
Django MSSQL Documentation

Release dev

Django MSSQL authors

May 21, 2014

Provides an ADO based Django database backend for Microsoft SQL Server.

Welcome to Django-mssql's documentation!

Contents:

1.1 Quickstart

1.1.1 Installation

- Install `django-mssql` with your favorite Python package manager:

```
pip install django-mssql
```

- Add the `DATABASES` configuration.

```
DATABASES = {
    'default': {
        'NAME': 'my_database',
        'ENGINE': 'sqlserver_ado',
        'HOST': 'dbserver\\ss2008',
        'USER': '',
        'PASSWORD': '',
    }
}
```

Note: Although the project is named `django-mssql` the python module is named `sqlserver_ado`.

1.1.2 Getting the code

The project code is hosted on [Bitbucket](#)

```
hg clone https://bitbucket.org/Manfre/django-mssql/
```

1.1.3 Dependencies

Django

Django 1.2 and newer are supported by the current release. Support for Django 1.1 requires getting code from tag `legacy-1.1`.

Python

This backend requires Python 2.6 or newer. Python 3.x support will be investigated when Django supports it.

PyWin32

PyWin32 build 212 or newer is required.

1.2 Settings

1.2.1 DATABASES

Please see the [Django documentation on DATABASES settings](#) for a complete list of available settings. *Django-mssql* builds upon these settings.

This is an example of a typical configuration:

```
DATABASES = {
    'default': {
        'NAME': 'my_database',
        'ENGINE': 'sqlserver_ado',
        'HOST': 'dbserver\\ss2008',
        'USER': '',
        'PASSWORD': '',
    }
}
```

ENGINE

This value must be set to `sqlserver_ado`.

HOST

Default: `'127.0.0.1'`

This defines the Microsoft SQL Server to establish a connection. This value can be a hostname or IP address.

PORT

Default: `''` (Empty string)

This defines the network port to use when connecting to the server. If not defined, the standard Microsoft SQL Server port will be used.

NAME

This is the name of the SQL server database.

USER

Default: '' (Empty string)

This defines the name of the user to use when authenticating to the server. When empty, a trusted connection (SSPI) will be used.

PASSWORD

Default: '' (Empty string)

When a USER is defined, this field should be the plain text password to use when authenticating.

Note: Any user or service that can read the configuration file can will be able to see the plain-text password. Trusted connections are recommended.

TEST_CREATE

Default: True

This setting is specific to the *django-mssql* backend and controls whether or not the test database will be created and destroyed during the test creation. This is useful when integrating to a legacy database with a complex schema that is created by another application or cannot be easily created by Django's syncdb.

```
DATABASES = {
    'default': {
        'NAME': 'test_legacy_database',
        'HOST': r'servername\ss2008',
        'TEST_NAME': 'test_legacy_database',
        'TEST_CREATE': False,
    }
}
```

Note: This is not intended to allow you to run tests against a QA, staging, or production database.

1.2.2 OPTIONS

Django-mssql provides a few extra OPTIONS that are specific to this backend. Please note that while the main database settings are UPPERCASE keys, the OPTIONS dictionary keys are expected to be lowercase (due to legacy reasons).

use_mars

Default: True

Set to `False` to disable *Multiple Active Recordsets*. It is not recommended to disable MARS. Without MARS enabled, you will probably end up seeing the error "Cannot create new connection because in manual or distributed transaction mode".

Note: This doesn't really work properly with the "SQLOLEDB" provider.

extra_params

Default: '' (Empty string)

This value will be appended to the generated connection string. Use this to provide any specific connection settings that are not controllable with the other settings.

provider

Default: 'SQLCLI10'

The SQL provider to use when connecting to the database. If this doesn't work, try 'SQLCLI11' or 'SQLOLEDB'.

Note: `use_mars = True` doesn't always work properly with 'SQLOLEDB' and can result in the error "Cannot create new connection because in manual or distributed transaction mode." if you try to filter a queryset with another queryset.

disable_avg_cast

Default: `False`

This backend will automatically `CAST` fields used by the `AVG` function as `FLOAT` to match the behavior of the core database backends. Set this to `True` if you need SQL server to retain the datatype of fields used with `AVG`.

New in version 1.1.

Note: SQL server maintains the datatype of the values used in `AVG`. The average of an `int` column will be an `int`. With this option set to `True`, `AVG([1, 2]) == 1`, not 1.5.

use_legacy_date_fields

Default: `True`

This setting alters which data types are used for the `DateField`, `DateTimeField`, and `TimeField` fields. When `True`, the fields will all use the `datetime` data type. When `False`, they will use `date`, `datetime`, and `time` data types.

