Going beyond the basics with Django & Databases

Dave Trollope
dave@knowledgehound.com

Biography - Dave Trollope

Senior Software Engineer @ KnowledgeHound

5 years developing a data analytics platform for survey data using python.

Have a survey you sent and want to analyze the results effectively? Reach out!

My past:

15 years developing in C, then a few years with Ruby, and then Python.

Java and Kotlin being interesting diversions along the way!

Managed engineering teams for 5 years before returning to being a hacker, ahem, a software engineer/individual contributor

Django & Databases: Basic Usage

Basic Usage of Django with Databases (otherwise known as an ORM) is:

- Creating Models and Migrations using very common field types
  - AutoField, Integer/FloatField, Big*Field, FileField, TextField etc
- Using <Model>.objects use any of the following patterns
  - filter(field=X), filter(field1=X, field2=Y, …), exclude(field=X), exclude(... you get the idea)
  - Following relationships in all of the above filter(field1__child_field=1) etc
  - get()
  - all(), iterator()
  - exists()
- You generally know what a “queryset” is

A lot of great apps can be built with only these basic patterns.

This presentation describes intermediate level patterns and capabilities.

Simple pattern improvements

Its widely known that you use .get() to retrieve an instance. How we use get() may differ but….

What is often overlooked is that you can pass any filter expression to get()

<Model>.object.get(field1=X) -> does the same as <Model>.objects.filter(field1=X).get()

- Some simple code improvements might look like:

  queryset = <Model>.object.filter(field1=X)
  queryset = queryset.exclude(field2=Y)
  queryset.get()

  Simplify to -> queryset.exclude(field2=Y).get(field1=X)

- When working in a shell, (IPython highly recommended), it's often simpler to just add .get() to the end of a previous command:

  <Model>.objects.filter(field1=X) -> Yields one result in the queryset… *so just recall the command and append* .get()

  obj = <Model>.objects.filter(field1=X).get() # Yes, some might consider this lazy, but in a shell the less the you type, the better 😁

None of these are going to change your world, but they make life just a tad easier and faster.

Leveraging Operators

Ok, exact field matching solves a lot of problems, but when you only know a part of the value you want to match, life is much simpler using the following operators (accelerating your data discovery):

startswith, endswith, contains, in, gt, gte, lt, lte

Operators are separated from fields using __ in the same way relationships are navigated:

field1__startswith='foo'

field1__in=[1,2,3]

Operators like *startswith*, *gt* (greater than) depend on the field type (text, numeric etc)

Text operators like *contains* have case insensitive versions prefixed by an 'i': *icontains*(), *iexact*()

Use *iexact* to get a case insensitive match on a whole string, common with email:

support_emails = <Model>.objects.filter(email__iexact='support@yourcompany.com')

Operators like '*in*' allow multiple values to be matched using an iterable, like a list.

Some folks might say this is all "basic" knowledge, but I'm reiterating just in case you don't know.

What about OR operations and filters?

When you know the basics, it's easy to do a lot of stuff. Until you need to query for **multiple values in different fields** eg. X in field1 OR Y in field2. You might do something like this:

```
if <Model>.objects.filter(field1=X).exists():
        do_stuff(<Model>.objects.filter(field1=X))
if <Model>.objects.filter(field2=Y).exists():
        do_stuff(<Model>.objects.filter(field2=Y))
```

It's ok, but you know its not great, a little hard to read quickly and it's inefficient!

Calling exists() and then duplicating the filter() when you do something with what does exist causes 2 database operations. Avoid that pattern!

Django Docs for exists() -> Use bool(some_queryset)

You might simplify this by using the OR operator. Err what?

While you may think of the OR operator in purely boolean terms, it's important to understand the difference between a logical OR (boolean - the keyword 'or') and a "bitwise" OR ('|'). A "bitwise" OR merges two objects at their most basic level e.g. numerics represented in 'bits' are merged to yield a new numeric value. **That means you can apply the "OR" operation to more complex objects like querysets**. This is a functional programming model, and is most easily understood using set's:

```
{1}|{2} -> Two sets OR'd together creates a new set : {1, 2}
set()|{2} -> A falsey set OR'd with a non falsey set yields a non falsey set: {2}
```

Understanding how to use operators with set's in python greatly reduces the verbosity of code. Avoid using add() etc

Extending this idea to Django querysets and you realize that you can simplify the above code to one simple expression by merging querysets with OR

```
do_stuff(<Model>.objects.filter(field1=X)|<Model>.objects.filter(field2=Y))
```

You can also use the AND (&) operator in the same way to get the intersection of these two filter expressions instead of the union. Think sets! That's why they are called QuerySet's!

