ORM Summary

Django ORM

- in Relational Databases, Data is stored as rows in a table
- so when we want to pull out data from database, we need to map these rows to objects. in the past people did this by hand which was repetitive and time consuming.

Traditional Workflow

```
# Write a sql query
sql = 'SELECT * FROM product'

# Send it to the databse
result = execute(sql)

# Read the result and map it to a bunch of objects
for row in result:
        product = Product()
        product.title = row['title']
        product.price = row['price']
```

so for each record we'll have to create a new object and set its attributes.

This is where an **Object Relational Mapper** comes into the picture. as the name implies, it maps objects to relational records. and that frees us from writing a lot of repetitive code.

- no need to write SQL code (99% of the time)
- instead we code in an object oriented programming language like python
- then the OOP code gets translated to SQL code at run time

> ***Does that mean we never need to write SQL?***

No , when dealing with complex problems, ORMs can't produce efficient SQL Queries. So that's when we need to jump in and write **Optimized SQL Queries** by hand. But keep in mind this may take away from some of the security, so proceed with caution.

ORMs help us:

- Reduce complexity in code
- Make the code more understandable
- Get more done in less time

A good example of Django ORM is Running Migrations. Also all the models we created, inherit from the `Model` class in Django which is also a part of Django ORM **Remember, the more code you write, the more bugs you create.**

> ***Best Code is no Code.***

> ***A good software engineer, delivers working software in time***

> ***Premature Optimization, is the root of all evils.***
>
> - Donald Knuth

Managers and Querysets

The first thing we need to understand about Django ORM is two concepts:

- **Managers**
- **Querysets**

Managers

```python
from django.shortcuts import render
from store.models import Product

def say_hello(request):
        Product.objects
```

Every Model in Django has an attribute called **objects**. This returns a **Manager** object.

> ***A Manager object is an interface to the database.***

It's like a remote control with a bunch of buttons that we can use to talk to our database.

- **.all()** - for pulling out all the objects in the products table
- **.get()** - for getting a single object
- **.filter()** - for filtering data
- and more ...

After calling most of these methods, we get something in return called a Queryset

---

QuerySets

A Queryset is an object that encapsulates a Query. So at some point, Django is going to evaluate this Queryset and this is when Django will generate the right SQL statement to send to our database.
This Means QuerySets are Lazy 🦥

> **But when does this happen?**

Under a few scenarios:

- When we Iterate over a QuerySet - `for item in queryset`
- When we convert the QuerySet into a list - `list(queryset)`
- When we access an individual item in the QuerySet - `queryset[0]`
- When we slice the QuerySet - `queryset[0:5]`

So as you can see, Querysets are evaluated at a later point.

Debugging Queries

Use Django-Debug-Toolbar to see the exact queries sent to the database and the response. This is very useful in debugging applications.

Why are Querysets Lazy?

Because we can use Queryset methods to build complex Queries. Imagine you want to filter a Queryset twice and also order the results by a specific column. If Querysets were not **Lazy**, we'd have 3 calls to the database and query executions.

So Querysets being lazy, allows us to build our queries and then execute once.

In Contrast, we have some Manager methods that return the result immediately, like the `count()` which returns the number of rows. It doesn't make sense for this method to be lazy, because we are getting a number as a result, and it doesn't really make sense to do something extra with this number like:

- Filtering
- Ordering
- etc.

Retrieving Objects

- **.all()** - Returns all rows - *QS*
- **.get(lookup=value)** - Returns a single object (PK) - *OBJ*
    - Throws exception if not found
    - Can use get_or_404
    - Or

- **.filter()** - Returns rows matching a criteria - *QS*
    - **.first()**
    - **.exists()**
- etc.

