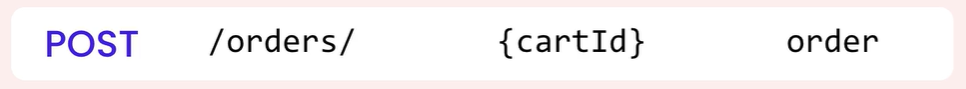


**Designing the API**:

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 using 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.

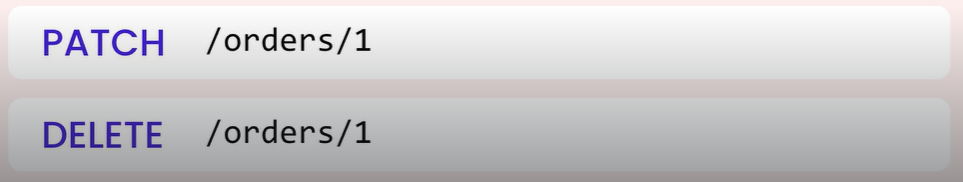
1. To **retrieve a specific order**:



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.

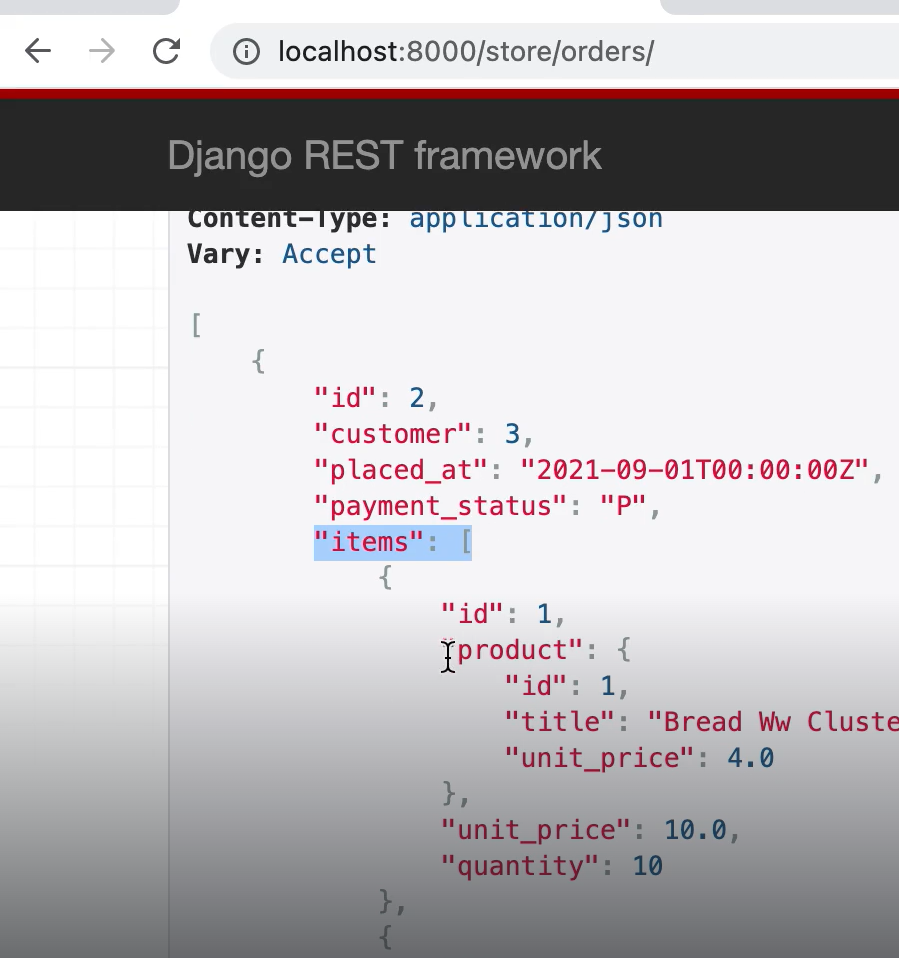


**Getting the Orders**:

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):

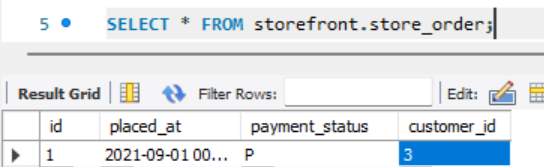
    queryset = Order.objects.all()

    serializer\_class = OrderSerializer

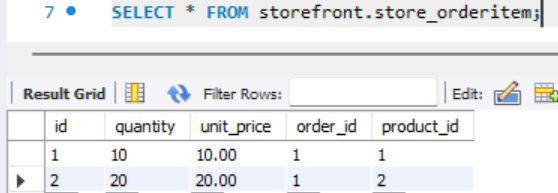
Now, we register an endpoint for managing our OrderViewSet in our urls module.

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

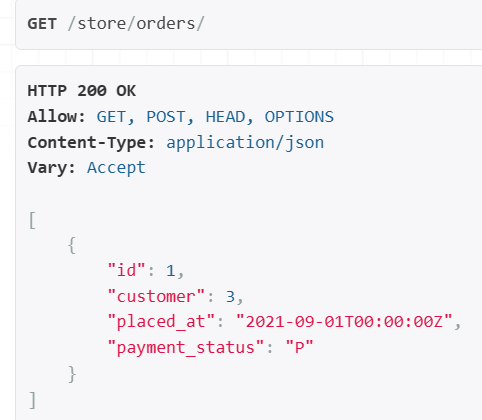
Let us manually create an order from our MySQL database.



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):

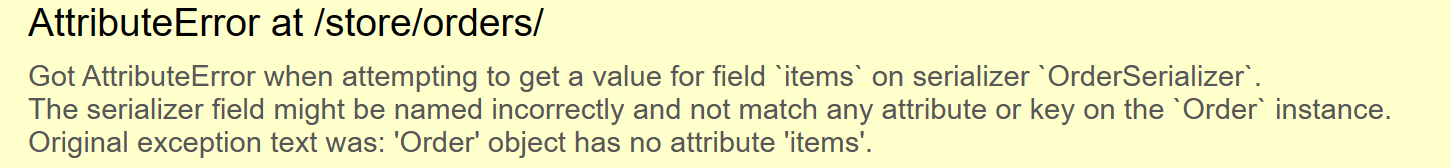
    items = OrderItemSerializer(many=True)

    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(

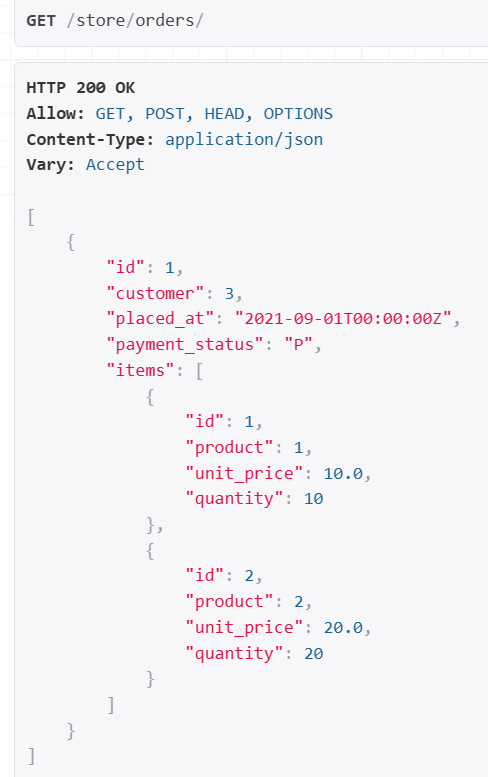
        Product, on\_delete=models.PROTECT, related\_name="orderitems"

    )

    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.

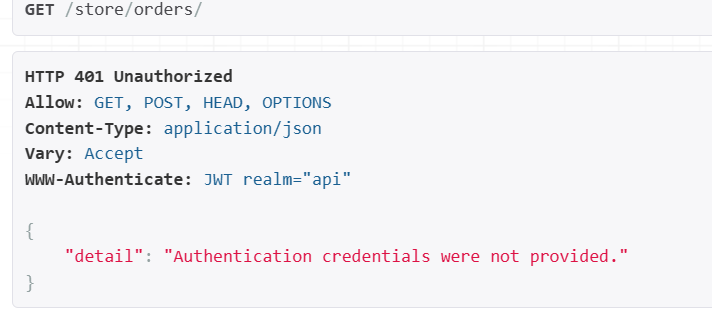
class OrderViewSet(ModelViewSet):

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

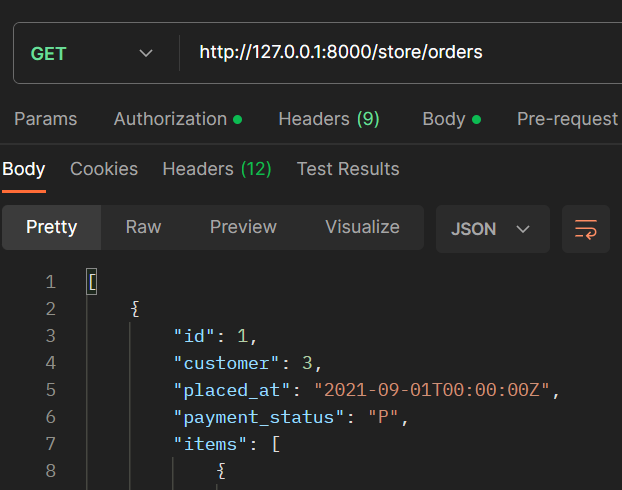
    permission\_classes = [IsAuthenticated] 🡪 *Here*

    serializer\_class = OrderSerializer

Now if we go to this endpoint,

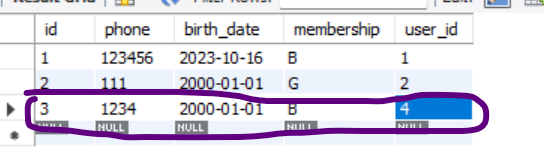


We will login as john.smith now in our Postman with access token,



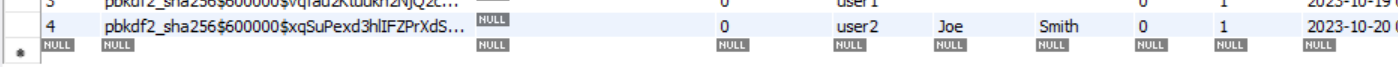
We see the orders.

