link,

🡨We go to endpoint for viewing details of that collection.

**Exercise**: When we click on collection hyperlink, we should be able to see details of that collection instead of plain ‘OK’ response.

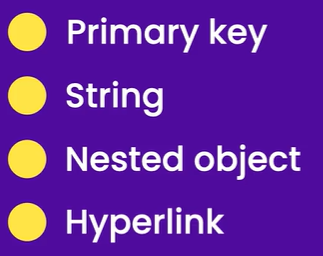
Note: Sometimes Django does not detect changes in Serializer relational fields and we get an error despite of providing ‘queryset’ or other arguments.



In this case , just restart your server and it will be fine.

**Summary**:

There are four ways to serialize a relationship,



**Model Serializers**:

So far we can see that there are two places we are defining almost same fields and their validation rules, *In models module and in serializers module*.

So if tomorrow we decide to change the validation rules for let us say, the title of our products, then there are two places we need to change that rule. There must be a better way, right?

This is where we use ***model serializers***, Using the model serializer class we can quickly create a serializer without all the duplication.

So in ProductSerializer let us change the base class to ModelSerializer

class ProductSerializer(serializers.ModelSerializer):

🡪We will create a *Meta* class inside, where we will set two attributes model and fields.

class ProductSerializer(serializers.ModelSerializer):

    class Meta:

        model = Product

        fields = ["id", "title", "unit\_price", "collection"]

We do not need to redefine all the fields like before, *we can have Django rest framework look up the definition of these fields in the Product model class and automatically create a Product Serializer for us*.

🡨We get this representation. For collection which is a related field, we get PrimarykeyRelatedField(because it is default setting for related fields in ModelSerializers)

Note: We can override default setting for related fields in ModelSerializers like below,

class ProductSerializer(serializers.ModelSerializer):

    class Meta:

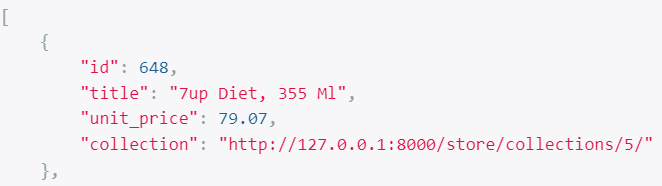
        model = Product

        fields = ["id", "title", "unit\_price", "collection"]

    collection = serializers.HyperlinkedRelatedField(

        queryset=Collection.objects.all(), view\_name="collection-detail"

    )



Now we get hyperlink instead of primary key of collection. It is better to stick with the default.

Note: Similarly we can use price field that we created instead of unit\_price.

class ProductSerializer(serializers.ModelSerializer):

    class Meta:

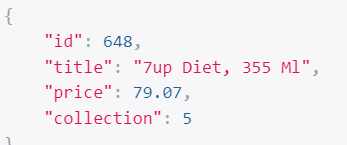
        model = Product

        fields = ["id", "title", "price", "collection"]

    price = serializers.DecimalField(

        max\_digits=6, decimal\_places=2, source="unit\_price"

    )



Still better to stick with the default names ☺.

🡪 Our custom created field inside Product Serializer, price\_with\_tax is the only extra field we want, so we can include it like this,

class ProductSerializer(serializers.ModelSerializer):

    class Meta:

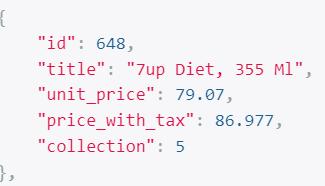
        model = Product

        fields = ["id", "title", "unit\_price", "price\_with\_tax", "collection"]

    price\_with\_tax = serializers.SerializerMethodField(method\_name="calculate\_tax")

    def calculate\_tax(self, product: Product):

        return product.unit\_price \* Decimal(1.1)



Note: Some developers use this bad practice, which we should avoid at all costs.

class ProductSerializer(serializers.ModelSerializer):

    class Meta:

        model = Product

        fields = "\_\_all\_\_"

they set fields to ‘\_\_all\_\_’ which means all fields in the product class. We should avoid setting our fields like this because *we need to separate internal and external representation of our Data*.

**Summary of serializers**:

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

**Deserializing Objects**:

Let us talk about Deserializing objects. It is completely opposite of serialization and it happens when we receive data from the client.

