

**Introduction**:

In this section we are going to put everything together and build a shopping cart API from A – Z. This is a fantastic opportunity for us to practice everything we have learned so far.

**Designing the API**:

So what operations do we need to support for a Shopping cart,

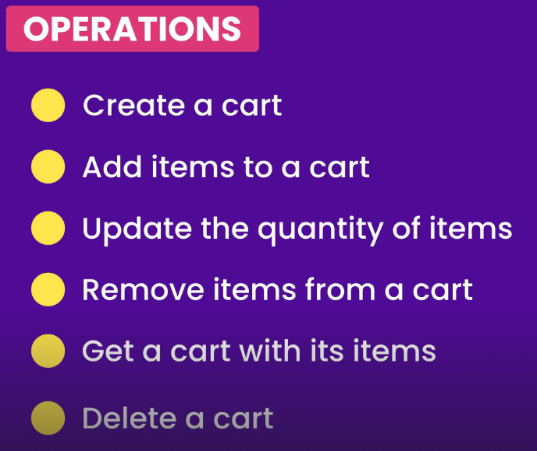


Figure out what endpoints we need to support,



For each operation, specify which *type* of HTTP request is sent to server (*get or post…*). Specify the *URL end point* as well as what we are going to send in *request* and what we will receive in *response*.

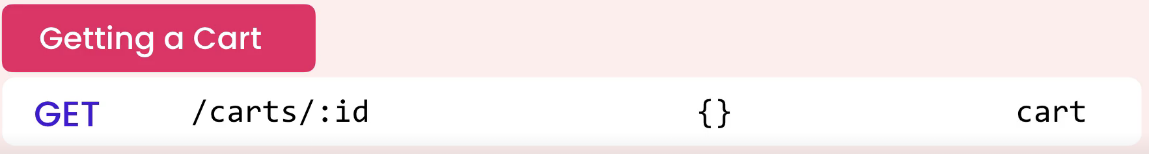
For example, for ***creating a Cart*** we are going to send a POST request to the carts endpoint.



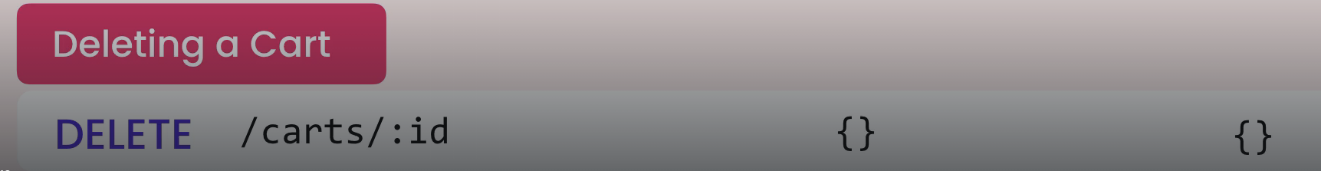
Note: The body of request is empty here, because our carts are anonymous. We do not want to force users to *login* before they can add items to their shopping cart. So *when creating a cart we are not going to send someone’s user ID or customer ID to get a cart back*. *So we send a* ***POST*** *request to this endpoint and get a* ***cart objec****t back from the server*.

This cart object has a unique identifier that we are going to save on the client for subsequent requests. So when the user adds an item to their shopping cart, we are going to send cart ID back to the server.

For ***getting a Cart***, we are going to send a GET request to the other end point and obviously we get a cart object back.



For ***deleting a cart***, we are going to send a delete request to this end point.



Now let us talk about cart items,

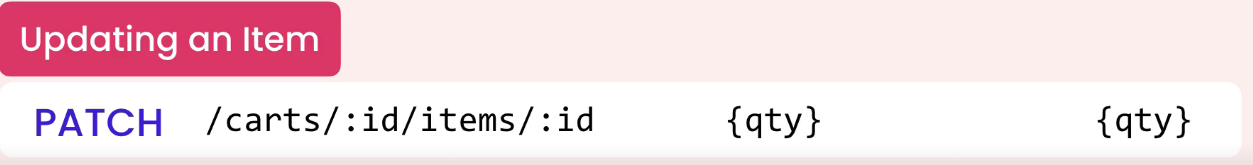
For ***adding an item to a cart***, we are going to send a post request to below end point. Since this end point represents the items of a particular shopping cart therefore we have a shopping cart ID.



In the body of the request we are only going to send two attributes, Product ID and quantity. Since we have cart ID in the URL, so all we need to send to the server is product ID and quantity. So we send this request and we get the item that was created.

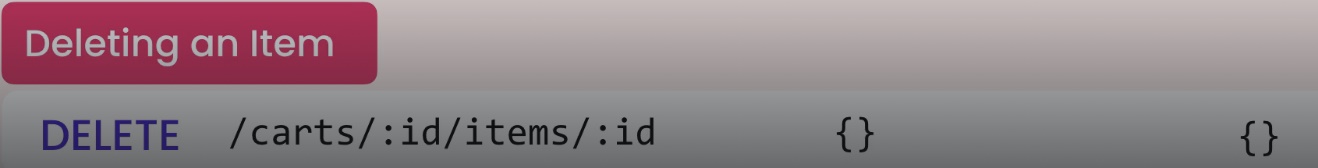
This item has a unique identifier which we will use for subsequent requests.

For ***updating a cart***, we are going to send a patch request to this other end point. Here we have two URL parameters. First one is cart id and other one is item id. In the body of the request we will get quantity and in the response we will get updated quantity.

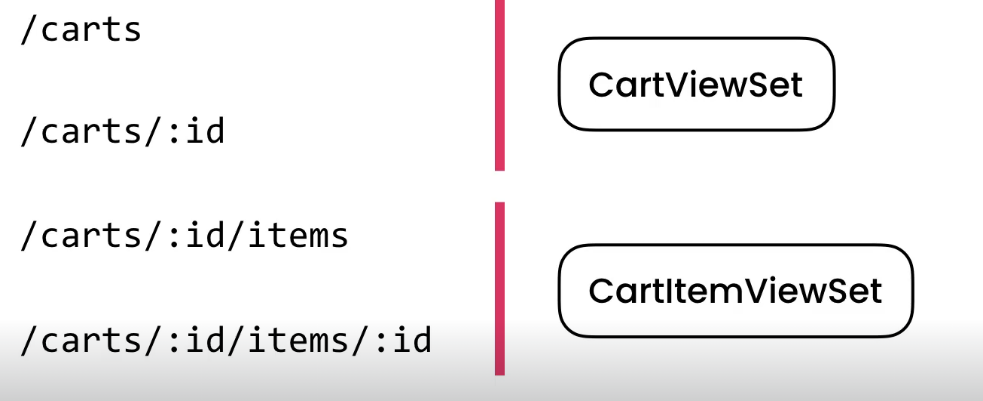


Note: Here we are only going to use patch request, not put request because with PUT we can replace an entire request.

For ***deleting an item***, we will just send a DELETE request to this end point.



So if we put all this together essentially we have 4 new end points. Two for carts and two for cart items.



We can implement the first two endpoints using a class called CartViewSet and the other endpoints using CartItemViewSet.

**Revisiting the Data Model**:

Before we build our API, we need to address a couple of issues in our data model. So let us look at our cart model.

class Cart(models.Model):

    created\_at = models.DateTimeField(auto\_now\_add=True)

Here we have a single field called created\_at, but as we know *Django automatically gives each model a primary key field and type of this field is an integer*.

So that means in our API, we are going to have a URL like this,

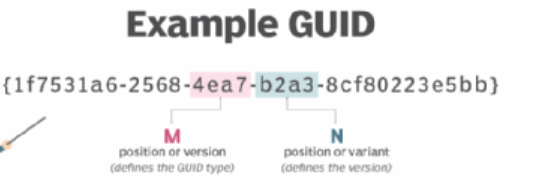
<http://127.0.0.1:8000/store/carts/2>

But there is a problem with this implementation. Which is a hacker can easily guess someone else’s cart ID and send a request to this endpoint to mess with that cart (*since it is really easy to guess these numbers 1,2,3,4 …*).

So to solve this problem, we are going to use

**GUID**: *Globally Unique Identifier*

Which is a long 32 character string.



It will make it much harder for a hacker to guess that string.

