Query Syntaxe

Throughout this guide, we'll refer to the following models, which refer to the world geography :

```
class Continent(models.Model):
    name = models.CharField(max_length=255, unique=True)


class Region(models.Model):
    name = models.CharField(max_length=255, unique=True)
    continent = models.ForeignKey(Continent, on_delete=models.CASCADE, related_name="regions")


class Country(models.Model):
    name = models.CharField(max_length=255, unique=True)
    area = models.BigIntegerField()
    population = models.BigIntegerField()
    region = models.ForeignKey(Region, on_delete=models.CASCADE, related_name="countries")


class River(models.Model):
    name = models.CharField(max_length=255, unique=True)
    discharge = models.IntegerField(null=True)
    length = models.IntegerField()
    countries = models.ManyToManyField(Country, related_name="rivers")


class Mountain(models.Model):
    name = models.CharField(max_length=255, unique=True)
    height = models.IntegerField()
    countries = models.ManyToManyField(Country, related_name="mountains")


class Disaster(models.Model):
    event = models.CharField(max_length=255)
    date = models.DateTimeField()
    country = models.ForeignKey(Country, on_delete=models.CASCADE, related_name="disasters")
    source = models.TextField()
    comment = models.TextField()
```

We have `Continent` that are composed of `Region` (*Northern Africa*, *Central America* , *Western Europe*, . . .) which are in turn composed of `Country`.

We then have `River` and `Mountain` that can be in multiple countries, and `Disaster` that can be in only one country.

In the following example, the used URL pattern is `[model]/?[query]`. For instance, if we want to query the continent `Africa` : `continent/?name=Africa`.

Filters

Querying on a model without providing any filter would yield only the first instance of this model, by the ordered defined in the model's class.

Filters are created by using the standard `field=value` of query strings. The `field` portion must correspond to a particular field of the queried model.

You can also query on related model using a dot `.` notation. For example, considering the following data :

```
{
    "event":"Flood",
    "date":"2009-11-04T00:00:00Z",
    "id":1,
    "comment":"...",
    "source":"IFRC",
    "country": {
        "area":591958,
        "id":110,
        "name":"Kenya",
        "population":50221100,
        "region":3,
        "rivers":[...],
        "mountains":[...],
        "disasters":[...]
    }
}
```

In order to find all the disaster in Kenya, on would use the following query string :

- `disaster/?country.name=Kenya`

Related field can be nested up to `DGEQ_MAX_FOREIGN_FIELD_DEPTH` :

- `disaster/?country.region.continent.name=Africa`

If you query directly on a related model, and not on one of its field (E.G. `country` instead of `country.name`), `dgeq` will use it's primary key (most of the time `id`). For instance, the following queries are the same since `id` is the PK of `Continent` :

- `country/?continent=1` / `country/?continent.id=1`

If not specified otherwise, `dgeq` only return the first result found. Append `&c:limit=X` to your query string, where `X` is the number of result you want, for more result :

- `disaster/?country.region.continent.name=Africa&c:limit=20`

We will explain later how `c:limit` works.

Value Types

The value portion can be of different types :

Type	Example	Description
`int`	`?field=2`	Plain integer
`float`	`?field=3.14`	Use dot `.` as decimal separator
`string`	`?field=string`	Plain string
`boolean`	`?field=1`	Use whole numbers (`0` is `False`, anything else is `True`)
`date`	`?field=2004-12-02T22:00`	An ISO 8601 compliant string
`null`	`?field=`	Do not put any value

Search modifier

A modifier may be used in front of the value portion of the query string to better filter the results. Only one modifier may be used, the second modifier character would be considered to be part of the value.

Modifier	Example	Description
`<`	`country/?population=<500000`	Less than
`[`	`country/?population=]500000`	Less than or equal
`>`	`country/?population=>500000`	Greater than
`]`	`country/?population=[500000`	Greater than or equal
`!`	`country/?population=!500000`	Different than
`^`	`country/?name=^United`	Start with a string
`$`	`country/?name=$Islands`	End with a string
`*`	`country/?name=*istan`	Contain a string
`~`	`country/?name=~z`	Do not contain a string

To combine search modifier, either use the comma `,` : `country/?population=[500000,]500000`, or create another `field=value` with the other modifier : `country/?population=[500000&population=]500000`

Modifiers are combined with a logical `AND`. For instance to get all the country with their name starting with `United`, but that does not contains `States` :

- `country/?name=^United,~States` or `country/?name=^United&name=~States`

By default, string search are case-sensitive, we will see later how to change that behaviour.