Simplify even more using Q()

But wait, you can even eliminate two calls to filter() and simplify that even further by using Django's Q() (read as a query) method

```
from django.db.models import Q

<Model>.objects.filter(Q(field1=X)|Q(field2=Y))
```

The hardest part about this is remembering to import Q from django.db.models

Q() wraps an expression in to an object that you can use operators on, so combining them yields a single expression and thus the simplest code.

It's shorthand for calling filter() twice and merging the results with the OR operator

It might actually be a little harder to read when you aren't completely familiar with Q() - but it quickly becomes second nature. Don't forget other operators can be used also!

Django Docs for Q(), and good examples

And there is also F()!

Now you know all about Q(), it's a good time to learn about F(). It's a wrapper that lets you access fields on a model and apply operations to that field at the database level. IE. An expression built with F is passed to SQL as part of the query.

Let's say you want to filter on field X and get rows that have values less than the field Y divided by 2.

You can write a filter such as:

<Model>.objects.filter(X__lt=F('Y')/2)

You can use any field expression as part of F() so you can follow a relationship and filter by the values of a field in a related model:

<Model>.objects.filter(X__lt=F('a__b')/2) # select rows where X is less than half of field b in model a

It should be noted, that these expressions are executed by the database, so you can't use lambdas or Python methods etc as part of the expression.

Django Docs for F()

Understanding values() and values_list()

Before we get in to the last query expression topic, let's talk about methods values() and values_list(). They are great tools for both optimizing your database queries and **reducing data to the minimum** needed to solve specific problems.

It's very common that you only need a single column (or two) of data to solve a problem. values() and values_list() reduce a query (which defaults to all columns) to **just the columns you need**, and thus creates more optimal queries. For example:

<Model>.objects.values('name', 'city') or <Model>.objects.values_list('name', 'city')

Creates an SQL query that selects just the columns/fields 'name' and 'city' (instead of all).

values() returns a list of dicts, and values_list() returns a list of tuples.

That probably isn't exactly what you expected - its a bit weird to have a method values_list() and then find that values() also returns a list! Let's look at a couple of examples to be clear.

Comparing outputs of values and value_list

<Model>.objects.values(‘name’, ‘city’):

[{'name': 'Dave', ‘city’: ‘Chicago’},
{'name': 'Jane', ‘city’: ‘Florida’}]

<Model>.objects.values_list(‘name’, ‘city’):

[('Dave', ‘Chicago’), ('Jane', ‘Florida’]

Depending on your use of this data, either might be appropriate, though sometimes, these are not as usable as you would like. Here are some tips for more usable patterns:

If you want to **create a map from one field value to another**, e.g. a dict keyed by name to city, use values_list and cast to a dict:
dict(<Model>.objects.values_list(‘name’, ‘city’)) -> {‘Dave’: ‘Chicago’, ‘Jane’: ‘Florida’}

If you **only need 1 value, use values_list with flat=True**:
dict(<Model>.objects.values_list(‘name’, flat=True)) -> [‘Dave’, ‘Jane’]

Use a **named tuple** to make accessing objects in a list more object like:
Import collections
citymap_tuple = collections.namedtuple('CityMap', ['name', 'city'])
[**citymap_tuple**(*v) for v in <Model>.objects.**values_list**('name', 'city')]

produces: [CityMap(name='Dave', city=’Chicago’), CityMap(name='Jane', city=’Florida’)]

Named Tuples allow access of data by attribute name and are far better than relying on numeric indexing of basic tuples, resulting in more maintainable code:

citymap[0].name instead of citymap[0][0]

Also, instead of using a list comprehension, use a generator for large queries

When using values() and values_list(), **remember to consider the overall application workflow** to ensure you aren’t creating unnecessary database queries by over using values() or values_list()

On to…. Subqueries!

You can do many things with Q(), F() to create rich database queries that solve a lot of problems and values() and values_list() lets you process the data you need very effectively. *But at some point, you just need to be able to create a Subquery*.

