Django Template Test Database Documentation

Release 1.0.1

Django Template Test Database developers and contributors

December 05, 2016

Contents

Contents

CHAPTER 1

Overview

Django template test database is a testing tool for django that provides an alternative to fixtures when tools like FactoryBoy aren't suitable. The use case is simple: for integration tests that require the test database to be populated with a specific (large) set of test data before they will even run. Loading this test data using fixtures would be very slow. This problem is solved by loading the test data during database creation at the database level and allows us to avoid all of the overhead by loading data through django.

CHAPTER 2

Best Practice

If your test data is large enough to make use of django-ttdb then I recommend using a minimum of two databases for testing. First use a sqlite database for all simple unit tests that do not require special database features (postgres). Using a sqlite db for testing in django runs in memory and is fast. Second define a postgres test database for integration tests which need a specific set of data to test against.

CHAPTER 3

Dependencies

- Django 1.8 and above
- mock
- postgresql
- psycopg2

CHAPTER 4

Installation

To use django-ttdb you must first install it using your preferred method:

```
$ pip install django-ttdb
```

To use django-ttdb you must use the test runner included with ttdb. In your settings file you can tell it to use the django-ttdb runner.:

```
TEST_RUNNER = 'ttdb.runner.TemplateDatabaseRunner'
```

CHAPTER 5

Usage

Then you have to define a which postgres databases you want to enable ttdb for. You do this by defining the TTDB setting as shown below:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'django_ttdb',
        'USER': 'postgres',
        'HOST': '127.0.0.1',
    }
}

TTDB = (
    'default',
)
```

Now we can use the template test database in our tests. There are a few ways to do this but the easiest way is using a decorator:

```
from ttdb import use_template_database
from django.test import TestCase

@use_template_database('integration')
class TestClassDecorator(TestCase):
    def test_class_decorator(self):
        """All tests inside class will use postgres template database."""
        pass

class TestDecorator(TestCase):
    @use_template_database('integration')
    def test_decorator(self):
        """Is running tests using the postgres template database."""
        pass
```

We can also use the *TemplateDBTestCase* class:

```
from ttdb import TemplateDBTestCase

class TestClass(TemplateDBTestCase):
    template_database = 'integration'

    def test_class(self):
        """Define the template_database and use inheritance rather than decorator."""
```

```
        pass
```

Note: Because the TestCase class patches the transaction management code when the test has completed the database is rolled back to it's original state. This means that we don't need to do anything special to preserve the test data between tests.

It also supports the TransactionTestCase. However because of the way that the TransactionTestCase works we have to customize the test case to not flush the database after every test to make sure that the data remains in the database. Instead of flushing the database after each test django-ttdb takes a different approach. It drops the database and creates it after each test:

```
from ttdb import TemplateDBTransactionTestCase
from ttdb import use_template_database
from django.test import TransactionTestCase

@use_template_database('integration')
class TestTransactions(TransactionTestCase):
    def test_transaction(self):
        """After running the template test db will be droped and created."""
        pass

class TestTransactionTwo(TemplateDBTransactionTestCase):
    template_database = 'integration'

    def test_transaction(self):
        pass
```

This behaviour will sometimes be undesirable, for example if the test case will clean up after it's self. In these cases we can tell django-ttdb to not drop and create the database after each test:

```
from ttdb import TemplateDBTransactionTestCase
from ttdb import use_template_database
from django.test import TransactionTestCase

@use_template_database('integration', reload_after_test=False)
class TestTransactions(TransactionTestCase):
    def test_transaction(self):
        """Database will remain upon test completion."""
        pass

class TestTransactionTwo(TemplateDBTransactionTestCase):
    template_database = 'integration'
    reload_after_test = False

    def test_transaction(self):
        pass
```

We also support the LiveServerTestCase. This is slightly different again. Because the LiveServerTestCase starts a django server running in a seperate thread we need to patch the database before the thread starts. To do this the database is patched in the setUpClass method and remains patched until all of the tests in the LiveServerTestCase have run. That means that unlike the TestCase and TransactionTestCase the template db will not be droped and created after each test, rather at the creation and destruction of the test class:

```
from django.test import LiveServerTestCase
from ttdb import use_template_database
from ttdb import TemplateDBLiveServerTestCase
```

```
@use_template_database('integration')
class TestLiveServer(LiveServerTestCase):
    def test_one(self):
        """Database not destroyed after."""
        pass

    def test_two(self):
        """Database destroyed after last test in class run."""
        pass

class TestLiveServer(TemplateDBLiveServerTestCase):
    template_database = 'integration'

    def test_one(self):
        """Database not destroyed after."""
        pass

    def test_two(self):
        """Database destroyed after last test in class run."""
        pass
```

Finally, the `use_template_database` decorator also works with the with statement:

```
from django.test import TestCase

class Test(TestCase):
    def test_with(self):
        """Test as with statement."""
        with use_template_database('integration', reload_after_test=False):
            # Test against integration database
            pass
        # Test against default sqlite database
```

CHAPTER 6

Integration with other test runners

django-ttdb should play nice with other test runners. One way to integrate it is by creating a test runner that subclasses other test runners:

```
from ttdb.runner import TemplateDatabaseRunner
from other_runner import OtherRunner

class MyTestRunner(OtherRunner, TemplateDatabaseRunner):
    option_list = OtherRunner.option_list + DiscoverRunner.option_list
```

Then in your settings file:

```
TEST_RUNNER = 'path.to.MyTestRunner'
```