Commands

A command is a particular query string that allow a finer control over the resulting query. These are provided as query string attributes but are namespaced with `c:` to distinguish them from filters. Some of these commands cannot be used together, such as `c:count` and `c:aggregate`.

Command	Example	Description
`c:show`	`country/c:show=name,id`	Only include the provided fields (comma `,` separated list).
`c:hide`	`country/c:hide=id,area`	Include all field except the provided fields (comma `,` separated list). Will be ignored if `c:show` is present.
`c:sort`	`country/c:sort=-area,id`	Sort the results by the provided fields (comma `,` separated list). Prepend an hyphen `-` to use descending order on a specific field.
`c:case`	`country/c:case=0`	Set whether a search should be case-sensitive (`1`) or not (`0`). Default to `1`.
`c:limit`	`country/c:limit=20`	Limit the results to at most `X` objects (default to `1`).
`c:start`	`country/c:start=10`	Start with the `Nth` object within the results of the query (first one is `0`). Combine with `c:start` to obtain a precise subset. For instance, using `c:start=10&c:limit=10` would yield the `10th` to `20th` objects. Default to `0`
`c:time`	`country/c:time=1`	Shows the time taken server-side in seconds to process your request.

Command	Example	Description
`c:count`	`country/c:count=1`	Return the number of found item in the field `count` of the response. `0` to not count (default), `1` to count, `2` to get only the count and not the results (`results` field of the response will be an empty list).
`c:related`	`country/c:related=0`	Allow to hide (`0`) related field (`ForeignKey`, `ManyToManyField` and their related fields) anywhere in the result. Massively reduce time taken by the request. Default is `1`.
`c:query`	`country/c:query=1`	Return the SQL query used to get the result minus the part retrieving related models' FK. Returned string is in the field `query` of the response.
`c:queryset`	`country/c:queryset=1`	Return a string representing the django' queryset used to get the result minus the part retrieving related models' FK. Returned string is in the field `queryset` of the response.
`c:aggregate`	`See below`	Described below this table
`c:annotate`	`See below`	Described below this table
`c:join`	`See below`	Described below this table

Note that the order of commands and filters within the query string does not matter.

`c:aggregate`

Sometimes you will need to retrieve values that are computed by summarizing or aggregating a collection of objects, you can use `c:aggregate` for that. The syntax is :

Aggregate are made up of key value pairs delimited by `^` : `key:value^key:value`. Valid keys are :

Key	Example	Description
`field`	`field=population`	Name of the field used for to compute the aggregation.
`to`	`to=population_avg`	Name of the field where the result of the aggregation will be displayed.
`func`	`func=avg`	Function to use for the aggregation.

- `max` - Maximum value of a field
- `min` - Minimum value of a field
- `sum` - Sum of a field
- `avg` - Average of a field
- `stddev`- Standard deviation of a field
- `var` - Variance of a field

You can declare multiple aggregate using a comma `,`. Each aggregate's `to` must be unique.

For instance, if you need the maximum, minimum and average population of countries in Asia: :

- `country/?region.continent.name=Asia&c:limit=100&c:aggregate=field=population^func=avg^to=populatior`

Aggregation can also be done on model related to the one being queried using dot `.` notation. Here the average height of mountains in France as an example :

- `country/?name=France&c:limit=100&c:aggregate=field=mountains.height^func=avg^to=mountain_avg`

`c:annotate`

Annotations are like aggregations, but over each item of the result. For instance, annotation allow you to get the average length of the rivers inside each country.

Annotation is declared the same way as aggregation (`key:value^key:value`) but with more keywords:

Key	Example	Description
`field`	`field=population`	Name of the field used for to compute the aggregation.
`to`	`to=population_avg`	Name of the field where the result of the aggregation will be displayed.
`func`	`func=avg`	Function to use for the aggregation.
`filters`	`filters=mountains.height=]1500'mountains.name=*Mount`	Allow to add an apostrophe `'` separated list of filters. These filters supports `search modifiers`.
`late`	`late=1` or `late=0`	Whether the annotation will be applied before (`0`) or after (`1`) the filtering of the main query. Default is `0`.

Annotations have the same functions as aggregations, and can also be done on model related to the one being queried using dot `.` notation.

Filters must be given related to the main query model, and not the model used for the annotation. So if you have a query on `country/` and want to annotate on `rivers` count your query must be :

- `country/?c:annotate=field=rivers^to=rivers_count^func=count^filters=rivers.length=>2000`

and not:

- `country/?c:annotate=field=rivers^to=rivers_count^func=count^filters=length=>2000`

note the field used in `filters`.

