

**Introduction**:

In this section we will learn about how to query and manipulate data using Django object relational mapper(ORM). We will also cover:

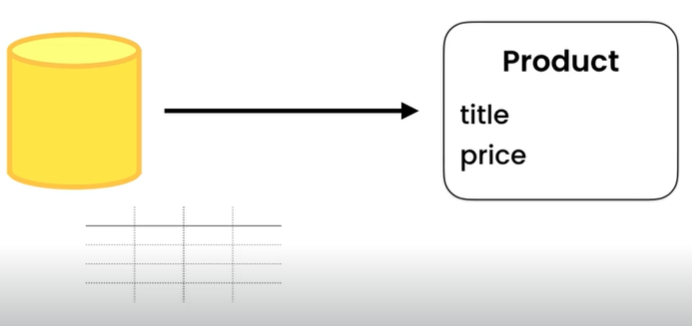
1. Filtering data
2. Sorting data
3. Grouping data

And much more.

By the end of this section we will be able to confidently pull out data or store it in a relational database.

**Django ORM**:

In Relational databases data is stored as rows in tables, so when we are pulling up data from relational database, we need to map these rows into objects.



Note: ORM allows developers to write code in programming language of their choice instead of using SQL queries to access, read or delete data from RDB by *forming a layer between the language and database*.

In the past we used to do this by hand which was pretty repetitive and time consuming, it looks like this…



🡪First we need to write SQL query

🡪send it to the database.

🡪Read the result and map them to a bunch of objects.

So for each record we had to create a new object and set its attributes.

This is where an Object relational mapper comes into picture.

*“An object relational mapper as the name implies, maps objects to relational records which frees us from writing a lot of repetitive code”*.

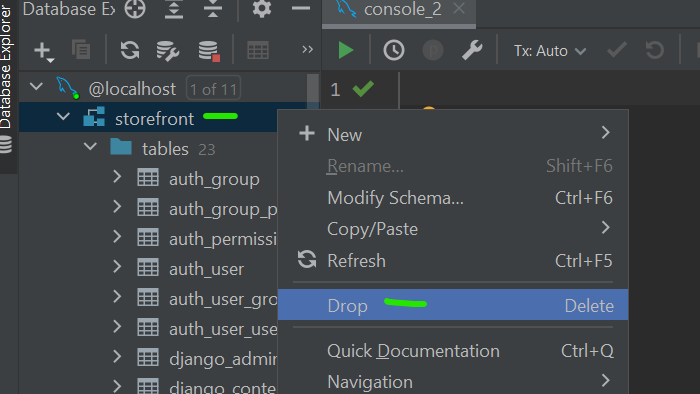
Now we do not have to write SQL code to query or manipulate data. We can code in an object oriented programming language like python. *ORM will then translate our python code into SQL code at runtime*.

A good example is Django migrations where we generated our database tables almost instantly without writing any SQL code.

Note: All the model classes we used so far, inherit from Model class in Django which is also a part of Django ORM. Since all of them inherit the functionality to query or manipulate data from the base model class.

**Resetting the database**:

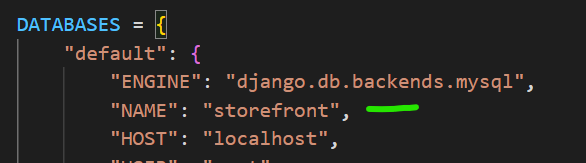
1. Go to localhost and Delete the existing *storefront* database using drop option.



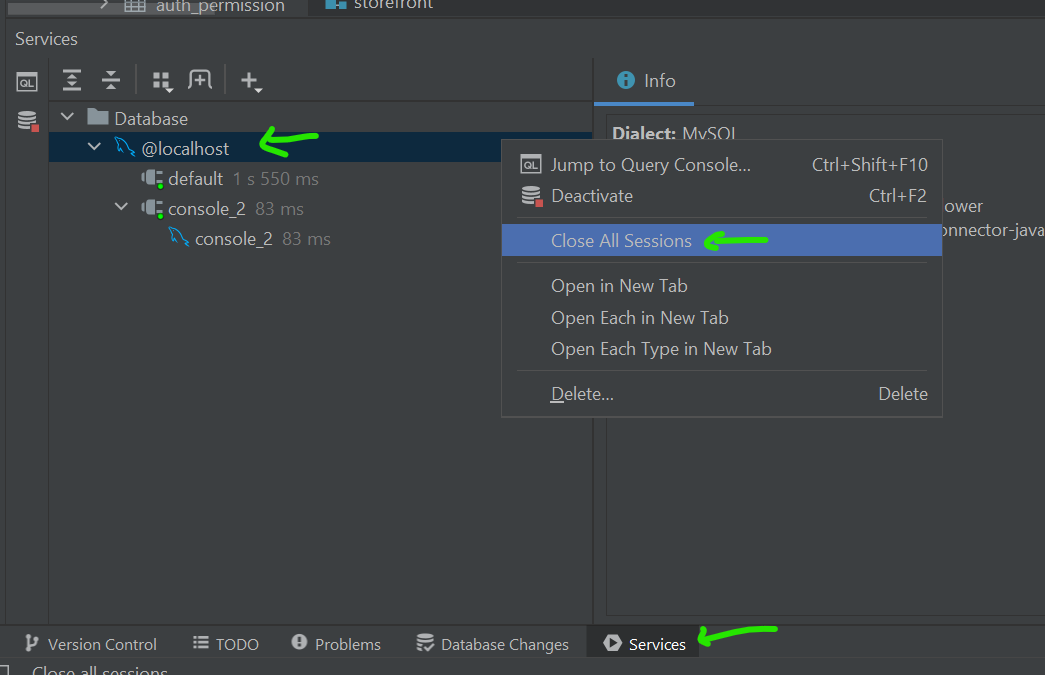
1. Write this query in new console.

CREATE DATABASE storefront

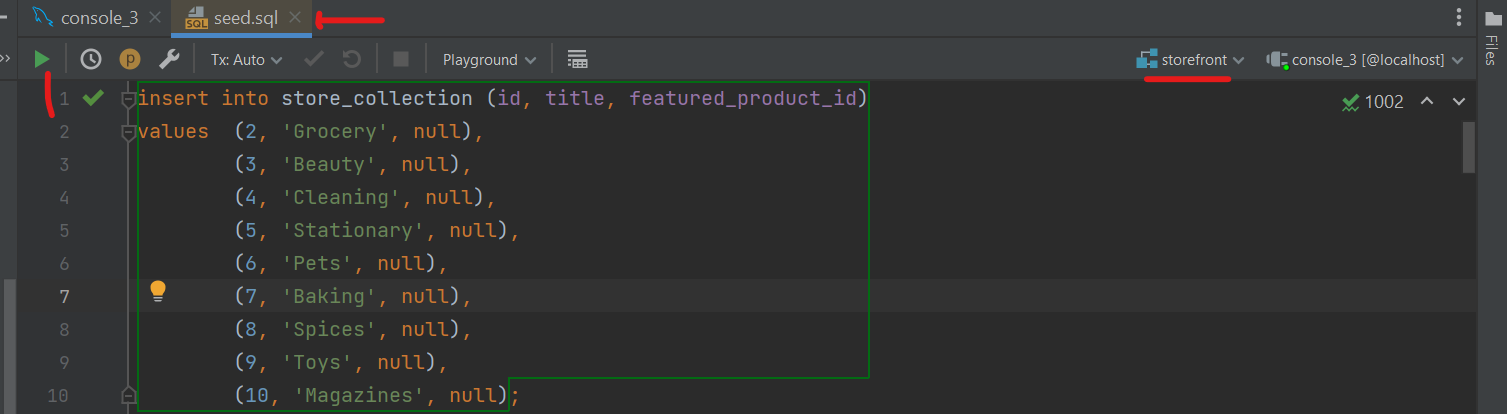
1. It will create a new database with the same name and to populate it run python manage.py migrate command to populate this database.

(make sure you spell database name in DATABASES setting properly, otherwise it will not work)

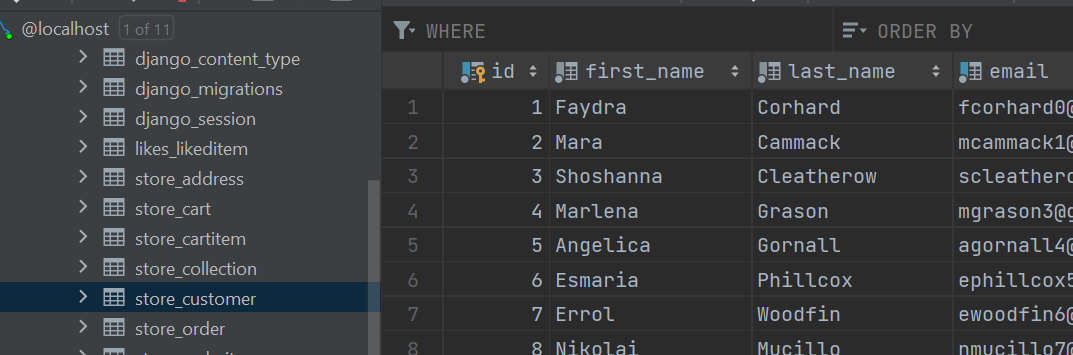
Note: To close all the sessions in DataGrip, go to sessions🡪localhost🡪Right click to open Close all Sessions



1. Drag and drop the seed SQL file into the window and execute all the queries at once. Make sure schema selected is *storefront*.



1. Now we can see records populated in our tables.



**Managers and QuerySets**:

To understand the concept of managers and QuerySets, let us go to our playgrounds app views module.

from django.shortcuts import render

from django.http import HttpResponse

def say\_hello(request):

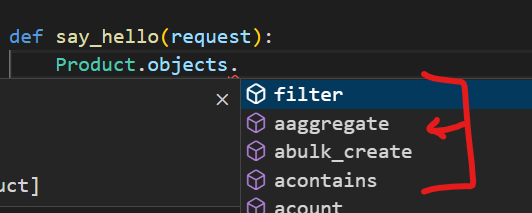
    return render(request, "hello.html", {"name": "Himanshu"})

Here we see our *say\_hello* function returning a simple HTTP response.

1. On the top import *Product* class from store.model.

Note: Every model in Django has an attribute called objects, which returns a manager object. A manager is like an interface to the database. *It is like a remote control with bunch of buttons we can use to talk to database*.

1. We can see a bunch of methods for querying or updating data. Like all() method for pulling all the objects in the *Product* table, get() to get a single object, filter() for filtering data and so on…



1. Most of these methods like all(), returns a QuerySets so when we call Product.objects.all() we do not get a list of products instead we get a QuerySet object.

Note: A QuerySet is an object that encapsulates a query. *At some point Django will evaluate this query set and generate correct SQL statement to send to our database*.

This will happen under these scenarios:

🡪First scenario is, when we iterate over a query set like below

def say\_hello(request):

    query\_set = Product.objects.all()

    for product in query\_set:

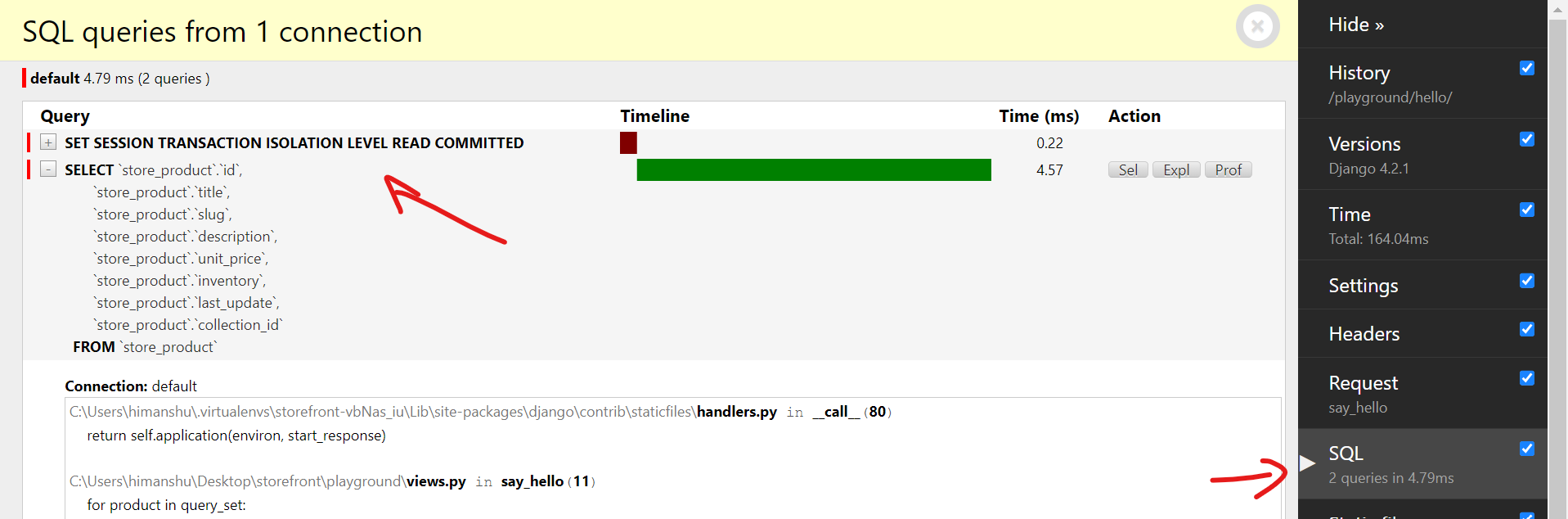
        print(product)

    return render(request, "hello.html", {"name": "Himanshu"})