New in version 1.4.

1.3 Usage

`Django-mssql` is a Django database backend and supports the interface for the paired Django version. It should behave the same as the core backends.

1.3.1 Executing Custom SQL

Please refer to the Django documentation for [Executing custom SQL](#) directly.

1.3.2 Stored Procedures

Django-mssql provides support for executing stored procedures, with and without parameters. The main function that should be used to execute a stored procedure is `callproc`. `callproc` will allow executing stored procedures with both input and output parameters, integer return values, and result sets.

```
def callproc(self, procname, parameters=None):
    """Call a stored database procedure with the given name.

    The sequence of parameters must contain one entry for each
    argument that the sproc expects. The result of the
    call is returned as modified copy of the input
    sequence. Input parameters are left untouched, output and
    input/output parameters replaced with possibly new values.

    The sproc may also provide a result set as output,
    which is available through the standard .fetch*() methods.

    Extension: A "return_value" property may be set on the
    cursor if the sproc defines an integer return value.
    """
```

Example:

This example assumes that there exists a stored procedure named `uspDoesSomething` that expects two parameters (int and varchar), and returns 1 when there is a result set.

```
from django.db import connection

cursor = connection.cursor()
try:
    cursor.callproc('[dbo].[uspDoesSomething]', [5, 'blah'])

    if cursor.return_value == 1:
        result_set = cursor.fetchall()
finally:
    cursor.close()
```

It is also possible to use the cursor's `execute` method to call a stored procedure, but `return_value` will not be set on the cursor and output parameters are not supported. This usage is intended for calling a stored procedure that returns a result set or nothing at all.

Example:

```
from django.db import connection

cursor = connection.cursor()
try:
    cursor.execute('EXEC [dbo].[uspFetchSomeData]')
    result_set = cursor.fetchall()
finally:
    cursor.close()
```

1.3.3 RawStoredProcedureManager

The `RawStoredProcedureManager` provides the `raw_callproc` method that will take the name of a stored procedure and use the result set that it returns to create instances of the model.

Example:

```
from sqlserver_ado.models import RawStoredProcedureManager

class MyModel(models.Model):
    ...

    objects = RawStoredProcedureManager()

    sproc_params = [1, 2, 3]
    MyModel.objects.raw_callproc('uspGetMyModels', sproc_params)
```

Note: The `db_column` name for the field must match the case of the database field as returned by the stored procedure, or the value will not be populated and will get fetched by the ORM when the field is later accessed.

New in version 1.2.

1.4 Management Commands

Adding `sql_app` to your `INSTALLED_APPS` adds the following custom management commands.

1.4.1 dbgui

This will launch SQL Management Studio, connected to your project's default database.

```
python manage.py dbgui
```

1.5 Datatypes

There are known issues related to Python/DB data types.

1.5.1 Dates and Times

When using *Django-mssql* with SQL Server 2005, all of the date related fields only support the *datetime* data type. Support for these legacy data types can be enabled using the `use_legacy_date_fields` option, or using the fields `LegacyDateField`, `LegacyDateTimeField`, and `LegacyTimeField` in `sqlserver_ado.fields`.

To allow migrating specific apps or only some of your models to the new date times, the model fields `DateField`, `DateTimeField`, and `TimeField` in `sqlserver_ado.fields` use the new data types regardless of the `use_legacy_date_fields` option.

```
from django.db import models
from sqlserver_ado.fields import DateField, DateTimeField, TimeField

class MyModel(models.Model):
    # when use_legacy_date_fields is False, models.*Field will behave like these
    a_real_date = DateField() # date data type
    a_datetime2 = DateTimeField() # datetime2 data type
    a_real_time = TimeField() # time data type

    # when use_legacy_date_fields is True, models.*Field will behave like these
```

```
a_date = LegacyDateField() # datetime data type
a_datetime = LegacyDateTime() # datetime data type
a_time = LegacyTimeField() # datetime data type
```

datetime limitations

With SQL Server 2005, only the `datetime` data type is usable with Django. This data type does not store enough precision to provide the full range of Python datetime dates and will round to increments of .000, .003, or .007 seconds. The earliest supported datetime date value is January 1, 1753.

SQL Server 2008 introduces a `datetime2` type, with support for fractional seconds and the full range of Python datetime dates. To use this time, either set the `:settings:'use_legacy_date_fields'` option to `False` or use the `sqlserver_ado.fields.DateTimeField` with your models.