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*

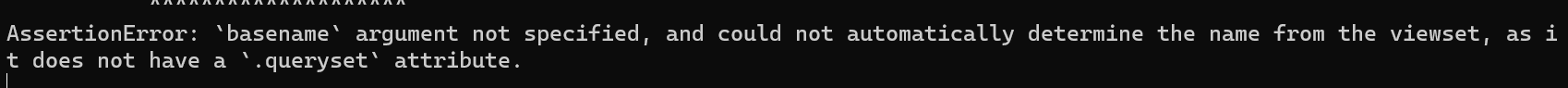
           return Order.objects.prefetch\_related("items\_\_product").all()

        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.

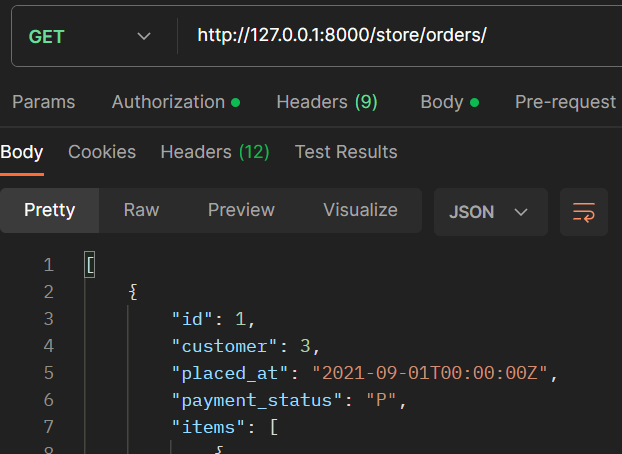
This error is coming because we removed the queryset attribute from our viewset and now we are overriding the get\_queryset method. Therefore DRF cannot figure out the basename for our endpoint.

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

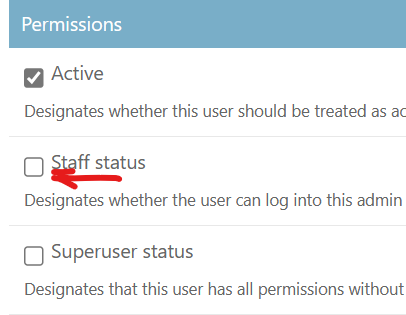
In the URLs module we added basename for our orders endpoint (*which creates* ***orders-list*** *and* ***orders-detail*** *views*).

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

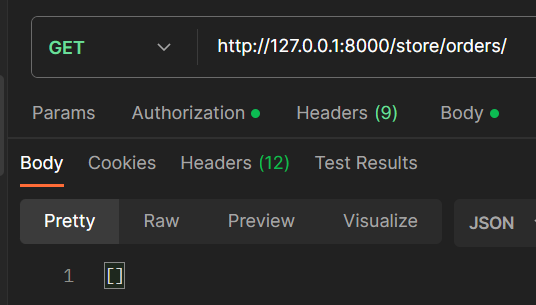
Our endpoint is working properly as before and as john.smith (*admin*) we can see all orders.



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

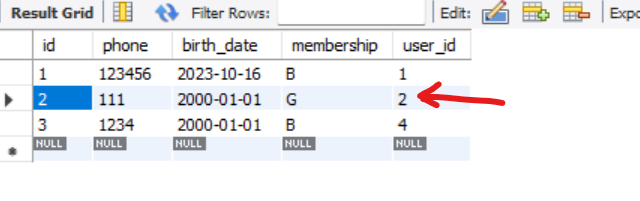
 and save.

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



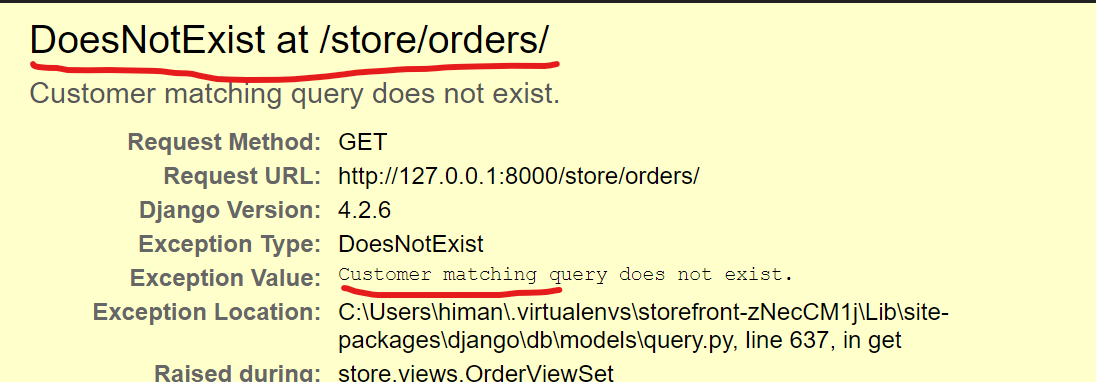
There is a tiny issue in our implementation. *If the current user does not have a customer record or a profile the get method will throw an exception*.

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

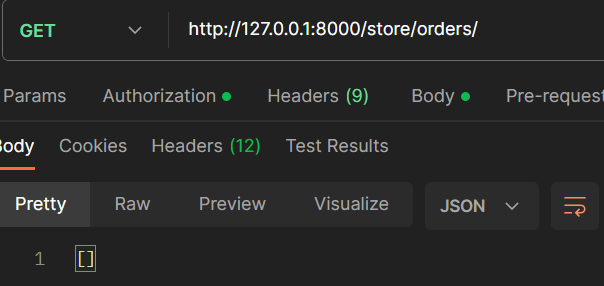
        if user.is\_staff:

            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.

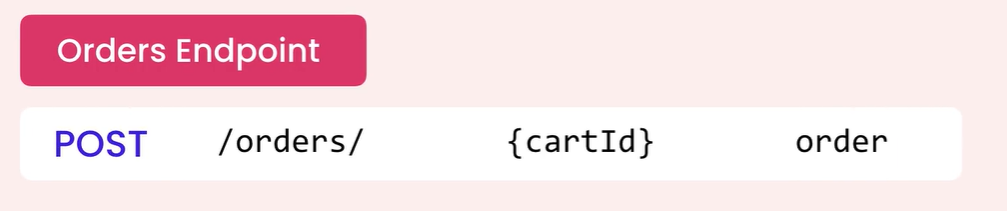
So we hit this orders/ endpoint for reading orders of a customer, we will end up creating a customer record. So this is a violation of ***command query separation*** principle.

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]

    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(

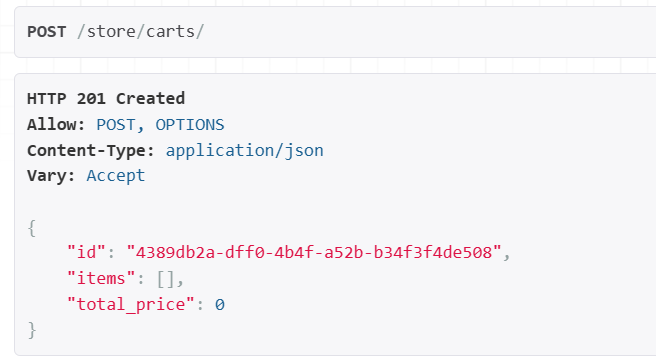
            user\_id=user.id

        )

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

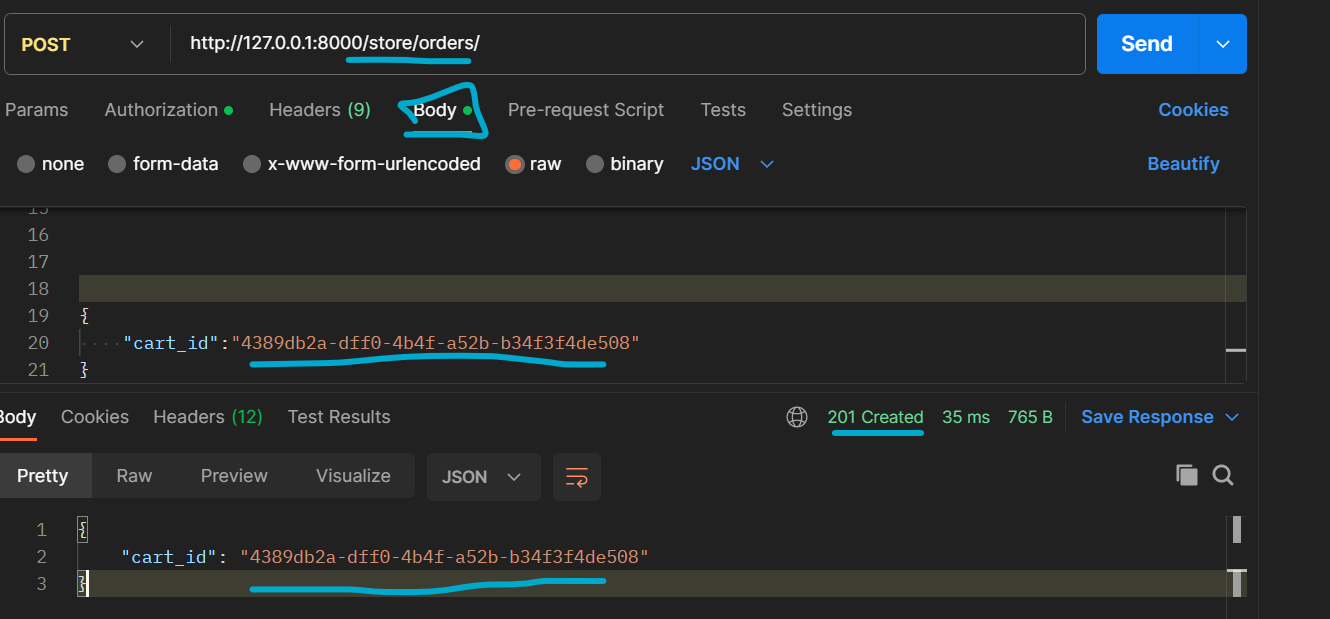
Here we are saying if our request method is POST we return *CreateOrderSerializer* otherwise we return OrderSerializer.