To see product on our browser window, go to

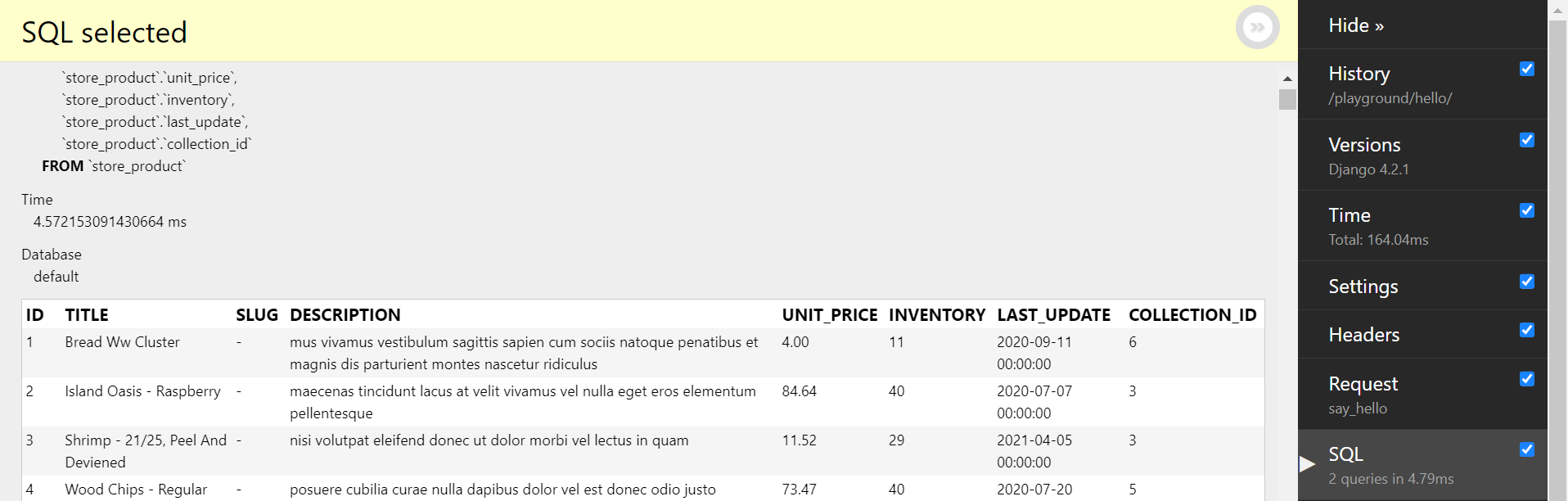
<http://127.0.0.1:8000/playground/hello/>

and open the SQL tab in Django debug toolbar



Here we see the actual SQL statements that Django sent to our database.(First query used by Django to do its job)

Our main query is the SELECT statement to get all the fields from the products table. Click on the SEL button to see the actual result that is returned from the database.



🡪Second scenario is when we convert a query set into a list.

list(query\_set). So when we call list function and give it the query set, it will be evaluated.

🡪Third scenario is when we access an individual element like first element query\_set[0] or slice this query\_set[0:5] and pick the first 5 elements then the query\_set will be evaluated.

Due to this behavior we say query sets are lazy, which means they are evaluated at a later point.

Note: Query sets are originally lazy and *Django does not simply call the database the moment we call all() method* is because we use query set methods to build complex queries.

For example we can use filter method to write large queries…

query\_set.filter().filter().order\_by()

first time we use filter it already returns a new query set but it is not going to be evaluated at this point. So right away we called filter method again to apply a second filter and lastly we called order\_by() method to sort the result.

*“So by chaining all these methods we can build a complex query and at some point when we iterate over that query or convert it to a list that query will be evaluated. If query sets were not lazy the moment we call all() method query will be evaluated”*. That will result in thousands or even millions of objects returned from the database.

**Summary**:

🡪Every model has an attribute called objects which returns a manager

🡪manager is an interface to the database which has a bunch of methods for querying and updating data.

🡪Most of the methods return a query set which is evaluated by iteration or converting to list.

Note: We do not get query set from manager methods like count() which returns number of total objects in database. Since it is a number there is no point applying any new methods like filter or do sorting on it.

**Retrieving Objects**:

In this lesson, we will learn about few different methods for retrieving objects.

In order to get a single object like the product with id = 1, we use get method and pass a special lookup parameter like *pk=1*.

Note: When we use *pk*, Django will automatically translate this to name of the primary key field (It might be *ID* or *SKU* anything). This is the benefit of using *pk* argument.

Get method returns an actual object not a query set.

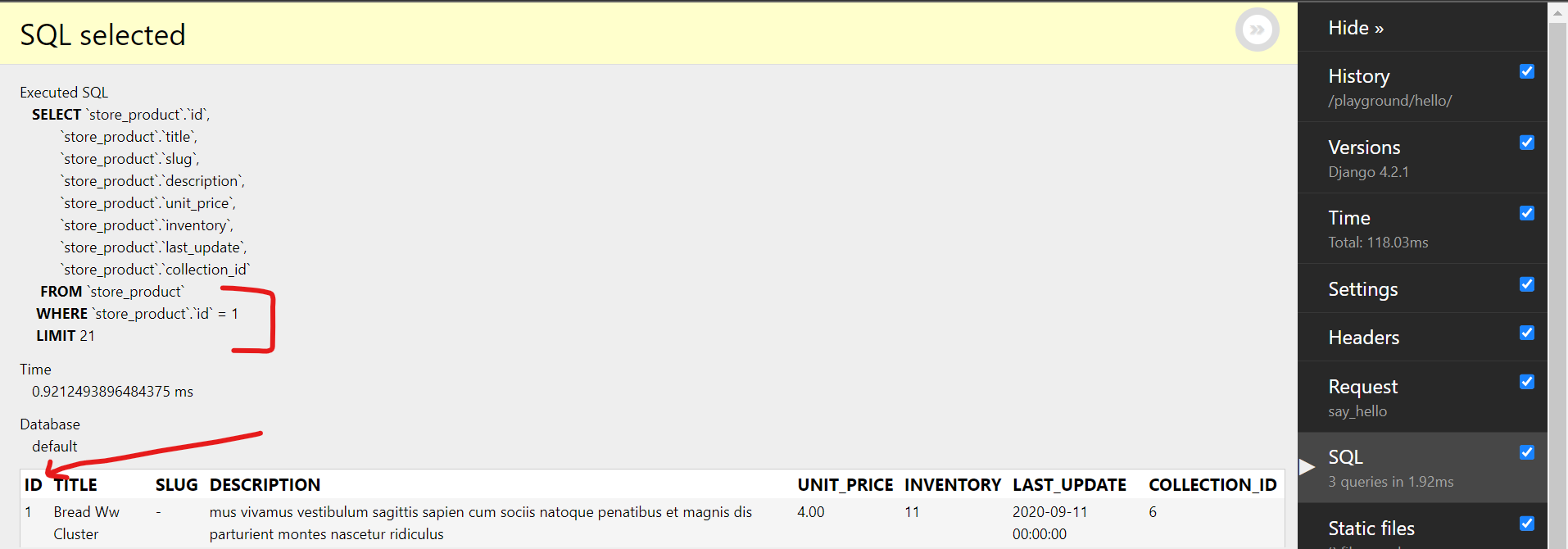
def say\_hello(request):

    product = Product.objects.get(pk=1)

    return render(request, "hello.html", {"name": "Himanshu"})

here *product* is an object that we get from get method.

In the browser we can see SQL query



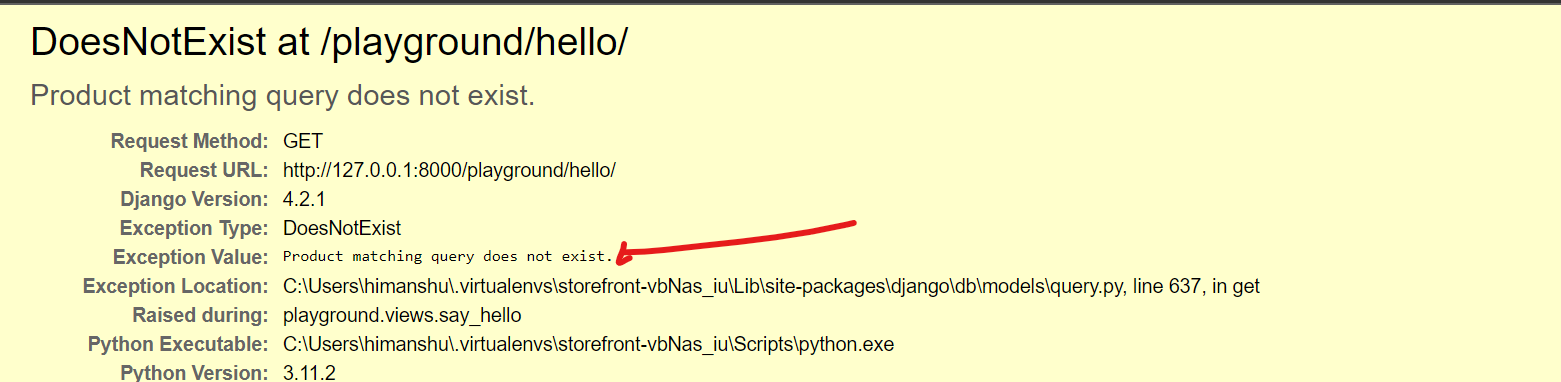
We are getting all the columns from *Product* table where ID = 1

**Handling exception**:

If get method is unable to find an object in our database, it will throw an exception

 product = Product.objects.get(pk=0)

After setting pk=0, we see an exception in browser



We can handle this exception in multiple ways:

1. Use try-catch block

from django.shortcuts import render

from django.core.exceptions import ObjectDoesNotExist

from store.models import Product

def say\_hello(request):

    try:

        product = Product.objects.get(pk=0)

    except ObjectDoesNotExist:

        pass

    return render(request, "hello.html", {"name": "Himanshu"})

1. Using filter and first method

def say\_hello(request):

    product = Product.objects.filter(pk=0).first()

    return render(request, "hello.html", {"name": "Himanshu"})

Instead of get we use filter method which returns a query set. We chain the query set with first method. If the query set is empty it returns none(*product is none*).

1. We can also use exists method to see if an object exist or not(*it returns a Boolean*).

  product\_exist = Product.objects.filter(pk=0).exists()

**Filtering Objects**:

Suppose we want to find all the products that are $20, so in filter method we can pass a keyword argument and say unit\_price=20

query\_set = Product.objects.filter(unit\_price=20)

It is pretty straightforward.

But if we want to find all the products that are more expensive than $20, we cannot use logical operators like > or <= here, because *unit\_price>20* is a Boolean expression(return either true or false)

Note: Filter method accepts keyword arguments to evaluate a query. After the *field name* type double underscores \_\_ followed by *lookup type*.

FieldName\_\_LookupType = value

You can find lookup types by searching for *Field Lookups in QuerySet API reference* document.

We have various lookup types for numbers, strings and dates.

🡪For greater than and greater than equal to use \_\_gte

query\_set = Product.objects.filter(unit\_price\_\_gte=20)

will return query set of all containing products with price greater than equal to $20.

🡪for query set between two values use \_\_range()

query\_set = Product.objects.filter(unit\_price\_\_range=(20, 30))

Note: we can render query set in our HTML by passing key value pair in dictionary like below

def say\_hello(request):

    query\_set = Product.objects.filter(unit\_price\_\_range=(20, 30))

    return render(

        request, "hello.html", {"name": "Himanshu", "products": list(query\_set)}

    )

Then we go to hello.html file and add an unordered and ordered list this way

<html>

  <body>

    {% if name %}

    <h1>Hello {{name}}</h1>

    {% else %}

    <h1>Hello World</h1>

    {%endif%}

    <ul>

      {%for product in products%}

      <li>{{product.title}} price is <b>${{product.unit\_price}}</b></li>

      {%endfor%}

    </ul>

  </body>

</html>

Finally, We see the results



🡪We can also filter objects across relationships

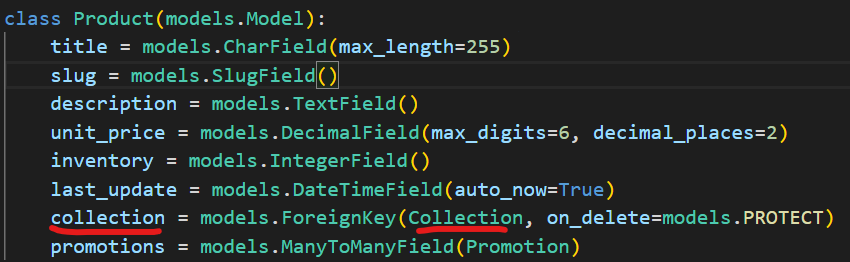
class Collection(models.Model):

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

    featured\_product = models.ForeignKey(

        "Product", on\_delete=models.SET\_NULL, null=True, related\_name="+"

    )



Here Collection model has a relationship with Product model. Let us assume we want to find all the products in collection #1

So we type collection followed by two underscores \_\_ (*to navigate the relationship*) followed by name of attributes or field like id of the Collection class.

*collection\_\_id=1*

We can further add a lookup type like gt or range,

*collection\_\_id\_\_range=(1,2,3)*

**Example**(lookup for string):

We want to find products that contain coffee in their title.

Solution:

def say\_hello(request):

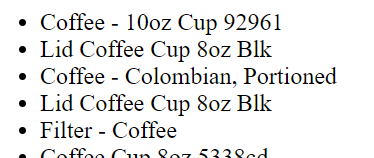
    query\_set = Product.objects.filter(title\_\_contains="coffee")

Using contains simply does not return anything on browser. The reason is that this contains lookup type is case-sensitive.

So to make it case-*insensitive* add an i before contains.

def say\_hello(request):

    query\_set = Product.objects.filter(title\_\_icontains="coffee")



Note: We also have startswith, endswith lookup types with case insensitive variation.

**Exercise**(lookup for date):

We want to find all the products that were updated in 2021.

def say\_hello(request):

    query\_set = Product.objects.filter(last\_update\_\_year=2021)

Note: We can extract individual components from Date like year, month, minute, second and so on. We can also compare it with a date value.