1.5.2 bigint

Prior to Django 1.3, `bigint` was not provided. This backend provided model fields to allow using the `bigint` datatype.

class `sqlserver_ado.fields.BigAutoField`

This is a `django.db.models.AutoField` for the `bigint` datatype.

class `sqlserver_ado.fields.BigIntegerField`

This was previously an `django.db.models.IntegerField` that specified the `bigint` datatype. As of Django 1.3, `django.db.models.BigIntegerField` is provided and should be used instead.

class `sqlserver_ado.fields.BigForeignKey`

This is a `django.db.models.ForeignKey` that should be used to reference either a `BigAutoField` or a `BigIntegerField`.

Note: If your (legacy) database using bigints for primary keys, then you'll need to replace any introspected `ForeignKey` fields with `BigForeignKey` for things to work as expected.

1.5.3 money

The `money` and `smallmoney` data types will be introspected as `DecimalField` with the appropriate values for `max_digits` and `decimal_places`. This does not mean that they are expected to work without issue.

1.5.4 Unsupported Types

These types may behave oddly in a Django application, as they have limits smaller than what Django expects from similar types:

- `smalldatetime`
- `tinyint`
- `real`

1.6 Testing

All tests are contained in the `tests` folder.

1.6.1 Django Support

Database Introspection

Located in folder `tests/test_inspectdb`. These tests verify the database introspection.

To run:

- set up the `SQLINSTANCE`
- `python manage.py inspectdb`

Main Tests

The tests for basic functionality and regressions are located in the `tests/test_main/` folder.

To run:

- set up the `SQLINSTANCE`
- `python manage.py test`

Note: The test app `apitest` contains a test suite for checking DB-API 2 conformance.

To run only the DB-API 2 tests:

```
python manage.py test apitest
```

Running Django Test Suite

To run the Django test suite, you will need to create a settings file that lists `'sqlserver_ado'` as the `ENGINE`.

Example settings:

```
DATABASES = {
    'default': {
        'ENGINE': 'sqlserver_ado',
        'NAME': 'django_framework',
        'HOST': r'localhost\ss2008',
        'USER': '',
        'PASSWORD': '',
    },
    'other': {
        'ENGINE': 'sqlserver_ado',
        'NAME': 'django_framework_other',
        'HOST': r'localhost\ss2008',
        'USER': '',
        'PASSWORD': '',
    }
}

SECRET_KEY = "django_tests_secret_key"
```

1.7 Changelog

1.7.1 v1.4

- Support for Django v1.3 has been removed.
- Corrected DB-API 2 testing documentation.
- Fixed issue with slicing logic that could prevent the compiler from finding and mapping column aliases properly.
- Improved the “return ID from insert” logic so it can properly extract the column data type from user defined fields with custom data type strings.
- Fixed case for identifiers in introspection. Thanks Mikhail Denisenko.
- Added option `use_legacy_date_fields` (defaults to `True`) to allow changing the `DatabaseCreation.data_types` to not use the Microsoft preferred date data types that were added with SQL Server 2008. [django-mssql issue #31](#)
- Improved accuracy of field type guessing with `inspectdb`. See *Introspecting custom fields*
- Fixed issue with identity insert using a cursor to the wrong database in a multi-database environment. Thanks Mikhail Denisenko
- Fixed constraint checking. [django-mssql issue #35](#) Thanks Mikhail Denisenko
- Enabled `can_introspect_autofield` database feature. [Django ticket #21097](#)
- Any date related field should now return from the database as the appropriate Python type, instead of always being a `datetime`.
- Backend now supports doing date lookups using a string. E.g. `Article.objects.filter(pub_date__startswith='2005')`
- `check_constraints` will now check all disabled and enabled constraints. This change was made to match the behavior tested by `backends.FkConstraintsTests.test_check_constraints`.
- Improved `date_interval_sql` support for the various date/time related datatypes. The `timedelta` value will control whether the database will `DATEADD` using `DAY` or `SECOND`. Trying to add seconds to a date, or days to a `time` will generate database exceptions.
- Fixed issue with provider detection that prevented `DataTypeCompatibility=80` from being automatically added to the connection string for the native client providers.
- Fixed SQL generation error that occurred when ordering the query based upon a column that is not being returned.
- Added savepoint support. MS SQL Server doesn't support savepoint commits and will no-op it. Other databases, e.g. postgresql, mostly use it as a way of freeing server resources in the middle of a transaction. Thanks Martijn Pieters.
- Minor cleanup of limit/offset SQL mangling to allow custom aggregates that require multiple column replacements. [django-mssql issue #40](#) Thanks Martijn Pieters for initial patch and tests.
- Savepoints cannot be used with MARS connections. [django-mssql issue #41](#)

