Queryset In Django

A queryset in Django is a collection of database query results, usually representing a set of records from a database table or model. Django provides a powerful and flexible query API for interacting with the database using querysets.

(1)Filtering:

(a)filter(**kwargs):

Filters the queryset based on specified conditions.

Syntax:

students = Person.objects.filter(is_student=True)

Output:

[<Person: John>, <Person: Michael>, <Person: Jessica>]

(b)exclude(**kwargs):

Excludes records that match the specified conditions.

Syntax:

non_students = Person.objects.exclude(is_student=True)

Output:

[<Person: Sarah>, <Person: Emily>, <Person: Benjamin>]

(2)Chaining Querysets:

(a)| (OR operator) and & (AND operator):

Combine query sets using logical OR and AND.

Syntax OR:

result = Person.objects.filter(is_student=True) | Person.objects.filter(age=25)

Output:

[<Person: John>, <Person: Emily>, <Person: Michael>, <Person: Jessica>]

Syntax And:

result = Person.objects.filter(is_student=True) & Person.objects.filter(age=25)

Output:

[<Person: John>]

(b)Q() objects:

Use complex conditions for filtering and chaining.

Syntax:

result = Person.objects.filter(Q(is_student=True) | Q(age=25))

Output:

[<Person: John>, <Person: Emily>, <Person: Michael>, <Person: Jessica>]

Syntax:

result = Person.objects.filter(Q(is_student=True) & Q(age=25))

Output:

[<Person: John>]
Syntax:
result = Person.objects.filter(~Q(is_student=True) | Q(age=25))
Output:
[<Person: John>, <Person: Emily>, <Person: Sarah>, <Person: Michael>, <Person: Benjamin>]

(3)Ordering:

(a)order_by(*fields):

Orders the queryset based on one or more fields where - (minus sign):Indicates descending order.
Syntax:
people_by_age_asc = Person.objects.all().order_by('-age')
where - represents the descending order and + represents the ascending order.
Output:
[<Person: Jessica>, <Person: John>, <Person: Emily>, <Person: Michael>, <Person: Sarah>, <Person: Benjamin>]

(4)Selecting Specific Fields:

(a)values(*fields):

Returns a queryset of dictionaries containing specific fields.
Syntax:
queryset = Location.objects.all().values(“sector”)
Output:
<QuerySet [
{'sector': 'Forest'},
{'sector': 'Forest'},
{'sector': 'Women, Children & Social Welfare'},
{'sector': 'Women, Children & Social Welfare'},
{'sector': 'Drinking Water'}, {'sector': 'Drinking Water'},
'...(remaining elements truncated)...'
]>
Explanation:
This part of the code retrieves distinct values from the "sector" field in the dataset.This can be useful for further operations like aggregation or annotation, where you want to perform calculations or analyses based on these distinct categories.

b)values_list(*fields):

Returns a queryset of tuples containing specific fields.
Syntax:
age_list = Person.objects.all().values_list('age', flat=True)

Output:
[25, 30, 25, 28, 22, 35]
Explanation:
When you use values_list() with flat=True, you are asking Django to return a flat list of values for a single field, instead of returning tuples or lists of values.
Syntax:
name_age_list = Person.objects.all().values_list('name', 'age')
Output:
[('John', 25), ('Sarah', 30), ('Emily', 25), ('Michael', 28), ('Jessica', 22), ('Benjamin', 35)]

(5)Aggregations:

aggregate(**kwargs):

Performs aggregate functions on queryset fields.
Syntax:
total_budget=queryset.aggregate(total_budget=Sum("commitments")["total_budget"]
Explanation:
The provided code calculates the total budget by summing up the values in the "commitments" field for all projects in the queryset. The calculated total budget is then accessed and stored in the total_budget variable. This approach is useful for retrieving aggregated information from a dataset without manually iterating through each record.

(6)Annotation:[टिप्पणी]

annotate(**kwargs):

Adds annotated fields based on calculations and aggregations.
Example:
queryset.values("sector").annotate(project_count=Count("sector"))
Explanation:
Meaning:It means to add notes (an-NOTE-tate) to text that you are reading, to offer explanation, comments or opinions to the author's words.
In Django, "annotate" refers to the process of adding computed or aggregated values to each item in a queryset. The annotate() method allows you to enrich your queryset with these additional calculated attributes.
Output:
<QuerySet [
{'sector': 'Agriculture, Livelihood', 'project_count': 2},
{'sector': 'Drinking Water', 'project_count': 6},
{'sector': 'Drinking Water, Irrigation, Alternate Energy', 'project_count': 2},
{'sector': 'Education', 'project_count': 2},
{'sector': 'Education, Health, Agriculture, Renewable Energy', 'project_count': 2},

{'sector': 'government', 'project_count': 1}
]>

Example 2:

queryset.values("sector").annotate(project_count=Count("sector"),budget=Sum("commitments"))