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



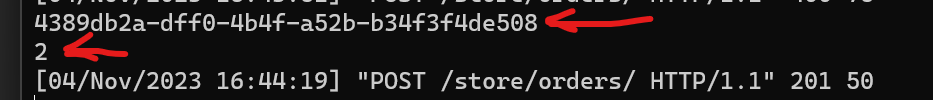
We copy the cart\_id 4389db2a-dff0-4b4f-a52b-b34f3f4de508

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"])

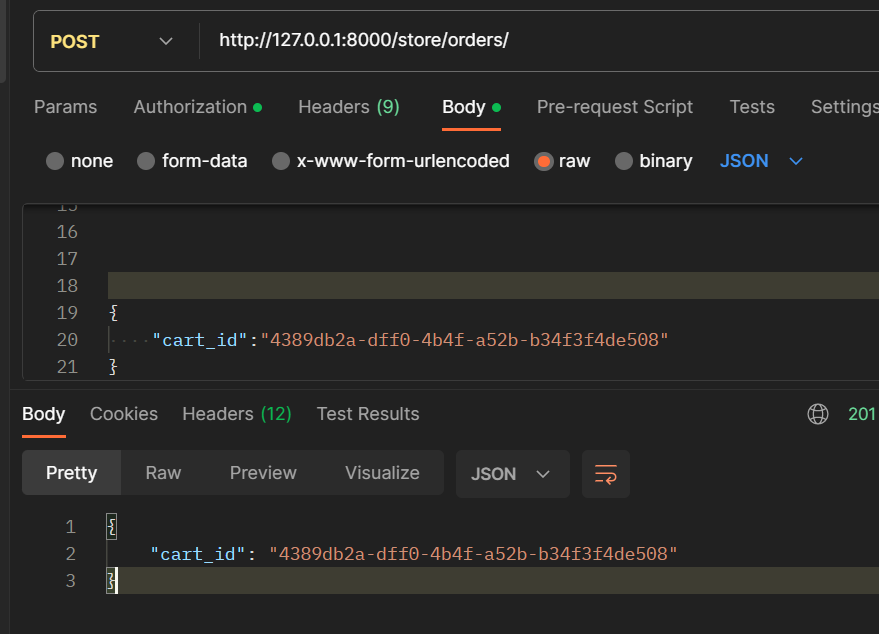
        (customer, created) = Customer.objects.get\_or\_create(

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

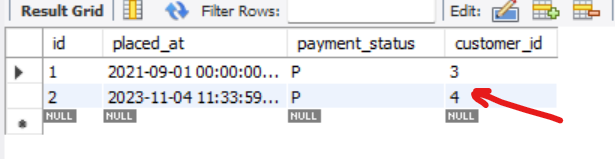
        )

        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.

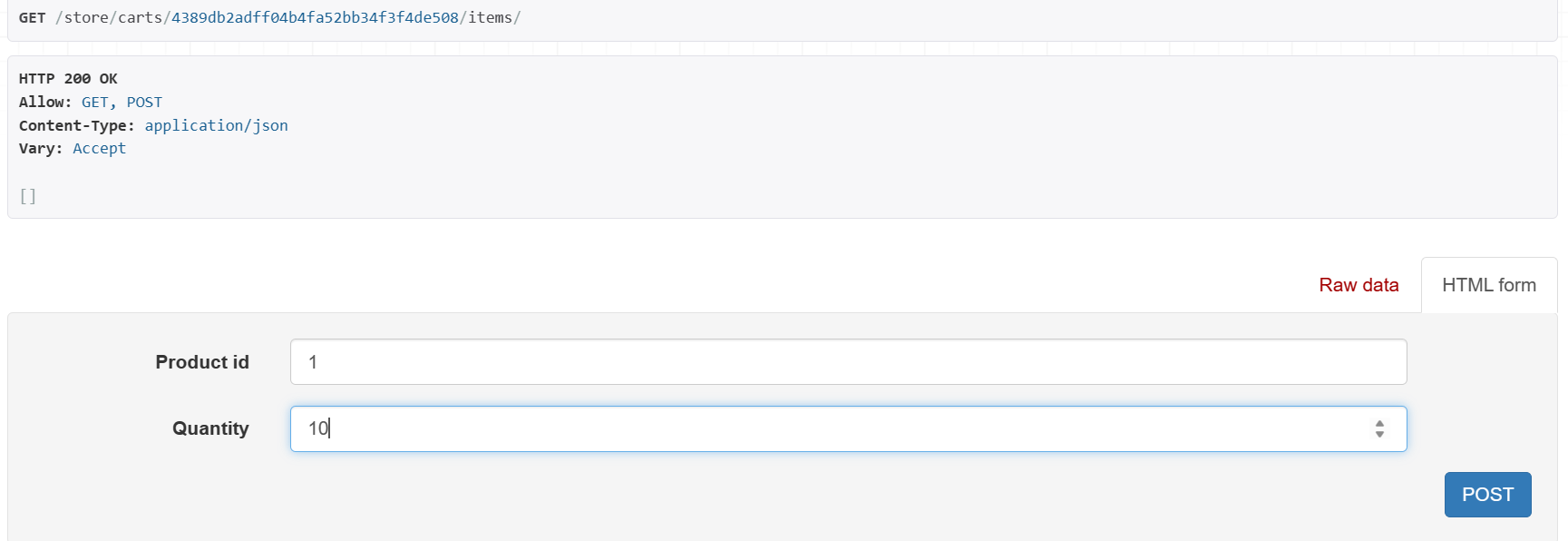
Let’s test our implementation.

We have a empty cart in cart table,

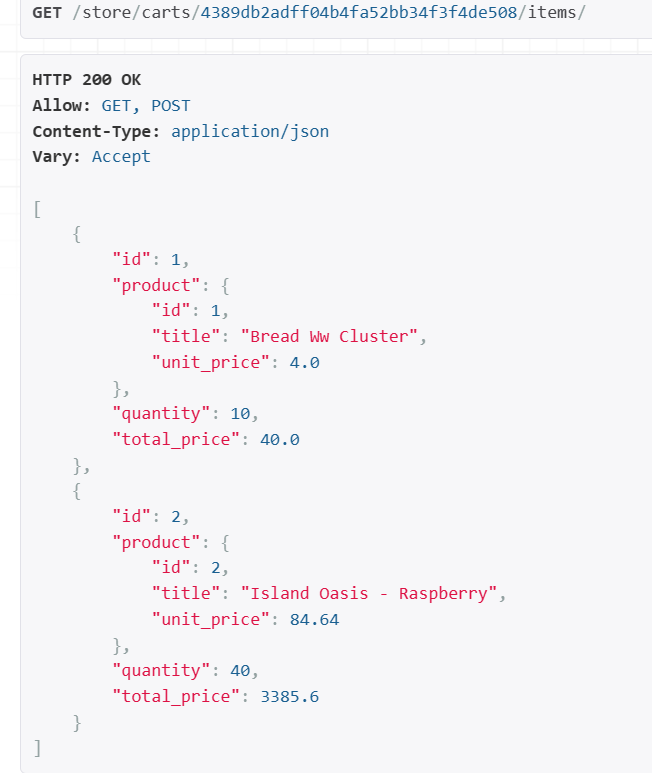


Let’s use this cart’s id and go to its /items endpoint,

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

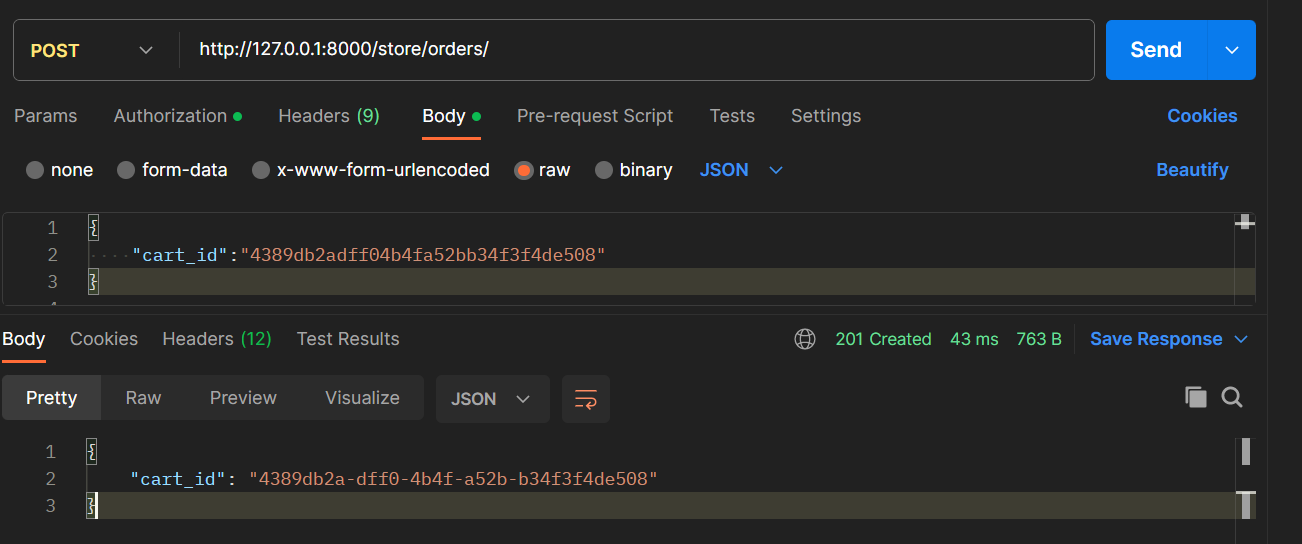


Now add some products to this cart,



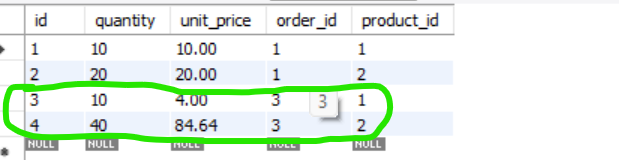
As a result we have two items in our shopping cart with id 4389db2adff04b4fa52bb34f3f4de508

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.

