django-dynamic-db-router Documentation

Release 0.1.1

Erik Swanson

August 24, 2016

Contents

Working with multiple databases within django is supported, but the syntax requires peppering `.using('my_database')` throughout all queries that need to be routed to different databases. This is especially painful when trying to use libraries that were written without multiple database support in mind. With this library, running complex queries across different databases is as simple as:

```
from dynamic_db_router import in_database

with in_database('non-default-db'):
    result = run_complex_query()
```

To set up you django project to be able to use this router, simply `pip install django-dynamic-db-router` and add `DATABASE_ROUTERS=['dynamic_db_router.DynamicDbRouter']` to your Django settings.

Django Dynamic DB Router includes a number of additional features, such as:

- Using `in_database` as a function decorator.
- Read and write protection controls.
- Load database configurations dynamically for the lifetime of the context manager.

For more information, and complete API documentation, see the quickstart guide or code documentation, linked below.

CHAPTER 1

Table of Contents

1.1 Installation

To install the latest release, type:

```
pip install django-dynamic-db-router
```

To install the latest code directly from source, type:

```
pip install git+git://github.com/ambitioninc/django-dynamic-db-router.git
```

1.2 Quickstart

First, make sure you have setup Django Dynamic DB Router by installing the package using `pip install django-dynamic-db-router` and make sure you have added `DATABASE_ROUTERS=['dynamic_db_router.DynamicDbRouter']` to your Django settings.

Once you have installed and set up `django-dynamic-db-router` all your routing of queries between different databases can be done with `dynamic_db_router.in_databases`, which can be used as a context manager or function decorator. There are two main use cases for `in_database`. The first is to connect to databases that are already configured in your Django settings `DATABASES` parameter. The second is to establish a connection to a database that has not been configured, and route queries there. Both of these use cases will be described below.

1.2.1 Routing to Configured Databases

In the case that you have multiple databases set up in your django project, you will should have a `settings.py` file containing something like:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'my_local_database',
        'USER': 'postgres',
        'PASSWORD': 'my-pass',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    },
    'external': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
```

```
        'NAME': 'my_external_database',
        'USER': 'postgres',
        'PASSWORD': 'my-pass',
        'HOST': 'example.com',
        'PORT': '5432',
    },
    # ...
}
DATABASE_ROUTERS = ['dynamic_db_router.DynamicDbRouter']
```

possibly with additional databases configured. With this configuration, all queries will, by default, go to the `'default'` database. To route queries to the `'external'` database, you could use Django's built in `.using('external')` method on querysets. However, this becomes cumbersome when doing a large number of queries, or using libraries that don't take multiple databases into account.

It is much easier to do these queries in a context manager, where you can write the queries such that they will automatically be routed to the correct database. This is what the `in_database` context manager provides:

```
from dynamic_db_router import in_database

from my_app.models import MyModel
from some_app.utils import complex_query_function

with in_database('external'):
    input = MyModel.objects.filter(field_a="okay")
    output = complex_query_function(input)
```

In the example above, the `MyModel.objects.filter` query will be performed against the `'external'` database, but so will any queries made by `complex_query_function`, which otherwise may be impossible to guarantee with `.using`. These queries will only be routed to the `'external'` database for the `with` block, and will automatically route to the `'default'` database outside of the `with` block again.

Often times when working with multiple databases, you want to control whether you can read and write to a given database. For this reason, the `in_database` context manager takes two optional parameters, `read` and `write`. These control whether reads and writes go to the provided database, respectively, and default to `read=True` and `write=False`.

Allowing writing to the `'external'` database then, would look like:

```
with in_database('external', write=True):
    MyModel.objects.create(
        field_a='bad',
        field_b=17774,
    )
```

It should be noted that with `write=False` attempts to write inside the context manager will *not* fail, but will be routed to the `'default'` database. This also holds for `read=False`. It is up to the user to understand what sorts queries are being run within the context manager. In the example above, if `write` were set to `False`, the `MyModel` object would still be created, but in the `'default'` database.

Additionally, if you want to define a function which always pulls its results from a certain database, `in_database` can be used as a function decorator:

```
from dynamic_db_router import in_database

@in_database('external')
def get_external_models_count(models):
    counts = {}
    for model in models:
```