For example, a client wants to create a new product. To do this *it should send a* ***POST request*** *to the products endpoint* and in the body of the request a product object must be included.

 Now *on the server, we have to* ***read the data*** *in the body of the request and* ***deserialize*** *it so we get a Product object and* ***store*** *it in the database*.

Now Let us do the same activity,

In our product\_list view function, we pass an array of strings as an argument to the @api\_view() decorator.

These strings specify the HTTP methods which support our *products/* endpoint.

@api\_view(['GET', 'POST'])

def product\_list(request):

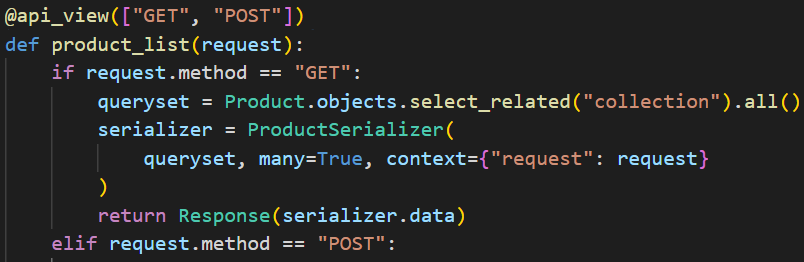
    queryset = Product.objects.select\_related("collection").all()

    serializer = ProductSerializer(queryset, many=True, context={"request": request})

    return Response(serializer.data)

Note: Previously we did not have to do this because ‘GET’ is supported by default. Since now we also need to support ‘POST’, we have to explicitly pass this array here.

🡪 Now we have to specify **if- elif** conditions inside view function to return different Response based on type of request.



🡪 In POST request we will read data in the body of request and deserialize it.

    elif request.method == 'POST':

        serializer = ProductSerializer(data=request.data)

Here in ProductSerializer we set keyword argument data to request.data. This request.data will be deserialized.

Note: After deserialization of this data, it will be available in an attribute called serializer.validated\_data. But before we can access this attribute, we have to first validate the data(*topic of next lesson…*). For now we simply return ‘OK’ Response.

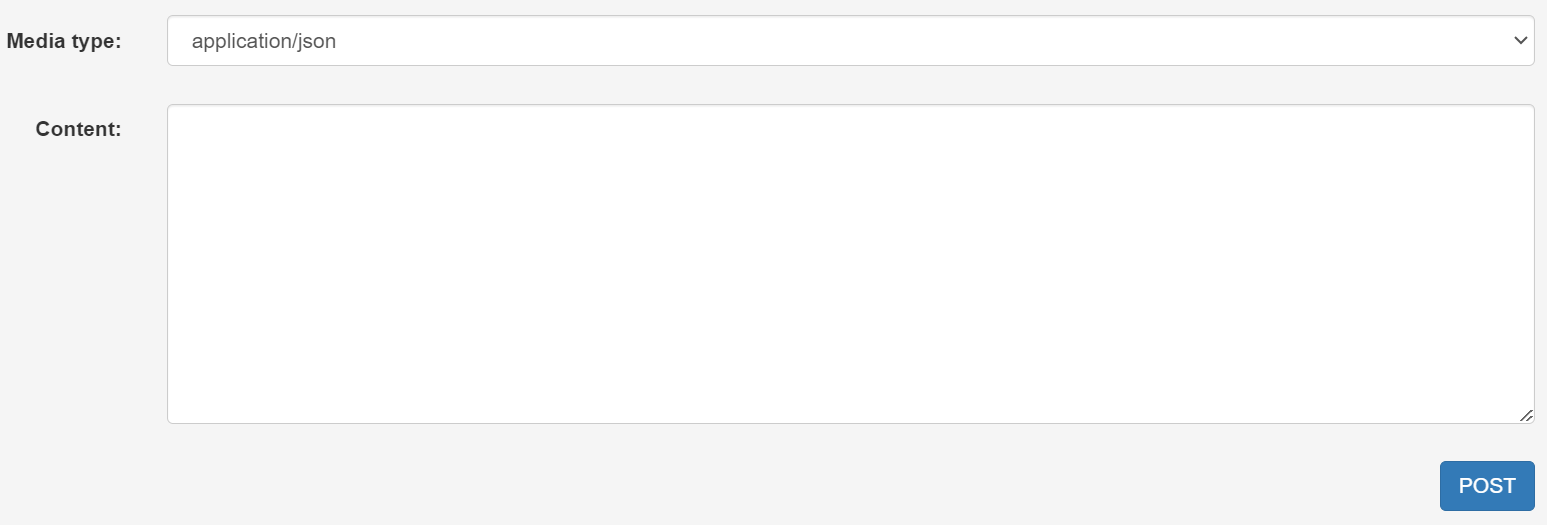
    elif request.method == "POST":

