djaq

Release 0.1.0

Paul Wolf

May 03, 2023

CONTENTS:

Djaq - pronounced “Jack” - is an alternative to the Django QuerySet API.

What sets it apart:

- No need to import models
- Clearer, more natural query syntax
- More powerful expressions for output
- More powerful, easier-to-write filter expressions
- More consistent query syntax without resorting to hacks like `F()` expressions, `annotate()`, `aggregate()`
- Column expressions are entirely evaluated in the database
- Extensible: you can write your own functions
- Pandas: Easily turn a query into a Pandas Dataframe

There is also a JSON representation of queries, so you can send queries from a client. It’s an instant API to your data. No need to write backend classes and serializers.

Djaq queries are strings. A query string for our example dataset might look like this:

```
DQ("Book", "name as title, publisher.name as publisher").go()
```

This retrieves a list of book titles with book publisher. But you can formulate far more sophisticated queries; see below. You can send Djaq queries from any language, Java, Javascript, golang, etc. to a Django application and get results as JSON. In contrast to REST frameworks, you have natural access to the Django ORM from the client.

Djaq sits on top of the Django ORM. It can happily be used alongside QuerySets.

CHAPTER

ONE

INSTALLATION

You need Python 3.6 or higher and Django 2.1 or higher.

Install:

```
pip install Djaq
```

The bleeding edge experience:

```
pip install https://github.com/paul-wolf/djaq/archive/master.zip
```

Now you can call the Djaq API:

```
from djaq import DjaqQuery as DQ

print(list(DQ("Book", "name as title, publisher.name as publisher").dicts()))

[{'title': 'Name grow along.', 'publisher': 'Long, Lewis and Wright'}, {'title': 'We pay␣
↪single record.', 'publisher':\
'Long, Lewis and Wright'}, {'title': 'Natural develop available manager.', 'publisher':
↪'Long, Lewis and Wright'}, {'\
title': 'Fight task international.', 'publisher': 'Long, Lewis and Wright'}, {'title':
↪'Discover floor phone.', 'publi\
sher': 'Long, Lewis and Wright'}]
```

1.1 Providing a Remote API

We'll assume below you are installing the Djaq UI. This is not required to provide an API but is useful to try things out.

Install the API and UI in settings:

```
INSTALLED_APPS = (
    '...',
    djaq.djaq_api,
    djaq.djaq_ui,
)
```

Configure urls in urls.py:

```
urlpatterns = [
    '...',
    path("dquery/", include("djaq.djaq_ui.urls")),
    path("djaq/", include("djaq.djaq_api.urls")),
]
```

You are done. You can start sending requests to:

```
/djaq/api/request/
```

The UI will be available at:

```
/dquery
```

Note the UI will send requests to the API endpoint so will not work without that being configured. You send a request in this form to the api endpoint:

```
{
    "queries": [
        {
            "model": "books.Book",
            "output": "id,\nname,\npages,\nprice,\nrating,\npublisher,\nalt_publisher,\
↪npubdate,\nin_print,\n",
            "where": "",
            "order_by": "",
            "limit": "100",
            "offset": "0"
        }
    ]
}
```

The UI will create this JSON for you if you want to avoid typing it.

You can also create objects, update them and delete them:

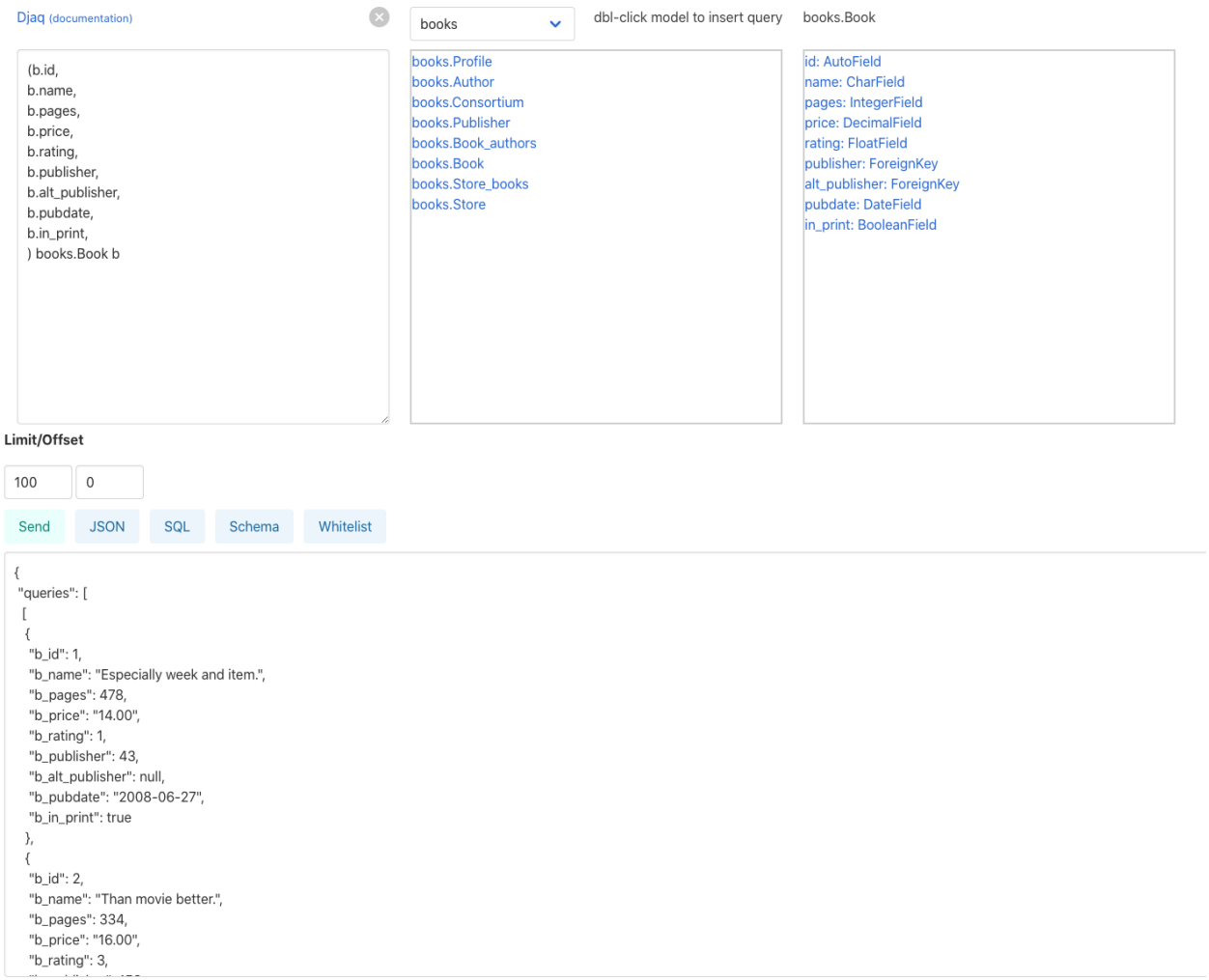
```
{
    "queries": [
        {
            "model": "books.Book",
            "output": "id,\nname,\npages,\nprice,\nrating,\npublisher,\nalt_publisher,\
↪npubdate,\nin_print,\n",
            "where": "",
            "order_by": "",
            "limit": "100",
            "offset": "0"
        }
    ],
    "creates": [
        {
            "model": "Book",
            "fields": {
                "name": "my new book"
            }
        }
```