```
        counts[model] = model.objects.count()
    return counts
```

Whenever this function is run, it will route the queries in the function to the `'external'` database. The decorator version of `in_database` takes all the same arguments as the context-manager version, so it is possible to control read/write permissions in the same way.

1.2.2 Dynamic Database Configuration and Routing

In addition to accessing databases that are already configured in `django.conf.settings.DATABASES`, `django-dynamic-db-router` can also be used to dynamically set up a database configuration, route queries to it and tear down the configuration as the context manager or decorated function exits.

In order for this to function properly, the database you are trying to connect to dynamically must already be set up with tables corresponding to whatever models you want to use to query them. Given such a database, dynamically connecting to it and querying it is as simple as passing `in_database` a dictionary with connection information, rather than a string:

```
from dynamic_db_router import in_database
from my_app.models import MyModel

external_db = {
    'ENGINE': 'django.db.backends.postgresql_psycopg2',
    'NAME': 'my_external_database',
    'USER': 'postgres',
    'PASSWORD': 'my-pass',
    'HOST': 'example.com',
    'PORT': '5432',
}

with in_database(external_db):
    target = MyModel.objects.get(field_b=17774)
```

In the example above, even though there is no entry for the database configuration in `settings.DATABASES`, `in_databases` is able to access the database, run the query, and clean up after itself.

When using a configuration as an argument, `in_databases` still supports read and write controls as described above, and supports use as a function decorator.

1.3 Code documentation

1.3.1 dynamic_db_router

class `dynamic_db_router.`**`in_database`**(*database*, *read=True*, *write=False*)

A decorator and context manager to do queries on a given database.

Parameters

- **`database`** (*`str or dict`*) – The database to run queries on. A string will route through the matching database in `django.conf.settings.DATABASES`. A dictionary will set up a connection with the given configuration and route queries to it.
- **`read`** (*`bool, optional`*) – Controls whether database reads will route through the provided database. If `False`, reads will route through the `'default'` database. Defaults to `True`.

- **write** (*bool, optional*) – Controls whether database writes will route to the provided database. If `False`, writes will route to the `'default'` database. Defaults to `False`.

When used as eithe a decorator or a context manager, *in_database* requires a single argument, which is the name of the database to route queries to, or a configuration dictionary for a database to route to.

Usage as a context manager:

```
from my_django_app.utils import tricky_query

with in_database('Database_A'):
    results = tricky_query()
```

Usage as a decorator:

```
from my_django_app.models import Account

@in_database('Database_B')
def lowest_id_account():
    Account.objects.order_by('-id')[0]
```

Used with a configuration dictionary:

```
db_config = {'ENGINE': 'django.db.backends.sqlite3',
             'NAME': 'path/to/mydatabase.db'}
with in_database(db_config):
    # Run queries
```

1.4 Contributing

Contributions and issues are most welcome! All issues and pull requests are handled through github on the ambitioninc repository. Also, please check for any existing issues before filing a new one. If you have a great idea but it involves big changes, please file a ticket before making a pull request! We want to make sure you don't spend your time coding something that might not fit the scope of the project.

1.4.1 Running the tests

To get the source source code and run the unit tests, run:

```
$ git clone git://github.com/ambitioninc/django-dynamic-db-router.git
$ cd django-dynamic-db-router
$ virtualenv env
$ . env/bin/activate
$ pip install nose
$ python setup.py install
$ python setup.py nosetests
```

While 100% code coverage does not make a library bug-free, it significantly reduces the number of easily caught bugs! Please make sure coverage is at 100% before submitting a pull request!

1.4.2 Code Quality

For code quality, please run flake8:

```
$ pip install flake8
$ flake8 .
```

1.4.3 Code Styling

Please arrange imports with the following style

```
# Standard library imports
import os

# Third party package imports
from mock import patch

# Local package imports
from dynamic_db_router.version import __version__
```

Please follow Google's python style guide wherever possible.

1.4.4 Building the docs

When in the project directory:

```
$ pip install -r requirements/docs.txt
$ python setup.py build_sphinx
$ open docs/_build/html/index.html
```

1.4.5 Release Checklist

Before a new release, please go through the following checklist:

- Bump version in dynamic_db_router/version.py
- Add a release note in docs/release_notes.rst
- Git tag the version
- Upload to pypi:

```
    pip install wheel
    python setup.py sdist bdist_wheel upload
```

1.4.6 Vulnerability Reporting

For any security issues, please do NOT file an issue or pull request on github! Please contact security@ambition.com with the GPG key provided on Ambition's website.

1.5 Release Notes

1.5.1 v0.1

- This is the initial release of django-dynamic-db-router. It includes a core featureset of:

- A `in_database` context-manager and function-decorator to route all queries to a database dynamically.
- Read and write protection controls on `in_database`.
- Dynamically load database configurations for the lifetime of the context-manager.
- 100% branch test coverage.
- Documentation at the package and API level.

I