        serializer = ProductSerializer(data=request.data)

        # serializer.validated\_data

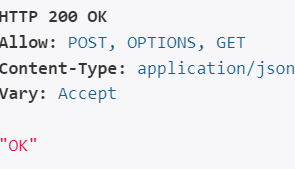
        return Response("OK")

🡪 In the browser if you look at bottom, we see a box for sending content to the server. This is one of the benefits of browsable API.



It is because we are now supporting ‘POST’ method at this endpoint.

🡪 just pass an empty object {} in the content box and click on POST.

🡨We get an ‘OK’ response.

**Data Validations**:

As we know before accessing serializer.validated\_data attribute, we first have to validate the data, otherwise we are going to get an exception. Let us see this in action by uncommenting serializer.validated\_data line.

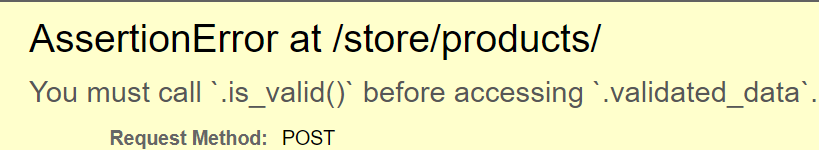
    elif request.method == "POST":

        serializer = ProductSerializer(data=request.data)

        serializer.validated\_data

        return Response("OK")

Now after we POST an empty object again, we get an exception.



So we call is\_valid() serializer method like this,

    elif request.method == "POST":

        serializer = ProductSerializer(data=request.data)

        if serializer.is\_valid():

            serializer.validated\_data

            return Response("OK")

        else:

            return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

if serializer.is\_valid is true we return an OK response otherwise we return a response with serializer.errors and status 400.

🡨 We get these array of validation errors for each field.

There is another way to write this code which is cleaner and more concise.

*Using raise\_exception = True(better approach)*:

With this way, we do not have to use If – else statements to handle validations. So if we get invalid data, Django rest framework is automatically going to return a Response with status 400.

    elif request.method == "POST":

        serializer = ProductSerializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        serializer.validated\_data

        return Response("OK")

Let us print validated\_data on the terminal and see what we get,

    elif request.method == "POST":

        serializer = ProductSerializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        print(serializer.validated\_data)

        return Response("OK")

let us POST a valid object this time,

{

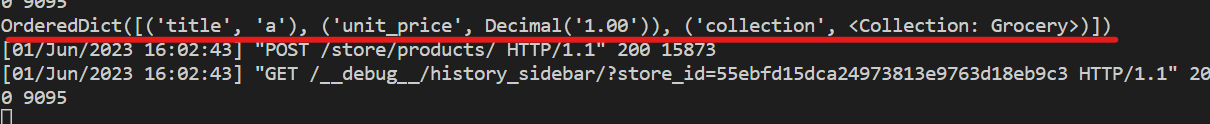
*"title": "a",*

*"unit\_price": 1,*

*"collection": 2*

}

We get a response “OK” and in the terminal we see,



This Ordered Dictionary is our validated\_data attribute and it has three key-value pairs. Title holds a string, unit\_price holds a decimal value and collection holds a collection object.

As part of Deserializing the request data, Django rest framework automatically retrieve a collection with ID that we specified(*in next lesson we will save this data in database*).

***Validation at the object level***:

There are situations where validating the request data involves comparing multiple fields.

Consider a scenario where a user registers, so we have username field as well as password and confirm password fields.

We want to make sure that these two fields are equal. With our current implementation we cannot achieve this because *our validation rules come from definition of model fields*.