So back to our model, we need to redefine the primary key field.

class Cart(models.Model):

    id = models.UUIDField(primary\_key=True, default=uuid4)

    created\_at = models.DateTimeField(auto\_now\_add=True)

We give our id field a default value of *uuid4* function which we import from *uuid* module of python.

from uuid import uuid4

When we create a cart object, django will automatically assign the *id* field to uuid.

Note: We are not calling *uuid ()* function just passing a reference to it (*default = uuid*). *If we call this function then at the time of creating a migration, a GUID will be generated and hardcoded into our migration file*.

Let us see this in action,

class Cart(models.Model):

    id = models.UUIDField(primary\_key=True, default=uuid4()) //call uuid function

    created\_at = models.DateTimeField(auto\_now\_add=True)

Then *makemigrations* and look at the migration file,

class Migration(migrations.Migration):

    dependencies = [

        ('store', '0006\_remove\_cart\_title'),

    ]

    operations = [

        migrations.AlterField(

            model\_name='cart',

            name='id',

            field=models.UUIDField(default=uuid.UUID('532558cc-8b53-4504-9e94-7128a5d9206e'), primary\_key=True, serialize=False),

        ),

    ]

Look we have GUID and this is hardcoded into our migration file. We do not want to use the same value for every shopping cart. So that is why we should not call this function just pass a reference to it.

So let us delete the latest migration file and just pass uuid4 as a reference.

class Cart(models.Model):

    id = models.UUIDField(primary\_key=True, default=uuid4) //just reference

    created\_at = models.DateTimeField(auto\_now\_add=True)

Now recreate the migration,

class Migration(migrations.Migration):

    dependencies = [

        ('store', '0006\_remove\_cart\_title'),

    ]

    operations = [

        migrations.AlterField(

            model\_name='cart',

            name='id',

            field=models.UUIDField(default=uuid.uuid4, primary\_key=True, serialize=False),

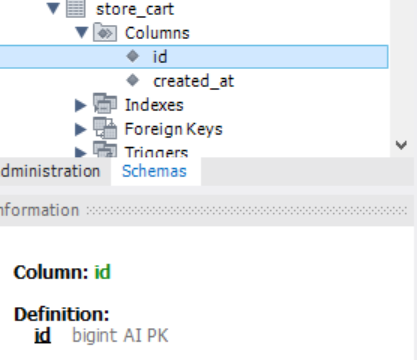
        ),

    ]

In this new migration, we do not have a hard coded GUID. The default is set to the reference to the uuid function of python.

Before we run this migration, let us talk about implications of making this change.

In our database, look at the cart table. Type of our primary key id is *big integer* (*takes 8 bytes in MySQL*). If convert this into GUID, we are going to store 32 bytes here.



In cartitem table we also have reference to cart\_id.

🡨 So for each record in this table, we are going to store 24 bytes extra data.

*Is this going to be an issue*?

Well it really depends. Some people are totally against using GUID in database because they take extra space and performance wise they are bit slower than integers.

But let us do some real assessment here. Let us imagine in our cartitem table, we have one million records.

1000000 \* 24 bytes = 24000000 bytes (*extra space in bytes*)

24000000 / 1024 = 23437.5 KB

23437.5 / 1024 = 22.88 MB (*extra space of data required*)

Is this an issue?

Well these days, disk space is really cheap and 22MB extra for one million records is really nothing (*also take into account that this is a temporary table, not going to grow indefinitely*). As people place orders we will move these records into order and orderitem table and tables will simply use integers not GUIDs.

(*Orders API that we will build later is going to be secure and going to be open to anonymous users, so a client has to authenticate and be authorized to access a particular order*)

Also take into account that in cart table, we have a created\_at field and with this we can keep track of abandoned carts. So if some people add items to the shopping cart and then forget about it. Every now and then we can clear this table (*we can run a job and delete all carts that are more than 3 months old*).

So our table is going to grow quite large.

*Now what about performance*?

Yes, in theory looking up a GUID key is slower than an integer key, but these days servers are powerful and also all these database engines are highly optimized. So we do not need to do any optimization here before we do a proper test and make sure there is going to be an issues.

**Premature Optimization is the root of all evils**.

-Donald Knuth

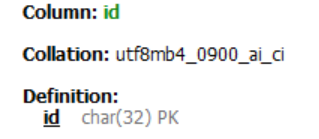
*In case, we need to do some optimization for performance*. We can keep the structure as it is (*bigint for cart*). But also add an additional column called uid or unique\_id or cart\_id whatever.

In that column we can store a GUID. So with this change we are not going to store a bunch of GUIDs into cartitem table (*only integers*).

But since each cart has a GUID, *in our API layer instead of a number we are going to use GUIDs but internally we will translate that GUID to a number*.

So this will complicate our queries a little bit. That is why premature optimization is not necessary at this point.

So keep the current change and run our migrations.



Our cart table has id of type char (32). String of 32 characters.

There are two more changes we need to apply here for the cartitem model.

class CartItem(models.Model):

    cart = models.ForeignKey(Cart, on\_delete=models.CASCADE)

    product = models.ForeignKey(Product, on\_delete=models.CASCADE)

    quantity = models.PositiveSmallIntegerField()

First change is that cart is a foreign key in cartitem model. But we will assign a related name to ‘items’. *So that means in our cart table will have a field called* ***items*** *instead of* ***cartitem\_set***.

class CartItem(models.Model):

    cart = models.ForeignKey(Cart, on\_delete=models.CASCADE, related\_name="items")

    product = models.ForeignKey(Product, on\_delete=models.CASCADE)

    quantity = models.PositiveSmallIntegerField()

The other change is to apply a unique constraint in cartitem table. Because *in this table we need to make sure that we only have a single instance of product in a shopping cart*. So if the client adds the same product in the same cart multiple times, instead of creating multiple products, we should only increase the quantity.

So using the unique constraint we can make sure there are no duplicate records for the same product in same cart.

We can do this using a *Meta* class.

    class Meta:

        unique\_together = [[]]

Here we set unique\_together attribute to list of lists, because we can have multiple unique constraints on different fields.

For example we can have a constraint on ‘cart’ and ‘product’ but we can have another constraint on 2 or 3 other fields. That is why we have a list of lists.

class CartItem(models.Model):

    cart = models.ForeignKey(Cart, on\_delete=models.CASCADE, related\_name="items")

    product = models.ForeignKey(Product, on\_delete=models.CASCADE)

    quantity = models.PositiveSmallIntegerField()

    class Meta:

        unique\_together = [["cart", "product"]]

Let us create another migration.

class Migration(migrations.Migration):

    dependencies = [

        ('store', '0007\_alter\_cart\_id'),

    ]

    operations = [

        migrations.AlterField(

            model\_name='cartitem',

            name='cart',

            field=models.ForeignKey(on\_delete=django.db.models.deletion.CASCADE, related\_name='items', to='store.cart'),

        ),

        migrations.AlterUniqueTogether(

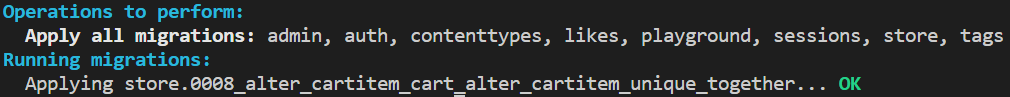
            name='cartitem',

            unique\_together={('cart', 'product')},

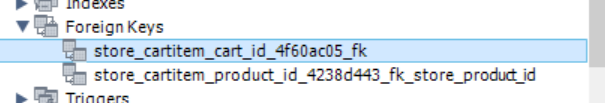
        ),

    ]

And run it



Our cartitem table now has a unique constraint on cart\_id and product\_id.



So our data model is in good shape. From next lesson we will build our API step by step.

**Creating a cart**:

We will start by creating a *CartSerializer*.

