Custom Database Backends

•••

Michael Manfre
Django Under The Hood 2016

About Me (@manfre)

- 2008
  - Hired at Semiconductor Research Corporation (SRC)
  - Used Django for the first time (it was on Windows)
  - Became maintainer of Django-mssql
- 2014
  - Django-mssql dropped SQL Server 2008 support. Me == Happy
- 2015
  - Joined Django Team
  - Database backend hack-days at Microsoft

Keep digging lower until your brain hurts

Does this make sense?

```python
Company.objects.annotate(
    salaries=F('ceo__salary')
).values('num_employees', 'salaries').aggregate(
    result=Sum(
        F('salaries') + F('num_employees'),
        output_field=models.IntegerField()
    )
)
```

```sql
SELECT SUM(("salaries" + "__col1"))
FROM (
    SELECT "app_company"."num_employees" AS Col1,
           "app_employee"."salary" AS "salaries",
           "app_company"."num_employees" AS "__col1"
    FROM "app_company" INNER JOIN  "app_employee" ON
         ("app_company"."ceo_id" = "app_employee"."id")
) subquery
```

Django IN Depth

James Bennett - PyCon 2015

https://www.youtube.com/watch?v=tkwZ1jG3XgA

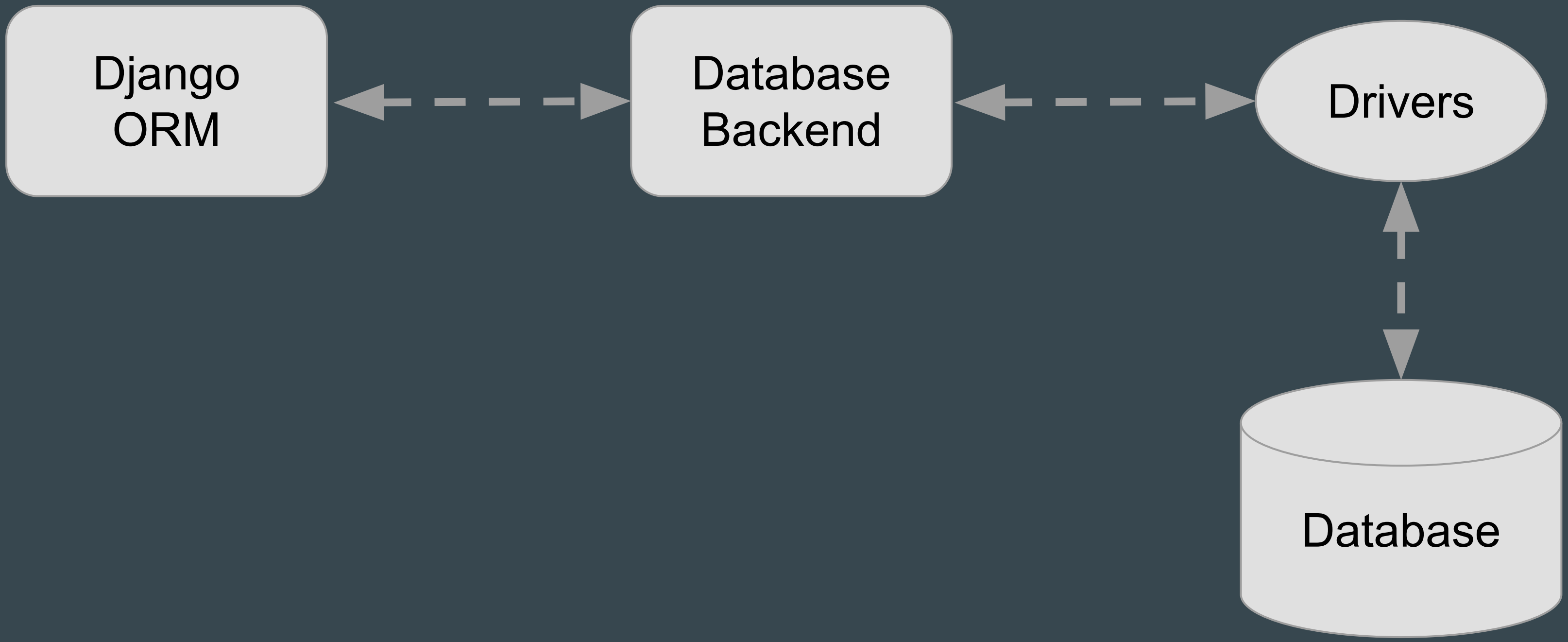
Down the rabbit hole…

- Model
- Manager
- QuerySet
- Query
- Expression
- SQLCompiler
- Database backend

Down the rabbit hole...

- Model
- Manager
- QuerySet
- Query
- Expression
- SQLCompiler
- Database backend

What does the database backend do?



PEP 249 - DB-API 2.0 Specification

- Connections
  - `close, commit, rollback, cursor`
- Cursors
  - `callproc, close, execute, executemany, fetchone, fetchmany, fetchall, nextset`
- Exceptions
  - `DatabaseError, IntegrityError, OperationalError,` …
- paramstyle
  - `qmark - ...WHERE name=?`
  - `format - ...WHERE name=%s`

Not All Databases Are Created Equal

- Which SQL dialect?
  - SQL-89, SQL-92, SQL:2008, ...
- Slicing syntax (LIMIT / OFFSET)
- Transaction support
- Supported datatypes
- Rules for subqueries
- Different aggregates and functions
- NULL
- Dates and times
  - Microseconds?
  - Timezones?
- quote_name - `[MyTable]` vs. "`MyTable`" vs. `` `MyTable` ``