So if we need anything extra, we need to override the validate method in our serializer class. This is actually defined in base class (ModelSerializer) but we are overriding it now.

It takes two parameters, self and data(*a dictionary basically*)

def validate(self, data):

Next we can define some conditions and return appropriate responses,

    def validate(self, data):

        if data['password'] != data['confirm\_password']:

            return serializers.ValidationError('Passwords do not match')

        else:

            return data

*Validate method should either return validation error or actual data which is a dictionary*.

Now we know how to validate data, next step is to save it in Database.

**Saving Objects**:

Let us talk about saving data, So our ProductSerialzer is inheriting the ModelSerializer class.

This ModelSerializer class has a **save** method that we can use for creating or updating a product.

So back to our view function under request.method == ‘POST’,

right after where we validate our serializer data, we call serializer.save()

    elif request.method == "POST":

        serializer = ProductSerializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        serializer.save()

        print(serializer.validated\_data)

        return Response("OK")

Technically we do not need to touch validated\_data attribute because the save method has some logic for extracting data from this dictionary to create or update the product. So we can delete the line and test our implementation up to this point.

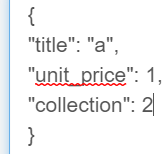
    elif request.method == "POST":

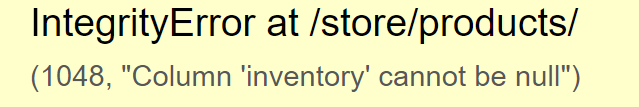
        serializer = ProductSerializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        serializer.save()

        return Response("OK")

🡨 we post our data and get an exception,

🡨 It comes because inventory is one of the required field in product class and we have not supplied this field in our request body.

In our Product model class,

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(1)])

    last\_update = models.DateTimeField(auto\_now=True)

    collection = models.ForeignKey(Collection, on\_delete=models.PROTECT)

    promotions = models.ManyToManyField(Promotion, blank=True)

title, slug, unit\_price, inventory, collection are required fields.

Now back in ProductSerializer, add the required fields,

(Put description here as well because when people see product they must see description as well although it is not required)

class ProductSerializer(serializers.ModelSerializer):

    class Meta:

        model = Product

        fields = [

            "id",

            "title",

            "description",

            "slug",

            "inventory",

            "unit\_price",

            "price\_with\_tax",

            "collection",

        ]

{

"title": "a",

"slug": "a",

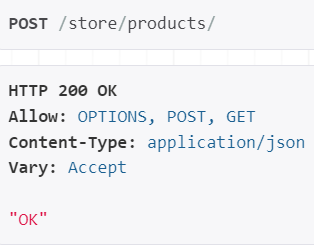
"unit\_price": 1,

"collection": 2,

"inventory": 1

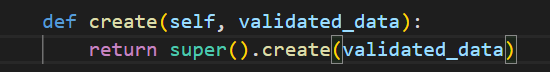
}

Let us POST this new data,

🡨 and we get a “OK” response this time and this product is created.

Sometimes we want to override how a product is created, *perhaps we want to set some special fields or associate a product with another object in the database*.

So back to our ProductSerializer class, Here we can override the create method,



This method takes the validated\_data dictionary as one of the argument

Inside the body of this create method, Create a product object and *unpack* validated\_data dictionary.

product = Product(\*\*validated\_data)

Then we can set those special fields, whatever they are

product.other\_fields = 1

save the product object

product.save()

Finally return product from this method,

    def create(self, validated\_data):

        product = Product(\*\*validated\_data)

        product.other\_fields = 1

        product.save()

        return product

so create is one of the method that exists in base ModelSerializer class, it is called by the same method, If we try to create a new class.

*Updating a Product by overriding* ***update*** *method*:

If we want to override how a product is updated then we can override the update method.



We have two arguments, ‘instance’ which is the product object in our case and validated\_data.

In the body of update method,

instance.unit\_price = validated\_data.get('unit\_price')

save the instance,

instance.save()

Return the instance,

    def update(self, instance, validated\_data):

        instance.unit\_price = validated\_data.get('unit\_price')

        instance.save()

        return instance

In this case , we really do not need to do this. We can rely on Django REST framework to automatically set all these fields for us.

Note: **save** method will call either create or update method depending on the state of the serializer.