Remember our cart model.

class Cart(models.Model):

    id = models.UUIDField(primary\_key=True, default=uuid4)

    created\_at = models.DateTimeField(auto\_now\_add=True)

On the basis of our model, we will create our serializer,

class CartSerializer(serializers.ModelSerializer):

    class Meta:

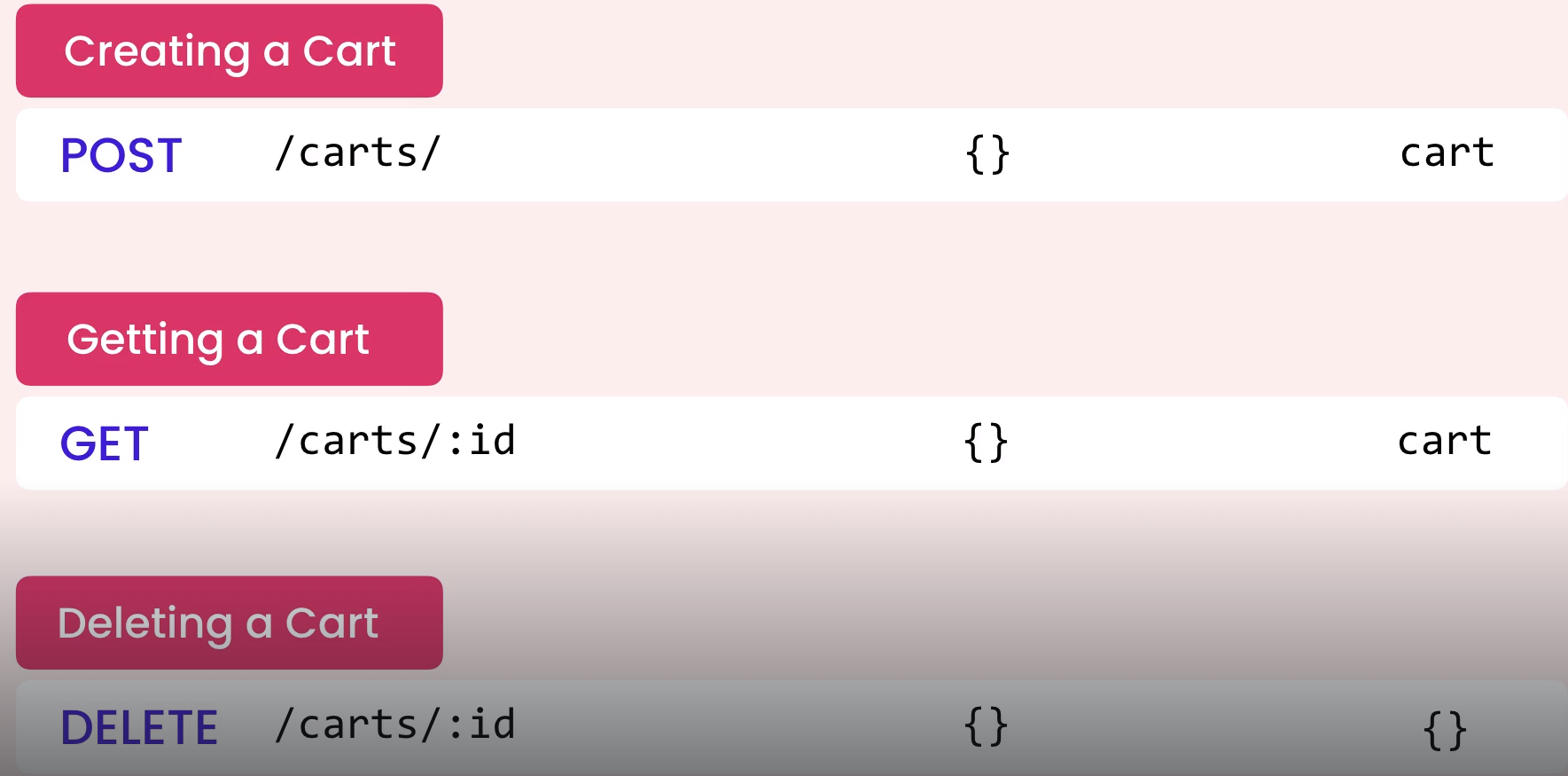
        model = Cart

        fields = ["id"]

Notice that we are not using *created\_at* field in the serializer it is because we do not need this field to return to client because it will be only used on the server for maintenance purposes.

Now we create a view called CartViewSet. The tricky part is we will not inherit from *ModelViewSet* class because this provides all operations like list, retrieve, create, update and delete.

But look at the operations we need to support our cart.



We need to be able to create a cart, get a cart and delete it, here we do not have a list operation (*therefore not a GET request to the cart endpoint and retrieve all carts*) otherwise all our cart IDs will be exposed to the outside.

Similarly, we do not have an update operation, so we cannot send a PUT or PATCH request because *it does not really make sense to update a cart, we only update the quantity of a cart item*.

So instead of extending to ModelViewSet we create a custom viewset. We can do this by using mixins, as you can see ModelViewSet is also a combination of various mixins with a *GenericViewSet*.

class ModelViewSet(mixins.CreateModelMixin,

                   mixins.RetrieveModelMixin,

                   mixins.UpdateModelMixin,

                   mixins.DestroyModelMixin,

                   mixins.ListModelMixin,

                   GenericViewSet):

    """

    A viewset that provides default `create()`, `retrieve()`, `update()`,

    `partial\_update()`, `destroy()` and `list()` actions.

    """

    pass

Here we do not need *ListModelMixin* and *UpdateModelMixin*.

To implement *CreateModelMixin* let us import it from rest framework mixins module along with *GenericViewSet* class from rest framework viewsets module.

from rest\_framework.viewsets import ModelViewSet, GenericViewSet

from rest\_framework.mixins import CreateModelMixin

And in our viewset,

class CartViewSet(CreateModelMixin, GenericViewSet):

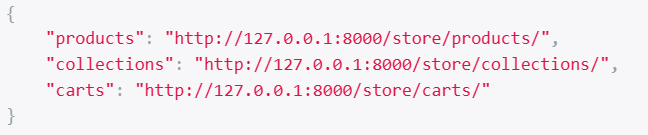
    queryset = Cart.objects.all()

    serializer\_class = CartSerializer

Next, we need to register our route, So in our urls.py.

router.register("carts", views.CartViewSet)

If we see in our API Root,



We have carts endpoint available now, which if we click, we get,

, because we have not implemented GET operation for now.

But we can POST to this endpoint. But there is a tiny issue.



We do not want to send the ID to the server only read it from the server and send an empty object.

We are seeing id field in the content box because we have mentioned it in our serializer as a field. *Here we need to declare this field as read only, so that we do not have to send it to the server and only going to read from the server*.

class CartSerializer(serializers.ModelSerializer):

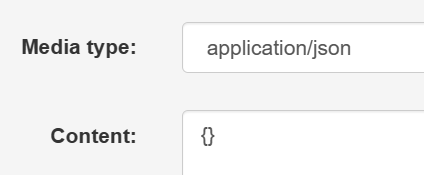
    id = serializers.UUIDField(read\_only=True)

    class Meta:

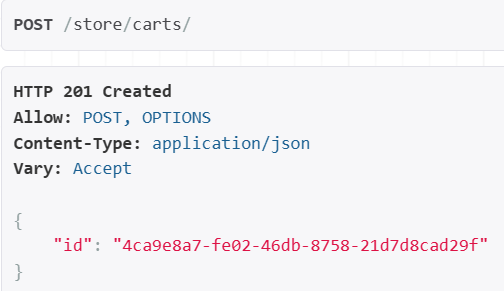
        model = Cart

        fields = ["id"]

Now we have an empty object,

, so let us POST it.

And we get a new cart with this ID,

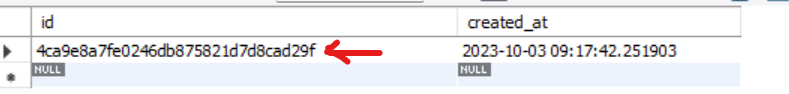


**Getting a cart**:

Now we are going to implement getting a cart.