**Exercise**(lookup for null):

All the products without the description

def say\_hello(request):

    query\_set = Product.objects.filter(description\_\_isnull=True)

**Filtering Exercises**:

1. Customers with .com accounts

def say\_hello(request):

    query\_set\_customer = Customer.objects.filter(email\_\_icontains=".com")

    return render(

        request,

        "hello.html",

        {"name": "Himanshu", "customers": list(query\_set\_customer)},

    )

In .html file

    <ul>

      {%for customer in customers%}

      <li>{{customer.first\_name}}'s email ID is : {{customer.email}}</b></li>

      {%endfor%}

    </ul>

1. Collections that don’t have a featured\_product

query\_set\_collection = Collection.objects.filter(featured\_product\_\_isnull=True)

1. Products with low inventory(less than 10)

    query\_set = Product.objects.filter(inventory\_\_lt=10)

1. Orders placed by customer with id=1

def say\_hello(request):

    query\_set\_order = Order.objects.filter(customer\_id=1)

    return render(

        request,

        "hello.html",

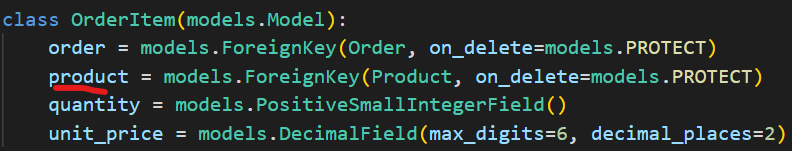
        {"name": "Himanshu", "orders": list(query\_set\_order)},

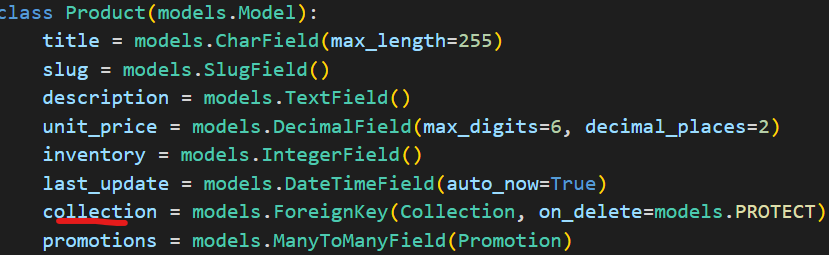
    )

1. Order items for products in collection 3

Solution:

Means The *product* with *collection\_id* = 3 in *OrderItem* Table.







def say\_hello(request):

    query\_set\_order\_item = OrderItem.objects.filter(product\_\_collection\_id=3)

    return render(

        request,

        "hello.html",

        {"name": "Himanshu", "order\_items": list(query\_set\_order\_item)},

    )

**Complex Lookups using Q objects**:

In this we will see how to apply multiple filters.

For example, We want to find all the products with inventory less than 10 AND unit price less than 20



We can pass multiple keyword arguments in same filter method, like this…

query\_set = Product.objects.filter(inventory\_\_lt=10, unit\_price\_\_lt=20)

OR we can pass multiple filter methods and chain them together like this,

query\_set = Product.objects.filter(inventory\_\_lt=10).filter(unit\_price\_\_lt=20)

*We get the same result two conditions in* WHERE *clause separated by* AND *operator*.

In case we want to use OR operator to combine these conditions, we need to use **Q** objects.

🡪First import Q class from Django

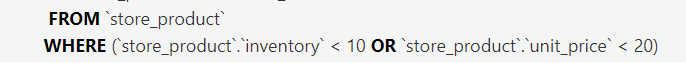
from django.db.models import Q

Q is short for query. Using this class we can represent query expression or *a piece of code that produces a value*.

🡪Using this Q class we can encapsulate a keyword argument like *Q(inventory\_id=1)* which translates to a *query expression*.

🡪Now, We can combine this query expression with another query using bitwise | (or) operator.

query\_set = Product.objects.filter(Q(inventory\_\_lt=10) | Q(unit\_price\_\_lt=20))



Note: To use NOT operator with Q objects use **~** symbol to negate the expression.

query\_set = Product.objects.filter(Q(inventory\_\_lt=10) & ~Q(unit\_price\_\_lt=20))

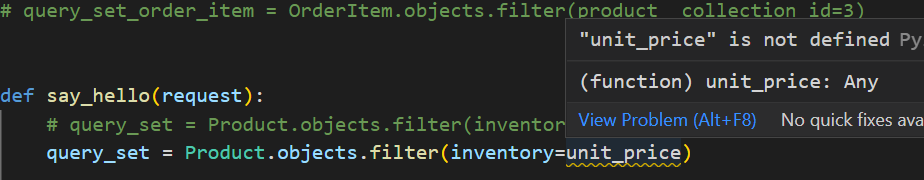


**Referencing fields using f objects**:

Sometimes when filtering data, we need to reference a particular field. For example, we want all the products where their inventory equals their unit price.

This problem does not make sense from Business POV, but the idea behind this is that what should we do, when we need to compare two fields.

We cannot simply do this…

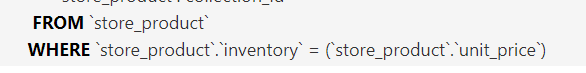


To solve this problem, we need to use a F object.

from django.db.models import Q, F

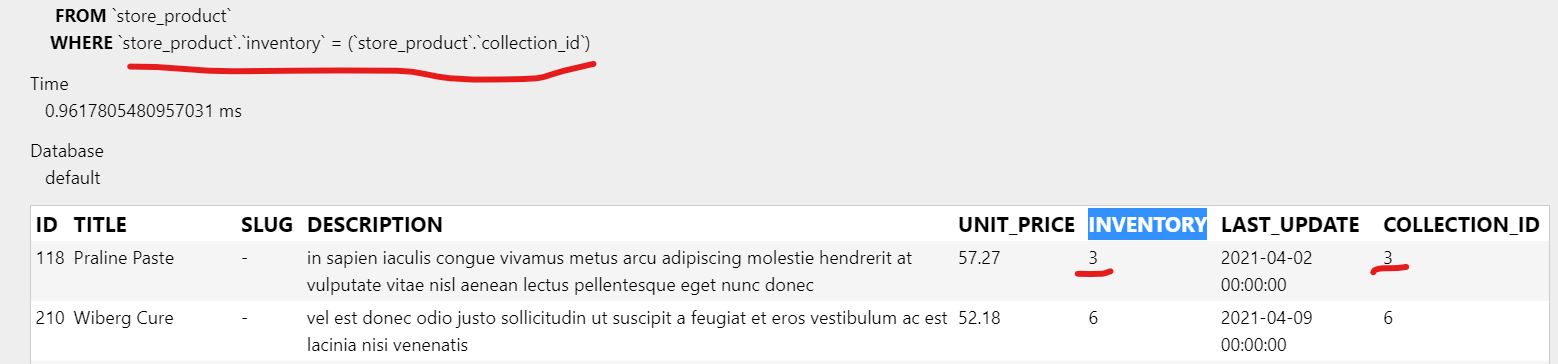
We import **F** class from the same models module. *Using this class we can reference a particular* ***field****(that is why it is called* ***F****)*.

query\_set = Product.objects.filter(inventory=F("unit\_price"))



Note: Using F object we can also reference a field in a related table.

query\_set = Product.objects.filter(inventory=F("collection\_id"))



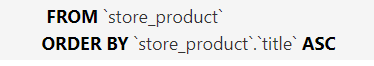
**Sorting**:

We already know about order\_by method with which we can sort the results by one or more field.

**Example 1**:

Arrange all products as per their title in ascending order

 query\_set = Product.objects.order\_by("title")



**Example 2**:

Arrange all products as per their title in descending order

Use **–** before the field name

query\_set = Product.objects.order\_by("-title")



**Example 3**:

Sort by unit price in ascending order and title in descending order

query\_set = Product.objects.order\_by("unit\_price", "-title")



Here by using query in this form *we are sorting our products from cheapest to most expensive and if we have few products which have the same price they will be sorted as per their title’s alphabetical order.*

Note:

1. order\_by method returns a query set object and one of the methods of query set is reverse. This will reverse the original direction of the sort.

 query\_set = Product.objects.order\_by("unit\_price", "-title").reverse()

It will sort products with price in descending order and title in ascending order.



1. We can also apply order\_by on filter method, because filter returns a query set object and order\_by is one of the methods of a query set object.

query\_set = Product.objects.filter(unit\_price\_\_lt=2).order\_by("unit\_price")

or

    query\_set = Product.objects.filter(collection\_id=6).order\_by("title")

Note: To get an individual object, like first element after sorting we can use square braces [0]

def say\_hello(request):

    query\_object = Product.objects.filter(collection\_id=6).order\_by("title")[0]

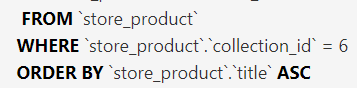
    return render(

        request,

        "hello.html",

        {"name": "Himanshu", "object": query\_object},

    )



Other way to write the same query, is by calling earliest method to sort the result in *ascending order* by title and get the first object.

query\_object = Product.objects.earliest("title")

Similarly we have latest method which returns the first object after sorting in *descending order*.

query\_object = Product.objects.latest("title")

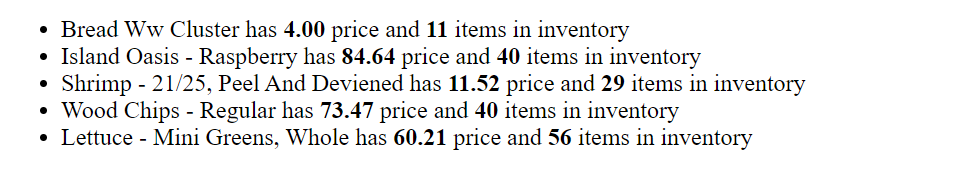
**Limiting Results**:

So our product table has 1000 products , but most often we do not want to show all these products to the user in one list. *Instead we want to show* *pages of products*.

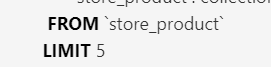
Let us say our page size is 5 and we want to show products on that page. To do that we use python’s array slicing syntax.

query\_set = Product.objects.all()[:5]

We see the first 5 items at indices 0,1,2,3 and 4 excluding 5th



In SQL tab



For next 5 products,

query\_set = Product.objects.all()[5:10]

We see the next items from 5,6,7,8 and 9



Notice, we have additional clause called offset in SQL query to skip last 5 objects.

**Selecting Fields to Query**:

When we query objects by default all their fields are read from the database.

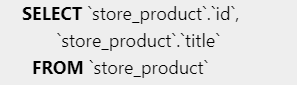


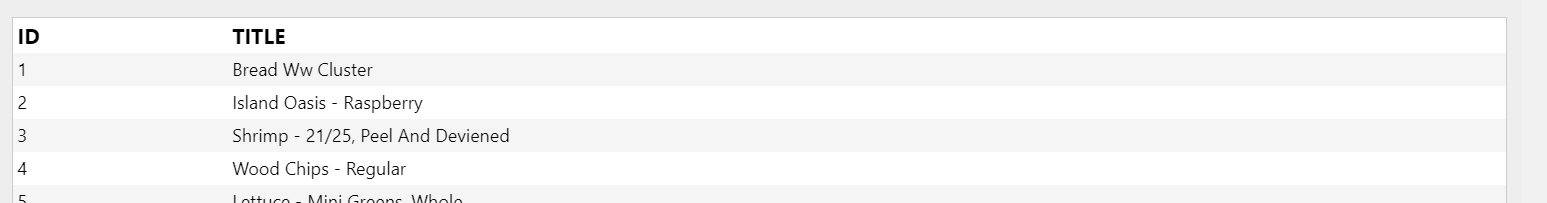
But what if we are interested only in a subset of these fields.

Some of these columns like the description column contains lot of text, what if we do not care about the values in this column and *only want to return the title and id of each product to user*. To do this we use values method.

query\_set = Product.objects.values("id", "title")

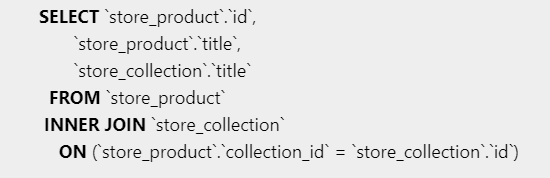
Here we use values method and specify the fields as arguments that we want to query.

🡨Only selecting id and title columns



We can also read related fields like collection

query\_set = Product.objects.values("id", "title", "collection\_\_title")

 used **\_\_** to access related field.

Notice that we have an inner join between product and collection table because we are reading a related field.

Note: With this method instead of getting a bunch of product instances, we get a bunch of dictionary objects.

Just simply print product object itself in html

    <ul>

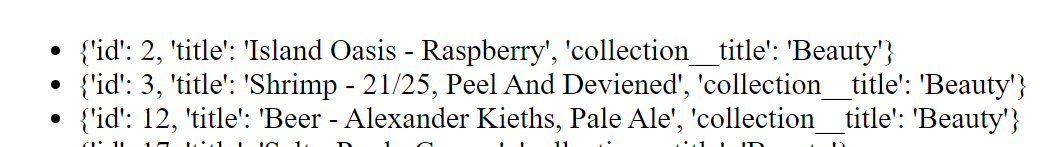
      {%for product in products%}

      <li>{{product}}</li>

      {%endfor%}

    </ul>

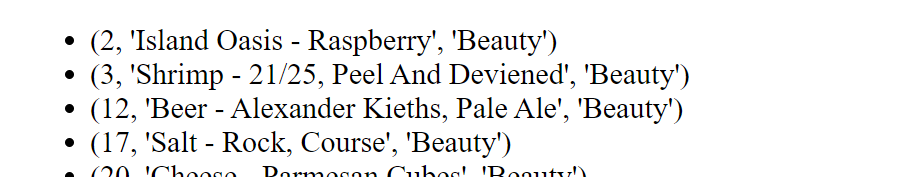
In browser



So we see each object in the result is a dictionary not a product instance. We see three key-value pairs *id, title and collection\_\_title*.