*Updating a product using* PUT *or* PATCH *request*:

To update the product, we should modify this view function,

@api\_view()

def product\_detail(request, id):

    product = get\_object\_or\_404(Product, pk=id)

    serializer = ProductSerializer(product)

    return Response(serializer.data)

We should be able to send a put or patch request to a particular product, like a product with id = 1,



@api\_view(['GET', 'PUT', 'PATCH'])

def product\_detail(request, id):

We can use PUT to update all properties or PATCH to update few properties or their subset. For reasons of simplicity we will keep only the PUT method here.

Applying the if-elif conditions based on request.method,

@api\_view(['GET', 'PUT'])

def product\_detail(request, id):

    if request.method == 'GET':

        product = get\_object\_or\_404(Product, pk=id)

        serializer = ProductSerializer(product)

        return Response(serializer.data)

    elif request.method == 'PUT':

In case of ‘PUT’, we need to deserialize the data, validate it and save the product object in database.

While Deserializing we pass data to our ProductSerializer as request.data

serializer = ProductSerializer(data=request.data)

But in addition to this, we should also pass a product instance. So that serializer try to update the attributes of that product using the data in the request.

Here is the product instance if you remember from before…

product = get\_object\_or\_404(Product, pk=id)

Keep this line outside of both blocks and use product in PUT request serializer.

@api\_view(["GET", "PUT"])

def product\_detail(request, id):

    product = get\_object\_or\_404(Product, pk=id)

    if request.method == "GET":

        serializer = ProductSerializer(product)

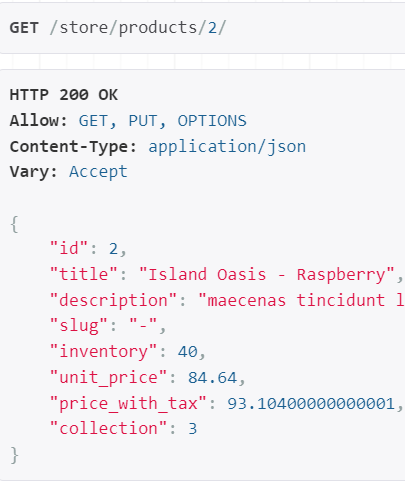
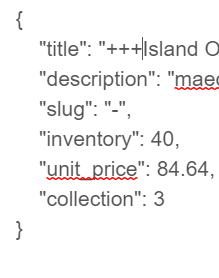
        return Response(serializer.data)

    elif request.method == "PUT":

        serializer = ProductSerializer(product, data=request.data)

Now, This *PUT request serializer* is instantiated with object to update which is product and request data.

Let us update a product with id 2…

🡪 🡪

**Deleting Objects**:

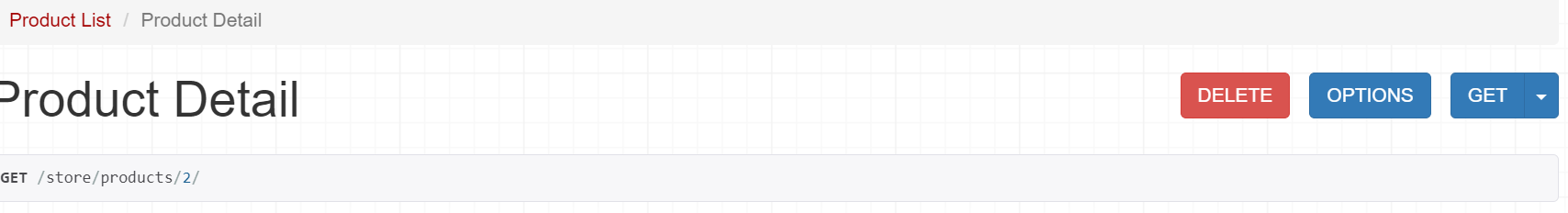
Let us delete a product, we are going to implement this functionality in the product detail view function, because this is where we work with a particular product.