Database Settings

```
DATABASES = {
    'default': {
        'ENGINE': 'sqlserver_ado',
        'HOST': r'localhost\sqlexpress',
        'NAME': 'djangoproject',
        'OPTIONS': {
            'provider': 'sqlncli11',
            'cast_avg_to_float': True,
        }
    }
}
```

- django.db.utils.ConnectionHandler
- Database backends must contain base.py

Minimal Database Backend

```python
from django.db.backends.postgresql_psycopg2 import base

class DatabaseWrapper(base.DatabaseWrapper):
    def get_new_connection(self, conn_params):
        conn = super(DatabaseWrapper,
                     self).get_new_connection(conn_params)
        conn.set_session(readonly=True)
        return conn
```

- django-postgres-readonly
  - https://github.com/opbeat/django-postgres-readonly
- django-sqlserver
  - https://github.com/denisenkom/django-sqlserver

Database API Classes

- DatabaseWrapper
- DatabaseFeatures
- DatabaseSchemaEditor
- DatabaseCreation
- DatabaseOperations
- DatabaseIntrospection
- DatabaseClient
- DatabaseValidation

DatabaseWrapper

- Manages PEP 249 connection
- Create cursors
- Enable/disable constraints
- Transaction handling
  - Commit, rollback, savepoints, auto commit, etc.
- `__init__()` is provided settings as a dict, not as settings module

Vendor

```
class DatabaseWrapper(BaseDatabaseWrapper):
    vendor = 'microsoft'
```

- String identifying the type of database
  - Built-in backends: `sqlite`, `postgresql`, `mysql`, `oracle`
  - Microsoft SQL Server backends: `microsoft`
- `as_{vendor}` override for `as_sql`
- Model Meta option `required_db_vendor`

Defining Lookups

```
class DatabaseWrapper(BaseDatabaseWrapper):
  operators = {
    "exact": "= %s",
    "iexact": "LIKE %s ESCAPE '\\'",
    "gte": ">= %s",
    "startswith": "LIKE %s ESCAPE '\\'",
  ...}
```

```
  pattern_esc = r"REPLACE(REPLACE(REPLACE({}, '\', '\\'),
'%%', '\%%'), '_', '\_')"

  pattern_ops = {
    'contains': r"LIKE CONCAT('%%', {}, '%%') ESCAPE '\'",
    'startswith': r"LIKE CONCAT({}, '%%') ESCAPE '\'",
  ...}
```