But there is a problem in this implementation.

First we create an order,

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

Then we create a bunch of orderitems,

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

And finally we delete the cart,

Cart.objects.filter(pk=cart\_id).delete()

So we have multiple updates in the database, Now *what if our database server goes offline in the middle of one of these updates, we will end up in an inconsistent state* (*some of the data will be missing*).

This is where we use a ***transaction***, using a transaction we can treat a block of code as atomic like an atom. *Either every code in that block should be executed or if something goes wrong, all the save changes should be rolled back*. This is the benefit of using *transaction*.

So on the top from *django.db* we import *transaction* module.

from django.db import transaction

Now we will wrap the entire code block inside *save* method with *transaction.atomic*(),

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        with transaction.atomic(): -🡪 *Here we use it*

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

            (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

            )

            order\_items = [

                OrderItem(

                    order=order,

                    product=item.product,

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

                    quantity=item.quantity,

                )

                for item in cart\_items

            ]

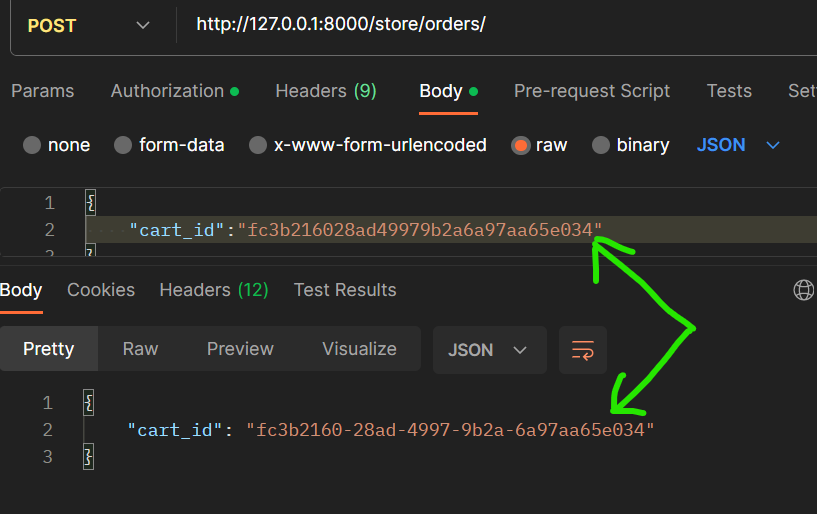
            OrderItem.objects.bulk\_create(order\_items)

            Cart.objects.filter(pk=cart\_id).delete()

This *transaction.atomic* function *returns a transaction object that we pass using with block*.

**Returning the Created Order**:

We have noticed, then when we post a cart\_id to orders endpoint instead of an order object we get a cart\_id back,



This is not how we design our API. But before solving this problem we need to understand why this happens and then come up with right solution.

class OrderViewSet(ModelViewSet): 🡪 *ModelViewSet*

    permission\_classes = [IsAuthenticated]

Our OrderViewSet is derived from *ModelViewSet*,

class ModelViewSet(mixins.CreateModelMixin,

                   mixins.RetrieveModelMixin,

                   mixins.UpdateModelMixin,

                   mixins.DestroyModelMixin,

                   mixins.ListModelMixin,

                   GenericViewSet):

Then look at *CreateModelMixin*,

class CreateModelMixin:

    """

    Create a model instance.

    """

    def create(self, request, \*args, \*\*kwargs):

        serializer = self.get\_serializer(data=request.data)

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

        self.perform\_create(serializer)

        headers = self.get\_success\_headers(serializer.data)

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

    def perform\_create(self, serializer):

        serializer.save()

    def get\_success\_headers(self, data):

        try:

            return {'Location': str(data[api\_settings.URL\_FIELD\_NAME])}

        except (TypeError, KeyError):

            return {}

Look the *create* method, first we get a serializer(*which comes from our viewset where we either set serializer class attribute or override get\_serializer class*)

After getting the serializer, we validate the data (*serializer.is\_valid*), save the changes and then finally using the same serializer we return the response back to the client (*Response(serializer.data, status=status.HTTP\_201\_CREATED)*).

In this case our serializer has only one field (*cart\_id*) and that is why what we send to the server is exactly what we get back.

Now to solve this problem and get the order object, we need to create a different serializer and then we need to override the create method because we cannot rely on this implementation.

*In the new implementation we are going to use two serializers. One serializer to deserialize the data and get the cart\_id and other serializer to return the order back to client*.

So back to the OrderViewSet,

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    def create(self, request, \*args, \*\*kwargs):

        serializer = CreateOrderSerializer(request.data)

Here we are going to override create method, so we create a serializer here CreateOrderSerializer and give it the request data.

Next we validate the data and save it(*exactly like CreateModelMixin*).

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    def create(self, request, \*args, \*\*kwargs):

        serializer = CreateOrderSerializer(request.data)

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

        serializer.save()

Now we are going to create another serializer, so we reset serializer to OrderSerializer and give it our order object.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    def create(self, request, \*args, \*\*kwargs):

        serializer = CreateOrderSerializer(request.data)

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

        order = serializer.save()

        serializer = OrderSerializer(order) 🡪 *this order object we get from* ***save***

*But where is Order object*?

Well we should return it from the *save* method.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        with transaction.atomic():

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

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

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

            )

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

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

                cart\_id=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)

            Cart.objects.filter(pk=cart\_id).delete()

            return order 🡪 *We are returning*

And finally we return this new serializer.data in response,

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    def create(self, request, \*args, \*\*kwargs):

        serializer = CreateOrderSerializer(data=request.data) *old serializer*

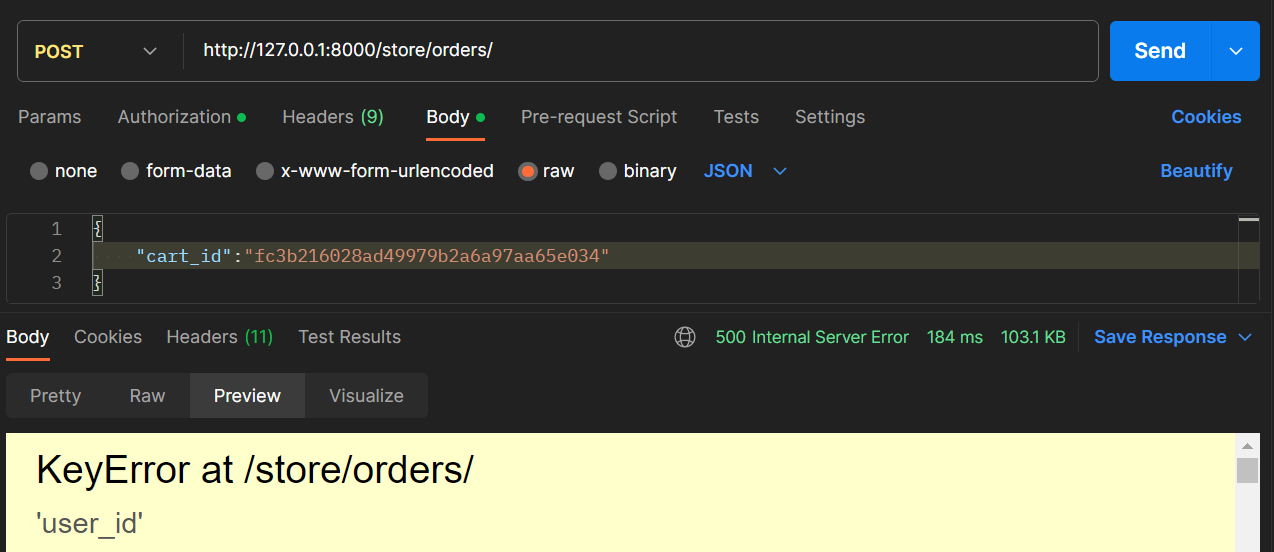
        serializer.is\_valid(raise\_exception=True)

        order = serializer.save()

        serializer = OrderSerializer(order) 🡪 *new serializer*

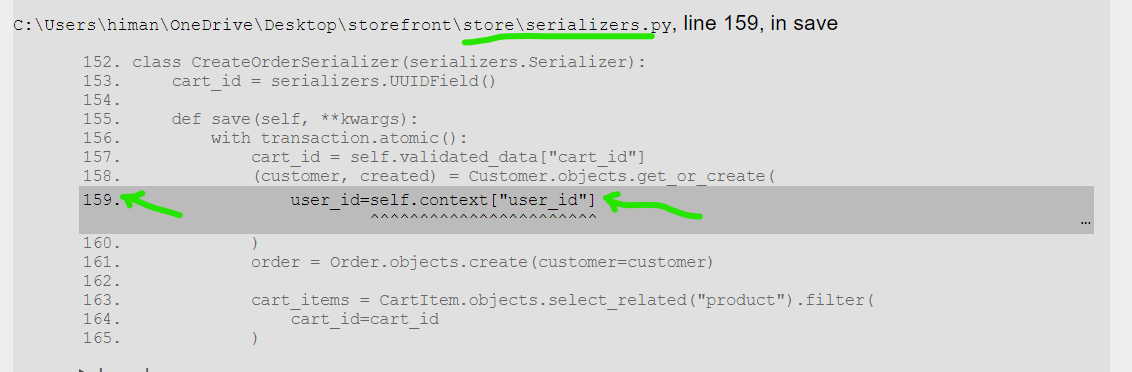
        return Response(serializer.data) 🡪 *Then we return new serializer*

Let’s test this implementation,



We get an exception but no meaningful message , just user\_id.