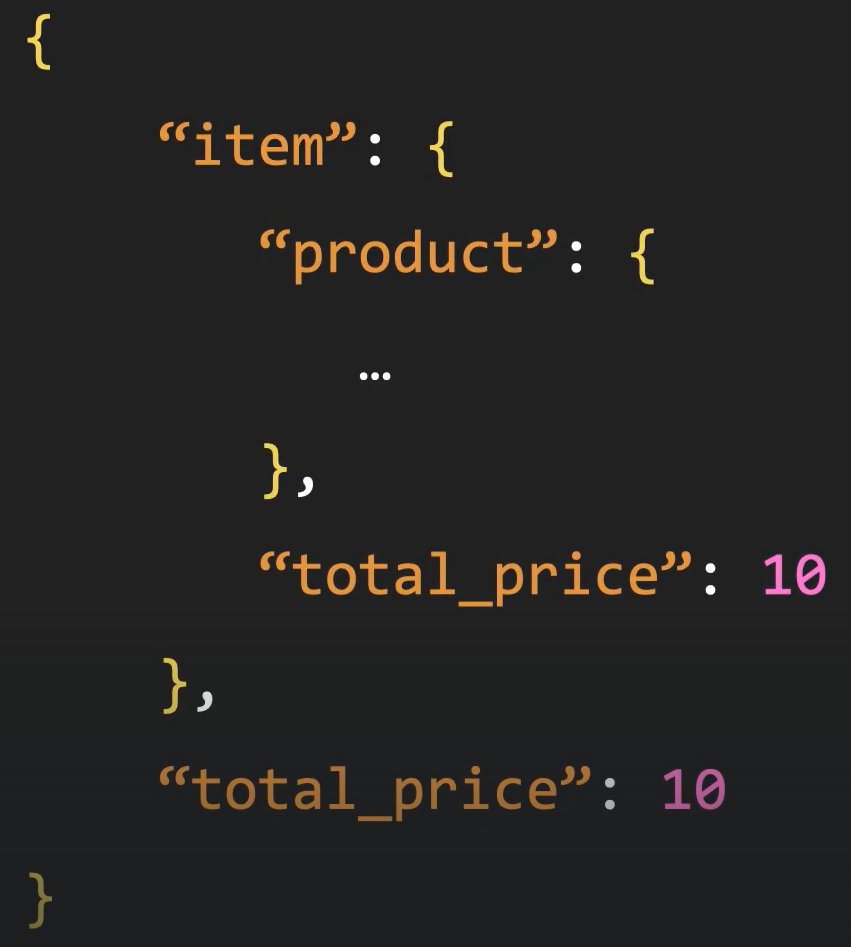
Here is what we need to implement for this operation.

🡪 After creating the cart, grab the cart id and add it in the URL.



🡪 Send a GET request to the URL and get that particular cart with all its items.

Currently we do not have a way to add items to a shopping cart, so we will go in the database and *manually populate 3 items in the cartItem table and then return all those items in our shopping cart*.



Now each item should include a product object as well as total price (*unit price \* quantity*). The cart object itself should also include total price.

Currently we are only displaying our cart id in our CartSerializer,

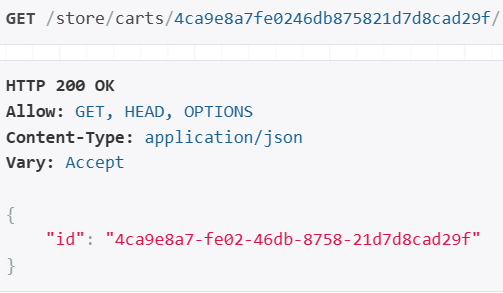
class CartSerializer(serializers.ModelSerializer):

    id = serializers.UUIDField(read\_only=True)

    class Meta:

        model = Cart

        fields = ["id"]



But not returning any cart items.

So let us add ‘items’ field in our CartSerializer,

class CartSerializer(serializers.ModelSerializer):

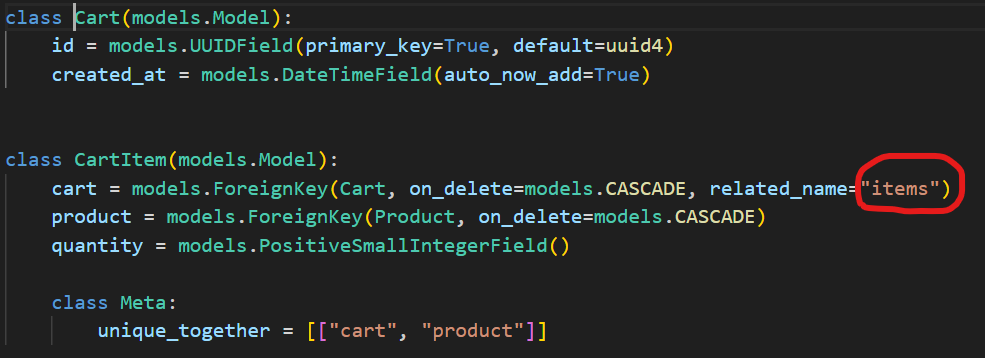
    id = serializers.UUIDField(read\_only=True)

    class Meta:

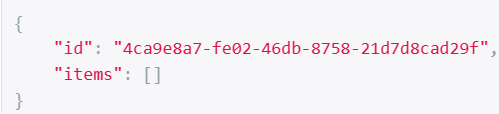
        model = Cart

        fields = ["id", "items"]

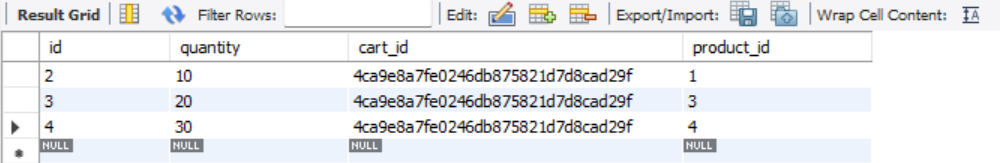
The reason we add a field called ‘*items’* is because CartItem table has a related\_name as ‘items’ in Cart table.



Now if we refresh our API page,



We get an empty array because our database does not have any data inside cartItem table. So, we will populate it now.



Now in the browser we refresh,



We have cart item ids here, but we want to return actual cart item objects, so here we need a serializer for cart items.

class CartItemSerializer(serializers.ModelSerializer):

    class Meta:

        model = CartItem

        fields = ["id", "product", "quantity"]

Now we need to use it in our CartSerializer, so we explicitly define items field and set it to CartItemSerializer like this,

class CartSerializer(serializers.ModelSerializer):

    id = serializers.UUIDField(read\_only=True)

    items = CartItemSerializer(many=True)

    class Meta:

        model = Cart

        fields = ["id", "items"]

Refreshing the page, we see



If instead of product id, we want to see actual product. We can explicitly define product field in cart item serializer and set it to ProductSerializer.

class CartItemSerializer(serializers.ModelSerializer):

    product = ProductSerializer()

    class Meta:

        model = CartItem

        fields = ["id", "product", "quantity"]

It seems we are getting all fields of product object.



What if we do not want to display all these fields here only selective fields. To do this we have to create another product serializer like this.

class SimpleProductSerializer(serializers.ModelSerializer):

    class Meta:

        model = Product

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

We can potentially reuse this in other situations where we want to return basic information about a product like id, title and unit price.

class CartItemSerializer(serializers.ModelSerializer):

    product = SimpleProductSerializer() 🡪 now we use this...

    class Meta:

        model = CartItem

        fields = ["id", "product", "quantity"]

And get only these fields.



Next, we will add total price for each item. This is going to be a calculated field.

class CartItemSerializer(serializers.ModelSerializer):

    product = SimpleProductSerializer()

    total\_price = serializers.SerializerMethodField(method\_name="get\_total\_price")

    def get\_total\_price(self, cart\_item: CartItem):

        return cart\_item.quantity \* cart\_item.product.unit\_price

🡪 10 \* 4 = 40

    class Meta:

        model = CartItem

        fields = ["id", "product", "quantity", "total\_price"]

In CartItemSerializer we added a new field called *total\_price* and set it to serializers.SerializerMethodField.



After implementing total price for cart items, we also need to implement total price for our cart.

In our get\_total\_price method we get prices from each item. For this purpose, we use *list comprehension*. It has a syntax of [*item for item in collection*]

    def get\_total\_price(self, cart: Cart):

        return sum(

            [item.quantity \* item.product.unit\_price for item in cart.items.all()]

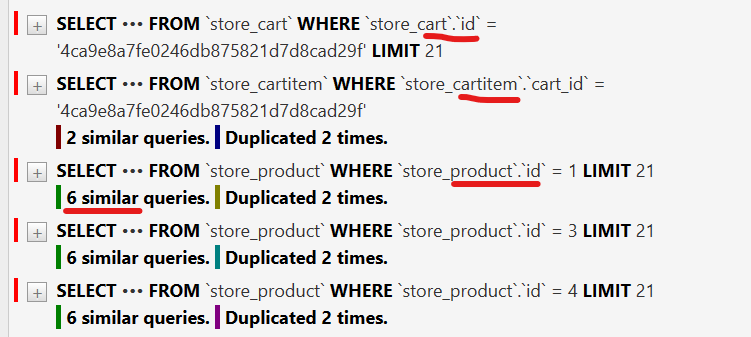
        )

Here our collection is *cart.items.all()* returns a queryset with all the items.