Filtering Objects

Find Products that are 20$:

```
queryset = Product.objects.filter(unit_price=20)
```

More than 20$:

```
queryset = Product.objects.filter(unit_price__gt=20)
```

More:

- `__gt` - Grater than (>)
- `__lt` - Less than (<)
- `__gte` - Grater than or Equal to (>=)
- `__lte` - Less than or Equal to (<=)
- `__range` - Takes a tuple of two numbers

Find More lookup types:
Search for ***Queryset API***

Rendering Data in Templates

```
return render(
        request,
        'template.html',
        {'context': list(queryset)}
)
```

```
<ul>
        {% for product in products %}
        <li>{{ product.title }}</li>
        {% endfor %}
</ul?
```

Filtering across relationships

Let's say we want Products in the collection named `beauty`. But we only referenced the `collection_id` in the product model. So how?

```
Product.objects.filter(collection__title__icontains='Beauty
')
Product.objects.filter(collection__id__gt=3)
```

String Lookup

- `__startswith`
- `__endswith`
- `__contains`
  Put an `i` behind each, and you'll remove case sensitivity. i.e `__istartswith`

Date Lookup

- `__year`
- `__month`
- `__day`
- ...

Checking for null

- `__isnull`

Exercise Examples

```
customers = Customer.objects.filter(email__endswith='.com')
products = Product.objects.filter(inventory__lt=10)
orders = Order.objects.filter(customer__id=1)
order_items =
OrderItem.objects.filter(product__collection__id=3)
```

Complex Lookup Using Q Objects

Let's say we want:
Products: inventory < 10 ***AND*** price < 20

There are a few ways to implement this query

- Pass Multiple keyword arguments to the **filter** method: `Product.objects.filter(inventory__lt=10, price__lt=20)`
- Chain the call to **filter** method `.filter(inventory__lt=10).filter(price__lt=20)`

But how to use the OR operator?

We need to use **Q** objects.

```
from django.db.models import Q

def view(request):
        queryset = Product.objects.filter(
                Q(inventory__lt=10) | Q(price__lt=20)
        )
```

- Q is short for Query
- With Q objects we can represent a query expression
- First we encapsulate each keyword argument in a Q object
- Than we use the **Bitwise**(| , &) operators to determine **OR/AND**. But if you want an **AND** then just use the methods above.
- We can also use the **NOT** operator (~)

Referencing Fields using F Objects

Sometimes when filtering data, we need to reference a **Particular Field**.
for example -
Products: **Inventory** = **Price**

> Or comparing any other two fields

F Objects

```python
from django.db.models import F

def view(request):
        queryset =
Product.objects.filter(inventory=F('unit_price'))
```

- **F** is short for field
- We can also reference a related field like `F('collection__id')`

Sorting

We can sort results based on one or multiple fields:

- Ascending
- Descending

```
# ASC
queryset = Products.objects.order_by('unit_price')
# DESC
queryset = Products.objects.order_by('-unit_price')
# Multiple Fields
queryset = Product.objects.order_by('unit_price', '-title')
```

can also call `reverse()` on this because it's a Queryset. Also it can be called after `filter()` because `order_by()` is a Queryset method.

Sometimes we want to **sort** the result and get the top object.

```
# no longer a queryset
Product.objects.order_by('unit_price')[0]
# Another way
Product.objects.earliest('unit_price')

Product.objects.latest('unit_price')
```

Limiting Results

```
# Basic Limiting (python slicing)
queryset = Product.objects.all()[:5] # TOP 5
# LIMIT in SQL

queryset = Product.objects.all()[5:10]
# LIMIT 5 OFFSET 5
```

Selecting Fields to Query

Sometimes we only need a subset of the fields in the records

```
queryset = Product.objects.values('id', 'title')
# SELECT id, title FROM products
queryset = Product.objects.values('id', 'title',
'collection__id')
# SELECT + JOIN
```

- With the `values()` method, instead of getting product instances, we get a bunch of dictionary objects
- We have a similar method called `values_list()` which returns **tuples** instead of **dictionaries**
- We can use `distinct()` for removing duplicates.

Deferring Fields

Just like the `values()` method, we have another method called `only()` which works the same way. the only difference is:

- `values()` returns dictionaries
- `only()` returns objects (Model Instances)

⚠ The `only` Danger

If you're not careful with this method, you'll have many calls to the database and potentially crash the whole application. This Behavior happens when you want to access a field that you haven't specified in `only()` arguments.
On the other hand, `values()` method does not have this behavior.

Defer

With this method we can defer the loading of some fields to later. (**Excluding**)

```
queryset = Product.objects.defer('description')
```

This has the same danger.

