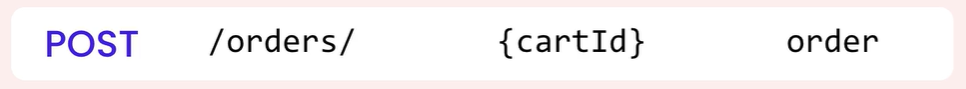


**Designing the API**:

In this section take our application to next level and build an orders API.

Let us spend a couple of minutes and plan out what we are going to build in this section. So *we are going to build a new endpoint for managing our orders*.

1. To **create an order**:



We simply send a POST request to /orders/ endpoint. All we need to include in the request body is the *cart ID* because *we can extract the current users ID from the JSON web token included in the request header*.

So that means this endpoint should only be open to authenticated users. Therefore creating cart id and user id we can create an order object and then return it to the client.

1. To **get an order**:



We should also support GET request at the same endpoint. But what the user sees depends on their permission.

If we are a regular user, we should be able to see our own orders. But if we are admin, we should be able to see all orders in the database.

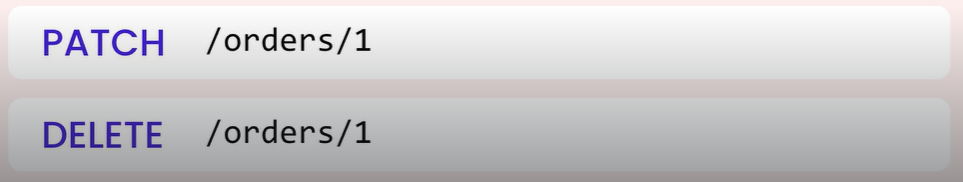
1. To **retrieve a specific order**:



Here we have the same policy. Only be able to retrieve our own order unless we are admin.

1. To **update or delete an order**:

Potentially we can also support PATCH and DELETE requests here for updating the status of an order or deleting it.

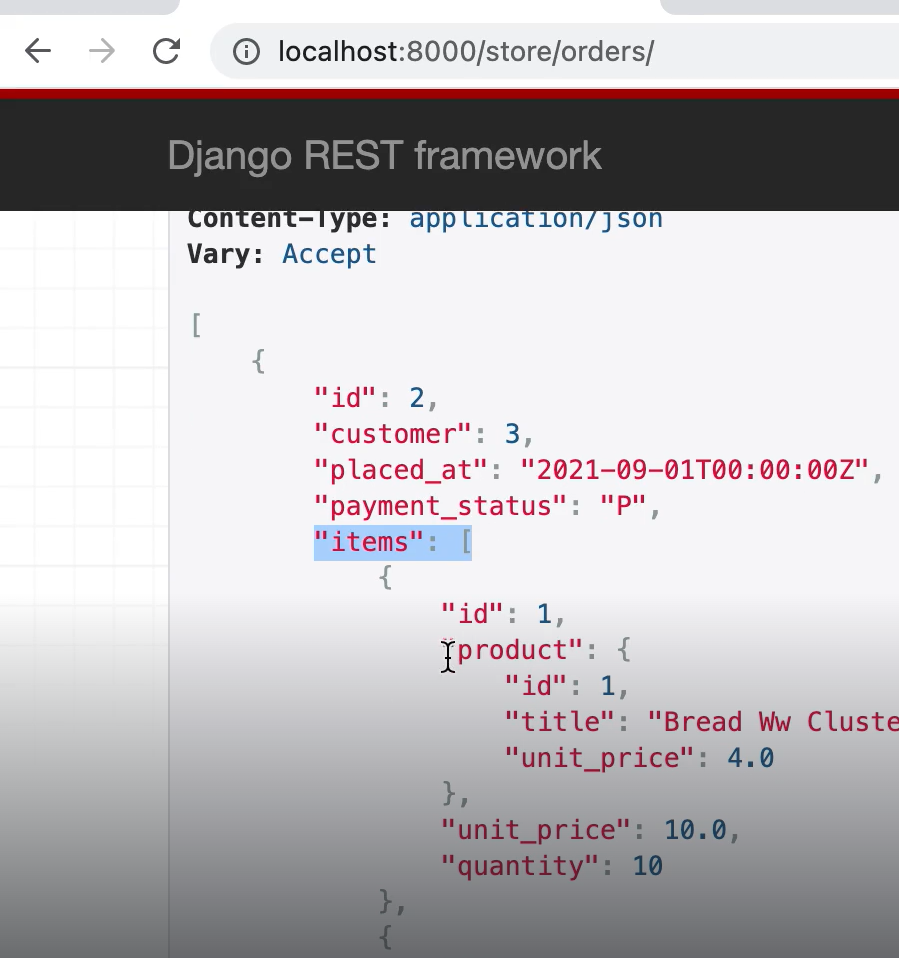


**Getting the Orders**:

First we are going to implement is getting an order because creating an order is a bit more complex. So we will start with something simple.

***Always Start simple, build a foundation and then gradually build things up step by step***.

We will create an orders/ endpoint and when we hit this end point we want to see all orders that we have in database (*At this point we do not care about permissions or any other aspect of this end point*).



After hitting this end point we see all attributes of the order, order items and each order item includes a nested product object.

So just like always we start with the serializer, we create a new class called OrderSerializer which extends *ModelSerializer*.

class OrderSerializer(serializers.ModelSerializer):

    class Meta:

        model = Order

        fields = [

            "id",

            "customer",

            "placed\_at",

            "payment\_status",

        ]

Here we create a Meta class and set model to Order and fields to list of fields we require.