1.7.2 v1.3.1

- Ensure Django knows to re-enable constraints. [django-mssql issue #29](#)

1.7.3 v1.3

- Backend now supports returning the ID from an insert without needing an additional query. This is disabled for SQL Server 2000 (assuming that version still works with this backend). [django-mssql issue #17](#)
 - This will work even if the table has a trigger. [django-mssql issue #20](#)
- Subqueries will have their ordering removed because SQL Server only supports it when using TOP or FOR XML. This relies upon the `with_col_aliases` argument to `SQLCompiler.as_sql` only being True when the query is a subquery, which is currently the case for all usages in Django 1.5 master. [django-mssql issue #18](#)
- UPDATE statements will now return the number of rows affected, instead of -1. [django-mssql issue #19](#)
- Apply fix for [Django ticket #12192](#). If QuerySet slicing would result in `LIMIT 0`, then it shouldn't reach the database because there will be no response.
- Implemented DatabaseOperation `cache_key_culling_sql`. [Django ticket #18330](#)
- Fixed `cast_avg_to_float` so that it only controls the cast for AVG and not mapping other aggregates.
- Improved IP address detection of HOST setting. [django-mssql issue #21](#)
- Set database feature `ignores_nulls_in_unique_constraints` = False because MSSQL cannot ignore NULLs in unique constraints.
- [django-mssql issue #26](#) Documented clustered index issue with Azure SQL. See *Azure requires clustered indices*.

1.7.4 v1.2

- Ensure master connection connects to the correct database name when `TEST_NAME` is not defined.
- `Connection.close()` will now reset `adoConn` to make sure it's gone before the `CoUninitialize`.
- Changed provider default from 'SQLOLEDB' to 'SQLNCLI10' with MARS enabled.
- Added `RawStoredProcedureManager`, which provides `raw_callproc` that works the same as `raw`, except expects the name of a stored procedure that returns a result set that matches the model.
- Documented known issue with database introspection with `DEBUG = True` and column names containing '%'. See *Introspecting tables with '%' columns*.
- Fixed error with `iendswith` string format operator.

1.7.5 v1.1

- Updated `SQLInsertCompiler` to work with Django 1.4
- Added support for `disable_constraint_checking`, which is required for `loaddata` to work properly.
- Implemented `DatabaseOperations.date_interval_sql` to allow using expressions like `end__lte=F('start')+delta`.
- Fixed date part extraction for `week_day`.
- `DatabaseWrapper` reports vendor as 'microsoft'.
- AVG function now matches core backend behaviors and will auto-cast to `float`, instead of maintaining datatype. Set database `OPTIONS` setting `disable_avg_cast` to turn off the auto-cast behavior.

- StdDev and Variance aggregate functions are now supported and will map to the proper MSSQL named functions. Includes work around for [Django ticket #18334](#).
- Monkey patched `django.db.backends.util.CursorWrapper` to allow using cursors as ContextManagers in Python 2.7. [Django ticket #17671](#).

1.8 Known Issues

1.8.1 Introspecting tables with '%' columns

Attempting to run `manage.py inspectdb` with `DEBUG = True` will raise `TypeError: not enough arguments for format string`. This is due to `CursorDebugWrapper` and its use of `%` format strings. If you encounter this problem, you can either rename the database column so it does not include a `'%'` (percent) character, or change `DEBUG = False` in your settings when you run `manage.py inspectdb`.

1.8.2 Introspecting custom fields

Some datatypes will be mapped to a custom model field provided by *Django-mssql*. If any of these fields are used, it will be necessary to add `import sqlserver_ado.fields` to the top of the `models.py` file. If using a version of Django prior to 1.7, it will be necessary to also remove the "models." prefix from any of these custom fields. [Django ticket #21090](#)

1.8.3 Azure requires clustered indices

From <http://msdn.microsoft.com/en-us/library/windowsazure/ee336245.aspx#cir>

Windows Azure SQL Database does not support tables without clustered indexes. A table must have a clustered index. If a table is created without a clustered constraint, a clustered index must be created before an insert operation is allowed on the table.

The workaround is to dump the create SQL, add a clustered index, manually apply the SQL to the database.