We have another method called values\_list, with which we get bunch of tuples instead of dictionaries.

query\_set = Product.objects.values\_list("id", "title", "collection\_\_title")



Now each object is a tuple of three values

Note: In SQL Inner Join keyword reflects records that have matching values in both tables.

**Exercise**:  
Select products that have been ordered and sort them by title.

Hint: In our database we have a product table that contains all the products, but not all products have been ordered. To find the products that have been ordered go to OrderItem table and navigate to product\_id table to see all the ordered product IDs. *Select all these product IDs and use them as an argument when filtering product*.

Solution:

First we need to list all product IDs in OrderItem table

query\_set\_order\_item = OrderItem.objects.values("product\_id").distinct()

with values we get all the values in product\_id column and with distinct we can remove the duplicates.

Now we pass this query set into Product table

query\_set\_product = Product.objects.filter(id\_\_in=query\_set\_order\_item)

Note: We use in lookup type to get only the records which are in id column



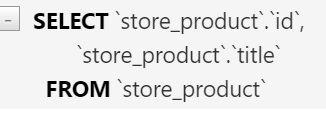
**Deferring Fields**:

In last lesson we learned how we can optimize our queries by specifying the fields that we want to query. We have another technique called *deferring fields*.

We use a method called only and with this we can specify the fields that we want to read from the database.

queryset = Product.objects.only("id", "title")

In syntax, It looks similar to values method. But actually with only method *we get instances of the Product class*, whereas with values method *we get dictionary objects*.

🡨Query also looks same

Note: we have to be very careful with only method, because we might accidently send many queries to the database under the hood.

Let us render the price of each Product as well along with title.

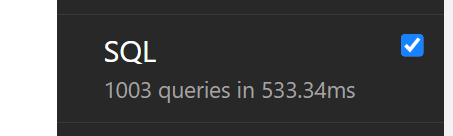
    <ul>

      {%for product in products%}

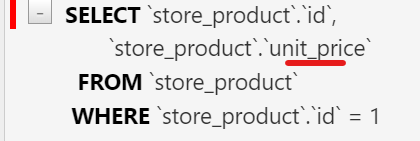
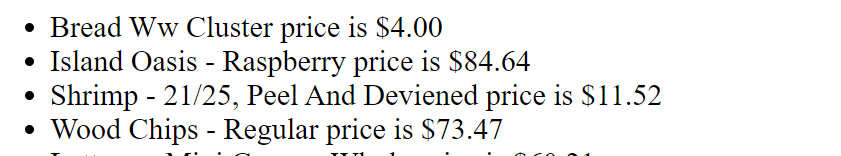
      <li>{{product.title}} price is ${{product.unit\_price}}</li>

      {%endfor%}

    </ul>

🡨1000 + queries!!

After the main query, we have 1000 individual queries to read their individual unit price

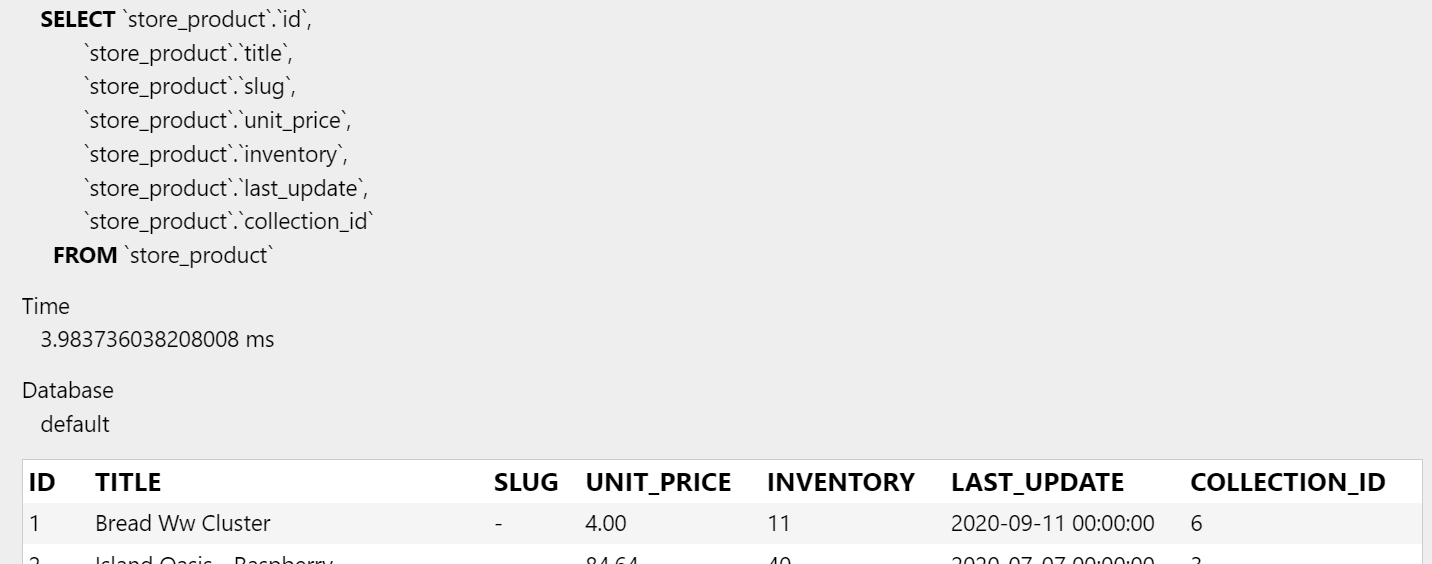
 

Values method do not have this issue, because dictionary objects do not show this type of behavior.

Defer method:

Opposite of only method and With this we can *defer the loading of certain fields to later*.

Suppose we want all columns of Product table except description field, so using defer method we can defer that field to later.



**Selecting Related Objects**:

Sometimes we need to preload a bunch of objects together. For example in this query set we are loading all the products.

queryset = Product.objects.all()

Now, let us go to html file and render the name of collection title for each individual product.

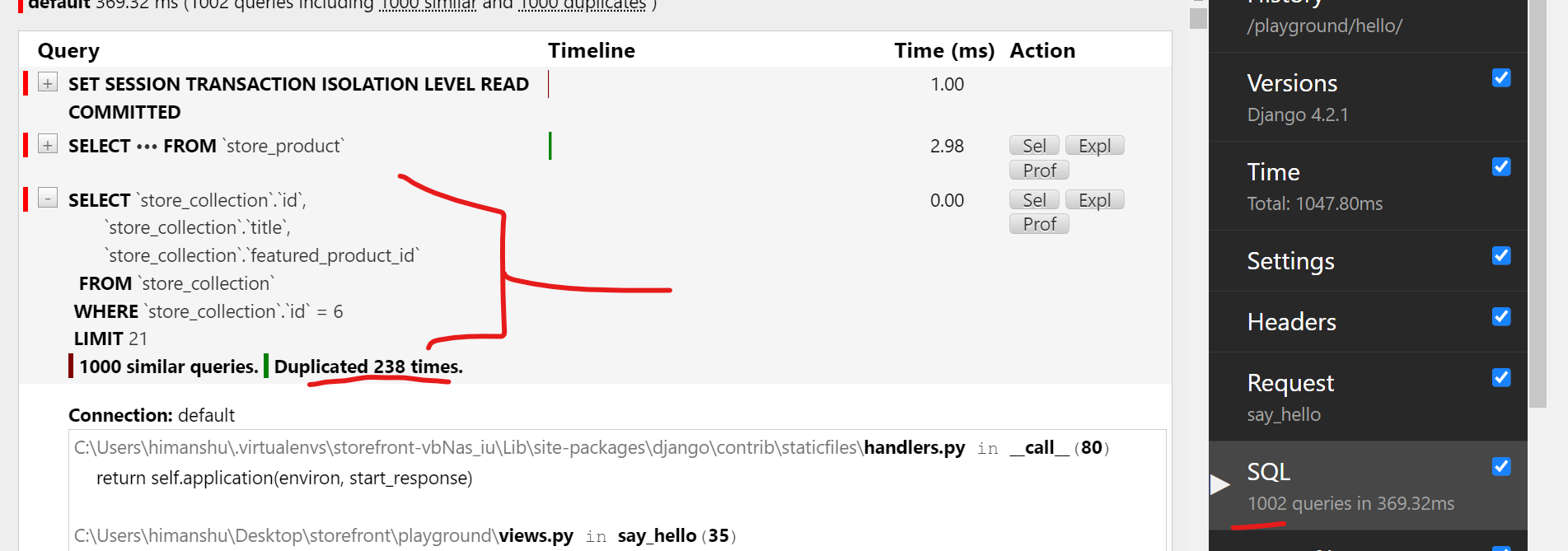
 <ul>

      {%for product in products%}

      <li>{{product.title}} -- {{product.collection.title}}</li>

      {%endfor%}

    </ul>



We see Django is sending 1000 extra queries to the database to read the collection of all these products.

It is because when we use all method, Django is only going to query product table, *it is not going to query related tables unless we specifically instruct it to do so*.

So in our case, we want to preload products with our collection. To do this before calling all method, we call select\_related method.

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

Now, we see our page loads instantly and only three queries in SQL tab.

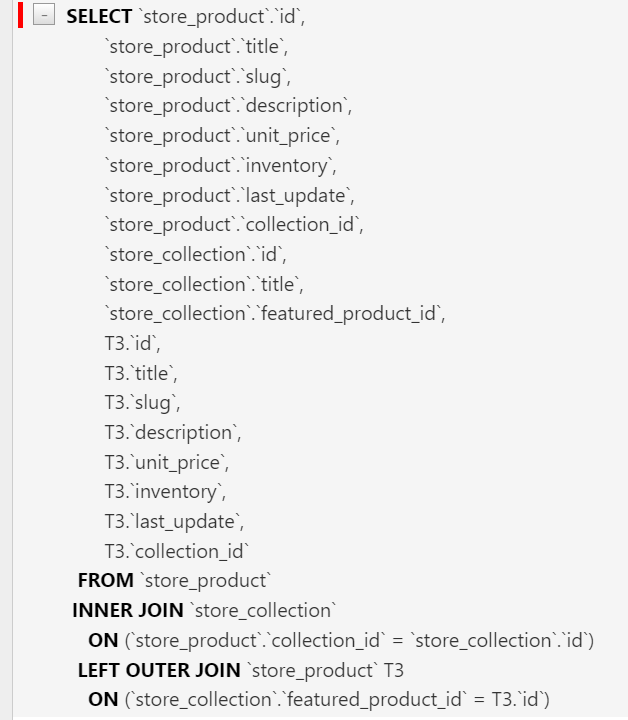


First we see all the fields of *product* table and *collection* table in SELECT statement followed by INNER JOIN of both tables.

So, when we use select\_related method Django creates a join between our tables.

Note: We can also span relationships, for example if we *want to preload a specific field in collection table*, we can do this,

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



**prefetch\_related**:

As *we use* select\_related *when the other end of the relationship has one instance like a Product is inside one collection*.

But we use prefetch\_related when the other end of relationship has many objects. For example promotions of a product.

class Product(models.Model):

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

    slug = models.SlugField()

    description = models.TextField()

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

    inventory = models.IntegerField()

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

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

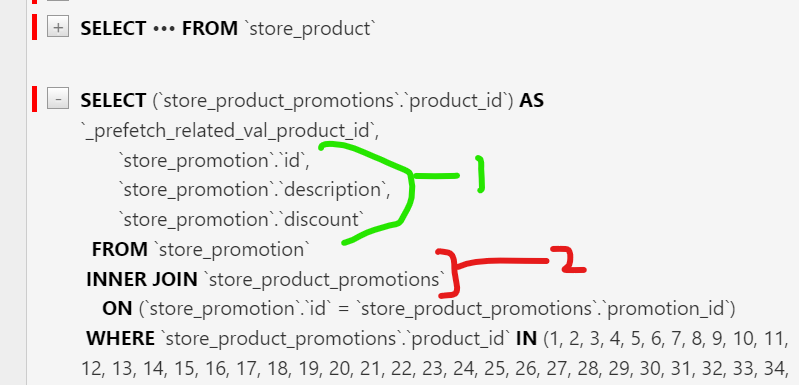
    promotions = models.ManyToManyField(Promotion)

Notice, Product class has one collection but it has many promotion.

So to preload the promotions, we use the prefetch method and field that we want to preload here is *promotions*.

queryset = Product.objects.prefetch\_related("promotions").all()

Now we have two queries first to read all fields for product table and second query to read the promotions for this product.



In this second query *we are reading three fields from promotions table and then inner joining them with product table*.

Note: We can also combine select\_related and prefetch\_related method in a query, suppose we want to load all the products with their promotions and respective collection.

   queryset = (

        Product.objects.prefetch\_related("promotions")

        .select\_related("collection")

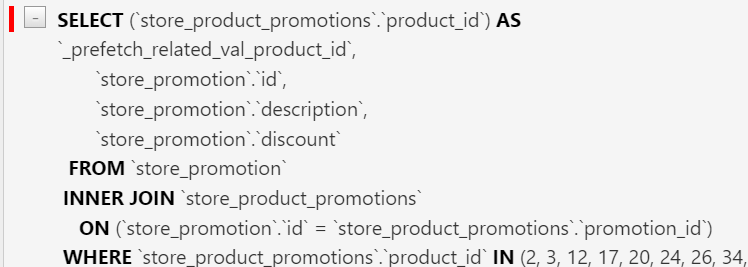
        .all()

    )

Sequence of calling these functions does not matter.

In SQL tab we see two queries

🡨We see first query as selecting all product and collection fields and joining them.



🡨Second query is to read promotions of all the products.