We got the total price of all the items in shopping cart now. But look at the Django debug toolbar. We have 11 queries to produce this result, which does not seem right.

First, we are getting a cart then we are getting items for this cart, then for each product we have an extra query to read the attributes of that product (*like title and unit\_price*).



This is where we need to use eager loading. So, when retrieving a cart, we want to eager load that cart with its items and products. *We will do this operation in our viewset*.

So back to our CartViewSet.

class CartViewSet(CreateModelMixin, GenericViewSet, RetrieveModelMixin):

    queryset = Cart.objects.all()

    serializer\_class = CartSerializer

This is where we have the queryset for retrieving our cart. Here we are going to call *prefetech\_related* *(since a cart can have multiple items*). *For foreign keys where we have a single related object, we use select\_related*.

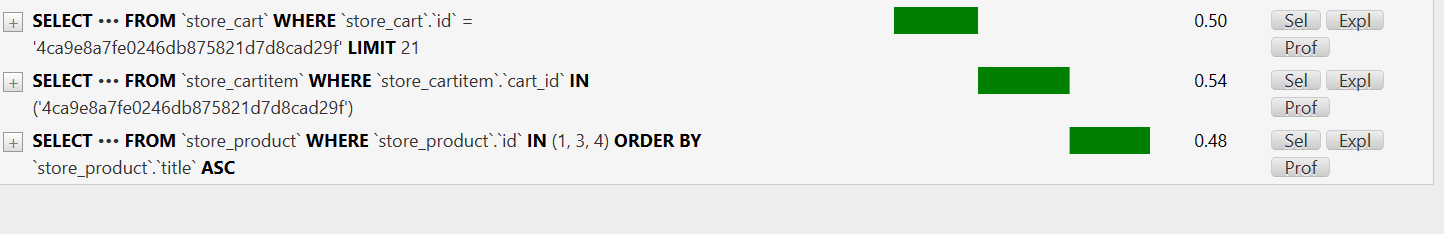
class CartViewSet(CreateModelMixin, GenericViewSet, RetrieveModelMixin):

    queryset = Cart.objects.prefetch\_related("items\_\_product").all()

    serializer\_class = CartSerializer

So we want to prefetch a cart with its items. Also for each item, we also want to preload the product (*so items\_\_product*).

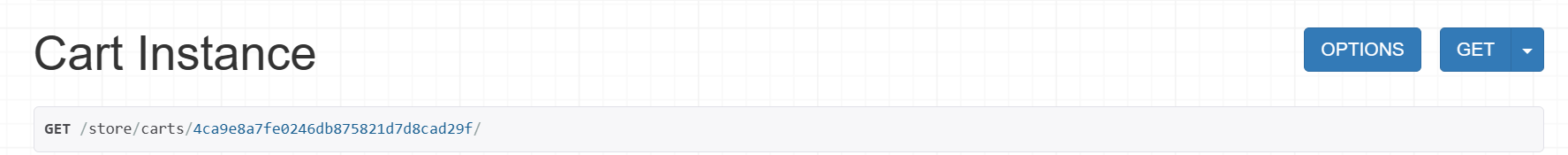
Now we see only 5 queries,



For getting a cart we have a query. Then next query is for retrieving items in that cart and finally one single query to retrieve all products referenced in this cart. So now we do not have a separate product query per cart item.

**Deleting a cart**:

Currently we do not support delete operation at this endpoint



That is why we do have a delete button here. But adding that button is as simple as adding another mixin in our viewset.

class CartViewSet(CreateModelMixin, GenericViewSet, RetrieveModelMixin):

    queryset = Cart.objects.prefetch\_related("items\_\_product").all()

    serializer\_class = CartSerializer

Currently we are supporting create and retrieve operations. So we will add another mixin here called DestroyModelMixin.

class CartViewSet(

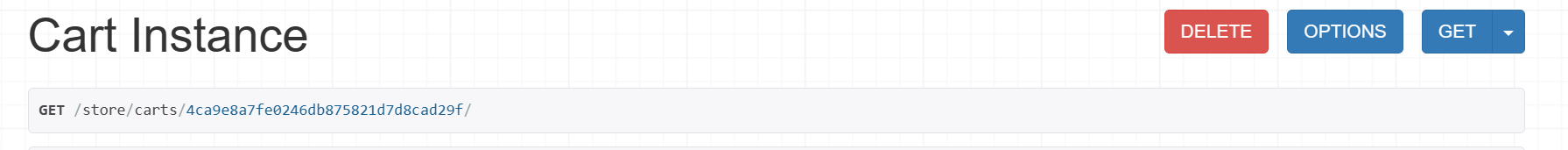
    CreateModelMixin, RetrieveModelMixin, DestroyModelMixin, GenericViewSet

):

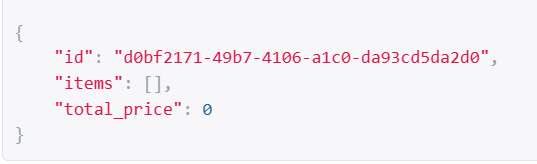
    queryset = Cart.objects.prefetch\_related("items\_\_product").all()

    serializer\_class = CartSerializer

After this change we have a delete button now,



We already have some data in our last cart so let us not delete it instead we will create a new empty cart.



Now if we delete it and refresh the page,

 we get 404.

**Getting cart items**:

This is what we need to implement,

🡪 If we go to *cart\_id/items* endpoint we should be able to see all the items in this particular cart\_id (*difference between this end point and previous one is that here we do not have cart specific properties like cart\_id and total\_price*) only array of cart items.

🡪 Second if we go to *cart\_id/items/cartitem\_id*, we should be able to see only that cartitem.

To implement this, we are going to create a new viewSet called CartItemViewSet and have it extend *ModelViewSet*, because here we support all operations (list cart items, retrieve a single item as well as update and delete operations).

class CartItemViewSet(ModelViewSet):

    def get\_queryset(self):

        return CartItem.objects.filter(cart\_id=self.kwargs["cart\_pk"])

    serializer\_class = CartItemSerializer

Here we do not want all cart items(*means not all cart items in all the cart id’s*) instead we want to filter them as per specific cart id from our URL.

Now, we need to register the routes. Since we have nested resources therefore we need to have nested routers.

carts\_router = routers.NestedDefaultRouter(router, "carts", lookup="cart")

carts\_router.register("items", views.CartItemViewSet, basename="cart-items")

Note: In *carts\_router* our lookup parameter is ‘cart’. So we will have a URL parameter called *cart\_pk*. That is how we extracted CartItem.objects.filter(cart\_id=self.kwargs["***cart\_pk***"])

With basename = ‘cart-items’, we will have two routes called cart-items-list and cart-items-detail.

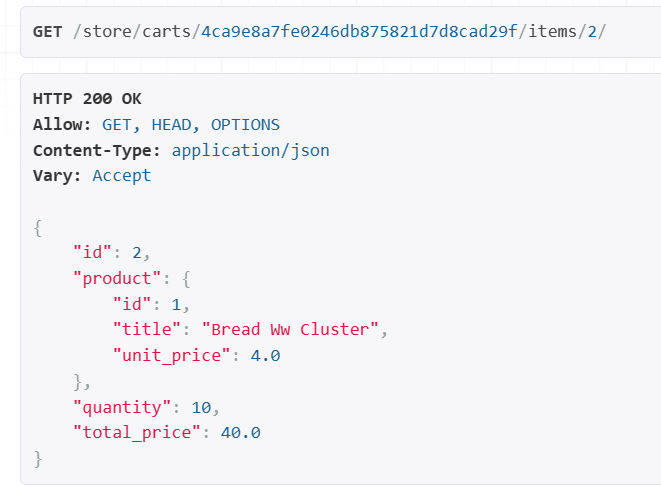
Then we append it in our urlpatterns,

urlpatterns = router.urls + products\_router.urls + carts\_router.urls

We go to our items endpoint now,

🡨 We get all items in this cart.