Next we need a ViewSet,

class OrderViewSet(ModelViewSet):

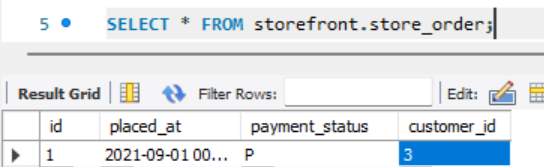
    queryset = Order.objects.all()

    serializer\_class = OrderSerializer

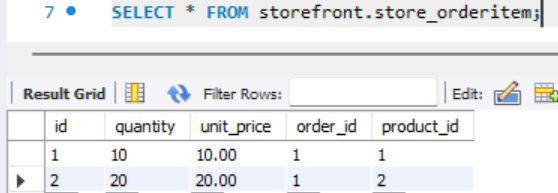
Now, we register an endpoint for managing our OrderViewSet in our urls module.

router.register("orders", views.OrderViewSet)

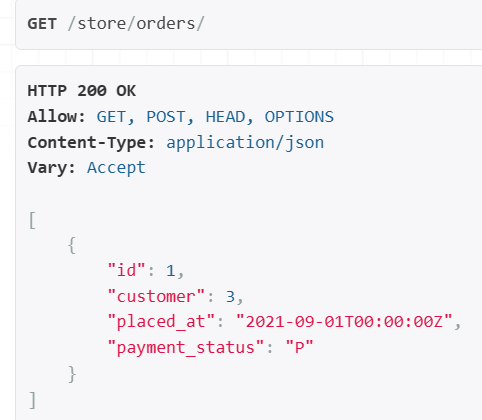
Let us manually create an order from our MySQL database.



And also some order items in order items table.



And we start to see order specific data in our orders/ endpoint



But one thing missing is our order items. So we will create a serializer for the same.

First look at OrderItem class to see which fields to include in our serializer,

class OrderItem(models.Model):

    order = models.ForeignKey(Order, on\_delete=models.PROTECT)

    product = models.ForeignKey(

        Product, on\_delete=models.PROTECT, related\_name="orderitems"

    )

    quantity = models.PositiveSmallIntegerField()

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

In our OrderItemSerializer,

class OrderItemSerializer(serializers.ModelSerializer):

    class Meta:

        model = OrderItem

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

We are not including order field here because we are going to use this serializer inside our OrderSerializer.

Now in our OrderSerializer, we add a field called items and set it to an instance of OrderItemSerializer class (*with many = True*)

class OrderSerializer(serializers.ModelSerializer):

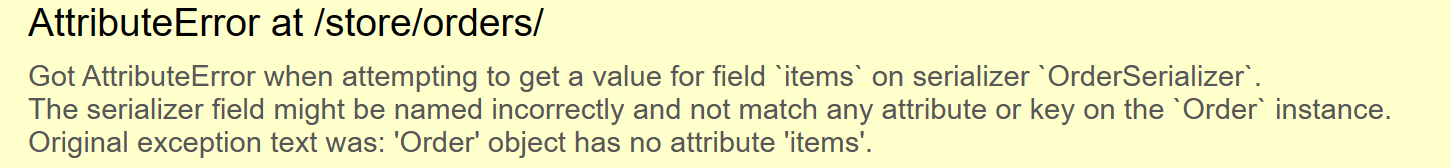
    items = OrderItemSerializer(many=True)

    class Meta:

        model = Order

        fields = ["id", "customer", "placed\_at", "payment\_status", "items"]

And refresh the page,



We get an attribute error 'Order' object has no attribute 'items'

So we set the *related\_name* attribute inside OrderItem model where we have foreign key to the Order model,

class OrderItem(models.Model):

    order = models.ForeignKey(Order, on\_delete=models.PROTECT, related\_name="items") 🡪 *Add this Here*

    product = models.ForeignKey(

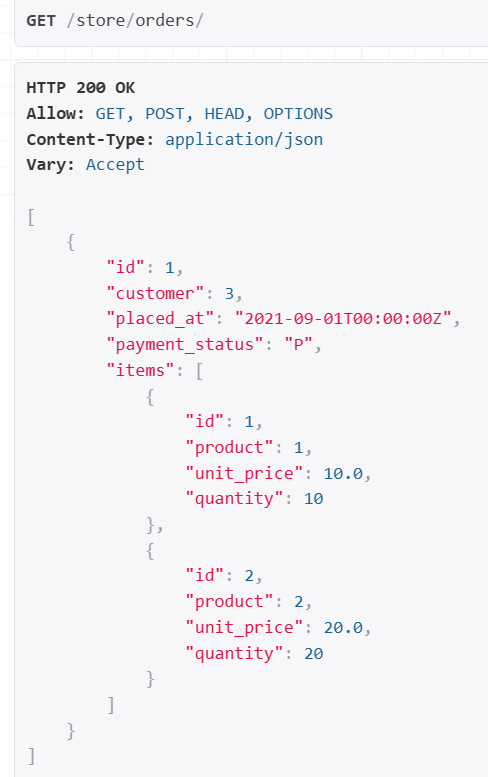
        Product, on\_delete=models.PROTECT, related\_name="orderitems"

    )

    quantity = models.PositiveSmallIntegerField()

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

Now refresh the page,

🡨 We get our order items.

But let us change product from product id to a nested product object. *This way we can return all critical information about each product, so the client does not have to send additional requests to for each product in the order*.

We set product to SimpleProductSerializer.

class OrderItemSerializer(serializers.ModelSerializer):

    product = SimpleProductSerializer() 🡪 *Define it here*

    class Meta:

        model = OrderItem

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

Remember we defined this serializer earlier,

class SimpleProductSerializer(serializers.ModelSerializer):

    class Meta:

        model = Product

        fields = ["id", "title", "unit\_price"] 🡪*only critical product info*

Refresh and we see each product is a nested object,



Note: I eager loaded product table, not told by Mosh.

class OrderViewSet(ModelViewSet):

    queryset = Order.objects.prefetch\_related("items\_\_product").all() 🡪 *here*

    serializer\_class = OrderSerializer

**Applying Permissions**:

Our orders endpoint is currently open to everyone including anonymous users which is not good. So first thing we need to do is to apply a permission class.