**Exercise**:

Get the last 5 orders with customer and items (including product referencing in each order item)

Solution:

1. Since we want to get the list of orders, therefore we first start with Order class and preload it with customers so we use select\_related.
2. Next we want to see last 5 orders, so we use order\_by with placed\_at field in descending order and limit result using python slice operator

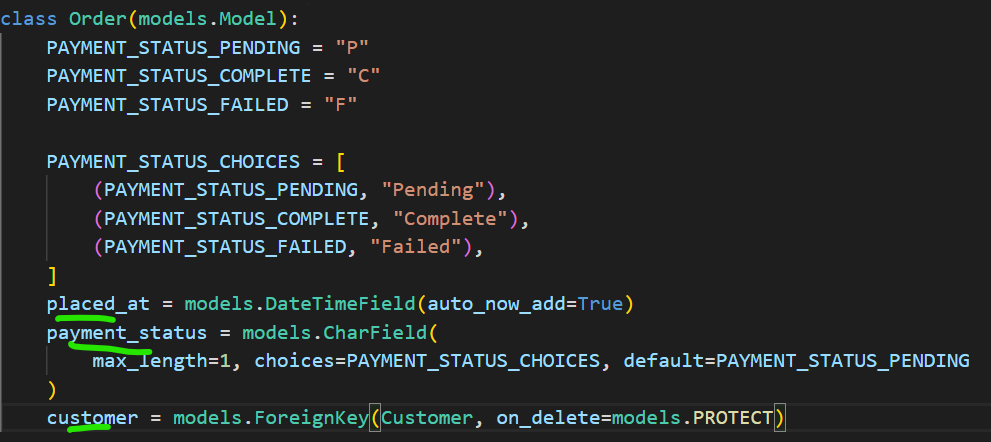
From first two points we get a query set like this.,

queryset = Order.objects.all().order\_by("-placed\_at")[:5].select\_related("customer")



1. Now we should preload the items for these orders, so we call prefetch\_related *because each order can have multiple items*. The main question is what is the name of the field that we will query.

For this look into order class



Here we have only three fields, placed\_at, payment\_status and customer we are not seeing any order item field. On the other hand look at OrderItem class

class OrderItem(models.Model):

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

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

    quantity = models.PositiveSmallIntegerField()

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

Here, we have order which is a foreign key to Order. So Django will create a reverse relationship and the name will be

**“orderitem\_set”** (name of target class in lowercase followed by \_set). This is the *convention Django uses to create a reverse relationship*.

Note: we can change the name of field created by reverse relationship by using related\_name argument.

order = models.ForeignKey(Order, on\_delete=models.PROTECT, related\_name="items")

1. Back to our query, we want to prefetch orderitem\_set

  queryset = (

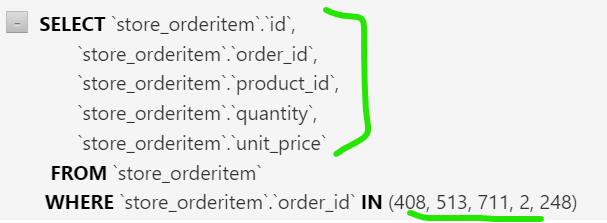
        Order.objects.all()

        .order\_by("-placed\_at")[:5]

        .select\_related("customer")

        .prefetch\_related("orderitem\_set")

    )



we get one extra query to load orderitem fields for given order id’s

1. Last step is to load the product referenced in each order, so span the relationship by adding \_\_ followed by product

    queryset = (

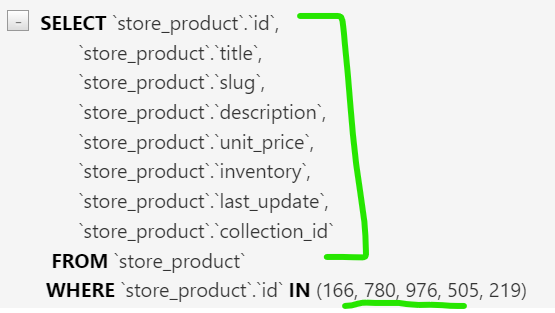
        Order.objects.all()

        .order\_by("-placed\_at")[:5]

        .select\_related("customer")

        .prefetch\_related("orderitem\_set\_\_product")

    )



Now we have one more query to read products referenced from given order items.

**Aggregating Objects**:

Sometimes we want to compute summaries like max or average price of our products. This is where we use the aggregate method.

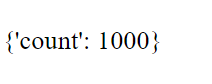
First we need to import it from Django

from django.db.models.aggregates import Max, Min, Count, Sum, Avg

Example:

Count number of objects in a table.

result = Product.objects.aggregate(count=Count("id"))

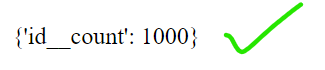


**Issue encountered**:

I used *aagregate* method instead of *aggregate* and got very different output.

result = Product.objects.aggregate(Count("id")) 🡪 Get dictionary **OK**

result = Product.objects.aaggregate(Count("id")) 🡪 Got object NOT **OK**



Note: If we do not specify any variable name(like we used *count=*) where we want to store count value,

result = Product.objects.aggregate(Count("id"))

We will get,



Example:

Find minimum price from list of prices

result = Product.objects.aggregate(min\_price=Min("unit\_price"))



Note: we get dictionary and not a query set from aggregate. But we can apply the aggregate on a query set obtained by methods like filter.

**Aggregate Exercises**:

Exercise 1:

1. How many orders do we have

result = Order.objects.aggregate(count=Count("id"))

1. How many units of product id: 1, we have sold

 result = OrderItem.objects.filter(product\_id=1).aggregate(

        units\_sold=Sum("quantity")

    )

1. How many *orders* has customer id: 1 placed?

result = Order.objects.filter(customer\_id=1).aggregate(count=Count("id"))

1. What is the min, max and average price of the *products* in collection id: 3?

    result = Product.objects.filter(collection\_id=3).aggregate(

        min\_price=Min("unit\_price"),

        max\_price=Max("unit\_price"),

        avg\_price=Avg("unit\_price"),

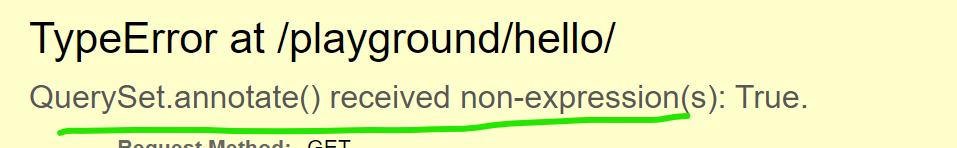
    )

**Annotating Objects**:

Sometimes we want to add additional attributes to our objects while querying them. This is where we use annotate method.

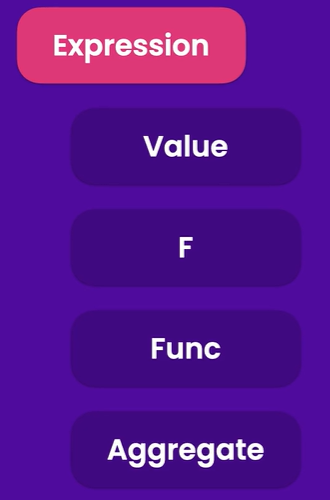
Let us say while querying Customer table, we want to give each customer a new field called is\_new and we want to set it to True.

queryset = Customer.objects.annotate(is\_new=True)



We get the error *QuerySet.annotate() received non-expression(s)*

This error is telling us that *we cannot pass a Boolean value to is\_new variable instead we should pass an* **expression object**.

🡨In Django we have an expression class which is the base class for all types of expressions.

**Derivatives of this class are**:

🡪Value (for representing simple values like a number, Boolean, string)

🡪F (we have seen before, Using F class we can reference field in the same or another table). F object is essentially an expression object.

🡪Func (Used for calling Database functions. Since all the DB engines have a bunch of functions for manipulating data). To call those functions we use the Func object.

🡪Aggregate (Base class for all aggregate classes. The aggregate classes that we learned about like Max, Min, Count they are all derived from Aggregate class)

Since we cannot pass Boolean value directly, we need to wrap it inside Value class, so we first import it from models module.

from django.db.models import Value

and then,

queryset = Customer.objects.annotate(is\_new=Value(True))

No errors now, and we can see our query now, with newly added field, populated with 1 which is numeric equivalent of Boolean value True.



**Example**:

Give Customer a new field called new\_id and set its value equal to id field.

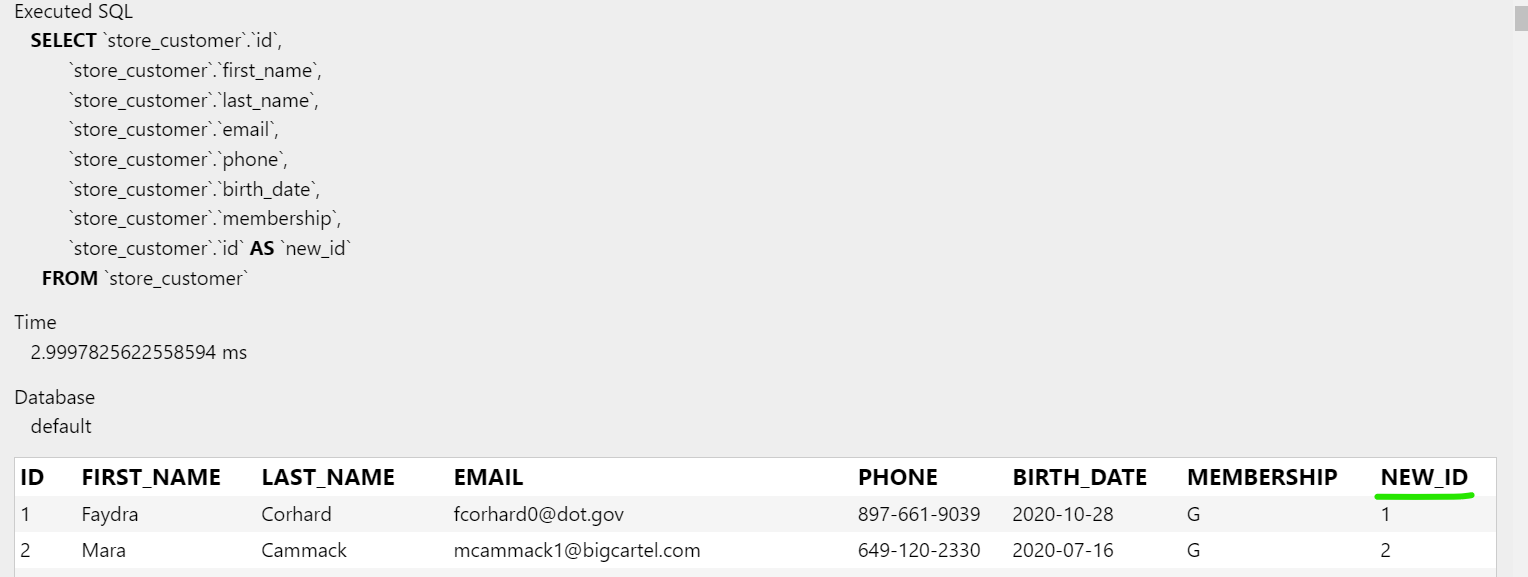
Solution:

First we need to import F class

from django.db.models import Value, F

Then we can reference the id field,

queryset = Customer.objects.annotate(new\_id=F("id"))



Note: we can also perform computations here like incrementing value of new\_id by 1

queryset = Customer.objects.annotate(new\_id=F("id") + 1)



**Calling Database Functions**:

Let us now call a database function. From the same module first pass Func class.

from django.db.models import Value, F, Func

**Example**:

Give our Customer table a new field called full\_name where you will store CONCAT result of two fields first\_name and last\_name.

Solution:

🡪Pretty much all databases has a function for concatenating strings. So we use Func *object* for calling that function.

queryset = Customer.objects.annotate(full\_name=Func())

🡪Inside Func object we *reference* first\_name and last\_name fields inside a F *object*.

  queryset = Customer.objects.annotate(

        full\_name=Func(F("first\_name"), F("last\_name"))

    )

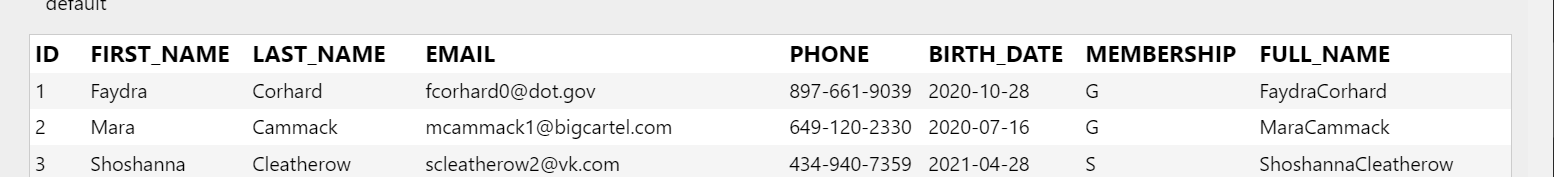
🡪Next we need to give it a *keyword argument* for specifying target function, which in our case is CONCAT.

 queryset = Customer.objects.annotate(

        full\_name=Func(F("first\_name"), F("last\_name"), function="CONCAT")

    )

We run our query and see this,



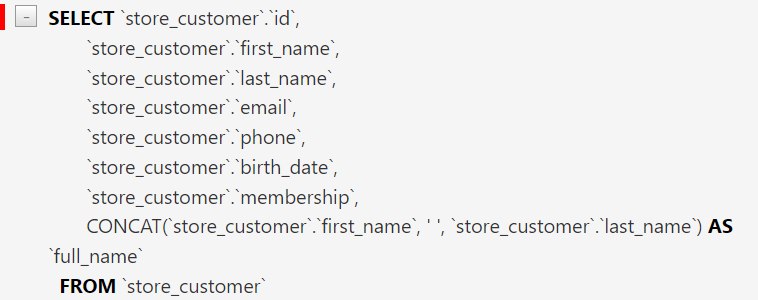
We get a new table called full\_name but there is no spaces between first\_name and last\_name

🡪So to add a space we use Value object. *We cannot simply pass an empty string in Func(cause it will take it as a field with no name)*. So our code finally will look like this,

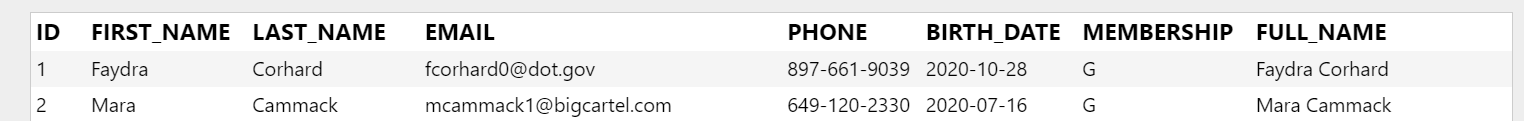
queryset = Customer.objects.annotate(

        full\_name=Func(F("first\_name"), Value(" "), F("last\_name"), function="CONCAT")

    )



Here we have all the Customer columns, but we also have one more column returned by CONCAT function.