Let’s look at the stack trace, where this exception was thrown,



In our serializer at this line, this is the code

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

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

            )

Back to the create method,

    def create(self, request, \*args, \*\*kwargs):

        serializer = CreateOrderSerializer(data=request.data)

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

        order = serializer.save()

        serializer = OrderSerializer(order)

        return Response(serializer.data)

When creating this CreateOrderSerializer, we needed to give it a *context* object. So we have access to the user\_id.

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

    def create(self, request, \*args, \*\*kwargs):

        serializer = CreateOrderSerializer(

            data=request.data, context={"user\_id": self.request.user.id}

        ) -🡪 *Here we provide context*

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

        order = serializer.save()

        serializer = OrderSerializer(order)

        return Response(serializer.data)

    def get\_serializer\_class(self):

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

            return CreateOrderSerializer

        return OrderSerializer

    # def get\_serializer\_context(self): 🡪 *Now we can remove this method*

    #     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(

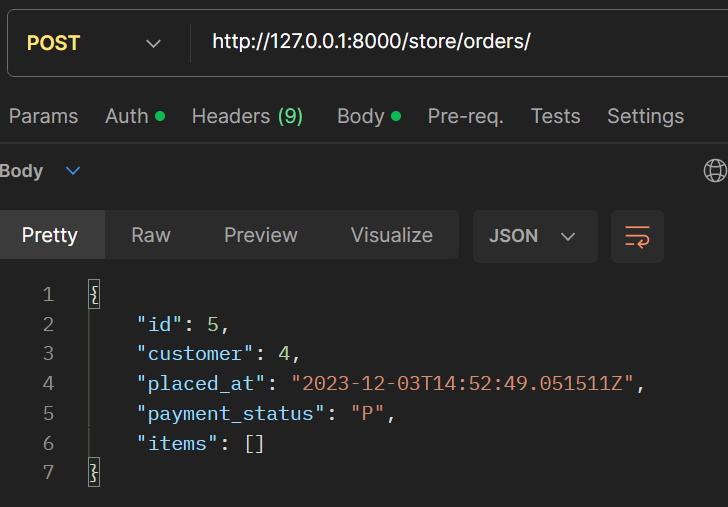
            user\_id=user.id

        )

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

Here we set context (*our dictionary with user\_id as key and value as self.request.user.id*)to our CreateOrderSerializer and remove the *get\_serializer\_context* method, since we are no longer relying on *create* method’s default implementation in *CreateModelMIxin* instead we are creating it from scratch.

If we refresh,



Now we get a new order object with given id.

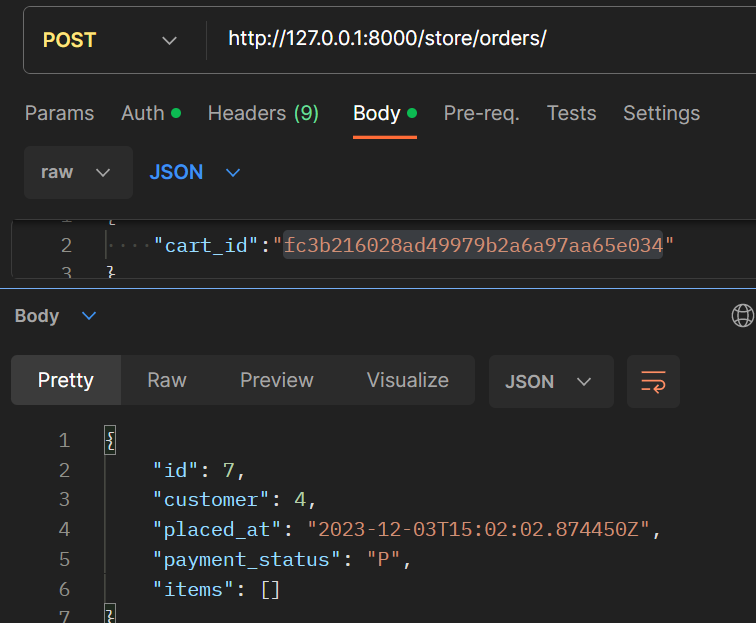
**Data Validation**:

So we implemented and tested the happy path where we received valid data but let’s see what happens when we receive invalid data.

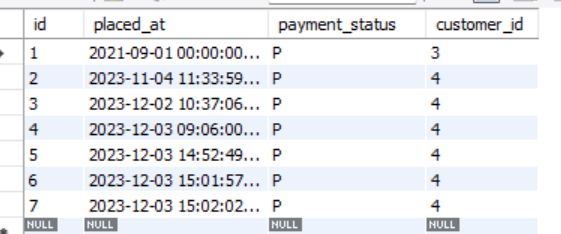
So there are two scenarios we need to consider here,

*One is that we received invalid cart that does not exist and other is we receive an empty cart, the cart with no items*…*in that case we don’t want to create an empty order without any items*.

Our cart is deleted, but if we post this cart\_id again,



Nothing happens here, but if we look at the orders table,



We are creating all these extra orders that don’t have any items.

In our save method, look at the implementation, we always create order right from the get go.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def save(self, \*\*kwargs):

        with transaction.atomic():

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

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

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

            )

            order = Order.objects.create(customer=customer) 🡪 *Here*

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

                cart\_id=cart\_id

            )

We create an order whether a given cart exist or not. This is not right.

So first we need to validate the cart\_id and for that we need to create our own validate method,

According to convention, start the method name with validate followed by underscore and name of the field.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def validate\_cart\_id(self,cart\_id): 🡪 *here*

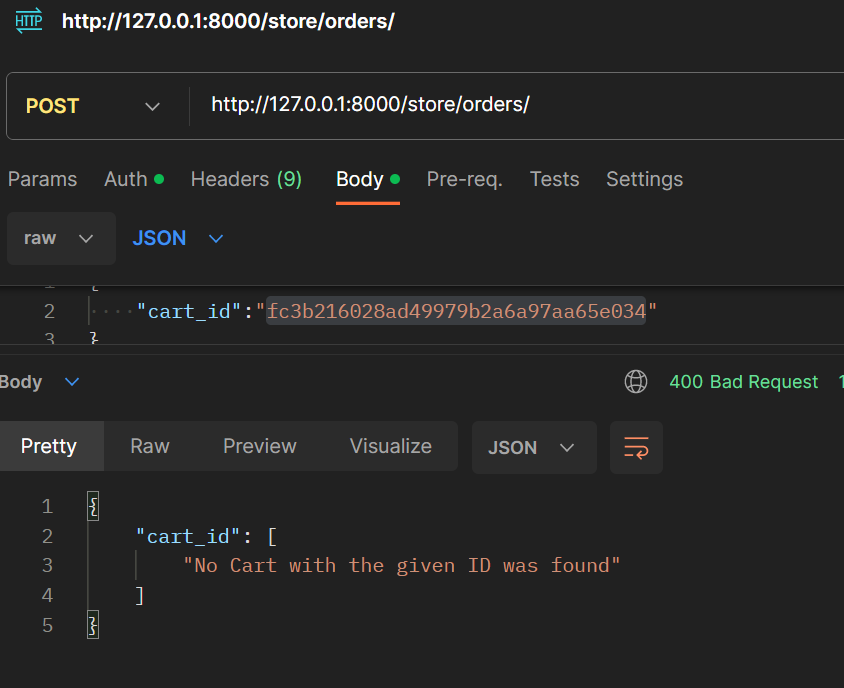
        if not Cart.objects.filter(pk=cart\_id).exists():

raise serializers.ValidationError('No Cart with the given ID was found')

        return cart\_id

If the cart with this cart\_id does not exist, then we are going to raise a validation error otherwise we are going to return cart\_id as the valid value.

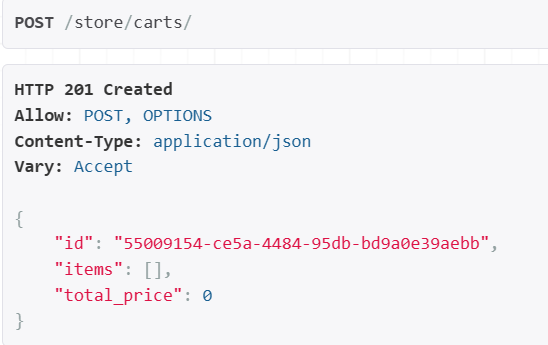
Let’s test this scenario before going further,



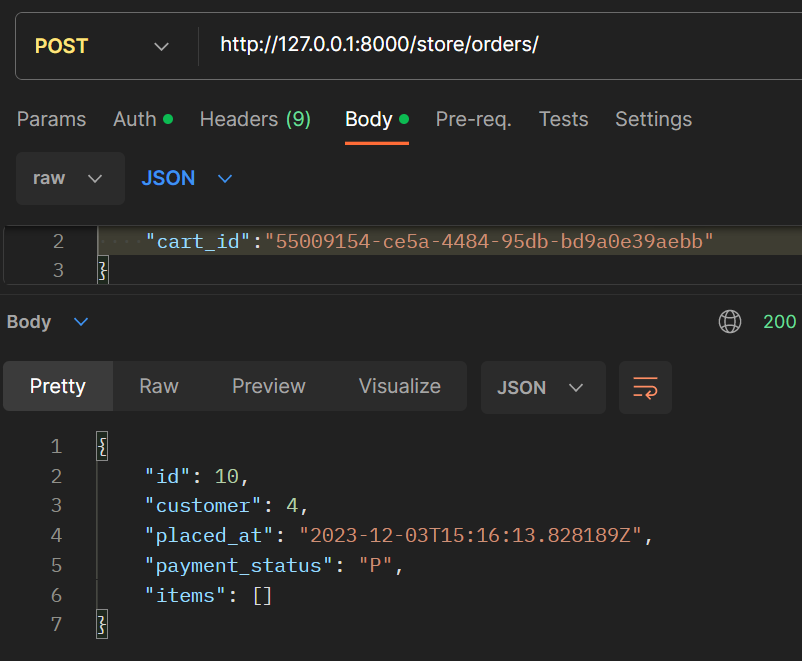
Now we get an error with status code 400 which means client sent invalid data.