One of the primary reasons you might want to use a Subquery is when **two models are loosely coupled** and don't have a direct relationship between them.

filter(), F(), Q() etc all work through direct associations with a model and it's impossible to cross loose boundaries

Django will automatically create subqueries internally if you pass a queryset to a filter:

<Model A>.objects.filter(name__in=<Model B>.objects.values_list('name', flat=True))

This works in many cases, but sometimes you want and need to create the Subquery yourself because of the shape of the query. Subquery allows you to do so.

Like F() and Q(), Subquery() is a wrapper, in this case for another queryset. It's also very common that your subquery will only need to return a single value for you do match on, and you can use values_list() for this! For example, find names in Model A that contain(case insensitively) common names stored in Model B:

<Model A>.objects.filter(name__icontains=Subquery(<Model B>.objects.values('name')))

Note: Typically you would assume values_list(..., flat=True) is necessary to return a single value - but subqueries are smart and figure it out!

Subquery, and a number of other great expression related tools were introduced in Django 1.11. It's time to upgrade if you still have applications on 1.x! Even if you are on 1.11 or later, it's probably time to upgrade also! KnowledgeHound over the last year upgraded all of its services to the latest LTS 3.x - (Topic for another talk perhaps?) - If you are having trouble upgrading, I'm happy to discuss offline - reach out.

Django docs for Subquery()

One last expression related topic: Annotations/Aggregations

Applications often need to ‘sum a bunch of stuff’ and it’s easy to leap writing a query that gets all the values and and sums them up in python, perhaps even using a defaultdict for ease, something like:

```
counts = defaultdict(0)
for name in <Model>.objects.filter(...).values_list(‘name’):
    counts[name] += 1
```

But what if the database could do it alot faster? This is becomes more important as you have more data to sum. Python is great, but the database is faster! Consider this:

```
from django.db.models import Count

<Model>.objects.order_by(‘name’).values(‘name’).annotate(name_count=Count(‘name’))
```

More about annotate()

Using the earlier examples, perhaps we want a count of names indexed by name. Combining dict() and values_list() with annotate() and Count() you get exactly that:

```
from django.db.models import Count

dict(<Model>.objects.order_by('name')
    .values_list('name').annotate(name_count=Count('name')))
```

as shown here →

```
{'Dave': 1, 'Jane': 1}
```

When you think about it - it makes a lot of sense - order all the values, then count them - but if you know a little about SQL **GROUP BY**, it's a bit counter intuitive and perhaps more complicated - but it is using GROUP BY under the hood - but it's worth learning.

Annotate is also useful in other ways because you can **supplement objects** with other fields. Combining this with F() allows you create new values on a resulting object set that has math operations applied in the database:

```
<Model>.objects.annotate(new_value=F('number')*3.14159)
```

At this point, you can probably see how all the knowledge of *annotate*, *values*, *values_list*, and *F* can be pieced together to produce objects that simplify application logic, removing unnecessary transformations in your python.

Basic Manager Extensions

Whenever you use “<Model>.objects” you are using the default Manager. A Manager is an abstraction that separates some operations from the model itself making it easier to test, debug and maintain.

It is very common to add class methods to a Model as **convenient accessors**, E.G.

- find_thing
- filter_by_thang
- etc

```
@classmethod
def filter_by_thang(cls, thang):
        return cls.objects.filter(field1=thang)
```

A cleaner solution is to create your own custom Manager (it’s easy and really not as scary as it seems!) and add these methods to it:

```
class YourManager(models.Manager):
        def filter_by_thang(self, thang, …):
                return self.filter(...)

class YourModel(models.Model):
        objects = YourManager()
```

Note, the switch to self here because these are now instance methods on ‘objects’

Because it inherits from django.db.models.Manager, **you get all the existing behaviour, and your new helper without burdening the Model** with this detail. You can use this class to also override and customize get_queryset etc. Thus, creating your own custom manager becomes quite powerful.

Remember, a model can have multiple managers, so it is useful to create a manager for different use cases, customizing the get_queryset() method for each use case. The alternative for multiple managers, is proxy models, but that is a more advanced topic.

Django Docs on Managers

And last but not least…. Editing migrations…

When you first come to Django and learn it's tools to generate migrations from the model's you write, it's easy to get into the habit of just generating migrations, checking it in and forgetting about it.

You might even be afraid of editing them because they are generated, and in any system it's best practice not to edit generated files. Understandable, given changes might be lost if the files were regenerated.

However, Django migrations can safely be edited because each new file is given a new ID**. Migration files aren't regenerated**. Don't be afraid, future migrations won't overwrite your changes, you just get a new file!