<QuerySet [
{'sector': 'Agriculture, Livelihood', 'project_count': 2, 'budget': 2886160}, {'sector': 'Drinking Water', 'project_count': 6, 'budget': 2191796},
{'sector': 'Drinking Water, Irrigation, Alternate Energy', 'project_count': 2, 'budget': 930028},
{'sector': 'Education', 'project_count': 2, 'budget': 1264450},
{'sector': 'government', 'project_count': 1, 'budget': 768914}
]>

Accessing the assigned value inside annotate

```
for count, item in enumerate(sector_counts):
        sector = item["sector"],
        budget = item["budget"]
```

(7)Difference between annotate and aggregate

(a)aggregate():

1)Use aggregate() when you want to calculate aggregated values across the entire queryset and retrieve summary information.
2)The result of aggregate() is a single dictionary with aggregated values, not related to individual records in the queryset.
3)Aggregations are performed on the entire queryset, providing a summary of the data.

(b)annotate():

1)Use annotate() when you want to add calculated attributes to each object in the queryset, often based on related data or more complex calculations.
2)The result of annotate() is a modified queryset with additional attributes (annotations) added to each individual object.

(8)Slicing and Pagination:

(a)all():

Returns all records in the queryset.

(b)first():

Returns the first record in the queryset.

(c)last():
Returns the last record in the queryset.
(d)distinct():
Removes duplicate records from the queryset.
e)count():
Returns the number of records in the queryset.

(9)Joins and Relationships:

(a)select_related():
select_related() is used to retrieve related objects using a SQL join operation. It works for ForeignKey and OneToOneField relationships. When you use select_related(), Django fetches the related object's data in the same SQL query that retrieves the main objects. This helps to avoid additional database queries for each related object.
Example:
authors = Author.objects.select_related('book')
Suppose you have an Author model with a ForeignKey relationship to a Book model. Using select_related(), you can fetch authors and their associated books in a single query:
Mechanism:
select_related() fetches the related data using a SQL join operation, which combines data from both tables into a single result set. This reduces the need for separate queries to retrieve the related objects.

(b)prefetch_related():
prefetch_related() is used to retrieve related objects using separate SQL queries for each relationship. It's particularly useful for ManyToManyField and reverse ForeignKey relationships. This optimization method retrieves all related objects in a more efficient manner than performing a separate query for each related object.
Example:
authors = Author.objects.prefetch_related('genres')
Suppose you have an Author model with a ManyToManyField relationship to a Genre model. Using prefetch_related(), you can fetch authors and their associated genres using fewer queries:
Mechanism:
prefetch_related() fetches the related data using a separate SQL query for each relationship. It then organizes the data into dictionaries or sets, which are used to efficiently populate the related objects without needing additional queries.

Differences

a)select_related() uses SQL joins to fetch related objects in a single query for ForeignKey and OneToOneField relationships whereas prefetch_related() uses separate queries to fetch related objects efficiently for ManyToManyField and reverse ForeignKey relationships.
b)Both methods reduce the number of queries needed to retrieve related data, improving performance by minimizing database round-trips. The choice between them depends on the type of relationship and the specific use case.

(10)Subqueries:

(a)Subquery():

Allows using subqueries within queries.

(b)OuterRef():

Represents an outer query reference within a subquery.

Model:

```
class Author(models.Model):
    name = models.CharField(max_length=50)
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    year_published = models.IntegerField()
```

Syntax:

```
from django.db.models import Subquery, OuterRef

same_year_books = Book.objects.filter(
    year_published=Subquery(
        Book.objects.filter(author=OuterRef('author'))
        .values('year_published')
    )
)
```

Explanation:

This query retrieves books where the author has published books in the same year as the book being considered.The Subquery() is used to create a subquery within the Book queryset. It filters books by the same author as the outer query's book (OuterRef('author')) and then extracts the year_published values of those books using .values('year_published').

(11)Raw SQL Queries:

raw(sql, params):

Executes raw SQL queries.

Topic:
Retrieve all authors using raw SQL query:
Retrieve authors with a specific name using a parameterized raw SQL query:
Syntax:
```
authors_raw = Author.objects.raw('SELECT * FROM yourapp_author')
authors_named_john = Author.objects.raw('SELECT * FROM yourapp_author
WHERE name = %s', [author_name])
```
Explanation:
This query retrieves all authors using a raw SQL query and returns an iterable of Author instances.
This query retrieves authors with the name "John" using a parameterized raw SQL query and returns an iterable of Author instances.

(11)F() and Expression() Objects:

F() objects and Expression() objects:

F() Allows referencing model fields in queries and Expression() allows us to perform arithmetic and bitwise operations.
Syntax:
```
from django.db.models import F, Expression, DecimalField
total_price_expression = Expression(F('price') * F('quantity'),
output_field=DecimalField())
```
Explanation:
F('price') * F('quantity') represents the calculation of the total price by multiplying the price field with the quantity field where output_field=DecimalField() specifies that the calculated result should be stored in a DecimalField.