Now the second scenario, we are going to send an empty cart to the server.

So for that first we need to create an empty cart.



Now let’s send this cart to orders endpoint,



We did not get any errors, but we get an extra order without any items.

So to solve that we go back to our validate method and add another if statement.

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def validate\_cart\_id(self, cart\_id):

        if not Cart.objects.filter(pk=cart\_id).exists():

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

        if CartItem.objects.filter(cart\_id=cart\_id).count() == 0:

            raise serializers.ValidationError("The cart is empty")

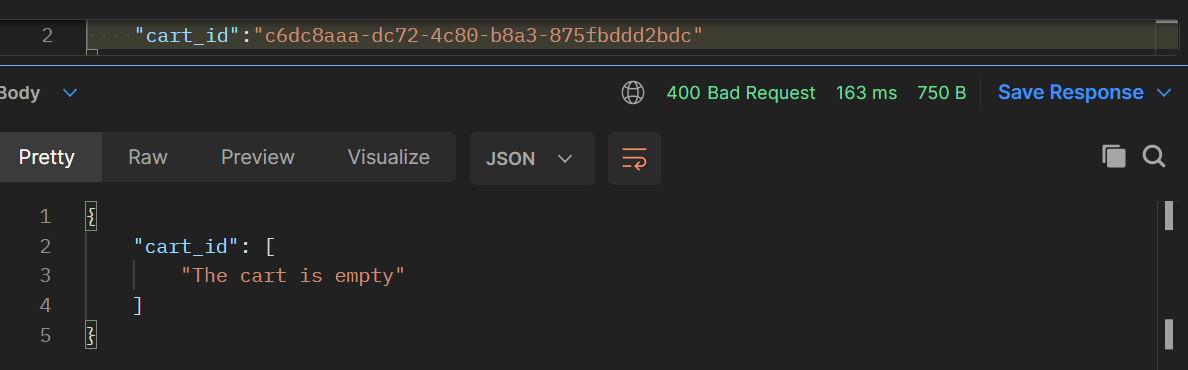
        return cart\_id

Here we check count of CartItems for this cart\_id, if 0 then we raise validation error.

Let’s create another shopping cart,



Now we get right validation error,



Here is how we should write code. Always do things step by step. Don’t try to do too many things.

🡪Focus on a single task.

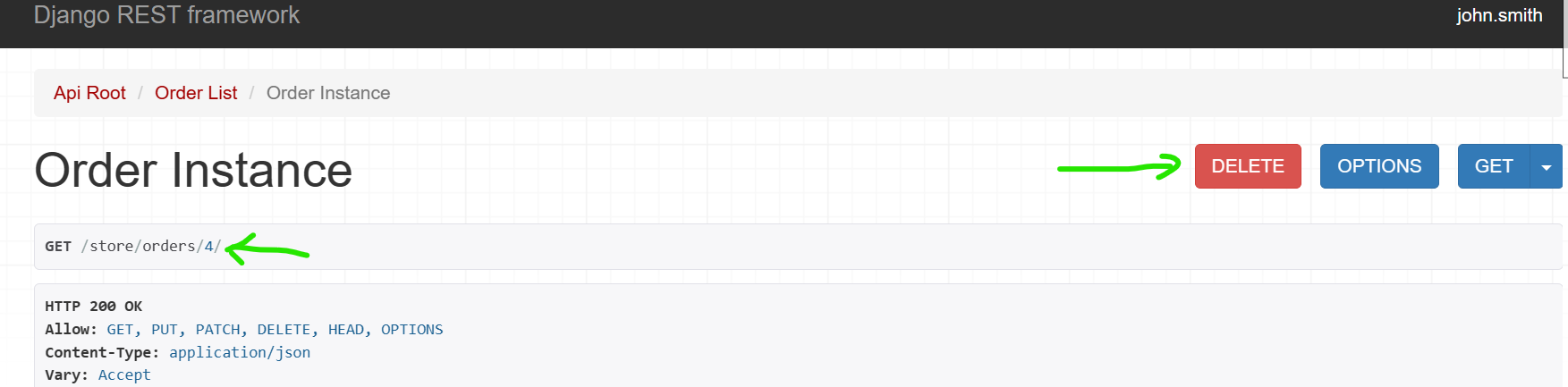
🡪 First build the happy path then improve it,

🡪 then refactor code,

🡪 then add data validation and so on.

**Revisiting the permissions**:

Let’s go to a specific order that we have placed,



We can delete and update this order. But this function should not be available to all authenticated users, we need to restrict this only to admins.

So back to our viewset,

class OrderViewSet(ModelViewSet):

    permission\_classes = [IsAuthenticated]

Instead of setting our permission classes here, we are going to override the *get\_permissions* function.

class OrderViewSet(ModelViewSet):

    def get\_permissions(self): 🡪 *Here*

        if self.request.method in ["PUT", "PATCH", "DELETE"]:

            return [IsAdminUser()]

        return [IsAuthenticated()]

Here we can say if request.method is in this list (*which includes PUT, PATCH or DELETE operations*) then we return *IsAdminUser*() otherwise we return *IsAuthenticated*() (*Make sure we are returning list of objects not permission classes*).

Here we can also argue that we might not want to support PUT requests because we are not going to update all properties of an existing order (*we want to update a subset of them like payment status*).

So let’s remove PUT method from get\_permissions and restrict HTTP methods.

class OrderViewSet(ModelViewSet):

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

🡪 *Make sure these are lowercase*

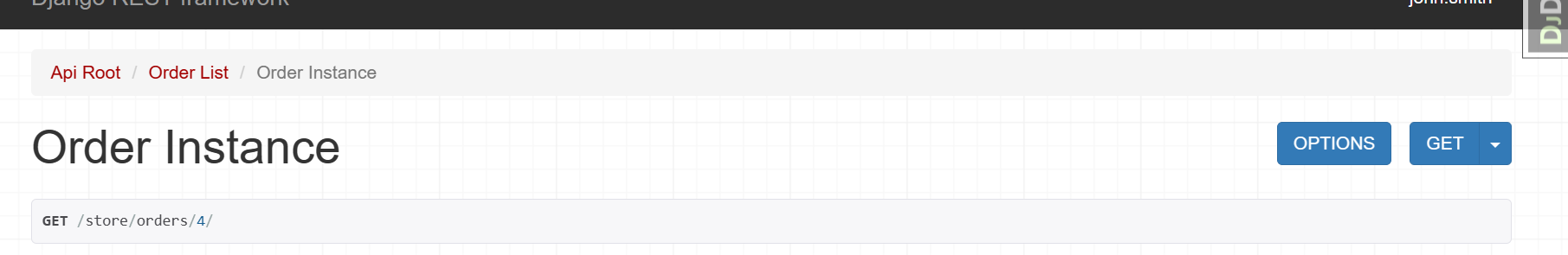
    def get\_permissions(self):

        if self.request.method in ["PATCH", "DELETE"]:

            return [IsAdminUser()]

        return [IsAuthenticated()]

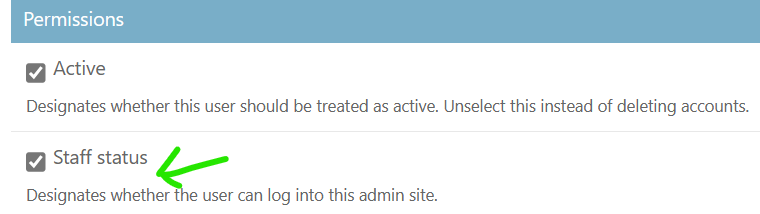
We can also enable *head* and *options* for some clients to see what operations are available at this endpoint.



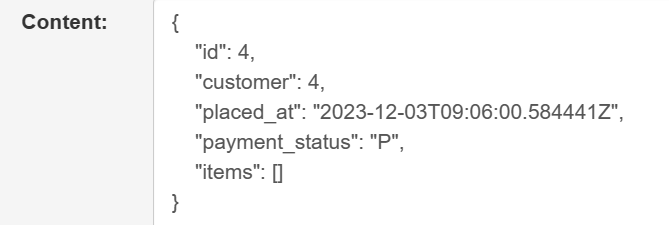
Now when we refresh this page, delete button is gone and we can no longer update this order.

**Updating an order**:

Let’s talk about updating an order, but first let’s make john smith a admin user.



After this we have access to delete and update order functions.



But we don’t want to patch all these properties, for now we only want to patch *payment\_status* and everything else should be read only.

We have two ways to implement this, Look at the OrderSerializer class,

class OrderSerializer(serializers.ModelSerializer):

    items = OrderItemSerializer(many=True)

    class Meta:

        model = Order

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

One option is to redefine all these fields and mark them as read only except payment\_status.

But this approach is not good because we will end up polluting this class and plus *in the future if we add new fields in the order class we always have to come back here, redefine those fields and mark them as read only*. So this is not a very solid solution.