Mapping Fields To Column Types

```
class DatabaseWrapper(BaseDatabaseWrapper):
    data_types = {
        'AutoField': 'int',
        'BigAutoField': 'bigint IDENTITY (1, 1)',
        'CharField': 'nvarchar(%(max_length)s)',
        ...
    }
    data_types_suffix = {
        'AutoField': 'IDENTITY (1, 1)',
    }

    data_type_check_constraints = {
        'PositiveIntegerField': '%(qn_column)s >= 0',
        'PositiveSmallIntegerField': '%(qn_column)s >= 0',
    }
```

Methods A Backend Needs To Implement

- Connections and cursors
  - `get_connection_params, get_new_connection, init_connection_state, create_cursor, is_usable`
- Transaction Management
  - `_set_autocommit, _start_transaction_under_autocommit`
- Foreign Key Constraints
  - `disable_constraint_checking, enable_constraint_checking, check_constraints`

CursorWrapper

- `django.db.backends.utils`
    - Converter functions: Python ←→ database (string)
- Wraps PEP 249 style Cursor
    - `callproc, execute, executemany, fetchone, fetchmany, fetchall, nextset`
- Instantiated by `DatabaseWrapper.make_cursor()`
- Converts backend exceptions using `DatabaseErrorWrapper`

CursorDebugWrapper

- `CursorDebugWrapper` adds timing metrics and logging to `DatabaseWrapper.queries_log`
    - Extends `CursorWrapper`
- Instantiated by `DatabaseWrapper.make_debug_cursor()`
- `DatabaseWrapper.force_debug_cursor == True` or `settings.DEBUG`

DatabaseFeatures

- Currently 64 features
- Backend identifies its supported functionality and behaviors
  - Can slice subqueries?
  - Provides native datatypes for real, UUID, etc.
- Django determines some features programmatically
  - `supports_transactions`, `supports_stddev`, etc.
- Many features are only used by the test suite
  - `test_db_allows_multiple_connections, can_introspect_*`

DatabaseSchemaEditor

- Used by migrations
- Generates the Data Definition Language (DDL) statements
  - `ALTER` ..., `DROP` ..., etc
- Migrations Under The Hood - Andrew Godwin - DUTH 2014
  - https://www.youtube.com/watch?v=-4jhPRfCRSM

DatabaseSchemaEditor - SQL Templates

```
sql_create_table = "CREATE TABLE %(table)s (%(definition)s)"
sql_rename_table = "ALTER TABLE %(old_table)s RENAME TO "
                                              "%(new_table)s"
sql_retablespace_table = "ALTER TABLE %(table)s SET TABLESPACE "
                                          "%(new_tablespace)s"
sql_delete_table = "DROP TABLE %(table)s CASCADE"

sql_create_column = "ALTER TABLE %(table)s ADD COLUMN "
                                   "%(column)s %(definition)s"
sql_alter_column = "ALTER TABLE %(table)s %(changes)s"
sql_alter_column_type = "ALTER COLUMN %(column)s TYPE %(type)s"
sql_alter_column_null = "ALTER COLUMN %(column)s DROP NOT NULL"

...
```

Altering A Field Is Complex

- BaseDatabaseSchemaEditor is almost 1,000 lines of code
    - Altering a field is about 300 lines
- Oracle - Catch specific DatabaseError thrown by `alter_field` and apply workaround.
    - Create nullable column, copy data, drop old column, rename column
    - Easier to maintain, but can be slow for large tables
- MSSQL - Reimplement `_alter_field` with fixes
    - More difficult to maintain

DatabaseSchemaEditor - quote_value

```python
def quote_value(self, value):
    # This is not safe against injection from user code
    if isinstance(value, DATE_AND_TIME_TYPES):
        return "'%s'" % value
    elif isinstance(value, six.string_types):
        return "'%s'" % value.replace("'", "''")
    elif isinstance(value, six.buffer_types):
        return "0x%s" % force_text(binascii.hexlify(value))
    elif isinstance(value, bool):
        return "1" if value else "0"
    else:
        return str(value)
```

