

Performing raw SQL queries

When the [model query APIs](#) don't go far enough, you can fall back to writing raw SQL. Django gives you two ways of performing raw SQL queries: you can use `Manager.raw()` to [perform raw queries and return model instances](#), or you can avoid the model layer entirely and [execute custom SQL directly](#).

Performing raw queries

The `raw()` manager method can be used to perform raw SQL queries that return model instances:

`Manager.raw(raw_query, params=None, translations=None)`

This method takes a raw SQL query, executes it, and returns a `django.db.models.query.RawQuerySet` instance. This `RawQuerySet` instance can be iterated over just like a normal `QuerySet` to provide object instances.

This is best illustrated with an example. Suppose you've got the following model:

```
class Person(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)
    birth_date = models.DateField(...)
```

You could then execute custom SQL like so:

```
>>> for p in Person.objects.raw('SELECT * FROM myapp_person'):
...     print(p)
John Smith
Jane Jones
```

Of course, this example isn't very exciting – it's exactly the same as running `Person.objects.all()`. However, `raw()` has a bunch of other options that make it very powerful.

Model table names

Where'd the name of the `Person` table come from in that example?

By default, Django figures out a database table name by joining the model's "app label" – the name you used in `manage.py startapp` – to the model's class name, with an underscore between them. In the example we've assumed that the `Person` model lives in an app named `myapp`, so its table would be `myapp_person`.

For more details check out the documentation for the `db_table` option, which also lets you manually set the database table name.

Warning

No checking is done on the SQL statement that is passed in to `.raw()`. Django expects that the statement will return a set of rows from the database, but does nothing to enforce that. If the query does not return rows, a (possibly cryptic) error will result.

Mapping query fields to model fields


`raw()` automatically maps fields in the query to fields on the model.

The order of fields in your query doesn't matter. In other words, both of the following queries work identically:

```
>>> Person.objects.raw('SELECT id, first_name, last_name, birth_date FROM myap
```

```
...
>>> Person.objects.raw('SELECT last_name, birth_date, first_name, id FROM myap
...

```



Matching is done by name. This means that you can use SQL's AS clauses to map fields in the query to model fields. So if you had some other table that had `Person` data in it, you could easily map it into `Person` instances:

```
>>> Person.objects.raw('''SELECT first AS first_name,
...                        last AS last_name,
...                        bd AS birth_date,
...                        pk as id,
...                        FROM some_other_table''')
```

As long as the names match, the model instances will be created correctly.

Alternatively, you can map fields in the query to model fields using the `translations` argument to `raw()`. This is a dictionary mapping names of fields in the query to names of fields on the model. For example, the above query could also be written:

```
>>> name_map = {'first': 'first_name', 'last': 'last_name', 'bd': 'birth_date'
>>> Person.objects.raw('SELECT * FROM some_other_table', translations=name_map
```



Index lookups

`raw()` supports indexing, so if you need only the first result you can write:

```
>>> first_person = Person.objects.raw('SELECT * from myapp_person')[0]
```

However, the indexing and slicing are not performed at the database level. If you have a big amount of `Person` objects in your database, it is more efficient to limit the query at the SQL level:

```
>>> first_person = Person.objects.raw('SELECT * from myapp_person LIMIT 1')[0]
```



Deferring model fields

Fields may also be left out:

```
>>> people = Person.objects.raw('SELECT id, first_name FROM myapp_person')
```

The `Person` objects returned by this query will be deferred model instances (see `defer()`). This means that the fields that are omitted from the query will be loaded on demand. For example:

```
>>> for p in Person.objects.raw('SELECT id, first_name FROM myapp_person'):
...     print(p.first_name, # This will be retrieved by the original query
...           p.last_name) # This will be retrieved on demand
...
John Smith
Jane Jones
```

From outward appearances, this looks like the query has retrieved both the first name and last name. However, this example actually issued 3 queries. Only the first names were retrieved by the `raw()` query –

This document is for Django's development version, which can be significantly different from previous releases. For older releases, use the version selector floating in the bottom right corner of this page.

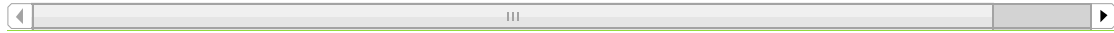
is the primary key to
`lidQuery` exception will

be raised if you forget to include the primary key.