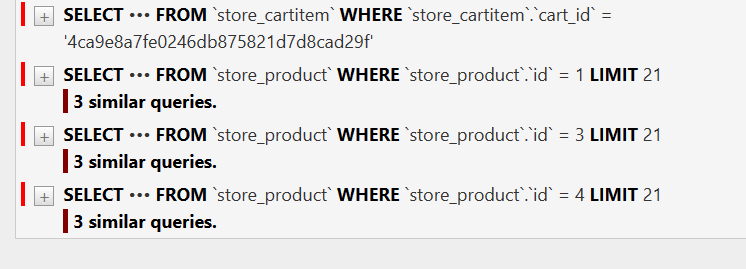
Now we go to a single item to retrieve its details,



If we try to access an item that does not exist in this cart,

🡨 we get 404.

Let us look at Django debug toolbar to see if do not have any extra queries.



We can see one query for loading cart items and 3 extra queries for reading product referenced by each item. This is unnecessary.

class CartItemViewSet(ModelViewSet):

    def get\_queryset(self):

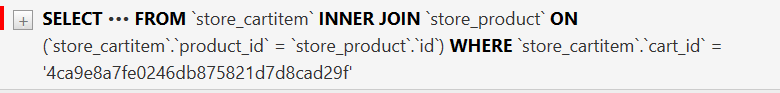
        return CartItem.objects.select\_related("product").filter(

            cart\_id=self.kwargs["cart\_pk"]

        )

    serializer\_class = CartItemSerializer

So we use *select\_related* to eager load ‘product’ table and now we have a single query (*inner join between product and cart item table*).



**Adding a Cart Item**:

If you go to items end point of a cart observe POST raw data,



This is the object that we need to pass to the server to add a product to this cart. What we see here is generated based on our *CartItemSerializer*.

class CartItemSerializer(serializers.ModelSerializer):

    product = SimpleProductSerializer()

    total\_price = serializers.SerializerMethodField(method\_name="get\_total\_price")

    def get\_total\_price(self, cart\_item: CartItem):

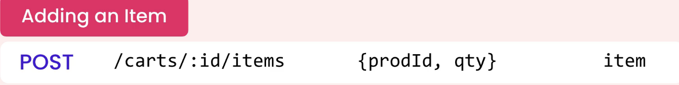
        return cart\_item.quantity \* cart\_item.product.unit\_price

    class Meta:

        model = CartItem

        fields = ["id", "product", "quantity", "total\_price"]

But we do not require to pass a product object with title and unit\_price, it does not make sense. All we really need is prod\_id and quantity.



One way to solve this problem (*not recommended, since it is ugly and redundant*),

class CartItemSerializer(serializers.ModelSerializer):

    product = SimpleProductSerializer(read\_only=True)

    total\_price = serializers.SerializerMethodField(method\_name="get\_total\_price")

    def get\_total\_price(self, cart\_item: CartItem):

        return cart\_item.quantity \* cart\_item.product.unit\_price

    class Meta:

        model = CartItem

        fields = ["id", "product", "product\_id", "quantity", "total\_price"]

Mark the product field as *read\_only* and add product\_id as a separate field inside meta class. It is not favourable when updating a cart item we don’t send product\_id just quantity.

What we really need here is a different object for adding an item to a shopping cart, in short a different serializer called *AddCartItemSerializer*.

class AddCartItemSerializer(serializers.ModelSerializer):

    class Meta:

        model = CartItem

        fields = ["id", "product\_id", "quantity"]

In our CartItemViewSet, we have this serializer class hardcoded instead *we want to dynamically return a serializer class depending on the request method*.

class CartItemViewSet(ModelViewSet):

    def get\_queryset(self):

        return CartItem.objects.select\_related("product").filter(

            cart\_id=self.kwargs["cart\_pk"]

        )

    serializer\_class = CartItemSerializer 🡪 **Hardcoded serializer**

So we will override it using *get\_serializer\_class* method.

class CartItemViewSet(ModelViewSet):

    def get\_queryset(self):

        return CartItem.objects.select\_related("product").filter(

            cart\_id=self.kwargs["cart\_pk"]

        )

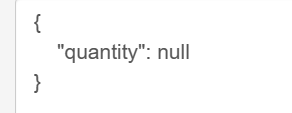
    def get\_serializer\_class(self):

        if self.request.method == "POST": 🡪 Serializer depending on request

            return AddCartItemSerializer

        return CartItemSerializer

Now we get the same result as before,

 and we don’t have product object here.

But where is *product\_id*?

Even though our AddCartItemSerializer has a product\_id attribute. *This attribute is generated dynamically at run time, so it is not a field we can reference here*.

So we have to explicitly define this field.

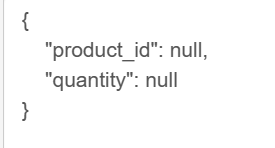
class AddCartItemSerializer(serializers.ModelSerializer):

    product\_id = serializers.IntegerField()

    class Meta:

        model = CartItem

        fields = ["id", "product\_id", "quantity"]

🡨Now we get product\_id and this is the object we will send to server.

Now let us implement the saving part because it is a little bit tricky because *when we add the same product to the cart multiple times, we do not want to create multiple cart item records, instead we want to update the quantity of an existing item*.

So in the AddCartItemSerializer we cannot rely on default implementation of the *save* method that comes from *ModelSerializer*. We have to reimplement the *save* method based on the requirements of our application.

In our save method, we need product\_id and quantity. Where do we get it? Well *behind the scenes there is a call to serializer.is\_valid(), when the data gets validated, we can get it from serializer.validated\_data (which is a dictionary)*.

    def save(self, \*\*kwargs):

        product\_id = self.validated\_data["product\_id"]

Since we are under our serializer, we can read the product id that we received from the client store it inside *product\_id*.

Similarly we can read the quantity as well,

    def save(self, \*\*kwargs):

        product\_id = self.validated\_data["product\_id"]

        quantity = self.validated\_data["quantity"]

But what about cart\_id, which is not in the request? Well it is available in the URL, but in the serializer we do not have access to URL parameters.

So go to *CartItemViewSet* and get the URL parameter from there then using a context object, pass it to the serializer.

class CartItemViewSet(ModelViewSet):

    def get\_queryset(self):

        return CartItem.objects.select\_related("product").filter(

            cart\_id=self.kwargs["cart\_pk"]

        )

    def get\_serializer\_context(self): 🡨 We get cart\_id from here,

        return {"cart\_id": self.kwargs["cart\_pk"]}

    def get\_serializer\_class(self):

        if self.request.method == "POST":

            return AddCartItemSerializer

        return CartItemSerializer

Then we read the cart\_id from context object inside our serializer’s save method,

    def save(self, \*\*kwargs):

        cart\_id = self.context["cart\_id"]

        product\_id = self.validated\_data["product\_id"]

        quantity = self.validated\_data["quantity"]

Now the save logic. We will start with *CartItem.objects.get* and get a cart item with two attributes (*cart\_id and product\_id*).

cart\_item = CartItem.objects.get(cart\_id=cart\_id, product\_id=product\_id)

So we get a cart\_item. If there is no such cart\_item, this line will throw an exception. So we need to wrap this inside a try – except block like this,

        try:

            cart\_item = CartItem.objects.get(cart\_id=cart\_id, product\_id=product\_id)

            #Updating an existing item

        except CartItem.DoesNotExist:

            #Creating a new item

Inside try block we will update an existing item and inside except we will create a new item.

Furthermore after we get cart\_item, let us update quantity.

        try:

            cart\_item = CartItem.objects.get(cart\_id=cart\_id, product\_id=product\_id)

            #Updating an existing item

            cart\_item.quantity += quantity 🡪 Update quantiy

            cart\_item.save() 🡪 then Save

And if the item does not exist,

        except CartItem.DoesNotExist:

            #Creating a new item

           CartItem.objects.create(

                cart\_id=cart\_id, \*\*self.validated\_data

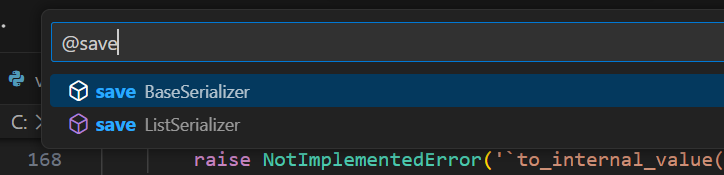
            )

We can mention product\_id = product\_id , quantity = quantity as well in our *create* method. But this is redundant so instead we unpack \*\*self.validated\_data dictionary like above.