DatabaseCreation

- Creates and destroys test databases
- “testserver” management command
- django.test.runner.DiscoverRunner

DatabaseCreation

```
class BaseDatabaseCreation(object):
    def create_test_db(...):
    def _create_test_db(...):     ←

    def clone_test_db(...):
    def _clone_test_db(...):      ←

    def destroy_test_db(...):
    def _destroy_test_db(...):    ←

    def sql_table_creation_suffix(...):

    def _get_test_db_name(...):
```

DatabaseIntrospection

- Used by inspectdb management command
- Ability to look at a database and find its various schema objects.
    - Table, column, index, etc.
- Reverse mapping for database types to Model Fields
    - Understands internal type representations for database driver

DatabaseClient

```python
class BaseDatabaseClient(object):
    """
    This class encapsulates all backend-specific methods
    for opening a client shell.
    """
    # This should be string representing the name of the executable
    # (e.g., "psql"). Subclasses must override this.
    executable_name = None

    def runshell(self):
        raise NotImplementedError(...)
```

DatabaseValidation

- Checks framework
  - Tags.database, Tags.models
- Model/schema validation
  - MySQL 255 char limit if unique index
- Ensure safe database settings
  - MySQL Strict Mode
- Check for missing add-ons
  - Regex CLR DLL

DatabaseValidation

```python
class BaseDatabaseValidation(object):
    """
    This class encapsulates all backend-specific validation.
    """
    def __init__(self, connection):
        self.connection = connection

    def check(self, **kwargs):
        return []

    def check_field(self, field, **kwargs):
        return []
```

DatabaseOperations

- `compiler_module`
- `Integer_field_ranges`
- Date and time helpers
    - Extraction, casting, truncation
- DB converters
- Transform values for database driver

as_sql

Query

- Query contains multiple lists of objects that as a whole represent the database operation.
- `as_sql` is called on everything to generate the SQL statement
- Code is massive and complex
- Sprawls across many files and thousands of lines of code
- Query maintains state of the merged queries.

```
Entry.objects.filter(...).exclude(...).filter(...)[2:5]
```

SQL Compilers

- SQLCompiler
  - `SELECT …`
- SQLInsertCompiler
  - `INSERT INTO …`
- SQLDeleteCompiler
  - `DELETE FROM …`
- SQLUpdateCompiler
  - `UPDATE … SET …`
- SQLAggregateCompiler
  - `SELECT … subquery`

Only Modify What You Need

```
from django.db.models.sql import compiler as c

class SQLCompiler(c.SQLCompiler):

   # customizations


class SQLInsertCompiler(c.SQLInsertCompiler, SQLCompiler):
    pass


class SQLDeleteCompiler(c.SQLDeleteCompiler, SQLCompiler):
    pass


class SQLUpdateCompiler(c.SQLUpdateCompiler, SQLCompiler):
    pass


class SQLAggregateCompiler(c.SQLAggregateCompiler, SQLCompiler):
    pass
```

SQLCompiler Customizations

- Subqueries are not the same for all databases
  - Mysql `as_subquery_condition`
- LIMIT / OFFSET syntax differences
- Return ID from insert
- Different syntax when inserting an IDENTITY value
- Fixing record count for updates

Dragon is sad because some databases think paging a query’s results should be difficult

Limiting QuerySets

```
Entry.objects.all()[:5]
```

```
Entry.objects.all()[1:5]
```

LIMIT / OFFSET - Postgresql, MySQL

Entry.objects.all()[:5]

```
SELECT ...
FROM blog_entry
LIMIT 5
```

Entry.objects.all()[1:5]

```
SELECT ...
FROM blog_entry
LIMIT 5 OFFSET 1
```

TOP / WHAT?!? - MSSQL 2008 (and earlier)