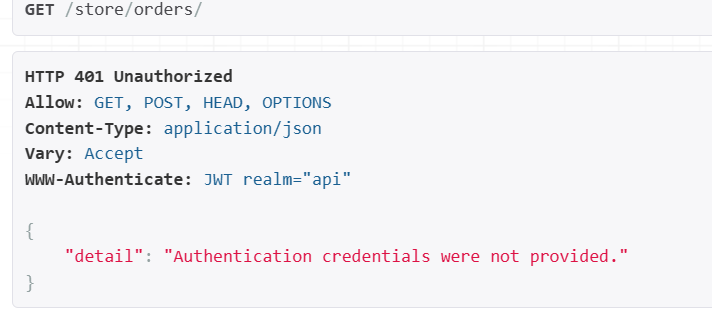
class OrderViewSet(ModelViewSet):

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

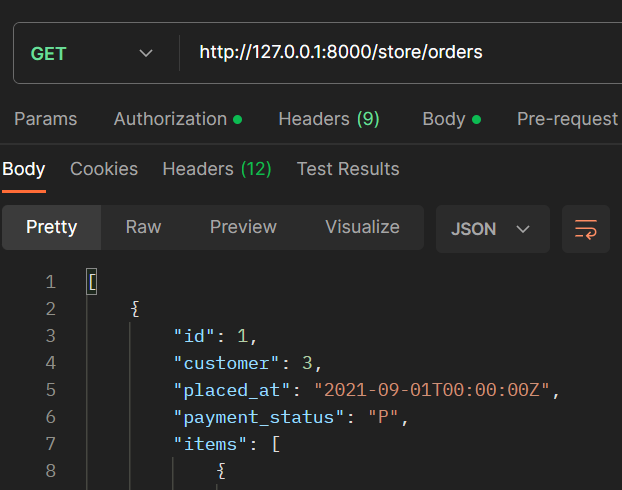
    permission\_classes = [IsAuthenticated] 🡪 *Here*

    serializer\_class = OrderSerializer

Now if we go to this endpoint,

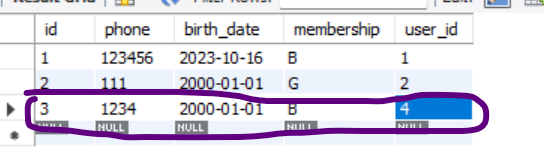


We will login as john.smith now in our Postman with access token,



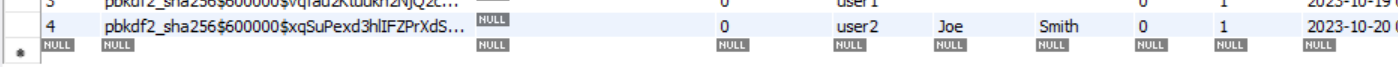
We see the orders.

But notice the customer id = 3. This is not john.smith it is some other customer.



Customer id = 3 is associated with user id = 4.

In the user table,



User id = 4 belongs to user2 which is *joe smith*. A completely different account.

In this case john.smith is an admin user, so he should be able to see all orders in the database. But if john is not an admin, he should only see his own orders.

Currently we have not implemented this rule.

Back to our viewset,

class OrderViewSet(ModelViewSet):

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

    permission\_classes = [IsAuthenticated]

    serializer\_class = OrderSerializer

Instead of using the queryset attribute directly we need to override *get\_queryset* method.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    serializer\_class = OrderSerializer

    def get\_queryset(self): 🡪 *here*

        return super().get\_queryset()

Here we say, if user is staff (*access to admin panel*) then he can see all records.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    serializer\_class = OrderSerializer

    def get\_queryset(self):

        if self.request.user.is\_staff: 🡪 *Here*

            return Order.objects.prefetch\_related("items\_\_product").all()

Otherwise, we will apply a filter and only retrieve orders of a specific customer. But *here we need to find customer id of the current user because customer\_id is not included in JWT*.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    serializer\_class = OrderSerializer

    def get\_queryset(self):

        if self.request.user.is\_staff:

            return Order.objects.prefetch\_related("items\_\_product").all()

        return Order.objects.filter(customer\_id= *??? How to get customer\_id ??*)

So *using the customer model, we will get user\_id which is self.request.user.id*.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    serializer\_class = OrderSerializer

    def get\_queryset(self):

        if self.request.user.is\_staff:

            return Order.objects.prefetch\_related("items\_\_product").all()

        customer\_id = Customer.objects.get(user\_id=self.request.user.id) 🡪 *Here*

        return Order.objects.filter(customer\_id=customer\_id) 🡪 *now we get customer\_id*

Note: Here we are going to retrieve a complete customer object (*get method returns object*)but all we need is customer\_id so we can apply a bit of premature optimization here and *only* pick the ‘*id*’ field.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    serializer\_class = OrderSerializer

    def get\_queryset(self):

        if self.request.user.is\_staff:

           return Order.objects.prefetch\_related("items\_\_product").all()

        customer\_id = Customer.objects.only("id").get(user\_id=self.request.user.id) 🡪 *used only*

        return Order.objects.filter(customer\_id=customer\_id)

In this code we have a couple of references to *self.request.user*, so we can simplify this code a little bit.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    serializer\_class = OrderSerializer

    def get\_queryset(self):

        user = self.request.user 🡪 *set it to user*

        if user.is\_staff: 🡪 *Then apply where needed to reference*

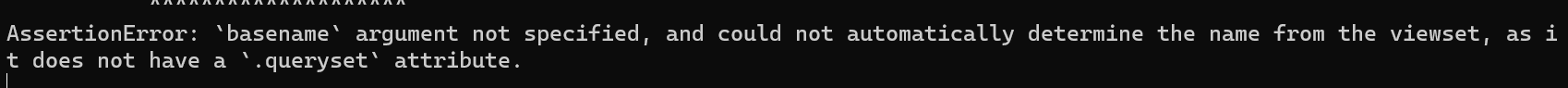
           return Order.objects.prefetch\_related("items\_\_product").all()

        customer\_id = Customer.objects.only("id").get(user\_id=user.id)

        return Order.objects.filter(customer\_id=customer\_id)

Let us see our changes,

But our server stopped working now and we get an error.



AssertionError: `basename` argument not specified, and could not automatically determine the name from the viewset, as it does not have a `.queryset` attribute.