Another Solution:

There is also a shorthand to achieve the same result. For this first we need to import Concat class from Django.

from django.db.models.functions import Concat

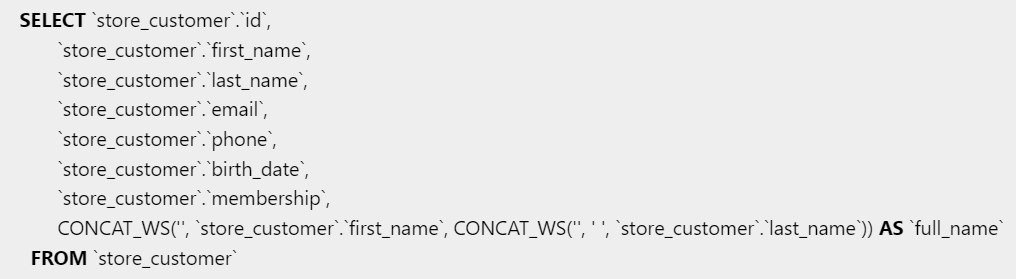
Now we can use Concat class instead of Func class and specify our fields same way as before.

    queryset = Customer.objects.annotate(

        full\_name=Concat("first\_name", Value(" "), "last\_name")

    )

Note: While using Concat class we do not have to wrap our field names inside F objects. But for introducing white space, we still need to use Value object.



Note: If you are curious about Django database functions refer to <https://docs.djangoproject.com/en/4.2/ref/models/database-functions/>

we will find here we have a bunch of functions for working with Dates, mathematical functions, functions for manipulating texts and so on. These functions are common across all Database engines, but *some database engines have their own unique functions*.

*To call those functions, we need to use* Func *object*.

**Grouping Data**:

Let us say, we want to see number of orders each customer has placed. So we will annotate our Customer table with a new field called *orders\_count* and to actually count the number of orders placed by each customer will use Count class.

from django.db.models.aggregates import Max, Min, Count, Sum, Avg

 queryset = Customer.objects.annotate(

        orders\_count=Count('')

    )

Now, the question is which field should be used to count the number of orders.

So In our customer class

class Customer(models.Model):

    MEMBERSHIP\_BRONZE = "B"

    MEMBERSHIP\_SILVER = "S"

    MEMBERSHIP\_GOLD = "G"

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, "Bronze"),

        (MEMBERSHIP\_SILVER, "Silver"),

        (MEMBERSHIP\_GOLD, "Gold"),

    ]

    first\_name = models.CharField(max\_length=255)

    last\_name = models.CharField(max\_length=255)

    email = models.EmailField(unique=True)

    phone = models.CharField(max\_length=255)

    birth\_date = models.DateField(null=True)

    membership = models.CharField(

        max\_length=1, choices=MEMBERSHIP\_CHOICES, default=MEMBERSHIP\_BRONZE

    )

We do not see any field for field related to order, But in order class, we see customer field,

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)

So we can safely assume that Django has created a field by reverse relationship. And its name as per Django convention should be order\_set(*target class in lower case followed by \_set*), right?

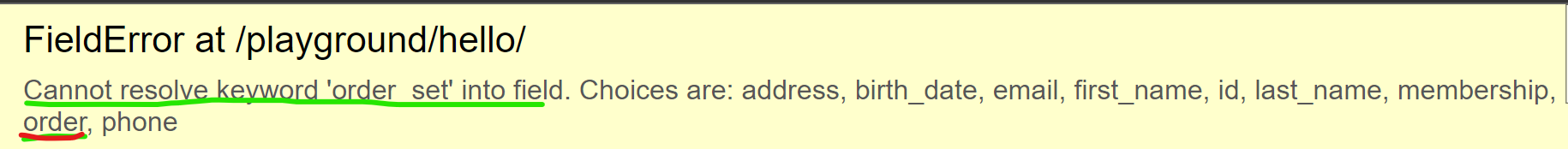
But as per Django developers we cannot use this naming convention here to count the number of orders for each customer.

queryset = Customer.objects.annotate(

        orders\_count=Count('order\_set')

    )

Using this field name in Count class we get exception error,



Look at the Choices and notice it says *order* in there not *order\_set*.

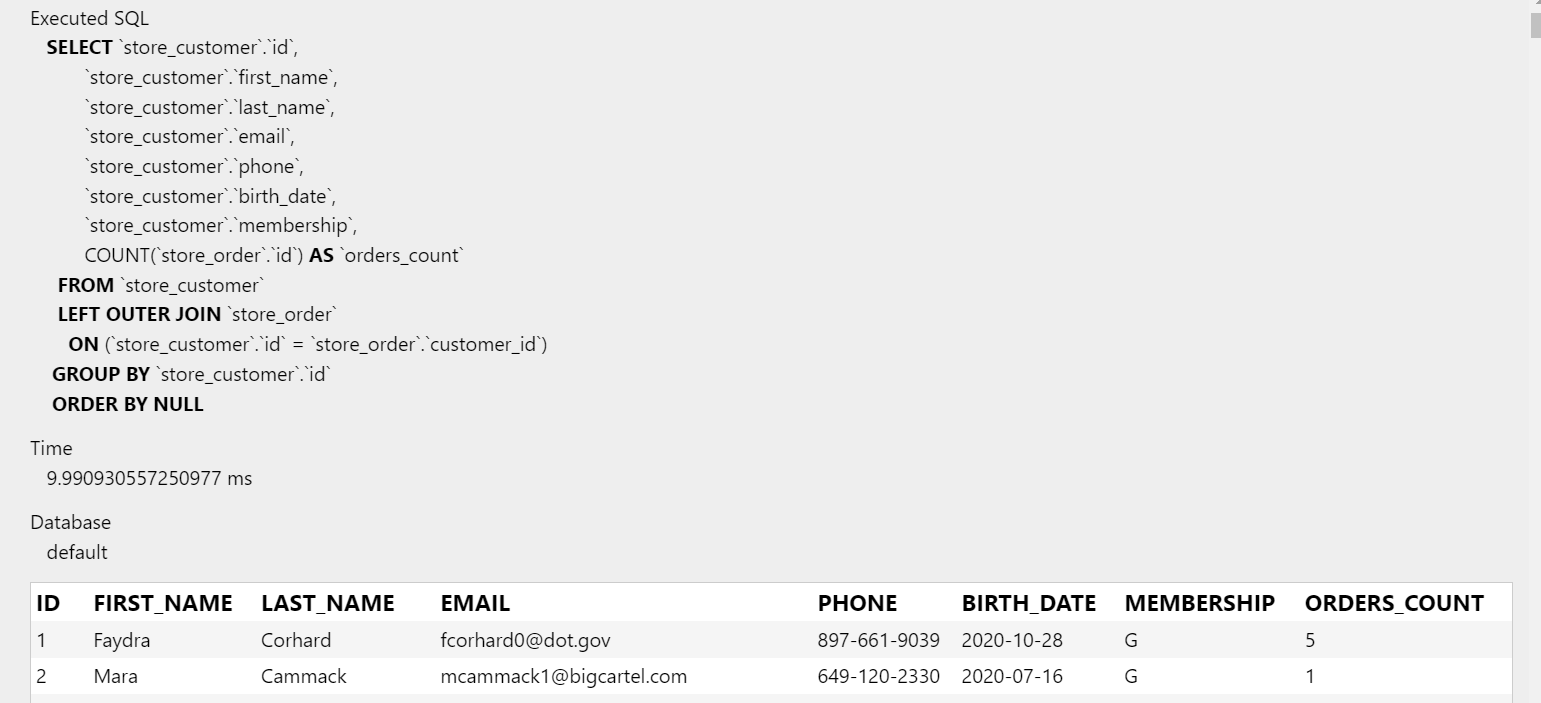
So let use the name of the field as *order*.

  queryset = Customer.objects.annotate(

        orders\_count=Count('order')

    )

Why does this happen, Need to know



First we get all the fields from Customer table

Notice that we have a LEFT OUTER JOIN between customer and order table because not every customer has placed an order.

Need to Know: Type of joins in SQL tables, Left Joins

And Finally we have a GROUP BY clause for grouping the number of orders for each customer.

**Working with Expression wrappers**:

Recap about what we learned so far:

🡪We talked about expression class which is the base class for all types of expressions.

🡪Derivatives of this class are Value, F, Func, Aggregate(base class for all aggregate classes like Count, Sum etc.)

Now, we are going to discuss another derivative of the expression of class called expression wrapper.

from django.db.models import Value, F, Func, ExpressionWrapper,

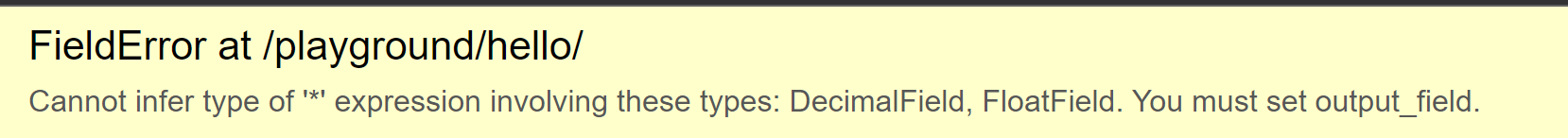
We use expression wrapper while building complex expressions. For example, annotate a new field to Product table called discounted\_price.

queryset = Product.objects.annotate(

        discounted\_price=F('unit\_price') \* 0.8

    )

We get a FieldError due to mixed types and we must set an output field.



It is because we are mixing a decimal field(which is unit\_price) with Float field(which is 0.8).

To solve this issue, *we need to wrap our expression inside an expression wrapper object and that is where we specify the type of the output field*.

To keep the code clean we wrap the expression outside our query set.

  discounted\_price = ExpressionWrapper(

        F("unit\_price") \* 0.8, output\_field=DecimalField()

    )

    queryset = Product.objects.annotate(discounted\_price=discounted\_price)



Note: Make sure to use parentheses after DecimalField() because we need to pass an object here(instance of the class) instead of plain class name.

And second you can import DecimalField class from same model class.

**Annotating Exercises**:

1. Customers with their last order id.

queryset = Customer.objects.annotate(last\_order\_id=Max("order"))

1. Collections and count of their products.

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

1. Customers with more than 5 orders.

 queryset = Customer.objects.annotate(orders\_count=Count("order")).

filter(orders\_count\_\_gt=5)

1. Customers and the total amount they have spent.

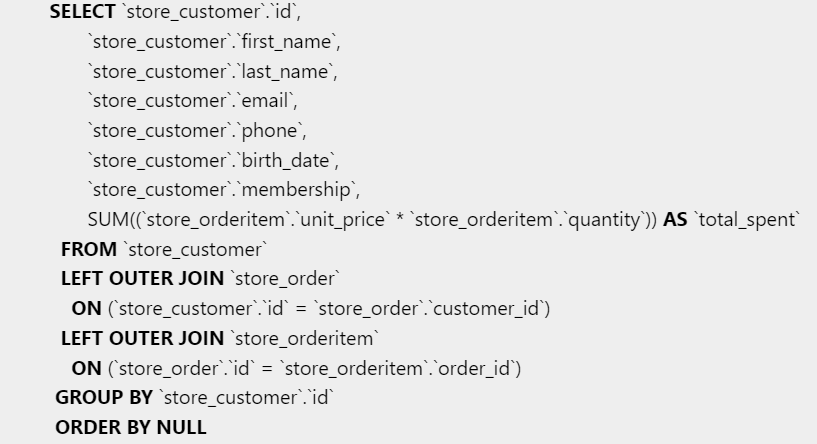
    queryset = Customer.objects.annotate(

        total\_spent=Sum(

            F("order\_\_orderitem\_\_unit\_price") \* F("order\_\_orderitem\_\_quantity")

        )

    )



1. Top 5 bestselling products and their total sales.

    queryset = Product.objects.annotate(

        total\_sales=Sum(F("orderitem\_\_unit\_price") \* F("orderitem\_\_quantity"))

    ).order\_by("-total\_sales")[:5]

**Querying Generic Relationships**:

Earlier in the course, we created tags app with two models,

class Tag(models.Model):

    label = models.CharField(max\_length=255)

class TaggedItem(models.Model):

    # What tag applied to what object

    tag = models.ForeignKey(Tag, on\_delete=models.CASCADE)

    content\_type = models.ForeignKey(ContentType, on\_delete=models.CASCADE)

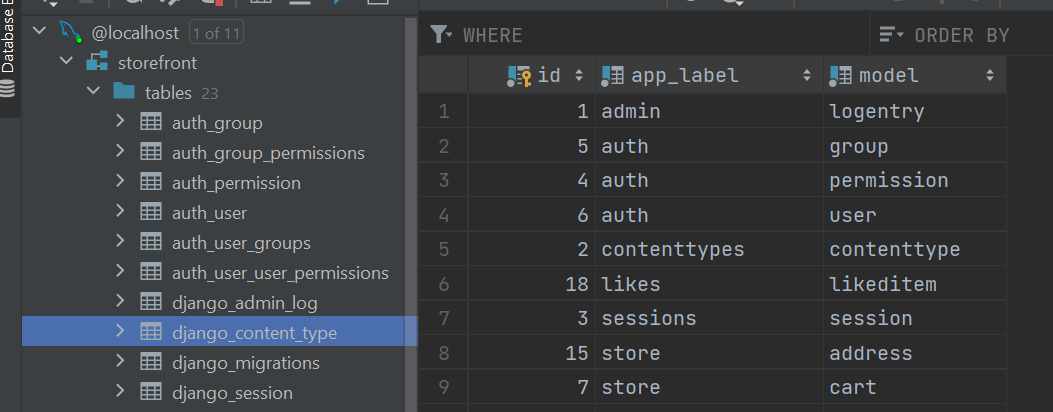
    object\_id = models.PositiveIntegerField()

    content\_object = GenericForeignKey()

We used ContentType framework to decouple this from the Store app. So this *Tags* app, know nothing about *Store* app and its models like *Product*.