Field created by annotations on `to` can be used in other commands, such as `c:sort`, `c:show` and even `c:aggregate`. They can also be used in filters, making it possible to filter on rivers average for instance.

An important thing to note is the `late` option. By default, annotations are done before the filtering of the main query, this can be change by settings `late=1` so that specific annotations are done after the filtering. This is useful since the result can drastically change according to the order of these two commands.

Let's see an example given 3 countries named `A`, `B` and `C` :

- Country `A` has two rivers with a length of 400 and 500 meters.
- Country `B` has two rivers with a length of 100 and 400.
- Country `C` has one rivers with a length of 100 meters

Now let's try counting the number of rivers in countries, filtering for rivers with length greater than 300.

- `country/?rivers.length=>300&c:show=name,rivers_count&c:annotate=field=rivers^func=count^to=rivers_c`


```
{
    "result": [
        {
            "name": "A",
            "rivers_count": 2
        },
        {
            "name": "B",
            "rivers_count": 2
        }
    ]
}
```


- `country/?rivers.length=>300&c:show=name,rivers_count&c:annotate=field=rivers^func=count^to=rivers_c`


```
{
    "result": [
        {
            "name": "A",
            "rivers_count": 2
        },
        {
            "name": "B",
            "rivers_count": 1
        }
    ]
}
```


Both queries return a list of countries that have at least one river with a length exceeding 300 meters, hence country `C` is excluded.

In the first query, the annotation precedes the filter, so the filter has no effect on the annotation.

The second query counts the number of rivers that have a length exceeding 300 meters for each country. The filter precedes the annotation, so the filter constrains the objects considered when calculating the annotation.

Another consequences of using `late=1` is that the field created on `to` cannot be used as a filtering term in the main query.

Let's see some examples of annotations:

- Country sorted (desc) by their longest river : `country/?c:limit=500&c:annotate=field=rivers.length^to=river_le`
- Country with at least 5 mountains taller than 2000 meters : `country/?c:limit=500&c:annotate=field=mountains^to=`
- Population of each continent : `continent/?c:limit=10&c:annotate=field=regions.countries.population^func=s`
- Average number of mountain in a country in the world: `country/?c:limit=500&c:annotate=field=mountains^to=mou`

`c:join`

The default behaviour of the API is to not resolve related models. Only their primary key will retrieved.

The `c:join` command allow to retrieve these models, that is retrieving their fields instead of just their `pk` in the result.

A join is made up of key value pairs delimited by `^` : `key:value^key:value`. Valid keys are :

Key	Example	Description
`field`	`field=continent`	Name of the field containing the related model.
`show`	`show=name'id`	Only include the provided fields from the object within the result (multiple field names separated by an apostrophe `'`).
`hide`	`hide=id'countries`	Include all field except the provided fields from the object within the result (multiple field names separated by an apostrophe `'`). Will be ignored if `show` is present.

The following keys only make sense when `field` is either a `ManyToManyField`, its related field, or the related field of a `ForeignKey`

Key	Example	Description
`start`	`start=10`	Start with the `Nth` object within the join (first is `0`). Default to `0`.
`limit`	`limit=20`	Limit the number of models in the join (default to all).
`sort`	`sort=-area'id`	Sort the joined models by the given field (apostrophe `'` separated list)
`filters`	`filters=rivers=[1000'mountains=<3000`	Use `filters` to add an apostrophe `'` separated list of filters. These filters supports `search modifiers`.

Here some example :

- Join the field `regions` of the model `Continent`, hiding their countries :`continent/?c:join=field=regions^hide=count`
- Join every earthquake of Japan : `country/?name=Japan&c:join=field=disasters^filters=event=*arthquake`
- Join the second highest mountain of China : `country/?name=China&c:join=field=mountains^show=name^start=2^li`

Note that you can do nested join using dot `.`. For instance to get the `Region` of a `Disaster` :

- `disaster/?id=1&c:join=field=country.region`

In this case, the field `country` will also be joined, but only its field `region` will be in the result. If you want to get other field, you must also join this field on its own :

- `disaster/?id=1&c:join=field=country,field=country.region`

The order of joins does not matter, these two request give the same result :

- `disaster/?id=1&c:join=field=country,field=country.region`

- `disaster/?id=1&c:join=field=country.region,field=country`