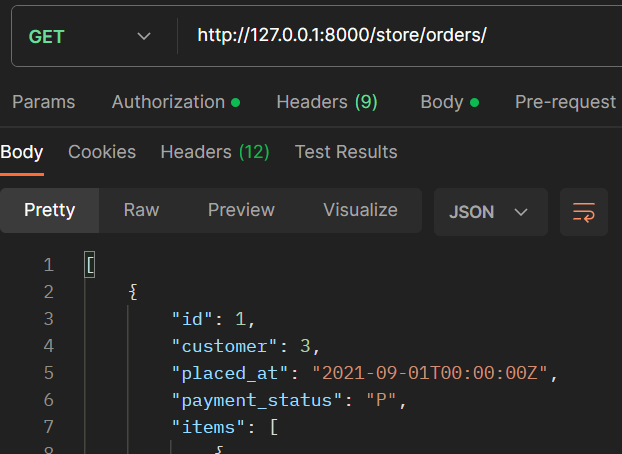
This error is coming because we removed the queryset attribute from our viewset and now we are overriding the get\_queryset method. Therefore DRF cannot figure out the basename for our endpoint.

router.register("orders", views.OrderViewSet, basename="orders")

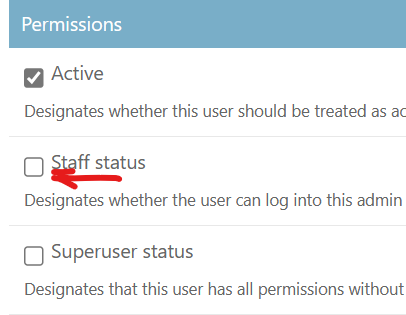
In the URLs module we added basename for our orders endpoint (*which creates* ***orders-list*** *and* ***orders-detail*** *views*).

Note: Sometimes need to restart webserver after fixing this error so that changes are picked up.

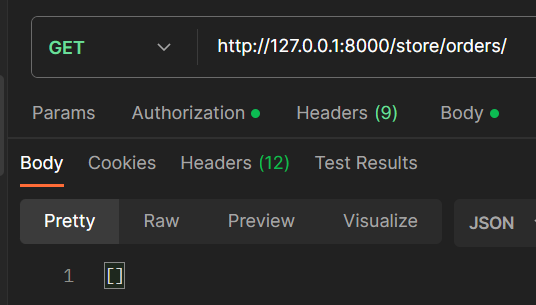
Our endpoint is working properly as before and as john.smith (*admin*) we can see all orders.



So let us remove the admin permission for john.smith from admin panel.

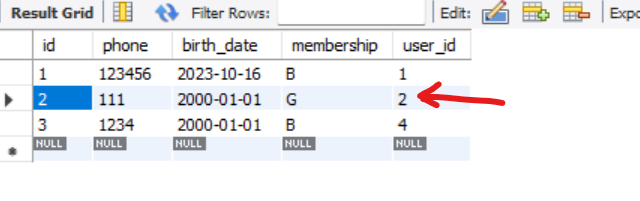
 and save.

Now we see empty array in orders endpoint as john.smith



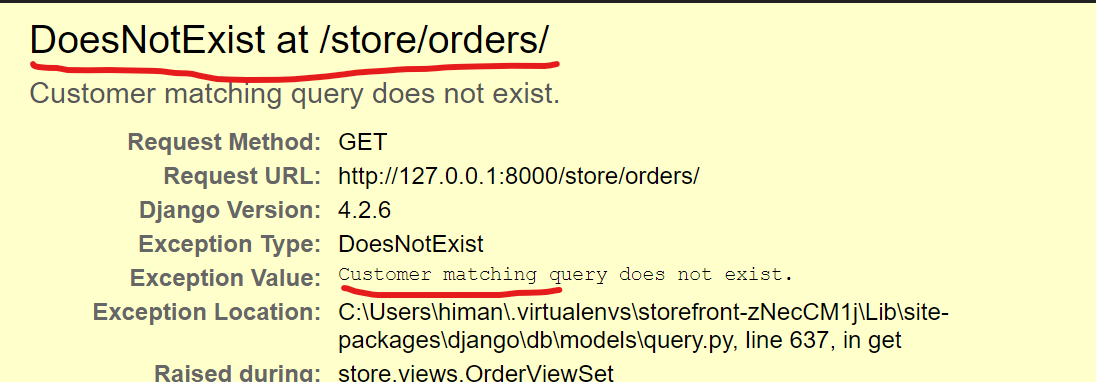
There is a tiny issue in our implementation. *If the current user does not have a customer record or a profile the get method will throw an exception*.

Let us see this in action, go to database, customer table and delete the record for john.smith.





After deleting this record if we send the GET request to orders/ endpoint again with john.smith user access token we get an exception.



It is because *get method always expects at least one record in the database. If we have zero records or more than one record matching our query criteria we are going to get an exception*.

So we will change get to *get\_or\_create*. *This method returns a tuple with two values. First value is the object we are reading and second is a Boolean that indicates if a record was created or not*.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    serializer\_class = OrderSerializer

    def get\_queryset(self):

        user = self.request.user

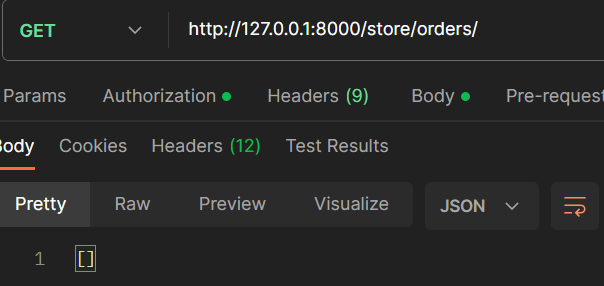
        if user.is\_staff:

            return Order.objects.prefetch\_related("items\_\_product").all()

        (customer\_id, created) = Customer.objects.only("id").get\_or\_create(user\_id=user.id) 🡪 *get\_or\_create*

        return Order.objects.filter(customer\_id=customer\_id)

Now we are not getting any exception,



Even though using this method we solved this problem, but here we are violating an important principle in programming, which is…

****

***“****It means our methods or functions should either be commands and change the state of the system or they should be queries which means they should only return data and should not change the state or data in the system***”**

Here our *get\_queryset* method is purely for getting data.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    serializer\_class = OrderSerializer

    def get\_queryset(self):

        user = self.request.user

        if user.is\_staff:

            return Order.objects.prefetch\_related("items\_\_product").all()

        (customer\_id, created) = Customer.objects.only("id").get\_or\_create(

            user\_id=user.id

        )

        return Order.objects.filter(customer\_id=customer\_id)

But while getting data we are also changing the state of the system while using *get\_or\_****create*** method.