A better approach will be to create a custom serializer for updating an order just like we created CreateOrderSerializer for creating an order.

class UpdateOrderSerializer(serializers.ModelSerializer):

    class Meta:

        model = Order

        fields = ["payment\_status"]

Now back to our viewset, let’s modify our get\_serializer\_class method accordingly.

class OrderViewSet(ModelViewSet):

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

    def get\_permissions(self):

        if self.request.method in ["PATCH", "DELETE"]:

            return [IsAdminUser()]

        return [IsAuthenticated()]

    def get\_serializer\_class(self):

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

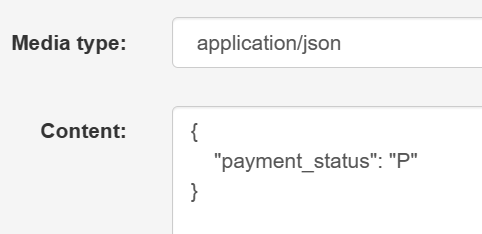
            return CreateOrderSerializer

        elif self.request.method == "PATCH": 🡪 *Here*

            return UpdateOrderSerializer

        return OrderSerializer

If we refresh the page now,



Let’s change the payment status to “C”



Now we have an updated order.

**Signals**:

*In Django we use* ***signals*** *to decouple our apps and prevent them from stepping on each other’s toes*.

Let’s understand them by using a real world example.

Back to our OrderViewSet, look at the implementation of the *get\_queryset* method.

    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 🡪 *Violation of command – query separation*

        )

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

Earlier we saw that , here we have a violation of command – query separation principle because *get\_queryset* is a query method which we call to get some data but we are potentially changing the state or data in the system (*when we used get\_or\_create method inside*).

Note: Command query separation principle means our methods should either be commands and change the state of the system or queries which means they should return data, but not both.

The reason we have to use *get\_or\_create* method here is because it is possible that at time this *get\_queryset* method is called we don’t have a customer associated with this user but what if *when someone registers we automatically create a customer record for that user*.

To implement this, we can go to UserCreateSerializer and in this serializer we can override the *save* method.

class UserCreateSerializer(BaseUserCreateSerializer):

    def save(self, \*\*kwargs):

        user =  super().save(\*\*kwargs) 🡪 we get user object

        Customer.objects.create(user) 🡪 Then create a customer record with user

    class Meta(BaseUserCreateSerializer.Meta):

        fields = ["id", "username", "password", "email", "first\_name", "last\_name"]

First we call the *save* method of the base class which creates and returns a user object then we can create a Customer using this user object. Pretty straightforward.

Earlier we discussed that it is not a good idea to touch customer profiles in this serializer (*like adding a birth\_date field*), because with this approach as our registration form gets more complex, this UserCreateSerializer has to worry about more aspects of this application which is not good.

So there is another way to solve this problem where we use *signals*.

***“****In Django our models have a bunch of signals or notifications that are fired at different times****”***.



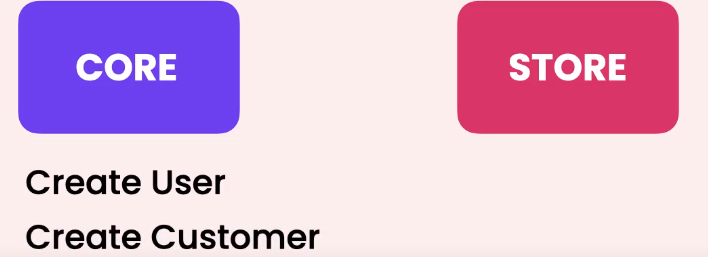
For example we have *pre\_save* (*fired before a model is saved*), *post\_save* (*fired after a model is saved*) and so on.

So *in our application we can listen to these notifications and do something*.

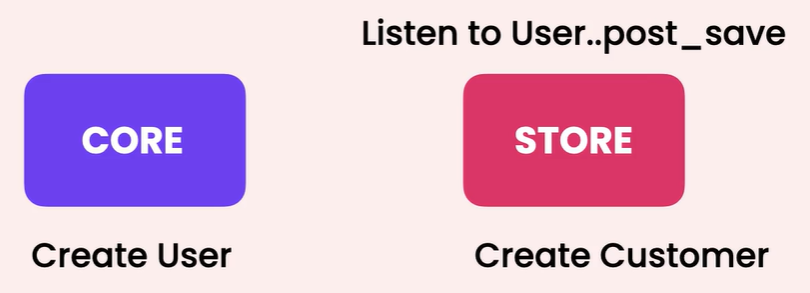
Here is our current implementation



In the core app where we have UserCreateSerializer, we are responsible for two things, creating a user and creating a customer.

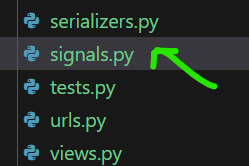


What if we go in the store app and *listen to post\_save signal of the User model which means we are going to get notified when a user registers and then we can create a Customer record right there*.



With this approach we have shifted the responsibility of creating a customer from the *core* app to the *store* app.

So undo the changes just done in UserCreateSerializer and in the store app let’s create a new file called *signals.py*.



In this file we will define a function called create\_customer\_for\_new\_user (*a very descriptive name*). This function should have two parameters *sender* (*the class which is sending the notification*) and our \*\*kwargs or keyword arguments.

def create\_customer\_for\_new\_user(sender, \*\*kwargs):

In our kwargs we have a key called ‘created’ which is a Boolean, so we can check to see if a new model instance is created. If so we create a customer.

from .models import Customer

def create\_customer\_for\_new\_user(sender, \*\*kwargs):

    if kwargs["created"]:

        Customer.objects.create(user=kwargs["instance"])

To get the instance we go to our kwargs and pick the instance.

Now we have this function, we need to tell Django that this function should be called when a user model is saved. So first we need to import a *receiver* decorator.

from django.dispatch import receiver

Then we need to apply this receiver over this function,

from django.dispatch import receiver

from .models import Customer

@receiver()

def create\_customer\_for\_new\_user(sender, \*\*kwargs):

    if kwargs["created"]:

        Customer.objects.create(user=kwargs["instance"])

Here we are passing two arguments to this *receiver*. First one is the signal we are interested in which in our case is *post\_save*.

from django.dispatch import receiver

from django.db.models.signals import post\_save 🡪 *import*

from .models import Customer

@receiver(post\_save)

def create\_customer\_for\_new\_user(sender, \*\*kwargs):

    if kwargs["created"]:

        Customer.objects.create(user=kwargs["instance"])

Next we don’t need to listen to post\_save signal of every model in this project and we are only interested in the post\_save invent of User model, so we set the *sender*.

Note: Here we don’t want to reference the *User* model defined in the core app because this is going to add a dependency from the store app to the core app.

So just like before we import settings module and then we reference settings.AUTH\_USER\_MODEL.

from django.dispatch import receiver

from django.conf import settings 🡪 *import settings*

from django.db.models.signals import post\_save

from .models import Customer

@receiver(post\_save, sender=settings.AUTH\_USER\_MODEL) 🡪 *refer AUTH\_USER\_MODEL*

def create\_customer\_for\_new\_user(sender, \*\*kwargs):

    if kwargs["created"]:

        Customer.objects.create(user=kwargs["instance"])

So we have defined a function and now Django knows that it should call this function every time a user model is saved.

But this code is not executed unless we import it somewhere. So in the store app, go to apps module.

from django.apps import AppConfig

class StoreConfig(AppConfig):

    default\_auto\_field = 'django.db.models.BigAutoField'

    name = 'store'

In this config class, we override the *ready* method. This method is called when this store app is ready or initialized.

from django.apps import AppConfig

class StoreConfig(AppConfig):

    default\_auto\_field = 'django.db.models.BigAutoField'

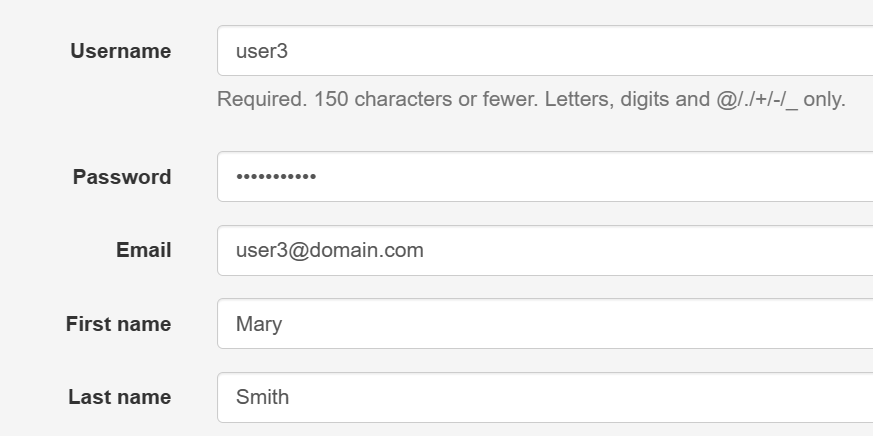
    name = 'store'

    def ready(self) -> None: 🡪 *Here*

        import store.signals

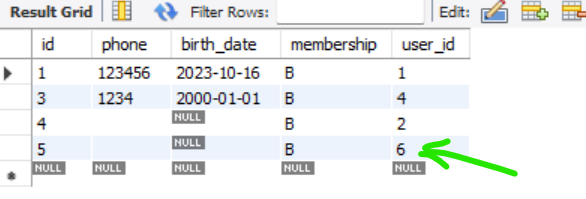
        return super().ready()

To test this implementation, go to */auth/users/* endpoint and create a new user.



 Our new user with user\_id = 6 is created now, let’s see if the respective customer is also created.

In the database inside customer table we can see a customer is created with this user\_id = 6.