Selecting Related Objects

Sometimes We need to load a bunch of objects together. Let's Imagine we need each product and the title of its collection next to it.
If we don't use `select_related()` method in our Queryset, we will have many unnecessary calls to the database.

```
queryset =
Product.objects.select_related('collection').all()
```

We also have another method called `prefetch_related()`.

- use `select_related()` when the other end of the relationship has 1 end
- use `prefetch_related()` when the other end of the relationship has many ends.
- A product can only have **One** collection so `select_related()`.
- A product can have **Many** promotions so `prefetch_related()`.

```
queryset = Order.objects.all().select_related('customer')
.prefetch_related('orderitem_set__product')[:5]
```

Aggregating Objects

Sometimes we want to compute summaries like:

- Min
- Max
- Average
- Sum
- etc.
  This is where we use the aggregate method

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

def view(request):
        queryset = Product.objects.aggregate(Count('id'))
```

- We use the aggregate function
- We pass an aggregation object like
  > - Min
  > - Max
  > - Sum
  > - Avg
  > - Count
- Then we pass the field to the aggregation object
- The aggregate Function does not return a Queryset

We can change the name of the key returned.

```
.aggregate(count=Count('id'))
```

We can also calculate multiple summaries

```
.aggregate(
        count=Count('id'),
        min_price=Min('unit_price')
)
```

Since `aggregate()` is a method of Querysets, we can apply it whenever we have a Queryset.

```
queryset = Product.objects.filter(collection__id=3)
.aggregate(count=Count('id'))
```

Annotating Objects

Sometimes we want to add additional attributes to our objects while Querying them. This is where we use the `annotate()` method

Let's say we want give each customer a new field called: `is_new` and set it to `True`

```
def view(request):
        queryset = Customer.objects.annotate(is_new=True)
```

Cannot Pass Boolean value

This throws an error that tells us we need to pass an `Expression` object.

Expressions

- `Value` - simple values
- `F` - field
- `Func` - database functions
- `Aggregate` - base class for aggregate classes

```
from django.db.models import Value
```

```
def view(request):
        queryset =
Customer.objects.annotate(is_new=Value(True))
```

Another example using `F`

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

def view(request):
        queryset = Customer.objects.annotate(new_id=F('id')
+ 1)
```

Calling Database Functions

First Name + Last Name

Using `CONCAT`

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

def view(request):
        queryset = Customer.objects
        .annotate(
                full_name=Func(
                        F('first_name'), Value(' ')
,F('last_name'),
                        function='CONCAT'
                )
        )
        # CONCAT(first_name, ' ', last_name) AS full_name
```

Better way

```
from django.db.models.functions import Concat

def view(request):
        queryset = Customer.objects.annotate(
                # No more target function / F
                full_name=Concat('first_name',Value(' '),
'last_name')
        )
```

Google `Django Database Functions` for more info

Special Functions

Some functions are common across all DBMS's but some are unique, for calling them we Have to use the `Func` object.

Grouping Data

Now let's say we want to see the number of orders each customer has placed:

```
def view(request):
        queryset = Customer.objects
        .annotate(orders_count=Count('order')) # REMEMBER
```

Left Join + Group By

Working with Expression Wrappers

Let's Recap

Expressions

- `Value` - simple values

- `F` - field

> - `Func` - database functions

> - `Aggregate` - base class for aggregate classes

There's also another one

Expression Wrapper

we use this class when building complex expressions.

Let's say we want to annotate our products and give them a new field called `discounted_price`

```
def view(request):
        queryset = Product.objects
        .annotate(
                discounted_price=F('unit_price') * 0.8
        )
```

This code throws an exception

> FieldError at /playground/hello
>
> Expression contains mixed types: `DecimalField`, `FloatField`. You must set `output_field`.

Solution

```
from django.db.models import ExpressionWrapper

def view(request):
        discounted_price = ExpressionWrapper(
                F('unit_price') * 0.8,
                 output_field=DecimalField() # Monetary
Float issue
        )
        queryset = Product.objects
        .annotate(
                discounted_price=discounted_price
        )
```

Querying Generic Relationships

Generics

- Tags
- Likes
- etc.

Find The tags for a given product.