Adding annotations

You can also execute queries containing fields that aren't defined on the model. For example, we could use PostgreSQL's `age()` function to get a list of people with their ages calculated by the database:

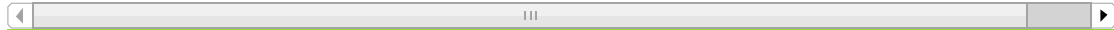
```
>>> people = Person.objects.raw('SELECT *, age(birth_date) AS age FROM myapp_p
>>> for p in people:
...     print("%s is %s." % (p.first_name, p.age))
John is 37.
Jane is 42.
...
```



Passing parameters into `raw()`

If you need to perform parameterized queries, you can use the `params` argument to `raw()`:

```
>>> lname = 'Doe'
>>> Person.objects.raw('SELECT * FROM myapp_person WHERE last_name = %s', [lname])
```



`params` is a list or dictionary of parameters. You'll use `%s` placeholders in the query string for a list, or `%(key)s` placeholders for a dictionary (where `key` is replaced by a dictionary key, of course), regardless of your database engine. Such placeholders will be replaced with parameters from the `params` argument.

Note

Dictionary params not supported with SQLite

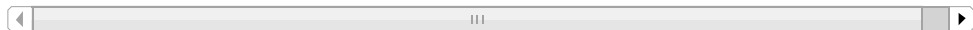
Dictionary params are not supported with the SQLite backend; with this backend, you must pass parameters as a list.

Warning

Do not use string formatting on raw queries!

It's tempting to write the above query as:

```
>>> query = 'SELECT * FROM myapp_person WHERE last_name = %s' % lname
>>> Person.objects.raw(query)
```



Don't.

Using the `params` argument completely protects you from [SQL injection attacks](#), a common exploit where attackers inject arbitrary SQL into your database. If you use string interpolation, sooner or later you'll fall victim to SQL injection. As long as you remember to always use the `params` argument you'll be protected.

Changed in Django 1.6:

In Django 1.5 and earlier, you could pass parameters as dictionaries when using PostgreSQL or MySQL, although this wasn't documented. Now you can also do this when using Oracle, and it is officially supported.

Executing custom SQL directly

Sometimes even `Manager.raw()` isn't quite enough: you might need to perform queries that don't map cleanly to models, or directly execute `UPDATE`, `INSERT`, or `DELETE` queries.

In these cases, you can always access the database directly, routing around the model layer entirely.

The object `django.db.connection` represents the default database connection. To use the database connection, call `connection.cursor()` to get a cursor object. Then, call `cursor.execute(sql, [params])` to execute the SQL and `cursor.fetchone()` or `cursor.fetchall()` to return the resulting rows.

For example:

```
from django.db import connection

def my_custom_sql(self):
    cursor = connection.cursor()

    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])

    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
    row = cursor.fetchone()

    return row
```

Changed in Django 1.6:

In Django 1.5 and earlier, after performing a data changing operation, you had to call `transaction.commit_unless_managed()` to ensure your changes were committed to the database. Since Django now defaults to database-level autocommit, this isn't necessary any longer.

Note that if you want to include literal percent signs in the query, you have to double them in the case you are passing parameters:

```
cursor.execute("SELECT foo FROM bar WHERE baz = '30%'")
cursor.execute("SELECT foo FROM bar WHERE baz = '30%%' and id = %s", [self.id])
```

If you are using [more than one database](#), you can use `django.db.connections` to obtain the connection (and cursor) for a specific database. `django.db.connections` is a dictionary-like object that allows you to retrieve a specific connection using its alias:

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
# Your code here...
```

By default, the Python DB API will return results without their field names, which means you end up with a `list` of values, rather than a `dict`. At a small performance cost, you can return results as a `dict` by using something like this:

```
def dictfetchall(cursor):
    "Returns all rows from a cursor as a dict"
    desc = cursor.description
    return [
        dict(zip([col[0] for col in desc], row))
        for row in cursor.fetchall()
    ]
```

Here is an example of the difference between the two:

```
>>> cursor.execute("SELECT id, parent_id from test LIMIT 2");
>>> cursor.fetchall()
((54360982L, None), (54360880L, None))

>>> cursor.execute("SELECT id, parent_id from test LIMIT 2");
>>> dictfetchall(cursor)
[{'parent_id': None, 'id': 54360982L}, {'parent_id': None, 'id': 54360880L}]
```

Connections and cursors

`connection` and `cursor` mostly implement the standard Python DB-API described in [PEP 249](#) — except when it comes to [transaction handling](#).

If you're not familiar with the Python DB-API, note that the SQL statement in `cursor.execute()` uses placeholders, `"%s"`, rather than adding parameters directly within the SQL. If you use this technique, the underlying database library will automatically escape your parameters as necessary.

Also note that Django expects the `"%s"` placeholder, *not* the `"?"` placeholder, which is used by the SQLite Python bindings. This is for the sake of consistency and sanity.

[« Managers](#)

[Database transactions »](#)

© 2005-2013 [Django Software Foundation](#) unless otherwise noted. Django is a registered trademark of the Django Software Foundation. [Linux Web hosting](#) graciously provided by Media Temple.