Entry.objects.all()[:5]

```
SELECT TOP 5 ...
FROM blog_entry
```

Entry.objects.all()[1:5]

```
SELECT _row_num, {outer}
FROM (SELECT ROW_NUMBER() OVER ( ORDER BY
{order}) as _row_num, {inner}) as QQQ
WHERE 1 < _row_num and _row_num <= 6
```

OFFSET / FETCH - MSSQL 2012

Entry.objects.all()[:5]

```
SELECT ...
FROM blog_entry
ORDER BY 1
OFFSET 0 ROWS
FETCH NEXT 5 ROWS ONLY
```

Entry.objects.all()[1:5]

```
SELECT ...
FROM blog_entry
ORDER BY 1
OFFSET 1 ROWS
FETCH NEXT 4 ROWS ONLY
```

Expressions

- All query expressions inherit from

  `django.db.models.expressions.BaseExpression`

  - Except for `F()`, which is a `Combinable`
- `BaseExpression.as_sql()` renders the SQL
- Some types of expressions provide the format string `template`
- Func based expressions provide `function` and `arg_joiner`
- “Customize your SQL” - Josh Smeaton
  - https://www.youtube.com/watch?v=9rEB6ra4aB8

Length

```
class Length(Transform):
    """Returns the number of characters in the expression"""
    function = 'LENGTH'
    lookup_name = 'length'

    def __init__(self, expression, **extra):
        output_field = extra.pop('output_field',
                                 fields.IntegerField())
        super(Length, self).__init__(
                                expression,
                                output_field=output_field,
                                **extra)

>>> Author.objects.filter(name__length__gt=7)
```

You Like To-may-toes And I Like To-mah-toes

```
@as_microsoft(Length)
def fix_length_name(self, compiler, connection):
    """T-SQL LEN()"""
    return self.as_sql(compiler, connection,
                       function='LEN')

@as_microsoft(Substr)
def three_substr_args(self, compiler, connection):
    """SUBSTRING() requires 3 args. Len is never implied"""
    if len(self.source_expressions) == 2:
        self.source_expressions.append(
            Value(2 ** 31 - 1))
    return self.as_sql(compiler, connection)
```

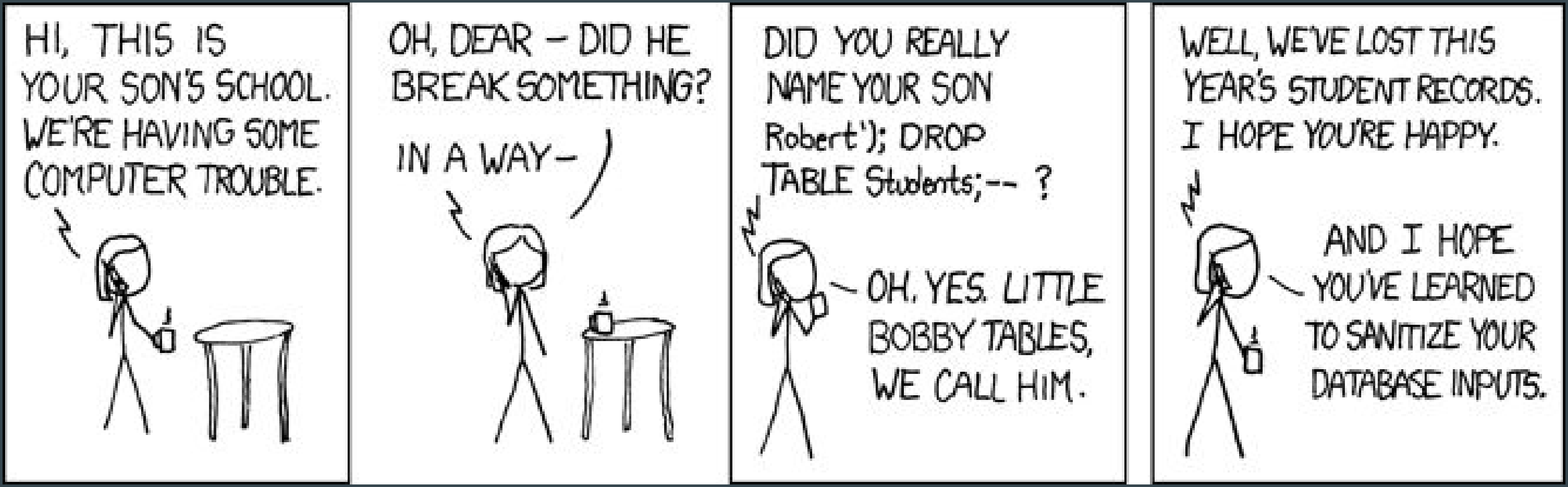
Fake It Till You Make It

```python
@as_microsoft(Greatest)
def emulate_greatest(self, compiler, connection):
    # SQL Server does not provide GREATEST function,
    # so we emulate it with a table value constructor
    # https://msdn.microsoft.com/en-us/library/dd776382.aspx
    template = '(SELECT MAX(value) FROM (VALUES '
               '(%(expressions)s)) AS _%(function)s(value))'
    return self.as_sql(compiler, connection,
                       arg_joiner='), (',
                       template=template)
```