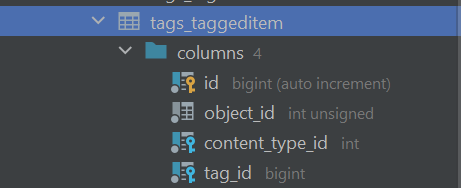
Due to this decoupling, we can use this app in any kind of project. But for now let us try to find tags for a given product.

In the database, navigate to django\_content\_type table.



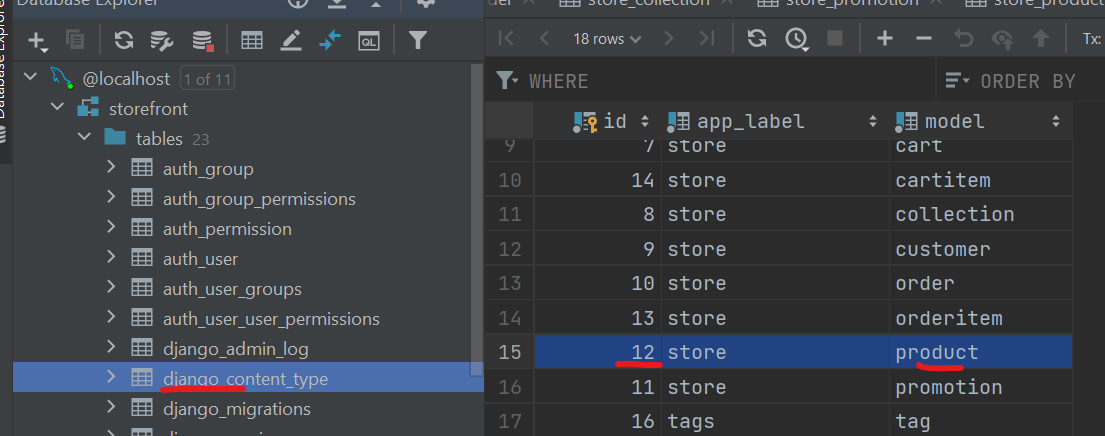
In this table, we can see all the models we have in our application, including automatically created models from *admin* and *auth* apps along with contenttype table from *contenttypes* app.

Back to our taggeditem table in tags app,



We have object\_id, content\_type\_id and tag\_id here. So *to find tags for a given product, first we have to find the* content\_type\_id *of Product model*.

In my database, at this moment content\_type\_id for Product model is 12



**The big picture is**:

If we know this id then we can go to taggeditem table and write a query to filter all records where *content\_type\_id = 12* and *object\_id equals id of the product* whose tags you want to find out.

**Same picture in form of code**:

🡪First we need to import ContentType model which represents contenttype table that we just saw in Database

from django.contrib.contenttypes.models import ContentType

🡪Import Product and TaggedItem classes from *store* and *tags* app respectively

from tags.models import TaggedItem

from store.models import Product

🡪Let us first find ContentType ID for Product model from ContentType table

 content\_type = ContentType.objects.get\_for\_model(Product)

Note: Since ContentType is a model, just like all the models we have created so far. But there are some special methods in its object manager like get\_for\_model which returns a content\_type instance for a given model(*Product*).

content\_type = ContentType.objects.get\_for\_model(Product)

This is how we will get content\_type\_id for Product model.

🡪This content\_type ID we can use to filter tagged items or products in our case.

TaggedItem.objects.filter(content\_type=content\_type)

🡪We give one more argument to filter method which should be object\_id(id of the product whose tags we want to query)

TaggedItem.objects.filter(content\_type=content\_type, object\_id=1)

Note: Here we have given a hard set value of 1 to object\_id to specifically query about that item/product. But ideal way of doing this will be to do this dynamically. For example *as per the product that customer is looking at, we should get product id from the URL and pass that instead*.

🡪Since the actual value of tag is not in TaggedItem table but instead it is in tag table(which is *a foreign key to TaggedItem*), so we preload our TaggedItem table with select\_rated(‘tag’)

   TaggedItem.objects.select\_related("tag").filter(

        content\_type=content\_type, object\_id=1

    )

Tip: When writing complex queries using ORM, use \ backslash for better understanding

    TaggedItem.objects \

    .select\_related("tag") \

    .filter(

        content\_type=content\_type,

        object\_id=1

    )

🡪 Now we get our query set, we can render it in HTML using key-value pairs

    return render(

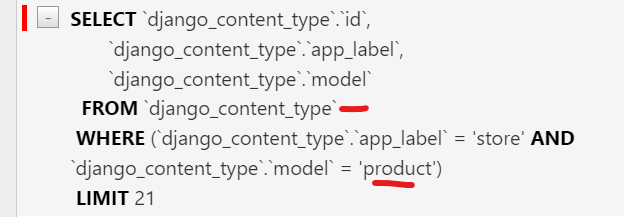
        request,

        "hello.html",

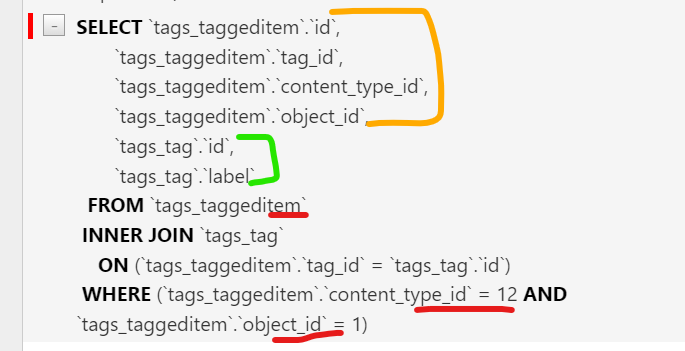
        {"name": "Himanshu", "tags": list(queryset)},

    )

🡪We see two queries



First query is for finding content\_type id for our Product model.



Second query is for reading the tags for a given product.

Need to do : Populate tag and taggedItem table

**Custom Managers**:

So our tags app is decoupled from store app, but writing code like this is not ideal. *Every time we want to find tags for a given object, first we have to find content\_type\_id, then we have to go to Tagged Item table, preload the tag field and apply multiple filters*.

It will be nicer if we could write a code like this to get the same result,

*TaggedItem.objects.get\_tags\_for(Product,1)*

Let us see how to implement this custom method using custom manager(not the default base manager we get from objects attribute), So go to our *Tags* app module.py file and

1. First we create a custom class called *TaggedItemManager* which will inherit from models.Manager which is a base class for all managers.
2. Inside this class we will implement our custom *get\_tags\_for* method. Its first parameter we will be self(represents method of the same class) other two will be *obj\_type* and *obj\_id*.
3. Move the query set we just created inside this custom function and replace Product model with obj\_type and object\_id’s hard set value with obj\_id. Return this query set directly from the function.

class TaggedItemManager(models.Manager):

    def get\_tags\_for(self, obj\_type, obj\_id):

        content\_type = ContentType.objects.get\_for\_model(obj\_type)

        return TaggedItem.objects.select\_related("tag").filter(

            content\_type=content\_type, object\_id=obj\_id

        )

1. Now we need to use this manager on our TaggedItem model so go to our TaggedItem class and add a new attribute here called objects and set it to an instance of *TaggedItemManager* (custom manager) class.

class TaggedItem(models.Model):

    # What tag applied to what object

    objects = TaggedItemManager()

    tag = models.ForeignKey(Tag, on\_delete=models.CASCADE)

    content\_type = models.ForeignKey(ContentType, on\_delete=models.CASCADE)

    object\_id = models.PositiveIntegerField()

    content\_object = GenericForeignKey()

1. We can go to playground and use this new method to get the tags for a given product.

queryset = TaggedItem.objects.get\_tags\_for(Product, 1)

**Understanding QuerySet cache**:

Let us talk about query set caching mechanism built into query sets.

queryset=Product.objects.all()

list(queryset)

Let us take this simple query as an example, where we get all the records from Product table.

We save that query in a variable called queryset and then convert it into a list, so that Django may evaluate it and send it to database to get the result.

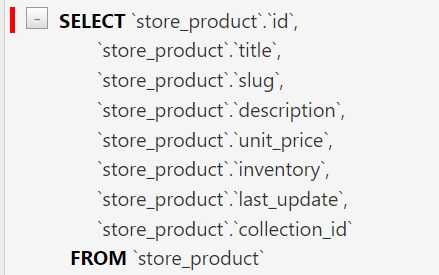
*Now this is an expensive operation because reading data from the disk is always slower than reading it from the memory*. *So when Django evaluates this query and gets the data from the database, it is going to store it somewhere in the memory called* ***query set cache***.

So second time, we convert this query set into a list, Django is not going to evaluate this query again or go to the database again. Instead it is just going to read the results from the query set cache. Same thing is going to happen if we read an individual object from the query set.

    queryset = Product.objects.all()

    list(queryset)

    queryset[0]

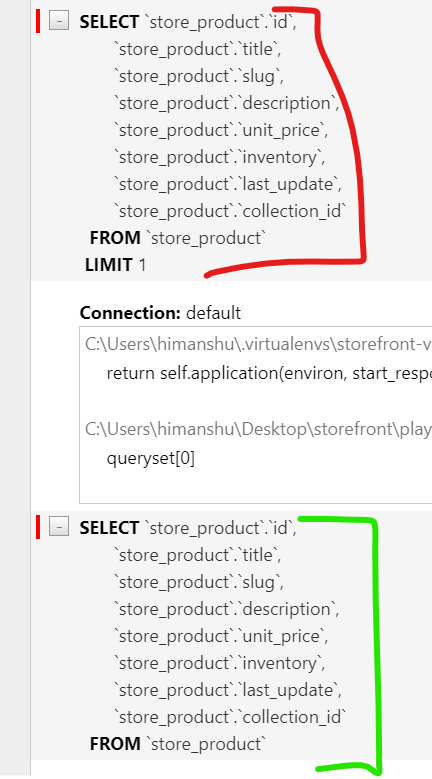
🡨See, Just this one query.

Note: If we try to read the single object first and then convert it to the list, we will end with two queries instead.

queryset = Product.objects.all()

    queryset[0]

    list(queryset)

See, Now we get two extra queries. First one to read first item of the table and second one is to read all items of Product table.

*So, keep this in mind, even though caching is a great optimization technique. But If you do not structure your code properly caching is going to come back and bite you*.

**Creating Objects**:

So far we have only been querying data, Now we will know how to insert a record in the database.

Let us create a collection object and set its title to ‘Video Game’ and set its featured\_product to *Product* with pk or id =1(preferred way is pk).

def say\_hello(request):

    collection = Collection()

    collection.title = 'Video Game'

    collection.featured\_product = Product(pk=1)

Another way to set featured\_product is to use the value of the primary key field,

def say\_hello(request):

    collection = Collection()

    collection.title = 'Video Game'

    collection.featured\_product\_id = 1

Note: Whichever way we write our featured\_product, one thing should be clear that *product should exist before we can create this collection*. In other words, we cannot create this collection and product at the same time. *This is how relational databases work, parent record should exist before we can create child record*.

Another approach of initializing this collection, is to use Key word arguments inside Collection class like this,

Collection(title='Video Games', featured\_product=Product(pk=1))

There are two things, wrong with this approach

🡪Intellisense do not suggest name of fields automatically.

🡪If we change name of any field in our model, those are not updated in keyword arguments.

Preferred way is to keep code like this

    collection = Collection()

    collection.title = "Video Games"

    collection.featured\_product = Product(pk=1)

To insert this collection in our database, all we have to do now is to call save method,

    collection = Collection()

    collection.title = "Video Games"

    collection.featured\_product = Product(pk=1)

    collection.save()

Every collection has this method for saving data.

Note: In this case, because we have not set the id for this collection, Django will treat this as an insert operation and will create it.



**Updating Objects**:

Let us update our “Video Games” collection title to “Games”

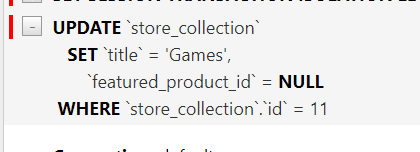
    collection = Collection(pk=11)

    collection.title = "Games"

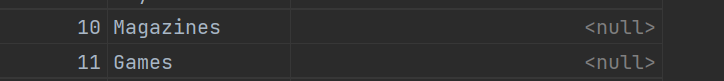
    collection.featured\_product = None

    collection.save()

We set the primary key for this collection and set title value to “Games” and featured\_product to None.

🡨In SQL tab we see an update statement to set collection title and featured\_product id.

Database also updated



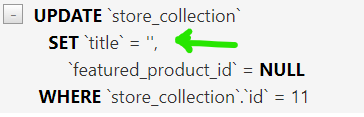
The tricky part:

Let us assume that we only want to update this featured\_product collection. So we do not set the title.

    collection = Collection(pk=11)

    collection.featured\_product = None

    collection.save()

🡨Django is setting the title to an empty string and this causes data loss!

So in order to update featured\_product\_id field, we end up losing title for this collection.