Now why would you edit a migration file anyway?

- To handle database adapter differences when multiple database types are used (e.g SQLlite and PG)
  - It's not uncommon that tests use SQLlite but your production uses PostgreSQL. Consider if it's worth it deviating, setup tests to use PG instead
- Renaming tables or fields.
  - Django handles renaming tables and fields really well. But those old names live on. Every time you run tests, you run all migrations. That time accumulates.
  - Consider editing the migrations to only create the tables and fields with the new name, and leave the migration files that rename them as empty migrations. This removes unnecessary work every time you run tests.
- Removing old packages
  - You might change a field type and generate a new migration. E.G. django--jsonfield to Django native JSONField. After doing this, it's impossible to remove the package from your installation until all references are removed, and that includes migrations

Won’t editing migrations break existing databases?

Every database stores state about which migrations have already been run, using the ID.

So…. existing databases will not be affected by editing migration files already executed.

This is why it’s important to leave valid migration files as empty when shuffling around migration logic,

For new databases, (e.g. new installs or when tests run), as long as the migration files build the same end state, there is no difference to the application code.

Do beware though - make sure your “new end state”, in terms of models, fields and types are exactly the same as current databases.

Bonus! Peeking at the SQL

Perhaps you want to do some database tuning and need to see the SQL, or perhaps you are just curious about what the SQL looks like for a Django generated query… There are a couple of ways.

For queryset operations that don't trigger execution of the query, it's really simple - add '.query' to the queryset and stringify it.

```
str(queryset.query)
```

In the dev environment, where you are running with DEBUG=True, you can look at the connection history to see executed SQL, for example the last 10 SQL operations:

```
from django.db import connection

connection.queries[-10:]
```

This returns a list of dicts with 'sql' and 'time' as keys, so you can do some rudimentary performance analysis too.

```
[{'sql': 'SELECT COUNT(*) AS "__count" FROM "app_model"',  'time': '0.001'}]
```

This is the only way to really see executed queries. For production environments, rely on your database to provide visibility.

Django docs for connection

Bonus! Understanding JSON Fields < Django 3.1

JSONField is a cool way to store arbitrary data in the database without many constraints. At KnowledgeHound, we routinely store visualization state as a JSONField. However, coming from a Django 1.x base, we relied heavily on django-jsonfield because there was no built-in support for JSON until v3.1 when the official JSONField was introduced.

We found PostgreSQL was handling the field as a real JSONB type, and we found some odd behaviours where JSON was being stored as a string instead of as a structured JSONB object. We quickly migrated to the official JSONField which recognizes these oddities and returns objects in the serializers we were using the field. The upgrade was simple enough:

Change the Model to use django's official JSONField and generate the migration as normal.

We found everything smooth sailing because there was no actual database schema update.

There are various online stories about how to upgrade/convert, including creating a new field and logically migrating. This makes sense if the database is using a text or VARCHAR type for the field which perhaps earlier versions of django-jsonfield did or maybe that was just custom app logic? We did not need to do this, so check the field type in the database itself before deciding on a path forward.

Creating a new migration creates one new tech debt item - migrations still use django-jsonfield so it can't be removed from your package list/requirements. To completely remove this package from your app, see slide "Editing Migrations" (later)

It should be noted that JSONField uses the PostgreSQL JSONB type. Checkout the references for a link to a comparison between the JSON and JSONB types in PG. There are some unique edge cases, but generally its a viable default. And of course, if you are using something other than PG, evaluate your path validate the path forward.

Recommended Reading/References

Django Queryset Expressions:

https://docs.djangoproject.com/en/4.1/ref/models/expressions

Django Model Field:

https://docs.djangoproject.com/en/4.1/ref/models/fields/

Django Managers: https://docs.djangoproject.com/en/4.1/topics/db/managers/

Django Release Process: https://docs.djangoproject.com/en/dev/internals/release-process/

PostgreSQL Types, (JSON):

https://www.postgresql.org/docs/current/datatype-json.html#:~:text=PostgreSQL%20offers%20two%20types%20for,sets%20of%20values%20as%20input.

Location of these slides and other interesting Survey related topics:

KnowledgeHound Blog

Krisp

If you didn’t hear our 3 year old Puppy during this presentation, you can thank Krisp - an app that does a great job noise cancelling dogs 😁

Visit krisp.ai for more information!

They are not a sponsor or anything, but I love their product - it’s invaluable for those working from home. See my LinkedIn post about it.