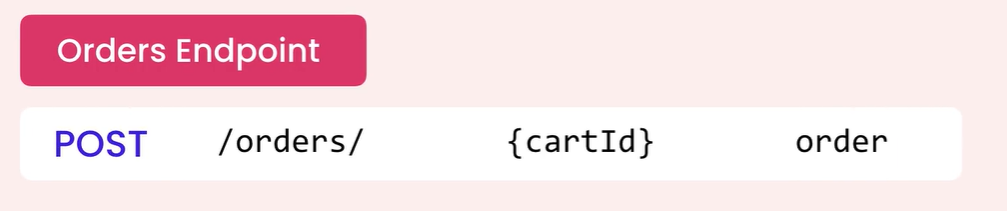
So we hit this orders/ endpoint for reading orders of a customer, we will end up creating a customer record. So this is a violation of ***command query separation*** principle.

When we violate this principle, our software becomes unpredictable and we get weird side effects. So later in this section we will learn better techniques for solving this problem.

Next we will create an order.

**Creating an Order**:

Earlier we have seen that when creating an order, all we need to send to the server is cart ID.



So we cannot use the *OrderSerializer*,

class OrderSerializer(serializers.ModelSerializer):

    items = OrderItemSerializer(many=True)

    class Meta:

        model = Order

        fields = ["id", "customer", "placed\_at", "payment\_status", "items"]

because the object that we are representing here (*look at the fields*) has completely different shape from the object we need to send to the server.

So we need a new serializer in our serializers module.

Here we are not going to use *ModelSerializer* class because we are not going to use a Meta class that is based on the Order model like this.

class CreateOrderSerializer(serializers.ModelSerializer):

    class Meta:

        model = Order

Because we want *cart\_id* which is not a field defined in Order model.

So instead of using ModelSerializer, we are going to use the *base Serializer* and here we are going to explicitly define cart\_id as a UUID field.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

Now in this serializer we need to override the *save* method because the logic for saving an order is very specific, it is not something we want Django to generate for us.

So *we have to go to the shopping cart table, grab all cart items, move them to order items table and then delete the shopping cart*.

For now we are just printing user\_id and cart\_id on the terminal.

For *cart\_id* we will get it from the list of validated data.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        print(self.validated\_data['cart\_id']) 🡪 *cart\_id is inside validated\_data*

What about *user\_id*? Here we do not have access to *request* object because we are inside a serializer. So *we have to go to our viewset and using a context object pass the user ID here*.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    serializer\_class = OrderSerializer

    def get\_serializer\_context(self): 🡪 *Here we override get\_serializer\_context*

        return {"user\_id": self.request.user.id}

Now back to our *save* method,

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        print(self.validated\_data["cart\_id"])

        print(self.context["user\_id"]) 🡪 *get user\_id from context*

Next step is to use this serializer in our viewset, so instead of hardcoding OrderSerializer as the default serializer in our viewset, we are going to override *get\_serializer\_class* method.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    serializer\_class = OrderSerializer

    def get\_serializer\_class(self): 🡪 *here*

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

            return CreateOrderSerializer

        return OrderSerializer

    def get\_serializer\_context(self):

        return {"user\_id": self.request.user.id}

    def get\_queryset(self):

        user = self.request.user

        if user.is\_staff:

            return Order.objects.prefetch\_related("items\_\_product").all()

        (customer\_id, created) = Customer.objects.only("id").get\_or\_create(

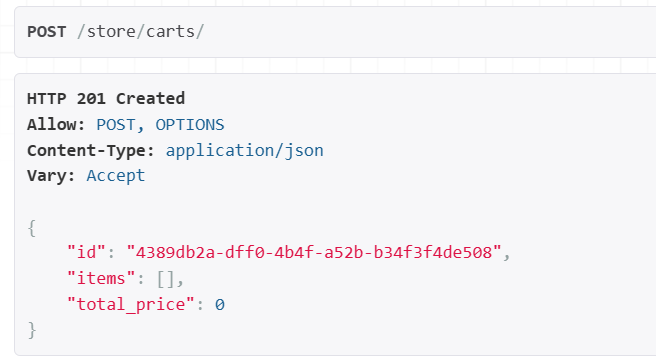
            user\_id=user.id

        )

        return Order.objects.filter(customer\_id=customer\_id)