(continues on next page)

(continued from previous page)

```
    ],
    "updates": [
        {
            "model": "Book",
            "pk": 37,
            "fields": {
                "name": "my new title"
            }
        }
    ],
    "deletes": [
        {
            "model": "Book",
            "pk": 37
        }
    ]
}
```

You can send multiple *queries*, *creates*, *updates*, *deletes* operations in a single request.



CHAPTER

TWO

SETTINGS

The API and UI will use the following settings:

- DJAQ_WHITELIST: a list of apps/models that the user is permitted to include in queries.
- DJAQ_PERMISSIONS: permissions required for staff and superuser.
- DJAQ_VALIDATOR: if using the remote API, you can specify a validator class to handle all requests. The value assigned must be a class derived from *djaq.query.ContextValidator*. The *request* object is always added to the context by default. You can examine this in the validator to make decisions like forbidding access to some users, etc.

In the following example, we allow the models from 'books' to be exposed as well as the *User* model. We also require the caller to be both a staff member and superuser:

```
DJAQ_WHITELIST = {
    "django.contrib.auth": ["User"],
    "books": [
    "Profile",
    "Author",
    "Consortium",
    "Publisher",
    "Book_authors",
    "Book",
    "Store_books",
    "Store",
    ],
}
DJAQ_PERMISSIONS = {
    "creates": True,
    "updates": True,
    "deletes": True,
    "staff": True,
    "superuser": True,
}
```

If we want to allow all models for an app, we can leave away the list of models. This will have the same effect as the setting above.

```
DJAQ_WHITELIST = {
    "django.contrib.auth": ["User"],
    "books": [],
}
```

For permissions, you can optionally require any requesting user to be staff and/or superuser. And you can deny or allow update operations. If you do not provide explicit permissions for update operations, the API will respond with 401 if one of those operations is attempted.

CHAPTER

THREE

QUERY USAGE GUIDE

Throughout, we use models somewhat like those from Django's bookshop example:

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

class Publisher(models.Model):
    name = models.CharField(max_length=300)

class Book(models.Model):
 name = models.CharField(max_length=300)
 pages = models.IntegerField()
 price = models.DecimalField(max_digits=10, decimal_places=2)
 rating = models.FloatField()
 authors = models.ManyToManyField(Author)
 publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)
 alt_publisher = models.ForeignKey(
     Publisher, related_name="alt_publisher", on_delete=models.CASCADE, null=True
 )
 pubdate = models.DateField()
 in_print = models.BooleanField(default=True)

class Store(models.Model):
    name = models.CharField(max_length=300)
    books = models.ManyToManyField(Book)
```

These examples use auto-generated titles and names and we have a slightly more complicated set of models than shown above.

The first thing you want to do is import the DjaqQuery class which we do with an alias:

```
from djaq import DjaqQuery as DQ
```

Let's get book title (name), price, discounted price, amount of discount and publisher name wherever the price is over 5.

```
result = \
  list(DQ("Book", """name,
    price as price,
```

(continues on next page)

(continued from previous page)

```
    0.2 as discount,
    price * 0.2 as discount_price,
    price - (price*0.2) as diff,
    publisher.name
    """
  ).where("price > 5").dicts())
```

`result` now contains a list of dicts each of which is a row in the result set. One example:

```
[{'name': 'Address such conference.',
  'price': Decimal('99.01'),
  'discount': Decimal('0.2'),
  'discount_price': Decimal('19.802'),
  'diff': Decimal('79.208'),
  'publisher_name': 'Arnold Inc'}]
```

Here is the structure of the syntax:

```
DjaqQuery([model_name|model], [<field_exp1>, ...])
.where(<condition_expression>)
.order_by(<field_exp1>, ...)
```

Whitespace does not matter too much. You could put things on separate lines.

Note as well that we usually in this tutorial use the *.go()* convenience method. The following two calls are pretty much equivalent:

```
DQ("Book", "name").go()

list(DQ("Book", "name").dicts())
```

The column expressions can be Django Model fields or arithmetic expressions or any expression supported by functions of your underlying database that are also whitelisted by Djaq. Postgresql has thousands of functions. About 350 of those are available in Djaq.

The syntax is similar to Python. Fields are identifiers that must be like Python identifiers, which they will be since we are referencing Django Models.

- `select source`: a comma seperated list column expressions. This can as well be a `list` of column expressions.
- `where`: an expression that evaluates to a boolean value; the same as Django `QuerySet.filter()` but with Djaq syntax
- `order_by`: a comma seperated list of column expressions, each of which can be prepended with minus, `-`, to indicate descending order rather than the default ascending order. This can as well be a `list` of column expressions.

Column expressions can be composed of multiple nested parenthetical expressions and conjoining boolean operators:

- `and`
- `or`

Comparisons:

>, <, <>, <=, >=

Equality:

==, !=

List membership:

`in`, `not in`

Identity:

`is`, `is not`

these can only be used with boolean values:

```
"in_print is True"
"in_print is not True"
```

We do not support this usage of `not`:

```
# Does not work
"not id == 3"
```

For which use:

```
"id != 3"
```

Columns are automatically given names. But you can give them your own name:

```
DQ("Book", "name as title, price as price, publisher.name as publisher").go()
```

or if we want to filter and get only books over 5 in price:

```
DQ("Book", "name as title, price as price, publisher.name as publisher") \
   .where("price > 5") \
   .go()
```

The following filter:

```
DQ("Book", "publisher.name as publisher" ).where("price > 5 and ilike(publisher.name, 'A%
↪')").go()
```

will be translated to SQL:

```
SELECT "books_publisher"."name" FROM books_book LEFT JOIN books_publisher  ON ("books_
↪book"."publisher_id" = "books_publisher"."id")
 WHERE ("books_book"."price" > 5 AND "books_publisher"."name" ILIKE \'A%\')'
```

Our example model also has an owner model called "Consortium" that is the owner of the publisher:

```
DQ("Book", "name, price, publisher.name, publisher.owner.name").limit(1).go()
[{'b_name': 'Range total author impact.', 'b_price': Decimal('12.00'), 'b_publisher_name
↪': 'Wright, Taylor and Fitzpatrick', 'b_publisher_owner_name': 'Publishers Group'}]
```

Check what SQL is generated:

```
In [20]:    DQ("Book", "name, price, publisher.name, publisher.owner.name").limit(1).
↪sql()
Out[20]: 'SELECT "books_book"."name", "books_book"."price", "books_publisher"."name",
↪"books_consortium"."name" FROM books_book LEFT JOIN books_publisher  ON ("books_book".
```

(continues on next page)

(continued from previous page)

```
↪"publisher_id" = "books_publisher"."id")  LEFT JOIN books_consortium  ON ("books_
↪publisher"."owner_id" = "books_consortium"."id")  LIMIT 1'
```

Signal that you want to summarise results using an aggregate function:

```
DQ("Book", "publisher.name as publisher, count(id) as book_count").go()

[
    {
        "publisher": "Martinez, Clark and Banks",
        "book_count": 6
    },
    {
        "publisher": "Fischer-Casey",
        "book_count": 9
    },
    etc.
]
```

Order by name:

```
DQ("Book", "name, price, publisher.name as publisher") \
.where("price > 5") \
.order_by("name") \
.go()
```

Get average, minimum and maximum prices:

```
DQ("Book", "avg(price) as average, min(price) as minimum, max(price) as maximum").go()
[
   {
      "average": "18.5287169247794985",
      "minimum": "3.00",
      "maximum": "99.01"
   }
]
```

Count all books:

```
DQ("Book", "count(id)").value()

1000
```

Get unique results with `distinct()`:

```
DQ("Book", "pubdate.year").where("regex(name, 'B.*') and pubdate.year > 2013").
↪distinct().order_by("-pubdate.year").go()
```

You can qualify model names with the app name or label used in *apps.py*:

```
DQ("books.Book", "name, publisher.name")
```

You'll need this if you have models from different apps with the same name.

To pass parameters, use variables in your query, like `{myvar}`:

```
In [30]: oldest = '2018-12-20'
    ...: list(DQ("Book", "name, pubdate").where("pubdate >= {oldest}").context({"oldest
↪": oldest}).limit(5).tuples())
Out[30]:
[('Available exactly blood.', datetime.date(2018, 12, 20)),
 ('Indicate Congress none always.', datetime.date(2018, 12, 24)),
 ('Old beautiful three program.', datetime.date(2018, 12, 25)),
 ('Oil onto mission.', datetime.date(2018, 12, 21)),
 ('Key same effect me.', datetime.date(2018, 12, 23))]
```

Notice that variables are not f-string placeholders! Avoid using f-strings to interpolate arguments.

CHAPTER

FOUR

API REFERENCE

```
DjaqQuery(
        model_source: Union[models.Model, str],
        select_source: Union[str, List, None] = None,
        name: str = None,
        whitelist=None,
    )
```

Construct a DjaqQuery object.

- model_source: the Django model as a string, optionally with label qualifier or a Model class.
- select_source: the output column expressions that may be aliased to a name with `.. as my_name`. This argument can be a string that separates the output columns with columns or a `list` with a column definition per element.
- name: provide a name for the query for use later in subqueries
- whitelist: provide a list of models to allow

4.1 context(context: Dict) -> DjaqQuery

- context: a dict that contains parameter names and values.

4.2 conditions(node: B) -> DjaqQuery

- node: an objects of class `B()` that encapsulates a filter condition which can be a complex, nested object.

4.3 count() -> int

Count the result set.

4.4 csv()

Return a generator that returns a comma separated value representation of the result set.

4.5 get(pk_value: any) -> Model

Return a Django model instance whose primary key is `pk_value`.

4.6 go()

A shortcut for `list(dq.dicts())`.

4.7 distinct()

Make results unique.

4.8 dicts()

Return a generator that returns dictionary representations of the result set.

4.9 json()

Return a generator that returns JSON representations of the result set.

4.10 limit(limit: int) -> DjaqQuery

Restrict the results to `limit` items.

4.11 objs()

Return a generator that returns objects of type `DQResult` that are essentially named tuples of the result set.

4.12 offset(offset: int) -> DjaqQuery

Start the results at `offset` of the filtered result set.

4.13 map(result_type: Union[callable, dataclasses.dataclass], data=None)

Return a generator that produces a result of a type specified by the dataclass of `result_type` or returns the type returned by the callable.

When using a dataclass, Djaq will try to map field names of the results to the dataclass field names. The names must be exact matches. Use `... as my_name` in your column definitions to get these match up.

When using a callable, your functional or other callable will receive a dict representation of the result set and you can return whatever you wish.

- result_type: a callable or dataclass
- data: the context dict for the query

4.14 order_by() -> DjaqQuery

4.15 qs() -> QuerySet

Return a Django `QuerySet` class with the filtered results. This QuerySet is exact what you get from `QuerySet.raw()`

4.16 rewind() -> DjaqQuery

This will reset the cursor if you have already started to iterate the results with one of the generator methods.

4.17 schema -> Dict

A property that returns a dict representing the schema of a model. Use like this:

```
DQ("Book").schema
```

4.18 schema_all(connection=None) -> Dict

A class method that return a dict of the schema for all models.