Back to get\_queryset method where we first encountered the problem, with our new implementation we no longer have to worry about creating a customer.

    def get\_queryset(self):

        user = self.request.user

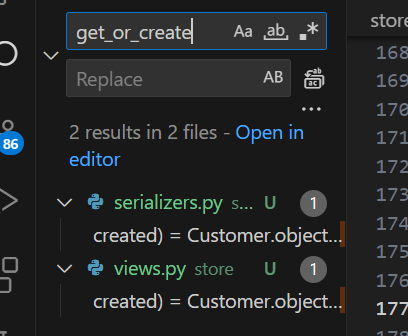
        if user.is\_staff:

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

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

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

Now we need to apply same change to couple of places.



Search for get\_or\_create in this project and do the same changes at those places as well.

**Creating Custom Signals**:

Every model in Django fires a bunch of signals at different times, now we can also create our own custom signals. For example *every time we receive an order in the store app, we can fire a signal like ‘order created’*.

*With this, other apps that are interested in this event can subscribe to this signal and get notified*.

In the store app, let’s look at *signals* module,

@receiver(post\_save, sender=settings.AUTH\_USER\_MODEL)

def create\_customer\_for\_new\_user(sender, \*\*kwargs):

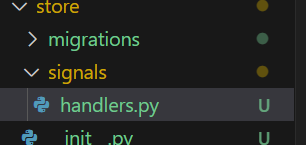
    if kwargs["created"]:

        Customer.objects.create(user=kwargs["instance"])

This function that we defined is called a ***signal handler***.

In this module we can also define the new signal but *its best to separate our signals from handlers*.

In The store app we will add a new folder called signals and then move signals to this module and call it *handlers* now.



from django.dispatch import receiver

from django.conf import settings

from django.db.models.signals import post\_save

from store.models import Customer 🡪 *We gave full path to models*

@receiver(post\_save, sender=settings.AUTH\_USER\_MODEL)

def create\_customer\_for\_new\_user(sender, \*\*kwargs):

    if kwargs["created"]:

        Customer.objects.create(user=kwargs["instance"])

Now we need to go to apps module and load store.signals.handlers.

from django.apps import AppConfig

class StoreConfig(AppConfig):

    default\_auto\_field = "django.db.models.BigAutoField"

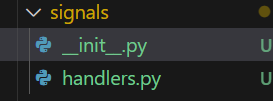
    name = "store"

    def ready(self) -> None:

        import store.signals.handlers 🡪 *Here*

        return super().ready()

In the signals folder, we are going to add a \_\_init\_\_.py module and this is where we will define our signals.



from django.dispatch import Signal

order\_created = Signal()

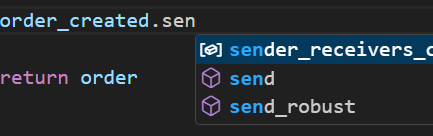
Here in \_\_init\_\_ module we import a *Signal* class and then we create a signal object (*we call it order\_created*). So *a signal is simply an instance of the Signal class*.

Now we need to fire this signal when an order is created and for this we will go to CreateOrderSerializer.

In the serializers module, First import order\_created object,

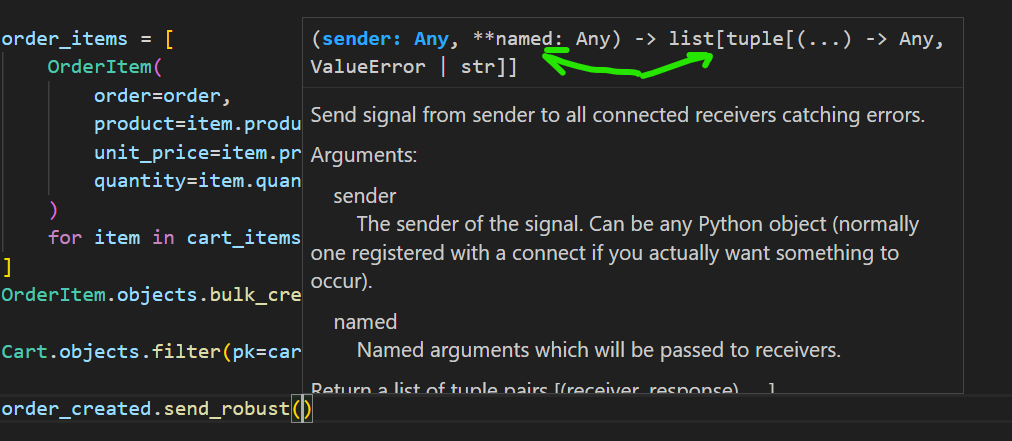
from .signals import order\_created

this order\_created object has a couple of methods for sending a signal.



We have *send* and *send\_robust*, the difference is that with *send*, *if one of the receivers fail and throws an exception, the other receivers are not notified*.

So here we will use *send\_robust*.



Look at the arguments, one of them is called *sender*. This is the *class that is sending the signal*.

So let’s go back to handlers module,

@receiver(post\_save, sender=settings.AUTH\_USER\_MODEL)

def create\_customer\_for\_new\_user(sender, \*\*kwargs):

    if kwargs["created"]:

        Customer.objects.create(user=kwargs["instance"])

Earlier when we defined this function, we gave it two parameters sender and kwargs.

So when calling a handler we need to supply the sender argument.

order\_created.send\_robust(self.\_\_class\_\_)

Here we get the current class from self*.\_\_class\_\_* , this is a magic attribute that returns class of the current instance.

Optionally we can supply additional data with our signal, in this case order that was created.

order\_created.send\_robust(self.\_\_class\_\_, order=order)

So we set order as order and pass it as a kwarg.

Here is our complete CreateOrderSerializer after changes,

class CreateOrderSerializer(serializers.Serializer):

    cart\_id = serializers.UUIDField()

    def validate\_cart\_id(self, cart\_id):

        if not Cart.objects.filter(pk=cart\_id).exists():

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

        if CartItem.objects.filter(cart\_id=cart\_id).count() == 0:

            raise serializers.ValidationError("The cart is empty")

        return cart\_id

    def save(self, \*\*kwargs):

        with transaction.atomic():

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

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

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

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

                cart\_id=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)

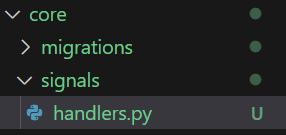
            Cart.objects.filter(pk=cart\_id).delete()

            order\_created.send\_robust(self.\_\_class\_\_, order=order) 🡪 *here*

            return order

Now in the store app, every time we create an order, this signal is fired. With this in place we can go to the core app and receive this signal.

So in the core app we create a new folder called signals and inside it a handlers module,



Inside this handlers module,

from store.signals import order\_created

from django.dispatch import receiver

@receiver(order\_created)

def on\_order\_created(sender, \*\*kwargs):

    print(kwargs["order"]) 🡪 *kwarg that we passed on firing this signal*

For now we will just print order that we received.

Next the final step, we need to load this module when this app is ready. So we need to go to apps module of core app and override *ready* method.

from django.apps import AppConfig

class CoreConfig(AppConfig):

    default\_auto\_field = "django.db.models.BigAutoField"

    name = "core"

    def ready(self) -> None: 🡪 *Here*

        import core.signals.handlers

At the orders endpoint we will create a new order,

**Side note**: By mistake we removed post method from this end point, so let’s add it in OrderViewSet class,

class OrderViewSet(ModelViewSet):

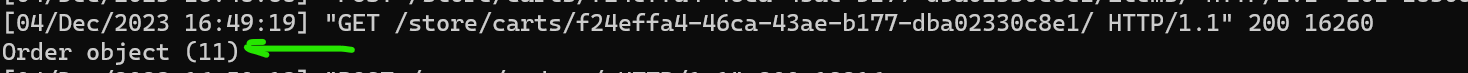
    http\_method\_names = ["get", "post", "patch", "delete", "head", "options"]

🡪 *post added now*,

Now we will pass a cart id to this orders endpoint (*make sure cart is valid with items inside it*),

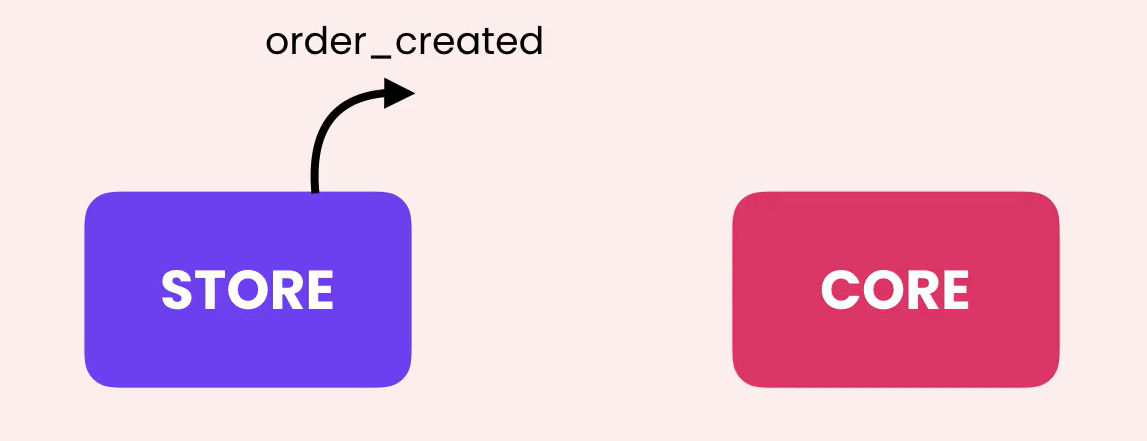


As soon as we post it,



We can see the order object that we received in the core.

In our current implementation our store app fires the *order\_created* event and the core app simply gets the order and prints it on the terminal.



The interesting part about this model is that the store app does not care what happens in the core app after this order is created.

Similarly we can have many other apps that are interested in this event and every time an order is created, those apps will get notified and do something that is relevant in their domain.