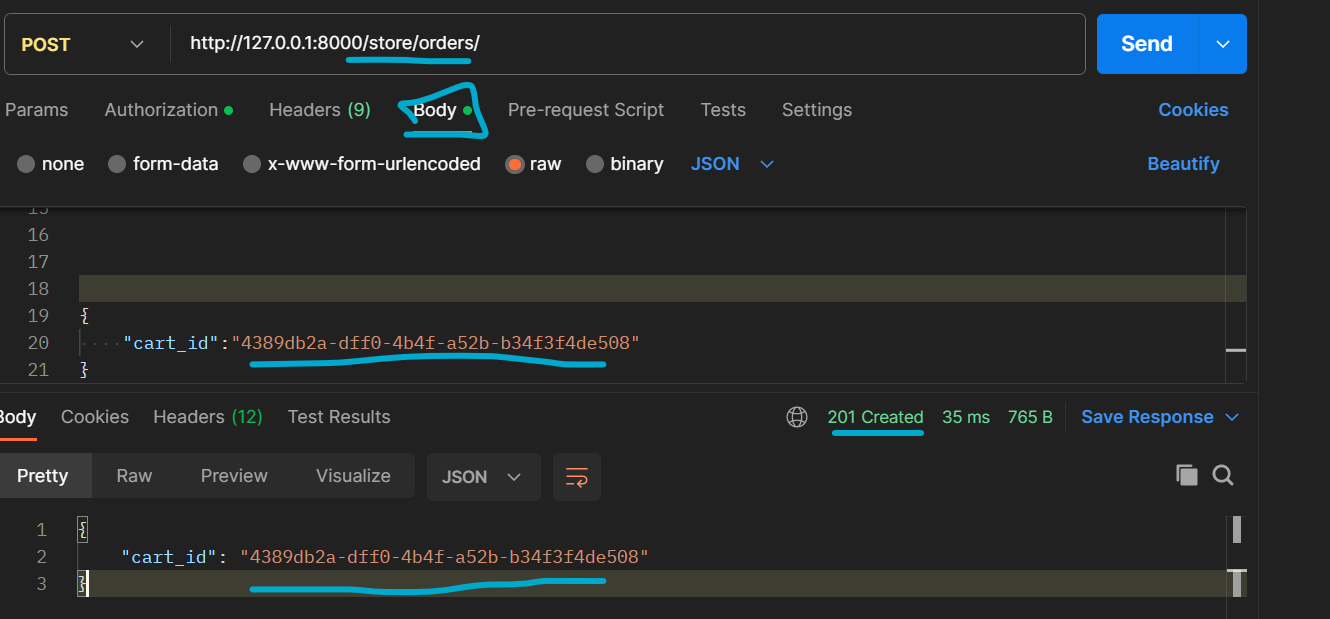
Here we are saying if our request method is POST we return *CreateOrderSerializer* otherwise we return OrderSerializer.

First let us create a cart for our user john.smith by going to carts endpoint and creating a new cart using POST request.



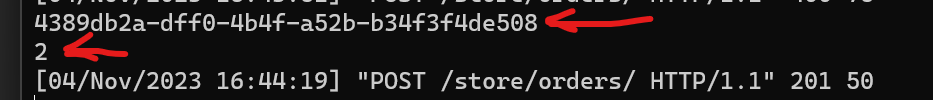
We copy the cart\_id 4389db2a-dff0-4b4f-a52b-b34f3f4de508

And at the orders endpoint, let us send this *cart\_id* to the server in the body of the request.



We got this cart\_id field back because it’s a field in our CreateOrderSerializer.

In the terminal,



We have the cart\_id and the user\_id. So with these two pieces of information we can create an order object.

Back to our serializer, we will create an Order record using Order.objects.*create* from the information we have in serializer and Order model.

class Order(models.Model):

    PAYMENT\_STATUS\_PENDING = "P"

    PAYMENT\_STATUS\_COMPLETE = "C"

    PAYMENT\_STATUS\_FAILED = "F"

    PAYMENT\_STATUS\_CHOICES = [

        (PAYMENT\_STATUS\_PENDING, "Pending"),

        (PAYMENT\_STATUS\_COMPLETE, "Complete"),

        (PAYMENT\_STATUS\_FAILED, "Failed"),

    ]

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

    payment\_status = models.CharField(

        max\_length=1, choices=PAYMENT\_STATUS\_CHOICES, default=PAYMENT\_STATUS\_PENDING

    )

    customer = models.ForeignKey(Customer, on\_delete=models.PROTECT)

    class Meta:

        permissions = [("cancel\_order", "Can cancel order")]

placed\_at field is added automatically and payment\_status field has a default value, so only field that we need to set is customer.

So we need to *get* customer object for the current user in our serializer,

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        print(self.validated\_data["cart\_id"])

        print(self.context["user\_id"])

        Customer.objects.get\_or\_create(user\_id = self.context["user\_id"])

Note: Getting customer object inside save method is not a violation of command query separation principle because with the save method we are changing the state of the system. So this *get\_or\_create* method is acting as a command not a query.

The violation was in other scenario where we had a query method and using that we changed the state of the system.

Now we get the customer object and using that object we create a new order (*Since this method returns a tuple, so let us unpack it*),

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        print(self.validated\_data["cart\_id"])

        print(self.context["user\_id"])

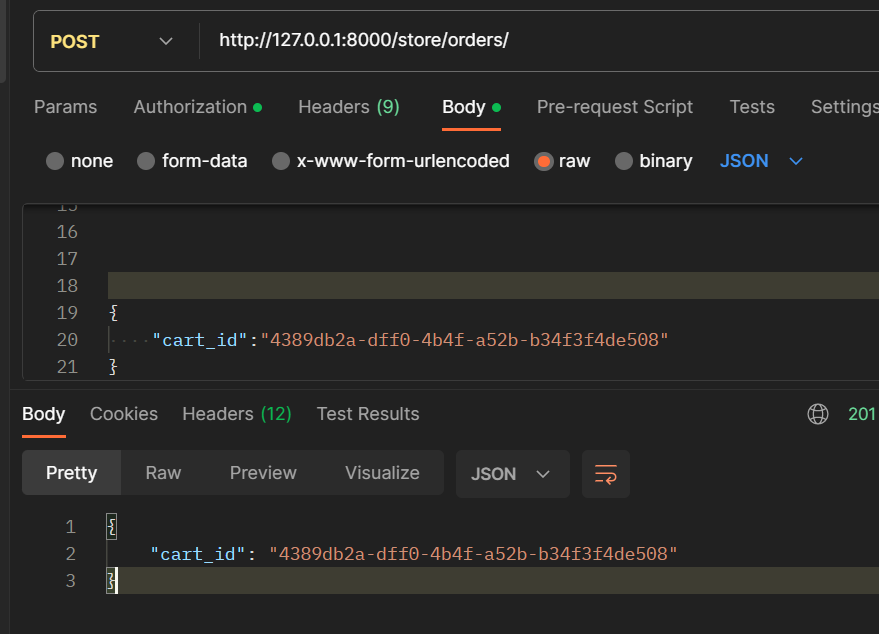
        (customer, created) = Customer.objects.get\_or\_create(

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

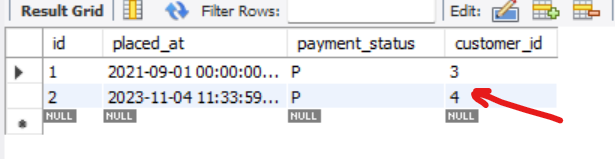
        )

        Order.objects.create(customer=customer)

Let us test our implementation up to this point, so POST our /store/orders/ endpoint,



No errors encountered and let us also check our database under order table,



We have a new order for customer\_id = 4 john.smith.