First we declare ‘DELETE’ in api\_view decorator,

@api\_view(["GET", "PUT", 'DELETE'])

def product\_detail(request, id):

As soon as we do this,



A delete option appears on our browsable API page. Now we will implement this operation

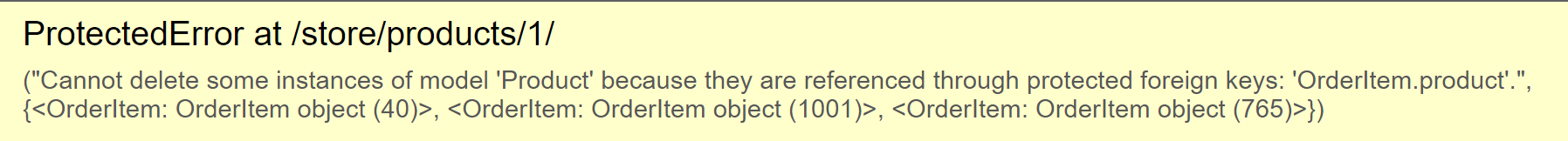
    elif request.method == 'DELETE':

        product.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

Here, we simply delete product and return a response of 204 no content. Quite often when we delete a resource we return an empty response with a status of 204.

Now let us test up to this point by going to product with id =1 and clicking on Delete button in our browsable API page.



Basically what this exception is saying is that we have order items that are associated with this product. So we cannot delete this product.

Now we cannot show this exception to the user, instead we need to return a proper response and in the body of response we should include an error message.

In the body of *request.method == ‘DELETE’*, let us first check to see if there are any order items which include this product

    elif request.method == "DELETE":

        if product.orderitem\_set.count > 0:

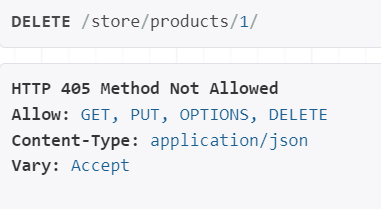
            return Response(status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED)

        product.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

If count>0, We return 405 response.

This time after delete we get,



Note: To check the appropriate HTTP status code to use in your code refer to httpstatuses.com.

*Include our custom message in body of response*:

Pass a dictionary as first argument in the Response, which will be converted to a JSON object.

        if product.orderitem\_set.count() > 0:

            return Response({},status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED)

In this dictionary we can add a key called error and set it to our own custom message.

        if product.orderitem\_set.count() > 0:

            return Response(

                {

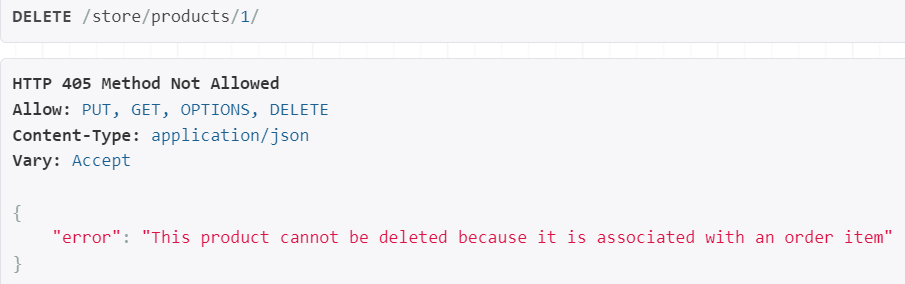
                    "error": "This product cannot be deleted because it is associated with an order item"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

This we get the error which end user can understand,



**Exercise**:

Implement a collections end point (store/collections/). At this end point we can get all the collections in our database as well as the number of products in each collection.

POST method should be available to create a new collection.



Another end point should be available to work with a particular collection, so that we can read /delete / update that collection.

Solution:

🡪 First create a *collection\_list* view function which lets us access all the collections.

@api\_view(["GET", "POST"])

def collection\_list(request):

    if request.method == "GET":

        queryset = Collection.objects.annotate(products\_count=Count("product")).all()

        serializer = CollectionSerializer(

            queryset, many=True, context={"request": request}

        )

        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)

Note: Here we *annotate* *products\_count* field and give it value of number of products assigned to this collection using *Count*.

We also add this *products\_count* field in CollectionSerializer like this,

class CollectionSerializer(serializers.ModelSerializer):

    class Meta:

        model = Collection

        fields = ["id", "title", "products\_count"]

    products\_count = serializers.IntegerField()

Note: It is one of the RESTful convention that when an end point creates a new resource, it should return a response with a status of 201.

        return Response(serializer.data, status=status.HTTP\_201\_CREATED)