```
DQ.schema_all()
```

You can pass it the connection name optionally.

4.19 sql() -> str

Return the SQL for the DjaqQuery.

4.20 tuples()

Return a generator that returns objects of type `Tuple` for the result set.

4.21 update_object(pk_value: any, update_function: callable, data: Dict, save=True)

This will update the object whose primary key is `pk_value` by calling `update_function()`, returning whatever the return value is of that callable.

4.22 value()

Return the first field of the first record of the result set. This mainly only makes sense when aggregating.

4.23 where(node: Union[str, B]) -> DjaqQuery

Define a filter condition for the query.

CHAPTER

FIVE

DJAQ MANAGEMENT COMMAND

If you include the `djaq_ui` app in your INSTALLED_APPS, you have the `./manage.py djaq` command at your disposal:

```
./manage.py djaq Book --output "pubdate.year" --where "pubdate.year > 2013" --order_by "␣
↪-pubdate.year"
```

Most features are available:

```
 ./manage.py djaq
usage: manage.py djaq [-h] [--output OUTPUT] [--where WHERE] [--order_by ORDER_BY] [--
↪format FORMAT] [--schema] [--dataclass] [--limit LIMIT]
                      [--offset OFFSET] [--distinct] [--sql] [--count] [--version] [-v {0,
↪1,2,3}] [--settings SETTINGS] [--pythonpath PYTHONPATH]
                      [--traceback] [--no-color] [--force-color] [--skip-checks]
                      model
```

Most options are obvious, but some or not related to queries. `--dataclass` is use to generate code for a dataclass corresponding to the given Django model:

```
 ./manage.py djaq Book --dataclass
@dataclass
class BookEntity:
    id: int
    name: str
    pages: int
    price: Decimal
    rating: int
    publisher: int
    alt_publisher: int
    pubdate: datetime.date
    in_print: bool
```

It's not very sophisticated but should save some typing.

CHAPTER

SIX

RESULT FORMATS

There are serveral ways to get results from a DjangoQuery:

- `dataframe()`: returns a pandas `DataFrame()` if pandas is installed
- `dicts()`: returns a generator yielding a dict for each result row
- `tuples()`: returns a generator yielding a tuple for each result row
- `objs()`: returns a generator that yields a `DQResult` object which is basically a named tuple
- `csv()`: returns a string that represents the entire csv document
- `qs()`: returns Django model instances

CHAPTER

SEVEN

PANDAS DATAFRAME

Djaq will return a pandas DataFrame provided that pandas is installed (`pip install pandas`):

```
In [1]: from djaq import DjaqQuery as DQ


In [5]: df = DQ("Book",
...:     """name as Booktitle,
...:     price as price,
...:     0.2 as discount,
...:     price * 0.2 as discount_price,
...:     price - (price*0.2) as diff,
...:     publisher.name
...:    """).where("price > 5").dataframe()

In [5]: df.head()
Out[5]:
                          b_name  price  discount  discount_price  diff          ␣
↪publisher_name
0           Especially week and item.   14.0       0.2             2.8  11.2            ␣
↪Sanchez-Tran
1                  Than movie better.   16.0       0.2             3.2  12.8            ␣
↪   Scott Inc
2      Add marriage sport side above.   23.0       0.2             4.6  18.4          ␣
↪Patrick-Carlson
3  Central federal knowledge any one.   18.0       0.2             3.6  14.4  Hicks,␣
↪Gray and Griffin
4                    Price size fast.   16.0       0.2             3.2  12.8          ␣
↪Murphy-Martinez
```

If pandas is not installed, an error will occur. If you are not using this feature, you do not need to install pandas.

CHAPTER

EIGHT

CONDITIONS

Condition objects are like the `Q()` class in Django QuerySets. The class is `B()` for Boolean. You can combine various expressions using the Python boolean operators:

```
c = (B("regex(name, {name}") & B("pages > {pages}")) |  B("rating > {rating}")
```

You then add the condition to a DjaqQuery:

```
DQ("Book", "my query...").conditions(c)
```

If you are using variable substitution as in this example, you'll want to pass context data. This might be from a Django request object (though it can be from any dict-like object).

It is a special feature that if there is a `B()` expression that has a variable like `pages` and the context is missing a variable called `pages`, that `B()` expression will be dropped from the final generated SQL.

The purpose of this is to provide a filter expression that is conditional on the presence of data on which it depends. If you have an html form with fields that might or might not be filled by the user to filter the data, you may want to implement a logic that says "if "name" is provided, search in the name field. If it is not provided, the user does not want to search by name.

When you write your conditional expressions, these are what would normally go into the filter part of the query:

```
rating = 2
price = 10
pages = 700
name = "Dr.*"
DQ("Book", """name as name,
price as price,
rating as rating,
pages as pages,
publisher.name as publisher
""") \
.where("regex(name, {name}) and pages > {pages} and rating > {rating} and price > {price}
↪") \
.context({"rating": rating, "price": price, "pages": pages, "name": name})
```

In the following example, it is not required to provide data for all fields, name, pages, rating, price. The conditional expressions will be refactored to accommodate only those expressions that have data provided.

```
from djaq import B
def book_list(request):

    c = (
```

(continues on next page)

(continued from previous page)

```
        B("regex(name, {name})")
        & B("pages > {pages}")
        & B("rating > {rating}")
        & B("price > {price}")
    )

    books = list(
        DQ("Book",
            "name as name, price as price, rating as rating, pages as pages, publisher.
↪name as publisher",
        )
        .conditions(c)           # add our conditions here
        .context(request.POST)   # add our context data here
        .limit(20)
        .dicts()
    )
    return render(request, "book_list.html", {"books": books})
```

You can check how your conditional expressions will look depending on the context data:

```
In [1]: from djaq.query import render_conditions
In [2]: from djaq.conditions import B
In [3]: c = B("regex(name, {name})") & B("pages > {pages}") & B("rating > {rating}") & B(
↪"price > '$(price)'")
In [4]: ctx = {"name": "sample", "pages": 300}
In [5]: render_conditions(c, ctx)
...:
Out[5]: "(((regex(b.name, {name}) and pages > {pages})))"
```

CHAPTER

NINE

FUNCTIONS

If a function is not defined by DjaqQuery, then the function name is checked with a whitelist of functions. There are approximately 350 functions available. These are currently only supported for Postgresql and only those will work that don't use syntax that is special to Postgresql. Additionally, the Postgis functions are only available if you have installed Postgis.

A user can define new functions at any time by adding to the custom functions. Here's an example of adding a regex matching function:

```
from djaq import djaq_functions
djaq_functions["REGEX"] = "{} ~ {}"
```

Now find all book names starting with 'B':

```
DQ("Book", "name").where("regex(name, 'B.*')").go()
```

We always want to use upper case for the function name when defining the function. Usage of a function is then case-insensitive. You may wish to make sure you are not over-writing existing functions. "REGEX" already exists, for instance.

You can also provide a `callable` to `DjaqQuery.functions`. The callable needs to take two arguments: the function name and a list of positional parameters and it must return SQL as a string that can either represent a column expression or some value expression from the underlying backend.

In the following:

```
DQ("Book", "name").where("like(upper(name), upper({name_search}))").context({"name_search
↪": name_search}).go()
```

`like()` is a Djaq-defined function that is converted to `field LIKE string`. Whereas `upper()` is sent to the underlying database because it's a common SQL function. Any function can be created or existing functions mutated by updating the `DjaqQuery.functions` dict where the key is the upper case function name and the value is a template string with `{}` placeholders. Arguments are positionally interpolated.

Above, we provided this example:

```
DQ("Book", """
   sum(iif(rating < 5, rating, 0)) as below_5,
   sum(iif(rating >= 5, rating, 0)) as above_5
   """)
```

We can simplify further by creating a new function. The IIF function is defined like this:

```
"CASE WHEN {} THEN {} ELSE {} END"
```

We can create a SUMIF function like this:

```
from djaq import djaq_functions
djaq_functions['SUMIF'] = "SUM(CASE WHEN {} THEN {} ELSE {} END)"
```

Now we can rewrite the above like this:

```
DQ("Book", """
    sumif(rating < 5, rating, 0) as below_5,
    sumif(rating >= 5, rating, 0) as above_5
    """)
```

Here's an example providing a function:

```
def concat(funcname, args):
    """Return args spliced by sql concat operator."""
    return " || ".join(args)

DjaqQuery.functions['CONCAT'] = concat
```

CHAPTER

TEN

COLUMN EXPRESSIONS

Doing column arithmetic is supported directly in the query syntax:

```
discount = 0.2
DQ("Book", """name,
    price as price,
    {discount} as discount,
    price * {discount} as discount_price,
    price - (price * {discount}) as diff
    """).context({"discount": discount})
```

These expressions are evaluated in the database.

You can use literals:

```
In [71]: DQ("Book", "name, 'great read'").limit(1).go()
Out[71]: [{'name': 'Station many chair pressure.', 'great_read': 'great read'}]
```

You can use the common operators and functions of your underlying db.

The usual arithmetic:

```
In [72]: DQ("Book", "name, 1+1").limit(1).go()
Out[72]: [{'name': 'Station many chair pressure.', '11': 2}]

In [38]: list(DQ("Book", "name, 2.0/4").limit(1).tuples())
Out[38]: [('Range total author impact.', Decimal('0.50000000000000000000'))]

In [44]: list(DQ("Book", "2*3").limit(1).tuples())
Out[44]: [(6,)]
```

Modulo:

```
In [55]: list(DQ("Book", "mod(4.0,3)").limit(1).tuples())
Out[55]: [(Decimal('1.0'),)]
```

Comparison as a boolean expression:

```
In [121]: DQ("Book", "2 > 3 as absurd").limit(1).go()
Out[121]: [{'absurd': False}]
```

While the syntax has a superficial resemblance to Python, you do not have access to any functions of the Python Standard Libary.

CHAPTER

ELEVEN

SUBQUERIES AND IN CLAUSE

You can reference subqueries within a Djaq query.

```
v_sub = DQ(Book, "id", name="v_sub").where("name == 'B*'")  # noqa: F841
DQ(Book, "name, price").where("id in '@v_sub'").go()
```

This evaluates to:

```
SELECT "books_book"."name", "books_book"."price"
   FROM books_book WHERE "books_book"."id"
   IN (SELECT "books_book"."id" FROM books_book WHERE "books_book"."name" LIKE \'B%\')
```

Note that user of single quotes and prepending the sub query name with @: `'@queryname'`

Make sure your subquery returns a single output column.

```
DQ("Book", "id").where("name == 'B*'", name='dq_sub')
dq = DQ("Book", "name, price").where("id in '@dq_sub'")
```

As with QuerySets it is nearly always faster to generate a sub query than use an itemised list.

If your subquery has parameters, these need to be supplied to the using query:

```
DQ("Book", "id", name="dq_sub").where("ilike(name, {spec})")
DQ("Book", "name, price").where("id in '@dq_sub'").context({"spec": "B%"})
```

CHAPTER

TWELVE

ORDER BY

You can order_by like this:

```
DQ("Book", "id").where("price > 20").order_by("name")
```

Descending order:

```
DQ("Book", "id").where("price > 20").order_by("-name")
```

You can have multple order by expressions.

```
DQ("Book", "name, publisher.name").where("price > 20").order_by("-name, publisher.name")
```

CHAPTER

THIRTEEN

TRUE, FALSE, NULL, EMPTY

`None`, `True`, `False` are replaced in SQL with `NULL`, `TRUE`, `FALSE`. All of the following work:

```
DQ("Book", "id, name").where("in_print is True")
DQ("Book", "id, name").where("in_print is not True")
DQ("Book", "id, name").where("in_print is False")
DQ("Book", "id, name").where("in_print == True")
```

To test for `NULL`, use `None`:

```
DQ("Book", "id, name").where("name is not None")
```

If you want to test for an empty or non-empty string, use `LENGTH()`:

```
DQ("Book", "id, name").where("length(name) > 0")
```

CHAPTER

FOURTEEN

DATETIMES

Datetimes are provided as strings in the iso format that your backend expects, like `2019-01-01 18:00:00`.

```
DQ("Book", "pubdate").where("pubdate > '2021-01-01'").go()
```

Get the difference between two dates:

```
DQ("Book", "pubdate, age({now}, pubdate) as age").context({"now": timezone.now()}).go()
```

You can access fields of a date, like `year`, `month`, `day`:

```
DQ("Book", "name, publisher.name, pubdate.year").where("pubdate.year < 2022").go()
```

CHAPTER

FIFTEEN

COUNT

There are a couple ways to count results. These both return the exact same thing:

```
DQ("Book").count()
DQ("Book", "count(id)").value()
```

CHAPTER

SIXTEEN

OFFSET, LIMIT, PAGING, SLICING

You can use `limit()` and `offset()` to page your results:

```
DjaqQuery("...").offset(1000).limit(100).tuples()
```

Which will provide you with the first hundred results starting from the 1000th record.

You cannot slice a DjaqQuery because this would frustrate a design goal of Djaq to provide the performance advantages of cursor-like behaviour and explicit semantics.

But you can slice the result of the QuerySet method:

```
DQ("Book").qs()[10:20]
```

CHAPTER

SEVENTEEN

SCHEMA

There is a function to get the schema available to a calling client:

```
from djaq.app_utils import get_schema
print(get_schema())
```

Pass the same whitelist you use for exposing the query endpoint:

```
wl = {"books": []}
print(get_schema(whitelist=wl))
```

You can get the schema from the DQ object as well:

```
DQ("Book").schema_all()
```

Or a specific model:

```
DQ("Book").schema
```

CHAPTER

EIGHTEEN

COMPARING TO DJANGO QUERYSETS

Djaq can be used in theory as a total replacement for Django QuerySets

Djaq has some important advantages over QuerySets. You will probably find Djaq queries easier to write and understand.

Djaq queries are easier to understand because they don't make you jump around mentally parsing the difference between _ and __ and there are far fewer "special cases" where you need to use different classes and functions to overcome syntactical constraints of QuerySets.

You can send Djaq queries over the wire for a remote api with minimal effort, like via a REST call, and receive JSON results. That's not possible with QuerySets.

Djaq is explicit about performance semantics. In contrast you need to have knowledge of and use QuerySets carefully to avoid performance pitfalls.

This section is intended to highlight differences for users with good familiarity with the `QuerySet` class for the purpose of comparing `DjaqQuery` and `QuerySet`.

Django provides significant options for adjusting query generation to fit different use cases, `only()`, `select_related()`, `prefetch_related()` are all useful for different cases. Here's a point-by-point comparison with Djaq:

- `only()`: Djaq always works in "only" mode. Only explicitly requested fields are returned with the exception of providing no output fields in which case Djaq produces all fields but with no foreign key relations.
- `select_related()`: The output field expression list only returns those columns explicitly defined. This feature makes loading of related fields non-lazy. In contrast, queries are always non-lazy in Djaq.
- `prefetch_related()`: When you have a m2m field as a column expression, the model hosting that field is repeated in results as many times as necessary. Another way is to use a separate query for the m2m related records. In any case, `prefetch_related()` is not relevant in Djaq.
- `F` expressions: These are QuerySet workarounds for not being able to write expressions in the query for things like column value arithmetic and other expressions you want to have the db calculate. Djaq lets you write these directly and naturally as part of its syntax.
- To aggregate with Querysets, you use `aggregate()`, or `annotate()` whereas Djaq aggregates results whenever an aggregate function appears in a column expression, just like you would expect it to.
- Model instances with QuerySets exactly represent the corresponding Django model. Djaq's usual return formats, like `dicts()`, `tuples()`, etc. are more akin to `QuerySet.values()` and `QuerySet.value_list()`.
- Slicing: QuerySets can bet sliced: `qs[100:150]` whereas you use `limit()`, `offset()` with Djaq: `dq.offset(100).limit(50)`
- Caching: QuerySets will cache results in a rather sophisticated manner to support slicing (above). With Djaq, you need to rerun the query each time unless you are caching results yourself. Djaq eschews caching as part of the query evaluation to encourage separation of concerns and prevent unintended performance results.

Filter expressions in Djaq have a single expression paradigm, unlike QuerySets. When you filter a QuerySet, because you are assigning values to a series of parameters, the only way to construct the final SQL WHERE is to logically conjoin the boolean assertions.

```
Book.objects.filter(name_startswith="Bar", pubdate__year__gt='2020')
```

means `name ILIKE 'Bar%' AND date_part("year", pubdate) > 2020`. Whereas Djaq is explicit:

```
DQ("Book").where("ilike(name, 'Bar*') and pubdate.year > 2020")
```

If you want to change your query to `OR` with querysets, you have to change how you construct the filter:

```
from django.db.models import Q
Book.objects.filter(Q(name_startswith="Bar") | Q(pubdate__year__gt=2020))
```

with Djaq, you just do the obvious, change the operator:

```
DQ("Book").where("ilike(name, 'Bar%') or pubdate.year > 2020")
```

Both QuerySets and DjaqQuerys let you add conditions incrementally:

```
DQ("Book").where("regex(name, 'B.*')").where("pubdate.year > 2020")

Book.objects.filter(name__startswith="B").filter(pubdate__year__gt='2020')
```

The presumption is to conjoin the two conditions with "AND" in both cases.

Let's look at some more query comparisons:

Get the average price of books:

```
DQ("Book","avg(price)").value()
```

compared to QuerySet:

```
Book.objects.aggregate(Avg('price'))
```

Get the difference from the average off the maximum price for each publisher:

```
DQ("Book", "publisher.name, max(price) - avg(price) as price_diff")
```

compared to QuerySet:

```
from django.db.models import Avg, Max
Book.objects.values("publisher__name").annotate(price_diff=Max('price') - Avg('price'))
```

Count books per publisher:

```
DQ("Publisher", "name as publisher, count(book) as num_books")
```

compared to QuerySet:

```
Publisher.objects.annotate(num_books=Count("book"))
```

Count books with ratings up to and over a number:

```
DQ("Book", """publisher.name,
    sumif(rating <= 3, rating, 0) as below_3,
    sumif(rating > 3, rating, 0) as above_3
    """).go()
```

compared to QuerySet:

```
from django.db.models import Count, Q
below_3 = Count('book', filter=Q(book__rating__lte=3))
above_3 = Count('book', filter=Q(book__rating__gt=3))
Publisher.objects.annotate(below_3=below_3).annotate(above_3=above_3)
```

Get average, maximum, minimum price of books:

```
DQ("Book", "avg(price), max(price), min(price)")
```

compared to QuerySet:

```
Book.objects.aggregate(Avg('price'), Max('price'), Min('price'))
```

Note that by default, you iterate using a generator. You cannot slice a generator. Use `limit()` and `offset()` to page results

Simple counts:

`DjaqQuery.value()`: when you know the result is a single row with a single value, you can immediately access it without further iterations:

```
DQ("Book", "count(id)").value()
```

will return a single integer value representing the count of books.

CHAPTER

NINETEEN

PARAMETERS AND VALIDATOR

We call the Django connection cursor approximately like this:

```
from django.db import connections
cursor = connections['default']
cursor.execute(sql, context_dict)
```

When we execute the resulting SQL query, named parameters are used. You *must* name your parameters. Positional parameters are not passed:

```
oldest = '2000-01-01'
DQ("Book", "id").where("pub_date >= '{oldest}").context({"oldest": oldest}).tuples()
```

Notice that any parameterised value must be represented in the query expression in curly braces. Note as well, this is not an f-string!