Now next step is storing order items which we will do next.

**Creating Order items**:

For creating order items, first we need to get items in the

cart( *cart with cart\_id*) and then for each cart item, we need to create an order item and save it in the database.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        (customer, created) = Customer.objects.get\_or\_create(

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

        )

        Order.objects.create(customer=customer)

        cart\_items = 🡪 *Here* CartItem.objects.filter(cart\_id=self.validated\_data["cart\_id"])

The *cart\_items* that we get from querying and filtering CartItem table *is more accurately a queryset but when we iterate this queryset we get a collection*.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        (customer, created) = Customer.objects.get\_or\_create(

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

        )

       Order.objects.create(customer=customer)

        cart\_items = CartItem.objects.filter(cart\_id=self.validated\_data["cart\_id"])

[OrderItem() for item in cart\_items] 🡪 *List comprehension*

To convert cart\_items into OrderItem() we are using list comprehension.

Here we use keyword arguments to initialize this OrderItem object.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        (customer, created) = Customer.objects.get\_or\_create(

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

        )

        order = Order.objects.create(customer=customer) 🡪 *get order object*

        cart\_items = CartItem.objects.filter(cart\_id=self.validated\_data["cart\_id"])

        [OrderItem(order=order, product=item.product) for item in cart\_items]

In OrderItem, we set order to the order object we just created and set product to item.product. This means *when retrieving these cart items we need to eager load them with their product, otherwise for each cart item we are going to sending an extra query to read the product of that item*.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        (customer, created) = Customer.objects.get\_or\_create(

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

        )

        order = Order.objects.create(customer=customer)

        cart\_items = CartItem.objects.select\_related("product").filter(

            cart\_id=self.validated\_data["cart\_id"] 🡪 *use select\_related*

        )

        [OrderItem(order=order, product=item.product) for item in cart\_items]

Next we set the unit price of the product at the time of placing the order and quantity as well. As a result we get a list of order\_items.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        (customer, created) = Customer.objects.get\_or\_create(

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

        )

        order = Order.objects.create(customer=customer)

        cart\_items = CartItem.objects.select\_related("product").filter(

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

        )

        order\_items = [ 🡪 *list of order\_items*

            OrderItem(

                order=order,

                product=item.product,

                unit\_price=item.product.unit\_price,

                quantity=item.quantity,

            )

            for item in cart\_items

        ]

Now we need to save these order\_items. *We are not going to iterate over them and save them all individually because that would be too many queries to the database.*

*So we are going to insert them in bulk* and for that we have a method called *bulk\_create*.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        (customer, created) = Customer.objects.get\_or\_create(

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

        )

        order = Order.objects.create(customer=customer)

        cart\_items = CartItem.objects.select\_related("product").filter(

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

        )

        order\_items = [

            OrderItem(

                order=order,

                product=item.product,

                unit\_price=item.product.unit\_price,

                quantity=item.quantity,

            )

            for item in cart\_items

        ]

        OrderItem.objects.bulk\_create(order\_items)

Here is *bulk\_create* we pass our order\_items for bulk creation.

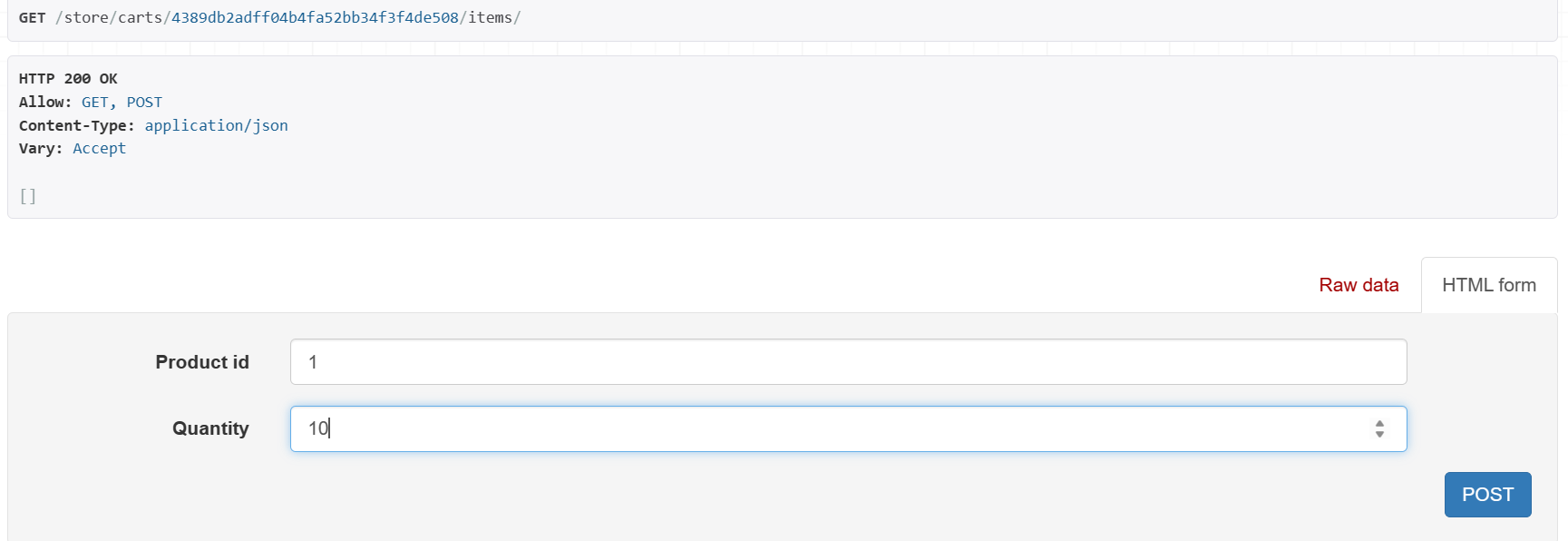
Let’s test our implementation.