So we created an object, but now is the tricky part.

Let us look at the default implementation of the *save* method.

Note: Go to serializers module and press ctrl+ shift + O. With this you can find all the symbols defined in this module.



Look at the save method, which is implemented in *BaseSerializer*.

   def save(self, \*\*kwargs):

        assert hasattr(self, '\_errors'), (

            'You must call `.is\_valid()` before calling `.save()`.'

        )

        assert not self.errors, (

            'You cannot call `.save()` on a serializer with invalid data.'

        )

        # Guard against incorrect use of `serializer.save(commit=False)`

        assert 'commit' not in kwargs, (

            "'commit' is not a valid keyword argument to the 'save()' method. "

            "If you need to access data before committing to the database then "

            "inspect 'serializer.validated\_data' instead. "

            "You can also pass additional keyword arguments to 'save()' if you "

            "need to set extra attributes on the saved model instance. "

            "For example: 'serializer.save(owner=request.user)'.'"

        )

        assert not hasattr(self, '\_data'), (

            "You cannot call `.save()` after accessing `serializer.data`."

            "If you need to access data before committing to the database then "

            "inspect 'serializer.validated\_data' instead. "

        )

        validated\_data = {\*\*self.validated\_data, \*\*kwargs}

        if self.instance is not None:

            self.instance = self.update(self.instance, validated\_data)

            assert self.instance is not None, (

                '`update()` did not return an object instance.'

            )

        else:

            self.instance = self.create(validated\_data)

            assert self.instance is not None, (

                '`create()` did not return an object instance.'

            )

        return self.instance

In the save method, first there are a bunch of assertions to validate the data, next we have our validated\_data dictionary. After that *we have a logic stating* ***if self.instance is not None*** *we will update the record otherwise we are going to create one., either way we are setting self.instance*.

***So the object that is updated or created should be returned from this save method and stored in self.instance attribute***.

So we have to follow the same pattern, so that each of these building blocks that talk to each other properly.

class AddCartItemSerializer(serializers.ModelSerializer):

    product\_id = serializers.IntegerField()

    def save(self, \*\*kwargs):

        cart\_id = self.context["cart\_id"]

        product\_id = self.validated\_data["product\_id"]

        quantity = self.validated\_data["quantity"]

        try:

            cart\_item = CartItem.objects.get(cart\_id=cart\_id, product\_id=product\_id)

            #Updating an existing item

            cart\_item.quantity += quantity

            cart\_item.save()

            self.instance = cart\_item

        except CartItem.DoesNotExist:

            #Creating a new item

            self.instance = CartItem.objects.create(

                cart\_id=cart\_id, \*\*self.validated\_data

            )

        return self.instance

    class Meta:

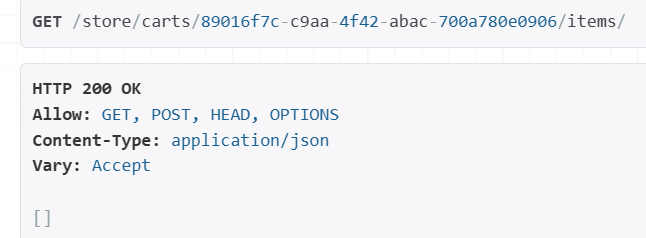
        model = CartItem

        fields = ["id", "product\_id", "quantity"]

So let us create a brand new cart and test our implementation in it.



Then go to items endpoint for this cart,



And *POST* this data,

{

"product\_id": 1,

"quantity": 10

}

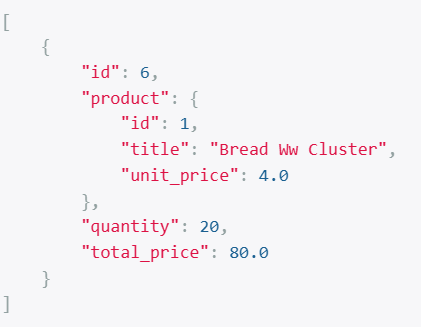


We get a new cart item in our cart.

Let us POST the same product\_id with the same quantity which is 10.



Now we see the same product with the updated quantity and we do not have duplicate records in our database.



We have only a single item with this quantity.

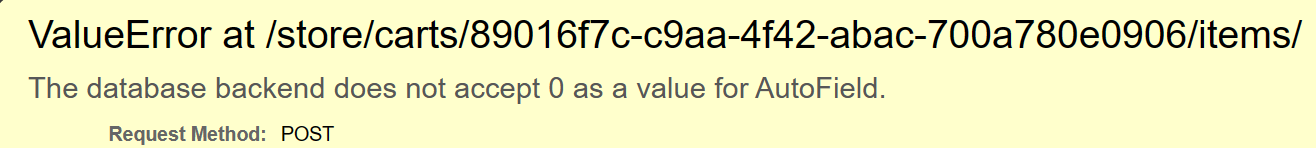
Since we have tested it with valid values and it works fine, now we will test it with invalid values (*like invalid product\_id*).

{

"product\_id": 0, 🡨 We do not have product with this id

"quantity": 1

}



Our application blows up, so we need to prevent this and return a meaningful error to the client.

Earlier in the course we talked about Data validation in serializers, *so we can either validate the entire object that we passed in our serializer or individual fields*.

To validate individual fields we have to follow a specific convention (*def validate\_fieldName*) like this,

 def validate\_product\_id(self, value):

// to validate product\_id field

Here we have two parameters, *self and the value we are validating*.

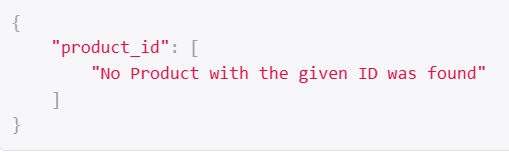
    def validate\_product\_id(self, value):

        if not Product.objects.filter(pk=value).exists():

            raise serializers.ValidationError("No Product with the given ID was found")

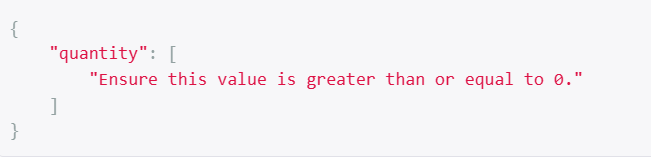
        return value

This is the same pattern we followed for validating the entire object. So we either raise a validation error or return the valid value.



Now we have a validation error associated with this field.

But what about second field ‘quantity’. Let us test with -1.



This error is actually coming from the definition of our CartItem Model.

class CartItem(models.Model):

    cart = models.ForeignKey(Cart, on\_delete=models.CASCADE, related\_name="items")

    product = models.ForeignKey(Product, on\_delete=models.CASCADE)

    quantity = models.PositiveSmallIntegerField()

    class Meta:

        unique\_together = [["cart", "product"]]

Here we have define *quantity* as *PositiveSmallIntegerField* that is why it does not accept negative values.