```
{myparam}
```

Therefore, when you add subqueries, their parameters have to be supplied at the same time.

Note what is happening here:

```
name_search = 'Bar.*'
DQ("Book", "id").where("regex(b.name, {name_search}").context(locals()).tuples()
```

To get all books starting with 'Bar'. Or:

```
DQ("Book", "name").where("like(upper(name), upper({name_search})").context(request.POST)
```

Provided that `request.POST` has a `name_search` key/value.

You can provide a validation class that will return context variables. The default class used is called `ContextValidator()`. You can override this to provide a validator that raises exceptions if data is not valid or mutates the context data, like coercing types from `str` to `int`:

```
class MyContextValidator(ContextValidator):
    def get(self, key, value):
        if key == 'order_no':
            return int(value)
        return value

    def context(self):
        if not 'order_no' in self.data:
```

(continues on next page)

(continued from previous page)

```
            raise Exception("Need order no")
        self.super().context()
```

Then add the validator:

```
order_no = "12345"
DQ("Orders", "order_no, customer").where("order_no == {order_no}")
    .validator(MyContextValidator)
    .context(locals())
    .tuples()
```

You can set your own validator class in Django settings:

```
DJAQ_VALIDATOR = MyValidator
```

The `request` parameter of the API view is added to the context and will be available to the validator as `request`.

CHAPTER

TWENTY

QUERY UI

You can optionally install a query user interface to try out queries on your own data set:

- After installing djaq, add `djaq.djaq_ui` to INSTALLED_APPS
- Add

```
path("dquery/", include("djaq.djaq_ui.urls")),
```

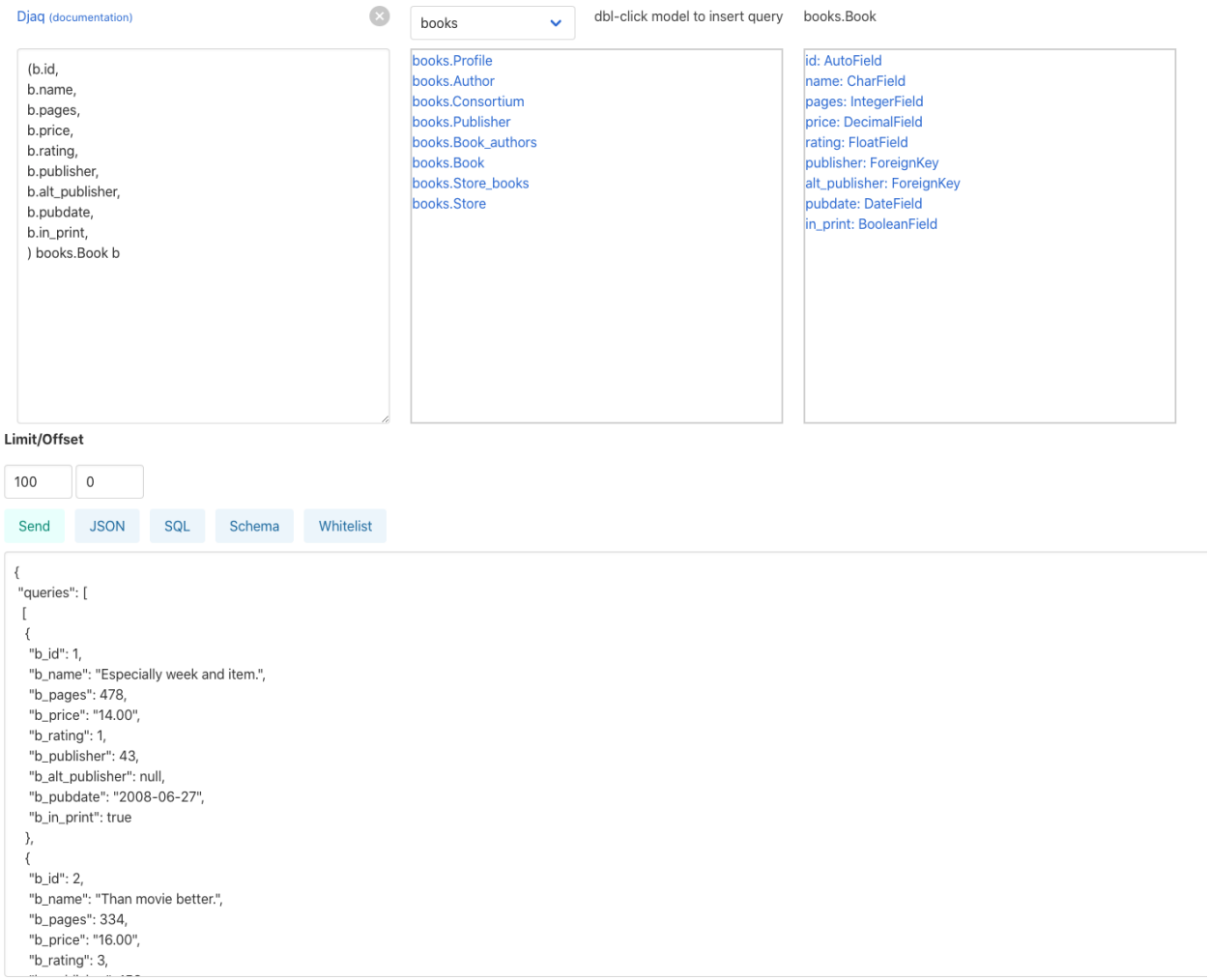
to `urlpatterns` in the site's `\ urls.py\`

Navigate to */dquery/* in your app and you should be able to try out queries.

- Send: call the API with the query
- JSON: show the json that will be sent as the request data
- SQL: show how the request will be sent to the database as sql
- Schema: render the schema that describe the available fields
- Whitelist: show the active whitelist. You can use this to generate a whilelist and edit it as required.

There is a dropdown control, `apps`. Select the Django app. Models for the selected app are listed below. If you click once on a model, the result field will show the schema for that model. If you double-click the model, it generates a query for you for all fields in that model. Once you do that, just press "Send" to see the results.

If the query pane has the focus, you can press shift-return to send the query request to the server.


Djaq (documentation)
books
dbl-click model to insert query
books.Book
(b.id,
b.name,
b.pages,
b.price,
b.rating,
b.publisher,
b.alt_publisher,
b.pubdate,
b.in_print,
) books.Book b
books.Profile
books.Author
books.Consortium
books.Publisher
books.Book_authors
books.Book
books.Store_books
books.Store
id: AutoField
name: CharField
pages: IntegerField
price: DecimalField
rating: FloatField
publisher: ForeignKey
alt_publisher: ForeignKey
pubdate: DateField
in_print: BooleanField
Limit/Offset
100
0
Send
JSON
SQL
Schema
Whitelist
{
"queries": [
[
{
"b_id": 1,
"b_name": "Especially week and item.",
"b_pages": 478,
"b_price": "14.00",
"b_rating": 1,
"b_publisher": 43,
"b_alt_publisher": null,
"b_pubdate": "2008-06-27",
"b_in_print": true
},
{
"b_id": 2,
"b_name": "Than movie better.",
"b_pages": 334,
"b_price": "16.00",
"b_rating": 3,

CHAPTER

TWENTYONE

REMOTE API

If you install the djaq_api app in INSTALLED_APPS, you have a remote api installed.

You POST requests to the endpoint, which by default is `/djaq/api/request/`

All requests have this overall structure:

```
{
    "queries": [],
    "updates": [],
    "deletes": [],
    "creates": []
}
```

Any section can be left away.

21.1 Remote Queries

Provide at least `model` as an argument:

```
{
    "queries": [
        {
            "model": "Book",
            "output": "id, name, price",
            "where": "id==3",
            "limit": 1,
        }
    ]
}
```

This will provide id, name, price for a Book with id of 3.

21.2 Remote Updates

Provide `model` and `pk` as arguments and then a set of field name/values:

```
{
    "updates": [
        {
            "model": "books.Book",
            "pk": 3,
            "fields": {
                "price": 3.99
            }
        }
    ]
}
```

21.3 Remote Creates

```
{
    "creates": [
        {
            "model": "books.Book",
            "fields": {
                "name": "My great american novel",
                "publisher_id": 10,
                "price": 3.99
            }
        }
    ]
}
```

21.4 Remote Deletes

Specify the model and primary key:

```
{
    "updates": [
        {
            "model": "books.Book",
            "pk": 3,
        }
    ]
}
```

21.5 Custom API

You can write your own custom API endpoint. Here is what a view function for your data layer might look like with Djaq:

```
@login_required
def djaq_view(request):
    data = json.loads(request.body.decode("utf-8"))
    model_name = data.get("model")
    output_expressions = data.get("where")
    order_by = data.get("order_by")
    offset = int(data.get("offset", 0) or 0)
    limit = int(data.get("limit", 0) or 0)
    context = data.get("context", dict() or dict())
    return JsonResponse({
        "result": list(
            DQ(model_name, output_expressions)
            .where(where)
            .order_by(order_by)
            .context(context)
            .limit(limit)
            .offset(offset)
            .dicts()
        )
        }
    )
```

You can now query any models in your entire Django deployment remotely, provided the authentication underlying the *login_required* is satisfied. This is a good solution if your endpoint is only available to trusted clients who hold a valid authentication token or to clients without authentication who are in your own network and over which you have complete control. It is a bad solution on its own for any public access since it exposes Django framework models, like users, permissions, etc.

Most likely you want to control access in two ways:

- Allow access to only some apps/models
- Allow access to only some rows in each table and possibly only some fields.

For controlling access to models, use the whitelist parameter in constructing the DjangoQuery:

```
DQ(model_name, column_expressions, whitelist={"books": ["Book", "Publisher",],}) \
    .context(context) \
    .limit(limit) \
    .offset(offset) \
    .dicts()
```

This restricts access to only the *book* app models, Book and Publish.

You probably need a couple more things if you want to expose this to a browser. But this gives an idea of what you can do. The caller now has access to any authorised model resource. Serialisation is all taken care of. Djaq comes already with a view similar to the above. You can just start calling and retrieving any data you wish. It's an instant API to your application provided you trust the client or have sufficient access control in place.

CHAPTER

TWENTYTWO

LIMITATIONS

The main limitation of Djaq at this time is that it only supports Postgresql.

CHAPTER

TWENTYTHREE

PERFORMANCE

You will probably experience Djaq calls as blazing fast compared to other remote frameworks. This is just because not much happens inbetween. Once the query is parsed, it is about as fast as you will ever get unless you do something fancy in a validator. The simplest possible serialization is used by default.

Once the query is parsed, it is about the same overhead as calling this:

```
conn = connections['default']
cursor = conn.cursor()
self.cursor = self.connection.cursor()
self.cursor.execute(sql)
```

Parsing is pretty fast and might be a negligible factor if you are parsing during a remote call as part of a view function.

But if you want to iterate over, say, a dictionary of variables locally, you'll want to parse once:

```
dq = DQ("Book", "name").where("ilike(name, {namestart})")
dq.parse()
for vars in var_list:
    results = list(dq.context(vars).tuples())
    '<do something with results>'
```

Note that each call of *context()* causes the cursor to execute again when *tuples()* is iterated.

CHAPTER

TWENTYFOUR

SAMPLE BOOKSHOP PROJECT

If you want to use Djaq right away in your own test project and you feel confident, crack on. In that case skip the following instructions for using the sample Bookshop project. Or, if you want to try the sample project, clone the django repo:

```
git clone git@github.com:paul-wolf/djaq.git
cd djaq/bookshop
```

If you clone the repo and use the sample project, you don't need to include Djaq as a requirement because it's included as a module by a softlink. Create the virtualenv:

The module itself does not install Django and there are no further requirements. Make sure you are in the `bookshop` directory:

```
python -m venv .venv && source .venv/bin/activate && pip install --upgrade pip && pip␣
↪install -r requirements.txt
```

We need a log directory:

```
mkdir log
```

Create a user. Let's assume a super user:

```
createsuperuser --username yourname
```

Now make sure there is a Postgresql instance running. The settings are like this:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'bookshop',
    },
```

So, it assumes peer authentication. Change to suit your needs. Now you can migrate. Make sure the virtualenv is activated!

```
./manage.py migrate
```

We provide a script to create some sample data:

```
./manage.py build_data --book-count 2000
```

This creates 2000 books and associated data.

There's a management command to run queries from the command line:

```
./manage.py djaq Book
```

Output of the command should look like this:

```
 ./manage.py djaq Book
[
   {
      "id": 2,
      "name": "Station many chair pressure.",
      "pages": 414,
      "price": "12.00",
      "rating": 2.0,
      "publisher": 99,
      "alt_publisher": null,
      "pubdate": "2016-11-30",
      "in_print": true
   },
   {
      "id": 3,
      "name": "Able sense quickly.",
      "pages": 408,
      "price": "29.00",
      "rating": 3.0,
      "publisher": 40,
      "alt_publisher": null,
      "pubdate": "2001-07-27",
      "in_print": true
   },
   ...
```

The best approach now would be to trial various queries using the Djaq UI as explained above.

Finally, checkout the settings for the bookshop. You will notice that many admin models are not accessible. In a real application we'd want to prevent access to user data and other data on perhaps a finer grained level.

Run the server:

```
./manage.py runserver
```

Now the query UI should be available here:

http://127.0.0.1:8000/dquery/

CHAPTER

TWENTYFIVE

INDICES AND TABLES

- genindex
- modindex
- search