We have a empty cart in cart table,

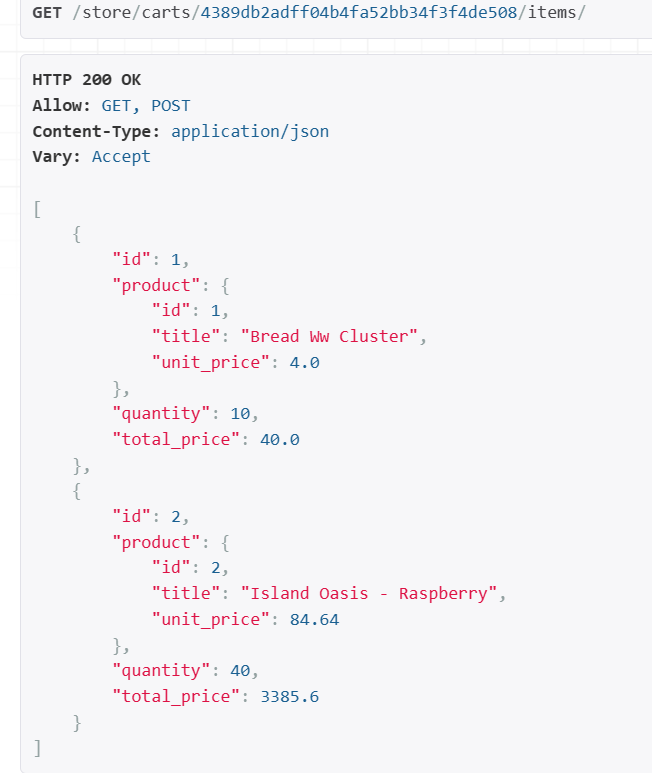


Let’s use this cart’s id and go to its /items endpoint,

**GET** /store/carts/4389db2adff04b4fa52bb34f3f4de508/items/

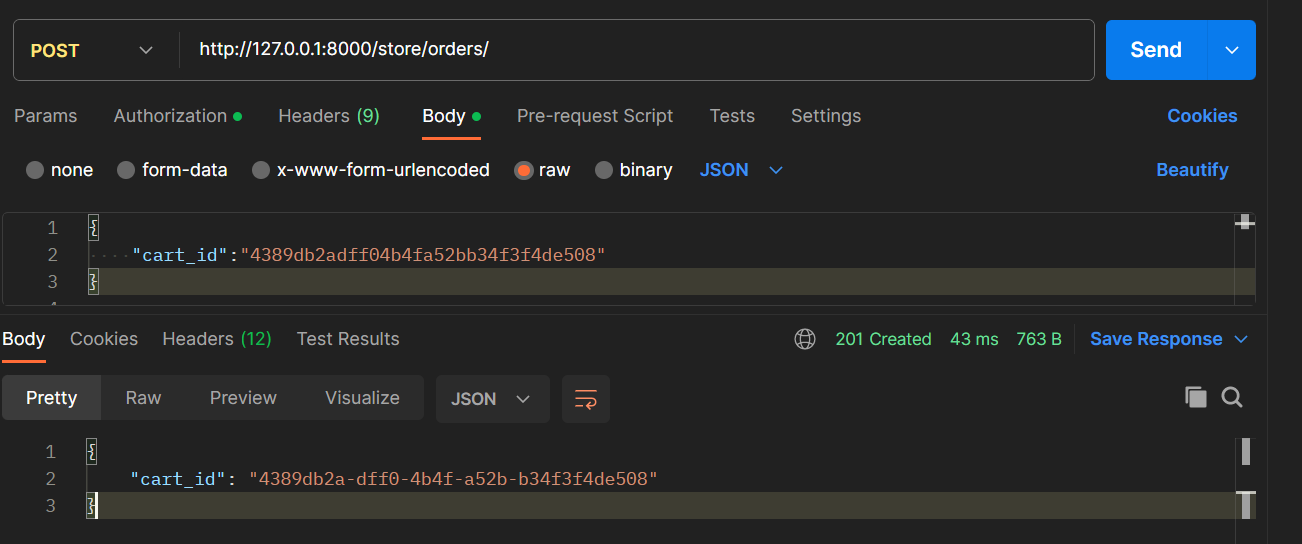


Now add some products to this cart,



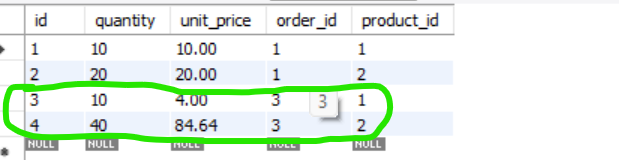
As a result we have two items in our shopping cart with id 4389db2adff04b4fa52bb34f3f4de508

Now we go to our orders endpoint and give it this cart id to create a new order.



We get 201 created.

In database under the orderitem table, we can see the same items (*due to bulk create*),



So we store the orderitem, finally we need to delete the shopping cart.

Cart.objects.filter(pk=self.validated\_data[‘cart\_id’]).delete()

Once again we need to extract *cart\_id* from validated\_data dictionary. So let us separate this variable and store it in a cart\_id, so we can reference it multiple times.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        cart\_id = self.validated\_data['cart\_id'] 🡪 *Here we separate it…*

        (customer, created) = Customer.objects.get\_or\_create(

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

        )

        order = Order.objects.create(customer=customer)

        cart\_items = CartItem.objects.select\_related("product").filter(

            cart\_id=cart\_id 🡪 *update our reference here*

        )

        order\_items = [

            OrderItem(

                order=order,

                product=item.product,

                unit\_price=item.product.unit\_price,

                quantity=item.quantity,

            )

            for item in cart\_items

        ]

        OrderItem.objects.bulk\_create(order\_items)

        Cart.objects.filter(pk=cart\_id).delete() 🡪 *Then delete the cart*

Here after filtering the cart as per the cart\_id we delete the cart.