Reason behind this behavior:

The collection object that we have in memory

collection = Collection(pk=11)

By default its title is set to an empty string. So even if we do not explicitly update this field, Django is going to include in our SQL statement. This is where Django is different from other ORMs. *Other ORMs has a feature called change tracking*, *So that SQL statement generated will be based on the fields that are updated*.

To properly update an object in Django:

1. First we have to read object from the database. So that we have all the values in memory and then update it.

collection = Collection.objects.get(pk=11)

Now we have all the collections with this collection id.

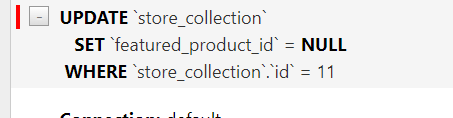
Note: Reading this object first may cause a performance penalty. But this is not an issue in most cases. *Do not try to prematurely optimize your code*.

1. If there is a performance issue encountered in this part of code, we can avoid it using base manager’s update method

With this method, we can avoid reading this collection first before updating it, so we can update it directly in the database.

Collection.objects.filter(pk=11).update(featured\_product=None)

First we filter out to target a particular collection and then update its field values using update method.

🡨This time Update statement is only setting the featured\_product.

**Deleting Objects**:

We have two options, either we can delete a single object or multiple objects in a query set. So if you have a collection we can delete that collection using delete method

*For deleting single collection*:

collection = Collection(pk=11)

collection.delete()

*For Deleting multiple collections*:

For Example, we want to delete all collections with id greater than 5

Collection.objects.filter(id\_\_gt=5).delete()

**Exercise**:

1. Create a shopping cart with an item

    cart = Cart()

    cart.save()

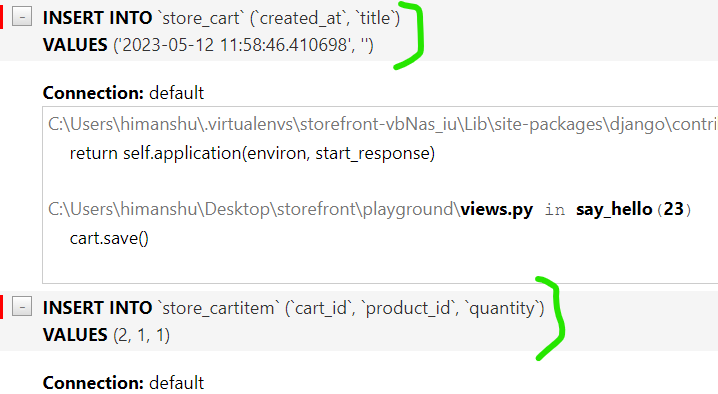
    item1 = CartItem()

    item1.cart = cart

    item1.product\_id = 1

    item1.quantity = 1

    item1.save()



1. Update the quantity of an item in a shopping cart

    item1= CartItem.objects.get(pk=2)

    item1.quantity=2

    item1.save()

or

CartItem.objects.filter(pk=1).update(quantity=2)

1. Remove a shopping cart with its items.

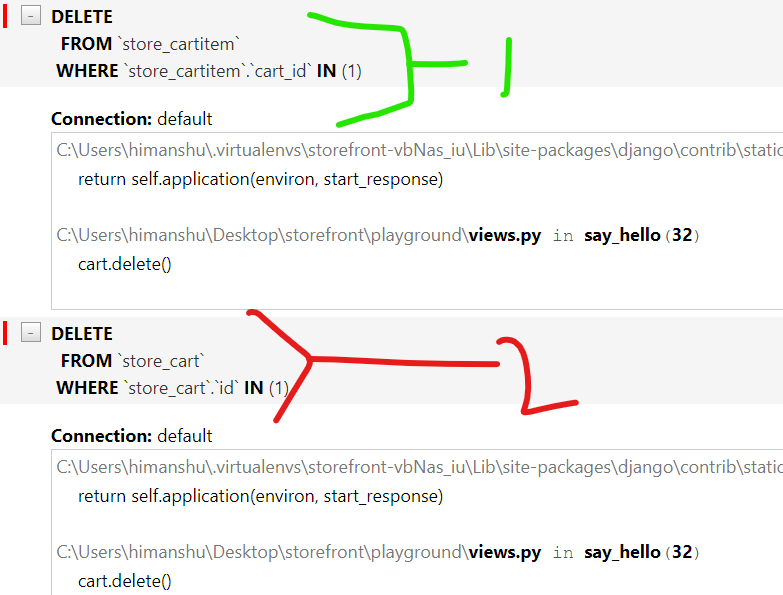
Cart.objects.filter(pk=2).delete()

OR

    cart = Cart(pk=1)

    cart.delete()

Since we have enabled cascading in the relationship between cart and cart items, deleting a cart automatically causes a deletion of its items.



**Transactions**:

Sometimes we want to make multiple changes in our database in an *atomic* way, *meaning all changes should be saved together or if one of the change fails, then all changes should be rolled back*.

For example, Saving an order with its items.

    order = Order()

    order.customer\_id=1

    order.save()

Note: We save the order first before its items. In relational databases, we should always create the parent record first before we can create child records.

Similarly creating an order item for this order

        item1 = OrderItem()

        item1.order = order

        item1.product\_id = 1

        item1.quantity = 1

        item1.unit\_price = 10

        item1.save()

Now Imagine while saving this order item, something crazy happens and we get an exception. In that case*, our database would be in inconsistent state (An Order without an order item!)*.

So this is where we use a transaction.

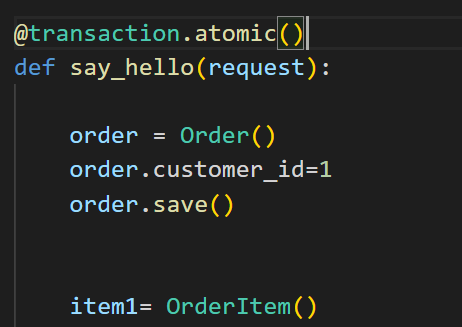
We are going to wrap both operations(creating order and order item) inside a transaction. Either both of these will be committed together or both will be rolled back if any operation fails.

Let us import transaction module first from django.db

from django.db import transaction

In this module we have a function called atomic which we can *use as a decorator or a context manager*.

We can use it over our view function

🡨This way our entire view function is wrapped inside a transaction.

But sometimes we might need more control over some part of view function that we want to wrap inside transaction. In those cases, we can use this as context manager.

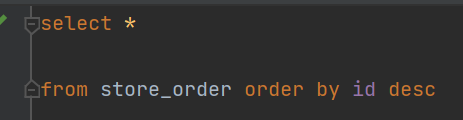
*Transaction as context manager*:

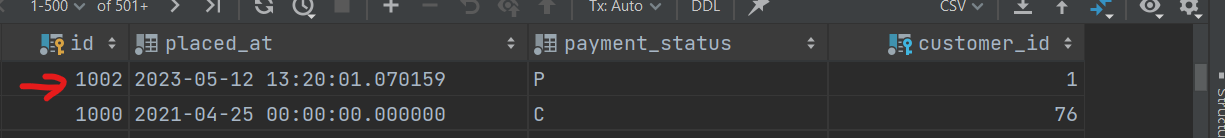


Inside view function, we can wrap the required part of code under a with block.

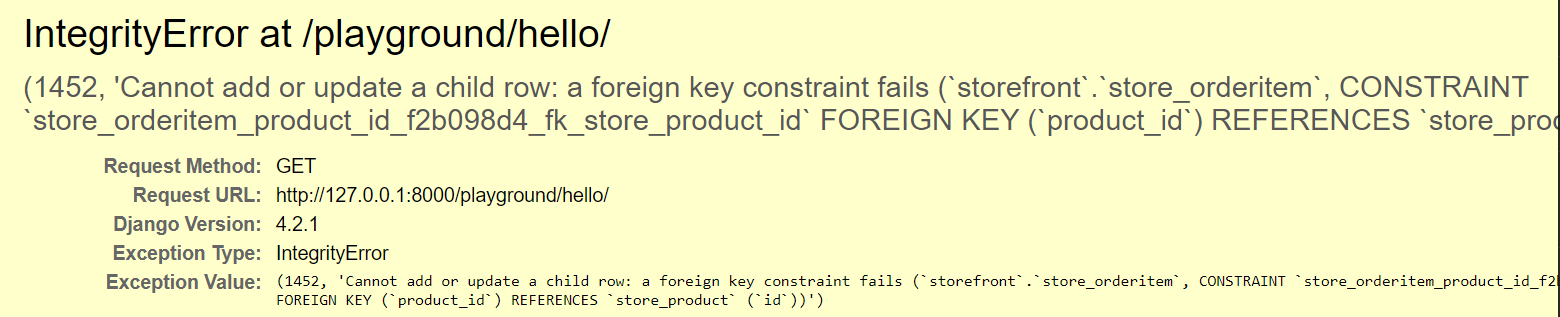
with transaction.atomic(), it returns a context manager.

Now we run the query in our database and we can see a new order





To demonstrate an exception, let’s go to our code and change item1.product\_id = -1 (there is no product with -1 ID)



We get Integrity error and our database is same like before.

**Executing Raw SQL queries**:

Sometimes implementing certain queries using Django ORM can get overly complex and you might end up with crazy annotations or filters. In those cases we can write our own SQL queries by hand and execute it directly using Django ORM.

We can use raw method which is also a manager method is used to execute raw SQL queries , e.g.

    Product.objects.raw('SELECT \* FROM store\_product')

Django is going to execute this query then it is going to map column names to field of product objects and return a bunch of products(*in other words, it will return a raw query set*).

Note: This query set we get from raw method is different from query sets we have seen so far. We cannot apply filter or annotate methods on this set.

But to evaluate this query set, we still need to convert it into a list.

 "results": list(queryset)



queryset = Product.objects.raw("SELECT id, title FROM store\_product")



Sometimes we want to execute queries that do not map to our model objects. In those cases *we can access the database directly and bypass the model layer*. Here instead of raw method we will use connections module.

from django.db import connection

In this module we have a method called cursor and it returns a *cursor object*.

cursor = connection.cursor()

This cursor object has a execute method, where we can pass any SQL statements. No Limitation(SELECT,INSERT,UPDATE, DELETE whatever…).

cursor.execute('')

After executing query, always close the cursor using close method, to release the allocated resources.

cursor.close(

Note: Proper way to use cursor is to *use a try-final block and inside final block we close the cursor*. This way if something went wrong, we do not end up with an open cursor.

We can also use a *with statement* in place of try-final block

    with connection.cursor() as cursor:

      cursor.execute('')

In this way we do not need to explicitly call close method and cursor will be closed even if there is an exception.

*callproc method*:

We also have another method called callproc or call procedure method for executing stored prcedures.

    with connection.cursor() as cursor:

        cursor.callproc('get\_customer',[1,2,'a'])

Here *get\_customer* is a stored procedure and we give it a bunch of parameters. This is much better and cleaner than writing your SQL queries in the middle of your python code.

So encapsulate your SQL queries inside of a stored procedure and then call it in python.

Query sets from exercises:

    # query\_set = Product.objects.filter(inventory\_\_lt=10).filter(unit\_price\_\_lt=20)

    # query\_set = Product.objects.values\_list("id", "title", "collection\_\_title")

    # result = Order.objects.filter(customer\_id=1).aggregate(count=Count("id"))

    # queryset = Customer.objects.annotate(

    #     full\_name=Func(F("first\_name"), Value(" "), F("last\_name"), function="CONCAT")

    # )

    # discounted\_price = ExpressionWrapper(

    #     F("unit\_price") \* 0.8, output\_field=DecimalField()

    # )

    # queryset = Customer.objects.annotate(last\_order=Max("order"))

    # queryset = Product.objects.annotate(

    #     total\_sales=Sum(F("orderitem\_\_unit\_price") \* F("orderitem\_\_quantity"))

    # ).order\_by("-total\_sales")[:5]

    # query\_set = Product.objects.filter(id=query\_set\_order\_item[1])

    # query\_object = Product.objects.filter(collection\_id=6).order\_by("title")[0]

    # query\_object = Product.objects.latest("title")

# query\_set\_customer = Customer.objects.filter(order\_\_=".com")

# query\_set\_collection = Collection.objects.filter(featured\_product\_\_isnull=True)

# query\_set\_order = Order.objects.filter(customer\_id=1)

# query\_set\_order\_item = OrderItem.objects.filter(product\_\_collection\_id=3)

# query\_set\_product = Product.objects.filter(id\_\_in=OrderItem.objects.values("product\_id").distinct()).order\_by("title")

Issue I am facing:

def say\_hello(request):

    # query\_set\_product\_id = OrderItem.objects.values("product\_id").distinct()

    query\_set\_product = Product.objects.filter(inventory\_\_gt=5).values(

        "title", "unit\_price"

    )

    return render(

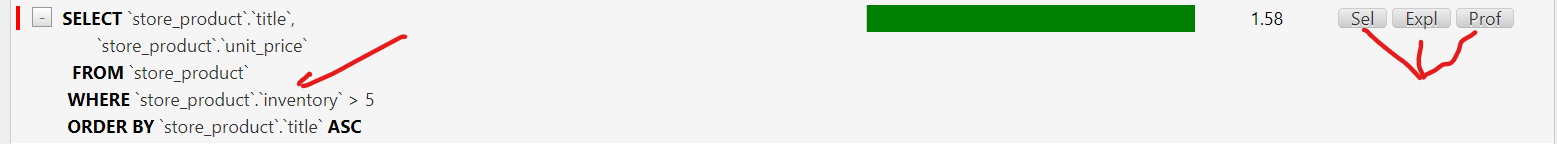
        request,

        "hello.html",

        {"name": "Himanshu", "products": list(query\_set\_product)},

    )

When I filter on the basis of inventory table value, I can see “Action” in django debug toolbar.



But If I Change inventory to unit\_price,

    query\_set\_product = Product.objects.filter(unit\_price\_\_gt=70).values(

        "title", "unit\_price"

    )

I do not see Actions on this query,