```
def hello(request):
    content_type =
ContentType.objects.get_for_model(Product)
    queryset =
TaggedItem.objects.select_related('tag').filter(
        content_type=content_type,
        object_id=1
    )
```

Custom Managers

The method we used for Querying Generic Relationships is not ideal.
Because each time we want to find the tags for a given object, there are many steps we need to take.
We make the process easier with **Custom Managers**.

The Idea:

```
TaggedItem.objects.get_tags_for(Product, 1)
```

Implementation

In the `models.py` module of the Target model

```
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
                        .filter(
                                content_type=content_type,
                                object_id=obj_id # Should
be Dynamic
                        )
                )
```

Now we need to use this manager in the `TaggedItem` model.

```
class TaggedItem(models.Model)
        objects = TaggedItemManager()
```

Understanding Queryset Cache

> Reading data from the **disk**, is always slower than reading it from the **memory**

```
queryset = Product.objects.all()

# Evaluate Queryset → read from disk
list(queryset)

# Read data from cache (Any other way for evaluation)
list(queryset)
```

this can be seen in action if you see the SQL queries sent to the database.

Tip

Caching only happens if we evaluate the full queryset.

Creating Objects

```
def view(request):
        collection = Collection()
        collection.title = 'Video Games'
        # Two ways (Product must be available)
        collection.featured_product = Product(pk=1)
        collection.featured_product_id = 1

        # Another way
        collection = Collection(title='Video Games')
        # ! No intellisense
        # ! Refactoring Problem

        collection.save()
        # Because we haven't specified ID, this is seen as
INSERTION
```

OR

```
# Does the same thing under the hood (call to save())
Collection.objects.create(title='a', featured_product_id=1)
```

Traditional Approach is preferred

Updating objects

```
def view(request):
        collection = Collection(pk=11)
        collection.title = 'Video Games' # → Change to
`Games`
```

```
        collection.featured_product = Product(pk=1)
        collection.save()
```

To properly update objects without data loss:

```
def view(request):
        collection = Collection.objects.get(pk=11)
        collection.featured_product = Product(pk=2)
        collection.save()
```

OR

```
# Without the read call but fragile
Colelction.objects.filter(pk=11).update(featured_product=No
ne)
# Without filter() it's like updating without WHERE clause
```

Deleting Objects

We can either delete **one** or **multiple** objects

Single Object

```
collection = Collection(pk=11)
collection.delete()
```

Multiple Objects

```
collection = Collection.objects.filter(id__gt=5).delete()
```

Transactions

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

saving order with its items

```
def view(request):
        order = Order()
        order.customer_id = 1
        order.save()

        order_item = OrderItem()
        order_item.order = order
        order_item.product_id = 1
        order_item.quantity = 1
        order_item.unit_price = 10
        order_item.save()
```

Now let's say, along the execution of this code, something crazy happens. then we might have an order without items (Inconsistency). so what can we do to prevent this ?

Transaction !

```
from django.db import transaction


# all the code
@transaction.atomic()
def view(request):
        order = Order()
        order.customer_id = 1
        order.save()

        order_item = OrderItem()
        order_item.order = order
        order_item.product_id = 1
        order_item.quantity = 1
        order_item.unit_price = 10
        order_item.save()

# OR use a context manager
def view(request):
#     ...
```

```
        with transaction.atomic():
                order = Order()
                order.customer_id = 1
                order.save()

                order_item = OrderItem()
                order_item.order = order
                order_item.product_id = 1
                order_item.quantity = 1
                order_item.unit_price = 10
                order_item.save()
```

Executing Raw SQL Queries

Sometimes writing an SQL Query with Django ORM get's overly complex. In these cases we can write raw SQL queries and send them to the database.

```
def view(request):
        queryset = Product.objects.raw('SELECT * FROM
store_product')
        # this method returns a raw queryset
```

Tip

Only use this approach if you realize that writing raw SQL is easier and cleaner / Faster

Bypass the model layer

Sometimes the query we want to execute does not have any associations to our models. For these scenarios we directly connect to the database from our code.

```
from django.db import connection
```

```python
def view(request):
        with connection.cursor() as cursor:
                cursor.execute('SQL QUERY')
                # Calling StoredProcedures
                cursor.callproc('proc_name', [parameters])
```