as_vendor

```
def as_microsoft(expression):
    """
    Decorated function is added to the provided expression
    as the Microsoft vender specific as_sql override.
    """
    def dec(func):
        setattr(expression, 'as_microsoft', func)
        return func
    return dec
```

SQL Injection


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

XKCD #327

SQL Injection

- Never add user provided values into the SQL

```
# NEVER DO THIS!!!
cursor.execute('SELECT … name = %s' % name)
Person.objects.raw('SELECT … name = %s' % name)
```

- Values are provided separately with params

```
cursor.execute('SELECT … name = %s', params=[name])
Person.objects.raw('SELECT … name = %s', params=[name])
```

- Database backends craft lots of raw SQL

Backend Specific Testing

Watch Your Step

- Database driver changes
- Python client package changes
- Database software changes
- New versions of Django

Trust, But Verify

Hard check Django version

```
from django import VERSION

if VERSION[:3] < (1,10,0) or VERSION[:2] >= (1,11):
    raise ImproperlyConfigured(...)
```

Soft check database version

```
class DatabaseWrapper(BaseDatabaseWrapper):
    def init_connection_state(self):
        sql_version = self.__get_dbms_version()
        if sql_version < VERSION_SQL2012:
            warnings.warn("Database version is not "
                          "officially supported",
                          DeprecationWarning)
```

Django’s Test Suite

- `python tests/runtests.py --settings=test_mssql`
- Shared code coverage
- Test Driven Development for custom database backends
  - Feature and bug fix PRs require tests that database backends get to use
  - Avoids “working as implemented” tests
- Test failures can be expected
  - PR to fix for the future Django. Local branch for now
  - Monkey patch tests with `@expectedFailure`
  - Different expected value for `assertNumQueries`
- Still need backend specific test suite!

Conditionally Testing A Backend

- Vendor string
    - Avoid doing this whenever possible.

```
@skipUnless(connection.vendor == 'postgresql',
            "Test only for PostgreSQL")
```

- DatabaseFeatures
    - skipIfDBFeature
    - skipUnlessDBFeature
    - skipUnlessAnyDBFeature

Non-Relation Backends

Closing Thoughts

Q & A

- Django In Depth - James Bennett
  - https://www.youtube.com/watch?v=tkwZ1jG3XgA
- Migrations Under The Hood - Andrew Godwin
  - https://www.youtube.com/watch?v=-4jhPRfCRSM
- Customize Your SQL - Josh Smeaton
  - https://www.youtube.com/watch?v=9rEB6ra4aB8
- Contact Me

IRC: manfre
Twitter: @manfre

github.com/manfre
keybase.io/manfre