But if you want to take it to next level and ensure that quantity is always greater than zero then we can use *validators* here.

class CartItem(models.Model):

    cart = models.ForeignKey(Cart, on\_delete=models.CASCADE, related\_name="items")

    product = models.ForeignKey(Product, on\_delete=models.CASCADE)

    quantity = models.PositiveSmallIntegerField(

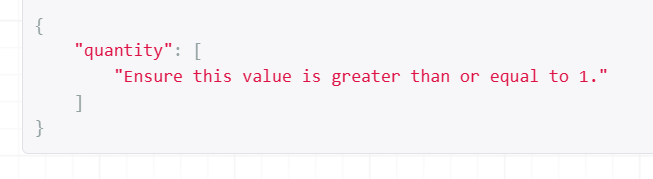
        validators=[MinValueValidator(1)] 🡨 Set min value to 1

    )

    class Meta:

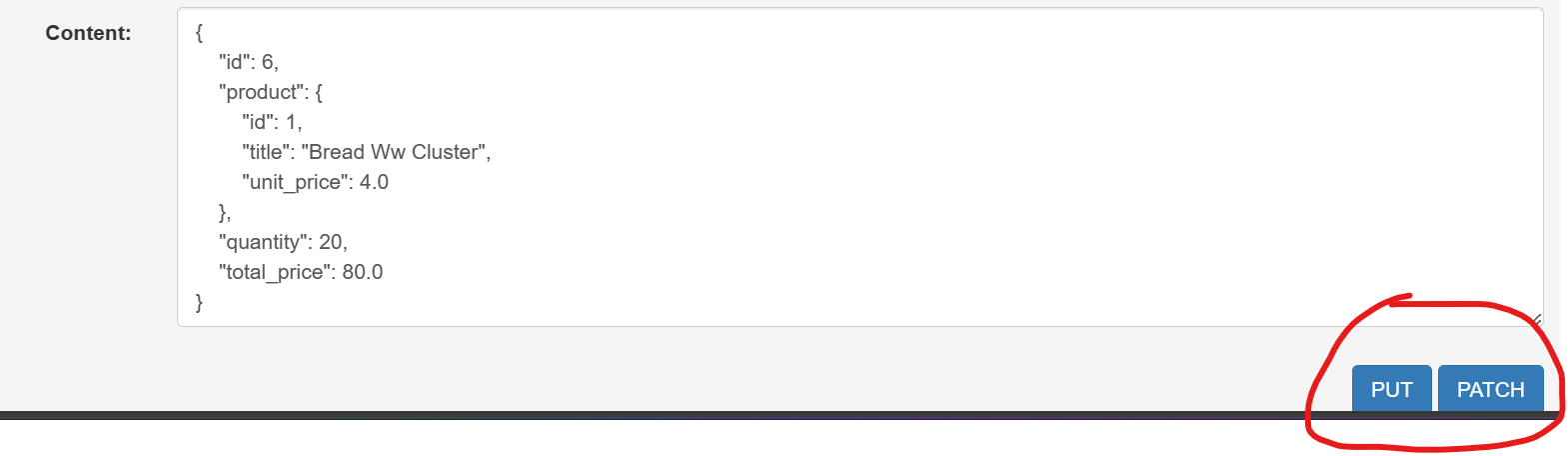
        unique\_together = [["cart", "product"]]

Now if we POST with quantity as zero, we get



**Updating a Cart Item**:

With every step our API is evolving and getting better. So now let us implement updating a cart item.



This is the object we currently pass to the server to update a cart item. But it does not make any sense, we just want to update the quantity.

So we will use the same technique as in previous lesson i.e. create a custom serializer called *UpdateCartItemSerializer* for updating our cart item.

class UpdateCartItemSerializer(serializers.ModelSerializer):

    class Meta:

        model = CartItem

        fields = ['quantity']

In our CartItemViewSet, we will add another *elif* statement for ‘PATCH’ request.

        elif self.request.method == "PATCH":

            return UpdateCartItemSerializer

We will not allow ‘PUT’ request in from our viewset, because we only want to update a single property of a cart item object.

We can avoid ‘PUT’ request in our viewset using an attribute called *http\_method\_names* and set it to list of methods we allow at this end point.

class CartItemViewSet(ModelViewSet):

    http\_method\_names = ["get", "post", "patch"]

    def get\_queryset(self):

        return CartItem.objects.select\_related("product").filter(

            cart\_id=self.kwargs["cart\_pk"]

        )

    def get\_serializer\_context(self):

        return {"cart\_id": self.kwargs["cart\_pk"]}

    def get\_serializer\_class(self):

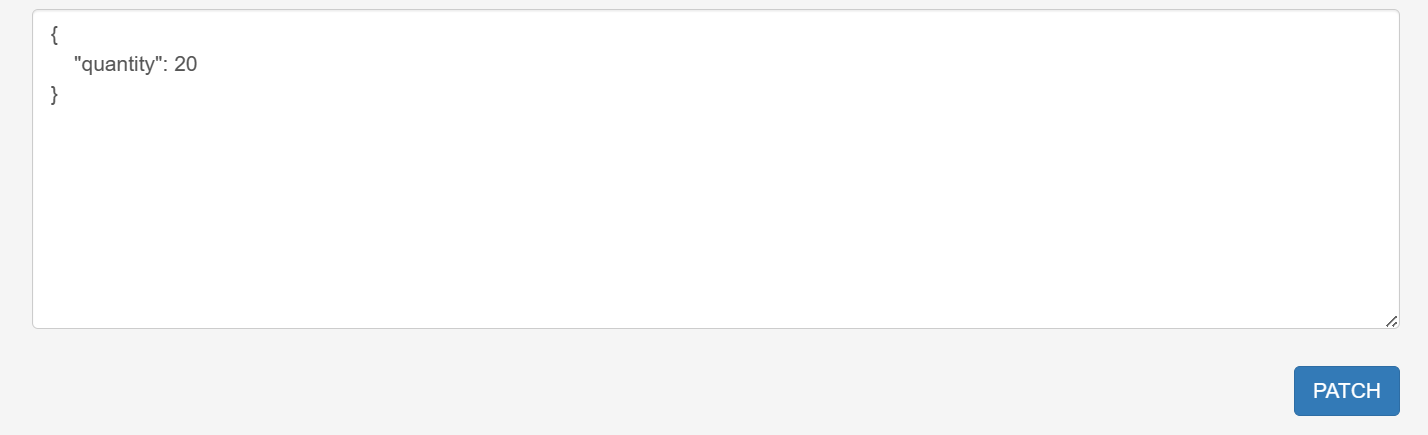
        if self.request.method == "POST":

            return AddCartItemSerializer

        elif self.request.method == "PATCH":

            return UpdateCartItemSerializer

        return CartItemSerializer



Now our PUT button has disappeared and we can only see quantity in our request object.

Let us update the quantity to 10 and our cart item is updated.



**Deleting a cart item**:

Now the final step, we want to delete a cart item. As you can see this functionality will be implemented for us if we just add ‘delete’ in the list of *http\_method\_names* and our CartItemViewSet extends ModelViewSet class.

class CartItemViewSet(ModelViewSet):

    http\_method\_names = ["get", "post", "patch", "delete"]

    def get\_queryset(self):

        return CartItem.objects.select\_related("product").filter(

            cart\_id=self.kwargs["cart\_pk"]

        )

    def get\_serializer\_context(self):

        return {"cart\_id": self.kwargs["cart\_pk"]}

    def get\_serializer\_class(self):

        if self.request.method == "POST":

            return AddCartItemSerializer

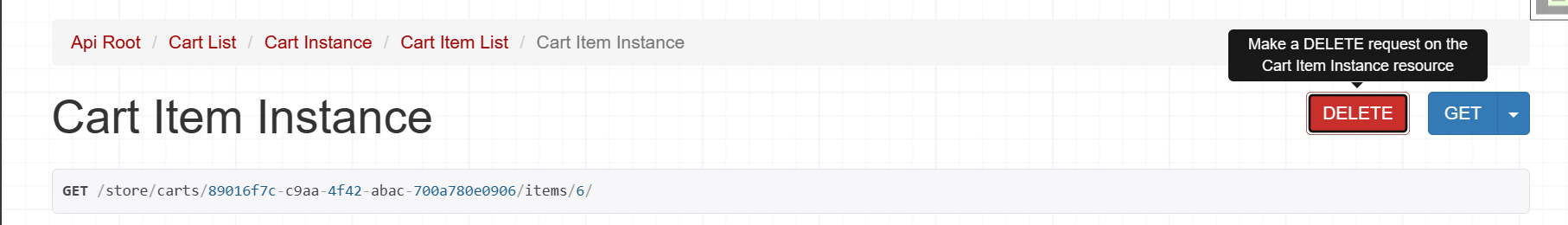
        elif self.request.method == "PATCH":

            return UpdateCartItemSerializer

        return CartItemSerializer

Note: All the methods allowed in list of http\_method\_names should be lowercase if we use ‘DELETE’ instead of ‘delete’, our method won’t be available anymore.

So just delete our cart item from delete button,



We get 204 no content,



And if we refresh,



We get 404 error and this resource is deleted from the server.